



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Scala for Machine Learning

Leverage Scala and Machine Learning to construct and study systems that can learn from data

Patrick R. Nicolas

[PACKT] open source*
PUBLISHING

community experience distilled

Scala for Machine Learning

Leverage Scala and Machine Learning to construct and study systems that can learn from data

Patrick R. Nicolas



BIRMINGHAM - MUMBAI

Scala for Machine Learning

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2015

Production reference: 2181215

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

ISBN 978-1-78355-874-2

www.packtpub.com

Credits

Author

Patrick R. Nicolas

Project Coordinator

Danuta Jones

Reviewers

Subhajit Datta
Rui Gonçalves
Patricia Hoffman, PhD
Md Zahidul Islam

Proofreaders

Simran Bhogal
Maria Gould
Paul Hindle
Elinor Perry-Smith
Chris Smith

Commissioning Editor

Owen Roberts

Indexers

Hemangini Bari
Mariammal Chettiar

Acquisition Editor

Owen Roberts

Graphics

Sheetal Aute
Valentina D'silva
Disha Haria
Abhinash Sahu

Content Development Editor

Mohammed Fahad

Production Coordinators

Arvindkumar Gupta
Shantanu N. Zagade

Technical Editors

Madhuri Das
Taabish Khan

Cover Work

Arvindkumar Gupta

Copy Editors

Janbal Dharmaraj
Vikrant Phadkay
Rashmi Sawant

About the Author

Patrick R. Nicolas is a lead R&D engineer at Dell in Santa Clara, California. He has 25 years of experience in software engineering and building large-scale applications in C++, Java, and Scala, and has held several managerial positions. His interests include real-time analytics, modeling, and optimization.

Special thanks to the Packt Publishing team: Mohammed Fahad for his patience and encouragement, Owen Roberts for the opportunity, and the reviewers for their guidance and dedication.

About the Reviewers

Subhajit Datta is a passionate software developer.

He did his Bachelor of Engineering in Information Technology (BE in IT) from Indian Institute of Engineering Science and Technology, Shibpur (IEST, Shibpur), formerly known as Bengal Engineering and Science University, Shibpur.

He completed his Master of Technology in Computer Science and Engineering (MTech CSE) from Indian Institute of Technology Bombay (IIT Bombay); his thesis focused on topics in natural language processing.

He has experience working in the investment banking domain and web application domain, and is a polyglot having worked on Java, Scala, Python, Unix shell scripting, VBScript, JavaScript, C#.Net, and PHP. He is interested in learning and applying new and different technologies.

He believes that choosing the right programming language, tool, and framework for the problem at hand is more important than trying to fit all problems in one technology.

He also has experience working in the Waterfall and Agile processes. He is excited about the Agile software development processes.

Rui Gonçalves is an all-round, hardworking, and dedicated software engineer. He is an enthusiast of software architecture, programming paradigms, algorithms, and data structures with the ambition of developing products and services that have a great impact on society.

He currently works at ShiftForward, where he is a software engineer in the online advertising field. He is focused on designing and implementing highly efficient, concurrent, and scalable systems as well as machine learning solutions. In order to achieve this, he uses Scala as the main development language of these systems on a day-to-day basis.

Patricia Hoffman, PhD, is a consultant at iCube Consulting Service Inc., with over 25 years of experience in modeling and simulation, of which the last six years concentrated on machine learning and data mining technologies. Her software development experience ranges from modeling stochastic partial differential equations to image processing. She is currently an adjunct faculty member at International Technical University, teaching machine learning courses. She also teaches machine learning and data mining at the University of California, Santa Cruz – Silicon Valley Campus. She was Chair of Association for Computing Machinery of the Data Mining Special Interest Group for the San Francisco Bay area for 5 years, organizing monthly lectures and five data mining conferences with over 350 participants.

Patricia has a long list of significant accomplishments. She developed the architecture and software development plan for a collaborative recommendation system while consulting as a data mining expert for Quantum Capital. While consulting for Revolution Analytics, she developed training materials for interfacing the R statistical language with IBM's Netezza data warehouse appliance.

She has also set up the systems used for communication and software development along with technical coordination for GTECH, a medical device start-up.

She has also technically directed, produced, and managed operations concepts and architecture analysis for hardware, software, and firmware. She has performed risk assessments and has written qualification letters, proposals, system specs, and interface control documents. Also, she has coordinated with subcontractors, associate contractors, and various Lockheed departments to produce analysis, documents, technology demonstrations, and integrated systems. She was the Chief Systems Engineer for a \$12 million image processing workstation development, and had scored 100 percent from the customer.

The various contributions of Patricia to the publications field are as follows:

- A unified view on the rotational symmetry of equilibria of nematic polymers, dipolar nematic polymers, and polymers in higher dimensional space, *Communications in Mathematical Sciences*, Volume 6, 949-974
- She worked as a technical editor on the book *Machine Learning in Action*, Peter Harrington, Manning Publications Co.
- A Distributed Architecture for the C3 I (Command, Control, Communications, and Intelligence) Collection Management Expert System, with Allen Rude, AIC Lockheed
- A book review of computer-supported cooperative work, *ACM/SIGCHI Bulletin*, Volume 21, Issue 2, pages 125-128, ISSN:0736-6906, 1989

Md Zahidul Islam is a software developer working for HSI Health and lives in Concord, California, with his wife.

He has a passion for functional programming, machine learning, and working with data. He is currently working with Scala, Apache Spark, MLlib, Ruby on Rails, ElasticSearch, MongoDB, and Backbone.js. Earlier in his career, he worked with C#, ASP.NET, and everything around the .NET ecosystem.

I would like to thank my wife, Sandra, who lovingly supports me in everything I do. I'd also like to thank Packt Publishing and its staff for the opportunity to contribute to this book.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

To Jennifer, for her kindness and support throughout this long journey.

Table of Contents

Preface	xiii
Chapter 1: Getting Started	1
Mathematical notation for the curious	2
Why machine learning?	2
Classification	3
Prediction	3
Optimization	3
Regression	3
Why Scala?	3
Abstraction	4
Higher-kind projection	4
Covariant functors for vectors	5
Contravariant functors for co-vectors	6
Monads	7
Scalability	7
Configurability	8
Maintainability	9
Computation on demand	9
Model categorization	9
Taxonomy of machine learning algorithms	10
Unsupervised learning	10
Clustering	10
Dimension reduction	11
Supervised learning	11
Generative models	11
Discriminative models	12
Semi-supervised learning	14
Reinforcement learning	14
Don't reinvent the wheel!	15
Tools and frameworks	15

Table of Contents

Java	15
Scala	15
Apache Commons Math	16
Description	16
Licensing	16
Installation	16
JFreeChart	17
Description	17
Licensing	17
Installation	17
Other libraries and frameworks	18
Source code	18
Context versus view bounds	19
Presentation	19
Primitives and implicits	20
Primitive types	20
Type conversions	21
Immutability	21
Performance of Scala iterators	22
Let's kick the tires	22
An overview of computational workflows	22
Writing a simple workflow	23
Step 1 – scoping the problem	24
Step 2 – loading data	26
Step 3 – preprocessing the data	27
Step 4 – discovering patterns	30
Step 5 – implementing the classifier	32
Step 6 – evaluating the model	40
Summary	43
Chapter 2: Hello World!	45
Modeling	45
A model by any other name	45
Model versus design	47
Selecting features	47
Extracting features	48
Defining a methodology	48
Monadic data transformation	49
Error handling	50
Explicit models	51
Implicit models	53
A workflow computational model	55
Supporting mathematical abstractions	56
Step 1 – variable declaration	56

Table of Contents

Step 2 – model definition	56
Step 3 – instantiation	57
Composing mixins to build a workflow	58
Understanding the problem	58
Defining modules	60
Instantiating the workflow	61
Modularization	63
Profiling data	66
Immutable statistics	66
Z-Score and Gauss	67
Assessing a model	68
Validation	68
Key quality metrics	69
F-score for binomial classification	70
F-score for multinomial classification	72
Cross-validation	75
One-fold cross validation	75
K-fold cross validation	77
Bias-variance decomposition	77
Overfitting	81
Summary	82
Chapter 3: Data Preprocessing	83
Time series in Scala	83
Types and operations	84
The magnet pattern	87
The transpose operator	87
The differential operator	88
Lazy views	89
Moving averages	89
The simple moving average	90
The weighted moving average	92
The exponential moving average	93
Fourier analysis	97
Discrete Fourier transform	98
DFT-based filtering	105
Detection of market cycles	108
The discrete Kalman filter	111
The state space estimation	113
The transition equation	113
The measurement equation	114
The recursive algorithm	114
Prediction	117
Correction	118

Table of Contents

Kalman smoothing	119
Fixed lag smoothing	121
Experimentation	121
Benefits and drawbacks	125
Alternative preprocessing techniques	126
Summary	126
Chapter 4: Unsupervised Learning	127
Clustering	128
K-means clustering	129
Measuring similarity	129
Defining the algorithm	131
Step 1 – cluster configuration	132
Step 2 – cluster assignment	135
Step 3 – reconstruction/error minimization	137
Step 4 – classification	140
The curse of dimensionality	140
Setting up the evaluation	142
Evaluating the results	144
Tuning the number of clusters	145
Validation	148
The expectation-maximization algorithm	149
Gaussian mixture models	149
Overview of EM	150
Implementation	151
Classification	154
Testing	154
The online EM algorithm	157
Dimension reduction	157
Principal components analysis	158
Algorithm	159
Implementation	160
Test case	162
Evaluation	163
Non-linear models	165
Kernel PCA	165
Manifolds	166
Performance considerations	166
K-means	166
EM	167
PCA	167
Summary	168
Chapter 5: Naïve Bayes Classifiers	169
Probabilistic graphical models	169
Naïve Bayes classifiers	171
Introducing the multinomial Naïve Bayes	172

Table of Contents

Formalism	174
The frequentist perspective	175
The predictive model	176
The zero-frequency problem	177
Implementation	178
Design	178
Training	179
Classification	185
F ₁ validation	186
Feature extraction	187
Testing	190
The Multivariate Bernoulli classification	191
Model	192
Implementation	192
Naïve Bayes and text mining	193
Basics of information retrieval	195
Implementation	196
Analyzing documents	197
Extracting the frequency of relative terms	199
Generating the features	200
Testing	201
Retrieving the textual information	201
Evaluating the text mining classifier	203
Pros and cons	205
Summary	206
Chapter 6: Regression and Regularization	207
Linear regression	207
One-variate linear regression	208
Implementation	208
Test case	210
Ordinary least squares regression	211
Design	212
Implementation	215
Test case 1 – trending	216
Test case 2 – feature selection	218
Regularization	225
L _n roughness penalty	226
Ridge regression	227
Design	228
Implementation	229
Test case	231
Numerical optimization	234
Logistic regression	236
Logistic function	236
Binomial classification	237

Table of Contents

Design	240
The training workflow	240
Step 1 – configuring the optimizer	242
Step 2 – computing the Jacobian matrix	243
Step 3 – managing the convergence of the optimizer	244
Step 4 – defining the least squares problem	244
Step 5 – minimizing the sum of square errors	245
Test	245
Classification	247
Summary	250
Chapter 7: Sequential Data Models	251
Markov decision processes	251
The Markov property	252
The first order discrete Markov chain	252
The hidden Markov model	253
Notations	255
The lambda model	256
Design	258
Evaluation – CF-1	260
Alpha – the forward pass	260
Beta – the backward pass	264
Training – CF-2	266
The Baum-Welch estimator (EM)	266
Decoding – CF-3	269
The Viterbi algorithm	270
Putting it all together	272
Test case 1 – training	275
Test case 2 – evaluation	277
HMM as a filtering technique	278
Conditional random fields	279
Introduction to CRF	279
Linear chain CRF	281
Regularized CRFs and text analytics	283
The feature functions model	284
Design	286
Implementation	287
Configuring the CRF classifier	287
Training the CRF model	290
Applying the CRF model	293
Tests	293
The training convergence profile	294
Impact of the size of the training set	295
Impact of the L_2 regularization factor	296

Comparing CRF and HMM	296
Performance consideration	297
Summary	298
Chapter 8: Kernel Models and Support Vector Machines	299
Kernel functions	300
An overview	300
Common discriminative kernels	302
Kernel monadic composition	304
Support vector machines	307
The linear SVM	307
The separable case – the hard margin	307
The nonseparable case – the soft margin	308
The nonlinear SVM	311
Max-margin classification	311
The kernel trick	312
Support vector classifiers – SVC	313
The binary SVC	313
LIBSVM	313
Design	314
Configuration parameters	315
Interface to LIBSVM	318
Training	319
Classification	323
C-penalty and margin	323
Kernel evaluation	326
Applications in risk analysis	331
Anomaly detection with one-class SVC	335
Support vector regression	337
An overview	337
SVR versus linear regression	339
Performance considerations	342
Summary	342
Chapter 9: Artificial Neural Networks	343
Feed-forward neural networks	343
The biological background	344
Mathematical background	345
The multilayer perceptron	347
The activation function	348
The network topology	349
Design	350
Configuration	351
Network components	351
The network topology	352

Table of Contents

Input and hidden layers	353
The output layer	354
Synapses	355
Connections	355
The initialization weights	356
The model	357
Problem types (modes)	357
Online training versus batch training	358
The training epoch	359
Step 1 – input forward propagation	361
Step 2 – error backpropagation	366
Step 3 – exit condition	372
Putting it all together	373
Training and classification	374
Regularization	374
The model generation	375
The Fast Fisher-Yates shuffle	376
Prediction	377
Model fitness	377
Evaluation	378
The execution profile	378
Impact of the learning rate	379
The impact of the momentum factor	381
The impact of the number of hidden layers	382
Test case	383
Implementation	385
Evaluation of models	386
Impact of the hidden layers' architecture	388
Convolution neural networks	390
Local receptive fields	390
Sharing of weights	391
Convolution layers	392
Subsampling layers	392
Putting it all together	393
Benefits and limitations	393
Summary	395
Chapter 10: Genetic Algorithms	397
Evolution	397
The origin	398
NP problems	398
Evolutionary computing	399
Genetic algorithms and machine learning	400
Genetic algorithm components	400

Table of Contents

Encoding	401
Value encoding	401
Predicate encoding	402
Solution encoding	403
The encoding scheme	404
Genetic operators	405
Selection	407
Crossover	408
Mutation	409
The fitness score	410
Implementation	410
Software design	411
Key components	412
Population	412
Chromosomes	413
Genes	413
Selection	416
Controlling the population growth	417
The GA configuration	417
Crossover	417
Population	418
Chromosomes	420
Genes	421
Mutation	422
Population	422
Chromosomes	422
Genes	423
Reproduction	423
Solver	424
GA for trading strategies	427
Definition of trading strategies	428
Trading operators	428
The cost function	429
Trading signals	430
Trading strategies	430
Trading signal encoding	432
A test case	432
Creating trading strategies	434
Configuring the optimizer	436
Finding the best trading strategy	437
Tests	437
Advantages and risks of genetic algorithms	440
Summary	441

Table of Contents

Chapter 11: Reinforcement Learning	443
Reinforcement learning	443
The problem	444
A solution – Q-learning	444
Terminology	445
Concepts	446
Value of a policy	447
The Bellman optimality equations	448
Temporal difference for model-free learning	450
Action-value iterative update	451
Implementation	452
Software design	452
The states and actions	453
The search space	454
The policy and action-value	456
The Q-learning components	458
The Q-learning training	460
Tail recursion to the rescue	462
The validation	463
The prediction	464
Option trading using Q-learning	465
The OptionProperty class	467
The OptionModel class	468
Quantization	469
Putting it all together	471
Evaluation	474
Pros and cons of reinforcement learning	477
Learning classifier systems	477
Introduction to LCS	478
Why LCS?	479
Terminology	479
Extended learning classifier systems	480
XCS components	482
Application to portfolio management	482
The XCS core data	484
XCS rules	485
Covering	487
An implementation example	487
Benefits and limitations of learning classifier systems	488
Summary	489
Chapter 12: Scalable Frameworks	491
An overview	492
Scala	493
Object creation	493
Streams	493

Table of Contents

Parallel collections	495
Processing a parallel collection	496
The benchmark framework	497
Performance evaluation	499
Scalability with Actors	502
The Actor model	502
Partitioning	504
Beyond actors – reactive programming	504
Akka	504
Master-workers	506
Exchange of messages	506
Worker actors	508
The workflow controller	508
The master actor	509
Master with routing	511
Distributed discrete Fourier transform	512
Limitations	515
Futures	515
The Actor life cycle	516
Blocking on futures	516
Handling future callbacks	518
Putting it all together	520
Apache Spark	521
Why Spark?	522
Design principles	523
In-memory persistency	523
Laziness	523
Transforms and actions	524
Shared variables	525
Experimenting with Spark	527
Deploying Spark	527
Using Spark shell	529
MLlib	530
RDD generation	531
K-means using Spark	532
Performance evaluation	534
Tuning parameters	535
Tests	535
Performance considerations	537
Pros and cons	537
Oxdata Sparkling Water	538
Summary	539
Appendix A: Basic Concepts	541
Scala programming	541
List of libraries and tools	541

Table of Contents

Code snippets format	542
Best practices	544
Encapsulation	544
Class constructor template	544
Companion objects versus case classes	545
Enumerations versus case classes	545
Overloading	546
Design template for immutable classifiers	546
Utility classes	548
Data extraction	548
Data sources	550
Extraction of documents	551
DMatrix class	552
Counter	552
Monitor	553
Mathematics	554
Linear algebra	554
QR decomposition	554
LU factorization	555
LDL decomposition	555
Cholesky factorization	555
Singular Value Decomposition	555
Eigenvalue decomposition	556
Algebraic and numerical libraries	556
First order predicate logic	557
Jacobian and Hessian matrices	558
Summary of optimization techniques	558
Gradient descent methods	559
Quasi-Newton algorithms	560
Nonlinear least squares minimization	562
Lagrange multipliers	563
Overview of dynamic programming	563
Finances 101	564
Fundamental analysis	564
Technical analysis	565
Terminology	566
Trading data	567
Trading signals and strategy	568
Price patterns	569
Options trading	569
Financial data sources	570
Suggested online courses	571
References	571
Index	573

Preface

Not a single day passes by that we do not hear about Big Data in the news media, technical conferences, and even coffee shops. The ever-increasing amount of data collected in process monitoring, research, or simple human behavior becomes valuable only if you extract knowledge from it. Machine learning is the essential tool to mine data for gold (knowledge).

This book covers the "what", "why", and "how" of machine learning:

- What are the objectives and the mathematical foundation of machine learning?
- Why is Scala the ideal programming language to implement machine learning algorithms?
- How can you apply machine learning to solve real-world problems?

Throughout this book, machine learning algorithms are described with diagrams, mathematical formulation, and documented snippets of Scala code, allowing you to understand these key concepts in your own unique way.

What this book covers

Chapter 1, Getting Started, introduces the basic concepts of statistical analysis, classification, regression, prediction, clustering, and optimization. This chapter covers the Scala languages features and libraries, followed by the implementation of a simple application.

Chapter 2, Hello World!, describes a typical workflow for classification, the concept of bias/variance trade-off, and validation using the Scala dependency injection applied to the technical analysis of financial markets.

Chapter 3, Data Preprocessing, covers time series analyses and leverages Scala to implement data preprocessing and smoothing techniques such as moving averages, discrete Fourier transform, and the Kalman recursive filter.

Chapter 4, Unsupervised Learning, focuses on the implementation of some of the most widely used clustering techniques, such as K-means, the expectation-maximization, and the principal component analysis as a dimension reduction method.

Chapter 5, Naïve Bayes Classifiers, introduces probabilistic graphical models, and then describes the implementation of the Naïve Bayes and the multivariate Bernoulli classifiers in the context of text mining.

Chapter 6, Regression and Regularization, covers a typical implementation of the linear and least squares regression, the ridge regression as a regularization technique, and finally, the logistic regression.

Chapter 7, Sequential Data Models, introduces the Markov processes followed by a full implementation of the hidden Markov model, and conditional random fields applied to pattern recognition in financial market data.

Chapter 8, Kernel Models and Support Vector Machines, covers the concept of kernel functions with implementation of support vector machine classification and regression, followed by the application of the one-class SVM to anomaly detection.

Chapter 9, Artificial Neural Networks, describes feed-forward neural networks followed by a full implementation of the multilayer perceptron classifier.

Chapter 10, Genetic Algorithms, covers the basics of evolutionary computing and the implementation of the different components of a multipurpose genetic algorithm.

Chapter 11, Reinforcement Learning, introduces the concept of reinforcement learning with an implementation of the Q-learning algorithm followed by a template to build a learning classifier system.

Chapter 12, Scalable Frameworks, covers some of the artifacts and frameworks to create scalable applications for machine learning such as Scala parallel collections, Akka, and the Apache Spark framework.

Appendix A, Basic Concepts, covers the Scala constructs used throughout the book, elements of linear algebra, and an introduction to investment and trading strategies.

Appendix B, References, provides a chapter-wise list of references for [source entry] in the respective chapters. This appendix is available as an online chapter at https://www.packtpub.com/sites/default/files/downloads/8742OS_AppendixB_References.pdf.

Short test applications using financial data illustrate the large variety of predictive, regression, and classification models.

The interdependencies between chapters are kept to a minimum. You can easily delve into any chapter once you complete *Chapter 1, Getting Started*, and *Chapter 2, Hello World!*.

What you need for this book

A decent command of the Scala programming language is a prerequisite. Reading through a mathematical formulation, conveniently defined in an information box, is optional. However, some basic knowledge of mathematics and statistics might be helpful to understand the inner workings of some algorithms.

The book uses the following libraries:

- Scala 2.10.3 or higher
- Java JDK 1.7.0_45 or 1.8.0_25
- SBT 0.13 or higher
- JFreeChart 1.0.1
- Apache Commons Math library 3.5 (*Chapter 3, Data Preprocessing*, *Chapter 4, Unsupervised Learning*, and *Chapter 6, Regression and Regularization*)
- Indian Institute of Technology Bombay CRF 0.2 (*Chapter 7, Sequential Data Models*)
- LIBSVM 0.1.6 (*Chapter 8, Kernel Models and Support Vector Machines*)
- Akka 2.2.4 or higher (or Typesafe activator 1.2.10 or higher) (*Chapter 12, Scalable Frameworks*)
- Apache Spark 1.3.0 or higher (*Chapter 12, Scalable Frameworks*)



Understanding the mathematical formulation of a model is optional.



Who this book is for

This book is for software developers with a background in Scala programming who want to learn how to create, validate, and apply machine learning algorithms.

The book is also beneficial to data scientists who want to explore functional programming or improve the scalability of their existing applications using Scala.

This book is designed as a tutorial with comparative hands-on exercises using technical analysis of financial markets.

Conventions

In this book, you will find a number of text styles 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: "Finally, the environment variables `JAVA_HOME`, `PATH`, and `CLASSPATH` have to be updated accordingly."

A block of code is set as follows:

```
[default]
val lsp = builder.model(lrJacobian)
    .weight(wMatrix)
    .target(labels)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
val lsp = builder.model(lrJacobian)
    .weight(wMatrix)
    .target(labels)
```

The source code block is described using a reference number embedded as a code comment:

```
[default]
val lsp = builder.model(lrJacobian) //1
    .weight(wMatrix)
    .target(labels)
```

The reference number is used in the chapter as follows: "The `model` instance is initialized with the Jacobian matrix, `lrJacobian` (line 1)."

Any command-line input or output is written as follows:

```
sbt/sbt assembly
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The loss function is then known as the **hinge loss**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.



Mathematical formulas (optional to read) appear in a box like this



For the sake of readability, the elements of the Scala code that are not essential to the understanding of an algorithm such as class, variable, and method qualifiers and validation of arguments, exceptions, or logging are omitted. The convention for code snippets is detailed in the *Code snippets format* section in *Appendix A, Basic Concepts*.

You will be provided with in-text citation of papers, conference, books, and instructional videos throughout the book. The sources are listed in the *Appendix B, References* using in the following format:

[In-text citation]

For example, in the chapter, you will find an instance as follows:

This time around RSS increases with λ before reaching a maximum for $\lambda > 60$. This behavior is consistent with other findings [6:12].

The respective [source entry] is mentioned in *Appendix B, References*, as follows:

[6:12] *Model selection and assessment* H. Bravo, R. Irizarry, 2010, available at <http://www.cbcn.umd.edu/~hcorrada/PracticalML/pdf/lectures/selection.pdf>.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in 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 at 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 from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. 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.

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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Getting Started

It is critical for any computer scientist to understand the different classes of machine learning algorithms and be able to select the ones that are relevant to the domain of their expertise and dataset. However, the application of these algorithms represents a small fraction of the overall effort needed to extract an accurate and performing model from input data. A common data mining workflow consists of the following sequential steps:

1. Defining the problem to solve.
2. Loading the data.
3. Preprocessing, analyzing, and filtering the input data.
4. Discovering patterns, affinities, clusters, and classes, if needed.
5. Selecting the model features and appropriate machine learning algorithm(s).
6. Refining and validating the model.
7. Improving the computational performance of the implementation.

In this book, each stage of the process is critical to build the *right* model.



It is impossible to describe the key machine learning algorithms and their implementations in detail in a single book. The sheer quantity of information and Scala code would overwhelm even the most dedicated readers. Each chapter focuses on the mathematics and code that are absolutely essential to the understanding of the topic. Developers are encouraged to browse through the following:

- The Scala coding convention and standard used in the book in the *Appendix A, Basic Concepts*
- API Scala docs
- A fully documented source code that is available online

This first chapter introduces you to the taxonomy of machine learning algorithms, the tools and frameworks used in the book, and a simple application of logistic regression to get your feet wet.

Mathematical notation for the curious

Each chapter contains a small section dedicated to the formulation of the algorithms for those interested in the mathematical concepts behind the science and art of machine learning. These sections are optional and defined within a tip box. For example, the mathematical expression of the mean and the variance of a variable X mentioned in a tip box will be as follows:



Convention and notation

This book uses zero-based indexing of datasets in the mathematical formulas.

M1: A set of N observations is denoted as $\{x_i\} = x_0, x_1, \dots, x_{N-1}$, and the arithmetic mean value for the random value with x_i as values is defined as:

$$E[X] = \mu = \frac{1}{n} \sum_{i=0}^{n-1} x_i$$

Why machine learning?

The explosion in the number of digital devices generates an ever-increasing amount of data. The best analogy I can find to describe the need, desire, and urgency to extract knowledge from large datasets is the process of extracting a precious metal from a mine, and in some cases, extracting blood from a stone.

Knowledge is quite often defined as a model that can be constantly updated or tweaked as new data comes into play. Models are obviously domain-specific ranging from credit risk assessment, face recognition, maximization of quality of service, classification of pathological symptoms of disease, optimization of computer networks, and security intrusion detection, to customers' online behavior and purchase history.

Machine learning problems are categorized as classification, prediction, optimization, and regression.

Classification

The purpose of classification is to extract knowledge from historical data. For instance, a classifier can be built to identify a disease from a set of symptoms. The scientist collects information regarding the body temperature (continuous variable), congestion (discrete variables *HIGH*, *MEDIUM*, and *LOW*), and the actual diagnostic (flu). This dataset is used to create a model such as *IF temperature > 102 AND congestion = HIGH THEN patient has the flu (probability 0.72)*, which doctors can use in their diagnostic.

Prediction

Once the model is trained using historical observations and validated against historical observations, it can be used to predict some outcome. A doctor collects symptoms from a patient, such as body temperature and nasal congestion, and anticipates the state of his/her health.

Optimization

Some global optimization problems are intractable using traditional linear and non-linear optimization methods. Machine learning techniques improve the chances that the optimization method converges toward a solution (intelligent search). You can imagine that fighting the spread of a new virus requires optimizing a process that may evolve over time as more symptoms and cases are uncovered.

Regression

Regression is a classification technique that is particularly suitable for a continuous model. Linear (least squares), polynomial, and logistic regressions are among the most commonly used techniques to fit a parametric model, or function, $y=f(x)$, $x=\{x_i\}$, to a dataset. Regression is sometimes regarded as a specialized case of classification for which the output variables are continuous instead of categorical.

Why Scala?

Like most functional languages, Scala provides developers and scientists with a toolbox to implement iterative computations that can be easily woven into a coherent dataflow. To some extent, Scala can be regarded as an extension of the popular MapReduce model for distributed computation of large amounts of data. Among the capabilities of the language, the following features are deemed essential in machine learning and statistical analysis.

Abstraction

Functors and **monads** are important concepts in functional programming. Monads are derived from the category and group theory that allow developers to create a high-level abstraction as illustrated in **Scalaz**, Twitter's **Algebird**, or Google's **Breeze Scala** libraries. More information about these libraries can be found at the following links:

- <https://github.com/scalaz>
- <https://github.com/twitter/algebird>
- <https://github.com/dlwh/breeze>

In mathematics, a category **M** is a structure that is defined by:

- Objects of some type: $\{x \in X, y \in Y, z \in Z, \dots\}$
- Morphisms or maps applied to these objects: $x \in X, y \in Y, f: x \rightarrow y$
- Composition of morphisms: $f: x \rightarrow y, g: y \rightarrow z \Rightarrow g \circ f: x \rightarrow z$

Covariant, contravariant functors, and **bifunctors** are well-understood concepts in algebraic topology that are related to manifold and vector bundles. They are commonly used in differential geometry and generation of non-linear models from data.

Higher-kind projection

Scientists define observations as sets or vectors of features. Classification problems rely on the estimation of the similarity between vectors of observations. One technique consists of comparing two vectors by computing the normalized inner product. A **co-vector** is defined as a linear map α of a vector to the inner product (field).



Inner product
M1: The definition of a $\langle \cdot, \cdot \rangle$ inner product and a α co-vector is as follows:

$$\langle \vec{v}, \vec{w} \rangle = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}| \cdot |\vec{w}|} \quad \alpha: \vec{v} \rightarrow \langle \vec{v}, \vec{w} \rangle$$

Let's define a vector as a constructor from any `_ => Vector[_]` field (or `Function1[_ , Vector]`). A co-vector is then defined as the mapping function of a vector to its `Vector[_] => _` field (or `Function1[Vector, _]`).

Let's define a two-dimensional (two types or fields) higher kind structure, `Hom`, that can be defined as either a vector or co-vector by fixing one of the two types:

```
type Hom[T] = {
    type Right[X] = Function1[X, T] // Co-vector
    type Left[X] = Function1[T, X]   // Vector
}
```

Tensors and manifolds

 Vectors and co-vectors are classes of tensor (contravariant and covariant). Tensors (fields) are used in manifold learning of nonlinear models and in the generation of kernel functions. Manifolds are briefly introduced in the *Manifolds* section under *Dimension reduction* in Chapter 4, *Unsupervised Learning*. The topic of tensor fields and manifold learning is beyond the scope of this book.

The projections of the higher kind, `Hom`, to the `Right` or `Left` single parameter types are known as functors, which are as follows:

- A covariant functor for the `right` projection
- A contravariant functor for the `left` projection.

Covariant functors for vectors

A **covariant functor** of a variable is a map $F: C \Rightarrow C$ such that:

- If $f: x \rightarrow y$ is a morphism on C , then $F(x) \rightarrow F(y)$ is also a morphism on C
- If $id: x \rightarrow x$ is the identity morphism on C , then $F(id)$ is also an identity morphism on C
- If $g: y \rightarrow z$ is also a morphism on C , then $F(g \circ f) = F(g) \circ F(f)$

The definition of the `F[U => V] := F[U] => F[V]` covariant functor in Scala is as follows:

```
trait Functor[M[_]] {
    def map[U, V](m: M[U])(f: U => V): M[V]
}
```

For example, let's consider an observation defined as a n dimension vector of a T type, Obs [T]. The constructor for the observation can be represented as Function1 [T, Obs]. Its ObsFunctor functor is implemented as follows:

```
trait ObsFunctor[T] extends Functor[(Hom[T])#Left] { self =>
    override def map[U,V](vu: Function1[T,U])(f: U =>V):
        Function1[T,V] = f.compose(vu)
}
```

The functor is qualified as a **covariant functor** because the morphism is applied to the return type of the element of Obs as Function1 [T, Obs]. The Hom projection of the two parameters types to a vector is implemented as (Hom[T])#Left.

Contravariant functors for co-vectors

A contravariant functor of one variable is a map $F: C \Rightarrow C$ such that:

- If $f: x \rightarrow y$ is a morphism on C , then $F(y) \rightarrow F(x)$ is also a morphism on C
- If $id: x \rightarrow x$ is the identity morphism on C , then $F(id)$ is also an identity morphism on C
- If $g: y \rightarrow z$ is also a morphism on C , then $F(g \circ f) = F(f) \circ F(g)$

The definition of the $F[U \Rightarrow V] := F[V] \Rightarrow F[U]$ contravariant functor in Scala is as follows:

```
trait CoFunctor[M[_]] {
    def map[U,V](m: M[U])(f: V =>U): M[V]
}
```

Note that the input and output types in the f morphism are reversed from the definition of a covariant functor. The constructor for the co-vector can be represented as Function1 [Obs, T]. Its CoObsFunctor functor is implemented as follows:

```
trait CoObsFunctor[T] extends CoFunctor[(Hom[T])#Right] {
    self =>
    override def map[U,V](vu: Function1[U,T])(f: V =>U):
        Function1[V,T] = f.andThen(vu)
}
```

Monads

Monads are structures in algebraic topology that are related to the category theory. Monads extend the concept of a functor to allow a composition known as the **monadic composition** of morphisms on a single type. They enable the chaining or weaving of computation into a sequence of steps or pipeline. The collections bundled with the Scala standard library (`List`, `Map`, and so on) are constructed as monads [1:1].

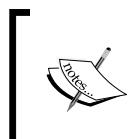
Monads provide the ability for those collections to perform the following functions:

- Create the collection
- Transform the elements of the collection
- Flatten nested collections

An example is as follows:

```
trait Monad[M[_]] {
    def unit[T] (a: T) : M[T]
    def map[U,V] (m: M[U]) (f: U => V) : M[V]
    def flatMap[U,V] (m: M[U]) (f: U => M[V]) : M[V]
}
```

Monads are therefore critical in machine learning as they enable you to compose multiple data transformation functions into a sequence or workflow. This property is applicable to any type of complex scientific computation [1:2].



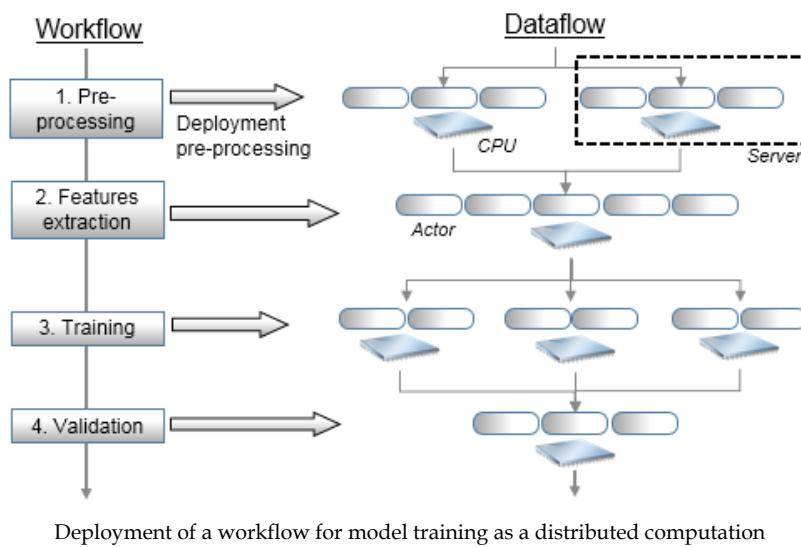
The monadic composition of kernel functions

Monads are used in the composition of kernel functions in the *Kernel monadic composition section under Kernel functions* section in *Chapter 8, Kernel Models and Support Vector Machines*.

Scalability

As seen previously, functors and monads enable parallelization and chaining of data processing functions by leveraging the Scala higher-order methods. In terms of implementation, **actors** are one of the core elements that make Scala scalable. Actors provide Scala developers with a high level of abstraction to build scalable, distributed, and concurrent applications. Actors hide the nitty-gritty implementation details of concurrency and the management of the underlying threads pool. Actors communicate through asynchronous immutable messages. A distributed computing Scala framework such as **Akka** or **Apache Spark** extends the capabilities of the Scala standard library to support computation on very large datasets. Akka and Apache Spark are described in detail in the last chapter of this book [1:3].

In a nutshell, a workflow is implemented as a sequence of activities or computational tasks. These tasks consist of high-order Scala methods such as `flatMap`, `map`, `fold`, `reduce`, `collect`, `join`, or `filter` that are applied to a large collection of observations. Scala provides developers with the tools to partition datasets and execute the tasks through a cluster of actors. Scala also supports message dispatching and routing between local and remote actors. A developer can decide to deploy a workflow either locally or across multiple CPU cores and servers with very few code alterations.



In the preceding diagram, a controller, that is, the master node, manages the sequence of tasks **1** to **4** similar to a scheduler. These tasks are actually executed over multiple worker nodes, which are implemented by actors. The master node or actor exchanges messages with the workers to manage the state of the execution of the workflow as well as its reliability, as illustrated in the *Scalability with Actors* section in *Chapter 12, Scalable Frameworks*. High availability of these tasks is implemented through a hierarchy of supervising actors.

Configurability

Scala supports **dependency injection** using a combination of abstract variables, self-referenced composition, and stackable traits. One of the most commonly used dependency injection patterns, the **cake pattern**, is described in the *Composing mixins to build a workflow* section in *Chapter 2, Hello World!*

Maintainability

Scala embeds **Domain Specific Languages (DSL)** natively. DSLs are syntactic layers built on top of Scala native libraries. DSLs allow software developers to abstract computation in terms that are easily understood by scientists. The most notorious application of DSLs is the definition of the emulation of the syntax used in the MATLAB program, which data scientists are familiar with.

Computation on demand

Lazy methods and values allow developers to execute functions and allocate computing resources on demand. The Spark framework relies on lazy variables and methods to chain **Resilient Distributed Datasets (RDD)**.

Model categorization

A model can be predictive, descriptive, or adaptive.

Predictive models discover patterns in historical data and extract fundamental trends and relationships between factors (or features). They are used to predict and classify future events or observations. Predictive analytics is used in a variety of fields, including marketing, insurance, and pharmaceuticals. Predictive models are created through supervised learning using a preselected training set.

Descriptive models attempt to find unusual patterns or affinities in data by grouping observations into clusters with similar properties. These models define the first and important step in knowledge discovery. They are generated through unsupervised learning.

A third category of models, known as **adaptive modeling**, is created through **reinforcement learning**. Reinforcement learning consists of one or several decision-making agents that recommend and possibly execute actions in the attempt of solving a problem, optimizing an objective function, or resolving constraints.

Taxonomy of machine learning algorithms

The purpose of machine learning is to teach computers to execute tasks without human intervention. An increasing number of applications such as genomics, social networking, advertising, or risk analysis generate a very large amount of data that can be analyzed or mined to extract knowledge or insight into a process, customer, or organization. Ultimately, machine learning algorithms consist of identifying and validating models to optimize a performance criterion using historical, present, and future data [1:4].

Data mining is the process of extracting or identifying patterns in a dataset.

Unsupervised learning

The goal of **unsupervised learning** is to discover patterns of regularities and irregularities in a set of observations. The process is known as density estimation in statistics is broken down into two categories: discovery of data clusters and discovery of latent factors. The methodology consists of processing input data to understand patterns similar to the natural learning process in infants or animals. Unsupervised learning does not require labeled data (or expected values), and therefore, it is easy to implement and execute because no expertise is needed to validate an output. However, it is possible to label the output of a clustering algorithm and use it for future classification.

Clustering

The purpose of **data clustering** is to partition a collection of data into a number of clusters or data segments. Practically, a clustering algorithm is used to organize observations into clusters by minimizing the distance between observations within a cluster and maximizing the distance between observations across clusters. A clustering algorithm consists of the following steps:

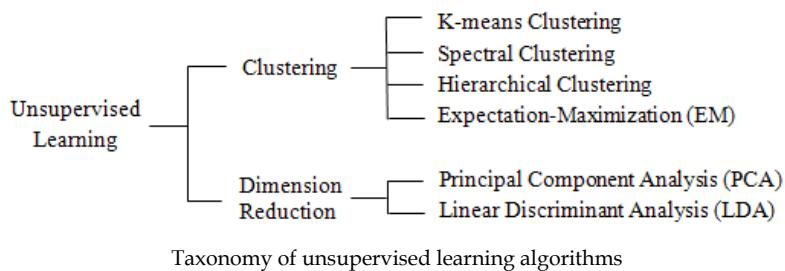
1. Creating a model by making an assumption on the input data.
2. Selecting the objective function or goal of the clustering.
3. Evaluating one or more algorithms to optimize the objective function.

Data clustering is also known as **data segmentation** or **data partitioning**.

Dimension reduction

Dimension reduction techniques aim at finding the smallest but most relevant set of features needed to build a reliable model. There are many reasons for reducing the number of features or parameters in a model, from avoiding overfitting to reducing computation costs.

There are many ways to classify the different techniques used to extract knowledge from data using unsupervised learning. The following taxonomy breaks down these techniques according to their purpose, although the list is far from being exhaustive, as shown in the following diagram:



Supervised learning

The best analogy for supervised learning is **function approximation** or **curve fitting**. In its simplest form, supervised learning attempts to find a relation or function $f: x \rightarrow y$ using a training set $\{x, y\}$. Supervised learning is far more accurate than any other learning strategy as long as the input (labeled data) is available and reliable. The downside is that a domain expert may be required to label (or tag) data as a training set.

Supervised machine learning algorithms can be broken into two categories:

- Generative models
- Discriminative models

Generative models

In order to simplify the description of a statistics formula, we adopt the following simplification: the probability of an X event is the same as the probability of the discrete X random variable to have a value x : $p(X) = p(X=x)$.

The notation for the joint probability is $p(X, Y) = p(X=x, Y=y)$.

The notation for the conditional probability is $p(X | Y) = p(X=x | Y=y)$.

Generative models attempt to fit a joint probability distribution, $p(X, Y)$, of two X and Y events (or random variables), representing two sets of observed and hidden x and y variables. Discriminative models compute the conditional probability, $p(Y | X)$, of an event or random variable Y of hidden variables y , given an event or random variable X of observed variables x . Generative models are commonly introduced through the Bayes' rule. The conditional probability of a Y event, given an X event, is computed as the product of the conditional probability of the X event, given the Y event, and the probability of the X event normalized by the probability of the Y event [1:5].

Bayes' rule

Joint probability for independent random variables, $X=x$ and $Y=y$, is given by:

$$p(X, Y) = p(X \cap Y) = p(X) \cdot p(Y)$$



Conditional probability of a random variable, $Y = y$, given $X = x$, is given by:

$$p(Y|X) = p(Y, X)/p(X)$$

Bayes' formula is given by:

$$p(Y|X) = p(X|Y) \cdot p(Y)/p(X)$$

The Bayes' rule is the foundation of the Naïve Bayes classifier, as described in the *Introducing the multinomial Naïve Bayes* section in *Chapter 5, Naïve Bayes Classifiers*.

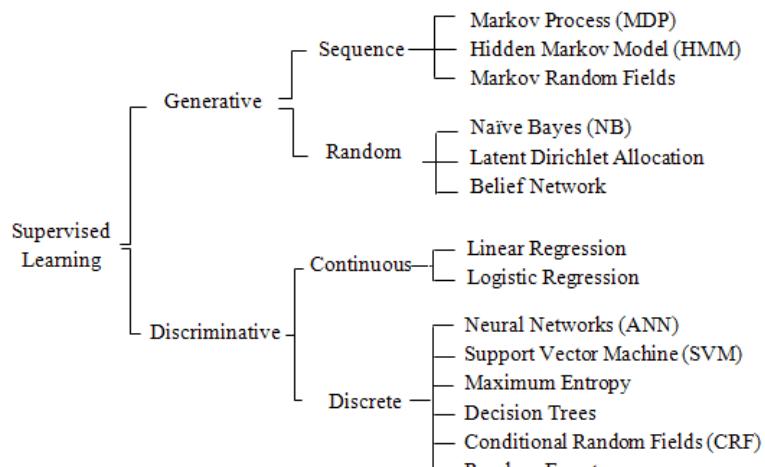
Discriminative models

Contrary to generative models, discriminative models compute the conditional probability $p(Y | X)$ directly, using the same algorithm for training and classification.

Generative and discriminative models have their respective advantages and disadvantages. Novice data scientists learn to match the appropriate algorithm to each problem through experimentation. Here is a brief guideline describing which type of models make sense according to the objective or criteria of the project:

Objective	Generative models	Discriminative models
Accuracy	Highly dependent on the training set.	This depends on the training set and algorithm configuration (that is, kernel functions)
Modeling requirements	There is a need to model both observed and hidden variables, which requires a significant amount of training.	The quality of the training set does not have to be as rigorous as for generative models.
Computation cost	This is usually low. For example, any graphical method derived from the Bayes' rule has low overhead.	Most algorithms rely on optimization of a convex function with significant performance overhead.
Constraints	These models assume some degree of independence among the model features.	Most discriminative algorithms accommodate dependencies between features.

We can further refine the taxonomy of supervised learning algorithms by segregating arbitrarily between sequential and random variables for generative models and breaking down discriminative methods as applied to continuous processes (regression) and discrete processes (classification):



Taxonomy of supervised learning algorithms

Semi-supervised learning

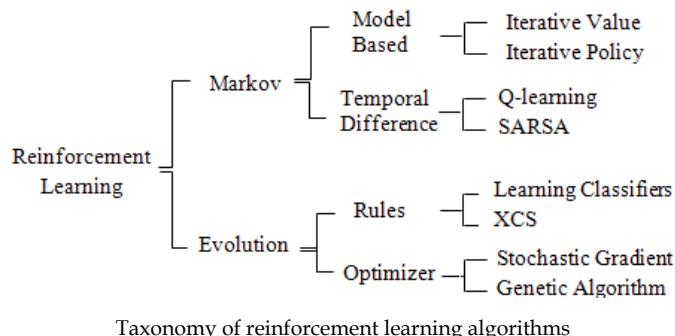
Semi-supervised learning is used to build models from a dataset with incomplete labels. Manifold learning and information geometry algorithms are commonly applied to large datasets that are partially labeled. The description of semi-supervised learning techniques is beyond the scope of this book.

Reinforcement learning

Reinforcement learning is not as well understood as supervised and unsupervised learning outside the realms of robotics or game strategy. However, since the 90s, genetic-algorithms-based classifiers have become increasingly popular to solve problems that require collaboration with a domain expert. For some types of applications, reinforcement learning algorithms output a set of recommended actions for the adaptive system to execute. In its simplest form, these algorithms estimate the best course of action. Most complex systems based on reinforcement learning establish and update policies that can be vetoed by an expert, if necessary. The foremost challenge developers of reinforcement learning systems face is that the recommended action or policy may depend on partially observable states.

Genetic algorithms are not usually considered part of the reinforcement learning toolbox. However, advanced models, such as learning classifier systems, use genetic algorithms to classify and reward the most performing rules and policies.

As with the two previous learning strategies, reinforcement learning models can be categorized as Markovian or evolutionary:



Taxonomy of reinforcement learning algorithms

This is a brief overview of machine learning algorithms with a suggested, approximate taxonomy. There are almost as many ways to introduce machine learning as there are data and computer scientists. We encourage you to browse through the list of references at the end of the book to find the documentation appropriate to your level of interest and understanding.

Don't reinvent the wheel!

There are numerous robust, accurate, and efficient Java libraries for mathematics, linear algebra, or optimization that have been widely used for many years:

- JBlas/Linpack (<https://github.com/mikiobraun/jblas>)
- Parallel Colt (<https://github.com/rwl/ParallelColt>)
- Apache Commons Math (<http://commons.apache.org/proper/commons-math>)

There is absolutely no need to rewrite, debug, and test these components in Scala. Developers should consider creating a wrapper or interface to his/her favorite and reliable Java library. The book leverages the Apache Commons Math library for some specific linear algebra algorithms.

Tools and frameworks

Before getting your hands dirty, you need to download and deploy a minimum set of tools and libraries; there is no need to reinvent the wheel after all. A few key components have to be installed in order to compile and run the source code described throughout the book. We focus on open source and commonly available libraries, although you are invited to experiment with equivalent tools of your choice. The learning curve for the frameworks described here is minimal.

Java

The code described in this book has been tested with JDK 1.7.0_45 and JDK 1.8.0_25 on Windows x64 and Mac OS X x64. You need to install the Java Development Kit if you have not already done so. Finally, the `JAVA_HOME`, `PATH`, and `CLASSPATH` environment variables have to be updated accordingly.

Scala

The code has been tested with Scala 2.10.4 and 2.11.4. We recommend that you use Scala Version 2.10.4 or higher with SBT 0.13 or higher. Let's assume that Scala runtime (REPL) and libraries have been properly installed and the `SCALA_HOME` and `PATH` environment variables have been updated.

The description and installation instructions of the **Scala plugin for Eclipse** (version 4.0 or higher) are available at <http://scala-ide.org/docs/user/gettingstarted.html>. You can also download the **Scala plugin for IntelliJ IDEA** (version 13 or higher) from the JetBrains website at <http://confluence.jetbrains.com/display/SCA/>.

The ubiquitous **Simple Build Tool (SBT)** will be our primary building engine. The syntax of the build file, `sbt/build.sbt`, conforms to the Version 0.13 and is used to compile and assemble the source code presented throughout the book. Sbt can be downloaded as part of Typesafe activator or directly from <http://www.scala-sbt.org/download.html>.

Apache Commons Math

Apache Commons Math is a Java library used for numerical processing, algebra, statistics, and optimization [1:6].

Description

This is a lightweight library that provides developers with a foundation of small, ready-to-use Java classes that can be easily weaved into a machine learning problem. The examples used throughout the book require Version 3.5 or higher.

The math library supports the following:

- Functions, differentiation, and integral and ordinary differential equations
- Statistics distributions
- Linear and nonlinear optimization
- Dense and sparse vectors and matrices
- Curve fitting, correlation, and regression

For more information, visit <http://commons.apache.org/proper/commons-math>.

Licensing

We need Apache Public License 2.0; the terms are available at <http://www.apache.org/licenses/LICENSE-2.0>.

Installation

The installation and deployment of the Apache Commons Math library are quite simple. The steps are as follows:

1. Go to the download page at http://commons.apache.org/proper/commons-math/download_math.cgi.
2. Download the latest .jar files to the binary section, `commons-math3-3.5-bin.zip` (for instance, for Version 3.5).
3. Unzip and install the .jar file.

4. Add commons-math3-3.5.jar to the classpath as follows:
 - **For Mac OS X:** export CLASSPATH=\$CLASSPATH:/Commons_Math_path/commons-math3-3.5.jar
 - **For Windows:** Go to system **Properties** | **Advanced system settings** | **Advanced** | **Environment Variables**, then edit the CLASSPATH variable
5. Add the commons-math3-3.5.jar file to your IDE environment if needed (that is, for Eclipse, go to **Project** | **Properties** | **Java Build Path** | **Libraries** | **Add External JARs** and for IntelliJ IDEA, go to **File** | **Project Structure** | **Project Settings** | **Libraries**).

You can also download commons-math3-3.5-src.zip from the **Source** section.

JFreeChart

JFreeChart is an open source chart and plotting Java library, widely used in the Java programmer community. It was originally created by David Gilbert [1:7].

Description

The library supports a variety of configurable plots and charts (scatter, dial, pie, area, bar, box and whisker, stacked, and 3D). We use JFreeChart to display the output of data processing and algorithms throughout the book, but you are encouraged to explore this great library on your own, as time permits.

Licensing

It is distributed under the terms of the **GNU Lesser General Public License (LGPL)**, which permits its use in proprietary applications.

Installation

To install and deploy JFreeChart, perform the following steps:

1. Visit <http://www.jfree.org/jfreechart/>.
2. Download the latest version from Source Forge at <http://sourceforge.net/projects/jfreechart/files>.
3. Unzip and deploy the .jar file.

4. Add `jfreechart-1.0.17.jar` (for Version 1.0.17) to the classpath as follows:
 - **For Mac OS X:** `export CLASSPATH=$CLASSPATH:/JFreeChart_path/jfreechart-1.0.17.jar`
 - **For Windows:** Go to system **Properties** | **Advanced system settings** | **Advanced** | **Environment Variables**, then edit the `CLASSPATH` variable
5. Add the `jfreechart-1.0.17.jar` file to your IDE environment, if needed

Other libraries and frameworks

Libraries and tools that are specific to a single chapter are introduced along with the topic. Scalable frameworks are presented in the last chapter along with the instructions to download them. Libraries related to the conditional random fields and support vector machines are described in their respective chapters.



Why not use the Scala algebra and numerical libraries?

Libraries such as Breeze, ScalaNLP, and Algebird are interesting Scala frameworks for linear algebra, numerical analysis, and machine learning. They provide even the most seasoned Scala programmer with a high-quality layer of abstraction. However, this book is designed as a tutorial that allows developers to write algorithms from the ground up using existing or legacy Java libraries [1:8].

Source code

The Scala programming language is used to implement and evaluate the machine learning techniques covered in *Scala for Machine Learning*. However, the source code snippets are reduced to the strict minimum essential to the understanding of machine learning algorithms discussed throughout the book. The formal implementation of these algorithms is available on the website of Packt Publishing (<http://www.packtpub.com>).



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.

Context versus view bounds

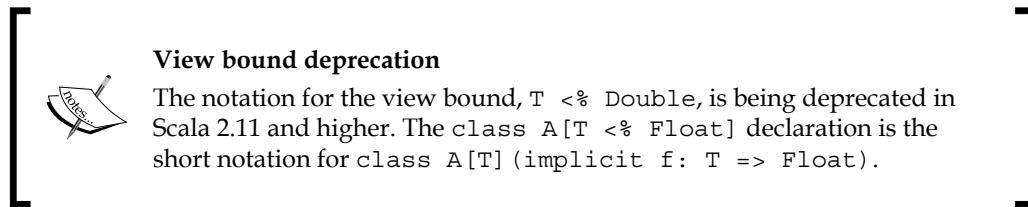
Most Scala classes discussed in the book are parameterized with the type associated with the discrete/categorical value (`Int`) or continuous value (`Double`). Context bounds would require that any type used by the client code has `Int` or `Double` as upper bounds:

```
class A[T <: Int] (param: Param)
class B[T <: Double] (param: Param)
```

Such a design introduces constraints on the client to inherit from simple types and to deal with covariance and contravariance for container types [1:9].

For this book, **view bounds** are used instead of context bounds because they only require an implicit conversion to the parameterized type to be defined:

```
class A[T <: AnyVal] (param: Param) (implicit f: T => Int)
class C[T <: AnyVal] (param: Param) (implicit f: T => Float)
```



Presentation

For the sake of readability of the implementation of algorithms, all nonessential code such as error checking, comments, exceptions, or imports are omitted. The following code elements are omitted in the code snippet presented in the book:

- Code documentation:


```
// ....
/* ... */
```
- Validation of class parameters and method arguments:


```
require( Math.abs(x) < EPS, " ...")
```
- Class qualifiers and scope declaration:


```
final protected class SVM { ... }
private[this] val lsError = ...
```

- Method qualifiers:

```
final protected def dot: = ...
```

- Exceptions:

```
try {
    correlate ...
} catch {
    case e: MathException => ....
}
Try {    .. } match {
    case Success(res) =>
    case Failure(e => ..
}
```

- Logging and debugging code:

```
private val logger = Logger.getLogger("...")
logger.info( ... )
```

- Nonessential annotation:

```
@inline def main = ....
@throw(classOf[IllegalStateException])
```

- Nonessential methods

The complete list of Scala code elements omitted in the code snippets in this book can be found in the *Code snippets format* section in the *Appendix A, Basic Concepts*.

Primitives and implicits

The algorithms presented in this book share the same primitive types, generic operators, and implicit conversions.

Primitive types

For the sake of readability of the code, the following primitive types will be used:

```
type DblPair = (Double, Double)
type DblArray = Array[Double]
type DblMatrix = Array[DblArray]
type DblVector = Vector[Double]
type XSeries[T] = Vector[T]           // One dimensional vector
type XVSeries[T] = Vector[Array[T]]   // multi-dimensional vector
```

The times series introduced in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*, is implemented as `XSeries[T]` or `XVSeries[T]` of a parameterized `T` type.



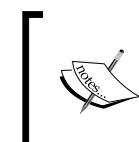
Make a note of these six types; they are used throughout the book.



Type conversions

Implicit conversion is an important feature of the Scala programming language. It allows developers to specify a type conversion for an entire library in a single place. Here are a few of the implicit type conversions that are used throughout the book:

```
object Types {
    object ScalaML {
        implicit def double2Array(x: Double): DblArray =
            Array[Double](x)
        implicit def dblPair2Vector(x: DblPair): Vector[DblPair] =
            Vector[DblPair](x._1, x._2)
        ...
    }
}
```



Library-specific conversion

The conversion between the primitive type listed here and types introduced in a particular library (such as, the Apache Commons Math library) are described in the relevant chapters.



Immutability

It is usually a good idea to reduce the number of states of an object. A method invocation transitions an object from one state to another. The larger the number of methods or states, the more cumbersome the testing process becomes.

There is no point in creating a model that is not defined (trained). Therefore, making the training of a model as part of the constructor of the class it implements makes a lot of sense. Therefore, the only public methods of a machine learning algorithm are as follows:

- Classification or prediction
- Validation
- Retrieval of model parameters (weights, latent variables, hidden states, and so on), if needed

Performance of Scala iterators

The evaluation of the performance of Scala high-order iterative methods is beyond the scope of this book. However, it is important to be aware of the trade-off of each method.

The `for` construct is to be avoided as a counting iterator. It is designed to implement the for-comprehensive monad (`map` and `flatMap`). The source code presented in this book uses the high-order `foreach` method instead.

Let's kick the tires

This final section introduces the key elements of the training and classification workflow. A test case using a simple logistic regression is used to illustrate each step of the computational workflow.

An overview of computational workflows

In its simplest form, a computational workflow to perform runtime processing of a dataset is composed of the following stages:

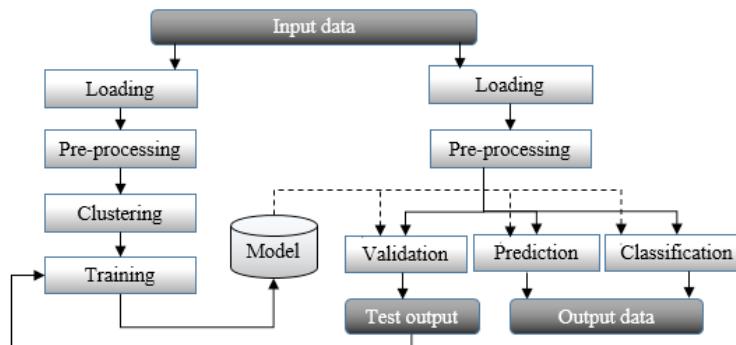
1. Loading the dataset from files, databases, or any streaming devices.
2. Splitting the dataset for parallel data processing.
3. Preprocessing data using filtering techniques, analysis of variance, and applying penalty and normalization functions whenever necessary.
4. Applying the model – either a set of clusters or classes – to classify new data.
5. Assessing the quality of the model.

A similar sequence of tasks is used to extract a model from a training dataset:

1. Loading the dataset from files, databases, or any streaming devices.
2. Splitting the dataset for parallel data processing.
3. Applying filtering techniques, analysis of variance, and penalty and normalization functions to the raw dataset whenever necessary.
4. Selecting the training, testing, and validation set from the cleansed input data.
5. Extracting key features and establishing affinity between a similar group of observations using clustering techniques or supervised learning algorithms.
6. Reducing the number of features to a manageable set of attributes to avoid overfitting the training set.

7. Validating the model and tuning the model by iterating steps 5, 6, and 7 until the error meets a predefined convergence criteria.
8. Storing the model in a file or database so that it can be applied to future observations.

Data clustering and data classification can be performed independent of each other or as part of a workflow that uses clustering techniques at the preprocessing stage of the training phase of a supervised learning algorithm. Data clustering does not require a model to be extracted from a training set, while classification can be performed only if a model has been built from the training set. The following image gives an overview of training, classification, and validation:



A generic data flow for training and running a model

The preceding diagram is an overview of a typical data mining processing pipeline. The first phase consists of extracting the model through clustering or training of a supervised learning algorithm. The model is then validated against test data for which the source is the same as the training set but with different observations. Once the model is created and validated, it can be used to classify real-time data or predict future behavior. Real-world workflows are more complex and require dynamic configuration to allow experimentation of different models. Several alternative classifiers can be used to perform a regression and different filtering algorithms are applied against input data, depending on the latent noise in the raw data.

Writing a simple workflow

This book relies on financial data to experiment with different learning strategies. The objective of the exercise is to build a model that can discriminate between volatile and nonvolatile trading sessions for stock or commodities. For the first example, we select a simplified version of the binomial logistic regression as our classifier as we treat stock-price-volume action as a continuous or pseudo-continuous process.

An introduction to the logistic regression



Logistic regression is explained in depth in the *Logistic regression* section in *Chapter 6, Regression and Regularization*. The model treated in this example is the simple binomial logistic regression classifier for two-dimension observations.

The steps for classification of trading sessions according to their volatility and volume is as follows:

1. Scoping the problem
2. Loading data
3. Preprocessing raw data
4. Discovering patterns, whenever possible
5. Implementing the classifier
6. Evaluating the model

Step 1 – scoping the problem

The objective is to create a model for stock price using its daily trading volume and volatility. Throughout the book, we will rely on financial data to evaluate and discuss the merits of different data processing and machine learning methods. In this example, the data is extracted from **Yahoo Finances** using the CSV format with the following fields:

- Date
- Price at open
- Highest price in the session
- Lowest price in the session
- Price at session close
- Volume
- Adjust price at session close

The `YahooFinancials` enumerator extracts the historical daily trading information from the Yahoo finance site:

```
type Fields = Array[String]
object YahooFinancials extends Enumeration {
    type YahooFinancials = Value
```

```

val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME, ADJ_CLOSE = Value

deftoDouble(v: Value): Fields => Double = //1
(s: Fields) => s(v.id).toDouble
def toDblArray(vs: Array[Value]): Fields => DblArray = //2
(s: Fields) => vs.map(v => s(v.id).toDouble)

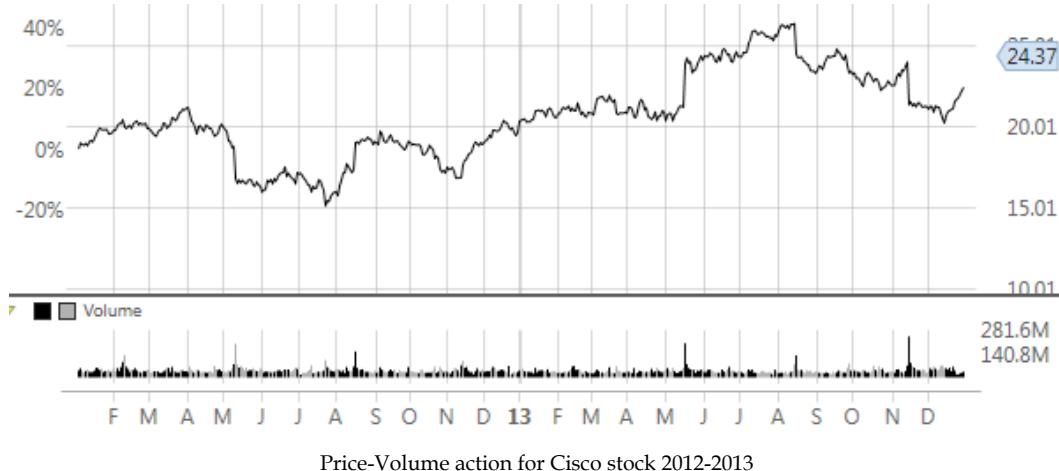
...
}

```

The `toDouble` method converts an array of string into a single value (line 1) and `toDblArray` converts an array of string into an array of values (line 2). The `YahooFinancials` enumerator is described in the *Data sources* section in *Appendix A, Basic Concepts* in detail.

Let's create a simple program that loads the content of the file, executes some simple preprocessing functions, and creates a simple model. We selected the CSCO stock price between January 1, 2012 and December 1, 2013 as our data input.

Let's consider the two variables, *price* and *volume*, as shown in the following screenshot. The top graph displays the variation of the price of Cisco stock over time and the bottom bar chart represents the daily trading volume on Cisco stock over time:



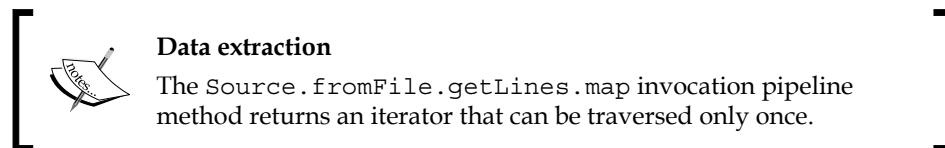
Step 2 – loading data

The second step is loading the dataset from a local or remote data storage.

Typically, large datasets are loaded from a database or distributed filesystems such as **Hadoop Distributed File System (HDFS)**. The `load` method takes an absolute pathname, extract, and transforms the input data from a file into a time series of a `Vector[DblPair]` type:

```
def load(fileName: String): Try[Vector[DblPair]] = Try {  
    val src = Source.fromFile(fileName) //3  
    val data = extract(src.getLines.map(_.split(",")).drop(1)) //4  
    src.close //5  
    data  
}
```

The data file is extracted through an invocation of the `Source.fromFile` static method (line 3), and then the fields are extracted through a map before the header (first row in the file) is removed using `drop` (line 4). The file has to be closed to avoid leaking of the file handle (line 5).



The purpose of the `extract` method is to generate a time series of two variables (*relative stock volatility* and *relative stock daily trading volume*):

```
def extract(cols: Iterator[Array[String]]): Xvseries[Double] = {  
    val features = Array[YahooFinancials](LOW, HIGH, VOLUME) //6  
    val conversion = YahooFinancials.toDblArray(features) //7  
    cols.map(c => conversion(c)).toVector  
        .map(x => Array[Double](1.0 - x(0)/x(1), x(2))) //8  
}
```

The only purpose of the `extract` method is to convert the raw textual data into a two-dimensional time series. The first step consists of selecting the three features to extract `LOW` (the lowest stock price in the session), `HIGH` (the highest price in the session), and `VOLUME` (trading volume for the session) (line 6). This feature set is used to convert each line of fields into a corresponding set of three values (line 7). Finally, the feature set is reduced to the following two variables (line 8):

- Relative volatility of the stock price in a session: $1.0 - LOW/HIGH$
- Trading volume for the stock in the session: `VOLUME`

Code readability

A long pipeline of Scala high-order methods make the code and underlying code quite difficult to read. It is recommended that you break down long chains of method calls, such as the following:

```
val cols = Source.fromFile.getLines.map(_.split(",")).  
toArray.drop(1)
```

We can break down method calls into several steps as follows:

```
val lines = Source.fromFile.getLines  
val fields = lines.map(_.split(",")).toArray  
val cols = fields.drop(1)
```

We strongly encourage you to consult the excellent guide *Effective Scala*, written by Marius Eriksen from Twitter. This is definitively a must read for any Scala developer [1:10].

Step 3 – preprocessing the data

The next step is to normalize the data in the range $[0.0, 1.0]$ to be trained by the binomial logistic regression. It is time to introduce an immutable and flexible normalization class.

Immutable normalization

The logistic regression relies on the sigmoid curve or logistic function is described in the *Logistic function* section in *Chapter 6, Regression and Regularization*. The logistic functions are used to segregate training data into classes. The output value of the logistic function ranges from 0 for $x = -\text{INFINITY}$ to 1 for $x = +\text{INFINITY}$. Therefore, it makes sense to normalize the input data or observation over $[0, 1]$.

Normalize or not normalize?

The purpose of normalizing data is to impose a single range of values for all the features, so the model does not favor any particular feature.

Normalization techniques include linear normalization and Z-score.

Normalization is an expensive operation that is not always needed.

The normalization is a linear transformation of the raw data that can be generalized to any range $[l, h]$.

Linear normalization

M2: [0, 1] Normalization of features $\{x_i\}$ with minimum x_{min} and maximum x_{max} values:



$$y_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

M3: [l, h] Normalization of features $\{x_i\}$:

$$y_i = \frac{x_i - x_{min}}{x_{max} - x_{min}} (h - l) + l$$

The normalization of input data in supervised learning has a specific requirement: the classification and prediction of new observations have to use the normalization parameters (*min* and *max*) extracted from the training set, so all the observations share the same scaling factor.

Let's define the `MinMax` normalization class. The class is immutable: the minimum, `min`, and maximum, `max`, values are computed within the constructor. The class takes a time series of a parameterized `T` type and values as arguments (line 8). The steps of the normalization process are defined as follows:

1. Initialize the minimum values for a given time series during instantiation (line 9).
2. Compute the normalization parameters (line 10) and normalize the input data (line 11).
3. Normalize any new data points reusing the normalization parameters (line 14):

```
class MinMax[T <: AnyVal] (val values: XSeries[T]) (f : T =>
Double) { //8
    val zero = (Double.MaxValue, -Double.MaxValue)
    val minMax = values./:(zero)((mM, x) => { //9
        val min = mM._1
        val max = mM._2
        (if(x < min) x else min, if(x > max) x else max)
    })
    case class ScaleFactors(low:Double ,high:Double, ratio: Double)
    var scaleFactors: Option[ScaleFactors] = None //10

    def min = minMax._1
    def max = minMax._2
    def normalize(low: Double, high: Double): DblVector //11
    def normalize(value: Double): Double
}
```

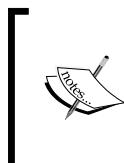
The class constructor computes the tuple of minimum and maximum values, `minMax`, using a fold (line 9). The `scaleFactors` scaling parameters are computed during the normalization of the time series (line 11), which are described as follows. The `normalize` method initializes the scaling factor parameters (line 12) before normalizing the input data (line 13):

```
def normalize(low: Double, high: Double): DblVector =  
    setScaleFactors(low, high).map( scale => { //12  
        values.map(x =>(x - min)*scale.ratio + scale.low) //13  
    }).getOrElse(/* ... */)  
  
def setScaleFactors(l: Double, h: Double): Option[ScaleFactors] ={  
    // .. error handling code  
    Some(ScaleFactors(l, h, (h - l)/(max - min))  
}
```

Subsequent observations use the same scaling factors extracted from the input time series in `normalize` (line 14):

```
def normalize(value: Double):Double = setScaleFactors.map(scale =>  
    if(value < min) scale.low  
    else if (value > max) scale.high  
    else (value - min)* scale.high + scale.low  
).getOrElse( /* ... */)
```

The `MinMax` class normalizes single variable observations.



The statistics class

The class that extracts the basic statistics from a `Stats` dataset, which is introduced in the *Profiling data* section in *Chapter 2, Hello World!*, inherits the `MinMax` class.

The test case with the binomial logistic regression uses a multiple variable normalization, implemented by the `MinMaxVector` class, which takes observations of the `XVSeries[Double]` type as inputs:

```
class MinMaxVector(series: XVSeries[Double]) {  
    val minMaxVector: Vector[MinMax[Double]] = //15  
        series.transpose.map(new MinMax[Double](__))  
    def normalize(low: Double, high: Double): XVSeries[Double]  
}
```

The constructor of the `MinMaxVector` class transposes the vector of array of observations in order to compute the minimum and maximum value for each dimension (line 15).

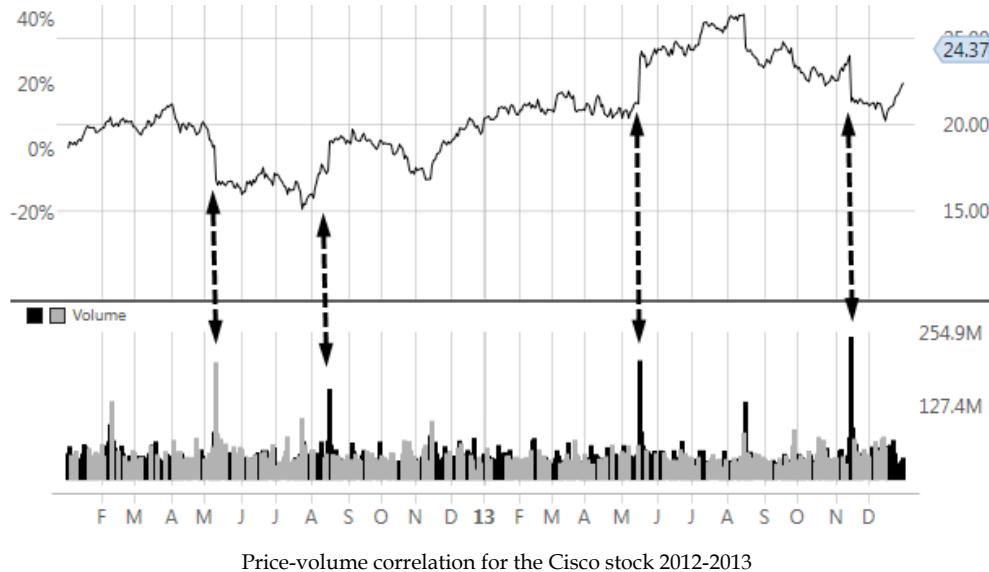
Step 4 – discovering patterns

The price action chart has a very interesting characteristic.

Analyzing data

At a closer look, a sudden change in price and increase in volume occurs about every three months or so. Experienced investors will undoubtedly recognize that these price-volume patterns are related to the release of quarterly earnings of Cisco. Such a regular but unpredictable pattern can be a source of concern or opportunity if risk can be properly managed. The strong reaction of the stock price to the release of corporate earnings may scare some long-term investors while enticing day traders.

The following graph visualizes the potential correlation between sudden price change (volatility) and heavy trading volume:



The next section is not required for the understanding of the test case. It illustrates the capabilities of JFreeChart as a simple visualization and plotting library.

Plotting data

Although charting is not the primary goal of this book, we thought that you will benefit from a brief introduction to JFreeChart.



Plotting classes

This section illustrates a simple Scala interface to JFreeChart Java classes. Reading this is not required for the understanding of machine learning. The visualization of the results of a computation is beyond the scope of this book.

Some of the classes used in visualization are described in the *Appendix A, Basic Concepts*.

The dataset (volatility and volume) is converted into internal JFreeChart data structures. The `ScatterPlot` class implements a simple configurable scatter plot with the following arguments:

- `config`: This includes information, labels, fonts, and so on, of the plot
- `theme`: This is the predefined theme for the plot (black, white background, and so on)

The code will be as follows:

```
class ScatterPlot(config: PlotInfo, theme: PlotTheme) { //16
    def display(xy: Vector[DblPair], width: Int, height) //17
    def display(xt: XVSeries[Double], width: Int, height)
    // ...
}
```

The `PlotTheme` class defines a specific theme or preconfiguration of the chart (line 16). The class offers a set of `display` methods to accommodate a wide range of data structures and configuration (line 17).



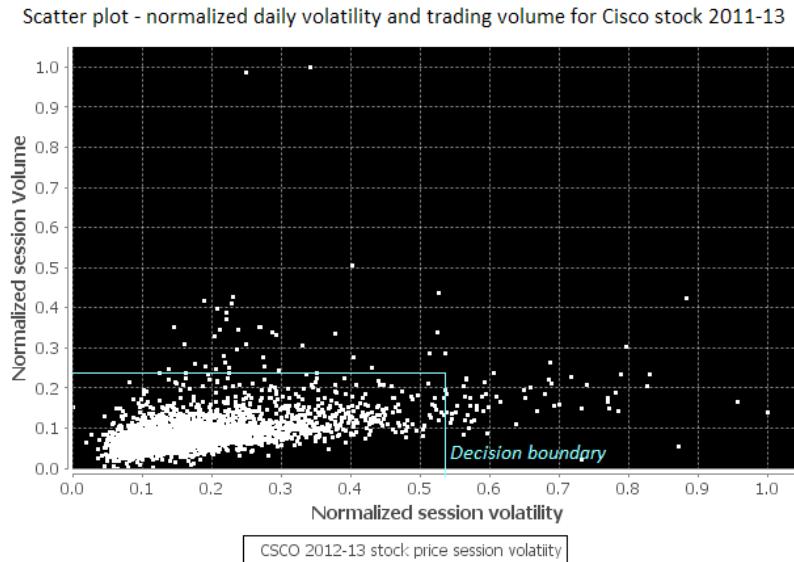
Visualization

The JFreeChart library is introduced as a robust charting tool. The code related to plots and charts is omitted from the book in order to keep the code snippets concise and dedicated to machine learning. On a few occasions, output data is formatted as a CSV file to be imported into a spreadsheet.

Getting Started

The `ScatterPlot.display` method is used to display the normalized input data used in the binomial logistic regression as follows:

```
val plot = new ScatterPlot(("CSCO 2012-2013",
    "Session High - Low", "Session Volume"), new BlackPlotTheme)
plot.display(volatility_vol, 250, 340)
```



A scatter plot of volatility and volume for the Cisco stock 2012-2013

The scatter plot shows a level of correlation between session volume and session volatility and confirms the initial finding in the stock price and volume chart. We can leverage this information to classify trading sessions by their volatility and volume. The next step is to create a two class model by loading a training set, observations, and expected values, into our logistic regression algorithm. The classes are delimited by a **decision boundary** (also known as a hyperplane) drawn on the scatter plot.

Visualizing labels – the normalized variation of the stock price between the opening and closing of the trading session is selected as the label for this classifier.

Step 5 – implementing the classifier

The objective of this training is to build a model that can discriminate between volatile and nonvolatile trading sessions. For the sake of the exercise, session volatility is defined as the relative difference between the session highest price and lower price. The total trading volume within a session constitutes the second parameter of the model. The relative price movement within a trading session (that is, *closing price/open price - 1*) is our expected values or labels.

Logistic regression is commonly used in statistics inference.

M4: Logistic regression model



$$f(\mathbf{x}|\mathbf{w}) = w_0 + \sum_{i=1}^{N-1} x_i w_i \quad l(\mathbf{x}|\mathbf{w}) = \frac{1}{1 + e^{-f(\mathbf{x}|\mathbf{w})}}$$

The first weight w_0 is known as the intercept. The binomial logistic regression is described in the *Logistic regression* section in *Chapter 6, Regression and Regularization*, in detail.

The following implementation of the binomial logistic regression classifier exposes a single `classify` method to comply with our desire to reduce the complexity and life cycle of objects. The model weights parameters are computed during training when the `LogBinRegression` class/model is instantiated. As mentioned earlier, the sections of the code nonessential to the understanding of the algorithm are omitted.

The `LogBinRegression` constructor has five arguments (line 18):

- `obsSet`: These are vector observations that represent volume and volatility
- `expected`: This is a vector of expected values
- `maxIters`: This is the maximum number of iterations allowed for the optimizer to extract the regression weights during training
- `eta`: This is the learning or training rate
- `eps`: This is the maximum value of the error (*predicted – expected*) for which the model is valid

The code is as follows:

```
class LogBinRegression(
    obsSet: Vector[DblArray],
    expected: Vector[Int],
    maxIters: Int,
    eta: Double,
    eps: Double) { //18

    val model: LogBinRegressionModel = train //19
    def classify(obs: DblArray): Try[(Int, Double)] //20
    def train: LogBinRegressionModel
    def intercept(weights: DblArray): Double
    ...
}
```

The `LogBinRegressionModel` model is generated through training during the instantiation of the `LogBinRegression` logistic regression class (line 19):

```
case class LogBinRegressionModel(val weights: DblArray)
```

The model is fully defined by its weights, as described in the mathematical formula **M3**. The `weights(0)` intercept represents the mean value of the prediction for observations for which variables are zero. The intercept does not have any specific meaning for most of the cases and it is not always computable.

Intercept or not intercept?

The intercept corresponds to the value of weights when the observations have null values. It is a common practice to estimate, whenever possible, the intercept for binomial linear or logistic regression independently from the slope of the model in the minimization of the error function. The multinomial regression models treat the intercept or weight w_0 as part of the regression model, as described in the *Ordinary least squares regression* section of *Chapter 6, Regression and Regularization*.

The code will be as follows:

```
def intercept(weights: DblArray): Double = {
    val zeroObs = obsSet.filter(!_.exists(_ > 0.01))
    if(zeroObs.size > 0)
        zeroObs.aggregate(0.0)((s, z) => s + dot(z, weights),
            _ + _) / zeroObs.size
    else 0.0
}
```

The `classify` methods takes new observations as inputs and compute the index of the classes (0 or 1) the observations belong to and the actual likelihood (line 20).

Selecting an optimizer

The goal of the training of a model using expected values is to compute the optimal weights that minimizes the **error or cost function**. We select the **batch gradient descent** algorithm to minimize the cumulative error between the predicted and expected values for all the observations. Although there are quite a few alternative optimizers, the gradient descent is quite robust and simple enough for this first chapter. The algorithm consists of updating the weights w_i of the regression model by minimizing the cost.

Cost function

M5: Cost (or *compound error* = *predicted* – *expected*):

$$\text{cost} = \frac{1}{2N} \sum_{j=0}^{N-1} (l(x_j|w_i) - y_j)^2$$



M6: The batch gradient descent method to update model weights w_i is as follows:

$$w'_i = w_i + \frac{\partial \text{cost}}{\partial w_i} = w_i + \frac{1}{N} \sum_{j=0}^{N-1} (l(x_j|w_i) - y_j) \cdot x_j$$

For those interested in learning about of optimization techniques, the *Summary of optimization techniques* section in the *Appendix A, Basic Concepts* presents an overview of the most commonly used optimizers. The batch descent gradient method is also used for the training of the multilayer perceptron (refer to *The training epoch* section under *The multilayer perceptron* section in *Chapter 9, Artificial Neural Networks*).

The execution of the batch gradient descent algorithm follows these steps:

1. Initialize the weights of the regression model.
2. Shuffle the order of observations and expected values.
3. Aggregate the cost or error for the entire observation set.
4. Update the model weights using the cost as the objective function.
5. Repeat from step 2 until either the maximum number of iterations is reached or the incremental update of the cost is close to zero.

The purpose of **shuffling** the order of the observations between iterations is to avoid the minimization of the cost reaching a local minimum.

Batch and stochastic gradient descent

The stochastic gradient descent is a variant of the gradient descent that updates the model weights after computing the error on each observation. Although the stochastic gradient descent requires a higher computation effort to process each observation, it converges toward the optimal value of weights fairly quickly after a small number of iterations. However, the stochastic gradient descent is sensitive to the initial value of the weights and the selection of the learning rate, which is usually defined by an adaptive formula.



Training the model

The `train` method consists of iterating through the computation of the weight using a simple descent gradient method. The method computes weights and returns an instance of the `LogBinRegressionModel` model:

```
def train: LogBinRegressionModel = {  
    val nWeights = obsSet.head.length + 1 //21  
    val init = Array.fill(nWeights) (Random.nextDouble) //22  
    val weights = gradientDescent(obsSet.zip(expected), 0.0, 0, init)  
    new LogBinRegressionModel(weights) //23  
}
```

The `train` method extracts the number of weights, `nWeights`, for the regression model as the *number of variables in each observation + 1* (line 21). The method initializes weights with random values over $[0, 1]$ (line 22). The weights are computed through the tail recursive `gradientDescent` method, and the method returns a new model for the binomial logistic regression (line 23).

Unwrapping values from Try

It is usually not recommended to invoke the `get` method to a `Try` value, unless it is enclosed in a `Try` statement. The best course of action is to do the following:

1. Catch the failure with `match{ case Success(m) => .. case Failure(e) => }`
2. Extract the `getOrElse(/* ... */)` result safely
3. Propagate the results as a `Try` type `map(_ .m)`

Let's take a look at the computation for weights through the minimization of the cost function in the `gradientDescent` method:

```
type LabelObs = Vector[(DblArray, Int)]  
  
@tailrec  
def gradientDescent(  
    obsAndLbl: LabelObs,  
    cost: Double,  
    nIters: Int,  
    weights: DblArray): DblArray = { //24
```

```

if(nIters >= maxIters)
    throw new IllegalStateException(..)//25
val shuffled = shuffle(obsAndLbl) //26
val errorGrad = shuffled.map{ case(x, y) => { //27
    val error = sigmoid(dot(x, weights)) - y
    (error, x.map(_ * error)) //28
} }.unzip

val scale = 0.5/obsAndLbl.size
val newCost = errorGrad._1 //29
.aggregate(0.0)((s,c) =>s + c*c, _ + _) *scale
val relativeError = cost/newCost - 1.0

if( Math.abs(relativeError) < eps) weights //30
else {
    val derivatives = Vector[Double](1.0) ++
        errorGrad._2.transpose.map(_.sum) //31
    val newWeights = weights.zip(derivatives)
        .map{ case (w, df) => w - eta*df} //32
    newWeights.toArray(weights)
    gradientDescent(shuffled, newCost, nIters+1, newWeights)//33
}
}
}

```

The `gradientDescent` method recurses on the vector of pairs (observations and expected values), `obsAndLbl`, `cost`, and the model weights (line 24). It throws an exception if the maximum number of iterations allowed for the optimization is reached (line 25). It shuffles the order of the observations (line 26) before computing the `errorGrad` derivatives of the cost over each weights (line 27). The computation of the derivative of the cost (or $error = predicted\ value - expected\ value$) in formula **M5** returns a pair of cumulative cost and derivative values using the formula (line 28).

Next, the method computes the overall compound cost using the formula **M4** (line 29), converts it to a relative incremental `relativeError` cost that is compared to the `eps` convergence criteria (line 30). The method extracts derivatives of cost over weights by transposing the matrix of errors, and then prepends the bias `1.0` value to match the array of weights (line 31).

Bias value

The purpose of the bias value is to prepend 1.0 to the vector of observation so it can be directly processed (for example, zip and dot) with the weights. For instance, a regression model for two-dimensional observations (x, y) has three weights (w_0, w_1, w_2). The bias value +1 is prepended to the observations to compute the predicted value 1.0: $w_0 + x.w_1 + y.w_2$.

This technique is used in the computation of the activation function of the multilayer perceptron, as described in the *The multilayer perceptron* section in *Chapter 9, Artificial Neural Networks*.



The formula **M6** updates the weights for the next iteration (line 32) before invoking the method with new weights, cost, and iteration count (line 33).

Let's take a look at the shuffling of the order of observations using a random sequence generator. The following implementation is an alternative to the Scala standard library method `scala.util.Random.shuffle` for shuffling elements of collections. The purpose is to change the order of observations and labels between iterations in order to prevent the optimizer to reach a local minimum. The `shuffle` method permutes the order in the `labelObs` vector of observations by partitioning it into segments of random size and reversing the order of the other segment:

```
val SPAN = 5
def shuffle(labelObs: LabelObs): LabelObs = {
    shuffle(new ArrayBuffer[Int](0, 0).map(labelObs(_))) //34
}
```

Once the order of the observations is updated, the vector of pair (observations, labels) is easily built through a map (line 34). The actual shuffling of the index is performed in the following `shuffle` recursive function:

```
val maxChunkSize = Random.nextInt(SPAN)+2 //35

@tailrec
def shuffle(indices: ArrayBuffer[Int], count: Int, start: Int):
    Array[Int] = {
    val end = start + Random.nextInt(maxChunkSize) //36
    val isOdd = ((count & 0x01) != 0x01)
    if(end >= sz)
        indices.toArray ++ slice(isOdd, start, sz) //37
    else
        shuffle(indices ++ slice(isOdd, start, end), count+1, end)
}
```

The maximum size of partition of the `maxChunkSize` vector observations is randomly computed (line 35). The method extracts the next slice (`start, end`) (line 36). The slice is either added to the existing indices vector and returned once all the observations have been shuffled (line 37) or passed to the next invocation.

The `slice` method returns an array of indices over the range (`start, end`) either in the right order if the number of segments processed is odd, or in reverse order if the number of segment processed is even:

```
def slice(isOdd: Boolean, start: Int, end: Int): Array[Int] = {
    val r = Range(start, end).toArray
    (if(isOdd) r else r.reverse)
}
```

Iterative versus tail recursive computation

The tail recursion in Scala is a very efficient alternative to the iterative algorithm. Tail recursion avoids the need to create a new stack frame for each invocation of the method. It is applied to the implementation of many machine learning algorithms presented throughout the book.

In order to train the model, we need to label the input data. The labeling process consists of associating the relative price movement during a session (price at *close*/*price at open - 1*) with one of the following two configurations:

- Volatile trading sessions with high trading volume
- Trading sessions with low volatility and low trading volume

The two classes of training observations are segregated by a decision boundary drawn on the scatter plot in the previous section. The labeling process is usually quite cumbersome and should be automated as much as possible.

Automated labeling

Although quite convenient, automated creation of training labels is not without risk as it may mislabel singular observations. This technique is used in this test for convenience, but it is not recommended unless a domain expert reviews the labels manually.

Classifying observations

Once the model is successfully created through training, it is available to classify new observation. The runtime classification of observations using the binomial logistic regression is implemented by the `classify` method:

```
def classify(obs: DblArray): Try[(Int, Double)] =  
    val linear = dot(obs, model.weights) //37  
    val prediction = sigmoid(linear)  
    (if(linear > 0.0) 1 else 0, prediction) //38  
}
```

The method applies the logistic function to the linear inner product, `linear`, of the new `obs` and `weights` observations of the model (line 37). The method returns the tuple (the predicted class of the observation {0, 1}, prediction value) where the class is defined by comparing the prediction to the boundary value 0.0 (line 38).

The computation of the `dot` product of weights and observations uses the bias value as follows:

```
def dot(obs: DblArray, weights: DblArray): Double =  
    weights.zip(Array[Double](1.0) ++ obs)  
    .aggregate(0.0){case (s, (w,x)) => s + w*x, _ + _ }
```

The alternative implementation of the `dot` product of weights and observations consists of extracting the first `w.head` weight:

```
def dot(x: DblArray, w: DblArray): Double =  
    x.zip(w.drop(1)).map {case (_x,_w) => _x*_w}.sum + w.head
```

The `dot` method is used in the `classify` method.

Step 6 – evaluating the model

The first step is to define the configuration parameters for the test: the maximum number of `NITERS` iterations, the `EPS` convergence criteria, the `ETA` learning rate, the decision boundary used to label the `BOUNDARY` training observations, and the path to the training and test sets:

```
val NITERS = 800; val EPS = 0.02; val ETA = 0.0001  
val path_training = "resources/data/chap1/CSCO.csv"  
val path_test = "resources/data/chap1/CSCO2.csv"
```

The various activities of creating and testing the model, loading, normalizing data, training the model, loading, and classifying test data is organized as a workflow using the monadic composition of the Try class:

```

for {
    volatilityVol <- load(path_training)      //39
    minMaxVec <- Try(new MinMaxVector(volatilityVol))    //40
    normVolatilityVol <- Try(minMaxVec.normalize(0.0,1.0))//41
    classifier <- logRegr(normVolatilityVol)      //42
    testValues <- load(path_test)      //43
    normTestValue0 <- minMaxVec.normalize(testValues(0))  //44
    class0 <- classifier.classify(normTestValue0)      //45
    normTestValue1 <- minMaxVec.normalize(testValues(1))
    class1 <- classifier.classify(normTestValue1)
} yield {
    val modelStr = model.toString
    ...
}

```

First, the daily trading volatility and volume for the `volatilityVol` stock price is loaded from file (line 39). The workflow initializes the multi-dimensional `MinMaxVec` normalizer (line 40) and uses it to normalize the training set (line 41). The `logRegr` method instantiates the binomial `classifier` logistic regression (line 42). The `testValues` test data is loaded from file (line 43), normalized using `MinMaxVec` already applied to the training data (line 44), and classified (line 45).

The `load` method extracts data (observations) of a `XVSeries[Double]` type from the file. The heavy lifting is done by the `extract` method (line 46), and then the file handle is closed (line 47) before returning the vector of raw observations:

```

def load(fileName: String): Try[XVSeries[Double], XSeries[Double]] = {
    val src = Source.fromFile(fileName)
    val data = extract(src.getLines.map(_.split(",")).drop(1)) //46
    src.close; data //47
}

```

The private `logRegr` method has the following two purposes:

- Labeling automatically the `obs` observations to generate the expected values (line 48)
- Initializing (instantiation and training of the model) the binomial logistic regression (line 49)

The code is as follows:

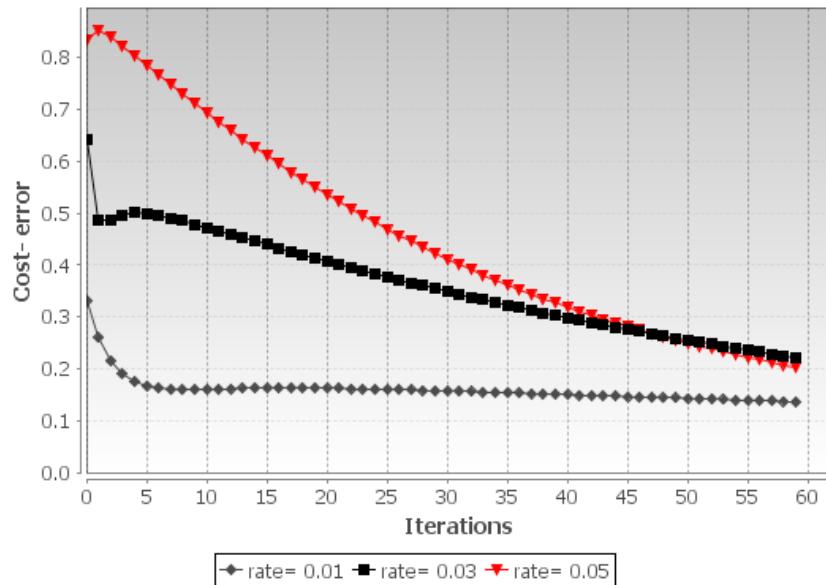
```
def logRegr(obs: XVSeries[Double]): Try[LogBinRegression] = Try {  
    val expected = normalize(labels._2).get //48  
    new LogBinRegression(obs, expected, NITERS, ETA, EPS) //49  
}
```

The method labels observations by evaluating if they belong to any one of the two classes delimited by the BOUNDARY condition, as illustrated in the scatter plot in a previous section.

Validation

The simple classification in this test case is provided for illustrating the runtime application of the model. It does not constitute a validation of the model by any stretch of imagination. The next chapter digs into validation methodologies (refer to the *Assessing a model* section in *Chapter 2, Hello World!*)

The training run is performed with three different values of the learning rate. The following chart illustrates the convergence of the batch gradient descent in the minimization of the cost, given different values of learning rates:



Impact of the learning rate on the batch gradient descent on the convergence of the cost (error)

As expected, the execution of the optimizer with a higher learning rate produces a steepest descent in the cost function.

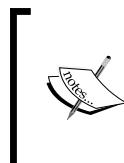
The execution of the test produces the following model:

iters = 495

weights: 0.859-3.6177923,-64.927832

input (0.0088, 4.10E7) normalized (0.063,0.061) class 1 prediction 0.515

input (0.0694, 3.68E8) normalized (0.517,0.641) class 0 prediction 0.001



Learning more about regressive models

The binomial logistic regression is merely used to illustrate the concept of training and prediction. It is described in the *Logistic regression* section in *Chapter 6, Regression and Regularization* in detail.

Summary

I hope you enjoyed this introduction to machine learning. You learned how to leverage your skills in Scala programming to create a simple logistic regression program for predicting stock price/volume action. Here are the highlights of this introductory chapter:

- From monadic composition and high order collection methods for parallelization to configurability and reusability patterns, Scala is the perfect fit to implement data mining and machine learning algorithms for large-scale projects.
- There are many logical steps to create and deploy a machine learning model.
- The implementation of the binomial logistic regression classifier presented as part of the test case is simple enough to encourage you to learn how to write and apply more advanced machine learning algorithms.

To the delight of Scala programming aficionados, the next chapter will dig deeper into building a flexible workflow by leveraging monadic data transformation and stackable traits.

2

Hello World!

In the first chapter, you were acquainted with some rudimentary concepts regarding data processing, clustering, and classification. This chapter is dedicated to the creation and maintenance of a flexible end-to-end workflow to train and classify data. The first section of the chapter introduces a data-centric (functional) approach to create number-crunching applications.

You will learn how to:

- Apply the concept of monadic design to create dynamic workflows
- Leverage some of Scala's advanced patterns, such as the cake pattern, to build portable computational workflows
- Take into account the bias-variance trade-off in selecting a model
- Overcome overfitting in modeling
- Break down data into training, test, and validation sets
- Implement model validation in Scala using precision, recall, and F score

Modeling

Data is the lifeline of any scientist, and the selection of data providers is critical in developing or evaluating any statistical inference or machine learning algorithm.

A model by any other name

We briefly introduced the concept of a **model** in the *Model categorization* section in *Chapter 1, Getting Started*.

What constitutes a model? Wikipedia provides a reasonably good definition of a model as understood by scientists [2:1]:

A scientific model seeks to represent empirical objects, phenomena, and physical processes in a logical and objective way.

...

Models that are rendered in software allow scientists to leverage computational power to simulate, visualize, manipulate and gain intuition about the entity, phenomenon or process being represented.

In statistics and the probabilistic theory, a model describes data that one might observe from a system to express any form of uncertainty and noise. A model allows us to infer rules, make predictions, and learn from data.

A model is composed of **features**, also known as **attributes** or **variables**, and a set of relation between those features. For instance, the model represented by the function $f(x, y) = x \cdot \sin(2y)$ has two features, x and y , and a relation, f . Those two features are assumed to be independent. If the model is subject to a constraint such as $f(x, y) < 20$, then the **conditional independence** is no longer valid.

An astute Scala programmer would associate a model to a monoid for which the set is a group of observations and the operator is the function implementing the model.

Models come in a variety of shapes and forms:

- **Parametric:** This consists of functions and equations (for example, $y = \sin(2t + w)$)
- **Differential:** This consists of ordinary and partial differential equations (for example, $dy = 2x.dx$)
- **Probabilistic:** This consists of probability distributions (for example, $p(x | c) = \exp(k \cdot \log x - x)/x!$)
- **Graphical:** This consists of graphs that abstract out the conditional independence between variables (for example, $p(x, y | c) = p(x | c).p(y | c)$)
- **Directed graphs:** This consists of temporal and spatial relationships (for example, a scheduler)
- **Numerical method:** This consists of computational methods such as finite difference, finite elements, or Newton-Raphson
- **Chemistry:** This consists of formula and components (for example, H_2O , $Fe + C_{12} = FeC_{13}$, and so on)

- **Taxonomy:** This consists of a semantic definition and relationship of concepts (for example, *APG/Eudicots/Rosids/Huaceae/Malvales*)
- **Grammar and lexicon:** This consists of a syntactic representation of documents (for example, the Scala programming language)
- **Inference logic:** This consists of rules (for example, *IF (stock vol > 1.5 * average) AND rsi > 80 THEN ...*)

Model versus design

The confusion between a model and design is quite common in computer science, the reason being that these terms have different meanings for different people depending on the subject. The following metaphors should help with your understanding of these two concepts:

- **Modeling:** This describes something you know. A model makes an assumption, which becomes an assertion if proven correct (for example, the US population, p , increases by 1.2 percent a year, $dp/dt = 1.012$).
- **Designing:** This manipulates the representation of things you don't know. Designing can be regarded as the exploration phase of modeling (for example, what are the features that contribute to the growth of the US population? Birth rate? Immigration? Economic conditions? Social policies?).

Selecting features

The selection of a model's features is the process of discovering and documenting the minimum set of variables required to build the model. Scientists assume that data contains many redundant or irrelevant features. Redundant features do not provide information already given by the selected features, and irrelevant features provide no useful information.

A **features selection** consists of two consecutive steps:

1. Searching for new feature subsets.
2. Evaluating these feature subsets using a scoring mechanism.

The process of evaluating each possible subset of features to find the one that maximizes the objective function or minimizes the error rate is computationally intractable for large datasets. A model with n features requires $2^n - 1$ evaluations.

Extracting features

An **observation** is a set of indirect measurements of hidden, also known as latent, variables, which may be noisy or contain a high degree of correlation and redundancies. Using raw observations in a classification task would very likely produce inaccurate results. Using all features in each observation also incurs a high computation cost.

The purpose of **features extraction** is to reduce the number of variables or dimensions of the model by eliminating redundant or irrelevant features. The features are extracted by transforming the original set of observations into a smaller set at the risk of losing some vital information embedded in the original set.

Defining a methodology

A data scientist has many options in selecting and implementing a classification or clustering algorithm.

Firstly, a mathematical or statistical model is to be selected to extract knowledge from the raw input data or the output of a data upstream transformation. The selection of the model is constrained by the following parameters:

- Business requirements such as accuracy of results or computation time
- Availability of training data, algorithms, and libraries
- Access to a domain or subject matter expert, if needed

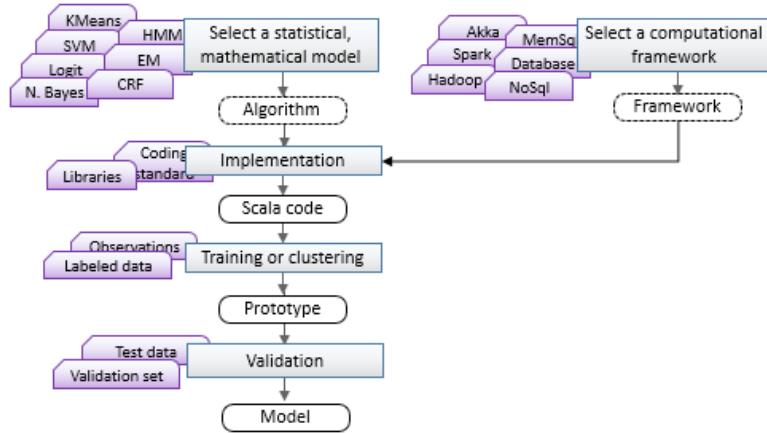
Secondly, the engineer has to select a computational and deployment framework suitable for the amount of data to be processed. The computational context is to be defined by the following parameters:

- Available resources such as machines, CPU, memory, or I/O bandwidth
- An implementation strategy such as iterative versus recursive computation or caching
- Requirements for the responsiveness of the overall process such as duration of computation or display of intermediate results

Thirdly, a domain expert has to tag or label the observations in order to generate an accurate classifier.

Finally, the model has to be validated against a reliable test dataset.

The following diagram illustrates the selection process to create a workflow:



Statistical and computation modeling for machine learning applications

Domain expertise, data science, and software engineering

A domain or subject matter expert is a person with authoritative or credited expertise in a particular area or topic. A chemist is an expert in the domain of chemistry and possibly related fields.

A data scientist solves problems related to data in a variety of fields, such as biological sciences, health care, marketing, or finances. Data and text mining, signal processing, statistical analysis, and modeling using machine learning algorithms are some of the activities performed by a data scientist.

A software developer performs all the tasks related to the creation of software applications, including analysis, design, coding, testing, and deployment.



The parameters of a data transformation may need to be reconfigured according to the output of the upstream data transformation. Scala's higher-order functions are particularly suitable for implementing configurable data transformations.

Monadic data transformation

The first step is to define a trait and method that describe the transformation of data by the computation units of a workflow. The data transformation is the foundation of any workflow for processing and classifying a dataset, training and validating a model, and displaying results.

There are two symbolic models used for defining a data processing or data transformation:

- **Explicit model:** The developer creates a model explicitly from a set of configuration parameters. Most of deterministic algorithms and unsupervised learning techniques use an explicit model.
- **Implicit model:** The developer provides a training set that is a set of labeled observations (observations with an expected outcome). A classifier extracts a model through the training set. Supervised learning techniques rely on models implicitly generated from labeled data.

Error handling

The simplest form of data transformation is **morphism** between the two `U` and `V` types. The data transformation enforces a *contract* for validating an input and returning either a value or an error. From now on, we use the following convention:

- **Input value:** The validation is implemented through a partial function of the `PartialFunction` type that is returned by the data transformation. A `MatchErr` error is thrown in case the input value does not meet the required condition (contract).
- **Output value:** The type of a return value is `Try[V]` for which an exception is returned in case of an error.

Reusability of partial functions

Reusability is another benefit of partial functions, which is illustrated in the following code snippet:



```
class F {  
    def f: PartialFunction[Int, Try[Double]] = case n:  
        Int ...  
    }  
    val pfn = (new F).f  
    pfn(4)  
    pfn(10)
```

Partial functions enable developers to implement methods that address the most common (primary) use cases for which input values have been tested. All other nontrivial use cases (or input values) generate a `MatchErr` exception. At a later stage in the development cycle, the developer can implement the code to handle the less common use cases.

Runtime validation of a partial function

It is a good practice to validate if a partial function is defined for a specific value of the argument:

```

for {
    pfn.isDefinedAt(input)
    value <- pfn(input)
} yield { ... }
```

This preemptive approach allows the developer to select an alternative method or a full function. It is an efficient alternative to catch a `MathErr` exception. The validation of a partial function is omitted throughout the book for the sake of clarity.

Therefore, the signature of a data transformation is defined as follows:

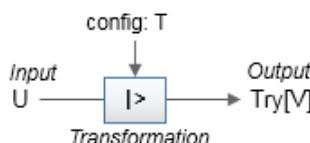
```
def |> : PartialFunction[U, Try[V]]
```

F# language references

The `|>` notation used as the signature of the transform is borrowed from the F# language [2:2].

Explicit models

The objective is to define a symbolic representation of the transformation of different types of data without exposing the internal state of the algorithm implementing the data transformation. The transformation on a dataset is performed using a model or configuration that is fully defined by the user, which is illustrated in the following diagram:



Visualization of explicit models

The transformation of an explicit configuration or model, `config`, is defined as an `ETransform` abstract class parameterized by the `T` type of the model:

```
abstract class ETransform[T] (val config: T) { //explicit model
    type U    // type of input
    type V    // type of output
    def |> : PartialFunction[U, Try[V]]    // data transformation
}
```

The input U type and output V type have to be defined in the subclasses of `ETransform`. The `|>` transform operator returns a partial function that can be reused for different input values.

The creation of a class that implements a specific transformation using an explicit configuration is quite simple: all you need is the definition of an input/output U/V type and an implementation of the `|>` transformation method.

Let's consider the extraction of data from a financial source, `DataSource`, that takes a list of functions that convert some text fields, `Fields`, into a `Double` value as the input and produce a list of observations of the `XSeries[Double]` type. The extraction parameters are defined in the `DataSourceConfig` class:

```
class DataSource(
    config: DataSourceConfig,      //1
    srcFilter: Option[Fields => Boolean] = None)
    extends ETransform[DataSourceConfig](config) { //2
    type U = List[Fields => Double]      //3
    type V = List[XSeries[Double]]        //4
    override def |> : PartialFunction[U, Try[V]] = { //5
        case u: U if (!u.isEmpty) => ...
    }
}
```

The `DataSourceConfig` configuration is explicitly provided as an argument of the constructor for `DataSource` (line 1). The constructor implements the basic type and data transformation associated with an explicit model (line 2). The class defines the U type of input values (line 3), V type of output values (line 4), and `|>` transformation method that returns a partial function (line 5).



The `DataSource` class

The *Data extraction* section of the *Appendix A, Basic Concepts* describes the `DataSource` class functionality. The `DataSource` class is used throughout the book.

Data transformations using an explicit model or configuration constitute a category with monadic operations. The monad associated with the `ETransform` class subclasses the definition of the higher kind, `_Monad`:

```
private val eTransformMonad = new _Monad[ETransform] {
    override def unit[T](t:T) = eTransform(t)      //6
    override def map[T,U](m: ETransform[T])         //7
        (f: T => U): ETransform[U] = eTransform( f(m.config) )
    override def flatMap[T,U](m: ETransform[T])     //8
        (f: T => ETransform[U]): ETransform[U] = f(m.config)
}
```

The singleton `eTransformMonad` implements the following basic monadic operators introduced in the *Monads* section under *Abstraction in Chapter 1, Getting Started*:

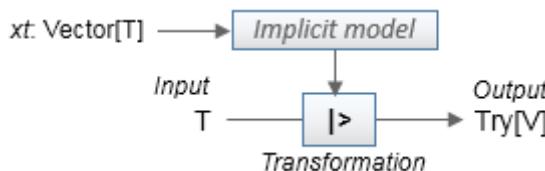
- The `unit` method is used to instantiate `ETransform` (line 6)
- The `map` is used to transform an `ETransform` object by morphing its elements (line 7)
- The `flatMap` is used to transform an `ETransform` object by instantiating its elements (line 8)

For practical purposes, an implicit class is created to convert an `ETransform` object to its associated monad, allowing transparent access to the `unit`, `map`, and `flatMap` methods:

```
implicit class eTransform2Monad[T] (fct: ETransform[T]) {
    def unit(t: T) = eTransformMonad.unit(t)
    final def map[U](f: T => U): ETransform[U] =
        eTransformMonad.map(fct)(f)
    final def flatMap[U](f: T => ETransform[U]): ETransform[U] =
        eTransformMonad.flatMap(fct)(f)
}
```

Implicit models

Supervised learning models are extracted from a training set. Transformations, such as classification or regression use the implicit models to process the input data, as illustrated in the following diagram:



Visualization of implicit models

The transformation for a model implicitly extracted from the training data is defined as an abstract `ITransform` class parameterized by the `T` type of observations, `xt`:

```
abstract class ITransform[T] (val xt: Vector[T]) { //Model input
    type V    // type of output
    def |> : PartialFunction[T, Try[V]] // data transformation
}
```

The type of the data collection is `Vector`, which is an immutable and effective container. An `ITransform` type is created by defining the `T` type of the observation, the `v` output of the data transformation, and the `|>` method that implements the transformation, usually a classification or regression. Let's consider the support vector machine algorithm, `SVM`, to illustrate the implementation of a data transformation using an implicit model:

```
class SVM[T <: AnyVal] ( //9
    config: SVMConfig,
    xt: Vector[Array[T]],
    expected: Vector[Double]) (implicit f: T => Double)
  extends ITransform[Array[T]] (xt) { //10

  type V = Double //11
  override def |> : PartialFunction[Array[T], Try[V]] = { //12
    case x: Array[T] if(x.length == data.size) => ...
  }
}
```

The support vector machine is a discriminative supervised learning algorithm described in *Chapter 8, Kernel Models and Support Vector Machines*. A support vector machine, `SVM`, is instantiated with a configuration and training set: the `xt` observations and `expected` data (line 9). Contrary to the explicit model, the `config` configuration does not define the model used in the data transformation; the model is implicitly generated from the training set of the `xt` input data and `expected` values. An `SVM` instance is created as an `ITransform` (line 10) by specifying the `V` output type (line 11) and overriding the `|>` transformation method (line 12).

The `|>` classification method produces a partial function that takes an `x` observation as an input and returns the prediction value of a `Double` type.

Similar to the explicit transformation, we define the monadic operation for the `ITransform` by overriding the `unit` (line 13), `map` (line 14), and `flatMap` (line 15) methods:

```
private val iTransformMonad = new _Monad[ITransform] {
  override def unit[T](t: T) = iTransform(Vector[T](t)) //13

  override def map[T,U](m: ITransform[T])(f: T => U):
    ITransform[U] = iTransform(m.xt.map(f)) //14

  override def flatMap[T,U](m: ITransform[T])
    (f: T=>ITransform[U]): ITransform[U] =
    iTransform(m.xt.flatMap(t => f(t).xt)) //15
}
```

Finally, let's create an implicit class to automatically convert an `ITransform` object into its associated monad so that it can access the `unit`, `map`, and `flatMap` monad methods transparently:

```
implicit class iTransform2Monad[T] (fct: ITransform[T]) {
    def unit(t: T) = iTransformMonad.unit(t)

    final def map[U](f: T => U): ITransform[U] =
        iTransformMonad.map(fct)(f)
    final def flatMap[U](f: T => ITransform[U]): ITransform[U] =
        iTransformMonad.flatMap(fct)(f)
    def filter(p: T => Boolean): ITransform[T] = //16
        iTransform(fct.xt.filter(p))
}
```

The `filter` method is strictly not an operator of the monad (line 16). However, it is commonly included to constrain (or guard) a sequence of transformation (for example, for comprehension closure). As stated in the *Presentation* section under *Source code in Chapter 1, Getting Started*, code related to exceptions, error checking, and validation of arguments is omitted.

Immutable transformations

The model for a data transformation (or a processing unit or classifier) class should be immutable. Any modification will alter the integrity of the model or parameters used to process data. In order to ensure that the same model is used in processing the input data for the entire lifetime of a transformation, we do the following:

- A model for an `ETransform` is defined as an argument of its constructor.
- The constructor of an `ITransform` generates the model from a given training set. The model has to be rebuilt from the training set (not altered), if it provides an incorrect outcome or prediction.

Models are created by the constructor of classifiers or data transformation classes to ensure their immutability. The design of an immutable transformation is described in the *Design template for immutable classifiers* section under *Scala programming* of the *Appendix A, Basic Concepts*.

A workflow computational model

Monads are very useful for manipulating and chaining data transformations using implicit configurations or explicit models. However, they are restricted to a single morphism $T \Rightarrow U$ type. More complex and flexible workflows require weaving transformations of different types using a generic factory pattern.

Traditional factory patterns rely on a combination of composition and inheritance and do not provide developers with the same level of flexibility as stackable traits.

In this section, we introduce you to the concept of modeling using mixins and a variant of the cake pattern to provide a workflow with three degrees of configurability.

Supporting mathematical abstractions

Stackable traits enable developers to follow a strict mathematical formalism while implementing a model in Scala. Scientists use a universally accepted template to solve a mathematical problem:

1. Declare the variables relevant to the problem.
2. Define a model (equations, algorithms, formulas, and so on) as the solution to the problem.
3. Instantiate the variables and execute the model to solve the problem.

Let's consider the example of the concept of kernel functions (described in the *Kernel functions* section in *Chapter 8, Kernel Models and Support Vector Machines*), a model that consists of a composition of two mathematical functions and its potential implementation in Scala.

Step 1 – variable declaration

The implementation consists of wrapping (scope) the two functions into traits and defining these functions as abstract values.

The mathematical formalism is as follows:

$$f: \mathbb{R}^n \rightarrow \mathbb{R}^n \quad g: \mathbb{R}^n \rightarrow \mathbb{R}$$

The Scala implementation is as follows:

```
type V = Vector[Double]
trait F { val f: V => V}
trait G { val g: V => Double }
```

Step 2 – model definition

The model is defined as the composition of the two functions. The `G` and `F` stack of traits describe the type of compatible functions that can be composed using the self-referenced `self: G with F` constraint.

The formalism will be $h = f \circ g$.

The Scala implementation is as follows:

```
class H {self: G with F => def apply(v:V): Double =g(f(v))}
```

Step 3 – instantiation

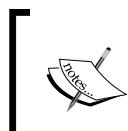
The model is executed once the f and g variables are instantiated.

The formalism will be as follows:

$$f: x \rightarrow e^x \quad g: x \rightarrow \sum_0^{n-1} x_i$$

The Scala implementation is as follows:

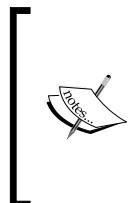
```
val h = new H with G with F {
    val f: V => V = (v: V) => v.map(Math.exp(_))
    val g: V => Double = (v: V) => v.sum
}
```



Lazy value triggers

In the preceding example, the value of $h(v) = g(f(v))$ can be automatically computed as soon as g and f are initialized, by declaring h a lazy value.

Clearly, Scala preserves the formalism of mathematical models, making it easier for scientists and developers to migrate their existing projects written in scientific-oriented languages, such as R.



Emulation of R

Most data scientists use the R language to create models and apply learning strategies. They may consider Scala as an alternative to R in some cases, as Scala preserves the mathematical formalism used in models implemented in R.

Let's extend the concept preservation of mathematical formalism to the dynamic creation of workflows using traits. The design pattern described in the next section is sometimes referred to as the **Cake pattern**.

Composing mixins to build a workflow

This section presents the key constructs behind the Cake pattern. A workflow composed of configurable data transformations requires a dynamic modularization (substitution) of the different stages of the workflow.

Traits and mixins



Mixins are traits that are stacked against a class. The composition of mixins and the Cake pattern described in this section are important for defining the sequences of data transformations. However, the topic is not directly related to machine learning and so you can skip this section.

The Cake pattern is an advanced class composition pattern that uses mixin traits to meet the demands of a configurable computation workflow. It is also known as stackable modification traits [2:4].

This is not an in-depth analysis of the **stackable trait injection** and **self-reference** in Scala. There are few interesting articles on dependencies injection that are worth a look [2:5].

Java relies on packages tightly coupled with the directory structure and prefixed to modularize the code base. Scala provides developers with a flexible and reusable approach to create and organize modules: traits. Traits can be nested, mixed with classes, stacked, and inherited.

Understanding the problem

Dependency injection is a fancy name for a reverse look-up and binding to dependencies. Let's consider a simple application that requires data preprocessing, classification, and validation. A simple implementation using traits looks like this:

```
val app = new Classification with Validation with PreProcessing {  
    val filter = ...  
}
```

If, at a later stage, you need to use an unsupervised clustering algorithm instead of a classifier, then the application has to be rewired:

```
val app = new Clustering with Validation with PreProcessing {  
    val filter = ...  
}
```

This approach results in code duplication and lack of flexibility. Moreover, the `filter` class member needs to be redefined for each new class in the composition of the application. The problem arises when there is a dependency between traits used in the composition. Let's consider the case for which the `filter` depends on the `validation` methodology.

Mixins linearization [2:6]

The linearization or invocation of methods between mixins follows a right-to-left and base-to-subtype pattern:

- Trait `B` extends `A`
- Trait `C` extends `A`
- Class `M` extends `N` with `C` with `B`

The Scala compiler implements the linearization as $A \Rightarrow B \Rightarrow C \Rightarrow N$.

Although you can define `filter` as an abstract value, it still has to be redefined each time a new validation type is introduced. The solution is to use the `self` type in the definition of the newly composed `PreProcessingWithValidation` trait:

```
trait PreProcessingWithValidation extends PreProcessing {
    self: Validation => val filter = ...
}
```

The application is built by stacking the `PreProcessingWithValidation` mixin against the `Classification` class:

```
val app = new Classification with PreProcessingWithValidation {
    val validation: Validation
}
```

Overriding def with val

It is advantageous to override the declaration of a method with a declaration of a value with the same signature. Contrary to a value that is assigned once for all during instantiation, a method may return a different value for each invocation. A `def` is a `proc` that can be redefined as a `def`, `val`, or `lazy val`. Therefore, you should not override a value declaration with a method with the same signature:

```
trait Validator { val g = (n: Int) => ... }
trait MyValidator extends Validator { def g(n: Int) =
... } //WRONG
```

Let's adapt and generalize this pattern to construct a boilerplate template in order to create dynamic computational workflows.

Defining modules

The first step is to generate different modules to encapsulate different types of data transformation.

Use case for describing the cake pattern

 It is difficult to build an example of a real-world workflow using classes and algorithms introduced later in the book. The following simple example is realistic enough to illustrate the different components of the Cake pattern.

Let's define a sequence of the three parameterized modules that each define a specific data transformation using the explicit configuration of the `Etransform` type:

- Sampling: This is used to extract a sample from raw data
- Normalization: This is used to normalize the sampled data over [0, 1]
- Aggregation: This is used to aggregate or reduce the data

The code will be as follows:

```
trait Sampling[T,A,B] {  
    val sampler: ETransform[T] { type U = A; type V = B }  
}  
trait Normalization[T,A,B] {  
    val normalizer: ETransform[T] { type U = A; type V = B }  
}  
trait Aggregation[T,A,B] {  
    val aggregator: ETransform[T] { type U = A; type V = B }  
}
```

The modules contain a single abstract value. One characteristic of the Cake pattern is to enforce strict modularity by initializing the abstract values with the type encapsulated in the module. One of the objectives in building the framework is allowing developers to create data transformation (inherited from `ETransform`) independently from any workflow.

Scala traits and Java packages

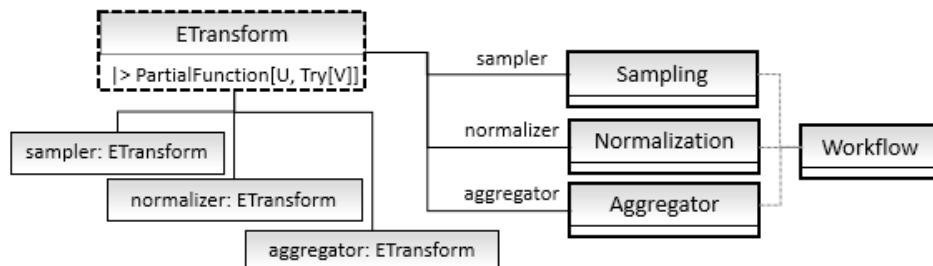
 There is a major difference between Scala and Java in terms of modularity. Java packages constrain developers into following a strict syntax that requires, for instance, the source file to have the same name as the class it contains. Scala modules based on stackable traits are far more flexible.

Instantiating the workflow

The next step is to *write* the different modules into a workflow. This is achieved by using the `self` reference to the stack of the three traits defined in the previous section:

```
class Workflow[T,U,V,W,Z] {
    self: Sampling[T,U,V] with
        Normalization[T,V,W] with
            Aggregation[T,W,Z] =>
    def |> (u: U): Try[Z] = for {
        v <- sampler |> u
        w <- normalizer |> v
        z <- aggregator |> w
    } yield z
}
```

A picture is worth a thousand words; the following UML class diagram illustrates the workflow factory (or Cake) design pattern:



The UML class diagram of the workflow factory

Finally, the workflow is instantiated by dynamically initializing the `sampler`, `normalizer`, and `aggregator` abstract values of the transformation as long as the signature (input and output types) matches the parameterized types defined in each module (line 1):

```
type Dbl_F = Function1[Double, Double]
val samples = 100; val normRatio = 10; val splits = 4

val workflow = new Workflow[Int, Dbl_F, DblVector, DblVector, Int]
  with Sampling[Int, Dbl_F, DblVector]
    with Normalization[Int, DblVector, DblVector]
      with Aggregation[Int, DblVector, Int] {
  val sampler = new ETransform[Int](samples) { /* .. */ } //1
  val normalizer = new ETransform[Int](normRatio) { /* .. */ }
  val aggregator = new ETransform[Int](splits) {/* .. */ }
}
```

Let's implement the data transformation function for each of the three modules/traits by assigning a transformation to the abstract values.

The first transformation, `sampler`, samples a `f` function with frequency as $1/samples$ over the interval $[0, 1]$. The second transformation, `normalizer`, normalizes the data over the range $[0, 1]$ using the `Stats` class introduced in the next chapter. The last transformation, `aggregator`, extracts the index of the large sample (value 1.0):

```
val sampler = new ETransform[Int](samples) { //2
  type U = Dbl_F //3
  type V = DblVector //4
  override def |> : PartialFunction[U, Try[V]] = {
    case f: U =>
      Try(Vector.tabulate(samples)(n => f(1.0*n/samples))) //5
  }
}
```

The `sampler` transformation uses a single model or configuration parameter, `sample`, (line 2). The `U` type of an input is defined as `Double => Double` (line 3) and the `V` type of an output is defined as a vector of floating point values, `DblVector` (line 4). In this particular case, the transformation consists of applying the input `f` function to a vector of increasing normalized values (line 5).

The `normalizer` and `aggregator` transforms follow the same design pattern as `sampler`:

```
val normalizer = new ETransform[Int](normRatio) {
  type U = DblVector; type V = DblVector
  override def |> : PartialFunction[U, Try[V]] = { case x: U
```

```

        if(x.size >0) => Try((Stats[Double](x)).normalize)
    }
}
val aggregator = new ETransform[Int](splits) {
    type U = DblVector; type V = Int
    override def |> : PartialFunction[U, Try[V]] = case x: U
        if(x.size > 0) => Try(Range(0,x.size).find(x(_)==1.0).get)
    }
}

```

The instantiation of the transformation function follows the template described in the *Explicit models* section in this chapter.

The workflow is now ready to process any function as an input:

```

val g = (x: Double) => Math.log(x+1.0) + Random.nextDouble
Try( workflow |> g ) //6

```

The workflow is executed by providing the input `g` function to the first `sampler` mixin (line 6).

Scala's strong type checking catches any inconsistent data types at compilation time. It reduces the development cycle because runtime errors are more difficult to track down.

Mixins composition for `ITransform`

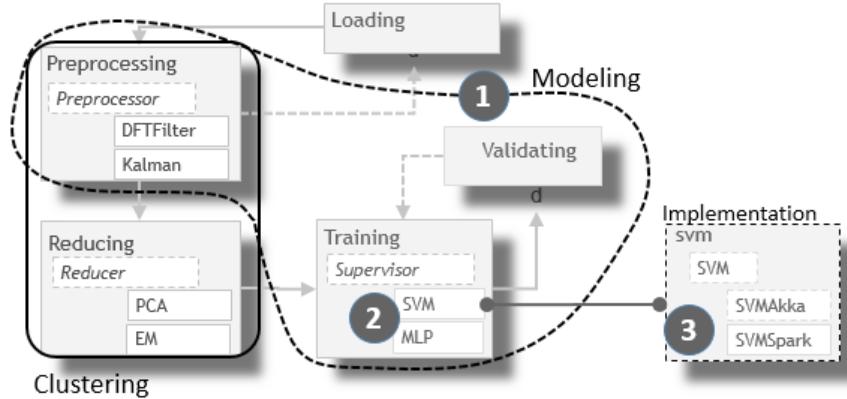
We arbitrary selected a data transformation using an explicit `ETransform` configuration to illustrate the concept of mixins composition. The same pattern applies to the implicit `ITransform` data transformation.

Modularization

The last step is the modularization of the workflow. For complex scientific computations, you need to be able to do the following:

1. Select the appropriate *workflow* as a sequence of modules or tasks according to the objective of the execution (regression, classification, clustering, and so on).
2. Select the appropriate *algorithm* to fulfill a task according to the data (noisy data, an incomplete training set, and so on).

3. Select the appropriate *implementation* of the algorithm according to the environment (distributed with a high-latency network, single host, and so on).



An Illustration of the dynamic creation of a workflow from modules/traits

Let's consider a simple preprocessing task defined in the `PreprocessingModule` module. The module (or task) is declared as a trait to hide its internal workings from other modules. The preprocessing task is executed by a preprocessor of a `Preprocessor` type. We arbitrary list two algorithms: the exponential moving average of the `ExpMovingAverage` type and the discrete Fourier transform low pass filter of the `DFTFilter` type as a potential preprocessor:

```
trait PreprocessingModule[T] {
    trait Preprocessor[T] { //7
        def execute(x: Vector[T]): Try[DblVector]
    }
    val preprocessor: Preprocessor[T] //8

    class ExpMovingAverage[T <: AnyVal] ( //9
        p: Int)
        (implicit num: Numeric[T], f: T => Double)
    extends Preprocessor[T] {

        val expMovingAvg = filtering.ExpMovingAverage[T](p) //10
        val pfn = expMovingAvg |> //11
```

```

override def execute(x: Vector[T]): Try[DblVector] =
  pfn(x).map(_.toVector)
}

class DFTFilter[T <: AnyVal] (
  fc: Double)
  (g: (Double,Double) =>Double)
  (implicit f : T => Double)
extends Preprocessor[T] { //12

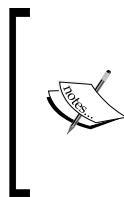
  val filter = filtering.DFTFir[T](g, fc, 1e-5)
  val pfn = filter |>
    override def execute(x: Vector[T]): Try[DblVector] =
      pfn(x).map(_.toVector)
}
}

```

The generic preprocessor trait, `Preprocessor`, declares a single `execute` method whose purpose is to filter an `x` input vector of an element of a `T` type for noise (line 7). The instance of the preprocessor is declared as an abstract class to be instantiated as one of the filtering algorithms (line 8).

The first filtering algorithm of an `ExpMovingAverage` type implements the `Preprocessor` trait and overrides the `execute` method (line 9). The class declares the algorithm but delegates its implementation to a class with an identical `org.scalaml.filtering.ExpMovingAverage` signature (line 10). The partial function returned from the `|>` method is instantiated as a `pfn` value, so it can be applied multiple times (line 11). The same design pattern is used for the discrete Fourier transform filter (line 12).

The filtering algorithm (`ExpMovingAverage` or `DFTFir`) is selected according to the profile or characteristic of the input data. Its implementation in the `org.scalaml.filtering` package depends on the environment (a single host, cluster, Apache spark, and so on).



Filtering algorithms

The filtering algorithms used to illustrate the concept of modularization in the context of the Cake pattern are described in detail in *Chapter 3, Data Preprocessing*.

Profiling data

The selection of a preprocessing, clustering, or classification algorithm depends highly on the quality and profile of the input data (observations and expected values whenever available). The *Step 3 - preprocessing the data* section under *Let's kick the tires in Chapter 1, Getting Started*, introduced the `MinMax` class for normalizing a dataset using the minimum and maximum values.

Immutable statistics

The mean and standard deviation are the most commonly used statistics.

Mean and variance

Arithmetic mean is defined as:

$$E[X] = \mu = \frac{1}{n} \sum_{i=0}^{n-1} x_i$$

Variance is defined as:


$$\text{Var}(X) = \frac{\sum(E(X) - x_j)^2}{n - 1}$$

Variance adjusted for a sampling bias is defined as:

$$\widehat{\text{Var}}(X) = \frac{1}{n - 1} \sum_{i=1}^n (x_i - E[X])^2$$

Let's extend the `MinMax` class with some basic statistics capabilities using `Stats`:

```
class Stats[T < : AnyVal] (
    values: Vector[T])(implicit f : T => Double)
extends MinMax[T](values) {

    val zero = (0.0, 0.0)
    val sums = values.//: (zero)((s,x) =>(s._1 +x, s._2 + x*x)) //1

    lazy val mean = sums._1/values.size //2
    lazy val variance =
        (sums._2 - mean*mean*values.size)/(values.size-1)
    lazy val stdDev = Math.sqrt(variance)
    ...
}
```

The Stats class implements **immutable statistics**. Its constructor computes the sum of values and sum of square values, sums (line 1). The statistics such as mean and variance are computed once when needed by declaring these values as lazy (line 2). The Stats class inherits the normalization functions of MinMax.

Z-Score and Gauss

The Gaussian distribution of the input data is implemented by the gauss method of the Stats class.

The Gaussian distribution

M1: Gaussian for a mean μ and a standard deviation σ transformation is defined as:

$$y = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The code is as follows:

```
def gauss(mu: Double, sigma: Double, x: Double): Double = {
    val y = (x - mu)/sigma
    INV_SQRT_2PI*Math.exp(-0.5*y*y)/sigma
}
val normal = gauss(1.0, 0.0, _: Double)
```

The computation of the normal distribution is computed as a partially applied function. The Z-score is computed as a normalization of the raw data taking into account the standard deviation.

Z-score normalization

M2: Z-score for a mean μ and a standard deviation σ is defined as:

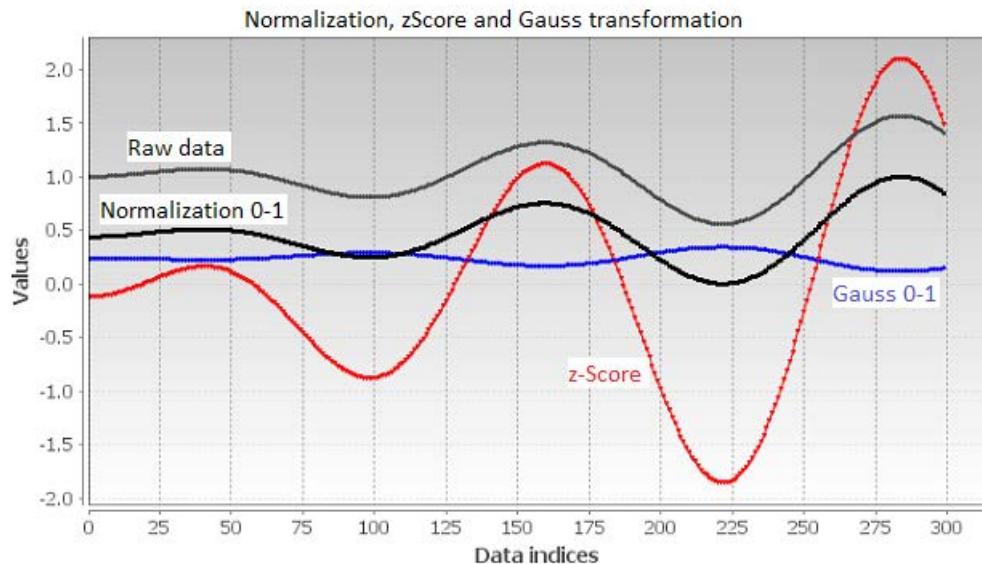
$$z_i = \frac{x_i - \mu}{\sigma}$$

The computation of the Z-score is implemented by the zScore method of Stats:

```
def zScore: DblVector = values.map(x => (x - mean) / stdDev )
```

Hello World!

The following graph illustrates the relative behavior of the zScore normalization and normal transformation:



A comparative analysis of linear, Gaussian, and Z-score normalization

Assessing a model

Evaluating a model is an essential part of the workflow. There is no point in creating the most sophisticated model if you do not have the tools to assess its quality. The validation process consists of defining some quantitative reliability criteria, setting a strategy such as a **K-fold cross-validation** scheme, and selecting the appropriate labeled data.

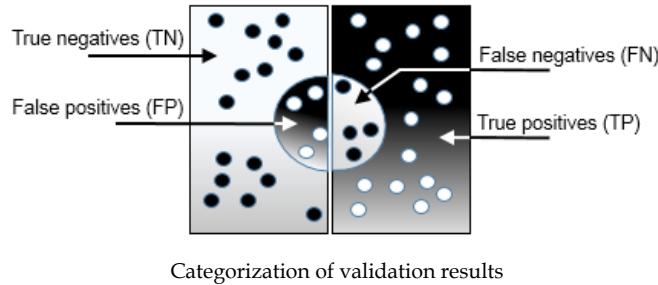
Validation

The purpose of this section is to create a reusable Scala class to validate models. For starters, the validation process relies on a set of metrics to quantify the fitness of a model generated through training.

Key quality metrics

Let's consider a simple classification model with two classes defined as positive (with respect to negative) represented with Black (with respect to White) color in the following diagram. Data scientists use the following terminologies:

- **True positives (TP):** These are observations that are correctly labeled as those that belong to the positive class (white dots on a dark background)
- **True negatives (TN):** These are observations that are correctly labeled as those that belong to the negative class (black dots on a light background)
- **False positives (FP):** These are observations incorrectly labeled as those that belong to the positive class (white dots on a dark background)
- **False negatives (FN):** These are observations incorrectly labeled as those that belong to the negative class (dark dots on a light background)



This simplistic representation can be extended to classification problems that involve more than two classes. For instance, false positives are defined as observations incorrectly labeled that belong to any class other than the correct one. These four factors are used for evaluating accuracy, precision, recall, and F and G measures, as follows:

- **Accuracy:** This is the percentage of observations correctly classified and is represented as ac .
- **Precision:** This is the percentage of observations correctly classified as positive in the group that the classifier has declared positive. It is represented as p .
- **Recall:** This is the percentage of observations labeled as positive that are correctly classified and is represented as r .
- **F_1 -measure or F_1 -score:** This measure strikes a balance between precision and recall. It is computed as the harmonic mean of the precision and recall with values ranging between 0 (worst score) and 1 (best score). It is represented as F_1 .

- **F_n score:** This is the generic F scoring method with an arbitrary n degree. It is represented as F_n .
- **G measure:** This is similar to the F-measure but is computed as the geometric mean of precision p and recall r . It is represented as G .

Validation metrics

M3: Accuracy ac , precision p , recall r , F_1 , F_n , and G scores are defined as follows:

$$ac = \frac{TP + TN}{TP + TN + FP + FN} \quad p = \frac{TP}{TP + FP} \quad r = \frac{TP}{TP + FN}$$

$$F_1 = \frac{2pr}{p+r} \quad F_n = \frac{(1+n^2)pr}{n^2p+r} \quad G = \sqrt{pr}$$

The computation of the precision, recall, and F_1 score depends on the number of classes used in the classifier. We will consider the following implementations:

- F-score validation for binomial (two classes) classification (that is, a positive and negative outcome)
- F-score validation for multinomial (more than two classes) classification

F-score for binomial classification

The binomial F validation computes the precision, recall, and F scores for the positive class.

Let's implement the F-score or F-measure as a specialized validation of the following:

```
trait Validation { def score: Double }
```

The `BinFValidation` class encapsulates the computation of the F_n score as well as precision and recall by counting the occurrences of TP , TN , FP , and FN values. It implements the **M3** formula. In the tradition of Scala programming, the class is immutable; it computes the counters for TP , TN , FP , and FN when the class is instantiated. The class takes the following three parameters:

- The expected values with the `0` value for a negative outcome and `1` for a positive outcome
- The set of observations, `xt`, is used for validating the model
- The predictive `predict` function classifies observations (line 1)

The code will be as follows:

```

class BinFValidation[T <: AnyVal] (
    expected: Vector[Int],
    xt: XVSeries[T])
    (predict: Array[T] => Int)(implicit f: T => Double)
extends Validation { //1

    val counters = {
        val predicted = xt.map( predict(_) )
        expected.zip(predicted)
            .aggregate(new Counter[Label])((cnt, ap) =>
                cnt + classify(ap._1, ap._2), _ ++ _) //2
    }

    override def score: Double = f1 //3
    lazy val f1 = 2.0*precision*recall/(precision + recall)
    lazy val precision = compute(FP) //4
    lazy val recall = compute(FN)

    def compute(n: Label): Double = {
        val denom = counters(TP) + counters(n)
        counters(TP).toDouble/denom
    }
    def classify(predicted: Int, expected: Int): Label = //5
        if(expected == predicted) if(expected == POSITIVE) TP else TN
        else if(expected == POSITIVE) FN else FP
    }
}

```

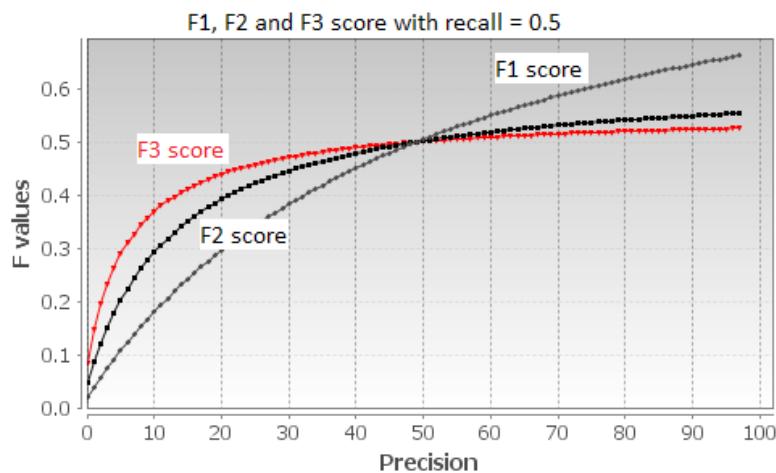
The constructor counts the number of occurrences for each of the four outcomes (TP , TN , FP , and FN) (line 2). The `precision`, `recall`, and `f1` values are defined as lazy values so they are computed only once, when they are accessed directly or the `score` method is invoked (line 4). The F_1 measure is the most commonly used scoring value for validating classifiers. Therefore, it is the default score (line 3). The `classify` private method extracts the qualifier from the `expected` and `predicted` values (line 5).

The `BinFValidation` class is independent of the type of classifier, training, labeling process, and type of observations.

Contrary to Java, which defines an enumerator as a class of types, Scala requires enumerators to be singletons. Enumerators extend the `scala.Enumeration` abstract class:

```
object Label extends Enumeration {
    type Label = Value
    val TP, TN, FP, FN = Value
}
```

The F-score formula with higher cardinality (F_n) with $n > 1$ favors precision over recall, which is shown in the following graph:



A comparative analysis of the impact of precision on F1, F2, and F3 score for a given recall

Multiclass scoring

Our implementation of the binomial validation computes the precision, recall, and F_1 score for the positive class only. The generic multinomial validation class presented in the next section computes these quality metrics for both positive and negative classes.

F-score for multinomial classification

The validation metric is defined by the M3 formula. The idea is quite simple: the precision and recall values are computed for all the classes and then they are averaged to produce a single precision and recall value for the entire model. The precision and recall for the entire model leverage the counts of TP , FP , FN , and TN introduced in the previous section.

There are two commonly used set of formulas to compute the precision and recall for a model:

- **Macro:** This method computes the precision and recall for each class, sums and then averages them up.
- **Micro:** This method sums the numerator and denominator of the precision and recall formulas for all the classes before computing the precision and recall.

We will use the macro formulas from now on.

Macro formulas for multinomial precision and recall

M4: The macro version of the precision p and recall r for a model of the c classes is computed as follows:

$$P = \frac{1}{c} \sum_{i=0}^{c-1} \frac{TP_i}{TP_i + FP_i} \quad r = \frac{1}{c} \sum_{i=0}^{c-1} \frac{TP_i}{TP_i + FN_i}$$

The computation of the precision and recall factor for a classifier with more than two classes requires the extraction and manipulation of the **confusion matrix**. We use the following convention: *expected values are defined as columns and predicted values are defined as rows*.

		Actual					
		1	2	3	4	5	6
Predicted	1	167	3	19	8	0	2
	2	11	107	3	27	4	12
	3	4	21	145	3	7	14
	4	9	17	4	179	20	0
	5	15	0	18	2	139	8
	6	1	6	0	24	8	164
		False negatives					

A confusion matrix for six class classification

The `MultiFValidation` multinomial validation class takes the following four parameters:

- The expected class index with the `0` value for a negative outcome and `1` for a positive outcome
- The set of observations, `xv`, is used for validating the model
- The number of classes in the model
- The `predict` predictive function classifies observations (line 7)

The code will be as follows:

```
class MultiFValidation[T <: AnyVal] {  
    expected: Vector[Int],  
    xv: XVSeries[T],  
    classes: Int  
    (predict: Array[T] => Int) (implicit f : T => Double)  
    extends Validation { //7  
  
        val confusionMatrix: Matrix[Int] = //8  
        labeled./:(Matrix[Int](classes)){case (m, (x,n)) =>  
            m + (n, predict(x), 1)} //9  
  
        val macroStats: DblPair = { //10  
            val pr= Range(0, classes)./:((0.0,0.0)((s, n) => {  
                val tp = confusionMatrix(n, n) //11  
                val fn = confusionMatrix.col(n).sum - tp //12  
                val fp = confusionMatrix.row(n).sum - tp //13  
                (s._1 + tp.toDouble/(tp + fp), s._2 +tp.toDouble/(tp + fn))  
            }))  
            (pr._1/classes, pr._2/classes)  
        }  
        lazy val precision: Double = macroStats._1  
        lazy val recall: Double = macroStats._1  
        def score: Double = 2.0*precision*recall/(precision+recall)  
    }  
}
```

The core element of the multiclass validation is the confusion matrix, `confusionMatrix` (line 8). Its elements at indices $(i, j) = (\text{index of expected class for an observation}, \text{index of the predicted class for the same observation})$ are computed using the expected and predictive outcome for each class (line 9).

As stated in the introduction of the section, we use the macro definition of the precision and recall (line 10). The count of a true positive, t_p , for each class corresponds to the diagonal element of the confusion matrix (line 11). The count of the f_n false negatives for a class is computed as the sum of the counts for all the predicted classes (column values), given an expected class except the true positive class (line 12). The count of the f_p false positives for a class is computed as the sum of the counts for all the expected classes (row values), given a predicted class except the true positive class (line 13).

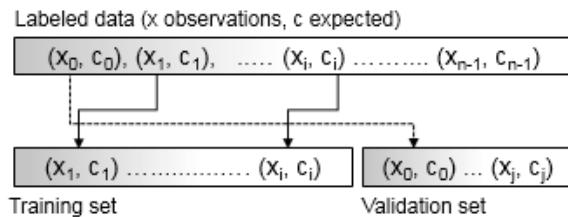
The formula for the computation of the F_1 score is the same as the formula used in the binomial validation.

Cross-validation

It is quite common that the labeled dataset (observations plus the expected outcome) available to the scientists is not very large. The solution is to break the original labeled dataset into K groups of data.

One-fold cross validation

The one-fold cross validation is the simplest scheme used for extracting a training set and validation set from a labeled dataset, as described in the following diagram:



An illustration of the generation of a one-fold validation set

The one-fold cross validation methodology consists of the following three steps:

1. Select the ratio of the size of the training set over the size of the validation set.
2. Randomly select the labeled observations for the validation phase.
3. Create the training set as the remaining labeled observations.

The one-fold cross validation is implemented by the `OneFoldXValidation` class. It takes the following three arguments: an `xt` vector of observations, the `expected` vector of expected classes, and `ratio` of the size of the training set over the size of the validation set (line 14):

```
type ValidationType[T] = Vector[(Array[T], Int)]
class OneFoldXValidation[T <: AnyVal] (
    xt: XVSeries[T],
    expected: Vector[Int],
    ratio: Double)(implicit f : T => Double) { //14
    val dataSet: (ValidationType[T], ValidationType[T]) //15
    def trainingSet: ValidationType[T] = dataSet._1
    def validationSet: ValidationType[T] = dataSet._1
}
```

The constructor of the `OneFoldXValidation` class generates the segregated training and validation sets from the set of observations and expected classes (line 15):

```
val dataSet: (Vector[LabeledData[T]], Vector[LabeledData[T]]) = {
    val labeledData = xt.drop(1).zip(expected) //16
    val trainingSize = (ratio*expected.size).floor.toInt //17

    val valSz = labeledData.size - trainingSize
    val adjSz = if(valSz < 2) 1
                else if(valSz >= labeledData.size) labeledData.size - 1
                else valSz //18
    val iter = labeledData.grouped(adjSz) //18
    val ordLabeledData = labeledData
        .map( _, Random.nextDouble) //19
        .sortWith( _._2 < _._2).unzip._1 //20

    (ordLabeledData.takeRight(adjValSz),
     ordLabeledData.dropRight(adjValSz)) //21
}
```

The initialization of the `OneFoldXValidation` class creates a `labeledData` vector of labeled observations by zipping the observations and the expected outcome (line 16). The training `ratio` value is used to compute the respective size of the training set (line 17) and validation set (line 18).

In order to create training and validations sets randomly, we zip the labeled dataset with a random generator (line 19), and then reorder the labeled dataset by sorting the random values (line 20). Finally, the method returns the pair of training set and validation set (line 21).

K-fold cross validation

The data scientist creates K training-validation datasets by selecting one of the groups as a validation set and then combining all the remaining groups into a training set, as illustrated in the following diagram. The process is known as the **K-fold cross validation** [2:7].



An illustration of the generation of a K-fold cross validation set

The third segment is used as validation data and all other dataset segments except S3 are combined into a single training set. This process is applied to each segment of the original labeled dataset.

Bias-variance decomposition

The challenge is to create a model that fits both the training set and subsequent observations to be classified during the validation phase.

If the model tightly fits the observations selected for training, there is a high probability that new observations may not be correctly classified. This is usually the case when the model is complex. This model is characterized as having a low bias with a high variance. Such a scenario can be attributed to the fact that the scientist is overly confident that the observations she/he selected for training are representative of the real world.

The probability of a new observation being classified as belonging to a positive class increases as the selected model fits loosely the training set. In this case, the model is characterized as having a high bias with a low variance.

The mathematical definition for the **bias**, **variance**, and **mean square error** (MSE) of the distribution are defined by the following formulas:

M5: Variance and bias for a true model, θ , is defined as:


$$\text{var } \hat{\theta} = E[(\hat{\theta} - E[\hat{\theta}])^2] \quad \text{bias}(\hat{\theta}) = \hat{\theta} - \theta \quad (\hat{\theta}: \theta \text{ estimate})$$

M6: Mean square error is defined as:

$$MSE = \text{var}(\hat{\theta}) + \text{bias}(\hat{\theta})^2$$

Let's illustrate the concept of bias, variance, and mean square error with an example. At this stage, you have not been introduced to most of the machines learning techniques. Therefore, we create a simulator to illustrate the relation between the bias and variance of a classifier. The components of the simulation are as follows:

- A training set, `training`
- A simulated target model of the target: `Double => Double` type extracted from the training set
- A set of possible models to evaluate

A model that exactly matches the training data overfits the target model. Models that approximate the target model will most likely underfit. The models in this example are defined by single variable functions.

These models are evaluated against a validation dataset. The `BiasVariance` class takes the target model, `target`, and the size of the `nValues` validation test as parameters (line 22). It merely implements the formula to compute the bias and variance for each model:

```
type Dbl_F = Double => Double
class BiasVariance[T](target: Dbl_F, nValues: Int)
  (implicit f: T => Double) //22
  def fit(models: List[Dbl_F]): List[DblPair] = { //23
    models.map(accumulate(_, models.size)) //24
  }
}
```

The `fit` method computes the variance and bias for each of the `models` model compared to the target model (line 23). It computes the mean, variance, and bias in the `accumulate` method (line 24):

```
def accumulate(f: Dbl_F, y:Double, numModels: Int): DblPair =
  Range(0, nValues) ./:(0.0, 0.0){ case((s,t) x) => {
    val diff = (f(x) - y)/numModels
    (s + diff*diff, t + Math.abs(f(x)-target(x))) //25
  } }
```

The training data is generated by the single variable function with the r_1 and r_2 noise components:

$$y = \frac{x}{5} \left(1 + \frac{1}{n} \sin \left(\frac{x}{10} + r1 \right) \right) + r2$$

The `accumulate` method returns a tuple (variance, bias) for each model, f (line 25). The model candidates are defined by the following family of single variable for values $n = 1, 2, \text{ and } 4$:

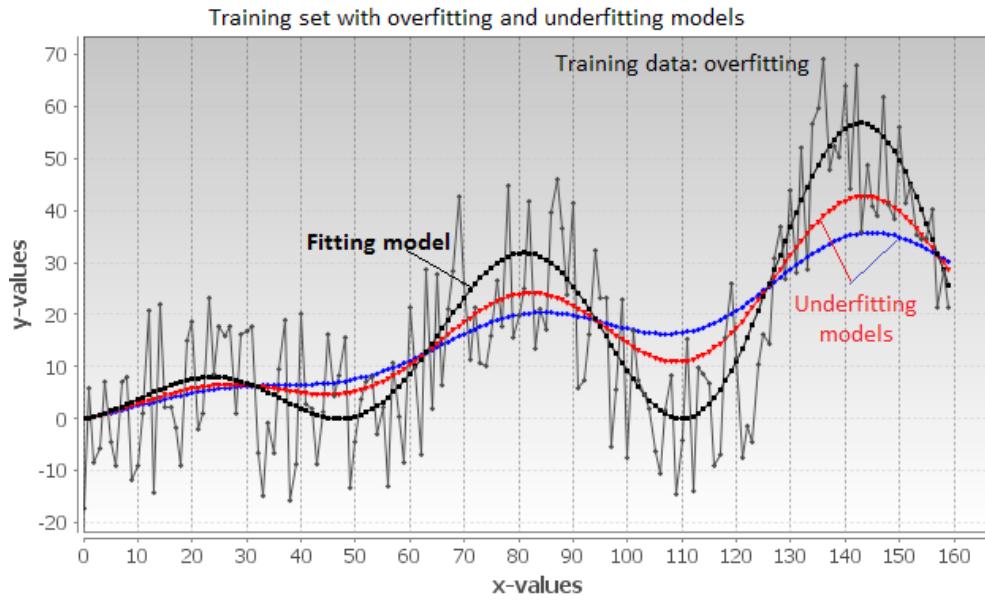
$$y = \frac{x}{5} \left(1 + \frac{1}{n} \sin \left(\frac{x}{10} \right) \right)$$

The target model (line 26) and `models` (line 27) belong to the same family of single variable functions:

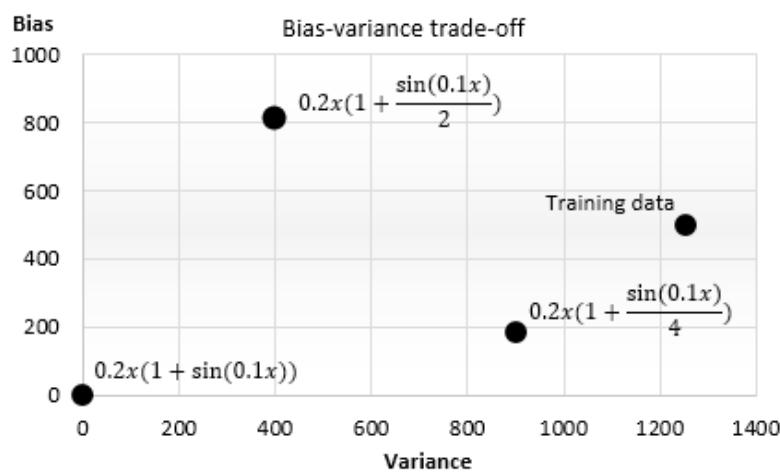
```
val template = (x: Double, n : Int) =>
  0.2*x*(1.0 + Math.sin(x*0.1)/n)
val training = (x: Double) => {
  val r1 = 0.45*(Random.nextDouble-0.5)
  val r2 = 38.0*(Random.nextDouble - 0.5) + Math.sin(x*0.3)
  0.2*x*(1.0 + Math.sin(x*0.1 + r1)) + r2
}
Val target = (x: Double) => template(x, 1) //26
val models = List[(Dbl_F, String)] ( //27
  ((x: Double) => template(x, 4), "Underfit1"),
  ((x: Double) => template(x, 2), "Underfit2"),
  ((x : Double) => training(x), "Overfit"),
  (target, "target"),
)
val evaluator = new BiasVariance[Double](target, 200)
evaluator.fit(models.map(_._1)) match { /* ... */ }
```

Hello World!

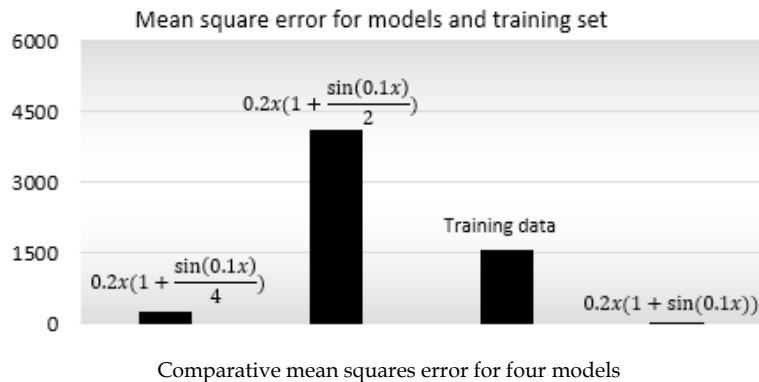
The **JFreeChart** library is used to display the training dataset and the models:



The model that replicates the training data overfits. The models that smooth the model with lower amplitude for the sine component of the template function underfit. The **variance-bias trade-off** for the different models and training data is illustrated in the following scatter chart:



The variance of each of the smoothing or approximating models is lower than the variance of the training set. As expected the target model, $0.2x(1+\sin(x/10))$, has no bias and no variance. The training set has a very high variance because it overfits any target model. The last chart compares the mean square error between each of the models, training set, and the target model:



Evaluating bias and variance

The section uses a fictitious target model and training set to illustrate the concept of the bias and variance of models. The bias and variance of machine learning models are actually estimated using validation data.

Overfitting

You can apply the methodology presented in the example to any classification and regression model. The list of models with low variance includes constant functions and models independent of the training set. High degree polynomials, complex functions, and deep neural networks have high variance. Linear regression applied to linear data has a low bias, while linear regression applied to nonlinear data has a higher bias [2:8].

Overfitting affects all aspects of the modeling process negatively, for example:

- It renders debugging difficult
- It makes the model too dependent of minor fluctuations (long tail) and noisy data
- It may discover irrelevant relationships between observed and latent features
- It leads to poor predictive performance

However, there are well-proven solutions to reduce overfitting [2:9]:

- Increasing the size of the training set whenever possible
- Reducing noise in labeled observations using smoothing and filtering techniques
- Decreasing the number of features using techniques such as principal components analysis, as discussed in the *Principal components analysis* section in *Chapter 4, Unsupervised Learning*
- Modeling observable and latent noisy data using Kalman or auto regressive models, as discussed in *Chapter 3, Data Preprocessing*
- Reducing inductive bias in a training set by applying cross-validation
- Penalizing extreme values for some of the model's features using regularization techniques, as discussed in the *Regularization* section in *Chapter 6, Regression and Regularization*

Summary

In this chapter, we established the framework for the different data processing units that will be introduced in this book. There is a very good reason why the topics of model validation and overfitting are explored early in this book. There is no point in building models and selecting algorithms if we do not have a methodology to evaluate their relative merits.

In this chapter, you were introduced to the following:

- The concept of monadic transformation for implicit and explicit models
- The versatility and cleanness of the Cake pattern and mixins composition in Scala as an effective scaffolding tool for data processing
- A robust methodology to validate machine learning models
- The challenge in fitting models to both training and real-world data

The next chapter will address the problem of overfitting by identifying outliers and reducing noise in data.

3

Data Preprocessing

Real-world data is usually noisy and inconsistent with missing observations. No classification, regression, or clustering model can extract relevant information from raw data.

Data preprocessing consists of cleaning, filtering, transforming, and normalizing raw observations using statistics in order to correlate features or groups of features, identify trends and models, and filter out noise. The purpose of cleansing raw data is as follows:

- To extract some basic knowledge from raw datasets
- To evaluate the quality of data and generate clean datasets for unsupervised or supervised learning

You should not underestimate the power of traditional statistical analysis methods to infer and classify information from textual or unstructured data.

In this chapter, you will learn how to:

- Apply commonly used moving average techniques to detect long-term trends in a time series
- Identify market and sector cycles using discrete Fourier series
- Leverage the discrete Kalman filter to extract the state of a linear dynamic system from incomplete and noisy observations

Time series in Scala

The overwhelming majority of examples used to illustrate the different machine algorithms in this book deal with time series or sequential, time-ordered set of observations.

Types and operations

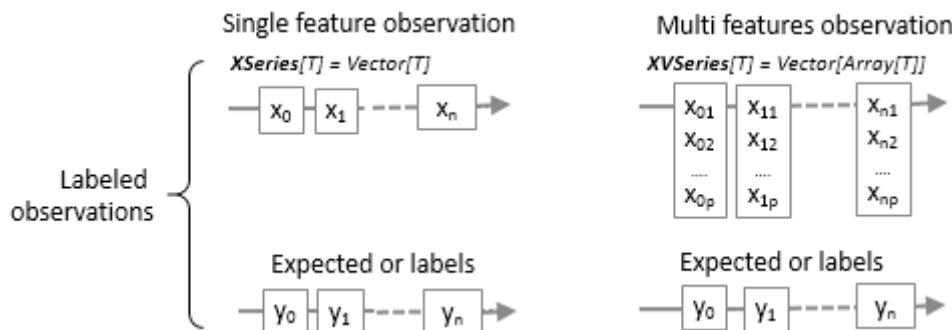
The *Primitives types* section under *Source code in Chapter 1, Getting Started*, introduced the types for a time series of a single `xSeries [T]` variable and multiple `xvSeries [T]` variables.

A time series of observations is a vector (a `Vector` type) of observation elements of the following types:

- A `T` type in the case of a single variable/feature observation
- An `Array[T]` type for observations with more than one variable/feature

A time series of labels or expected values is a single variable vector for which elements may have a primitive `Int` type for classification and `Double` for regression.

A time series of labeled observations is a pair of a vector of observations and a vector of labels:



The two generic `xSeries` and `xvSeries` types for the time series will be used as the two primary classes for the input data, from now on.

[

Structure of labeled observations

Throughout the book, labeled observations are defined either as a pair of vector of observations and a vector of labels/expected values or as a vector of a pair of {observation, label/expected value}.

]

The `Stats` class introduced in the *Profiling data* section in *Chapter 2, Hello World!*, implements some basic statistics and normalization for single variable observations. Let's create an `XTSeries` singleton to compute the statistics and normalize multidimensional observations:

```
object XTSeries {
    def zipWithShift[T] (xv: XSeries[T], n: Int): Vector[(T, T)] =
        xv.drop(n).zip(xv.view.dropRight(n)) //1

    def zipWithShift1[T] (xv: XSeries[T]): Vector[(T, T)] =
        xv.zip(xv.view.drop(n))

    def statistics[T <: AnyVal] (xt: XVSeries[T])
        (implicit f: T =>: Double): Vector[Stats[T]] =
        xt.transpose.map(Stats[T]( _ )) //2

    def normalize[T <: AnyVal] ( //3
        xt: XSeries[T], low: Double, high: Double)
        (implicit ordering: Ordering[T],
         f: T => Double): Try[DblVector] =
        Try(Stats[T](xt).normalize(low, high))
    ...
}
```

The first method of the `XTSeries` singleton generates a vector of a pair of elements by zipping the last $size - n$ elements of a time series with its first $size - n$ elements (line 1). The `statistics` (line 2) and `normalize` (line 3) methods operate on both the single and multivariable observations. These three methods are subsets of functionalities implemented in `XTSeries`.

Create a time series of the `XVSeries[T]` type by zipping the two `x` and `y` vectors and converting the pair into an array:

```
def zipToSeries[T: ClassTag] (
    x: Vector[T], y: Vector[T]): XVSeries[T]
```

Split a single or multidimensional time series, `xv`, into a two-time series at index, n :

```
def splitAt[T] (xv: XSeries[T], n: Int): (XSeries[T], XSeries[T])
```

Apply a `zScore` transform to a single dimension time series:

```
def zScore[T <: AnyVal] (xt: XSeries[T])
    (implicit f: T => Double): Try[DblVector]
```

Apply a `zScore` transform to a multidimension time series:

```
def zScores[T <: AnyVal](xt: XVSeries[T])
  (implicit f: T => Double): Try[XVSeries[Double]]
```

Transform a single dimension time series `x` into a new time series whose elements are $x(n) - x(n-1)$:

```
def delta(x: DblVector): DblVector
```

Transform a single dimension time series `x` into a new time series which elements if $(x(n) - x(n-1) > 0.0) 1 \text{ else } 0$:

```
def binaryDelta(x: DblVector): Vector[Int]
```

Compute the sum of the squared error between the two `x` and `z` arrays:

```
def sse[T <: AnyVal](x: Array[T], z: Array[T])
  (implicit f: T => Double): Double
```

Compute the mean squared error between the two `x` and `z` arrays:

```
def mse[T <: AnyVal](x: Array[T], z: Array[T])
  (implicit f: T => Double): Double
```

Compute the mean squared error between the two `x` and `z` vectors:

```
def mse(x: DblVector, z: DblVector): Double
```

Compute the statistics for each feature of a multidimensional time series:

```
def statistics[T <: AnyVal](xt: XVSeries[T])
  (implicit f: T => Double): Vector[Stats[T]]
```

Apply a `f` function to a zipped pair of multidimensional vectors of the `XVSeries` type:

```
def zipToVector[T](x: XVSeries[T], y: XVSeries[T])
  (f: (Array[T], Array[T]) => Double): XSeries[Double] =
  x.zip(y.view).map{ case (x, y) => f(x, y)}
```

The magnet pattern

Some operations on the time series that are implemented as the `XTSERIES` methods may have a large variety of input and output types. Scala and Java support method overloading that has the following limitations:

- It does not prevent the type collision caused by the erasure type in the JVM
- It does not allow lifting to a single, generic function
- It does not completely reduce code redundancy

The transpose operator

Let's consider the transpose operator for any kind of multidimensional time series. The transpose operator can be objectified as the `Transpose` trait:

```
sealed trait Transpose {
    type Result //4
    def apply(): Result //5
}
```

The trait has an abstract `Result` type (line 4) and an abstract `apply()` constructor (line 5) that allows us to create a generic `transpose` method with any kind of combination of input and output types. The conversion type for the input and output types of the `transpose` method is defined as `implicit`:

```
implicit def xvSeries2Matrix[T: ClassTag] (from: XVSeries[T]) =
    new Transpose { type Result = Array[Array[T]] //6
        def apply(): Result = from.toArray.transpose
    }
implicit def list2Matrix[T: ClassTag] (from: List[Array[T]]) =
    new Transpose { type Result = Array[Array[T]] //7
        def apply(): Result = from.toArray.transpose
    }
...
```

The first `xvSeries2Matrix` implicit transposes a time series of the `XVSeries[T]` type into a matrix with elements of the `T` type (line 6). The `list2Matrix` implicit transposes a time series of the `List[Array[T]]` type into a matrix with elements of the `T` type (line 7).

The generic `transpose` method is written as follows:

```
def transpose(tpose: Transpose): tpose.Result = tpose()
```

The differential operator

The second candidate for the magnet pattern is the computation of the differential in a time series. The purpose is to generate the time series $\{x_{t+1} - x_t\}$ from a time series $\{x_t\}$:

```
sealed trait Difference[T] {  
    type Result  
    def apply(f: (Double, Double) => T): Result  
}
```

The `Difference` trait allows us to compute the differential of a time series with arbitrary element types. For instance, the differential of a one-dimensional vector of the `Double` type is defined by the following implicit conversion:

```
implicit def vector2Double[T](x: DblVector) = new Difference[T] {  
    type Result = Vector[T]  
    def apply(f: (Double, Double) => T): Result = //8  
        zipWithShift(x, 1).collect{case(next,prev) =>f(prev,next)}  
}
```

The `apply()` constructor takes one argument: the user-defined `f` function that computes the difference between two consecutive elements of the time series (line 8). The generic difference method is as follows:

```
def difference[T] (  
    diff: Difference[T],  
    f: (Double, Double) => T): diff.Result = diff(f)
```

Here are some of the predefined differential operators of a time series for which the output of the operator has the `Double` (line 9), `Int` (line 10), and `Boolean` (line 11) types:

```
val diffDouble = (x: Double,y: Double) => y -x //9  
val diffInt = (x: Double,y: Double) => if(y > x) 1 else 0 //10  
val diffBoolean = (x: Double,y: Double) => (y > x) //11
```

The differential operator is used to implement the `labeledData` method to generate labeled data from observations with two features and a target (labels) dataset:

```
def differentialData[T] (  
    x: DblVector,  
    y: DblVector,  
    target: DblVector,  
    f: (Double,Double) =>T): Try[(XVSeries[Double],Vector[T])] =  
    Try((zipToSeries(x,y), difference(target, f)))
```

The structure of the labeled data is the pair of observations and the differential of target values.

Lazy views

A view in Scala is a proxy collection that represents a collection but implements the data transformation or higher-order method lazily. The elements of a view are defined as lazy values, which are instantiated on demand.

One important advantage of views over a **strict** (or fully allocated) collection is the reduced memory consumption.

Let's take a look at the `aggregator` data transformation introduced in the *Instantiating the workflow* section under *A workflow computational model* in Chapter 2, *Hello World!*. There is no need to allocate the entire set of `x.size` of elements: the higher-order `find` method may exit after only a few elements have been read (line 12):

```
val aggregator = new ETransform[Int] (splits) {
    override def |> : PartialFunction[U, Try[V]] = {
        case x: U if (!x.isEmpty) =>
            Try( Range(0, x.size).view.find(x(_) == 1.0).get) //12
    }
}
```

Views, iterators, and streams

Views, iterators, and streams share the same objective of constructing elements on demand. There are, however, some major differences:

- Iterators do not persist elements of the collection (read once)
- Streams allow operations to be performed on the collection with an undefined size

Moving averages

Moving averages provide data analysts and scientists with a basic predictive model. Despite its simplicity, the moving average method is widely used in a variety of fields, such as marketing survey, consumer behavior, or sport statistics. Traders use the moving average method to identify different levels of support and resistance for the price of a given security.

An average reducing function

Let's consider the time series $x_t = x(t)$ and function $f(x_{t-p}, \dots, x_t)$ that reduces the last p observations into a value or average. The estimation of the observation at t is defined by the following formula:



$$\tilde{x}_t = f(x_{t-p+1}, \dots, x_t) \quad \forall t \geq p$$

Here, f is an average reducing function from the previous p data points.

The simple moving average

The simple moving average is the simplest form of the moving averages algorithms [3:1]. The simple moving average of period p estimates the value at time t by computing the average value of the previous p observations using the following formula.

The simple moving average

M1: The simple moving average of a time series $\{x_t\}$ with a period p is computed as the average of the last p observations:



$$\tilde{x}_t = \begin{cases} \frac{1}{p} \sum_{j=t-p+1}^t x_j & \forall t \geq p \\ 0 & \forall t < p \end{cases}$$

M2: The computation is implemented iteratively using the following formula:

$$\tilde{x}_t = \tilde{x}_{t-1} + \frac{1}{p} (x_t - x_{t-p}) \quad \forall t \geq p$$

Here, \tilde{x}_t is the estimate or simple moving average value at time t .

Let's build a class hierarchy of moving average algorithms, with the parameterized `MovingAverage` trait as its root:

```
trait MovingAverage[T]
```

We use the generic `XSeries[T]` type and the data transform with the `ETransform` explicit configuration, introduced in the *Explicit models* section under *Monadic data transformation* in *Chapter 2, Hello World!*, to implement the simple moving average, `SimpleMovingAverage`:

```
class SimpleMovingAverage[T <: AnyVal](period: Int)
    (implicit num: Numeric[T], f: T => Double) //1
    extends Etransform[Int](period) with MovingAverage[T] {

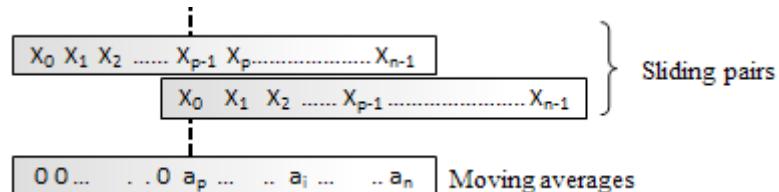
    type U = XSeries[T] //2
    type V = DblVector //3

    val zeros = Vector.fill(0.0)(period-1)
    override def |> : PartialFunction[U, Try[V]] = {
        case xt: U if( xt.size >= period ) => {
            val splits = xt.splitAt(period)
            val slider = xt.take(xt.size - period).zip(splits._2) //4

            val zero = splits._1.sum/period //5
            Try( zeros ++ slider.scanLeft(zero) {
                case (s, (x,y)) => s + (x - y)/period }) //7
        }
    }
}
```

The class is parameterized for the `T` type of elements of the input time series; *we cannot make any assumption regarding the type of input data*. The type of the elements of the output time series is `Double`. The implicit instantiation of the `Numeric[T]` class is required by the `sum` and / arithmetic operators (line 1). The simple moving average implements `ETransform` by defining the abstract `U` types for the input (line 2) and `V` for the output (line 3) as a time series, `DblVector`.

The implementation has a few interesting elements. First, the set of observations is duplicated and the index in the resulting clone instance is shifted by `p` observations before being zipped with the original to the array of a pair of `slider` values (line 4):



The sliding algorithm to compute moving averages

The average value is initialized with the mean value of the first period data points (line 5). The first period values of the trends are initialized to zero (line 6). The method concatenates the initial null values and the computed average values to implement the **M2** formula (line 7).

The weighted moving average

The weighted moving average method is an extension of the simple moving average by computing the weighted average of the last p observations [3:2]. The weights α_j are assigned to each of the last p data points x_j and are normalized by the sum of the weights.

The weighted moving average

M3: The weighted moving average of a series $\{x_i\}$ with a period p and a normalized weights distribution $\{\alpha_j\}$ is given by the following formula:



$$\tilde{x}_t = \sum_{j=t-p+1}^t \alpha_{j-t+p-1} x_j \quad \forall t \geq p \quad \text{subject to } \sum_{i=0}^{p-1} \alpha_i = 1$$

$0 \quad \forall t < p$

Here, \tilde{x}_t is the estimate or simple moving average value at time t .

The implementation of the `WeightedMovingAverage` class requires the computation of the last p (`weights.size`) data points. There is no simple iterative formula to compute the weighted moving average at time $t + 1$ using the moving average at time t :

```
class WeightedMovingAverage[@specialized(Double) T <: AnyVal] (
    weights: DblArray)
    (implicit num: Numeric[T], f: T => Double)
extends SimpleMovingAverage[T](weights.length) { //8

    override def |> : PartialFunction[U, Try[V]] = {
        case xt: U if(xt.size >= weights.length) => {
            val smoothed = (config to xt.size).map(i =>
                xt.slice(i - config, i).zip(weights) //9
                    .map { case(x, w) => x * w }.sum //10
            )
            Try(zeros ++ smoothed) //11
        }
    }
}
```

The computation of the weighted moving average is a bit more involved than the simple moving average. Therefore, we specify the generation of the byte code that is dedicated to the `Double` type using the specialized annotation. The weighted moving average inherits the `SimpleMovingAverage` class, and therefore, implements the `ETransform` explicit transformation for a configuration of weights, with input observations of the `XSeries[T]` type and output observations of the `DblVector` type. The implementation of `M3` formula generates a smoothed time series by slicing (line 9) the input time series and then computing the inner product of weights and the slice of the time series (line 10).

As with the simple moving average, the output is the concatenation of the initial `weights.size` null values, `zeros`, and the smoothed data (line 11).

The exponential moving average

The exponential moving average is widely used in financial analysis and marketing surveys because it favors the latest values. The older the value, the less impact it has on the moving average value at time t [3:3].

The exponential moving average

M4: The exponential moving average of a series $\{x_t\}$ and smoothing factor α is computed by the following iterative formula:

$$\begin{aligned}\tilde{x}_t &= (1 - \alpha)\tilde{x}_{t-1} + \alpha x_t \quad \forall t > 0 \quad 0 < \alpha < 1 \\ \tilde{x}_0 &= x_0\end{aligned}$$

Here, \tilde{x} is the value of the exponential average at t .

The implementation of the `ExpMovingAverage` class is rather simple. The constructor has a single α argument (the decay rate) (line 12):

```
class ExpMovingAverage[@specialized(Double) T <: AnyVal] (
  alpha: Double)      //12
  (implicit f: T => Double)
extends ETransform[Double](alpha) with MovingAverage[T] { //13

  type U = XSeries[T]      //14
  type V = DblVector       //15

  override def |> : PartialFunction[U, Try[V]] = {
    case xt: U if( xt.size > 0) => {
      val alpha_1 = 1-alpha
      var y: Double = data(0)
      Try( xt.view.map(x => {
        val z = x*alpha + y*alpha_1; y = z; z})) //16
    }
  }
}
```

```
    }
}
}
```

The exponential moving average implements the `ETransform` (line 13) by defining the abstract `U` types for the input (line 14) as a time series named `xseries[T]` and `V` for the output (line 15) as a time series named `DblVector`. The `|>` method applies the **M4** formula to all the observations of the time series within a `map` (line 16).

The version of the constructor that uses the `p` period to compute $\alpha = 1/(p+1)$ as an argument is implemented using the Scala `apply` method:

```
def apply[T <: AnyVal] (p: Int)
  (implicit f: T => Double): ExpMovingAverage[T] =
  new ExpMovingAverage[T] (2/(p + 1))
```

Let's compare the results generated from these three moving averages methods with the original price. We use a data source, `DataSource`, to load and extract values from the historical daily closing stock price of Bank of America (BAC), which is available at the Yahoo Financials pages. The `DataSink` class is responsible for formatting and saving the results into a CSV file for further analysis. The `DataSource` and `DataSink` classes are described in detail in the *Data extraction* section in the *Appendix A, Basic Concepts*:

```
import YahooFinancials._
val hp = p >>1
val w = Array.tabulate(p) (n =>
  if(n == hp) 1.0 else 1.0/(Math.abs(n - hp)+1)) //17
val sum = w.sum
val weights = w.map { _ / sum } //18

val dataSrc = DataSource(s"$RESOURCE_PATH$symbol.csv", false)//19
val pfnsMvAve = SimpleMovingAverage[Double] (p) |> //20
val pfnsWMvAve = WeightedMovingAverage[Double] (weights) |>
val pfnsEMvAve = ExpMovingAverage[Double] (p) |>

for {
  price <- dataSrc.get(adjClose) //21
  if(pfnsMvSve.isDefinedAt(price) )
  sMvOut <- pfnsMvAve(price) //22
  if(pfnsWMvSve.isDefinedAt(price)
  eMvOut <- pfnsWMvAve(price)
  if(pfnsEMvSve.isDefinedAt(price)
  wMvOut <- pfnsEMvAve(price)
} yield {
  val dataSink = DataSink[Double] (s"$OUTPUT_PATH$p.csv")
  val results = List[DblSeries] (price, sMvOut, eMvOut, wMvOut)
  dataSink |> results //23
}
```

 **isDefinedAt**

Each of the partial function is validated by a call to `isDefinedAt`. From now on, the validation of a partial function will be omitted throughout the book for the sake of clarity.

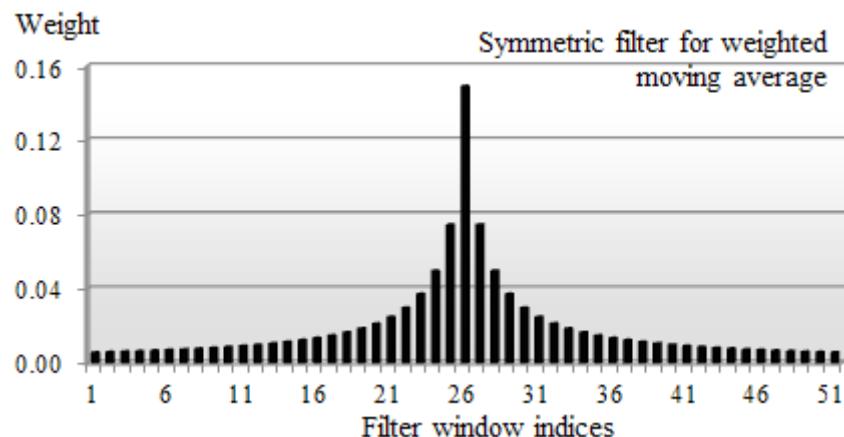
The coefficients for the weighted moving average are generated (line 17) and normalized (line 18). The trading data regarding the ticker symbol, BAC, is extracted from the Yahoo Finances CSV file (line 19), `YahooFinancials`, using the `adjClose` extractor (line 20). The next step is to initialize the `pfnSMvAve`, `pfnWMvAve`, and `pfnEMvAve` partial functions related to each of the moving average (line 21). The invocation of the partial functions with `price` as an argument generates the three smoothed time series (line 22).

Finally, a `DataSink` instance formats and dumps the results into a file (line 23).

 **Implicit postfixOps**

The instantiation of the `filter` $|>$ partial function requires that the post fix operation, `postfixOps`, be made visible by importing `scala.language.postfixOps`.

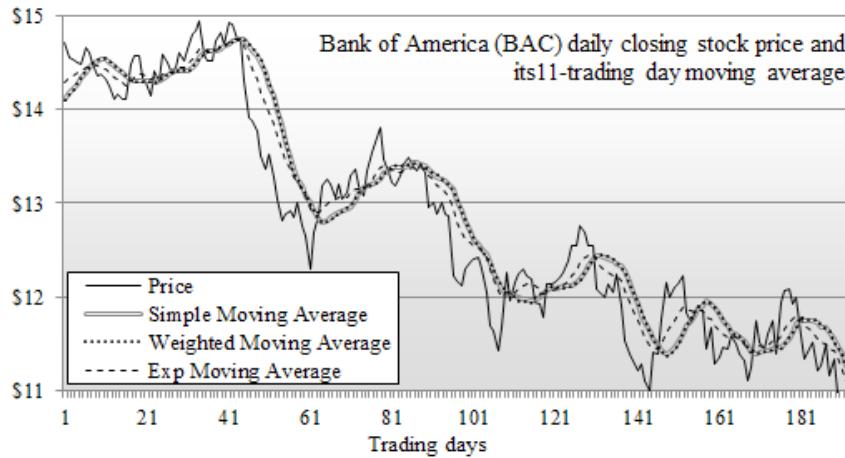
The weighted moving average method relies on a symmetric distribution of normalized weights computed by a function passed as an argument of the generic `tabulate` method. Note that the original price time series is displayed if one of the specific moving averages cannot be computed. The following graph is an example of a symmetric filter for weighted moving averages:



An example of a symmetric filter for weighted moving averages

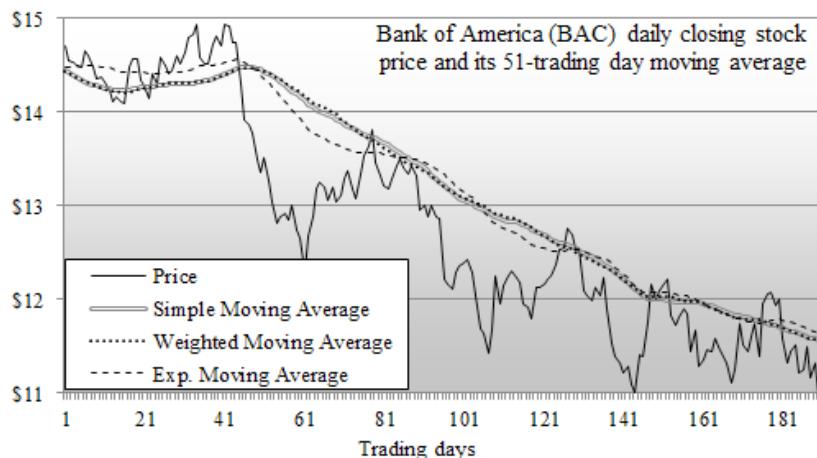
Data Preprocessing

The three moving average techniques are applied to the price of the stock of Bank of America stock (BAC) over 200 trading days. Both the simple and weighted moving averages use a period of 11 trading days. The exponential moving average method uses a scaling factor of $2/(11+1) = 0.1667$:



11-day moving averages of the historical stock price of Bank of America

The three techniques filter the noise out of the original historical price time series. The exponential moving average reacts to a sudden price fluctuation despite the fact that the smoothing factor is low. If you increase the period to 51 trading days (which is equivalent to two calendar months), the simple and weighted moving averages produce a time series smoother than the exponential moving average with $\alpha = 2/(p+1) = 0.038$:



51-day moving averages of the historical stock price of Bank of America

You are invited to experiment further with different smooth factors and weight distributions. You will be able to confirm the following basic rule: as the period of the moving average increases, noise with decreasing frequencies is eliminated. In other words, the window of allowed frequencies is shrinking. The moving average acts as a **low-pass filter** that preserves only lower frequencies.

Fine-tuning the period of a smoothing factor is time consuming. Spectral analysis, or more specifically, Fourier analysis transforms a time series into a sequence of frequencies, which provide the statistician with a more powerful frequency analysis tool.

The moving average on a multidimensional time series

The moving average techniques are presented for a single feature or variable time series, for the sake of simplicity. Moving averages on multidimensional time series are computed by executing a single variable moving average for each feature using the `transform` method of `XTSeries`, which is introduced in the first section. For example, the simple moving average applied to a multidimensional time series, `xt`. The smoothed values are computed as follows:

```
val pfnMv = SimpleMovingAverage[Double] (period) |>
  val smoothed = transform(xt, pfnMv)
```

Fourier analysis

The purpose of **spectral density estimation** is to measure the amplitude of a signal or a time series according to its frequency [3:4]. The objective is to estimate the spectral density by detecting periodicities in the dataset. A scientist can better understand a signal or time series by analyzing its harmonics.

The spectral theory

Spectral analysis for a time series should not be confused with the spectral theory, a subset of linear algebra that studies eigenfunctions on **Hilbert** and **Banach** spaces. In fact, harmonic analysis and Fourier analysis are regarded as subsets of the spectral theory.

Let's explore the concept behind the discrete Fourier series as well as its benefits as applied to financial markets. The **Fourier analysis** approximates any generic function as the sum of trigonometric functions, sine and cosine.

Complex Fourier transform

This section focuses on the discrete Fourier series for real values. The generic Fourier transform applies to complex values [3:5].

The decomposition in a basic trigonometric function process is known as the **Fourier transform** [3:6].

Discrete Fourier transform

A time series $\{x_i\}$ can be represented as a discrete real-time domain function f , $x = f(t)$. In the 18th century, Jean Baptiste Joseph Fourier demonstrated that any continuous periodic function f can be represented as a linear combination of sine and cosine functions. The **discrete Fourier transform (DFT)** is a linear transformation that converts a time series into a list of coefficients of a finite combination of complex or real trigonometric functions, ordered by their frequencies.

The frequency ω of each trigonometric function defines one of the harmonics of the signal. The space that represents the signal amplitude versus frequency of the signal is known as the **frequency domain**. The generic DFT transforms a time series into a sequence of frequencies defined as complex numbers $a + j\varphi$ ($j^2 = -1$), where a is the amplitude of the frequency and φ is the phase.

This section is dedicated to the real DFT that converts a time series into an ordered sequence of frequencies with real values.

Real discrete Fourier transform

M5: A periodic function f can be represented as an infinite combination of sine and cosine functions:

$$f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos(nx) + \sum_{k=1}^{\infty} b_k \sin(nx)$$

M6: The Fourier cosine transform of a function f is defined as:

$$\mathcal{F}^c(f, k) = \int_{-\infty}^{\infty} \cos(2\pi kx) f(x) dx$$

M7: The discrete real cosine series of a function $f(-x) = f(x)$ is defined as:

$$f(x) = f(-x) = \frac{a_0}{2} + \sum_{k=1}^{2N-3} a_k \cos(kx) \quad \text{where } a_k = \frac{2}{\pi} \int_0^{\pi} f(t) \cos(kt) . dt$$

M8: The Fourier sine transform of a function is defined as:



$$\mathcal{F}^s(f, k) = \int_{-\infty}^{\infty} \sin(2\pi kx) f(x) dx$$

M9: The discrete real sine series of a function $f(-x) = f(x)$ is defined as:

$$f(x) = -f(-x) = \sum_{k=1}^{2N-3} b_k \sin(kx) \quad \text{where } b_k = \frac{2}{\pi} \int_0^{\pi} f(t) \sin(kt) . dt$$

The computation of the Fourier trigonometric series is time consuming with an asymptotic time complexity of $O(n^2)$. Scientists and mathematicians have been working to make the computation as effective as possible. The most common numerical algorithm used to compute the Fourier series is the **Fast Fourier Transform (FFT)** created by J.W. Cooley and J. Tukey [3:7].

The algorithm called Radix-2 version recursively breaks down the Fourier transform for a time series of N data points into any combination of N_1 and N_2 sized segments such as $N = N_1 N_2$. Ultimately, the discrete Fourier transform is applied to the deeper-nested segments.



The Cooley-Tukey algorithm

I encourage you to implement the Radix-2 Cooley-Tukey algorithm in Scala using a tail recursion.

The Radix-2 implementation requires that the number of data points is $N=2^n$ for even functions (sine) and $N=2^n+1$ for cosine. There are two approaches to meet this constraint:

- Reduce the actual number of points to the next lower radix, $2^n < N$
- Extend the original time series by padding it with 0 to the next higher radix, $N < 2^n+1$

Padding the time series is the preferred option because it does not affect the original set of observations.

Let's define a `DTransform` trait for any variant of the discrete Fourier transform. The first step is to wrap the default configuration parameters used in the Apache Commons Math library into a `Config` singleton:

```
trait DTransform {
    object Config {
        final val FORWARD = TransformType.FORWARD
        final val INVERSE = TransformType.INVERSE
        final val SINE = DstNormalization.STANDARD_DST_I
        final val COSINE = DctNormalization.STANDARD_DCT_I
    }
    ...
}
```

The main purpose of the `DTransform` trait is to pad the `vec` time series with zero values:

```
def pad(vec: DblVector,
       even: Boolean = true)(implicit f: T => Double): DblArray = {
    val newSize = padSize(vec.size, even) //1
    val arr: DblVector = vec.map(_.toDouble)
    if( newSize > 0) arr ++ Array.fill(newSize)(0.0) else arr //2
}

def padSize(xtSz: Int, even: Boolean= true): Int = {
    val sz = if( even ) xtSz else xtSz-1 //3
    if( (sz & (sz-1)) == 0) 0
    else {
        var bitPos = 0
        do { bitPos += 1 } while( (sz >> bitPos) > 0) //4
        (if(even) (1<<bitPos) else (1<<bitPos)+1) - xtSz
    }
}
```

The `pad` method computes the optimal size of the frequency vector as 2^N by invoking the `padSize` method (line 1). It then concatenates the padding with the original time series or vector of observations (line 2). The `padSize` method adjusts the size of the data depending on whether the time series has initially an even or odd number of observations (line 3). It relies on bit operations to find the next radix, N (line 4).

The while loop



Scala developers prefer Scala higher-order methods for collections to implement the iterative computation. However, nothing prevents you from using the traditional `while` or `do { ... } while` loop if either readability or performance is an issue.

The fast implementation of the padding method, `pad`, consists of detecting the number of N observations as a power of 2 (the next highest radix). The method evaluates if N and $(N-1)$ are zero after it shifts the number of bits in the value, N . The code illustrates the effective use of implicit conversion to make the code readable in the `pad` method:

```
val arr: DblVector = vec.map(_.toDouble)
```

The next step is to write the `DFT` class for the real sine and cosine discrete transforms by subclassing `DTransform`. The class relies on the padding mechanism implemented in `DTransform` whenever necessary:

```
class DFT[@specialized(Double) T <: AnyVal] (
    eps: Double)(implicit f: T => Double)
    extends ETransform[Double](eps) with DTransform { //5
    type U = XSeries[T] //6
    type V = DblVector

    override def |> : PartialFunction[U, Try[V]] = { //7
        case xv: U if(xv.size >= 2) => fwrd(xv).map(_.toArray)
    }
}
```

We treat the discrete Fourier transform as a transformation on the time series using an explicit `ETransform` configuration (line 5). The `U` data type of the input and the `V` type of the output have to be defined (line 6). The `|>` transformation function delegates the computation to the `fwrd` method (line 7):

```
def fwrd(xv: U): Try[(RealTransformer, DblArray)] = {
    val rdt = if(Math.abs(xv.head) < config) //8
        new FastSineTransformer(SINE) //9
    else new FastCosineTransformer(COSINE) //10

    val padded = pad(xv.map(_.toDouble), xv.head == 0.0).toArray
    Try( (rdt, rdt.transform(padded, FORWARD)) )
}
```

The `fwd` method selects the discrete Fourier sine series if the first value of the time series is 0.0, otherwise it selects the discrete cosine series. This implementation automates the selection of the appropriate series by evaluating `xt.head` (line 8). The transformation invokes the `FastSineTransformer` (line 9) and `FastCosineTransformer` (line 10) classes of the Apache Commons Math library [3:8] introduced in the first chapter.

This example uses the standard formulation of the cosine and sine transformations, defined by the `COSINE` argument. The orthogonal normalization that normalizes the frequency by a factor of $1/\sqrt{2(N-1)}$, where N is the size of the time series, generates a cleaner frequency spectrum for a higher computation cost.

The `@specialized` annotation

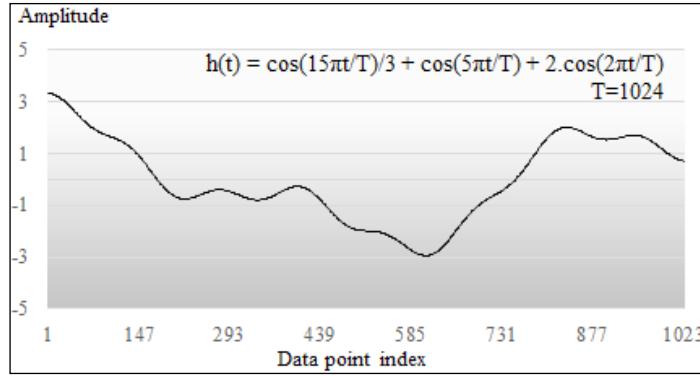
The `@specialized(Double)` annotation is used to instruct the Scala compiler to generate a specialized and more efficient version of the class for the `Double` type. The drawback of the specialization is the duplication of byte code as the specialized version coexists with the parameterized classes [3:9].

In order to illustrate the different concepts behind DFTs, let's consider the case of a time series generated by a `h` sequence of sinusoidal functions:

```
val F = Array[Double] (2.0, 5.0, 15.0)
val A = Array[Double] (2.0, 1.0, 0.33)

def harmonic(x: Double, n: Int): Double =
    A(n)*Math.cos(Math.PI*F(n)*x)
val h = (x: Double) =>
    Range(0, A.size).aggregate(0.0)((s, i) =>
        s + harmonic(x, i), _ + _)
```

As the signal is synthetically created, we can select the size of the time series to avoid padding. The first value in the time series is not null, so the number of observations is 2^n+1 . The data generated by the `h` function is plotted as follows:



An example of sinusoidal time series

Let's extract the frequencies' spectrum for the time series generated by the `h` function. The data points are created by tabulating the `h` function. The frequencies spectrum is computed with a simple invocation of the explicit `| >` data transformation of the DFT class:

```

val OUTPUT1 = "output/chap3/simulated.csv"
val OUTPUT2 = "output/chap3/smoothed.csv"
val FREQ_SIZE = 1025; val INV_FREQ = 1.0/FREQ_SIZE

val pfnDFT = DFT[Double] |> //11
for {
    values <- Try(Vector.tabulate(FREQ_SIZE)
                  (n => h(n*INV_FREQ))) //12
    output1 <- DataSink[Double](OUTPUT1).write(values)
    spectrum <- pfnDFT(values)
    output2 <- DataSink[Double](OUTPUT2).write(spectrum) //13
} yield {
    val results = format(spectrum.take(DISPLAY_SIZE),
    "x/1025", SHORT)
    show(s"$DISPLAY_SIZE frequencies: $results")
}

```

The execution of the data simulator follows these steps:

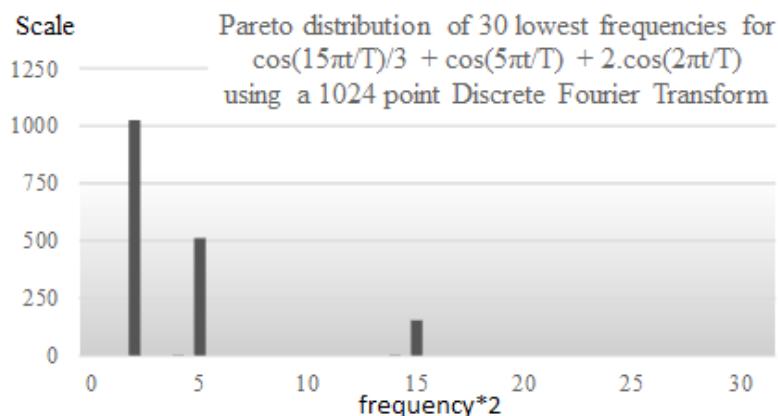
1. Generate a raw data with the 3-harmonic `h` function (line 12).
2. Instantiate the partial function generated by the transformation (line 11).
3. Store the resulting frequencies in a data sink (filesystem) (line 13).

Data sinks and spreadsheets



In this particular case, the results of the discrete Fourier transform are dumped into a CSV file so that it can be loaded into a spreadsheet. Some spreadsheets support a set of filtering techniques that can be used to validate the result of the example. A simpler alternative would be to use JFreeChart.

The spectrum of frequencies of the time series, plotted for the first 32 points, clearly shows three frequencies at $k = 2, 5$, and 15 . This result is expected because the original signal is composed of three sinusoidal functions. The amplitude of these frequencies are $1024/1$, $1024/2$, and $1024/6$, respectively. The following plot represents the first 32 harmonics for the time series:



The frequency spectrum for a three-frequency sinusoidal

The next step is to use the frequencies spectrum to create a low-pass filter using DFT. There are many algorithms available to implement a low or pass band filter in the time domain, from autoregressive models to the Butterworth algorithm. However, the discrete Fourier transform is still a very popular technique to smooth signals and identify trends.

Big Data



A DFT for a large time series can be very computation intensive. One option is to treat the time series as a continuous signal and sample it using the **Nyquist** frequency. The Nyquist frequency is half of the sampling rate of a continuous signal.

DFT-based filtering

The purpose of this section is to introduce, describe, and implement a noise filtering mechanism that leverages the discrete Fourier transform. The idea is quite simple: the forward and inverse Fourier series are used sequentially to convert the raw data from the time domain to the frequency domain and back. The only input you need to supply is a function g that modifies the sequence of frequencies. This operation is known as the convolution of the filter g and the frequencies' spectrum.

A convolution is similar to an inner product of two time series in the frequencies domain. Mathematically, the convolution is defined as follows:

Convolution

M10: The convolution of two functions f and g is defined as:

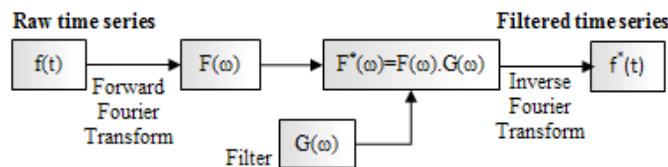
$$\langle f, g \rangle = \int_{-\infty}^{\infty} f(t) \cdot g(x-t) dt$$

M11: The convolution F of a time series $x = \{x_i\}$ with a frequency spectrum ω^x and a filter f in frequency domain ω^f is defined as:

$$F(x * f) = F(x) \cdot F(f) = \sum_{j=0}^{N-1} \omega_j^x \cdot \omega_{k-j}^f$$

Let's apply the convolution to our filtering problem. The filtering algorithm using the discrete Fourier transform consists of five steps:

1. Pad the time series to enable the discrete sine or cosine transform.
2. Generate the ordered sequence of frequencies using the forward transform F .
3. Select the filter function G in the frequency domain and a cutoff frequency.
4. Convolute the sequence of frequency with the filter function G .
5. Generate the filtered signal in the time domain by applying the inverse DFT transform to the convoluted frequencies.



A diagram of the discrete Fourier filter

The most commonly used low-pass filter functions are known as the `sinc` and `sinc2` functions, which are defined as a rectangular function and triangular function, respectively. These functions are partially applied functions that are derived from a generic `convol` method. The simplest `sinc` function returns 1 for frequencies below a cutoff frequency, `fC`, and 0 if the frequency is higher:

```
val convol = (n: Int, f: Double, fC: Double) =>
    if( Math.pow(f, n) < fC) 1.0 else 0.0
val sinc = convol(1, _: Double, _:Double)
val sinc2 = convol(2, _: Double, _:Double)
val sinc4 = convol(4, _: Double, _:Double)
```

Partially applied functions versus partial functions

Partial functions and partially applied functions are not actually related.

A partial function f' is a function that is applied to a subset X' of the input space X . It does not execute all possible input values:



$$f: X \rightarrow Y \quad X' \subset X \quad f': X' \rightarrow Y$$

A partially applied function f'' is a function value for which the user supplies the value for one or more arguments. The projection reduces the dimension of the input space (X, Z) :

$$f: (X, Z) \rightarrow Y \quad f'': X \rightarrow Y$$

The `DFTFilter` class inherits from the `DFT` class in order to reuse the `fwrd` forward transform function. The `g` frequency domain function is an attribute of the filter. The `g` function takes the `fC` frequency cutoff value as the second argument (line 14). The two `sinc` and `sinc2` filters defined in the previous section are examples of filtering functions:

```
class DFTFilter[@specialized(Double) T <: AnyVal] (
    fC: Double,
    eps: Double)
    (g: (Double, Double) =>Double) (implicit f: T => Double)
extends DFT[T](eps) { //14

    override def |> : PartialFunction[U, Try[V]] = {
        case xt: U if( xt.size >= 2 ) => {
            fwrd(xt).map{ case(trf, freq) => { //15
                val cutOff = fC*freq.size
                val filtered = freq.zipWithIndex
                    .map{ case(x, n) => x*g(n, cutOff) } //16
                trf.transform(filtered, INVERSE).toVector } //17
            }
        }
    }
}
```

The filtering process follows three steps:

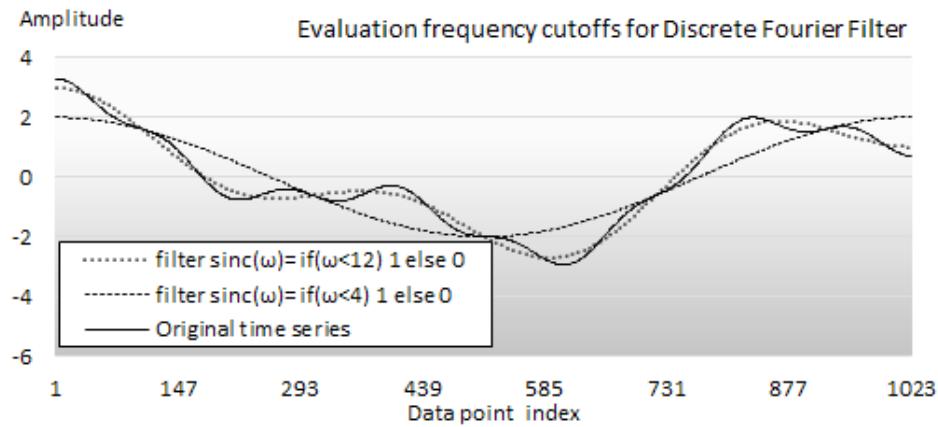
1. Computation of the `fwrd` discrete Fourier forward transformation (sine or cosine) (line 15).
2. Apply the filter function (formula M11) through a Scala `map` method (line 16).
3. Apply the inverse transform to the frequencies (line 17).

Let's evaluate the impact of the cutoff values on the filtered data. The implementation of the test program consists of loading the data from the file (line 19) and then invoking the `DFTFilter` of the `pfnDFTfilter` partial function (line 19):

```
import YahooFinancials._

val inputFile = s"$RESOURCE_PATH$symbol.csv"
val src = DataSource(input, false, true, 1)
val CUTOFF = 0.005
val pfnDFTfilter = DFTFilter[Double](CUTOFF)(sinc) |>
for {
    price <- src.get(adjClose) //18
    filtered <- pfnDFTfilter(price) //19
}
yield { /* ... */ }
```

Filtering out the noise is accomplished by selecting the cutoff value between any of the three harmonics with the respective frequencies of 2, 5, and 15. The original and the two filtered time series are plotted on the following graph:



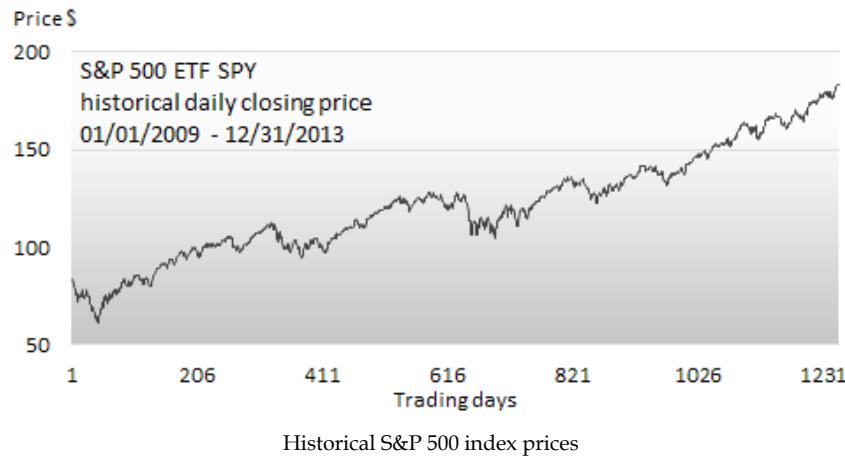
Plotting of the discrete Fourier filter-based smoothing

As you would expect, the low-pass filter with a cutoff value of 12 eliminates the noise with the highest frequencies. The filter with the cutoff value 4 cancels out the second harmonic (low-frequency noise), leaving out only the main trend cycle.

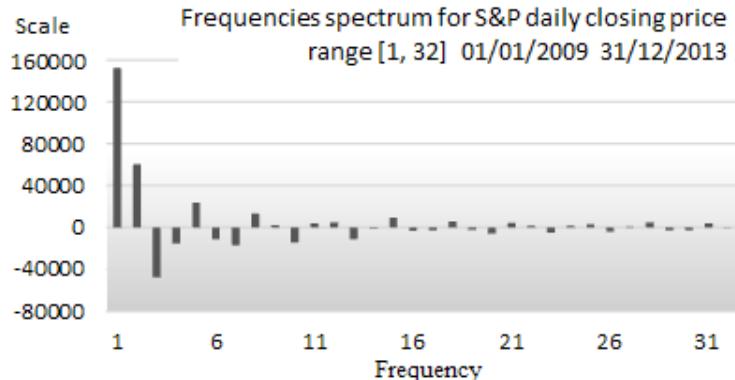
Detection of market cycles

Using the discrete Fourier transform to generate the frequencies spectrum of a periodical time series is easy. However, what about real-world signals such as the time series that represent the historical price of a stock?

The purpose of the next exercise is to detect, if any, the long term cycle(s) of the overall stock market by applying the discrete Fourier transform to the quote of the S&P 500 index between January 1, 2009 and December 31, 2013, as illustrated in the following graph:



The first step is to apply the DFT to extract a frequencies spectrum for the S&P 500 historical prices, as shown in the following graph, with the first 32 harmonics:



Frequencies spectrum for the historical S&P index

The frequency domain chart highlights some interesting characteristics regarding the S&P 500 historical prices:

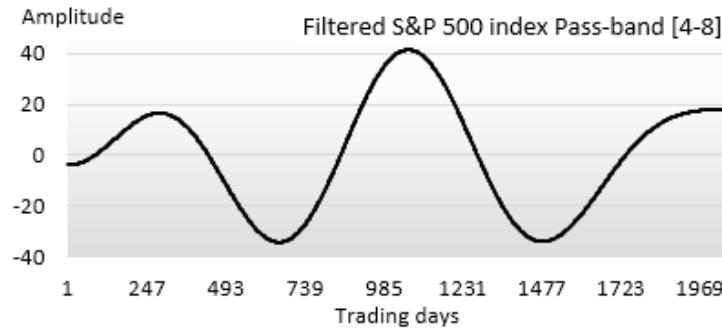
- Both positive and negative amplitudes are present, as you would expect in a time series with complex values. The cosine series contributes to the positive amplitudes while the sine series affects both positive and negative amplitudes, ($\cos(x+\pi) = \sin(x)$).
- The decay of the amplitude along the frequencies is steep enough to warrant further analysis beyond the first harmonic, which represents the main trend of the historical stock price. The next step is to apply a band-pass filter technique to the S&P 500 historical data in order to identify short-term trends with lower periodicity.

A low-pass filter is limited to reduce or cancel out the noise in the raw data. In this case, a band-pass filter using a range or window of frequencies is appropriate to isolate the frequency or the group of frequencies that characterize a specific cycle. The `sinc` function, which was introduced in the previous section to implement a low-pass filter, is modified to enforce the band-pass filter within a window, $[w_1, w_2]$, as follows:

```
def sinc(f: Double, w: (Double, Double)): Double =
  if(f > w._1 && f < w._2) 1.0 else 0.0
```

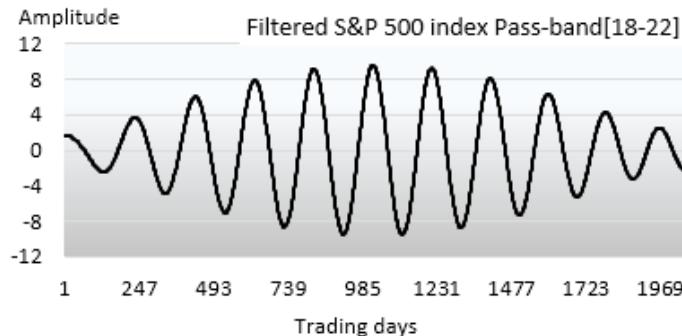
Data Preprocessing

Let's define a DFT-based band-pass filter with a window of width 4, $w=(i, i+4)$, with i ranging between 2 and 20. Applying the window [4, 8] isolates the impact of the second harmonic on the price. As we eliminate the main upward trend with frequencies less than 4, all the filtered data varies within a short range relative to the main trend. The following graph shows the output of this filter:



The output of a band-pass DFT filter range 4-8 on the historical S&P index

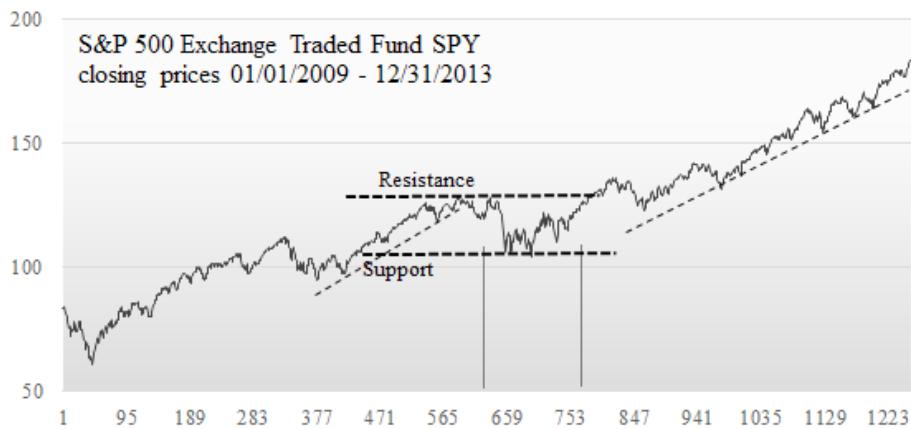
In this case, we filter the S&P 500 index around the third group of harmonics with frequencies ranging from 18 to 22; the signal is converted into a familiar sinusoidal function, as shown here:



The output of a band-pass DFT filter range 18-22 on the historical S&P index

There is a possible rational explanation for the shape of the S&P 500 data filtered by a band-pass filter with a frequency of 20, as illustrated in the previous graph. The S&P 500 historical data plot shows that the frequency of the fluctuation in the middle of the uptrend (trading sessions 620 to 770) increases significantly.

This phenomenon can be explained by the fact that the S&P 500 index reaches a resistance level around the trading session 545 when the existing uptrend breaks. A tug of war starts between the bulls, betting the market nudges higher, and the bears, who are expecting a correction. The back and forth between the traders ends when the S&P 500 index breaks through its resistance and resumes a strong uptrend characterized by a high amplitude low frequency, as shown in the following graph:



An illustration of support and resistance levels for the historical S&P 500 index prices

One of the limitations of using the discrete Fourier-based filters to clean up data is that it requires the data scientist to extract the frequencies spectrum and modify the filter on a regular basis, as he or she is never sure that the most recent batch of data does not introduce noise with a different frequency. The Kalman filter addresses this limitation.

The discrete Kalman filter

The Kalman filter is a mathematical model that provides an accurate and recursive computation approach to estimate the previous states and predict the future states of a process for which some variables may be unknown. R.E. Kalman introduced it in the early 60s to model dynamics systems and predict a trajectory in aerospace [3:10]. Today, the Kalman filter is used to discover a relationship between two observed variables that may or may not be associated with other hidden variables. In this respect, the Kalman filter shares some similarities with the Hidden Markov models, as described in the *The hidden Markov model* section in *Chapter 7, Sequential Data Models* [3:11].

The Kalman filter is used as:

- A predictor of the next data point from the current observation
- A filter that weeds out noise by processing the last two observations
- A smoothing model that identifies trends from a history of observations

Smoothing versus filtering

 Smoothing is an operation that removes high-frequency fluctuations from a time series or signal. Filtering consists of selecting a range of frequencies to process the data. In this regard, smoothing is somewhat similar to low-pass filtering. The only difference is that a low-pass filter is usually implemented through linear methods.

Conceptually, the Kalman filter estimates the state of a system from noisy observations. The Kalman filter has two characteristics:

- **Recursive:** A new state is predicted and corrected using the input of a previous state
- **Optimal:** This is an optimal estimator because it minimizes the mean square error of the estimated parameters (against actual values)

The Kalman filter is one of the stochastic models that are used in adaptive control [3:12].

Kalman and nonlinear systems

 The Kalman filter estimates the internal state of a linear dynamic system. However, it can be extended to a nonlinear state space model using linear or quadratic approximation functions. These filters are known as, you guessed it, **Extended Kalman Filters (EKF)**, the theory of which is beyond the scope of this book.

The following section is dedicated to discrete Kalman filters for linear systems, as applied to financial engineering. A continuous signal can be converted to a time series using the Nyquist frequency.

The state space estimation

The Kalman filter model consists of two core elements of a dynamic system: a process that generates data and a measurement that collects data. These elements are referred to as the state space model. Mathematically speaking, the state space model consists of two equations:

- **The transition equation:** This describes the dynamics of the system, including the unobserved variables
- **The measurement equation:** This describes the relationship between the observed and unobserved variables

The transition equation

Let's consider a system with a linear state x_t of n variables and a control input vector u_t . The prediction of the state at time t is computed by a linear stochastic equation (**M12**):

$$x_t = A_t \cdot x_{t-1} + B_t \cdot u_t + w_t$$

- A_t is the square matrix of dimension n that represents the transition from a state x_{t-1} at $t-1$ to a state x_t at t . The matrix is intrinsic to the dynamic system under consideration.
- B_t is a n by n matrix that describes the control input model (an external action on the system or model). It is applied to the control vector, u_t .
- w_t represents the noise generated by the system, or from a probabilistic point of view, it represents the uncertainty on the model. It is known as the process white noise.

The control input vector represents the external input (or control) to the state of the system. Most systems, including our financial example later in this chapter, have no external input to the state of the model.



A white and Gaussian noise

A white noise is a Gaussian noise, following a normal distribution with zero mean.

The measurement equation

The measurement of m values z_t of the state of the system is defined by the following equation (M13):

$$z_t = H_t \cdot x_t + v_t$$

- H_t is a m by n matrix that models the dependency of the measurement to the state of the system.
- v_t is the white noise introduced by the measuring devices. Similarly to the process noise, v follows a Gaussian distribution with zero mean and a variance R , known as the **measurement noise covariance**.

The time dependency model

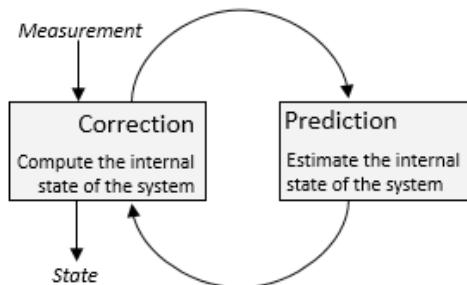
We cannot assume that the parameters of the generalized discrete Kalman filter, such as the state transition A_t , control input B_t and observation matrices (or measurement dependency) H_t are independent of time. However, these parameters are constant in most practical applications.

The recursive algorithm

The set of equations for the discrete Kalman filter is implemented as a recursive computation with two distinct steps:

- The algorithm uses the transition equations to estimate the next observation
- The estimation is created with the actual measurement for this observation

The recursion is visualized in the following diagram:



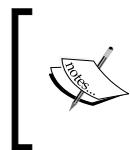
An overview diagram of the recursive Kalman algorithm

Let's illustrate the prediction and correction phases in the context of filtering financial data, in a manner similar to the moving average and Fourier transform. The objective is to extract the trend and the transitory component of the yield of the 10-year Treasury bond. The Kalman filter is particularly suitable for the analysis of interest rates for two reasons:

- Yields are the results of multiple factors, some of which are not directly observable.
- Yields are influenced by the policy of the Federal Reserve that can be easily modeled by the control matrix.

The 10-year Treasury bond has a higher trading volume than bonds with longer maturity, making trends in interest rates a bit more reliable [3:13].

Applying the Kalman filter to clean raw data requires you to define a model that encompasses both observed and non-observed states. In the case of the trend analysis, we can safely create our model with a two-variable state: the current yield x_t and the previous yield x_{t-1} .



The state of dynamic systems

The term "state" refers to the state of the dynamic system under consideration and not the state of the execution of the algorithm.

This implementation of the Kalman filter uses the Apache Commons Math library. Therefore, we need to specify the implicit conversion from our primitives, introduced in the *Primitives and implicits* section in *Chapter 1, Getting Started*, to the `RealMatrix`, `RealVector`, `Array2DRowRealMatrix`, and `ArrayRealVector` Apache Commons Math types:

```
implicit def double2RealMatrix(x: DblMatrix): RealMatrix =
  new Array2DRowRealMatrix(x)
implicit def double2RealRow(x: DblVector): RealMatrix =
  new Array2DRowRealMatrix(x)
implicit def double2RealVector(x: DblVector): RealVector =
  new ArrayRealVector(x)
```

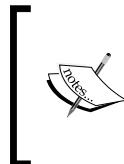
The client code has to import the implicit conversion functions within its scope.

The Kalman model assumes that the process and measurement noise follows a Gaussian distribution, also known as a white noise. For the sake of maintainability, the generation of the white noise is encapsulated in the `QRNoise` class with the following arguments (line 1):

- `qr`: This is the tuple of scale factors for the process noise matrix Q and the measurement noise R
- `profile`: This is the noise profile with the normal distribution as default

The two `noiseQ` and `noiseR` methods generate an array of two independent white noise elements (line 2):

```
val normal = Stats.normal(_)
class QRNoise(qr: DblPair, profile: Double=>Double = normal){ //1
    def q = profile(qr._1)
    def r = profile(qr._2)
    lazy val noiseQ = Array[Double](q, q)    //2
    lazy val noiseR = Array[Double](r, r)
}
```



Experimenting with a noise profile

Although the discrete Kalman filter assumes that the noise profile follows a normal distribution, the `QRNoise` class allows the user to experiment with different noise profiles.

The easiest approach to manage the matrices and vectors used in the recursion is to define them as arguments of a `kalmanConfig` configuration class. The arguments of the configuration follow the naming convention defined in the mathematical formulas: A is the state transition matrix, B is the control matrix, H is the matrix of observations that define the dependencies between the measurement and system state, and P is the covariance error matrix:

```
case class KalmanConfig(A: DblMatrix, B: DblMatrix,
                        H: DblMatrix, P: DblMatrix)
```

Let's implement the Kalman filter as a `DKalman` transformation of the `ETransform` type on a time series with a predefined `KalmanConfig` configuration:

```
class DKalman(config: KalmanConfig)(implicit qrNoise: QRNoise)
  extends ETransform[KalmanConfig](config) {
  type U = Vector[DblPair]    //3
  type V = Vector[DblPair]    //4
  type KRState = (KalmanFilter, RealVector) //5
  override def |> : PartialFunction[U, Try[V]] =
  ...
}
```

As with any explicit data transformation, we need to specify the `U` and `V` types (lines 3 and 4), which are identical. The Kalman filter does not alter the structure of the data, it alters only the values. We define an internal state for the `KRState` Kalman computation by creating a tuple of two `KalmanFilter` and `RealVector` (line 5) Apache Commons Math types.

The key elements of the filter are now in place and it's time to implement the prediction-correction cycle portion of the Kalman algorithm.

Prediction

The prediction phase consists of estimating the x state (yield of the Treasury bond) using the transition equation. We assume that the Federal Reserve has no material effect on the interest rates, making the B control input matrix null. The transition equation can be easily resolved using simple operations on matrices:

$$\begin{bmatrix} \hat{x}_t \\ \hat{x}_{t-1} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} x_t \\ x_{t-1} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} u_t \\ u_{t-1} \end{bmatrix} + \begin{bmatrix} w_t \\ w_{t-1} \end{bmatrix}$$

Visualization of the transition equation of the Kalman filter

The purpose of this exercise is to evaluate the impact of the different parameters of the transition matrix A in terms of smoothing.

The control input matrix B

In this example, the control matrix B is null because there is no known, deterministic external action on the yield of the 10-year Treasury bond. However, the yield can be affected by unknown parameters that we represent as hidden variables. For example, the matrix B can be used to model the decision of the Federal Reserve regarding asset purchases and federal fund rates.

The mathematics behind the Kalman filter presented as a reference to the implementation in Scala, use the same notation for matrices and vectors. It is absolutely not a prerequisite to understand the Kalman filter and its implementation in the next section. If you have a natural inclination toward linear algebra, the following describe the two equations for the prediction step.

The prediction step

M14: The prediction of the state at time t is computed by extrapolating the state estimate:

$$\hat{x}_t' = A_t \cdot \hat{x}_{t-1} + B_t \cdot u_t$$



- A is the square matrix of dimension n that represents the transition from state x at $t-1$ to state x at time t
- \hat{x}'_t is the predicted state of the system based on the current state and the model A
- B is the vector of n dimension that describes the input to the state

M15: The mean square error matrix, P , which is to be minimized, is updated using the following formula:

$$P_t' = A_t \cdot P_{t-1} \cdot A_t^T + Q_t$$

- A^T is the transpose of the state transition matrix
- Q is the process white noise described as a Gaussian distribution with a zero mean and a variance Q , known as the noise covariance

The state transition matrix is implemented using the matrix and vector classes included in the Apache Commons Math library. The types of matrices and vectors are automatically converted into the `RealMatrix` and `RealVector` classes.

The implementation of the equation **M14** is as follows:

```
x = A.operate(x).add(qrNoise.create(0.03, 0.1))
```

The new state is predicted (or estimated), and then used as an input to the correction step.

Correction

The second step of the recursive Kalman algorithm is the correction of the estimated yield of the 10-year Treasury bond with the actual yield. In this example, the white noise of the measurement is negligible. The measurement equation is simple because the state is represented by the current and previous yield and their measurement, z :

$$\begin{bmatrix} z_t \\ z_{t-1} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \cdot \begin{bmatrix} \hat{x}_t \\ \hat{x}_{t-1} \end{bmatrix} + \begin{bmatrix} v_t \\ v_{t-1} \end{bmatrix}$$

Visualization of the measurement equation of the Kalman filter

The sequence of mathematical equations of the correction phase consists of updating the estimation of the state x using the actual values z and computing the Kalman gain, K .

The correction step

M16: The state of the system x is estimated from the actual measurement z using the following formula:

$$\hat{x}_t = \hat{x}_t' + K_t(z_t - H_t \cdot \hat{x}_t')$$

- r_t is the residual between the predicted measurement and the actual measured values
- K_t is the Kalman gain for the correction factor

M17: The Kalman gain is computed as:

$$K_t = P_t' \cdot H_t^T (H_t \cdot P_t' \cdot H_t^T + R_t)^{-1}$$

Here, H^T is the matrix transpose of H and P_t' is the estimate of the error covariance.



Kalman smoothing

It is time to put our knowledge of the transition and measurement equations to test. The Apache Commons Math library defines the two `DefaultProcessModel` and `DefaultMeasurementModel` classes to encapsulate the components of the matrices and vectors. The historical values for the yield of the 10-year Treasury bond are loaded through the `DataSource` method and mapped to the smoothed series that is the output of the filter:

```
override def |> : PartialFunction[U, Try[V]] = {
  case xt: U if( !xt.isEmpty) => Try(
    xt.map { case(current, prev) => {
      val models = initialize(current, prev) //6
      val nState = newState(models) //7
      (nState(0), nState(1)) //8
    }})
  )
}
```

The data transformation for the Kalman filter initializes the process and measurement model for each data point in the private `initialize` (line 6) method, updates the state using the transition and correction equations iteratively in the `newState` method (line 7), and returns the filtered series of pair values (line 8).

Exception handling

The code to catch and process exceptions thrown by the Apache Commons Math library is omitted as the standard practice in the book. As far as the execution of the Kalman filter is concerned, the following exceptions have to be handled:

- `NonSquareMatrixException`
- `DimensionMismatchException`
- `MatrixDimensionMismatchException`

The `initialize` method encapsulates the initialization of the `pModel` process model (line 9) and the `mModel` measurement (observations dependencies) model (line 10), as defined in the Apache Commons Math library:

```
def initialize(current: Double, prev: Double): KRState = {
    val pModel = new DefaultProcessModel(config.A, config.B,
        Q, input, config.P) //9
    val mModel = new DefaultMeasurementModel(config.H, R) //10
    val in = Array[Double](current, prev)
    (new KalmanFilter(pModel, mModel), new ArrayRealVector(in))
}
```

The exceptions thrown by the Apache Commons Math API are caught and processed through the `Try` monad. The iterative prediction and correction of the smoothed yields of 10-year Treasury bond is implemented by the `newState` method. The method iterates through the following steps:

1. Estimate the new values of the state by invoking the Apache Commons Math `KalmanFilter.predict` method that implements the **M14** formula (line 11).
2. Apply the **M12** formula to the new state x at time t (line 12).
3. Compute the measured value z at time t using the **M13** formula (line 13).
4. Invoke the Apache Commons Math `KalmanFilter.correct` method to implement the **M16** formula (line 14).
5. Return the estimated value of the state x by invoking the Apache Commons Math `KalmanFilter.getStateEstimation` method (line 15).

The code will be as follows:

```
def newState(state: KRState): DblArray = {
    state._1.predict //11
    val x = config.A.operate(state._2).add(qrNoise.noisyQ) //12
    val z = config.H.operate(x).add(qrNoise.noisyR) //13
    state._1.correct(z) //14
    state._1.getStateEstimation //15
}
```

The exit condition

In the code snippet for the `newState` method, the iteration for specific data points exits when the maximum number of iterations is reached. A more elaborate implementation consists of either evaluating the matrix P at each iteration or estimation converged within a predefined range.

Fixed lag smoothing

So far, we have studied the Kalman filtering algorithm. We need to adapt it to the smoothing of a time series. The **fixed lag smoothing** technique consists of backward correcting previous data points, taking into account the latest actual value.

A N-lag smoother defines the input as a vector $X = \{x_{t-N-1}, x_{t-N-2}, \dots, x_t\}$ for which the value x_{t-N-j} is corrected taking into account the current value of x_t .

The strategy is quite similar to the hidden Markov model forward and backward passes (refer to the *Evaluation – CF-1* section under *The hidden Markov model* in *Chapter 7, Sequential Data Models*).

Complex strategies for lag smoothing

There are numerous formulas or methodologies to implement an accurate fixed lag smoothing strategy and correct the predicted observations. Such strategies are beyond the scope of this book.

Experimentation

The objective is to smoothen the yield of the 10-year Treasury bond using a **two-step lag smoothing** algorithm.

The two-step lag smoothing



M18: The two-step lag smoothing algorithm for state S_t using a single smoothing factor α is defined as:

$$S_t = [x_{t+1}, x_t]^T \quad \text{with} \quad \begin{vmatrix} x_{t+1} \\ x_t \end{vmatrix} = \begin{vmatrix} \alpha & 1 - \alpha \\ 1 & 0 \end{vmatrix} \cdot \begin{vmatrix} x_t \\ x_{t-1} \end{vmatrix}$$

The state equation updates the values of the state $[x_t, x_{t-1}]$ using the previous state $[x_{t-1}, x_{t-2}]$, where x_t represents the yield of the 10-year Treasury bond at time t . This is accomplished by shifting the values of the original time series $\{x_0, \dots, x_{n-1}\}$ by 1 using the drop method, $X_1 = \{x_1, \dots, x_{n-1}\}$, creating a copy of the original time series without the last element $X_2 = \{x_0, \dots, x_{n-2}\}$, and zipping X_1 and X_2 . This process is implemented by the `zipWithShift` method, which is introduced in the first section of the chapter.

The resulting sequence of a state vector $S_k = [x_k, x_{k-1}]^T$ is processed by the Kalman algorithm, as shown in the following code:

```
Import YahooFinancials._
val RESOURCE_DIR = "resources/data/chap3/"
implicit val qrNoise = new QRNoise((0.7, 0.3)) //16

val H: DblMatrix = ((0.9, 0.0), (0.0, 0.1)) //17
val P0: DblMatrix = ((0.4, 0.3), (0.5, 0.4)) //18
val ALPHA1 = 0.5; val ALPHA2 = 0.8
val src = DataSource(s"$${RESOURCE_DIR}$$${symbol}.csv", false)

(src.get(adjClose)).map(zt => { //19
    twoStepLagSmoothen(zt, ALPHA1) //20
    twoStepLagSmoothen(zt, ALPHA2)
})
```

An implicit noise instance

The noise for the process and measurement is defined as an implicit argument to the `DKalman` Kalman filter for the following two reasons:



- The profile of the noise is specific to the process or system under evaluation and its measurement; it is independent of the `A`, `B`, and `H` Kalman configuration parameters. Therefore, it cannot be a member of the `KalmanConfig` class.
- The same noise characteristics should be shared with other alternative filtering techniques, if needed.

The white noise for the process and measurement is initialized implicitly with the `qrNoise` value (line 16). The code initializes the matrices `H` of the measurement dependencies on the state (line 17) and `P0` that contains the initial covariance errors (line 18). The input data is extracted from a CSV file that contains the daily Yahoo financial data (line 19). Finally, the method executes the `twoStepLagSmoothen` two-step lag smoothing algorithm with two different `ALPHA1` and `ALPHA2` alpha parameter values (line 20).

Let's take a look at the `twoStepLagSmoothen` method:

```
def twoStepLagSmoothen(zSeries: DblVector, alpha: Double): Int = {
    val A: DblMatrix = ((alpha, 1.0-alpha), (1.0, 0.0)) //21
    val xt = zipWithShift(1) //22
    val pfnKalman = DKalman(A, H, P0) |> //23
    pfnKalman(xt).map(filtered => //24
        display(zSeries, filtered.map(_._1), alpha)
    )
}
```

The `twoStepLagSmoothen` method takes two arguments:

- A `zSeries` single variable time series
- A `alpha` state transition parameter

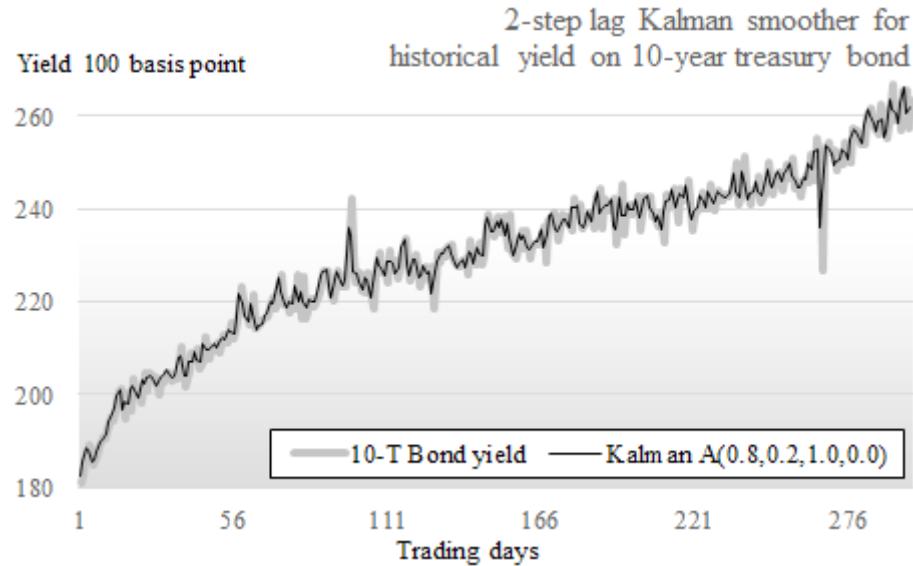
It initializes the state transition matrix `A` using the `alpha` exponential moving average decay parameter (line 21). It creates the two-step lag time series, `xt`, using the `zipWithShift` method (line 22). It extracts the `pfnKalman` partial function (line 23), processes, and finally, displays the two-step lag time series (line 24).

Modeling state transition and noise

The state transition and the noise related to the process have to be selected carefully. The resolution of the state equations relies on the **Cholesky** (QR) decomposition, which requires a nonnegative definite matrix. The implementation in the Apache Commons Math library throws a `NonPositiveDefiniteMatrixException` exception if the principle is violated.

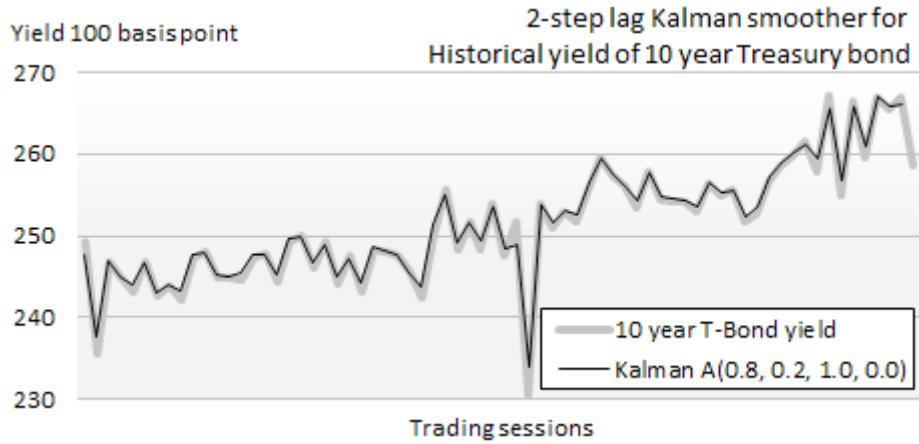
Data Preprocessing

The smoothed yield is plotted along the raw data as follows:



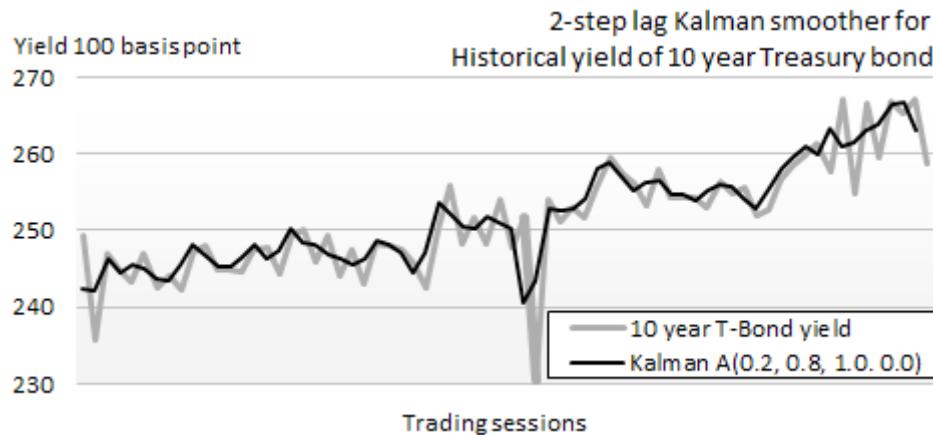
The output of the Kalman filter for the 10-year Treasury-Bond historical prices

The Kalman filter is able to smooth the historical yield of the 10-year Treasury bond while preserving the spikes and lower frequency noise. Let's analyze the data for a shorter period during which the noise is the strongest, between the 190th and the 275th trading days:



The output of the Kalman filter for the 10-year Treasury bond prices 0.8-02

The high frequency noise has been significantly reduced without cancelling the actual spikes. The distribution (0.8, 0.2) takes into consideration the previous state and favors the predicted value. Contrarily, a run with a state transition matrix A [0.2, 0.8, 0.0, 1.0] that favors the latest measurement will preserve the noise, as seen in the following graph:



The output of the Kalman filter for the 10-year Treasury bond price 0.2-0.8

Benefits and drawbacks

The Kalman filter is a very useful and powerful tool used to help you understand the distribution of the noise between the process and observation. Contrary to the low or band-pass filters based on the discrete Fourier transform, the Kalman filter does not require the computation of the frequencies spectrum or assume the range of frequencies of the noise.

However, the linear discrete Kalman filter has its limitations, which are as follows:

- The noise generated by both the process and the measurement has to be Gaussian. Processes with non-Gaussian noise can be modeled with techniques such as a Gaussian sum filter or adaptive Gaussian mixture [3:14].
- It requires that the underlying process is linear. However, researchers have been able to formulate extensions to the Kalman filter, known as the **extended Kalman filter (EKF)**, to filter signals from nonlinear dynamic systems, at the cost of significant computational complexity.

The continuous-time Kalman filter



The Kalman filter is not restricted to dynamic systems with discrete states x . The case of continuous state-time is handled by modifying the state transition equation, so the estimated state is computed as the derivative dx/dt .

Alternative preprocessing techniques

For the sake of space and your time, this chapter introduced and applied three filtering and smoothing classes of algorithms. Moving averages, Fourier series, and the Kalman filter are far from being the only techniques used in cleaning raw data. The alternative techniques can be classified into the following categories:

- Autoregressive models that encompass **Autoregressive Moving Average (ARMA)**, **Autoregressive Integrated Moving Average (ARIMA)**, **generalized autoregressive conditional heteroskedasticity (GARCH)**, and Box-Jenkins that relies on some form of autocorrelation function.
- **Curve-fitting** algorithms that include the polynomial and geometric fit with the ordinary least squares method, nonlinear least squares using the **Levenberg-Marquardt** optimizer, and probability distribution fitting.
- Nonlinear dynamic systems with a Gaussian noise such as a **particle filter**.
- Hidden Markov models, as described in the *The hidden Markov model* section in *Chapter 7, Sequential Data Models*.

Summary

This completes the overview of the most commonly used data filtering and smoothing techniques. There are other types of data preprocessing algorithms such as normalization, analysis, and reduction of variance; the identification of missing values is also essential to avoid the **garbage-in garbage-out** conundrum that plagues so many projects that use machine learning for regression or classification.

Scala can be effectively used to make the code understandable and avoid cluttering methods with unnecessary arguments.

The three techniques presented in this chapter, from the simplest moving averages and Fourier transform to the more elaborate Kalman filter, go a long way in setting up data for the concepts introduced in the next chapter: unsupervised learning and more specifically, clustering.

4

Unsupervised Learning

Labeling a set of observations for classification or regression can be a daunting task, especially in the case of a large feature set. In some cases, labeled observations are either unavailable or not possible to create. In an attempt to extract some hidden associations or structures from observations, the data scientist relies on unsupervised learning techniques to detect patterns or similarity in data.

The goal of unsupervised learning is to discover patterns of regularities and irregularities in a set of observations. These techniques are also applied in reducing the solution or feature space.

There are numerous unsupervised algorithms; some are more appropriate to handle dependent features while others generate affinity groups in the case of hidden features [4:1]. In this chapter, you will learn three of the most common unsupervised learning algorithms:

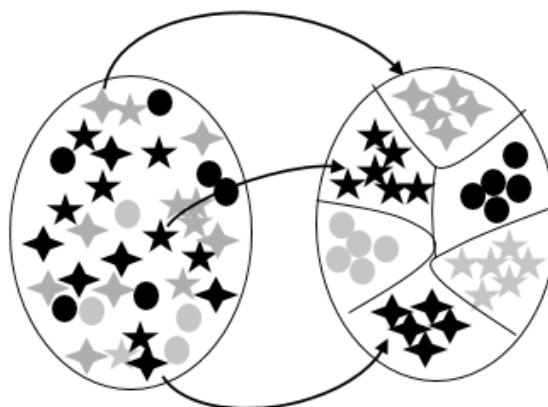
- **K-means:** This used for clustering observed features
- **Expectation-maximization (EM):** This is used for clustering observed and latent features
- **Principal Components Analysis (PCA):** This is used to reduce the dimension of the model

Any of these algorithms can be applied to technical analysis or fundamental analysis. Fundamental analysis of financial ratios and technical analysis of price movements is discussed in the *Technical analysis* section under *Finances 101* in the *Appendix A, Basic Concepts*. The K-means algorithm is fully implemented in Scala while expectation-maximization and Principal Components Analysis leverage the Apache Commons Math library.

The chapter concludes with a brief overview of dimension reduction techniques for non-linear models.

Clustering

Problems involving a large number of features for large datasets become quickly intractable, and it is quite difficult to evaluate the independence between features. Any computation that requires some level of optimization and, at a minimum, the computation of first order derivatives requires a significant amount of computing power to manipulate high-dimension matrices. As with many engineering fields, a divide-and-conquer approach to classifying very large datasets is quite effective. The objective is to reduce very large sets of observations into a small group of observations that share some common attributes.



Visualization of data clustering

This approach is known as vector quantization. Vector quantization is a method that divides a set of observations into groups of similar size. The main benefit of vector quantization is that the analysis using a representative of each group is far simpler than an analysis of the entire dataset [4:2].

Clustering, also known as **cluster analysis**, is a form of vector quantization that relies on a concept of distance or similarity to generate groups known as clusters.

 **Learning vector quantization (LVQ)**
Vector quantization should not be confused with **learning vector quantization**; learning vector quantization is a special case of artificial neural networks that relies on a winner-take-all learning strategy to compress signals, images, or videos.

This chapter introduces two of the most commonly applied clustering algorithms:

- **K-means:** This is used for quantitative types and minimizes the total error (known as the reconstruction error) given the number of clusters and the distance formula.
- **Expectation-maximization (EM):** This is a two-step probabilistic approach that maximizes the likelihood estimates of a set of parameters. EM is particularly suitable for handling missing data.

K-means clustering

K-means is a popular clustering algorithm that can be implemented either iteratively or recursively. The representative of each cluster is computed as the center of the cluster, known as the **centroid**. The similarity between observations within a single cluster relies on the concept of distance (or similarity) between observations.

Measuring similarity

There are many ways to measure the similarity between observations. The most appropriate measure has to be intuitive and avoid computational complexity. This section reviews three similarity measures:

- The Manhattan distance
- The Euclidean distance
- The normalized inner or dot product

The Manhattan distance is defined by the absolute distance between two variables or vectors, $\{x_i\}$ and $\{y_i\}$, of the same size (M1):

$$d(x, y) = \sum |x_i - y_i|$$

The implementation is generic enough to compute the distance between two vectors of elements of different types as long as an implicit conversion between each of these types to the `Double` values is already defined, as follows:

```
def manhattan[T <: AnyVal, U <: AnyVal] (
  x: Array[T],
  y: Array[U])(implicit f: T => Double): Double =
  (x, y).zipped.map{case (u, v) => Math.abs(u-v)}.sum
```

The ubiquitous Euclidean distance between two vectors, $\{x_i\}$ and $\{y_i\}$, of the same size is defined by the following formula (M2):

$$d(x, y) = \sum (x_i - y_i)^2$$

The code will be as follows:

```
def euclidean[T <: AnyVal, U <: AnyVal] (
    x: Array[T],
    y: Array[U])(implicit f: T => Double): Double =
  Math.sqrt((x,y).zipped.map{case (u,v) => u-v}.map(sqr(_)).sum)
```

The normalized inner product or cosine distance between two vectors, $\{x_i\}$ and $\{y_i\}$, is defined by the following formula (M3):

$$d(x, y) = \frac{\sum x_i y_i}{(\sum x_i^2 \sum y_i^2)^{1/2}}$$

In this implementation, the computation of the dot product and the norms for each dataset is done simultaneously using the tuple within the `fold` method:

```
def cosine[T <: AnyVal, U <: AnyVal] (
    x: Array[T],
    y: Array[U])(implicit f: T => Double): Double = {
  val norms = (x,y).zipped
    .map{ case (u,v) => Array[Double](u*v, u*u, v*v) }
    ./:(Array.fill(3)(0.0))((s, t) => s ++ t)
  norms(0)/Math.sqrt(norms(1)*norms(2))
}
```

Performance of zip and zipped

The scalar product of two vectors is one of the most common operations. It is tempting to implement the dot product using the generic `zip` method:



```
def dot(x:Array[Double], y:Array[Double]):  
  Array[Double] =  
    x.zip(y).map{case(x, y) => f(x,y) }
```

A functional alternative is to use the `Tuple2.zipped` method:

```
def dot(x:Array[Double], y:Array[Double]):  
  Array[Double] =  
    (x, y).zipped map (_ * _)
```

If readability is not a primary issue, you can always implement the `dot` method with a `while` loop, which prevents you from using the ubiquitous `while` loop.

Defining the algorithm

The main advantage of the K-means algorithm (and the reason for its popularity) is its simplicity [4:3].

K-means objective



M4: Let's consider K clusters, $\{C_k\}$ with means or centroids, $\{m_k\}$. The K-means algorithm is indeed an optimization problem whose objective is to minimize the reconstruction or total error defined as the total sum of the distance:

$$\min_{c_k} \sum_1^K \sum_{x_i \in C_k} d(x_i, m_k)$$

The four steps of the K-means algorithm are as follows:

1. Cluster configuration (initializing the centroids or means m_k of the K clusters).
2. Cluster assignment (assigning observations to the nearest cluster given the centroids m_k).

3. Error minimization (computing the total reconstruction error):
 1. Compute centroids m_k that minimize the total reconstruction error for the current assignment.
 2. Reassign the observations given the new centroids m_k .
 3. Repeat the computation of the total reconstruction error until no observations are reassigned.
4. Classification of a new observation by assigning the observation to the closest cluster.

We need to define the components of the K-means in Scala before implementing the algorithm.

Step 1 – cluster configuration

Let's create the two main components of the K-means algorithms: clusters of observations and the implementation of the K-means algorithm.

Defining clusters

The first step is to define a cluster. A cluster is defined by the following parameters:

- The centroid (`center`) (line 1)
- The indices of the observations that belong to this cluster (`members`) (line 2)

The code will be as follows:

```
class Cluster[T <: AnyVal] (val center: DblArray)
    (implicit f: T => Double) { //1
  type DistanceFunc[T] = (DblArray, Array[T]) => Double
  val members = new ListBuffer[Int] //2
  def moveCenter(xt: XSeries[T]): Cluster[T]
  ...
}
```

The cluster is responsible for managing its members (data points) at any point of the iterative computation of the K-means algorithm. It is assumed that a cluster will never contain the same data points twice. The two key methods in the `Cluster` class are as follows:

- `moveCenter`: This recomputes the centroid of a cluster
- `stdDev`: This computes the standard deviation of the distance between all the observation members and the centroid

The constructor of the `Cluster` class is implemented by the `apply` method in the companion object. For convenience, refer to the *Class constructor template* section in the *Appendix A, Basic Concepts*:

```
object Cluster {
    def apply[T <: AnyVal](center: DblArray)
        (implicit f: T => Double): Cluster[T] =
        new Cluster[T](center)
}
```

Let's take a look at the `moveCenter` method. It creates a new cluster with the existing members and a new centroid. The computation of the values of the centroid requires the transposition of the matrix of observations by features into a matrix of feature by observations (line 3). The new centroid is computed by normalizing the sum of each feature across all the observations by the number of data points (line 4):

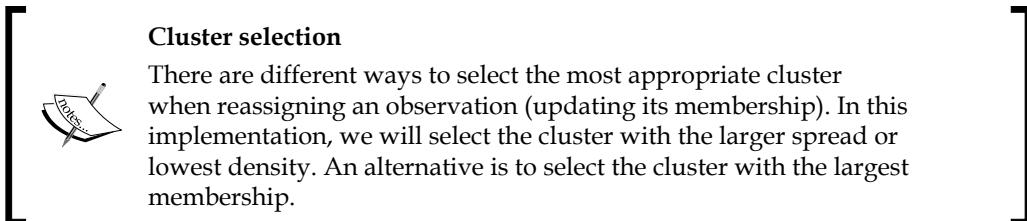
```
def moveCenter(xt: XVSeries[T])
    (implicit m: Manifest[T], num: Numeric[T])
    : Cluster[T] = {
    val sum = transpose(members.map( xt(_)).toList)
        .map(_.sum) //3
    Cluster[T](sum.map( _ / members.size).toArray) //4
}
```

The `stdDev` method computes the standard deviation of all the observations contained in the cluster relative to its center. The distance value between each member and the centroid is extracted through a map invocation (line 5). It is then loaded into a statistics instance to compute the standard deviation (line 6). The function to compute the distance between the center and an observation is an argument of the method. The default distance is euclidean:

```
def stdDev(xt: XVSeries[T], distance: DistanceFunc): Double = {
    val ts = members.map(xt(_)).map(distance(center,_)) //5
    Stats[Double](ts).stdDev //6
}
```

Cluster selection

There are different ways to select the most appropriate cluster when reassigning an observation (updating its membership). In this implementation, we will select the cluster with the larger spread or lowest density. An alternative is to select the cluster with the largest membership.



Initializing clusters

The initialization of the cluster centroids is important to ensure fast convergence of the K-means algorithm. Solutions range from the simple random generation of centroids to the application of genetic algorithms to evaluate the fitness of centroid candidates. We selected an efficient and fast initialization algorithm developed by M. Agha and W. Ashour [4:4].

The steps of the initialization are as follows:

1. Compute the standard deviation of the set of observations.
2. Compute the index of the feature $\{x_{k,0}, x_{k,1} \dots x_{k,n}\}$ with the maximum standard deviation.
3. Rank the observations by their increasing value of standard deviation for the dimension k .
4. Divide the ranked observations set equally into K sets $\{S_m\}$.
5. Find the median value's $size(S_m)/2$.
6. Use the resulting observations as initial centroids.

Let's deconstruct the implementation of the Agha-Ashour algorithm in the `initialize` method:

```
def initialize(xt: U): V = {
    val stats = statistics(xt)    //7
    val maxSDevVar = Range(0, stats.size)  //8
        .maxBy(stats(_).stdDev)

    val rankedObs = xt.zipWithIndex
        .map{case (x, n) => (x(maxSDevVar), n)}
        .sortWith(_._1 < _._1)  //9

    val halfSegSize = ((rankedObs.size>>1)/_config.K)
        .floor.toInt //10
    val centroids = rankedObs
        .filter(isContained(_, halfSegSize, rankedObs.size))
        .map{ case(x, n) => xt(n)} //11
    centroids.aggregate(List[Cluster[T]]())((xs, c) =>
        Cluster[T](c) :: xs, _ ::: _) //12
}
```

The `statistics` method on time series of the `xvseries` type is defined in the *Time series in Scala* section in *Chapter 3, Data Preprocessing* (line 7). The dimension (or feature) with the `maxSDevVar` maximum variance or standard deviation is computed using the `maxBy` method on a `stats` instance (line 8). Then, the observations are ranked by the increasing value of the `rankedObs` standard deviation (line 9).

The ordered sequence of observations is then broken into the `xt.size/_config.K` segments (line 10) and the indices of the centroids are selected as the midpoint (or median) observations of those segments using the `isContained` filtering condition (line 11):

```
def isContained(t: (T, Int), hSz: Int, dim: Int): Boolean =
  (t._2 % hSz == 0) && (t._2 % (hSz << 1) != 0)
```

Finally, the list of clusters is generated using an aggregate call on the set of centroids (line 12).

Step 2 – cluster assignment

The second step in the K-means algorithm is the assignment of the observations to the clusters for which the centroids have been initialized in step 1. This feat is accomplished by the private `assignToClusters` method:

```
def assignToClusters(xt: U, clusters: V,
  members: Array[Int]): Int = {
  xt.zipWithIndex.filter{ case(x, n) => { //13
    val nearestCluster = getNearestCluster(clusters, x) //14
    val reassigned = nearestCluster != members(n)

    clusters(nearestCluster) += n //15
    members(n) = nearestCluster //16
    reassigned
  }}.size
}
```

The core of the assignment of observations to each cluster is the filter on the time series (line 13). The filter computes the index of the closest cluster and checks whether the observation is to be reassigned (line 14). The observation at the `n` index is added to the nearest cluster, `clusters(nearestCluster)` (line 15). The current membership of the observations is then updated (line 16).

The cluster closest to an observation data is computed by the private `getNearestCluster` method as follows:

```
def getNearestCluster(clusters: V, x: Array[T]): Int =  
  clusters.zipWithIndex./:(Double.MaxValue, 0)) {  
    case (p, (c, n) ) => {  
      val measure = distance(c.center, x) //17  
      if( measure < p._1) (measure, n) else p  
    } }._2
```

A fold is used to extract the cluster that is closest to the `x` observation from the list of clusters using the distance metric defined in the K-means constructor (line 17).

As with other data processing units, the extraction of K-means clusters is encapsulated in a data transformation so that clustering can be integrated into a workflow using the composition of mixins described in the *Composing mixins to build a workflow* section in *Chapter 2, Hello World!*

K-means algorithm exit condition

In some rare instances, the algorithm may reassign the same few observations between clusters, preventing its convergence toward a solution in a reasonable time. Therefore, it is recommended that you add a maximum number of iterations as an exit condition. If K-means does not converge with the maximum number of iterations, then the cluster centroids need to be reinitialized and the iterative (or recursive) execution needs to be restarted.



The `| >` transformation requires that the computation of the standard deviation of the distance of the observations related to the centroid, `c`, is computed in the `stdDev` method:

```
type KMeansModel [T] = List[Cluster[T]]  
def stdDev [T] (  
  clusters: KMeansModel[T],  
  xt: XVSeries[T],  
  distance: DistanceFunc[T]): DblVector =  
  clusters.map( _.stdDev(xt, distance)).toVector
```

Centroid versus mean

The terms centroid and mean refer to the same entity: the center of a cluster. This chapter uses these two terms interchangeably.



Step 3 – reconstruction/error minimization

The clusters are initialized with predefined set of observations as their members. The algorithm updates the membership of each cluster by minimizing the total reconstruction error. There are two effective strategies to execute the K-means algorithm:

- Tail recursive execution
- Iterative execution

Creating K-means components

Let's declare the K-means algorithm class, `KMeans`, with its public methods. `KMeans` implements an `ITransform` data transformation using an implicit model extracted from a training set and is described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 18). The configuration of the `KMeansConfig` type consists of the tuple `(K, maxIter)` with `K` being the number of clusters and `maxIter` being the maximum number of iterations allowed for the convergence of the algorithm:

```
case class KMeansConfig(val K: Int, maxIter: Int)
```

The `KMeans` class takes the following three arguments:

- `config`: This is the configuration used for the execution of the algorithm
- `distance`: This is the function used to compute the distance between any observation and a cluster centroid
- `xt`: This is the training set

The implicit conversion of the `T` type to a `Double` is implemented as a view bound. The instantiation of the `KMeans` class initializes a `V` type of output from K-means as `Cluster[T]` (line 20). The `num` instance of the `Numeric` class has to be passed implicitly as a class parameter because it is required by the `sortWith` invocation in `initialize`, the `maxBy` method, and the `Cluster.moveCenter` method (line 19). The `Manifest` is required to preserve the erasure type for `Array[T]` in the JVM:

```
class KMeans[T <: AnyVal](config: KMeansConfig, //18
    distance: DistanceFunc[T],
    xt: XVSeries[T])
    (implicit m: Manifest[T], num: Numeric[T], f: T=>Double) //19
extends ITransform[Array[T]](xt) with Monitor[T] {

    type V = Cluster[T] //20
    val model: Option[KMeansModel[T]] = train
    def train: Option[KMeansModel[T]]
    override def |> : PartialFunction[U, Try[V]]
    ...
}
```

The KMeansModel model is defined as the list of clusters extracted through training.

Tail recursive implementation

The transformation or clustering function is implemented by the train training method that creates a partial function with `xvSeries[T]` as the input and `KMeansModel[T]` as the output:

```
def train: Option[KMeansModel[T]] = Try {
    // STEP 1
    val clusters = initialize(xt) //21
    if( clusters.isEmpty) /* ... */
    else {
        // STEP 2
        val members = Array.fill(xt.size)(0)
        assignToClusters(xt, clusters, members) //22
        var iters = 0

        // Declaration of the tail recursion def update
        if( iters >= _config.maxIters )
            throw new IllegalStateException( /* .. */)
        // STEP 3
        update(clusters, xt, members) //23
    }
} match {
    case Success(clusters) => Some(clusters)
    case Failure(e) => /* ... */
}
```

The K-means training algorithm is implemented through the following three steps:

1. Initialize the cluster's centroid using the `initialize` method (line 21).
2. Assign observations to each cluster using the `assignToClusters` method (line 22).
3. Recompute the total error reconstruction using the `update` recursive method (line 23).

The computation of the total error reconstruction is implemented as a tail recursive method, `update`, as follows:

```
@tailrec
def update(clusters: KMeansModel[T], xt: U,
           members: Array[Int]): KMeansModel[T] = { //24

    val newClusters = clusters.map( c => {
```

```

        if( c.size > 0) c.moveCenter(xt) //25
        else clusters.filter( _.size >0)
            .maxBy(_.stdDev(xt, distance)) //26
    })
iters += 1
if(iters >= config.maxIters ||           //27
    assignToClusters(xt, newClusters, members) ==0)
    newClusters
else
    update(newClusters, xt, membership) //28
}

```

The recursion takes the following three arguments (line 24):

- The current list of `clusters` that is updated during the recursion
- The `xt` input time series
- The indices of membership to the clusters, `members`

A new list of clusters, `newClusters`, is computed by either recalculating each centroid if the cluster is not empty (line 25) or evaluating the standard deviation of the distance of each observation relative to each centroid (line 26). The execution exits when either the maximum number of the `maxIters` recursive calls is reached or when no more observations are reassigned to a different cluster (line 27). Otherwise, the method invokes itself with an updated list of clusters (line 28).

Iterative implementation

The implementation of an iterative execution is presented for an informational purpose. It follows the same sequence of calls as with the recursive implementation. The new clusters are computed (line 29) and the execution exits when either the maximum number of allowed iterations is reached (line 30) or when no more observations are reassigned to a different cluster (line 31):

```

val members = Array.fill(xt.size)(0)
assignToClusters(xt, clusters, members)
var newClusters: KMeansModel[T] = List.empty[Cluster[T]]
Range(0, maxIters).find( _ => { //29
    newClusters = clusters.map( c => { //30
        if( c.size > 0) c.moveCenter(xt)
        else clusters.filter( _.size > 0)
            .maxBy(_.stdDev(xt, distance))
    })
    assignToClusters(xt, newClusters, members) > 0 //31
}).map(_ => newClusters)

```

The density of the clusters is computed in the `KMeans` class as follows:

```
def density: Option[DblVector] =  
  model.map( _.map( c =>  
    c.getMembers.map(xt(_)).map( distance(c.center, _) ).sum)
```

Step 4 – classification

The objective of the classification is to assign an observation to a cluster with the closest centroid:

```
override def |> : PartialFunction[Array[T], Try[V]] = {  
  case x: Array[T] if( x.length == dimension(xt)  
    && model != None) =>  
    Try (model.map( _.minBy(c => distance(c.center,x))).get )  
}
```

The most appropriate cluster is computed by selecting the `c` cluster whose center is the closest to the `x` observation using the `minBy` higher order method.

The curse of dimensionality

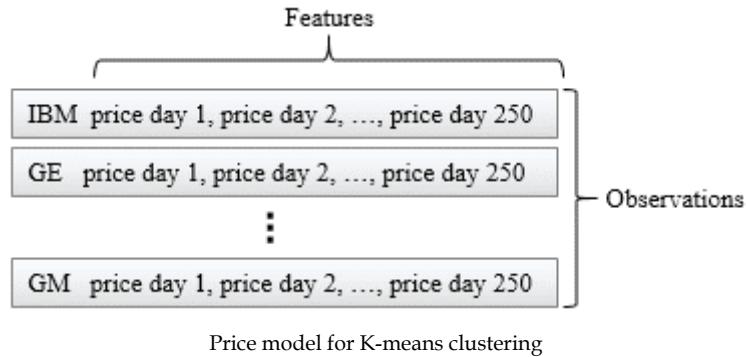
A model with a significant number of features (high dimensions) requires a larger number of observations in order to extract relevant and reliable clusters. K-means clustering with very small datasets (< 50) produces models with high bias and a limited number of clusters, which are affected by the order of observations [4:5]. I have been using the following simple empirical rule of thumb for a training set of size n , expected K clusters, and N features: $n < K.N$.



Dimensionality and the size of the training set

The issue of sizing the training set given the dimensionality of a model is not specific to unsupervised learning algorithms. All supervised learning techniques face the same challenge to set up a viable training plan.

Whichever empirical rule you follow, such a restriction is particularly an issue for analyzing stocks using historical quotes. Let's consider our examples of using technical analysis to categorize stocks according to their price behavior over a period of 1 year (or approximately 250 trading days). The dimension of the problem is 250 (250 daily closing prices). The number of stocks (observations) would have exceeded several hundred!



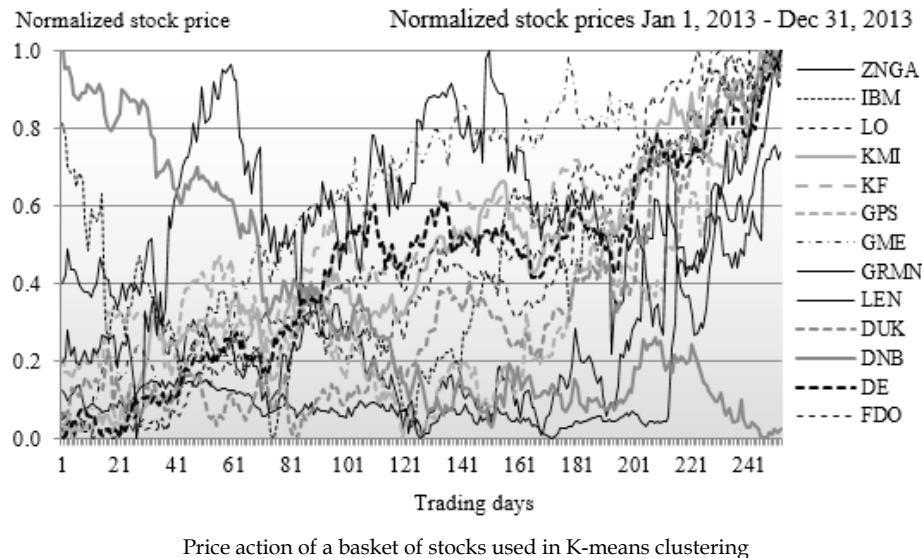
There are options to get around this limitation and shrink the numbers of observations, which are as follows:

- Sampling the trading data without losing a significant amount of information from the raw data, assuming that the distribution of observations follows a known probability density function.
- Smoothing the data to remove the noise as seen in *Chapter 3, Data Preprocessing*, assuming that the noise is Gaussian. In our test, a smoothing technique will remove the price outliers for each stock and therefore reduce the number of features (trading session). This approach differs from the first (sampling) technique because it does not require an assumption that the dataset follows a known density function. On the other hand, the reduction of features will be less significant.

These approaches are workaround solutions at best, used for the sake of this tutorial. You need to consider the quality of your data before applying these techniques to the actual commercial applications. The principal component analysis introduced in the last paragraph of this chapter is one of the most reliable dimension reduction techniques.

Setting up the evaluation

The objective is to extract clusters from a set of stock price actions during a period of time between January 1 and Dec 31, 2013 as features. For this test, 127 stocks are randomly selected from the S&P 500 list. The following chart visualizes the behavior of the normalized price of a subset of these 127 stocks:



The key is to select the appropriate features prior to clustering and the time window to operate on. It would make sense to consider the entire historical price over the 252 trading days as a feature. However, the number of observations (stocks) is too limited to use the entire price range. The observations are the stock closing prices for each trading session between the 80th and 130th trading days. The adjusted daily closing prices are normalized using their respective minimum and maximum values.

First, let's create a simple method to compute the density of the clusters:

```
val MAX_ITERS = 150
def density(K: Int, obs: XSeries[Double]): DblVector =
  KMeans[Double](KMeansConfig(K, MAX_ITERS)).density.get //32
```

The density method invokes `KMeans.density` described in step 3. Let's load the data from CSV files using the `DataSource` class, as described in the *Data extraction* section in the *Appendix A, Basic Concepts*:

```
import YahooFinancials._
val START_INDEX = 80; val NUM_SAMPLES = 50 //33
val PATH = "resources/data/chap4/"
```

```
type INPUT = Array[String] => Double
val extractor = adjClose :: List[INPUT]() //34
val symbolFiles = DataSource.listSymbolFiles(PATH) //35

for {
    prices <- getPrices //36
    values <- Try(getPricesRange(prices)) //37
    stdDev <- Try(ks.map( density(_, values.toVector))) //38
    pfnKmeans <- Try {
        KMeans[Double](KMeansConfig(5,MAX_ITERS),values.toVector) |>
    } //39
    predict <- pfnKmeans(values.head) //40
} yield {
    val results = s"""Daily prices ${prices.size} stocks"""
    | \nClusters density ${stdDev.mkString(", ")}"""
    .stripMargin
    show(results)
}
```

As mentioned earlier, the cluster analysis applies to the closing price in the range between the 80th and 130th trading days (line 33). The extractor function retrieves the adjusted closing price for a stock from YahooFinancials (line 34). The list of stock tickers (or symbols) are extracted as a list of CSV filenames located in path (line 35). For instance, the ticker symbol for General Electric Corp. is GE and the trading data is located in GE.csv.

The execution extracts 50 daily prices using DataSource and then filters out the incorrectly formatted data using filter (line 36):

```
type XVSeriesSet = Array[XVSeries[Double]]
def getPrices: Try[XVSeriesSet] = Try {
    symbolFiles.map( DataSource(_, path) |> extractor )
    .filter( _.isSuccess ).map( _.get )
}
```

The historical stock prices for the trading session between the 80th and 130th days are generated by the getPricesRange closure (line 37):

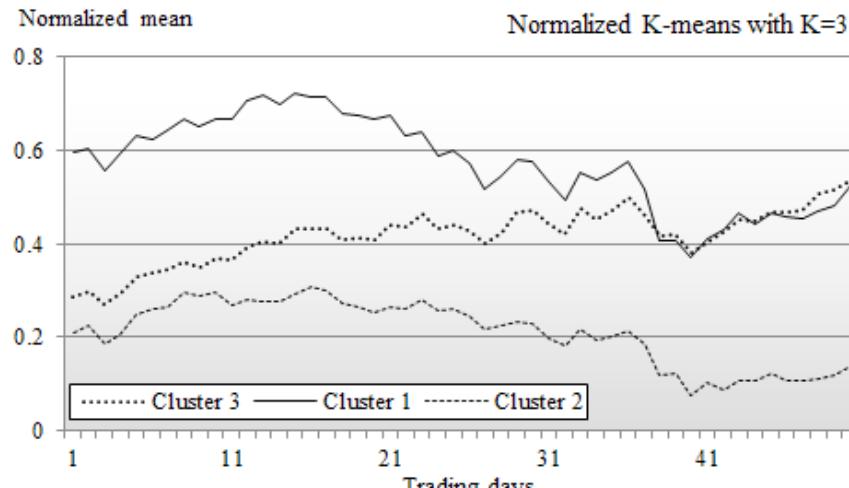
```
def getPricesRange(prices: XVSeriesSet) =
    prices.view.map(_.head.toArray)
    .map( _.drop(START_INDEX).take(NUM_SAMPLES) )
```

It computes the density of the clusters by invoking the density method for each ks value of the number of clusters (line 38).

The `pfnKmeans` partial classification function is created for a 5-cluster, `KMeans` (line 39), and then used to classify one of the observations (line 40).

Evaluating the results

The first test run is executed with $K=3$ clusters. The mean (or centroid) vector for each cluster is plotted as follows:



A chart of means of clusters using K-means K=3

The means vectors of the three clusters are quite distinctive. The top and bottom means **1** and **2** in the chart have the respective standard deviation of 0.34 and 0.27 and share a very similar pattern. The difference between the elements of the **1** and **2** cluster mean vectors is almost constant: 0.37. The cluster with a mean vector **3** represents the group of stocks that behave like the stocks in cluster **2** at the beginning of the time period and behave like the stocks in cluster **1** toward the end of the time period.

This behavior can be easily explained by the fact that the time window or trading period, the 80th to 130th trading day, correspond to the shift in the monetary policy of the federal reserve in regard to the quantitative easing program. Here is the partial list of stocks for each of the clusters whose centroid values are displayed on the chart:

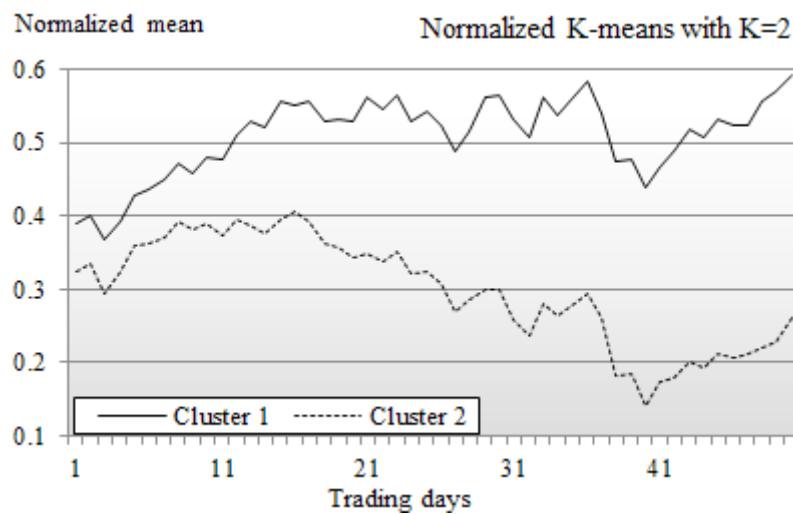
Cluster 1	AET, AHS, BBBY, BRCM, C, CB, CL, CLX, COH, CVX, CYH, DE, ...
Cluster 2	AA, AAPL, ADBE, ADSK, AFAM, AMZN, AU, BHI, BTU, CAT, CCL, ...
Cluster 3	ADM, ADP, AXP, BA, BBT, BEN, BK, BSX, CA, CBS, CCE, CELG, CHK, ...

Let's evaluate the impact of the number of clusters K on the characteristics of each cluster.

Tuning the number of clusters

We repeat the previous test on the 127 stocks and the same time window with the number of clusters varying from 2 to 15.

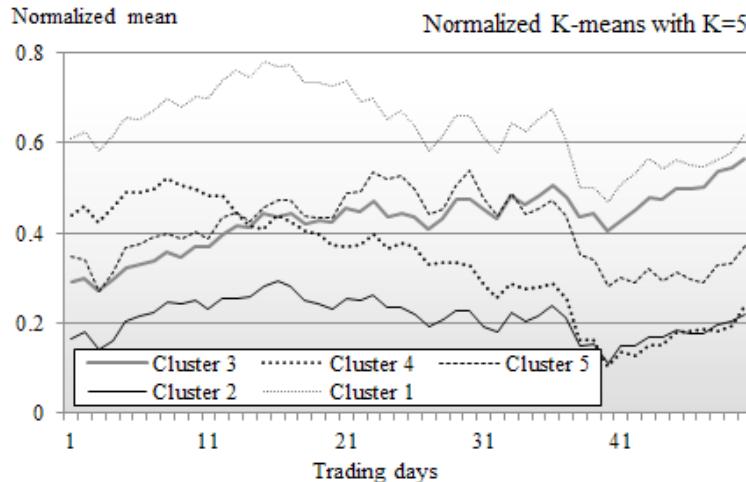
The mean (or centroid) vector for each cluster for $K = 2$ is plotted as follows:



A chart of means of clusters using K-means $K=2$

The chart of the results of the K-means algorithms with 2 clusters shows that the mean vector for the cluster labeled **2** is similar to the mean vector labeled **3** on the chart with $K = 5$ clusters. However, the cluster with the mean vector **1** reflects somewhat the aggregation or summation of the mean vectors for the clusters **1** and **3** in the chart $K = 5$. The aggregation effect explains why the standard deviation for the cluster **1** (0.55) is twice as much as the standard deviation for the cluster **2** (0.28).

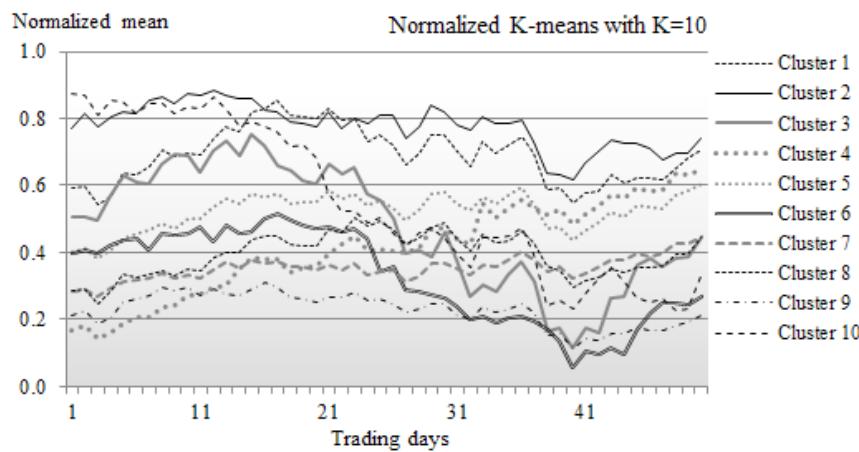
The mean (or centroid) vector for each cluster for $K = 5$ is plotted as follows:



A chart of means of clusters using K-means K=5

In this chart, we can assess that the clusters **1** (with the highest mean), **2** (with the lowest mean), and **3** are very similar to the clusters with the same labels in the chart for $K = 3$. The cluster with the mean vector **4** contains stocks whose behaviors are quite similar to those in cluster **3**, but in the opposite direction. In other words, the stocks in cluster **3** and **4** reacted in opposite ways following the announcement of the change in the monetary policy.

In the tests with high values of K , the distinction between the different clusters becomes murky, as shown in the following chart for $K = 10$:



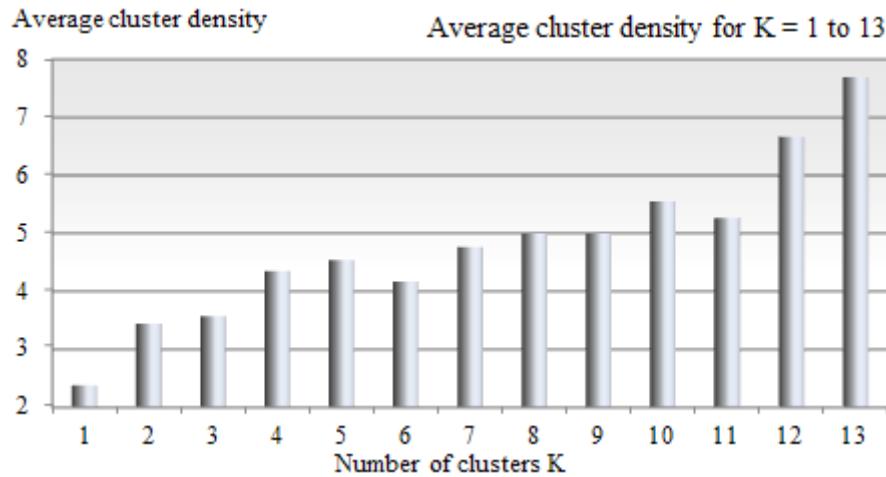
A chart of means of clusters using K-means K=10

The means for clusters **1**, **2**, and **3** seen in the first chart for the case $K = 2$ are still visible. It is fair to assume that these are very likely the most reliable clusters. These clusters happened to have a low standard deviation or high density.

Let's define the density of a cluster, C_j , with a centroid, c_j , as the inverse of the Euclidean distance between all the members of each cluster and its mean (or centroid) (M6):

$$d(C_j) = 1 / \sum_{x \in C_j} (x - c_j)^2$$

The density of the cluster is plotted against the number of clusters with $K = 1$ to $K = 13$:



A bar chart of the average cluster density for $K = 1$ to 13

As expected, the average density of each cluster increases as K increases. From this experiment, we can draw the simple conclusion that the density of each cluster does not significantly increase in the test runs for $K = 5$ and beyond. You may observe that the density does not always increase as the number of clusters increases ($K = 6$ and $K = 11$). The anomaly can be explained by the following three factors:

- The original data is noisy
- The model is somewhat dependent on the initialization of the centroids
- The exit condition is too loose

Validation

There are several methodologies to validate the output of a K-means algorithm from purity to mutual information [4:6]. One effective way to validate the output of a clustering algorithm is to label each cluster and run those clusters through a new batch of labeled observations. For example, if during one of these tests, you find that one of the clusters, CC , contains most of the commodity-related stocks, then you can select another commodity-related stock, SC , which is not part of the first batch, and run the entire clustering algorithm again. If SC is a subset of CC , then K-means has performed as expected. If this is the case, you should run a new set of stocks, some of which are commodity-related, and measure the number of true positives, true negatives, false positives, and false negatives. The values for the precision, recall, and F_1 score introduced in the *Assessing a model* section in *Chapter 2, Hello World!*, confirms whether the tuning parameters and labels you selected for your cluster are indeed correct.

F1 validation for K-means



The quality of the clusters, as measured by the F_1 score, depends on the rule, policy, or formula used to label observations (that is, label a cluster with the industry with the highest relative percentage of stocks in the cluster). This process is quite subjective. The only sure way to validate a methodology is to evaluate several labeling schemes and select the one that generate the highest F_1 score.

An alternative to measure the homogeneity of the distribution of observations across the clusters is to compute the statistical entropy. A low entropy value indicates that the clusters have a low level of impurity. Entropy can be used to find the optimal number of clusters K .

We reviewed some of the tuning parameters that affect the quality of the results of the K-means clustering, which are as follows:

- Initial selection of a centroid
- Number of K clusters

In some cases, the similarity criterion (that is, the Euclidean distance or cosine distance) can have an impact on the *cleanliness* or density of the clusters.

The final and important consideration is the computational complexity of the K-means algorithm. The previous sections of the chapter described some of the performance issues with K-means and possible remedies.

Despite its many benefits, the K-means algorithm does not handle missing data or unobserved features very well. Features that depend on each other indirectly may in fact depend on a common hidden (also known as latent) feature. The expectation-maximization algorithm described in the next section addresses some of these limitations.

The expectation-maximization algorithm

The expectation-maximization algorithm was originally introduced to estimate the maximum likelihood in the case of incomplete data [4:7]. It is an iterative method to compute the model features that maximize the likely estimate for observed values, taking into account unobserved values.

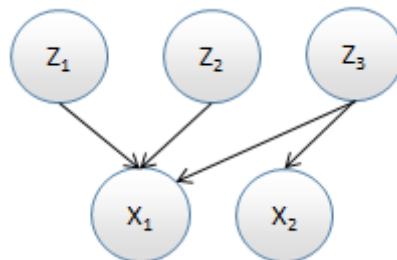
The iterative algorithm consists of computing the following:

- The expectation, E , of the maximum likelihood for the observed data by inferring the latent values (E-step)
- The model features that maximize the expectation E (M-step)

The expectation-maximization algorithm is applied to solve clustering problems by assuming that each latent variable follows a normal or Gaussian distribution. This is similar to the K-means algorithm for which the distance of each data point to the center of each cluster follows a Gaussian distribution [4:8]. Therefore, a set of latent variables is a mixture of Gaussian distributions.

Gaussian mixture models

Latent variables, Z_i , can be visualized as the behavior (or symptoms) of a model (observed) X for which Z are the root causes of the behavior:



Visualization of observed and latent features

The latent values, Z , follow a Gaussian distribution. For the statisticians among us, the mathematics of a mixture model is described here:

Maximization of the log likelihood

M7: If $x = \{x_i\}$ is a set of observed features associated with latent features $z = \{z_i\}$, the probability for the feature x_i of the observation x , given a model parameter θ , is defined as:



$$p(x_i|\theta) = \sum_z p(x_i, z|\theta)$$

M8: The objective is to maximize the likelihood, $L(\theta)$, as shown here:

$$\mathcal{L}(\theta) = \sum_{i=0}^{N-1} \log \left\{ \sum_z p(x_i, z|\theta) \right\} \quad \tilde{\theta} = \text{argmax } \mathcal{L}(\theta)$$

Overview of EM

As far as the implementation is concerned, the expectation-maximization algorithm can be broken down into three stages:

1. The computation of the log likelihood for the model features given some latent variables (initial step).
2. The computation of the expectation of the log likelihood at iteration t (E step).
3. The maximization of the expectation at iteration t (M step).

The E step

M9: The expectation, Q , of the complete data log likelihood for the model parameters, θ_n , at iteration, n , is computed using the posterior distribution of latent variable, z , $p(z|x, \theta)$, and the joint probability of the observation and the latent variable:



$$Q(\theta, \theta^n) = \sum_z p(z|x_i, \theta^n) \cdot \log p(x_i, z|\theta)$$

The M-step

M10: The expectation function Q is maximized for the model features θ to compute the model parameters θ_{n+1} for the next iteration:

$$\theta^{n+1} = \arg \max_{\theta} Q(\theta, \theta^n) \quad |\theta^{n+1} - \theta^n| < \varepsilon$$

A formal, detailed, but short mathematical formulation of the EM algorithm can be found in S. Borman's tutorial [4:9].

Implementation

Let's implement the three steps (initial step, E step, and M step) in Scala. The internal calculations of the EM algorithm are a bit complex and overwhelming. You may not benefit much from the details of a specific implementation such as computation of the eigenvalues of the covariance matrix of the expectation of the log likelihood. This implementation hides some complexities using the Apache Commons Math library package [4:10].



The inner workings of EM

You may want to download the source code for the implementation of the EM algorithm in the Apache Commons Math library, if you need to understand the condition for which an exception is thrown.

The expectation-maximization algorithm of the `MultivariateEM` type is implemented as a data transformation of an `ITransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*. The two arguments of the constructors are the number of `K` clusters (or gauss distribution) and the `xt` training set (line 1). The constructor initializes the `V` type of the output as `EMCluster` (line 2):

```
class MultivariateEM[T <: AnyVal] (K: Int,
  xt: XVSeries[T]) (implicit f: T => Double)
  extends ITransform[Array[T]] (xt) with Monitor[T] { //1
  type V = EMCluster //2
  val model: Option[EMModel] = train //3
  override def |> : PartialFunction[U, Try[V]]
}
```

The multivariate expectation-maximization class has a model that consists of a list of EM clusters of the `EMCluster` type. The `Monitor` trait is used to collect the profiling information during training (refer to the *Monitor* section under *Utility classes* in the *Appendix A, Basic Concepts*).

The information about an EM cluster, `EMCluster`, is defined by `key`, the centroid or means value, and density of the cluster that is the standard deviation of the distance of all the data points to the mean (line 4):

```
case class EMCluster(key: Double, val means: DblArray,
  val density: DblArray) //4
type EMModel = List[EMCluster]
```

The implementation of the EM algorithm in the `train` method uses the Apache Commons Math `MultivariateNormalMixture` for the Gaussian mixture model and `MultivariateNormalMixtureExpectationMaximization` for the EM algorithm:

```
def train: Option[EMModel] = Try {
  val data: DblMatrix = xt //5
  val multivariateEM = new EM(data)
  multivariateEM.fit(estimate(data, K)) //6

  val newMixture = multivariateEM.getFittedModel //7
  val components = newMixture.getComponents.toList //8
  components.map(p => EMCluster(p.getKey, p.getValue.getMeans,
    p.getValue.getStandardDeviations)) //9
} match {/* ... */}
```

Let's take a look at the main `train` method of the `MultivariateEM` wrapper class. The first step is to convert the time series into a primitive matrix of `Double` with observations/historical quotes as rows and the stock symbols as columns.

The `xt` time series of the `xvSeries[T]` type is converted to a `DblMatrix` through an induced implicit conversion (line 5).

The initial mixture of Gaussian distributions can be provided by the user or can be extracted from the `estimate` datasets (line 6). The `getFittedModel` triggers the M-step (line 7).

Conversion from Java and Scala collections



Java primitives need to be converted to Scala types using the `import scala.collection.JavaConversions` package. For example, `java.util.List` is converted to `scala.collection.immutable.List` by invoking the `asScalaIterator` method of the `WrapAsScala` class, one of the base traits of `JavaConversions`.

The Apache Commons Math `getComponents` method returns a `java.util.List` that is converted to `scala.collection.immutable.List` by invoking the `toList` method (line 8). Finally, the `data transform` returns a list of cluster information of the `EMCluster` type (line 9).

Third-party library exceptions

Scala does not enforce the declaration of exceptions as part of the signature of a method. Therefore, there is no guarantee that all types of exceptions will be caught locally. This problem occurs when exceptions are thrown from a third-party library in two scenarios:

- The documentation of the API does not list all the types of exceptions
- The library is updated and a new type of exception is added to a method

One easy workaround is to leverage the Scala exception-handling mechanism:

```
Try {
  ...
} match {
  case Success(results) => ...
  case Failure(exception) => ...
}
```

Classification

The classification of a new observation or data points is implemented by the `| >` method:

```
override def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T]
    if(isModel && x.length == dimension(xt)) =>
      Try(model.map(_.minBy(c => euclidean(c.means, x))).get)
}
```

The `| >` method is similar to the `KMeans.| >` classifier.

Testing

Let's apply the `MultivariateEM` class to the clustering of the same 127 stocks used in evaluating the K-means algorithm.

As discussed in the *The curse of dimensionality* section, the number of stocks (127) to analyze restricts the number of observations to be used by the EM algorithm. A simple option is to filter out some of the noise of the stock's prices and apply a simple sampling method. The maximum sampling rate is restricted by the frequencies in the spectrum of noises of different types in the historical price of every stock.

Filtering and sampling

The preprocessing of the data using a combination of a simple moving average and fixed interval sampling prior to clustering is very rudimentary in this example. For instance, we cannot assume that the historical price of all the stocks share the same noise characteristics. The noise pattern in the quotation of momentum and heavily traded stocks is certainly different from blue-chip securities with a strong ownership, and these stocks are held by large mutual funds.

The sampling rate should take into account the spectrum of frequency of the noise. It should be set as at least twice the frequency of the noise with the lowest frequency.

The object of the test is to evaluate the impact of the sampling rate, `samplingRate`, and the number of `K` clusters used in the EM algorithm:

```
val K = 4; val period = 8
val smAve = SimpleMovingAverage[Double](period) //10
val pfnSmAve = smAve |> //11
```

```
val obs = symbolFiles.map(sym => (
  for {
    xs <- DataSource(sym, path, true, 1) |> extractor //12
    values <- pfnSmAve(xs.head) //13
    y <- Try {
      values.view.zipWithIndex.drop(period+1).toVector
        .filter(_._2 % samplingRate == 0)
        .map(_._1).toArray //14
    }
  } yield y).get)

em(K, obs) //15
```

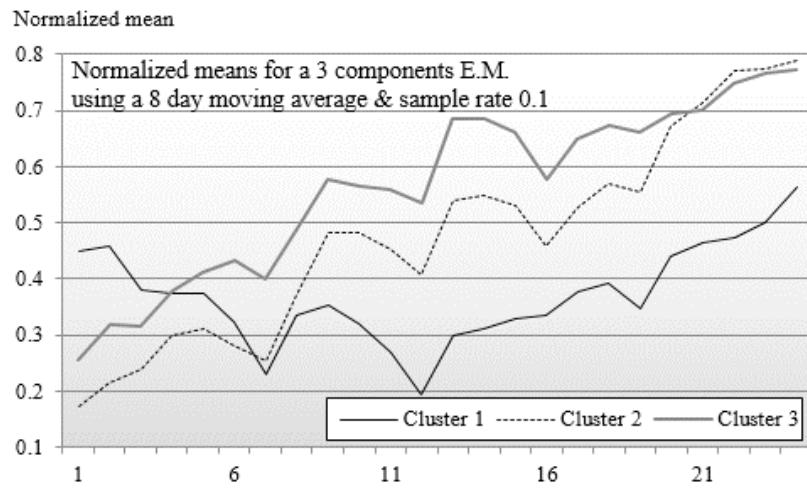
The first step is to create a simple moving average with a predefined period (line 10), as described in the *The simple moving average* section in *Chapter 3, Data Preprocessing*. The test code instantiates the `pfnSmAve` partial function that implements the moving average computation (line 11). The symbols of the stocks under consideration are extracted from the name of the files in the path directory. The historical data is contained in the CSV file whose name is `path/STOCK_NAME.csv` (line 12).

The execution of the moving average (line 13) generates a set of smoothed values that is sampled given a sampling rate, `samplingRate` (line 14). Finally, the expectation-maximization algorithm is instantiated to cluster the sampled data in the `em` method (line 15):

```
def em(K: Int, obs: DblMatrix): Int = {
  val em = MultivariateEM[Double](K, obs.toVector) //16
  show(s"${em.toString}") //17
}
```

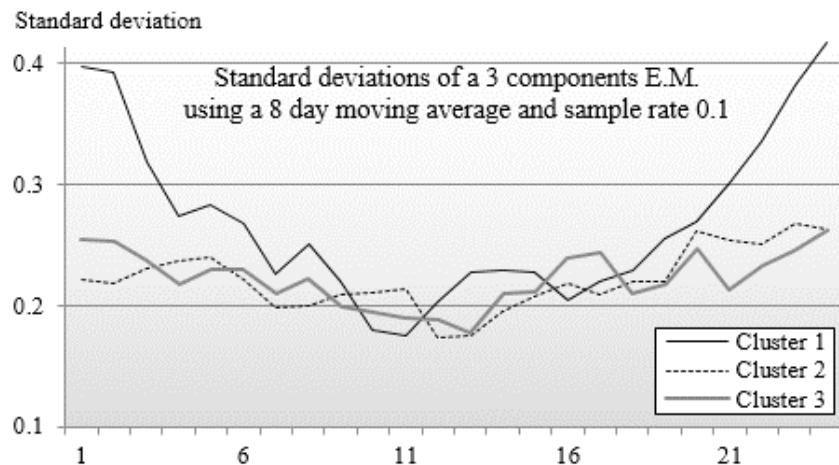
The `em` method instantiates the EM algorithm for a specific number `K` of clusters (line 16). The content of the model is displayed by invoking `MultivariateEM.toString`. The results are aggregated, then displayed in a textual format on the standard output (line 17).

The first test is to execute the EM algorithm with $K = 3$ clusters and a sampling period of 10 on data smoothed by a simple moving average with period of 8. The sampling of historical prices of the 127 stocks between January 1, 2013 and December 31, 2013 with a frequency of 0.1 hertz produces 24 data points. The following chart displays the mean of each of the three clusters:



A chart of the normalized means per cluster using EM K=3

The mean vectors of clusters 2 and 3 have similar patterns, which may suggest that a set of three clusters is accurate enough to provide a first insight into the similarity within groups of stocks. The following is a chart of the normalized standard deviation per cluster using EM with $K = 3$:



A chart of the normalized standard deviation per cluster using EM K=3

The distribution of the standard deviation along with the mean vector of each cluster can be explained by the fact that the price of stocks from a couple of industries went down in synergy, while others went up as a semi-homogenous group following the announcement from the Federal Reserve that the monthly quantity of bonds purchased as part of the quantitative easing program would be reduced in the near future.

Relation to K-means



You may wonder what is the relation between EM and K-means, as both the techniques address the same problem. The K-means algorithm assigns each observation uniquely to one and only one cluster. The EM algorithm assigns an observation based on posterior probability. K-means is a special case of the EM for Gaussian mixtures [4:11].

The online EM algorithm

Online learning is a powerful strategy for training a clustering model when dealing with very large datasets. This strategy has regained interest from scientists lately. The description of the online EM algorithm is beyond the scope of this tutorial. However, you may need to know that there are several algorithms for online EM available if you ever have to deal with large datasets, such as batch EM, stepwise EM, incremental EM, and Monte Carlo EM [4:12].

Dimension reduction

Without prior knowledge of the problem domain, data scientists include all possible features in their first attempt to create a classification, prediction, or regression model. After all, making assumptions is a poor and dangerous approach to reduce the search space. It is not uncommon for a model to use hundreds of features, adding complexity and significant computation costs to build and validate the model.

Noise filtering techniques reduce the sensitivity of the model to features that are associated with sporadic behavior. However, these noise-related features are not known prior to the training phase, and therefore, cannot be completely discarded. As a consequence, training of the model becomes a very cumbersome and time-consuming task.

Overfitting is another hurdle that can arise from a large feature set. A training set of limited size does not allow you to create an accurate model with a large number of features.

Dimension reduction techniques alleviate these problems by detecting features that have little influence on the overall model behavior.

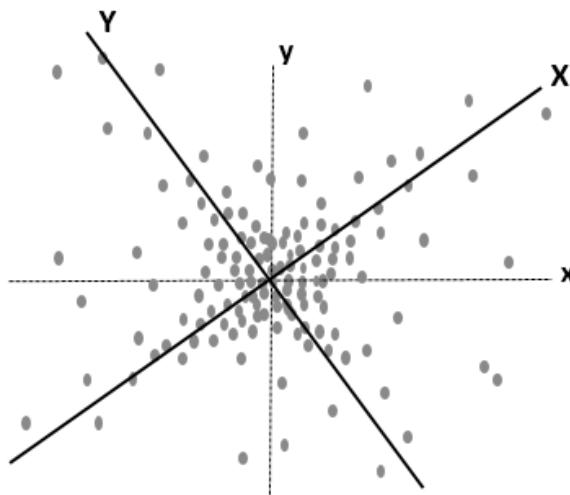
There are three approaches to reduce the number of features in a model:

- Statistical analysis solutions such as ANOVA for smaller feature sets
- Regularization and shrinking techniques, which are introduced in the *Regularization* section in *Chapter 6, Regression and Regularization*
- Algorithms that maximize the variance of the dataset by transforming the covariance matrix

The next section introduces one of the most commonly used algorithms of the third category: principal component analysis.

Principal components analysis

The purpose of principal components analysis is to transform the original set of features into a new set of ordered features by decreasing the order of variance. The original observations are transformed into a set of variables with a lower degree of correlation. Let's consider a model with two features, $\{x, y\}$, and a set of observations, $\{x_i, y_i\}$, plotted on the following chart:



Visualization of principal components analysis for a two-dimensional model

The x and y features are converted into two X and Y variables (that is rotation) to appropriately match the distribution of observations. The variable with the highest variance is known as the first principal component. The variable with the n^{th} highest variance is known as the n^{th} principal component.

Algorithm

I highly recommend that you read the tutorial from Lindsay Smith [4:13] that describes the PCA algorithm in a very concrete and simple way using a two-dimensional model.

PCA and covariance matrix

M11: The covariance of two X and Y features with the observations set $\{x_i, y_i\}$ and their respective mean values is defined as:

$$\text{cov}(X, Y) = \frac{1}{n-1} \sum (x_i - \bar{x})(y_i - \bar{y})$$

Here, \bar{x} and \bar{y} are the respective mean values for the observations, x and y .

M12: The covariance is computed from the Z-score of each observation:

$$x_i = (x_i - \bar{x})/\sigma$$

M13: For a model with n features, x_i , the covariance matrix is defined as:



$$\Sigma = \begin{bmatrix} \text{cov}(x_0, x_0) & \dots & \text{cov}(x_0, x_{n-1}) \\ \vdots & \text{cov}(x_i, x_j) & \vdots \\ \text{cov}(x_{n-1}, x_0) & \dots & \text{cov}(x_{n-1}, x_{n-1}) \end{bmatrix}$$

M14: The transformation of x to X consists of computing the eigenvalues of the covariance matrix:

$$\Sigma' = W^T \Sigma W = \|\text{cov}(X_i, X_j)\| \text{ and } X = W^T x$$

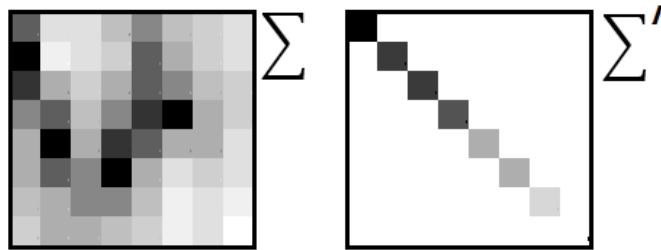
M15: The eigenvalues are ranked by their decreasing order of variance. Finally, the m top eigenvalues for which the cumulative of variance exceeds a predefined threshold (percentage of the trace of the matrix) are the principal components:

$$Z = \{X_i \mid 1:m \mid \sum_{k=1}^m \text{cov}(x_k, x_k) > \epsilon \cdot \text{Tr}(\text{cov})\}$$

The algorithm is implemented in five steps:

1. Compute the Z-score for the observations by standardizing the mean and standard deviation.
2. Compute the covariance matrix Σ for the original set of observations.
3. Compute the new covariance matrix Σ' for the observations with the transformed features by extracting the eigenvalues and eigenvectors.
4. Convert the matrix to rank eigenvalues by decreasing the order of variance. The ordered eigenvalues are the principal components.
5. Select the principal components for which the total sum of variance exceeds a threshold by a percentage of the trace of the new covariance matrix.

The extraction of principal components by diagonalization of the covariance matrix Σ is visualized in the following diagram. The shades of grey used to represent the covariance value varies from white (lowest value) to black (highest value):



Visualization of the extraction of eigenvalues in PCA

The eigenvalues (variance of X) are ranked by the decreasing order of their values. The PCA algorithm succeeds when the cumulative value of the last eigenvalues (the right-bottom section of the diagonal matrix) becomes insignificant.

Implementation

The principal components analysis can be easily implemented using the Apache Commons Math library methods that compute the eigenvalues and eigenvectors. The `PCA` class is defined as a data transformation of the `ITransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*

The `PCA` class has a single argument: the `xt` training set (line 1). The output type has a `Double` for the projection of an observation along with the eigenvectors (line 2). The constructor defines the z-score `norm` function (line 3):

```
class PCA[@specialized(Double) T <: AnyVal] (  
    xt: XVSeries[T]) (implicit f: T => Double)
```

```

extends ITransform[Array[T]](xt) with Monitor[T] { //1
  type V = Double //2

  val norm = (xv: XVSeries[T]) => zScores(xv) //3
  val model: Option[PCAModel] = train //4
  override def |> : PartialFunction[U, Try[V]]
}

```

The model for the PCA algorithm is defined by the `PCAModel` case class (line 4).

[

Triggering an implicit conversion



Implicit conversions can be invoked through an assignment to fully declared variables. For example, the conversion from `XVSeries[T]` to `XVseries[Double]` is invoked by the declaring the type of the target variable: `val z: XVSeries[Double] = xv` (line 4).

]

The model for the PCA algorithm, `PCAModel`, consists of the covariance matrix, covariance defined in the formula **M11**, and array of eigenvalues computed in the formula **M16**:

```

case class PCAModel(val covariance: DblMatrix,
  val eigenvalues: DblArray)

```

The `|>` transformative method implements the computation of the principal components (that is, the eigenvector and eigenvalues):

```

def train: Option[PCAModel] = zScores(xt).map(x => { //5
  val obs: DblMatrix = x.toArray
  val cov = new Covariance(obs).getCovarianceMatrix //6

  val transform = new EigenDecomposition(cov) //7
  val eigenVectors = transform.getV //8
  val eigenValues =
    new ArrayRealVector(transform.getRealEigenvalues)

  val covariance = obs.multiply(eigenVectors).getData //9
  PCAModel(covariance, eigenValues.toArray) //10
}) match {/* ... */}

```

The normalization function `zScores` performs the Z-score transformation (formula **M12**) (line 5). Next, the method computes the covariance matrix from the normalized data (line 6). The eigenvectors, `eigenVectors`, are computed (line 7) and then retrieved using the `getv` method in the Apache Commons Math `EigenDecomposition` class (line 8). The method computes the diagonal, transformed covariance matrix from the eigenvector (line 9). Finally, the data transformation returns an instance of the PCA model (line 10).

The `|>` predictive method consists of projecting an observation onto the principal components:

```
override def |> : PartialFunction[Array[T], Try[V]] = {  
    case x: Array[T]  
    if(isModel && x.length == dimension(xt)) =>  
        Try( inner(x, model.get.eigenvalues) )  
    }  
}
```

The `inner` method of the `XTSerie` object computes the dot product of the values `x` and the `eigenvalues` model.

Test case

Let's apply the PCA algorithm to extract a subset of the features that represents some of the financial metrics ratios of 34 S&P 500 companies. The metrics under consideration are as follows:

- Trailing Price-to-Earnings ratio (PE)
- Price-to-Sale ratio (PS)
- Price-to-Book ratio (PB)
- Return on Equity (ROE)
- Operation Margin (OM)

The financial metrics are described in the *Terminology* section under *Finances 101* in the *Appendix A, Basic Concepts*.

The input data is specified with the following format as a tuple (a ticker symbol and an array of five financial ratios, PE, PS, PB, ROE, and OM):

```
val data = Array[(String, DblVector)] (  
    // Ticker          PE      PS      PB      ROE      OM  
    ("QCOM", Array[Double](20.8, 5.32, 3.65, 17.65, 29.2)),  
    ("IBM",  Array[Double](13, 1.22, 12.2, 88.1, 19.9)),  
    ...  
)
```

The client code that executes the PCA algorithm is defined simply as follows:

```
val dim = data.head._2.size
val input = data.map( _._2.take(dim))
val pca = new PCA[Double](input) //11
show(s"PCA model: ${pca.toString}") //12
```

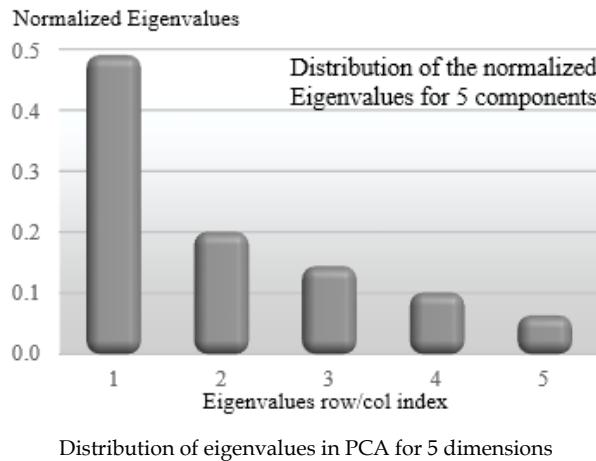
The PCA is instantiated with the `input` data (line 11) and then displayed in a textual format (line 12).

Evaluation

The first test on the 34 financial ratios uses a model that has five dimensions. As expected, the algorithm produces a list of five ordered eigenvalues:

2.5321, 1.0350, 0.7438, 0.5218, 0.3284

Let's plot the relative value of the eigenvalues (that is, relative importance of each feature) on the following bar chart:



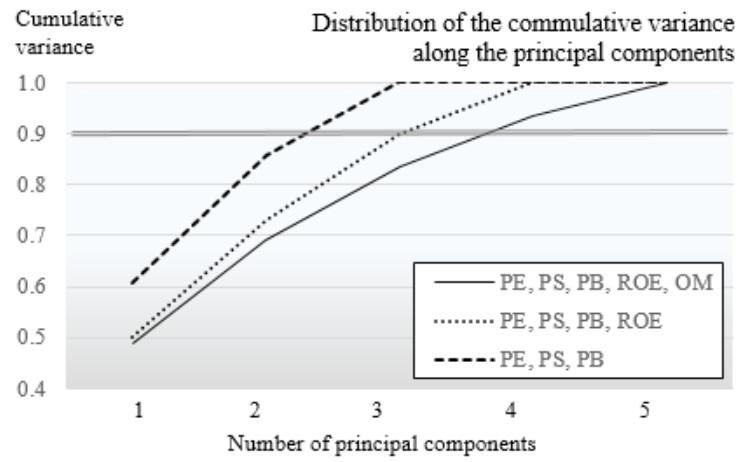
The chart shows that three out of five features account for 85 percent of the total variance (trace of the transformed covariance matrix). I invite you to experiment with different combinations of these features. The selection of a subset of the existing features is as simple as applying Scala's `take` or `drop` methods:

```
data.map( _._2.take(dim))
```

Let's plot the cumulative eigenvalues for the three different model configurations:

- **Five features:** PE, PS, PB, ROE, and OM
- **Four features:** PE, PS, PB, and ROE
- **Three features:** PE, PS, and PB

The graph will be as follows:



Distribution of eigenvalues in PCA for 3, 4, and 5 features

The chart displays the cumulative value of eigenvalues that are the variance of the transformed features, X_i . If we apply a threshold of 90 percent to the cumulative variance, then the number of principal components for each test model is as follows:

- {PE, PS, PB}: 2
- {PE, PS, PB, ROE}: 3
- {PE, PS, PB, ROE, OM}: 3

In conclusion, the PCA algorithm reduced the dimension of the model by 33 percent for the three-feature model, 25 percent for the four-feature model, and 40 percent for the five-feature model for a threshold of 90 percent.

Cross-validation of PCA

 Like any other unsupervised learning technique, the resulting principal components have to be validated through a one or K-fold cross-validation methodology using a regression estimator such as **Partial Least Square Regression** (PLSR) or the **Predicted Residual Error Sum of Squares** (PRESS). For those who are not afraid of statistics, I recommend that you read *Fast Cross-validation in Robust PCA* by S. Engelen and M. Hubert [4:14]. You need to be aware of the fact that the implementation of these regression estimators is not simple. The validation of the PCA is beyond the scope of this book.

Principal components analysis is a special case of the more general factor analysis. The latter class of algorithm does not require the transformation of the covariance matrix to be orthogonal.

Non-linear models

The principal components analysis technique requires the model to be linear. Although the study of such algorithms is beyond the scope of this book, it's worth mentioning two approaches that extend PCA for non-linear models:

- Kernel PCA
- Manifold learning

Kernel PCA

PCA extracts a set of orthogonal linear projections of an array of correlated values, $X = \{x_i\}$. The kernel PCA algorithm consists of extracting a similar set of orthogonal projection of the inner product matrix, $X^T X$. Nonlinearity is supported by applying a kernel function to the inner product. Kernel functions are described in the *Kernel functions* section in *Chapter 8, Kernel Models and Support Vector Machines*. The kernel PCA is an attempt to extract a low dimension feature set (or manifold) from the original observation space. The linear PCA is the projection on the tangent space of the manifold.

Manifolds

The concept of a manifold is borrowed from differential geometry. **Manifolds** generalize the notions of curves in a two-dimension space or surfaces in a three dimension space to a higher dimension. Nonlinear models are associated with Riemann manifolds whose metric is the inner product, $X^T X$, on a tangent space. The manifold represents a low dimension feature space embedded into the original observation space. The idea is to project the principal components from the linear tangent space to a manifold using a exponential map. This feat is accomplished using a variety of techniques from Local Linear Embedding and density preserving maps to Laplacian Eigenmaps [4:15].

The vector of observations cannot be directly used on a manifold because metrics such as norms or inner products depend on the location of the manifold the vector is applied to. Computation on manifolds relies on tensors such as contravariant and covariant vectors. Tensors algebra is supported by covariant and contravariant functors, which is introduced in the *Abstraction* section in *Chapter 1, Getting Started*.

Techniques that use differentiable manifolds are known as spectral dimensionality reduction.

Alternative dimension reduction techniques

Here are some more alternative techniques, listed as references: factor analysis, principal factor analysis, maximum likelihood factor analysis, independent component analysis, Random projection, nonlinear ICA, Kohonen's self-organizing maps, neural networks, and multidimensional scaling, just to name a few [4:16].

Manifold learning algorithms such as classifiers and dimension reduction techniques are associated with semi-supervised learning.

Performance considerations

The three unsupervised learning techniques share the same limitation – a high computational complexity.

K-means

The K-means has the computational complexity of $O(iKnm)$, where i is the number of iterations (or recursions), K is the number of clusters, n is the number of observations, and m is the number of features. Here are some remedies to the poor performance of the K-means algorithm:

- Reducing the average number of iterations by seeding the centroid using a technique such as initialization by ranking the variance of the initial cluster, as described in the beginning of this chapter
- Using a parallel implementation of K-means and leveraging a large-scale framework such as Hadoop or Spark
- Reducing the number of outliers and features by filtering out the noise with a smoothing algorithm such as a discrete Fourier transform or a Kalman filter
- Decreasing the dimensions of the model by following a two-step process:
 1. Execute a first pass with a smaller number of clusters K and/or a loose exit condition regarding the reassignment of data points. The data points close to each centroid are aggregated into a single observation.
 2. Execute a second pass on the aggregated observations.

EM

The computational complexity of the expectation-maximization algorithm for each iteration (E + M steps) is $O(m^2n)$, where m is the number of hidden or latent variables and n is the number of observations.

A partial list of suggested performance improvement includes:

- Filtering of raw data to remove noise and outliers
- Using a sparse matrix on a large feature set to reduce the complexity of the covariance matrix, if possible
- Applying the Gaussian mixture model wherever possible—the assumption of Gaussian distribution simplifies the computation of the log likelihood
- Using a parallel data processing framework such as Apache Hadoop or Spark, as discussed in the *Apache Spark* section in *Chapter 12, Scalable Frameworks*
- Using a kernel function to reduce the estimate of covariance in the E-step

PCA

The computational complexity of the extraction of the principal components is $O(m^2n + n^3)$, where m is the number of features and n is the number of observations. The first term represents the computational complexity for computing the covariance matrix. The second term reflects the computational complexity of the eigenvalue decomposition.

The list of potential performance improvements or alternative solutions for PCA includes the following:

- Assuming that the variance is Gaussian
- Using a sparse matrix to compute eigenvalues for problems with large feature sets and missing data
- Investigating alternatives to PCA to reduce the dimension of a model such as the **discrete Fourier transform (DFT)** or **singular value decomposition (SVD)** [4:17]
- Using PCA in conjunction with EM (at the research stage)
- Deploying a dataset on a parallel data processing framework such as Apache Hadoop or Spark, as described in the *Apache Spark* section in *Chapter 12, Scalable Frameworks*

Summary

This completes the overview of three of the most commonly used unsupervised learning techniques:

- K-means for clustering fully observed features of a model with reasonable dimensions
- Expectation-maximization for clustering a combination of observed and latent features
- Principal components analysis to transform and extract the most critical features in terms of variance for linear models

Manifold learning for non-linear models is a technically challenging field with great potential in terms of dynamic object recognition [4:18].

The key point to remember is that unsupervised learning techniques are used:

- By themselves to extract structures and associations from unlabelled observations
- As a preprocessing stage to supervised learning in reducing the number of features prior to the training phase

The distinction between unsupervised and supervised learning is not as strict as you may think. For instance, the K-means algorithm can be enhanced to support classification.

In the next chapter, we will address the second use case and cover supervised learning techniques starting with generative models.

5

Naïve Bayes Classifiers

This chapter introduces the most common and simple generative classifiers – Naïve Bayes. As mentioned earlier, generative classifiers are supervised learning algorithms that attempt to fit a *joint probability distribution* $p(X, Y)$ of two X and Y events, representing two sets of observed and hidden (or latent) variables, x and y .

In this chapter, you will learn, and hopefully appreciate, the simplicity of the Naïve Bayes technique through a concrete example. Then, you will learn how to build a Naïve Bayes classifier to predict the stock price movement, given some prior technical indicators in the analysis of financial markets.

Finally, you will learn how to apply Naïve Bayes to text mining by predicting stock prices using financial news feed and press releases.

Probabilistic graphical models

Let's start with a refresher course in basic statistics.

Given two events or observations X and Y , the joint probability of X and Y is defined as $p(X, Y) = p(X \cap Y)$. If the observations X and Y are not related, an assumption known as conditional independence, then $p(X, Y) = p(X).p(Y)$. The conditional probability of an event Y , given X , is defined as $p(Y | X) = p(X, Y)/p(X)$.

These two definitions are quite simple. However, **probabilistic reasoning** can be difficult to read in the case of large numbers of variables and sequences of conditional probabilities. As a picture is worth a thousand words, researchers introduced **graphical models** to describe a probabilistic relation between random variables using graphs [5:1].

There are two categories of graphs, and therefore, graphical models, which are as follows:

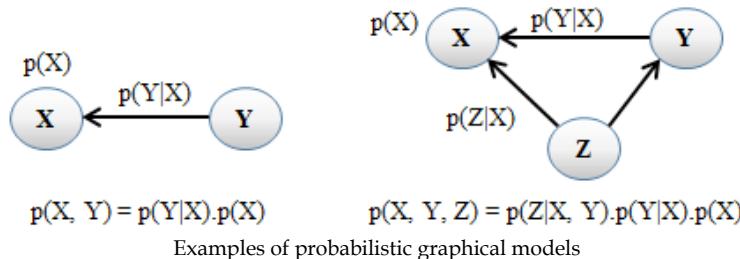
- Directed graphs such as Bayesian networks
- Undirected graphs such as conditional random fields (refer to the *Conditional random fields* section in *Chapter 7, Sequential Data Models*)

Directed graphical models are directed acyclic graphs that have been introduced to:

- Provide a simple way to visualize a probabilistic model
- Describe the conditional dependence between variables
- Represent a statistical inference in terms of connectivity between graphical objects

A Bayesian network is a directed graphical model that defines a joint probability over a set of variables [5:2].

The two joint probabilities $p(X, Y)$ and $p(X, Y, Z)$ can be graphically modeled using Bayesian networks, as follows:



The conditional probability $p(Y|X)$ is represented by an arrow directed from the output (or symptoms) Y to the input (or cause) X . Elaborate models can be described as a large directed graph between variables.



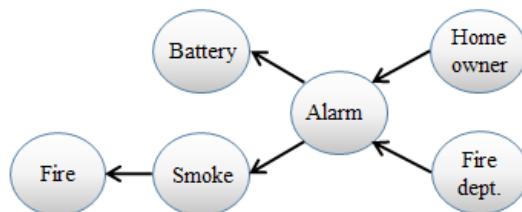
A metaphor for graphical models

From a software engineering perspective, graphical models visualize probabilistic equations in the same way the UML class diagram visualizes the object-oriented source code.

Here is an example of a real-world Bayesian network; the functioning of a smoke detector:

1. A fire may generate smoke.
2. Smoke may trigger an alarm.
3. A depleted battery may trigger an alarm.
4. The alarm may alert the homeowner.
5. The alarm may alert the fire department.

The flow diagram is as follows:



A Bayesian network for smoke detectors

This representation may be a bit counterintuitive, as the vertices are directed from the symptoms (or output) to the cause (or input). Directed graphical models are used in many different models, besides Bayesian networks [5:3].

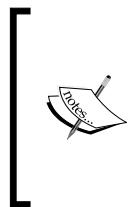


Plate models

There are several alternate representations of probabilistic models, besides the directed acyclic graph, such as the plate model commonly used for the **Latent Dirichlet Allocation (LDA)** [5:4].

The Naïve Bayes models are probabilistic models based on the Bayes's theorem under the assumption of features independence, as mentioned in the *Generative models* section under *Supervised learning* in *Chapter 1, Getting Started*.

Naïve Bayes classifiers

The conditional independence between X features is an essential requirement for the Naïve Bayes classifier. It also restricts its applicability. The Naïve Bayes classification is better understood through simple and concrete examples [5:5].

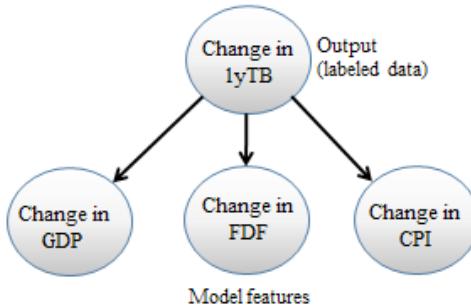
Introducing the multinomial Naïve Bayes

Let's consider the problem of how to predict a change in interest rates.

The first step is to list the factors that potentially may trigger or cause an increase or decrease in the interest rates. For the sake of illustrating Naïve Bayes, we will select the **consumer price index (CPI)** and change the **Federal fund rate (FDF)** and the **gross domestic product (GDP)**, as the first set of features. The terminology is described in the *Terminology* section under *Finances 101* in the *Appendix A, Basic Concepts*.

The use case is used to predict the direction of the change in the yield of the **1-year Treasury bill (1yTB)**, taking into account the change in the current CPI, FDF, and GDP. The objective is, therefore, to create a predictive model using a combination of these three features.

It is assumed that there is no available financial investment expert who can supply rules or policies to predict interest rates. Therefore, the model depends highly on the historical data. Intuitively, if one feature always increases when the yield of the 1-year Treasury bill increases, then we can conclude that there is a strong correlation of causal relationship between the features and the output variation in interest rates.



The Naive Bayes model for predicting the change in the yield of the 1-year T-bill

The correlation (or cause-effect relationship) is derived from the historical data. The methodology consists of counting the number of times each feature either increases (**UP**) or decreases (**DOWN**) and recording the corresponding expected outcome, as illustrated in the following table:

ID	GDP	FDF	CPI	1y-TB
1	UP	DOWN	UP	UP
2	UP	UP	UP	UP
3	DOWN	UP	DOWN	DOWN

4	UP	DOWN	DOWN	DOWN
...				
256	DOWN	DOWN	UP	DOWN

First, let's tabulate the number of occurrences of each change (**UP** and **DOWN**) for the three features and the output value (the direction of the change in the yield of the 1-year Treasury bill):

Number	GDP	FDF	CPI	1yTB
UP	169	184	175	159
DOWN	97	72	81	97
Total	256	256	256	256
UP/Total	0.66	0.72	0.68	0.625

Next, let's compute the number of positive directions for each of the features when the yield of the 1-year Treasury bill increases (159 occurrences):

Number	GDP	Fed Funds	CPI
UP	110	136	127
DOWN	49	23	32
Total	159	159	159
UP/Total	0.69	0.85	0.80

From the preceding table, we conclude that the yield of the 1-year Treasury bill increases when the GDP increases (69 percent of the time), the rate of the Federal funds increases (85 percent of the time), and the CPI increases (80 percent of the time).

Let's formalize the Naïve Bayes model before turning these findings into a probabilistic model.

Formalism

Let's start by clarifying the terminologies used in the Bayesian model:

- **Class prior probability or class prior:** This is the probability of a class
- **Likelihood:** This is the probability to observe a value or event, given a class, also known as the probability of the predictor, given a class
- **Evidence:** This is the probability of observations that occur, also known as the prior probability of the predictor
- **Posterior probability:** This is the probability of an observation x being in a given class

No model can be simpler! The log likelihood, $\log p(x_i | C_j)$, is commonly used instead of the likelihood in order to reduce the impact of the features x_i that have a low likelihood.

The objective of the Naïve Bayes classification of a new observation is to compute the class that has the highest log likelihood. The mathematical notation for the Naïve Bayes model is also straightforward.

The Naïve Bayes classification

M1: The posterior probability $p(C_j | x)$ is defined as:

$$p(C_j | x) = \frac{p(x | C_j) \cdot p(C_j)}{p(x)}$$

Here, $x = \{x_i\} (0, n-1)$ is a set of n features. $\{C_j\}$ is a set of classes with their class prior $p(C_j)$. $x = \{x_i\} (0, n-1)$ with a set of n features. $p(x | C_j)$ is the likelihood for each feature

 M2: The computation of the posterior probability $p(C_j | x)$ is simplified by the assumption of conditional independence of features:

$$p(C_j | x) = \prod_{i=0}^{n-1} p(x_i | C_j) \cdot p(C_j)$$

Here, x_i are independent and the probabilities are normalized for evidence $p(x) = 1$.

M3: The **maximum likelihood estimate (MLE)** is defined as:

$$\mathcal{L}(C_j|x) = \sum_{i=0}^{n-1} \{ \log(p(x_i|C_i)) + \log(p(C_j)) \}$$

 M4: The Naïve Bayes classification of an observation x of a class C_m is defined as:

$$C_m = \arg \max_j \mathcal{L}(C_j|x)$$

This particular use case has a major drawback—the GDP statistics are provided quarterly, while the CPI data is made available once a month and a change in FDF rate is rather infrequent.

The frequentist perspective

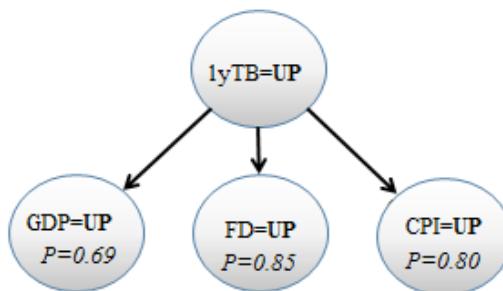
The ability to compute the posterior probability depends on the formulation of the likelihood using historical data. A simple solution is to count the occurrences of observations for each class and compute the frequency.

Let's consider the first example that predicts the direction of the change in the yield of the 1-year Treasury bill, given changes in the GDP, FDF, and CPI.

The results are expressed with simple probabilistic formulas and a directed graphical model:

$$\begin{aligned} P(\text{GDP=UP} | \text{1yTB=UP}) &= 110/159 \\ P(\text{1yTB=UP}) &= \text{num occurrences (1yTB=UP)} / \text{total num of occurrences} = 159/256 \\ p(\text{1yTB=UP} | \text{GDP=UP, FDF=UP, CPI=UP}) &= p(\text{GDP=UP} | \text{1yTB=UP}) \times \\ &\quad p(\text{FDF=UP} | \text{1yTB=UP}) \times \end{aligned}$$

$$p(CPI=UP | 1yTB=UP) \times \\ p(1yTB=UP) = 0.69 \times 0.85 \times 0.80 \times \\ 0.625$$



The Bayesian network for the prediction of the change of the yield of the 1-year Treasury bill

 **Overfitting**
The Naïve Bayes model is not immune to overfitting if the number of observations is not large enough relative to the number of features. One approach to address this problem is to perform a feature selection using the mutual information exclusion [5:6].

This problem is not a good candidate for a Bayesian classification for the following two reasons:

- The training set is not large enough to compute accurate prior probabilities and generate a stable model. Decades of quarterly GDP data is needed to train and validate the model.
- The features have different rates of change, which predominately favor the features with the highest frequency; in this case, the CPI.

Let's select another use case for which a large historical dataset is available and can be automatically labeled.

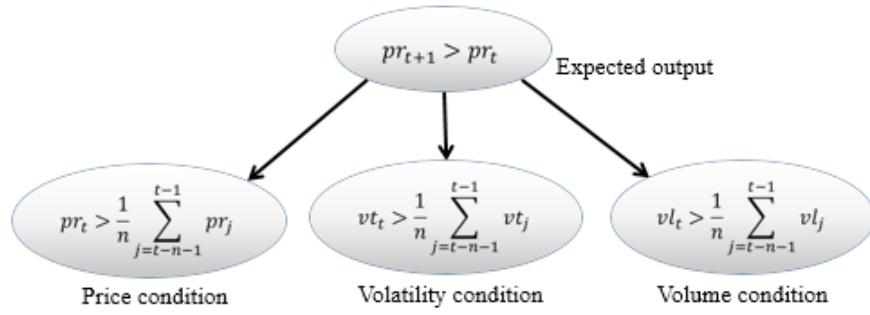
The predictive model

The predictive model is the second use case that consists of predicting the direction of the change of the closing price of a stock, $pr_{t+1} = \{UP, DOWN\}$, at trading day $t + 1$, given the history of its direction of the price, volume, and volatility for the previous t days, pr_i for $i = 0$ to $i = t$. The volume and volatility features have already been used in the *Writing a simple workflow* section in *Chapter 1, Getting Started*.

Therefore, the three features under consideration are as follows:

- The closing price pr_t of the last trading session, t , is above or below the average closing price over the n previous trading days, $[t-n, t]$.
- The volume of the last trading day vl_t is above or below the average volume of the n previous trading days
- The volatility on the last trading day vt_t is above or below the average volatility of the previous n trading days

The directed graphic model can be expressed using one output variable (the price at session $t + 1$ is greater than the price at session t) and three features: the price condition (1), volume condition (2), and volatility condition (3).



The Bayesian model for predicting the future direction of the stock price

This model works under the assumption that there is at least one observation or ideally few observations for each feature and expected value.

The zero-frequency problem

It is possible that the training set does not contain any data actually observed for a feature for a specific label or class. In this case, the mean is $0/N = 0$, and therefore, the likelihood is null, making classification unfeasible. The case for which there are only few observations for a feature in a given class is also an issue, as it skews the likelihood.

There are a couple of correcting or smoothing formulas for unobserved features or features with a low number of occurrences that address this issue, such as the **Laplace** and **Lidstone** smoothing formulas.

The smoothing factor for counters

M5: The Laplace smoothing formula of the mean k/N out of N observations of features of dimension n is defined as:



$$\mu' = \frac{k + 1}{N + n}$$

M6: The Lidstone smoothing formula with a factor a is defined as:

$$\mu' = \frac{k + a}{N + a.n}$$

The two formulas are commonly used in natural language processing applications, for which the occurrence of a specific word or tag is a feature [5:7].

Implementation

I think it is time to write some Scala code and toy around with Naïve Bayes. Let's start with an overview of the software components.

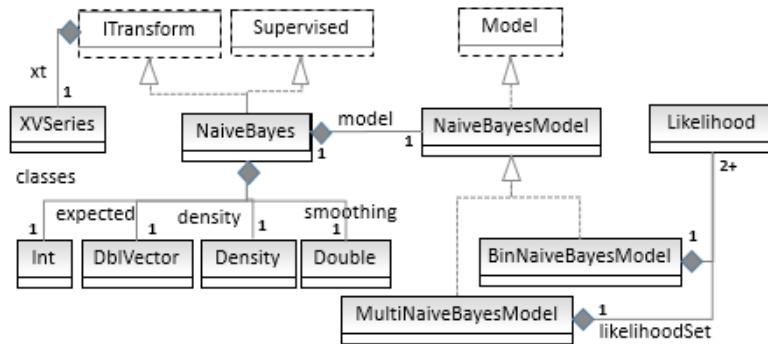
Design

Our implementation of the Naïve Bayes classifier uses the following components:

- A generic model, `NaiveBayesModel`, of the `Model` type that is initialized through training during the instantiation of the class.
- A model for the `BinNaiveBayesModel` binomial classification, which subclasses `NaiveBayesModel`. The model consists of a pair of positive and negative `Likelihood` class instances.
- A model for the `MultiNaiveBayesModel` multinomial classification.
- The `NaiveBayes` classifier class has four parameters: a smoothing function, such as `Laplace` and a set of observations of the `xvSeries` type, a set of labels of the `DblVector` type, a log density function of the `LogDensity` type, and the number of classes.

The principle of software architecture applied to the implementation of classifiers is described in the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*.

The key software components of the Naïve Bayes classifier are described in the following UML class diagram:



The UML class diagram for the Naïve Bayes classifier

The UML diagram omits the helper traits or classes such as `Monitor` or `Apache Commons Math` components.

Training

The objective of the training phase is to build a model consisting of the likelihood for each feature and the class prior. The likelihood for a feature is identified as follows:

- The number of occurrences k of this feature for $N > k$ observations in the case of binary features or counters
- The mean value for all the observations for this feature in the case of numeric or continuous features

It is assumed, for the sake of this test case, that the features, that is, technical analysis indicators, such as price, volume, and volatility are conditionally independent. This assumption is not actually correct.

Conditional dependency

Recent models, known as **Hidden Naïve Bayes (HNB)**, relax the restrictions on the independence between features. The HNB algorithm uses conditional mutual information to describe the interdependency between some of the features [5:8].

Let's write the code to train the binomial and multinomial Naïve Bayes.

Class likelihood

The first step is to define the class likelihood for each feature using historical data.

The `Likelihood` class has the following attributes (line 1):

- The label for the `label` observation
- An array of tuple Laplace or Lidstone smoothed mean and standard deviation, `muSigma`
- The prior probability of a prior class

As with any code snippet presented in this book, the validation of class parameters and method arguments are omitted in order to keep the code readable:

```
class Likelihood[T <: AnyVal] {  
    val label: Int,  
    val muSigma: Vector[DblPair],  
    val prior: Double) (implicit f: T => Double) { //1  
  
    def score(obs: Array[T], logDensity: LogDensity): Double = //2  
        (obs, muSigma).zipped  
        .map{ case(x, (mu, sig)) => (x, mu, sig)}  
        .//:(0.0)((prob, entry) => {  
            val x = entry._1  
            val mean = entry._2  
            val stdDev = entry._3  
            val logLikelihood = logDensity(mean, stdDev, x) //3  
            val adjLogLikelihood = if(logLikelihood < MINLOGARG)  
                MINLOGVALUE else logLikelihood  
            prob + Math.log(adjLogLikelihood) //4  
        }) + Math.log(prior)  
    }  
}
```

The parameterized `Likelihood` class has the following two purposes:

- Define the statistics regarding a class C_k : its label, its mean and standard deviation, and the prior probability $p(C_k)$.
- Compute the score of a new observation for its runtime classification (line 2). The computation of the log of the likelihood uses a `logDensity` method of the `LogDensity` type (line 3). As seen in the next section, the log density can be either a Gaussian or a Bernoulli distribution. The `score` method uses Scala's `zipped` method to merge the observation values with the labeled values and implements the M3 formula (line 4).

The Gaussian mixture is particularly suited for modeling datasets, for which the features have large sets of discrete values or are continuous variables. The conditional probabilities for the feature x is described by the normal probability density function [5:9].

The log likelihood using the Gaussian density

M7: For a Lidstone or Laplace smoothed mean μ' and a standard deviation σ , the log likelihood of a posterior probability for a Gaussian distribution is defined as:

$$\mathcal{L}(C_j|x) = \sum_{i=0}^{n-1} \left\{ -\frac{1}{2} \log(2\pi) - \log(\sigma) - \frac{(x - \mu')^2}{2\sigma^2} + \log(p(C_j)) \right\}$$

The log of the Gauss, `logGauss`, and the log of the Normal distribution, `logNormal`, are defined in the `stats` class, which was introduced in the *Profiling data* section in *Chapter 2, Hello World!*:

```
def logGauss(mean: Double, stdDev: Double, x: Double): Double = {
    val y = (x - mean)/stdDev
    -LOG_2PI - Math.log(stdDev) - 0.5*y*y
}
val logNormal = logGauss(0.0, 1.0, _: Double)
```

The `logNormal` computation is implemented as a partial applied function.

The functions of the `LogDensity` type compute the probability density for each feature (line 5):

```
type LogDensity = (Double*) => Double
```

Binomial model

The next step is to define the `BinNaiveBayesModel` model for a two-class classification scheme. The two-class model consists of two `Likelihood` instances: positives for the label UP (value 1) and negatives for the label DOWN (value 0).

In order to make the model generic, we created a `NaiveBayesModel` trait that can be extended as needed to support both the binomial and multinomial Naïve Bayes models, as follows:

```
trait NaiveBayesModel[T] {
    def classify(x: Array[T], logDensity: LogDensity): Int //5
}
```

The `classify` method uses the trained model to classify a multivariate observation x of the `Array[T]` type given a `logDensity` probability density function (line 5). The method returns the class the observation belongs to.

Let's start with the definition of the `BinNaiveBayesModel` class that implements the binomial Naïve Bayes:

```
class BinNaiveBayesModel[T <: AnyVal] (
    pos: Likelihood[T],
    neg: Likelihood[T]) (implicit f: T => Double)
extends NaiveBayesModel[T] { //6

    override def classify(x: Array[T], logDensity: logDensity): Int = //7
        if(pos.score(x,density) > neg.score(x,density)) 1 else 0
    ...
}
```

The constructor for the `BinNaiveBayesModel` class takes two arguments:

- `pos`: Class likelihood for observations with a positive outcome
- `neg`: Class likelihood for observations with a negative outcome (line 6)

The `classify` method is called by the `|>` operator in the Naïve Bayes classifier. It returns 1 if the observation x matches the `Likelihood` class that contains the positive cases, and 0 otherwise (line 7).

Model validation

The parameters of the Naïve Bayes model (likelihood) are computed through training and the `model` value is instantiated regardless of whether the model is actually validated in this example. A commercial application would require the model to be validated using a methodology such as the K-fold validation and F1 measure, as described in the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*.

The multinomial model

The multinomial Naïve Bayes model defined by the `MultiNaiveBayesModel` class is very similar to the `BinNaiveBayesModel`:

```
class MultiNaiveBayesModel[T <: AnyVal] //8
    likelihoodSet: Seq[Likelihood[T]]) (implicit f: T => Double)
extends NaiveBayesModel[T] {
```

```

override def classify(x: Array[T], logDensity: LogDensity): Int = {
    val <<< = (p1: Likelihood[T], p2: Likelihood[T]) =>
        p1.score(x, density) > p1.score(x, density) //9
    likelihoodSet.sortWith(<<<).head.label //10
}
...
}

```

The multinomial Naïve Bayes model differs from its binomial counterpart as follows:

- Its constructor requires a sequence of class likelihood `likelihoodSet` (line 8).
- The `classify` runtime classification method sorts the class likelihoods by their score (posterior probability) using the `<<<` function (line 9). The method returns the ID of the class with the highest log likelihood (line 10).

Classifier components

The Naïve Bayes algorithm is implemented as a data transformation using a model implicitly extracted from a training set of the `ITransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*

The attributes of the multinomial Naïve Bayes are as follows:

- The smoothing formula (Laplace, Lidstone, and so on), `smoothing`
- The set of multivariable observations defined as `xt`
- The expected values (or labels) associated with the set of observations, `expected`
- The log of the probability density function, `logDensity`
- The number of classes—two for the binomial Naïve Bayes with the `BinNaiveBayesModel` type or more for the multinomial Naïve Bayes with the `MultiNaiveBayesModel` type (line 11)

The code will be as follows:

```

class NaiveBayes[T <: AnyVal] (
    smoothing: Double,
    xt: XVSeries[T],
    expected: Vector[Int],
    logDensity: LogDensity,
    classes: Int)(implicit f: T => Double)
extends ITransform[Array[T]](xt)

```

```
    with Supervised[T, Array[T]] with Monitor[Double] { //11
  type V = Int //12
  val model: Option[NaiveBayesModel[T]] //13
  def train(expected: Int): Likelihood[T]
  ...
}
```

The `Monitor` trait defines miscellaneous logging and display functions.

Data transformation of the `ITransform` type requires the output type `v` to be specified (line 12). The output of the Naïve Bayes is the index of the class an observation belongs to. The `model` type of the model can be either `BinNaiveBayesModel` for two classes or `MultiNaiveBayesModel` for a multinomial model (line 13):

```
val model: Option[NaiveBayesModel[T]] = Try {
  if(classes == 2)
    BinNaiveBayesModel[T](train(1), train(0))
  else
    MultiNaiveBayesModel[T](List.tabulate(classes)(train(_)))
} match {
  case Success(_model) => Some(_model)
  case Failure(e) => /* ... */
}
```

Training and class instantiation

There are several benefits of allowing the instantiation of the Naïve Bayes mode only once when it is trained. It prevents the client code from invoking the algorithm on an untrained or partially trained model, and it reduces the number of states of the model (untrained, partially trained, trained, validated, and so on). It is an elegant way to hide the details of the training of the model from the user.

The `train` method is applied to each class. It takes the index or label of the class and generates its log likelihood data (line 14):

```
def train(index: Int): Likelihood[T] = { //14
  val xv: XVSeries[Double] = xt
  val values = xv.zip(expected) //15
    .filter(_._2 == index).map(_._1) //16
  if(values.isEmpty)
    throw new IllegalStateException(/* ... */)
```

```

    val dim = dimension(xt)
    val meanStd = statistics(values).map(stat =>
      (stat.lidstoneMean(smoothing, dim), stat.stdDev)) //17
    Likelihood(index, meanStd, values.size.toDouble/xv.size) //18
  }

```

The training set is generated by zipping the `xt` vector of observations with expected classes, `expected` (line 15). The method filters out the observation for which the label does not correspond to this class (line 16). The `meanStd` pair (mean and standard deviation) is computed using the Lidstone smoothing factor (line 17). Finally, the training method returns the class likelihood corresponding to the index `label` (line 18).

The `NaiveBayes` class also defines the `|>` runtime classification method and the F_1 validation methods. Both methods are described in the next section.



Handling missing data

Naïve Bayes has a no-nonsense approach to handling missing data. You just ignore the attribute in the observations for which the value is missing. In this case, the prior for this particular attribute for these observations is not computed. This workaround is obviously made possible because of the conditional independence between features.

The `apply` constructor for `NaiveBayes` returns the `NaiveBayes` type:

```

object NaiveBayes {
  def apply[T <: AnyVal] (
    smoothing: Double,
    xt: XVSeries[T],
    expected: Vector[Int],
    logDensity: LogDensity,
    classes: Int) (implicit f: T => Double): NaiveBayes[T] =
    new NaiveBayes[T](smoothing, xt, y, logDensity, classes)

  ...
}

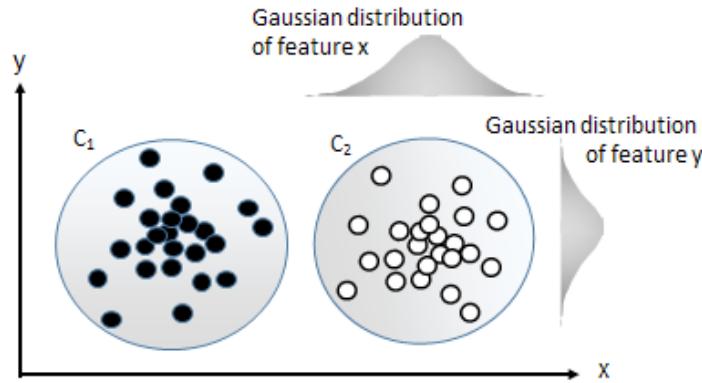
```

Classification

The likelihood and class prior that have been computed through training is used for validating the model and classifying new observations.

The score represents the log of likelihood estimate (or the posterior probability), which is computed as the summation of the log of the Gaussian distribution using the mean and standard deviation extracted from the training phase and the log of the class likelihood.

The Naïve Bayes classification using the Gaussian distribution is illustrated using the two C_1 and C_2 classes and a model with two features (x and y):



An illustration of the Gaussian Naive Bayes using 2-dimensional model

The `|>` method returns the partial function that implements the runtime classification of a new x observation using one of the two Naïve Bayes models. The `model` and the `logDensity` functions are used to assign the x observation to the appropriate class (line 19):

```
override def |> : PartialFunction[Array[T], Try[V]] = {  
    case x: Array[T] if(x.length >0 && model != None) =>  
        Try(model.map(_.classify(x, logDensity)).get) //19  
}
```

F₁ validation

Finally, the Naïve Bayes classifier is implemented by the `NaiveBayes` class. It implements the training and runtime classification using the Naïve Bayes formula. In order to force the developer to define a validation for any new supervised learning technique, the class inherits from the `Supervised` trait that declares the `validate` validation method:

```
trait Supervised[T, V] {  
    self: ITransform[V] => //20  
    def validate(xt: XVSeries[T],  
                expected: Vector[V]): Try[Double] //21  
}
```

The validation of a model applies only to a data transformation of the `ITransform` type (line 20).

The `validate` method takes the following arguments (line 21):

- An `xt` time series of multidimensional observations
- An `expected` vector of expected class values

By default, the `validate` method returns the F_1 score for the model, as described in the *Assessing a model* section in *Chapter 2, Hello World!*

Let's implement the key functionality of the `Supervised` trait for the Naïve Bayes classifier:

```
override def validate(
    xt: XVSeries[T],
    expected: Vector[V]): Try[Double] = Try {           //22
  val predict = model.get.classify(_:Array[Int], logDensity) //23
  MultiFValidation(expected, xt, classes)(predict).score //24
}
```

The predictive `predict` partially applied function is created by assigning a predicted class to a new `x` observation (line 23), and then the prediction, the index of classes, is loaded into the `MultiFValidation` class to compute the F_1 score (line 24).

Feature extraction

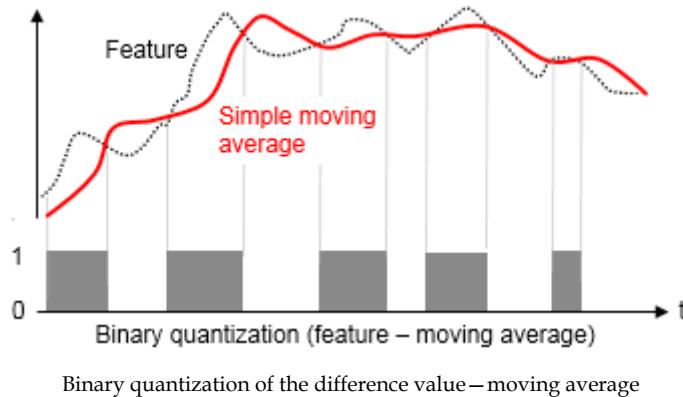
The most critical element in the training of a supervised learning algorithm is the creation of labeled data. Fortunately, in this case, the labels (or expected classes) can be automatically generated. The objective is to predict the direction of the price of a stock for the next trading day, taking into account the moving average price, volume, and volatility over the last n days.

The extraction of features follows these six steps:

1. Extract historical trading data of each feature (that is, price, volume, and volatility).
2. Compute the simple moving average of each feature.
3. Compute the difference between the value and moving average of each feature.
4. Normalize the difference by assigning 1 for positive values and 0 for negative values.

5. Generate a time series of the difference between the closing price of the stock and the closing price of the previous trading session.
6. Normalize the difference by assigning 1 for positive values and 0 for negative values.

The following diagram illustrates the feature extractions for steps 1 to 4:



The first step is to extract the average price, volume, and volatility (that is, $1 - \text{low}/\text{high}$) for each stock during the period of Jan 1, 2000 and Dec 31, 2014 with daily and weekly closing prices. Let's use the simple moving average to compute these averages for the $[t-n, t]$ window.

The extractor variable defines the list of features to extract from the financial data source, as described in the *Data extraction* and *Data sources* section in the *Appendix A, Basic Concepts*:

```
val extractor = toDouble(CLOSE)    // stock closing price
                     :: ratio(HIGH, LOW) //volatility(HIGH-LOW)/HIGH
                     :: toDouble(VOLUME)   // daily stock trading volume
                     :: List[Array[String] =>Double] ()
```

The naming convention for the trading data and metrics is described in the *Trading data* section under *Technical analysis* in the *Appendix A, Basic Concepts*.

The training and validation of the binomial Naïve Bayes is implemented using a monadic composition:

```
val trainRatio = 0.8    //25
val period = 4
val symbol ="IBM"
val path = "resources/chap5"
```

```

val pfnMv = SimpleMovingAverage[Double](period, false) |> //26
val pfnSrc = DataSource(symbol, path, true, 1) |> //27

for {
    obs <- pfnSrc(extractor) //28
    (x, delta) <- computeDeltas(obs) //29
    expected <- Try{difference(x.head.toVector, diffInt)}//30
    features <- Try { transpose(delta) } //31
    labeled <- //32
    OneFoldXValidation[Int](features, expected, trainRatio)
    nb <- NaiveBayes[Int](1.0, labeled.trainingSet) //33
    f1Score <- nb.validate(labeled.validationSet) //34
}
yield {
    val labels = Array[String](
        "price/ave price", "volatility/ave. volatility",
        "volume/ave. volume"
    )
    show(s"\nModel: ${nb.toString(labels)}")
}

```

The first step is to distribute the observations between the training set and validation set. The `trainRatio` value (line 25) defines the ratio of the original observation set to be included in the training set. The simple moving average values are generated by the `pfnMv` partial function (line 26). The extracting `pfnSrc` partial function (line 27) is used to generate the three trading time series, price, volatility, and volume (line 28).

The next step consists of applying the `pfnMv` simple moving average to the `obs` multidimensional time series (line 29) using the `computeRatios` method:

```

type LabeledPairs = (XVSeries[Double], Vector[Array[Int]])

def computeDeltas(obs: XVSeries[Double]): Try[LabeledPairs] =
Try{
    val sm = obs.map(_.toVector).map( pfnMv(_).get.toArray) //35
    val x = obs.map(_.drop(period-1) )
    (x, x.zip(sm).map{ case(x,y) => x.zip(y).map(delta(_)) })//36
}

```

The `computeDeltas` method computes the time series of the `sm` observations smoothed with a simple moving average (line 35). The method generates a time series of 0 and 1 for each of the three features in the `xs` observation set and `sm` smoothed dataset (line 36).

Next, the call to the `difference` differential computation generates the labels (0 and 1) representing the change in the direction of the price of a security between two consecutive trading sessions: 0 if the price is decreased and 1 if the price is increased (line 30) (refer to the *The differential operator* section under *Time series in Scala* in *Chapter 3, Data Preprocessing*).

The features for the training of the Naïve Bayes model are extracted from these ratios by transposing the ratios-time series matrix in the `transpose` method of the `XTSerie`s singleton (line 31).

Next, the training set and validation set are extracted from the `features` set using the `OneFoldXValidation` class, which was introduced in the *One-fold cross validation* section under *Cross-validation* in *Chapter 2, Hello World!* (line 32).

Selecting the training data

In our example, the training set is simplistically the first `trainRatio` multiplied by the size of dataset observations. Practical applications use a K-fold cross-validation technique to validate models, as described in the *K-fold cross validation* section under *Assessing a model* in *Chapter 2, Hello World!*. A simpler alternative is to create the training set by picking observations randomly and using the remaining data for validation.

The last two stages in the workflow consists of training the Naïve Bayes model by instantiating the `NaiveBayes` class (line 33) and computing the F_1 score for different values of the smoothing coefficient of the simple moving average applied to the stock price, volatility, and volume (line 34).

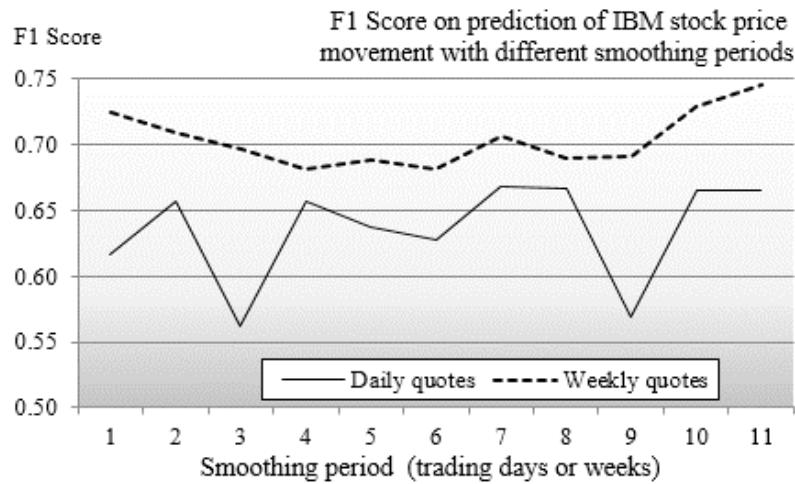
The implicit conversion

The `NaiveBayes` class operates on elements of the `Int` and `Double` types, and therefore, assumes that there is conversion between `Int` and `Double` (view bounded). The Scala compiler may generate a warning because the conversion from `Int` to `Double` has not been defined. Although Scala relies on its own conversion functions, I would recommend that you explicitly define and control your conversion function:

```
implicit def intToDouble(n: Int): Double = n.toDouble
```

Testing

The next chart plots the value of the F_1 measure of the predictor of the direction of the IBM stock using price, volume, and volatility over the previous n trading days, with n varying from 1 to 12 trading days:



A graph of the F1-measure for the validation of the Naïve Bayes model

The preceding chart illustrates the impact of the value of the averaging period (number of trading days) on the quality of the multinomial Naïve Bayes prediction, using the value of the stock price, volatility, and volume relative to their average over the averaging period.

From this experiment, we conclude the following:

- The prediction of the stock movement using the average price, volume, and volatility is not very good. The F_1 score for the models using weekly (with respect to daily) closing prices varies between 0.68 and 0.74 (with respect to 0.56 and 0.66).
- The prediction using weekly closing prices is more accurate than the prediction using the daily closing prices. In this particular example, the distribution of the weekly closing prices is more reflective of an intermediate term trend than the distribution of daily prices.
- The prediction is somewhat independent of the period used to average the features.

The Multivariate Bernoulli classification

The previous example uses the Gaussian distribution for features that are essentially binary ($UP = 1$ and $DOWN = 0$) to represent the change in value. The mean value is computed as the ratio of the number of observations for which $x_i = UP$ over the total number of observations.

As stated in the first section, the Gaussian distribution is more appropriate for either continuous features or binary features for very large labeled datasets. The example is the perfect candidate for the **Bernoulli** model.

Model

The Bernoulli model differs from the Naïve Bayes classifier in such a way that it penalizes the feature x that does not have any observation; the Naïve Bayes classifier ignores it [5:10].

The Bernoulli mixture model

M8: For a feature function f_k with $f_k = 1$, if the feature is observed, and a value of 0 otherwise, and the probability p of the observed feature x_k belongs to the class C_j , then the posterior probability is computed as follows:

$$p(f_i|C_j) = \prod_{k=0}^{n-1} (f_k \cdot p(x_k|C_j) + (1 - f_k)(1 - p(x_k|C_j))$$

Implementation

The implementation of the Bernoulli model consists of modifying the `score` function in the `Likelihood` class using the Bernoulli density method, `bernoulli`, defined in the `Stats` object:

```
object Stats {  
    def bernoulli(mean: Double, p: Int): Double =  
        mean*p + (1-mean)*(1-p)  
    def bernoulli(x: Double*): Double = bernoulli(x(0), x(1).toInt)  
    ...  
}
```

The first version of the Bernoulli algorithm is the direct implementation of the M8 mathematical formula. The second version uses the signature of the `Density` (`Double*`) \Rightarrow `Double` type.

The mean value is the same as in the Gaussian density function. The binary feature is implemented as an `Int` type with the value `UP = 1` (with respect to `DOWN = 0`) for the upward (with respect to downward) direction of the financial technical indicator.

Naïve Bayes and text mining

The multinomial Naïve Bayes classifier is particularly suited for **text mining**. The Naïve Bayes formula is quite effective to classify the following entities:

- E-mail spams
- Business news stories
- Movie reviews
- Technical papers as per field of expertise

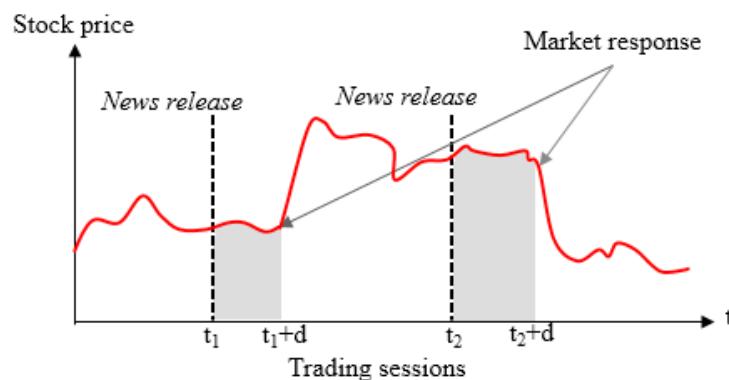
This third use case consists of predicting the direction of a stock given the financial news. There are two type of news that affect the stock of a particular company:

- **Macro trends:** Economic or social news such as conflicts, economic trends, or labor market statistics
- **Micro updates:** Financial or market news related to a specific company such as earnings, change in ownership, or press releases

Macroeconomic news related to a specific company have the potential to affect the sentiments of investors toward the company and may lead to a sudden shift in the price of its stock. Another important feature is the average time it takes for investors to react to the news and affect the price of the stock.

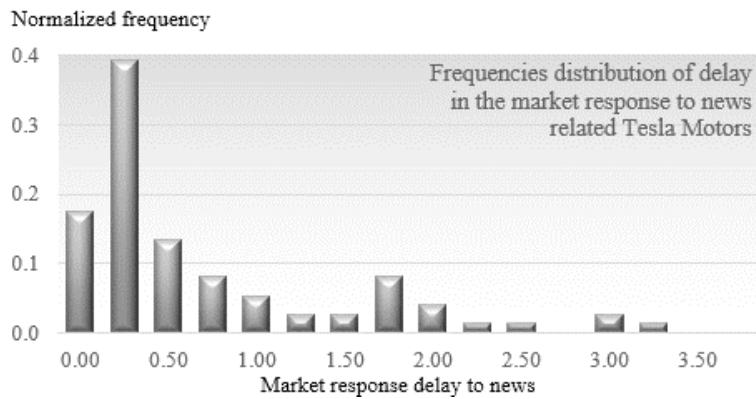
- Long-term investors may react within days or even weeks
- Short-term traders adjust their positions within hours, sometimes within the same trading session

The average time the market takes to react to a significant financial news on a company is illustrated in the following chart:



An illustration of the reaction of investors on the price of a stock following a news release

The delay in the market response is a relevant feature only if the variance of the response time is significant. The distribution of the frequencies of the delay in the market response to any newsworthy articles regarding TSLA is fairly constant. It shows that the stock prices react within the same day in 82 percent of the cases, as seen in the following bar chart:

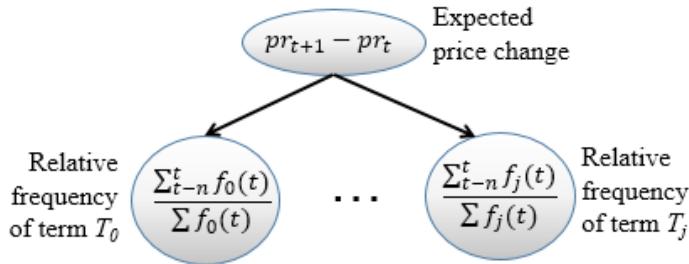


The distribution of the frequencies of the reaction of investors on the price of a stock following a news release

The frequency peak for a market response delay of 1.75 days can be explained by the fact that some news are released over the weekend and investors have to wait until the following Monday to drive the stock price higher or lower. Another challenge is to assign any shift of a stock price to a specific news release, taking into account that some news can be redundant, confusing, or simultaneous.

Therefore, the model features for predicting the stock price pr_{t+1} are the relative frequency f_i of an occurrence of a term T_i within a time window $[t-n, t]$, where t and n are trading days.

The following graphical model formally describes the causal relation or conditional dependency of the relative change of the stock price between two consecutive trading sessions t and $t + 1$, given the relative frequency of appearance of some key terms in the media:



The Bayesian model for the prediction of the stock movement given financial news

For this exercise, the observation sets are the corpus of news feeds and articles released by the most prominent financial news organizations, such as Bloomberg or CNBC. The first step is to devise a methodology to extract and select the most relevant terms associated with a specific stock.

Basics of information retrieval

A full discussion of information retrieval and text mining is beyond the scope of this book [5:11]. For the sake of simplicity, the model will rely on a very simple scheme for extracting relevant terms and computing their relative frequency. The following 10-step sequence of actions describe one of the numerous methodologies used to extract the most relevant terms from a corpus:

1. Create or extract the timestamp for each news article.
2. Extract the title, paragraph, and sentences of each article using a Markovian classifier.
3. Extract the terms from each sentence using regular expressions.
4. Correct terms for typos using a dictionary and metric such as the **Levenshtein** distance.
5. Remove the nonstop words.
6. Perform **stemming** and **lemmatization**.
7. Extract bags of words and generate a list of **n-grams** (as a sequence of n terms).
8. Apply a **tagging model** build using a maximum entropy or conditional random field to extract nouns and adjectives (for example, *NN*, *NNP*, and so on).

9. Match the terms against a dictionary that supports senses, hyponyms, and synonyms, such as **WordNet**.
10. Disambiguate word sense using Wikipedia's repository **DBpedia** [5:12].

 **Text extraction from the Web**

The methodology discussed in this section does not include the process of searching and extracting news and articles from the Web that requires additional steps such as search, crawling, and scraping [5:13].

Implementation

Let's apply the text mining methodology template to predict the direction of a stock, given the financial news. The algorithm relies on a sequence of seven simple steps:

1. Searching and loading the news articles related to a given company and its stock as a D_t document of the Document type.
2. Extracting the date: T timestamp of the article using a regular expression.
3. Ordering the D_t documents as per the timestamp.
4. Extracting the $\{T_{i,D}\}$ terms from the content of each D_t document.
5. Aggregating the $\{T_{t,D}\}$ terms for all the D_t documents that share the same publication date t .
6. Computing the rtf relative frequency of each $\{T_{i,D}\}$ term for the date t , as the ratio of number of its occurrences in all the articles released at t to the total number of its occurrences of the term in the entire corpus.
7. Normalizing the relative frequency for the average number of articles per date, $nrtf$.

Text analysis metrics

M9: The relative frequency of occurrences for term (or keyword) t_i with n_i^a occurrences in the article a is defined as follows:

$$rtf(t_i) = \frac{\sum_{a \in Dt} n_i^a}{\sum_{a \in Corpus} n_i^a}$$

 M10: The relative frequency of occurrences of a term t_i normalized by the daily average number of articles for which N_a is the total number of articles and N_d is the number of days in the survey is defined as follows:

$$nrft(t_i) = \frac{rtf(t_i) \cdot N_d}{N_a}$$

The news articles are *minimalist* documents with a timestamp, title, and content, as implemented by the Document class:

```
case class Document[T <: AnyVal] ( //1
  date: T, title: String, content: String)
  (implicit f: T => Double)
```

The date timestamp has a type bounded to the `Long` type, so `T` can be converted to the current time in milliseconds of the JVM (line 1).

Analyzing documents

This section is dedicated to the implementation of the simple text analyzer. Its purpose is to convert a set of documents of the `Document` type; in our case, news articles, into a distribution of relative frequencies of keywords.

The `TextAnalyzer` class implements a data transformation of the `ETransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*. It transforms a sequence of documents into a sequence of relative frequency distribution.

The `TextAnalyzer` class has the following two arguments (line 4):

- A simple text parser, `parser`, that extracts an array of keywords from the title and content of each news articles (line 2).
- A `lexicon` type that lists keywords used in monitoring news related to a company and their synonyms. The synonyms or terms that are semantically similar to each keywords are defined in an immutable map.

The code will be as follows:

```
type TermsRF = Map[String, Double]
type TextParser = String => Array[String] //2
type Lexicon = immutable.Map[String, String] //3
type Corpus[T] = Seq[Document[T]]

class TextAnalyzer[T <: AnyVal] ( //4
    parser: TextParser,
    lexicon: Lexicon)(implicit f: T => Double)
  extends ETransform[Lexicon](lexicon) {

  type U = Corpus[T] //5
  type V = Seq[TermsRF] //6

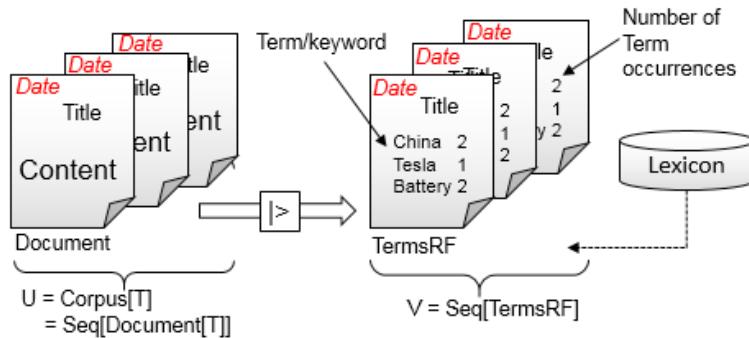
  override def |> : PartialFunction[U, Try[V]] = {
    case docs: U => Try(score(docs))
  }

  def score(corpus: Corpus[T]): Seq[TermsRF] //7
  def quantize(termsRFSeq: Seq[TermsRF]): //8
    Try[(Array[String], XVSeries[Double])]
  def count(term: String): Counter[String] //9
}
```

The `U` type of an input into the data transformation `|>` is the corpus or sequence of news articles (line 5). The `V` type of the output from the data transformation is the sequence of relative frequency distribution of the `TermsRF` type (line 6).

The `score` private method does the heavy lifting for the class (line 7). The `quantize` method creates a homogenous set of observed features (line 8) and the `count` method counts the number of occurrences of terms or keywords across the documents or news articles that share the same publication date (line 9).

The following diagram describes the different components of the text mining process:



An illustration of the components of the text mining procedure

Extracting the frequency of relative terms

Let's dive into the score method:

```
def score(corpus: Corpus[T]): Seq[TermsRF] = {
    val termsCount = corpus.map(doc => //10
        (doc.date, count(doc.content))) //Seq[(T, Counter[String])]

    val termsCountMap = termsCount.groupBy(_._1).map{
        case (t, seq) => (t, seq.aggregate(new Counter[String])
            ((s, cnt) => s ++ cnt._2, _ ++ _)) //11
    }
    val termsCountPerDate = termsCountMap.toSeq
        .sortWith(_._1 < _._1).unzip._2 //12
    val allTermsCounts = termsCountPerDate
        .aggregate(new Counter[String])((s, cnt) =>
            s ++ cnt, _ ++ _) //13

    termsCountPerDate.map(_ /allTermsCounts).map(_.toMap) //14
}
```

The first step in the execution of the `score` method is the computation of the number of occurrences of keywords of the `lexicon` type on each of the document/news article (line 10). The computation of the number of occurrences is implemented by the `count` method:

```
def count(term: String): Counter[String] =
    parser(term) ./:(new Counter[String])((cnt, w) => //16
        if(lexicon.contains(w)) cnt + lexicon(w) else cnt)
```

The method relies on the term Counter counting class that subclasses `mutable.Map[String, Int]`, as described in the *Counter* section under *Scala programming* in the *Appendix A, Basic Concepts*. It uses a fold to update the count for each of the terms associated with a keyword (line 16). The `count` term for the entire corpus is computed by aggregating the terms count for all the documents (line 11).

The next step consists of aggregating the count of the keywords across the document for each timestamp. A `termsCountMap` map with the date as the key and keywords counter, as values are generated by invoking the `groupBy` higher-order method (line 11). Next, the `score` method extracts a sorted sequence of keywords counts, `termsCountPerDate` (line 12). The total counts for each keyword over the `allTermsCounts` entire corpus (line 13) is used to compute the relative or normalized keywords frequencies (formulas **M9** and **M10**) (line 14).

Generating the features

There is no guarantee that all the news articles associated with a specific publication date are used in the model. The `quantize` method assigns a relative frequency of **0.0** for keywords that are missing from the news articles, as illustrated in the following table:

	Keyword 1	Keyword 2	Keyword 3	...	Keyword N
Date 1	0.42	0.00	0.07		0.23
Date 2	0.00	0.11	0.18		0.04
...					
Date J	0.13	0.29	0.00		0.00

A table on relative frequencies of keywords as per the publishing date

The `quantize` method transforms a sequence of term-relative frequencies into a pair keywords and observations:

```
def quantize(termsRFSeq: Seq[TermsRF]): Try[(Array[String], XVSeries[Double])] = Try {
    val keywords = lexicon.values.toArray.distinct //15
    val relFrequencies =
        termsRFSeq.map( tf => //16
            keywords.map(key =>
                if(tf.contains(key)) tf.get(key).get else 0.0))
    (keywords, relFrequencies.toVector) //17
}
```

The `quantize` method extracts an array of keywords from the lexicon (line 15). The `relFrequencies` vector of features is generated by assigning the relative 0.0 keyword frequency for keywords that are not detected across the news articles published at a specific date (line 16). Finally, the key-value pair (keywords and relative keyword frequency) (line 17).

Sparse relative frequencies vector

Text analysis and natural language processing deals with very large feature sets, with potentially hundreds of thousands of features or keywords. Such computations would be almost intractable if it was not for the fact that the vast majority of keywords are not present in each document. It is a common practice to use sparse vectors and sparse matrices to reduce the memory consumption during training.

Testing

For testing purpose, let's select the news articles that mention Tesla Motors and its ticker symbol TSLA over a period of 2 months.

Retrieving the textual information

Let's start implementing and defining the two components of `TextAnalyzer`: the `parsing` function and the `lexicon` variable:

```
val pathLexicon = "resources/text/lexicon.txt"
val LEXICON = loadLexicon //18

def parse(content: String): Array[String] = {
    val regExpr = "[ '|'.|.|?|!|:|\"|]"
    content.trim.toLowerCase.replace(regExpr, " ") //19
    .split(" ") //20
    .filter(_.length > 2) //21
}
```

The lexicon is loaded from a file (line 18). The `parse` method uses a simple `regExpr` regular expression to replace any punctuation into a space character (line 19), which is used as a word delimiter (line 20). All the words shorter than three characters are discounted (line 21).

Let's describe the workflow to load, parse, and analyze news articles related to the company, Tesla Inc. and its stock, ticker symbol TSLA.

The first step is to load and clean all the articles (corpus) defined in the pathCorpus directory (line 22). This task is performed by the `DocumentsSource` class, as described in the *Data extraction* section under *Scala programming* in the *Appendix A, Basic Concepts*:

```
val pathCorpus = "resources/text/chap5/"      //22
val dateFormat = new SimpleDateFormat("MM.dd.yyyy")
val pfnDocs = DocumentsSource(dateFormat, pathCorpus) |> //23

val textAnalyzer = TextAnalyzer[Long](parse, LEXICON)
val pfnText = textAnalyzer |> //24

for {
    corpus <- pfnDocs(None)    //25
    termsFreq <- pfnText(corpus) //26
    featuresSet <- textAnalyzer.quantize(termsFreq) //27
    expected <- Try(difference(TSLA_QUOTES, diffInt)) //28
    nb <- NaiveBayes[Double](1.0,
                            featuresSet._2.zip(expected)) //29
} yield {
    show(s"Naive Bayes model${nb.toString(quantized._1)}")
    ...
}
```

A document source is fully defined by the path of the data input files and the format used in the timestamp (line 23). The text analyzer and its explicit `pfnText` data transformation is instantiated (line 24). The text processing pipeline is defined by the following steps:

1. The transformation of an input source file into a corpus (a sequence of news articles) using the `pfnDoc` partial function (line 25).
2. The transformation of a corpus into a sequence of a `termsFreq` relative keyword frequency vector using the `pfnText` partial function (line 26).
3. The transformation of a sequence of relative keywords frequency vector into a `featuresSet` using `quantize` (line 27) (refer to the *The differential operator* section under *Time series in Scala* in *Chapter 3, Data Preprocessing*).
4. The creation of the binomial `NaiveBayes` model using the pair (`featuresSet._2` and `expected`) as training data (line 29).

The expected class values (0,1) are extracted from the daily stock price for Tesla Motors, `TSLA_QUOTES`:

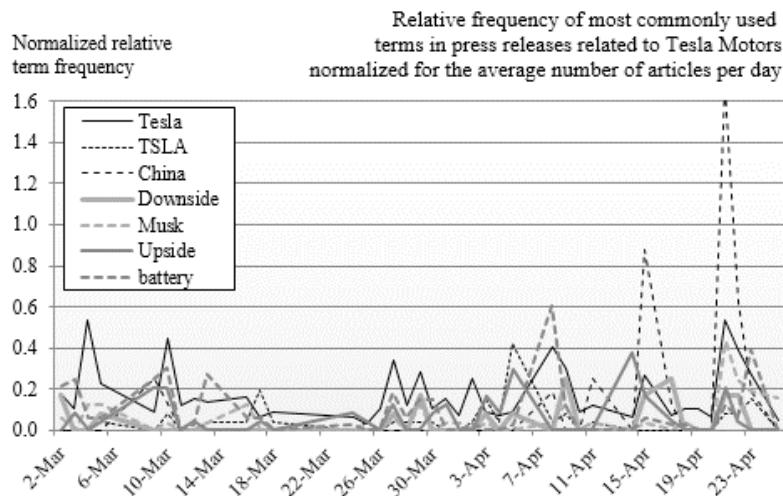
```
val TSLA_QUOTES = Array[Double] (250.56, 254.84, ... )
```

The semantic analysis

This example uses a very primitive semantic map (lexicon) for the sake of illustrating the benefits and inner workings of the multinomial Naïve Bayes algorithm. Commercial applications involving sentiment analysis or topic analysis require a deeper understanding of semantic associations and extraction of topics using advanced generative models, such as the Latent Dirichlet allocation.

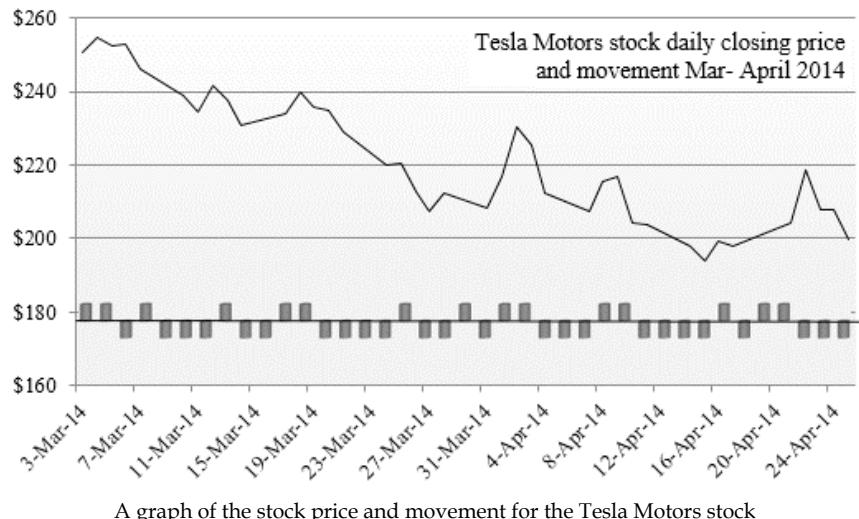
Evaluating the text mining classifier

The following chart describes the frequency of occurrences of some of the keywords related to either Tesla Motors or its stock ticker TSLA:



A graph of the relative frequency of a partial list of stock-related terms

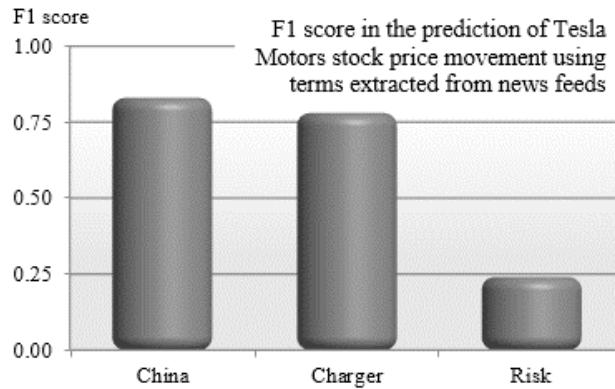
The following chart plots the expected change in the direction of the stock price for the trading day following the press release(s) or news article(s):



A graph of the stock price and movement for the Tesla Motors stock

The preceding chart displays the historical price of the stock TSLA with the direction (UP and DOWN). The classification of 15 percent of the labeled data selected for the validation of the classifier has an F_1 score of 0.71. You need to keep in mind that no preprocessing or clustering was performed to isolate the most relevant features/keywords. We initially selected the keywords according to the frequency of their occurrences in the financial news.

It is fair to assume that some of the keywords have a more significant impact on the direction of the stock price than others. One simple but interesting exercise is to record the value of the F_1 score for a validation for which only the observations that have a high number of occurrences of a specific keyword are used, as shown in the following graph:



A bar chart representing predominant keywords in predicting the TSLA stock movement

The preceding bar chart shows that the terms **China**, representing all the mentions of the activities of Tesla Motors in China, and **Charger**, which covers all the references to the charging stations, have a significant positive impact on the direction of the stock with a probability averaging to 75 percent. The terms under the **Risk** category have a negative impact on the direction of the stock with a probability of 68 percent, or a positive impact of the direction of the stock with a probability of 32 percent. Within the remaining eight categories, 72 percent of them were unusable as a predictor of the direction of the stock price.

This approach can be used for selecting features as an alternative to mutual information for using classifiers that are more elaborate. However, it should not be regarded as the primary methodology for selecting features, but instead as a by-product of the Naïve Bayes formula applied to models with a very small number of relevant features. Techniques such as the principal components analysis, as described in the *Principal components analysis* section under *Dimension reduction* in *Chapter 4, Unsupervised Learning*, are available to reduce the dimension of the problem and make Naïve Bayes a viable classifier.

Pros and cons

The examples selected in this chapter do not do justice to the versatility and accuracy of the Naïve Bayes family of classifiers.

The Naïve Bayes algorithm is a simple and robust generative classifier that relies on prior conditional probabilities to extract a model from a training dataset. The Naïve Bayes model has its benefits, as mentioned here:

- It is easy to implement and parallelize

- It has a very low computational complexity: $O((n+c)*m)$, where m is the number of features, C is the number of classes, and n is the number of observations
- It handles missing data
- It supports incremental updates, insertions, and deletions

However, Naïve Bayes is not a silver bullet. It has the following disadvantages:

- It requires a large training set to achieve reasonable accuracy
- The assumption of the independence of features is not practical in the real world
- It requires dealing with the zero-frequency problem for counters

Summary

There is a reason why the Naïve Bayes model is one of the first supervised learning techniques taught in a machine learning course: it is simple and robust. As a matter of fact, this is the first technique that should come to mind when you are considering creating a model from a labeled dataset, as long as the features are conditionally independent.

This chapter also introduced you to the basics of text mining as an application of Naïve Bayes.

Despite all its benefits, the Naïve Bayes classifier assumes that the features are conditionally independent, a limitation that cannot be always overcome. In the case of the classification of documents or news releases, Naïve Bayes incorrectly assumes that terms are semantically independent: the two entities' age and date of birth are highly correlated. The discriminative classifiers described in the next few chapters address some of Naïve Bayes' limitations [5:14].

This chapter does not treat temporal dependencies, sequence of events, or conditional dependencies between observed and hidden features. These types of dependencies necessitate a different approach to modeling that is the subject of *Chapter 7, Sequential Data Models*.

6

Regression and Regularization

In the first chapter, we briefly introduced the binary logistic regression (the binomial logistic regression for a single variable) as our first test case. The purpose was to illustrate the concept of discriminative classification. There are many more regression models, starting with the ubiquitous ordinary least square linear regression and the logistic regression [6:1].

The purpose of regression is to minimize a loss function, with the **residual sum of squares (RSS)** being one that is commonly used. The problem of overfitting described in the *Overfitting* section under *Assessing a model* in *Chapter 2, Hello World!*, can be addressed by adding a **penalty term** to the loss function. The penalty term is an element of the larger concept of **regularization**.

The first section of this chapter will describe and implement the linear **least-squares regression**. The second section will introduce the concept of regularization with an implementation of the **ridge regression**. Finally, the logistic regression will be revisited in detail from the perspective of a classification model.

Linear regression

Linear regression is by far the most widely used, or at least the most commonly known, regression method. The terminology is usually associated with the concept of fitting a model to data and minimizing the errors between the expected and predicted values by computing the sum of square errors, residual sum of square errors, or least-square errors.

The least squares problems fall into the following two categories:

- Ordinary least squares
- Nonlinear least squares

One-variate linear regression

Let's start with the simplest form of linear regression, which is the single variable regression, in order to introduce the terms and concepts behind linear regression. In its simplest interpretation, the one-variate linear regression consists of fitting a line to a set of data points $\{x, y\}$.

M1: A single variable linear regression for a model f with weights w_j for features x_j and labels (or expected values) y_j is given by the following formula:


$$\tilde{w} = \arg \min_{w,r} \left\{ \sum_{j=0}^{N-1} (y_j - f(x_j|w))^2 \right\} \quad f(x|w) = w_0 + w_1 x$$

Here, w_1 is the slope, w_0 is the intercept, f is the linear function that minimizes the RSS, and (x_j, y_j) is a set of n observations.

The RSS is also known as the **sum of squared errors (SSE)**. The **mean squared error (MSE)** for n observations is defined as the ratio RSS/n .

Terminology



The terminology used in the scientific literature regarding regression is a bit confusing at times. Regression weights are also known as regression coefficients or regression parameters. The weights are referred to as w in formulas and the source code throughout the chapter, although β is also used in reference books.

Implementation

Let's create a `SingleLinearRegression` parameterized class to implement the **M1** formula. The linear regression is a data transformation that uses a model implicitly derived or built from data. Therefore, the simple linear regression implements the `ITransform` trait, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*

The `SingleLinearRegression` class takes the following two arguments:

- An `xt` vector of single variable observations
- A vector of expected values or labels (line 1)

The code will be as follows:

```
class SingleLinearRegression[T <: AnyVal] (
  xt: XSeries[T],
  expected: Vector[T])(implicit f: T => Double)
  extends ITransform[T](xt) with Monitor[Double] { //1
    type V = Double //2

    val model: Option[DblPair] = train //3
    def train: Option[DblPair]
    override def |> : PartialFunction[T, Try[V]]
    def slope: Option[Double] = model.map(_._1)
    def intercept: Option[Double] = model.map(_._2)
}
```

The `Monitor` trait is used to collect the profiling information during training (refer to the `Monitor` section under *Utility classes* in the *Appendix A, Basic Concepts*).

The class has to define the type of the output of the `|>` prediction method, which is a `Double` (line 2).

Model instantiation

The model parameters are computed through training and the model is instantiated regardless of whether the model is actually validated. A commercial application requires the model to be validated using a methodology such as the K-fold validation, as described in the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*.

The training generates the model defined as the regression weights (slope and intercept) (line 3). The model is set as `None` if an exception is thrown during training:

```
def train: Option[DblPair] = {
  val regr = new SimpleRegression(true) //4
  regr.addData(zipToSeries(xt, expected).toArray) //5
  Some((regr.getSlope, regr.getIntercept)) //6
}
```

The regression weights or coefficients, that is the `model` tuple, are computed using the `SimpleRegression` class from the `stats.regression` package of the Apache Commons Math library with the `true` argument to trigger the computation of the intercept (line 4). The input time series and the labels (or expected values) are zipped to generate an array of two values (input and expected) (line 5). The `model` is initialized with the slope and intercept computed during the training (line 6).

The `zipToSeries` method of the `XTSerie`s object is described in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*.

private versus private[this]

A `private` value or variable can be accessed only by all the instances of a class. A value declared `private [this]` can be manipulated only by the `this` instance. For example, the value `model` can be accessed only by the `this` instance of `SingleLinearRegression`.

Test case

For our first test case, we compute the single variate linear regression of the price of the Copper ETF (ticker symbol: CU) over a period of 6 months (January 1, 2013 to June 30, 2013):

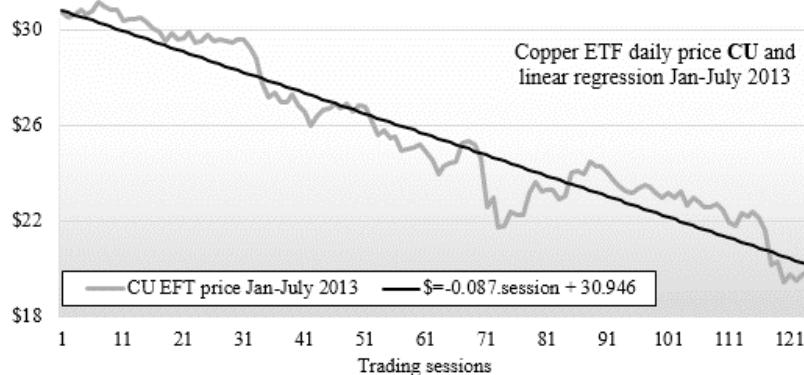
```
val path = "resources/data/chap6/CU.csv"
for {
    price <- DataSource(path, false, true, 1) get adjClose //7
    days <- Try(Vector.tabulate(price.size) (_.toDouble)) //8
    linRegr <- SingleLinearRegression[Double] (days, price) //9
} yield {
    if( linRegr.isModel ) {
        val slope = linRegr.slope.get
        val intercept = linRegr.intercept.get
        val error = mse(days, price, slope, intercept)//10
    }
    ...
}
```

The daily closing price of the ETF CU is extracted from a CSV file (line 7) as the expected values using a `DataSource` instance, as described in the *Data extraction and Data sources* section in the *Appendix A, Basic Concepts*. The `x` values `days` are automatically generated as a linear function (line 8). The expected values (`price`) and sessions (`days`) are the inputs to the instantiation of the simple linear regression (line 9).

Once the model is created successfully, the test code computes the `mse` mean squared error of the predicted and expected values (line 10):

```
def mse(
    predicted: DblVector,
    expected: DblVector,
    slope: Double,
    intercept: Double): Double = {
    val predicted = xt.map( slope*_ + intercept)
    XTSeries.mse(predicted, expected) //11
}
```

The mean least squared error is computed using the `mse` method of `XTSeries` (line 11). The original stock price and linear regression equation are plotted on the following chart:



The total least square error is 0.926.

Although the single variable linear regression is convenient, it is limited to a scalar time series. Let's consider the case of multiple variables.

Ordinary least squares regression

The **ordinary least squares regression** computes the parameters w of a linear function $y = f(x_0, x_1, \dots, x_d)$ by minimizing the residual sum of squares. The optimization problem is solved by performing vector and matrix operations (transposition, inversion, and substitution).

M2: The minimization of the loss function is given by the following formula:



$$\tilde{w} = \arg \min_{w,r} \left\{ \sum_{j=0}^{N-1} (y_j - f(x_j|w))^2 \right\} \quad f(x|w) = w_0 + \sum_1^{D-1} w_d x_d$$

Here, w is the weights or parameters of the regression, $(x_i, y_i)_{i:0, n-1}$ is the n observations of a vector x and an expected output value y , and f is the linear multivariate function, $y = f(x_0, x_1, \dots, x_d)$.

There are several methodologies to minimize the residual sum of squares for a linear regression:

- Resolution of the set of n equations with d variables (weights) using the **QR decomposition** of the n by d matrix, representing the time series of n observations of a vector of d dimensions with $n \geq d$ [6:2]
- **Singular value decomposition** on the observations-features matrix, in the case where the dimension d exceeds the number of observations n [6:3]
- **Gradient descent** [6:4]
- **Stochastic gradient descent** [6:5]

An overview of these matrix decompositions and optimization techniques can be found in the *Linear algebra* and *Summary of optimization techniques* sections in the *Appendix A, Basic Concepts*.

The QR decomposition generates the smallest relative error MSE for the most common least squares problem. The technique is used in our implementation of the least squares regression.

Design

The implementation of the least squares regression leverages the Apache Commons Math library implementation of the ordinary least squares regression [6:6].

This chapter describes several types of regression algorithms. It makes sense to define a generic `Regression` trait that defines the key element component of a regression algorithm.

- A model of the `RegressionModel` type (line 1)
- Two methods to access the components of the regression model: `weights` and `rss` (line 2 and 3)

- A `train` polymorphic method that implements the training of this specific regression algorithm (line 4)
- A `training` protected method that wraps `train` into a `Try` monad

The code will be as follows:

```
trait Regression {
  val model: Option[RegressionModel] = training //1

  def weights: Option[DblArray] = model.map(_.weights)//2
  def rss: Option[Double] = model.map(_.rss) //3
  def isModel: Boolean = model != None

  protected def train: RegressionModel //4
  def training: Option[RegressionModel] = Try(train) match {
    case Success(_model) => Some(_model)
    case Failure(e) => e match {
      case err: MatchError => { ... }           case _ => { ... }
    }
  }
}
```

The model is simply defined by its `weights` and its residual sum of squares (line 5):

```
case class RegressionModel( //5
  val weights: DblArray, val rss: Double) extends Model

object RegressionModel {
  def dot[T <: AnyVal](x: Array[T],
    w: DblArray)(implicit f: T => Double): Double =
    x.zip(weights.drop(1))
      .map{ case(x, w) => x*w}.sum + weights.head //6
}
```

The `RegressionModel` companion object implements the computation of the dot inner product of the `weights` regression and an observation, `x` (line 6). The `dot` method is used throughout the chapter.

Let's create a `MultiLinearRegression` class as a data transformation whose model is implicitly derived from the input data (training set), as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*:

```
class MultiLinearRegression[T <: AnyVal] (
  xt: XSeries[T],
  expected: DblVector)(implicit f: T => Double)
  extends ITransform[Array[T]](xt) with Regression
```

```

        with Monitor[Double] { //7
    type V = Double //8

    override def train: Option[RegressionModel] //9
    override def |> : PartialFunction[Array[T], Try[V]] //10
}

```

The `MultiLinearRegression` class takes two arguments: the multidimensional time series of the `xt` observations and the vector of expected values (line 7). The class implements the `ITransform` trait and needs to define the type of the output value for the prediction or regression, `V` as a `Double` (line 8). The constructor for `MultiLinearRegression` creates the `model` through training (line 9). The `ITransform` trait's `|>` method implements the runtime prediction for the multilinear regression (line 10).

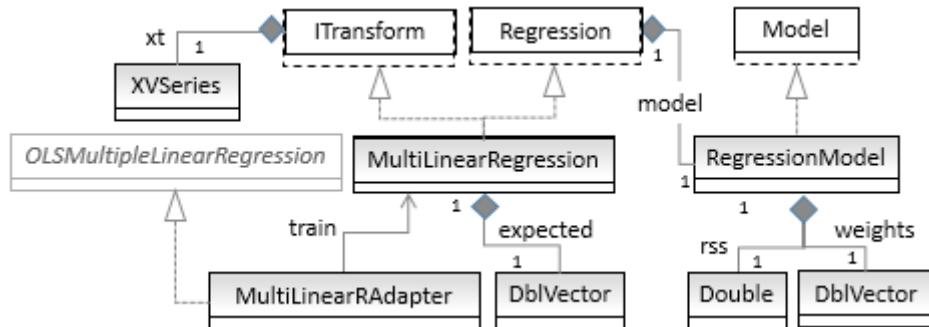
The `Monitor` trait is used to collect the profiling information during training (refer to the *Monitor* section under *Utility classes* in the *Appendix A, Basic Concepts*).

[

The regression model
 The RSS is included in the model because it provides the client code with the important information regarding the accuracy of the underlying technique used to minimize the loss function.

]

The relationship between the different components of the multilinear regression is described in the following UML class diagram:



The UML class diagram for the multilinear (OLS) regression

The UML diagram omits the helper traits and classes such as `Monitor` or the Apache Commons Math components.

Implementation

The training is performed during the instantiation of the `MultiLinearRegression` class (refer to the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*):

```
def train: RegressionModel = {
    val olsMlr = new MultiLinearRAdapter //11
    olsMlr.createModel(expected, data) //12
    RegressionModel(olsMlr.weights, olsMlr.rss) //13
}
```

The functionality of the ordinary least squares regression in the Apache Commons Math library is accessed through an `olsMlr` reference to the `MultiLinearRAdapter` adapter class (line 11).

The `train` method creates the model by invoking the `OLSMultipleLinearRegression` Apache Commons Math class (line 12) and returns the regression model (line 13). The various methods of the class are accessed through the `MultiLinearRAdapter` adapter class:

```
class MultiLinearRAdapter extends OLSMultipleLinearRegression {
    def createModel(y: DblVector, x: Vector[DblArray]): Unit =
        super.newSampleData(y.toArray, x.toArray)

    def weights: DblArray = estimateRegressionParameters
    def rss: Double = calculateResidualSumOfSquares
}
```

The `createModel`, `weights`, and `rss` methods route the request to the corresponding methods in `OLSMultipleLinearRegression`.

The `Try{ }` Scala exception handling monad is used as the return type for the `train` method in order to catch the different types of exceptions thrown by the Apache Commons Math library such as `MathIllegalArgumentException`, `MathRuntimeException`, or `OutOfRangeException`.

Exception handling

Wrapping up invocation of methods in a third party with a `Try { }` Scala exception handler matters for a couple of reasons:

- It makes debugging easier by segregating your code from the third party
- It allows your code to recover from the exception by reexecuting the same function with an alternative third-party library method, whenever possible

The predictive algorithm for the ordinary least squares regression is implemented by the `| >` data transformation. The method predicts the output value, given a model and an input value, x :

```
def |> : PartialFunction[Array[T], Try[V]] = {  
    case x: Array[T] if isModel &&  
        x.length == model.get.size-1  
    => Try( dot(x, model.get) ) //14  
}
```

The predictive value is computed using the `dot` method defined in the `RegressionModel` singleton, which was introduced earlier in this section (line 14).

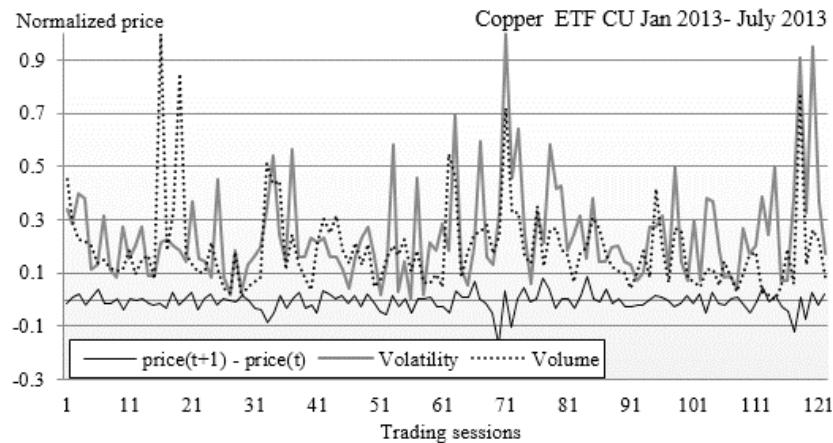
Test case 1 – trending

Trending consists of extracting the long-term movement in a time series. Trend lines are detected using a multivariate least squares regression. The objective of this first test is to evaluate the filtering capability of the ordinary least squares regression.

The regression is performed on the relative price variation of the Copper ETF (ticker symbol: CU). The selected features are `volatility` and `volume`, and the label or target variable is the price change between two consecutive y trading sessions.

The naming convention for the trading data and metrics is described in the *Trading data* section under *Technical analysis* in the *Appendix A, Basic Concepts*.

The volume, volatility, and price variation for CU between January 1, 2013 and June 30, 2013 are plotted on the following chart:



The chart for price variation, volatility, and trading volume for Copper ETF

Let's write the client code to compute the multivariate linear regression, $price\ change = w_0 + volatility.w_1 + volume.w_2$:

```

import YahooFinancials._
val path = "resources/data/chap6/CU.csv" //15
val src = DataSource(path, true, true, 1) //16

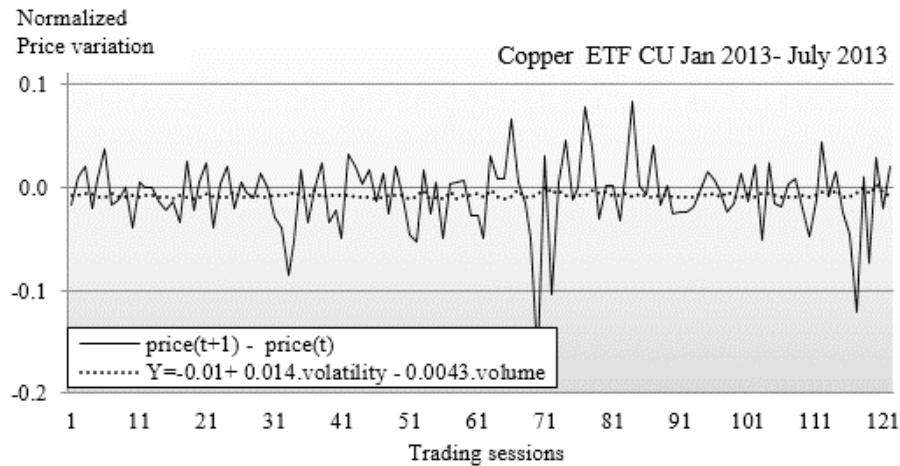
for {
    price <- src.get(adjClose) //17
    volatility <- src.get(volatility) //18
    volume <- src.get(volume) //19
    (features, expected) <- differentialData(volatility,
                                              volume, price, diffDouble) //20
    regression <- MultiLinearRegression[Double] (
        features, expected) //21
} yield {
    if( regression.isModel ) {
        val trend = features.map(dot(_,regression.weights.get))
        display(expected, trend) //22
    }
}
}

```

Let's take a look at the steps required for the execution of the test: it consists of collecting data, extracting the features and expected values, and training the multilinear regression model:

1. Locate the CSV formatted data source file (line 15).
2. Create a data source extractor, `DataSource`, for the trading session closing price, the volatility session, and the volume session for the ETF CU (line 16).
3. Extract the price of the ETF (line 17), its volatility within a trading session (line 18), and the trading volume during the session (line 19) using the `DataSource` transform.
4. Generate the labeled data as a pair of features (relative volatility and relative volume for the ETF) and expected outcome (0, 1) for training the model for which 1 represents the increase in the price and 0 represents the decrease in the price (line 20). The `differentialData` generic method of the `XTSerie`s singleton is described in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*.
5. The multilinear regression is instantiated using the `features` set and the `expected` change in the daily ETF price (line 21).
6. Display the expected and trending values using JFreeChart (line 22).

The time series of expected values and the data predicted by the regression are plotted on the following chart:



The price variation and the least squares regression for the Copper ETF according to volatility and volume

The least squares regression model is defined by the linear function for the estimation of price variation as follows:

$$\text{price}(t+1)-\text{price}(t) = -0.01 + 0.014 \text{ volatility} - 0.0042 \cdot \text{volume}$$

The estimated price change (the dotted line in the preceding chart) represents the long-term trend from which the noise is filtered out. In other words, the least squares regression operates as a simple low-pass filter as an alternative to some of the filtering techniques such as the discrete Fourier transform or the Kalman filter, as described in *Chapter 3, Data Preprocessing* [6:7].

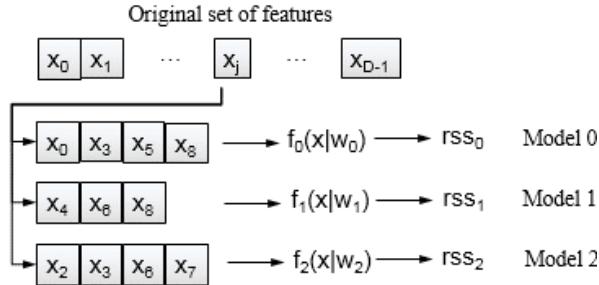
Although trend detection is an interesting application of the least squares regression, the method has limited filtering capabilities for time series [6:8]:

- It is sensitive to outliers
- The first and last few observations need to be discarded
- As a deterministic method, it does not support noise analysis (distribution, frequencies, and so on)

Test case 2 – feature selection

The second test case is related to feature selection. The objective is to discover which subset of initial features generates the most accurate regression model, that is, the model with the smallest residual sum of squares on the training set.

Let's consider an initial set of D features $\{x_i\}$. The objective is to estimate the subset of features $\{x_{id}\}$ that are most relevant to the set of observations using a least squares regression. Each subset of features is associated with a $f_j(x|w_j)$ model:



A diagram for the features set selection

The ordinary least square regression is used to select the model parameters w in the case the feature set is small. Performing the regression of each subset of a large original feature set is not practical.

M3: The features selection can be expressed mathematically as follows:

$$\hat{f}_j = \arg \min_{f_j} \left\{ \sum_{i=0}^{n-1} (y_i - f_j(x|w))^2 \right\} \quad f_j(x|w) = w_{j0} + \sum_{d=1}^{D_j-1} w_{jd} x_d$$

Here, w_{jk} is the weights of the regression for the function/model $f_j(x, y)$, $y_{i,0,n-1}$ is the n observations of a vector x and expected output value y , and f is the linear multivariate function, $y = f(x_0, x_1, \dots, x_d)$.

Let's consider the following four financial time series over the period from January 1, 2009 to December 31, 2013:

- The exchange rate of Chinese Yuan to US Dollar
- The S&P 500 index
- The spot price of gold
- The 10-year Treasury bond price

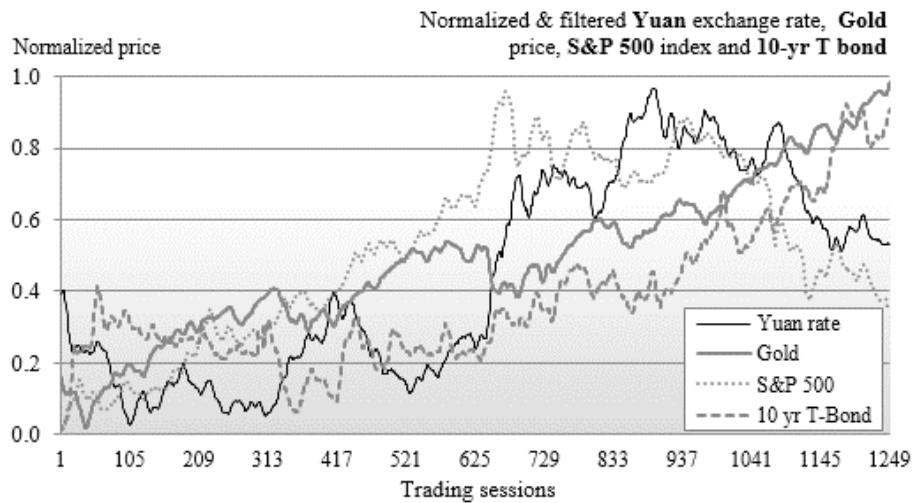
The problem is to estimate which combination of the S&P 500 index, gold price, and 10-year Treasury bond price variables is the most correlated to the exchange rate of the Yuan. For practical reasons, we use the Exchange Trade Funds CYN as the proxy for the Yuan/US dollar exchange rate (similarly, SPY, GLD, and TLT for the S&P 500 index, spot price of gold, and 10-year Treasury bond price, respectively).

Automation of features extraction



The code in this section implements an ad hoc extraction of features with an arbitrary fixed set of models. The process can be easily automated with an optimizer (the gradient descent, genetic algorithm, and so on) using $1/RSS$ as the objective function.

The number of models to evaluate is relatively small, so an ad hoc approach to compute the RSS for each combination is acceptable. Let's take a look at the following graph:



The graph of the Chinese Yuan exchange rate, gold price, 10-year Treasury bond price, and S&P 500 index

The `getRSS` method implements the computation of the RSS value given a set of `xt` observations, `y` expected (smoothed) values, and `featureLabels` labels for features and then returns a textual result:

```
def getRSS(
    xt: Vector[DblArray],
    expected: DblVector,
    featureLabels: Array[String]): String = {
  val regression =
    new MultiLinearRAdapter[Double](xt, expected) //23
  val modelStr = regression.weights.get.view
```

```

    .zipWithIndex.map{ case( w, n) => {
      val weights_str = format(w, emptyString, SHORT)
      if(n == 0 ) s"${featureLabels(n)} = $weights_str"
      else s"$weights_str.${featureLabels(n)}"
    }}.mkString(" + ")
    s"model: $modelStr\nRSS =${regression.get.rss}" //24
  }
}

```

The `getRss` method merely trains the model by instantiating the multilinear regression class (line 23). Once the regression model is trained during the instantiation of the `MultiLinearRegression` class, the coefficients of the regression weights and the RSS values are stringized (line 24). The `getRss` method is invoked for any combination of the `ETF`, `GLD`, `SPY`, and `TLT` variables against the `CNY` label.

Let's take a look at the following test code:

```

val SMOOTHING_PERIOD: Int = 16 //25
val path = "resources/data/chap6/"
val symbols = Array[String] ("CNY", "GLD", "SPY", "TLT") //26
val movAvg = SimpleMovingAverage[Double] (SMOOTHING_PERIOD) //27

for {
  pfnMovAve <- Try(movAvg |>) //28
  smoothed <- filter(pfnMovAve) //29
  models <- createModels(smoothed) //30
  rsses <- Try(getModelsRss(models, smoothed)) //31
  (mses, tss) <- totalSquaresError(models,smoothed.head) //32
} yield {
  s"""$rsses.mkString("\n")\n$mses.mkString("\n")"""
  | \nResidual error= $tss".stripMargin
}

```

The dataset is large (1,260 trading sessions) and noisy enough to warrant filtering using a simple moving average with a period of 16 trading sessions (line 25). The purpose of the test is to evaluate the possible correlation between the four ETFs: `CNY`, `GLD`, `SPY`, and `TLT` (line 26). The execution test instantiates the simple moving average (line 27), as described in the *The simple moving average* section in *Chapter 3, Data Preprocessing*.

The workflow executes the following steps:

1. Instantiate a simple moving average `pfnMovAve` partial function (line 28).
2. Generate smoothed historical prices for the CNY, GLD, SPY, and TLT ETFs using the `filter` function (line 29):

```
Type PFNMOVAVE = PartialFunction[DblVector, Try[DblVector]]
```

```
def filter(pfnMovAve: PFNMOVEAVE): Try[Array[DblVector]] = Try {  
    symbols.map(s => DataSource(s"$path$s.csv", true, true, 1))  
        .map(_.get(adjClose))  
        .map(pfnMovAve(_)).map(_.get)
```

3. Generate the list of features for each model using the `createModels` method (line 30):

```
type Models = List[(Array[String], DblMatrix)]
```

```
def createModels(smoothed: Array[DblVector]): Try[Models] =  
Try {  
    val features = smoothed.drop(1).map(_.toArray) //33  
    List[(Array[String], DblMatrix)]() //34  
        (Array[String]("CNY", "SPY", "GLD", "TLT"), features.transpose),  
        (Array[String]("CNY", "GLD", "TLT"), features.drop(1).transpose),  
        (Array[String]("CNY", "SPY", "GLD"), features.take(2).transpose),  
        (Array[String]("CNY", "SPY", "TLT"), features.zipWithIndex  
            .filter(_.2 != 1).map(_.1).transpose),  
        (Array[String]("CNY", "GLD"), features.slice(1, 2).transpose)  
    )  
}
```

The smoothed values for CNY are used as the expected values. Therefore, they are removed from the features list (line 33). The five models are evaluated by adding or removing elements from the features list (line 34).

4. Next, the workflow computes the residual sum of squares for all the models using `getModelsRss` (line 31). The method invokes `getRss`, which was introduced earlier in this section, for each model (line 35):

```
def getModelsRss(  
    models: Models,  
    y: Array[DblVector]): List[String] =  
models.map{ case (labels, m) =>  
    s"${getRss(m.toVector, y.head, labels)}" } //35
```

5. Finally, the last step of the workflow consists of computing the `mses` mean squared errors for each model and the total squared errors (line 33):

```
def totalSquaresError(
    models: Models,
    expected: DblVector): Try[(List[String], Double)] = Try {
    val errors = models.map{case (labels,m) =>
        rssSum(m, expected)._1}//36
    val mses = models.zip(errors)
        .map{case(f,e) => s"MSE: ${f._1.mkString(" ")} = $e"}
    (mses, Math.sqrt(errors.sum)/models.size) //37
}
```

The `totalSquaresError` method computes the error for each model by summing the RSS value, `rssSum`, for each model (line 36). The method returns a pair of an array of the mean squared error for each model and the total squared error (line 37).

The RSS does not always provide an accurate visualization of the fitness of the regression model. The fitness of the regression model is commonly assessed using the **r^2 statistics**. The r^2 value is a number that indicates how well data fits into a statistical model.

[



M4: The RSS and r^2 statistics are defined by the following formulae:

$$r^2 = 1 - \frac{RSS}{TSS} \quad TSS = \sum_{i=0}^{n-1} (y_i - \bar{f}(x|w))^2 \quad \bar{f} = \sum_f f_j$$

]

The implementation of the computation of the r^2 statistics is simple. For each model f_j , the `rssSum` method computes the tuple (rss and least squares errors), as defined in the **M4** formula:

```
def rssSum(xt: DblMatrix, expected: DblVector): DblPair = {
    val regression =
        MultiLinearRegression[Double](xt, expected) //38
    val pfnRegr = regression |> //39
    val results = sse(expected.toArray, xt.map(pfnRegr(_).get))
        (regression.rss, results) //40
}
```

Regression and Regularization

The `rssSum` method instantiates the `MultiLinearRegression` class (line 38), retrieves the RSS value, and then validates the `pfnRegr` regressive model (line 39) against the expected values (line 40). The output of the test is presented in the following screenshot:

```
CNY = f(SPY, GLD, TLT)
0.16089780923264457 + -0.21189413823325406.x1 + 0.26299169969099795.x2 + 0.3556562652009136.x3
RSS: 3.681353535940423

CNY = f(SPY, TLT)
0.2039015515038045 + -0.03796334296279046.x1 + 0.26219728078589966.x2
RSS: 3.8589613138639227

CNY = f(GLD, TLT)
0.19290917330324198 + 0.015507174710195552.x1 + 0.2204800144601237.x2
RSS: 3.8849688539396317

CNY = f(SPY, GLD)
0.22242202699107552 + 0.17842973203100937.x1 + -0.12099602178260839.x2
RSS: 4.6681933948464645

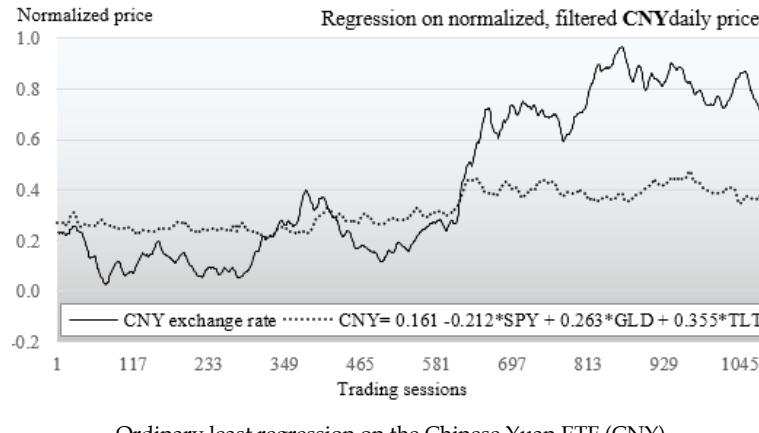
CNY = f(SPY)
0.20238901847764892 + 0.1251591898720694.x1
RSS: 4.7291908591838405

CNY = f(TLT)
0.19724352413716711 + 0.22501420632545652.x1
RSS: 3.8876824376753705

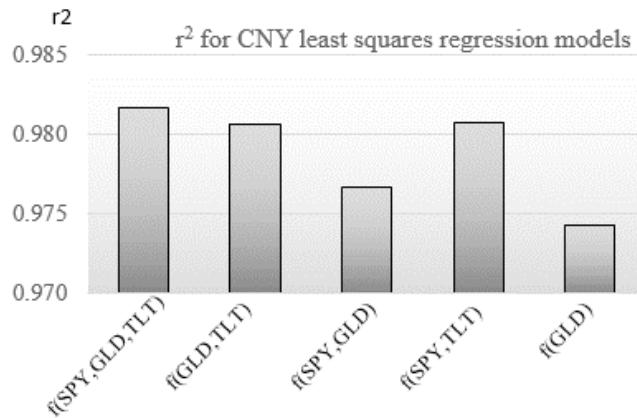
CNY = f(GLD)
0.198195293931846 + 0.16413676262473123.x1
RSS: 5.149661975952835
```

The output results clearly show that the three variable regression, $CNY = f(SPY, GLD, TLT)$, is the most accurate or fittest model for the CNY time series, followed by $CNY = f(SPY, TLT)$. Therefore, the feature selection process generates the features set, $\{SPY, GLD, TLT\}$.

Let's plot the model against the raw data:



The regression model smoothed the original CNY time series. It weeded out all but the most significant price variation. The graph plotting the r^2 value for each of the model confirms that the three features model $CNY=f(SPY, GLD, TLT)$ is the most accurate:



The general linear regression

The concept of a linear regression is not restricted to polynomial fitting models such as $y = w_0 + w_1x + w_2x^2 + \dots + w_nx^n$. Regression models can be also defined as a linear combination of basis functions such as ϕ_j : $y = w_0 + w_1\phi_1(x) + w_2\phi_2(x) + \dots + w_n\phi_n(x)$ [6:9].

Regularization

The ordinary least squares method for finding the regression parameters is a specific case of the maximum likelihood. Therefore, regression models are subject to the same challenge in terms of overfitting as any other discriminative models. You are already aware of the fact that regularization is used to reduce model complexity and avoid overfitting, as stated in the *Overfitting* section in *Chapter 2, Hello World!*

L_n roughness penalty

Regularization consists of adding a $J(w)$ penalty function to the loss function (or RSS in the case of a regressive classifier) in order to prevent the model parameters (also known as weights) from reaching high values. A model that fits a training set very well tends to have many features variables with relatively large weights. This process is known as **shrinkage**. Practically, shrinkage involves adding a function with model parameters as an argument to the loss function (**M5**):

$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w}_d} \left\{ \sum_{i=0}^{n-1} (y_i - f(\mathbf{x}_i | \mathbf{w}))^2 + \lambda J(\mathbf{w}) \right\}$$

The penalty function is completely independent of the training set $\{x, y\}$. The penalty term is usually expressed as a power to the function of the norm of the model parameters (or weights) w_d . For a model of D dimension, the generic L_p-norm is defined as follows (**M6**):

$$J_{pq}(\mathbf{w}) = \|\mathbf{w}\|_p^q = \left[\sum_{d=1}^{D-1} |w_d|^p \right]^{q/p}$$

Notation

Regularization applies to parameters or weights associated with observations. In order to be consistent with our notation, w_0 being the intercept value, the regularization applies to the parameters $w_1 \dots w_d$.

The two most commonly used penalty functions for regularization are L₁ and L₂.

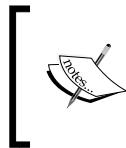
Regularization in machine learning

The regularization technique is not specific to the linear or logistic regression. Any algorithm that minimizes the residual sum of squares, such as a support vector machine or feed-forward neural network, can be regularized by adding a roughness penalty function to the RSS.

The L₁ regularization applied to the linear regression is known as the **lasso regularization**. The **ridge regression** is a linear regression that uses the L₂ regularization penalty.

You may wonder which regularization makes sense for a given training set. In a nutshell, L_2 and L_1 regularizations differ in terms of computation efficiency, estimation, and features selection [6:10] [6:11]:

- **Model estimation:** L_1 generates a sparser estimation of the regression parameters than L_2 . For a large nonsparse dataset, L_2 has a smaller estimation error than L_1 .
- **Feature selection:** L_1 is more effective in reducing the regression weights for features with a higher value than L_2 . Therefore, L_1 is a reliable features selection tool.
- **Overfitting:** Both L_1 and L_2 reduce the impact of overfitting. However, L_1 has a significant advantage in overcoming overfitting (or excessive complexity of a model); for the same reason, L_1 is more appropriate for selecting features.
- **Computation:** L_2 is conducive to a more efficient computation model. The summation of the loss function and L_2 penalty, w^2 , is a continuous and differentiable function for which the first and second derivatives can be computed (**convex minimization**). The L_1 term is the summation of $|w_i|$ and therefore not differentiable.



Terminology

The ridge regression is sometimes called the **penalized least squares regression**. The L_2 regularization is also known as the **weight decay**.

Let's implement the ridge regression, and then evaluate the impact of the L_2 -norm penalty factor.

Ridge regression

The ridge regression is a multivariate linear regression with an L_2 -norm penalty term (M7):

$$\tilde{w}_{\text{ridge}} = \arg \min_w \left\{ \sum_{j=0}^{N-1} (y - w_0 - w^T x)^2 + \lambda |w|_2^2 \right\} \quad |w|_2^2 = \sum_1^{D-1} w_d^2$$

The computation of the ridge regression parameters requires the resolution of a system of linear equations that are similar to the linear regression.

M8: The matrix representation of ridge regression closed form for an input dataset X , a regularization factor λ , and expected values vector y is defined as follows (I is the identity matrix):



$$(X^T X - \lambda \cdot I) \hat{w}_{Ridge} = X^T y$$

M9: The matrices equation is resolved using the QR decomposition as follows:

$$(X^T X - \lambda \cdot I) = Q \begin{bmatrix} R \\ 0 \end{bmatrix} \quad w_{Ridge} = Q^T y \begin{bmatrix} R \\ 0 \end{bmatrix}^{-1}$$

Design

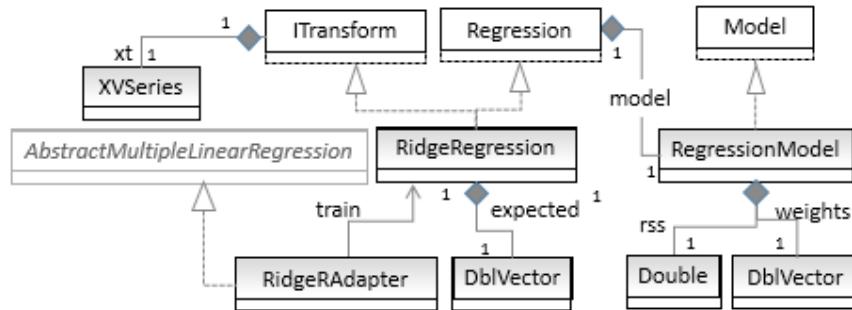
The implementation of the ridge regression adds the L_2 regularization term to the multiple linear regression computation of the Apache Commons Math Library. The methods of `RidgeRegression` have the same signature as their ordinary least squares counterparts except for the lambda L_2 penalty term (line 1):

```
class RidgeRegression[T <: AnyVal] ( //1
    xt: XvSeries[T],
    expected: DblVector,
    lambda: Double) (implicit f: T => Double)
  extends ITransform[Array[T]](xt) with Regression
    with Monitor[Double] { //2

  type V = Double //3
  override def train: Option[RegressionModel] //4
  override def |> : PartialFunction[Array[T], Try[V]]
}
```

The `RidgeRegression` class is implemented as an `ITransform` data transformation whose model is implicitly derived from the input data (training set), as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 2). The `V` type of the output of the `|>` predictive function is a `Double` (line 3). The model is created through training during the instantiation of the class (line 4).

The relationship between the different components of the ridge regression is described in the following UML class diagram:



The UML class diagram for the ridge regression

The UML diagram omits the helper traits or classes such as `Monitor` or the Apache Commons Math components.

Implementation

Let's take a look at the training method, `train`:

```

def train: RegressionModel = {
  val mlr = new RidgeRAdapter(lambda, xt.head.size) //5
  mlr.createModel(data, expected) //6
  RegressionModel(mlr.getWeights, mlr.getRss) //7
}
  
```

It is rather simple; it initialized and executed the regression algorithm implemented in the `RidgeRAdapter` class (line 5), which acts as an adapter to the internal Apache Commons Math library `AbstractMultipleLinearRegression` class in the `org.apache.commons.math3.stat.regression` package (line 6). The method returns a fully initialized regression model that is similar to the ordinary least squared regression (line 7).

Let's take a look at the `RidgeRAdapter` adapter class:

```

class RidgeRAdapter(
  lambda: Double,
  dim: Int) extends AbstractMultipleLinearRegression {
  var qr: QRDecomposition = _ //8
  
```

```
def createModel(x: DblMatrix, y: DblVector): Unit = { //9
    this.newXSampleData(x) //10
    super.newYSampleData(y.toArray)
}
def getWeights: DblArray = calculateBeta.toArray //11
def getRss: Double = rss
}
```

The constructor for the RidgeRAdapter class takes two parameters: the lambda L_2 penalty parameter and the number of features, dim , in an observation. The QR decomposition in the AbstractMultipleLinearRegression base class does not process the penalty term (line 8). Therefore, the creation of the model has to be redefined in the `createModel` method (line 9), which requires to override the `newXSampleData` method (line 10):

```
override protected def newXSampleData(x: DblMatrix): Unit = {
    super.newXSampleData(x) //12
    val r: RealMatrix = getX
    Range(0, dim).foreach(i =>
        r.setEntry(i, i, r.getEntry(i,i) + lambda) ) //13
    qr = new QRDecomposition(r) //14
}
```

The `newXSampleData` method overrides the default observations-features r matrix (line 12) by adding the λ coefficient to its diagonal elements (line 13), and then updating the QR decomposition components (line 14).

The weights for the ridge regression models is computed by implementing the **M6** formula (line 11) in the `calculateBeta` overridden method (line 15):

```
override protected def calculateBeta: RealVector =
    qr.getSolver().solve(getY()) //15
```

The predictive algorithm for the ordinary least squares regression is implemented by the `|>` data transformation. The method predicts the output value, given a model and an input x value (line 16):

```
def |> : PartialFunction[Array[T], Try[V]] = {
    case x: Array[T] if(isModel &&
        x.length == model.get.size-1) =>
        Try( dot(x, model.get) ) //16
}
```

Test case

The objective of the test case is to identify the impact of the L_2 penalization on the RSS value and then compare the predicted values with the original values.

Let's consider the first test case related to the regression on the daily price variation of the Copper ETF (symbol: CU) using the stock daily volatility and volume as features. The implementation of the extraction of observations is identical to that for the least squares regression, as described in the previous section:

```

val LAMBDA: Double = 0.5
val src = DataSource(path, true, true, 1) //17

for {
    price <- src.get(adjClose) //18
    volatility <- src.get(volatility) //19
    volume <- src.get(volume) //20
    (features, expected) <- differentialData(volatility,
                                                volume, price, diffDouble) //21
    regression <- RidgeRegression[Double](features,
                                         expected, LAMBDA) //22
} yield {
    if( regression.isModel ) {
        val trend = features
            .map( dot(_, regression.weights.get) ) //23
    }
}

val y1 = predict(0.2, expected, volatility, volume) //24
val y2 = predict(5.0, expected, volatility, volume)
val output = (2 until 10 by 2).map( n =>
    predict(n*0.1, expected, volatility, volume) )
}
}

```

Let's take a look at the steps required for the execution of the test. The steps consist of collecting data, extracting the features and expected values, and training the ridge regression model:

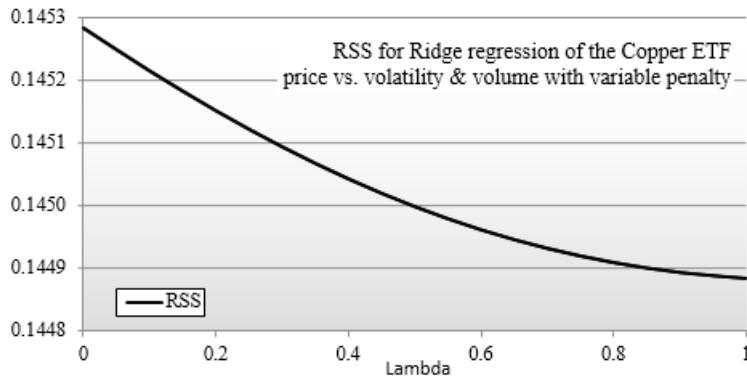
1. Create a data source extractor for the price trading session closing, the volatility session, and the volume session for the ETF CU using the DataSource transformation (line 17).
2. Extract the closing price of the ETF (line 18), its volatility within a trading session (line 19), and the volume trading during the same session (line 20).

3. Generate the labeled data as a pair of features (the relative volatility and relative volume for the ETF) and the expected outcome $\{0, 1\}$ for training the model, where 1 represents the increase in the price and 0 represents the decrease in the price (line 21). The `differentialData` generic method of the `XTSeries` singleton is described in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*.
4. Instantiate the ridge regression using the features set and the expected change in the daily stock price (line 22).
5. Compute the trend values using the `dot` function of the `RegressionModel` singleton (line 23).
6. Execute a using the ridge regression is implemented by the `predict` method (line 24).

The code is as follows:

```
def predict(  
    lambda: Double,  
    deltaPrice: DblVector,  
    volatility: DblVector,  
    volume: DblVector): DblVector = {  
  
    val observations = zipToSeries(volatility, volume)//25  
    val regression = new RidgeRegression[Double](observations,  
        deltaPrice, lambda)  
    val fnRegr = regression |> //26  
    observations.map( fnRegr(_).get) //27  
}
```

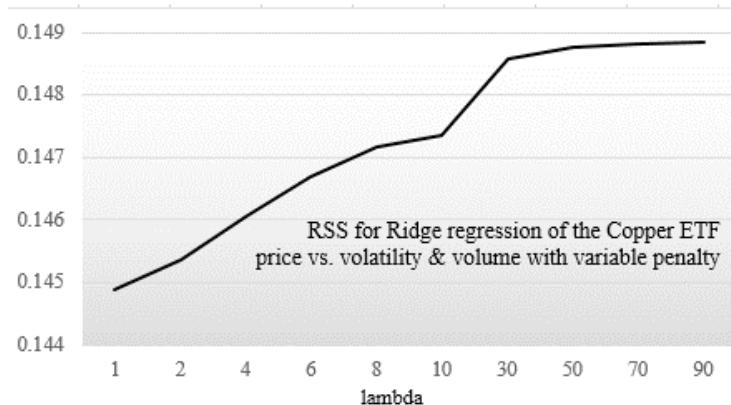
The observations are extracted from the `volatility` and `volume` time series (line 25). The predictive method for the `fnRegr` ridge regression (line 26) is applied to each observation (line 27). The RSS value, `rss`, is plotted for different values of λ , as shown in the following chart:



The graph of RSS versus lambda for the Copper ETF

The residual sum of squares decreases as λ increases. The curve seems to be reaching for a minimum around $\lambda = 1$. The case of $\lambda = 0$ corresponds to the least squares regression.

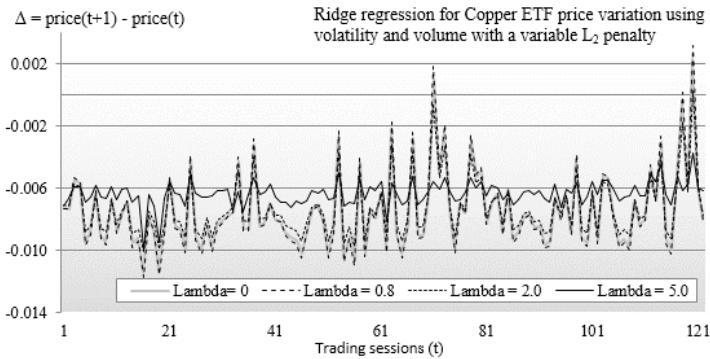
Next, let's plot the RSS value for λ varying between 1 and 100:



The graph of RSS versus a large value Lambda for the Copper ETF

This time around, the value of RSS increases with λ before reaching a maximum for $\lambda > 60$. This behavior is consistent with other findings [6:12]. As λ increases, the overfitting gets more expensive, and therefore, the RSS value increases.

Let's plot the predicted price variation of the Copper ETF using the ridge regression with different values of lambda (λ):



The graph of ridge regression on the Copper ETF price variation with a variable, lambda

The original price variation of the Copper ETF, $\Delta = \text{price}(t + 1) - \text{price}(t)$, is plotted as $\lambda = 0$. Let's analyze the behavior of the predictive model for different values of λ :

- The predicted values for $\lambda = 0.8$ is very similar to the original data.
- The predicted values for $\lambda = 2$ follow the pattern of the original data with a reduction of large variations (peaks and troughs).
- The predicted values for $\lambda = 5$ corresponds to a smoothed dataset. The pattern of the original data is preserved but the magnitude of the price variation is significantly reduced.

The logistic regression, which was briefly introduced in the *Let's kick the tires* section in *Chapter 1, Getting Started*, is the next logical regression model to be discussed. The logistic regression relies on optimization methods. Let's go through a short refresher course in optimization before diving into the logistic regression.

Numerical optimization

This section briefly introduces the different optimization algorithms that can be applied to minimize the loss function, with or without a penalty term. These algorithms are described in more detail in the *Summary of optimization techniques* section in the *Appendix A, Basic Concepts*.

First, let's define the **least squares problem**. The minimization of the loss function consists of nullifying the first order derivatives, which in turn generates a system of D equations (also known as the gradient equations), D being the number of regression weights (parameters). The weights are iteratively computed by solving the system of equations using a numerical optimization algorithm.

M10: The definition of the least squares-based loss function for residual r_i , weights w , a model f , input data x_i , and expected values y_i is as follows:

$$\mathcal{L}(w) = \sum_{i=0}^{n-1} r_i(w)^2 \quad r_i(w) = y_i - f(x_i|w)$$

M10: The generation of gradient equations with a Jacobian J matrix (refer to the *Mathematics* section in the *Appendix A, Basic Concepts*) after minimization of the loss function L is defined as follows:



$$\sum_{i=0}^{n-1} r_i(w) J_{id}(w) = 0 \quad J_{id}(w) = -\frac{\partial r_i(w)}{\partial w_d}$$

M11: The iterative approximation using the Taylor series on the model f for k iterations on the computation of weights w is defined as follows:

$$f(x_i|w) - f(x_i|w^{(k)}) \sim \sum_{jd=0}^{D-1} \frac{\partial f(x_i|w^{(k)})}{\partial w_d} (w - w^{(k)})$$

The logistic regression is a nonlinear function. Therefore, it requires the nonlinear minimization of the sum of least squares. The optimization algorithms for the nonlinear least squares problems can be divided into two categories:

- **Newton** (or 2nd order techniques): These algorithms calculate the second order derivatives (the Hessian matrix) to compute the regression weights that nullify the gradient. The two most common algorithms in this category are the Gauss-Newton and Levenberg-Marquardt methods (refer to the *Nonlinear least squares minimization* section in the *Appendix A, Basic Concepts*). Both algorithms are included in the Apache Commons Math library.
- **Quasi-Newton** (or 1st order techniques): First order algorithms do not compute but estimate the second order derivatives of the least squares residuals from the Jacobian matrix. These methods can minimize any real-valued functions, not just the least squares summation. This category of algorithms includes the Davidon-Fletcher-Powell and the Broyden-Fletcher-Goldfarb-Shannon methods (refer to the *Quasi-Newton algorithms* section in the *Appendix A, Basic Concepts*).

Logistic regression

Despite its name, the *logistic regression* is a classifier. As a matter of fact, the logistic regression is one of the most commonly used discriminative learning techniques because of its simplicity and its ability to leverage a large variety of optimization algorithms. The technique is used to quantify the relationship between an observed target (or expected) variable y and a set of variables x that it depends on. Once the model is created (trained), it is available to classify real-time data.

A logistic regression can be either binomial (two classes) or multinomial (three or more classes). In a binomial classification, the observed outcome is defined as {true, false}, {0, 1}, or {-1, +1}.

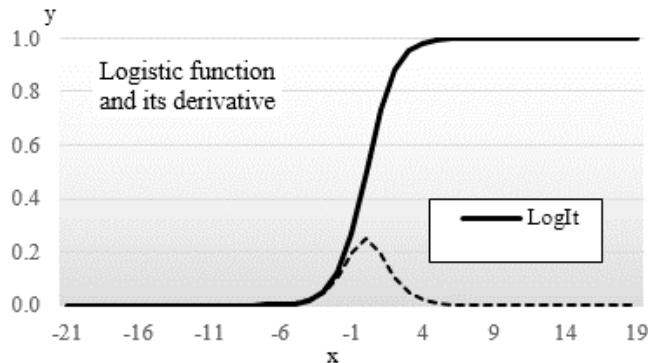
Logistic function

The conditional probability in a linear regression model is a linear function of its weights [6:13]. The logistic regression model addresses the nonlinear regression problem by defining the logarithm of the conditional probability as a linear function of its parameters.

First, let's introduce the logistic function and its derivative, which are defined as follows (M12):

$$f(x) = \frac{1}{(1 - e^{-x})} \quad \frac{df}{dx} = f(x)(1 - f(x))$$

The logistic function and its derivative are illustrated in the following graph:



The graph of the logistic function and its derivative

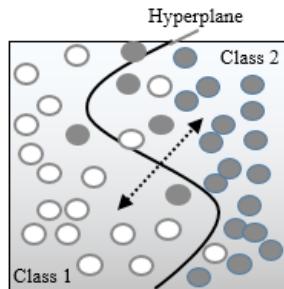
The remainder of this section is dedicated to the application of the multivariate logistic regression to the binomial classification.

Binomial classification

The logistic regression is popular for several reasons; some are as follows:

- It is available with most statistical software packages and open source libraries
- Its S-shape describes the combined effect of several explanatory variables
- Its range of values [0, 1] is intuitive from a probabilistic perspective

Let's consider the classification problem using two classes. As discussed in the *Validation* section in *Chapter 2, Hello World!*, even the best classifier produces false positives and false negatives. The training procedure for a binomial classification is illustrated in the following diagram:



An illustration of the binomial classification for a two-dimension dataset

The purpose of the training is to compute the **hyperplane** that separates the observations into two categories or classes. Mathematically speaking, a hyperplane in an n-dimensional space (number of features) is a subspace of $n - 1$ dimensions, as described in the *Manifolds* section in *Chapter 4, Unsupervised Learning*. The separating hyperplane of a three-dimension space is a curved surface. The separating hyperplane of a two-dimension problem (plane) is a line. In our preceding example, the hyperplane segregates/separates a training set into two very distinct classes (or groups), **Class 1** and **Class 2**, in an attempt to reduce the overlap (false positive and false negative).

The equation of the hyperplane is defined as the logistic function of the dot product of the regression parameters (or weights) and features.

The logistic function accentuates the difference between the two groups of training observations, separated by the hyperplane. It *pushes the observations away* from the separating hyperplane toward either classes.

In the case of two classes, $c1$ and $c2$ with their respective probabilities, $p(C=c1 | X=x_i | w) = p(x_i | w)$ and $p(C=c2 | X=x_i | w) = 1 - p(x_i | w)$, where w is the model parameters set or weights in the case of the logistic regression.

The logistic regression

M13: The log likelihood for N observations x_i given regression weights w is defined as:

$$\mathcal{L}(w) = \sum_{i=0}^{N-1} \log p(x_i | w)$$

M14: Conditional probabilities $p(x | w)$ with regression weights w , using the logistic function for N observations with d features $\{x_{ij}\}_{j=0:d-1}$ is defined as:



$$x_i = \{1, x_{i0}, \dots, x_{id-1}\} \quad p(x_i | w) = \frac{1}{1 + e^{-w^T x_i}} \quad w^T x_i = \sum_{j=0}^d w_j x_{ij}$$

M15: The sum of square errors, sse , for the binomial logistic regression with weights w , input values x_i , and expected binary outcome y is as follows:

$$sse(w) = \frac{1}{2} \sum_{i=0}^{N-1} \left\{ y_i - \log(1 + e^{-w^T x_i}) \right\}^2 \quad y \in \{0,1\}$$

M16: The computation of the weights w of the logistic regression by maximizing the log likelihood, given the input data x_i and expected outcome (labels) y_i is defined as:

$$\frac{\partial \mathcal{L}(\tilde{w})}{\partial w_j} = \sum_{i=0}^{N-1} x_{ij} \left(y_i - \frac{1}{1 + e^{-\tilde{w}^T x_i}} \right) = 0$$

Let's implement the logistic regression without regularization using the Apache Commons Math library. The library contains several least squares optimizers that allow you to specify the optimizer minimizing algorithm for the loss function in the logistic regression class, `LogisticRegression`.

The constructor for the `LogisticRegression` class follows a very familiar pattern: it defines an `ITransform` data transformation, whose model is implicitly derived from the input data (training set), as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 2). The output of the `|>` predictor is a class ID, and therefore, the `V` type of the output is an `Int` (line 3):

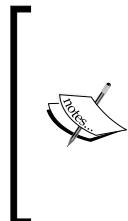
```
class LogisticRegression[T <: AnyVal] (
    xt: XVSeries[T],
    expected: Vector[Int],
    optimizer: LogisticRegressionOptimizer) //1
    (implicit f: T => Double)
extends ITransform[Array[T]](xt) with Regression
with Monitor[Double] { //2

    type V = Int //3
    override def train: RegressionModel //4
    def |> : PartialFunction[Array[T], Try[V]]
}
```

The parameters of the logistic regression class are the multivariate `xt` time series (features), the target or expected classes, `expected`, and the `optimizer` used to minimize the loss function or residual sum of squares (line 1). In the case of the binomial logistic regression, `expected` are assigned the value of `1` for one class and `0` for the other.

The `Monitor` trait is used to collect the profiling information during training (refer to the `Monitor` section under *Utility classes* in the *Appendix A, Basic Concepts*).

The purpose of the training is to determine the regression weights that minimize the loss function, as defined in the **M14** formula as well as the residual sum of squares (line 4).



Target values

There is no specific rule to assign the two values to the observed data for the binomial logistic regression: $\{-1, +1\}$, $\{0, 1\}$, or $\{\text{false}, \text{true}\}$. The values pair $\{0, 1\}$ is convenient because it allows the developer to reuse the code for multinomial logistic regression using normalized class values.

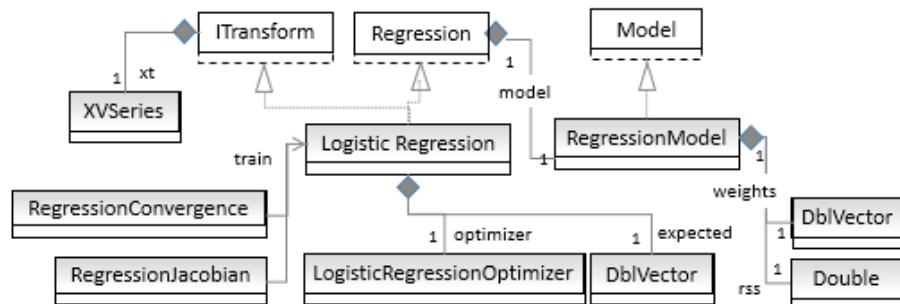
For convenience, the definition and configuration of the optimizer are encapsulated in the `LogisticRegressionOptimizer` class.

Design

The implementation of the logistic regression uses the following components:

- A `RegressionModel` model of the `Model` type that is initialized through training during the instantiation of the classifier. We reuse the `RegressionModel` type, which was introduced in the *Linear regression* section.
- The logistic regression class, `LogisticRegression`, that implements an `ITransform` for the prediction of future observations
- An adapter class named `RegressionJacobian` for the computation of the Jacobian
- An adapter class named `RegressionConvergence` to manage the convergence criteria and exit condition of the minimization of the sum of square errors

The key software components of the logistic regression are described in the following UML class diagram:



The UML class diagram for the logistic regression

The UML diagram omits the helper traits or classes such as `Monitor` or the Apache Commons Math components.

The training workflow

Our implementation of the training of the logistic regression model leverages either the Gauss-Newton or the Levenberg-Marquardt nonlinear least squares optimizers, (refer to the *Nonlinear least squares minimization* section in the *Appendix A, Basic Concepts*) packaged with the Apache Commons Math library.

The training of the logistic regression is performed by the `train` method.

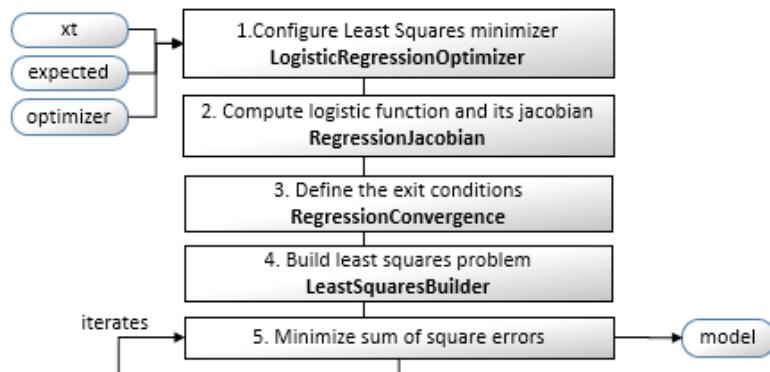
Handling exceptions from the Apache Commons Math library

The training of the logistic regression using the Apache Commons Math library requires handling the `ConvergenceException`, `DimensionMismatchException`, `TooManyEvaluationsException`, `TooManyIterationsException`, and `MathRuntimeException` exceptions. Debugging is greatly facilitated by understanding the context of these exceptions in the Apache library source code.

The implementation of the training method, `train`, relies on the following five steps:

1. Select and configure the least squares optimizer.
2. Define the logistic function and its Jacobian.
3. Specify the convergence and exit criteria.
4. Compute the residuals using the least squares problem builder.
5. Run the optimizer.

The workflow and the Apache Commons Math classes used in the training of the logistic regression are visualized by the following flow diagram:



The workflow for training the logistic regression using the Apache Commons Math library

The first four steps are required by the Apache Commons Math library to initialize the configuration of the logistic regression prior to the minimization of the loss function. Let's start with the configuration of the least squares optimizer:

```
def train: RegressionModel = {
  val weights0 = Array.fill(data.head.length +1)(INITIAL_WEIGHT)
  val lrJacobian = new RegressionJacobian(data, weights0) //5
```

```
val exitCheck = new RegressionConvergence(optimizer) //6

def createBuilder: LeastSquaresProblem //7
val optimum = optimizer.optimize(createBuilder) //8
RegressionModel(optimum.getPoint.toArray, optimum.getRMS)
}
```

The `train` method implements the last four steps of the computation of the regression model:

- Computation of logistic values and the Jacobian matrix (line 5).
- Initialization of the convergence criteria (line 6).
- Definition of the least square problem (line 7).
- Minimization of the sum of square errors (line 8). It is performed by the optimizer as part of the constructor of `LogisticRegression`.

Step 1 – configuring the optimizer

In this step, you have to specify the algorithm to minimize the residual of the sum of the squares. The `LogisticRegressionOptimizer` class is responsible for configuring the optimizer. The class has the following two purposes:

- Encapsulating the configuration parameters for the optimizer
- Invoking the `LeastSquaresOptimizer` interface defined in the Apache Commons Math library

The code will be as follows:

```
class LogisticRegressionOptimizer(
    maxIters: Int,
    maxEvals: Int,
    eps: Double,
    lsOptimizer: LeastSquaresOptimizer) { //9
  def optimize(lsProblem: LeastSquaresProblem): Optimum =
    lsOptimizer.optimize(lsProblem)
}
```

The configuration of the logistic regression optimizer is defined as the maximum number of iterations, `maxIters`, the maximum number of evaluations, `maxEval`, for the logistic function and its derivatives, the `eps` convergence criteria of the residual sum of squares, and the instance of the least squares problem (line 9).

Step 2 – computing the Jacobian matrix

The next step consists of computing the value of the logistic function and its first order partial derivatives with respect to the weights by overriding the `value` method of the `fitting.leastsquares.MultivariateJacobianFunction` Apache Commons Math interface:

```

class RegressionJacobian[T <: AnyVal] ( //10
    xv: XvSeries[T],
    weights0: DblArray)(implicit f: T => Double)
extends MultivariateJacobianFunction {

  type GradientJacobian = Pair[RealVector, RealMatrix]
  override def value(w: RealVector): GradientJacobian = { //11
    val gradient = xv.map( g => { //12
      val f = logistic(dot(g, w)) //13
      (f, f*(1.0-f)) //14
    })
    xv.zipWithIndex //15
    ./: (Array.ofDim[Double](xv.size, weights0.size)) {
      case (j, (x,i)) => {
        val df = gradient(i)._2
        Range(0, x.size).foreach(n => j(i)(n+1) = x(n)*df)
        j(i)(0) = 1.0; j //16
      }
    }
    (new ArrayRealVector(gradient.map(_._1).toArray),
     new Array2DRowRealMatrix(jacobian)) //17
  }
}

```

The constructor for the `RegressionJacobian` class requires the following two arguments (line 10):

- The `xv` time series of observations
- The `weights0` initial regression weights

The `value` method uses the `RealVector`, `RealMatrix`, `ArrayRealVector`, and `Array2DRowRealMatrix` primitive types defined in the `org.apache.commons.math3.linear` Apache Commons Math package (line 11). It takes the `w` regression weight as an argument, computes the gradient (line 12) of the logistic function (line 13) for each data point, and returns the value and its derivative (line 14).

The Jacobian matrix is populated with the values of the derivative of the logistic function (line 15). The first element of each column of the Jacobian matrix is set to 1.0 to take into account the intercept (line 16). Finally, the value function returns the pair of gradient values and the Jacobian matrix using types that comply with the signature of the value method in the Apache Commons Math library (line 17).

Step 3 – managing the convergence of the optimizer

The third step defines the exit condition for the optimizer. It is accomplished by overriding the converged method of the parameterized ConvergenceChecker interface in the org.apache.commons.math3.optim Java package:

```
val exitCheck = new ConvergenceChecker[PointVectorValuePair] {
    override def converged(
        iters: Int,
        prev: PointVectorValuePair,
        current: PointVectorValuePair): Boolean =
        sse(prev.getValue, current.getValue) < optimizer.eps
        && iters >= optimizer.maxIterations //18
}
```

This implementation computes the convergence or exit condition as follows:

- The sse sum of square errors between weights of two consecutive iterations is smaller than the eps convergence criteria
- The iters value exceeds the maximum number of iterations, maxIterations, allowed (line 18)

Step 4 – defining the least squares problem

The Apache Commons Math least squares optimizer package requires all the input to the nonlinear least squares minimizer to be defined as an instance of the LeastSquareProblem generated by the factory LeastSquareBuilder class:

```
def createBuilder: LeastSquaresProblem =
    (new LeastSquaresBuilder).model(lrJacobian) //19
    .weight(MatrixUtils.createRealDiagonalMatrix(
        Array.fill(xt.size)(1.0))) //20
    .target(expected.toArray) //21
    .checkerPair(exitCheck) //22
    .maxEvaluations(optimizer.maxEvals) //23
    .start(weights0) //24
    .maxIterations(optimizer.maxIterations) //25
    .build
```

The diagonal elements of the weights matrix are initialized to 1.0 (line 20). Besides the initialization of the model with the `lrJacobian` Jacobian matrix (line 19), the sequence of method invocations sets the maximum number of evaluations (line 23), maximum number of iterations (line 25), and the exit condition (line 22).

The regression weights are initialized with the `weights0` weights as arguments of the constructor for `LogisticRegression` (line 24). Finally, the expected or target values are initialized (line 21).

Step 5 – minimizing the sum of square errors

The training is executed with a simple call to the `lsp` least squares minimizer:

```
val optimum = optimizer.optimize(lsp)
(optimum.getPoint.toArray, optimum.getRMS)
```

The regression coefficients (or weights) and the **residuals mean square (RMS)** are returned by invoking the `getPoint` method on the `Optimum` class of the Apache Commons Math library.

Test

Let's test our implementation of the binomial multivariate logistic regression using the example of the price variation versus volatility and volume of the Copper ETF, which is used in the previous two sections. The only difference is that we need to define the target values as 0 if the ETF price decreases between two consecutive trading sessions, and 1 otherwise:

```
import YahooFinancials._
val maxIters = 250
val maxEvals = 4500
val eps = 1e-7

val src = DataSource(path, true, true, 1) //26
val optimizer = new LevenbergMarquardtOptimizer //27

for {
    price <- src.get(adjClose) //28
    volatility <- src.get(volatility) //29
    volume <- src.get(volume) //30
    (features, expected) <- differentialData(volatility,
    volume, price, diffInt) //31
    lsOpt <- LogisticRegressionOptimizer(maxIters, maxEvals,
    eps, optimizer) //32
```

```
regr <- LogisticRegression[Double](features, expected, lsOpt)
pfnRegr <- Try(regr |>) //33
}
yield {
  show(s"${LogisticRegressionEval.toString(regr)}")
  val predicted = features.map(pfnRegr(_))
  val delta = predicted.view.zip(expected.view)
    .map{case(p, e) => if(p.get == e) 1 else 0}.sum
  show(s"Accuracy: ${delta.toDouble/expected.size}")
}
```

Let's take a look at the steps required for the execution of the test that consists of collecting data, initializing the parameters for the minimization of the sum of square errors, training a logistic regression model, and running the prediction:

1. Create a `src` data source to extract the market and trading data (line 26).
2. Select the `LevenbergMarquardtOptimizer` Levenberg-Marquardt algorithm as `optimizer` (line 27).
3. Load the daily closing price (line 28), volatility within a trading session (line 29), and the volume daily trading (line 30) for the ETF CU.
4. Generate the labeled data as a pair of features (the relative volatility and relative volume for the ETF) and the `expected` outcome {0, 1} for training the model for which 1 represents the increase in the price and 0 represents the decrease in the price (line 31). The `differentialData` generic method of the `XTSeries` singleton is described in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*.
5. Instantiate the `lsOpt` optimizer to minimize the sum of square errors during training (line 32).
6. Train the `regr` model and return the `pfnRegr` predictor partial function (line 33).

There are many alternative optimizers available to minimize the sum of square errors optimizers (refer to the *Nonlinear least squares minimization* section in the *Appendix A, Basic Concepts*).

Levenberg-Marquardt parameters

The driver code uses the `LevenbergMarquardtOptimizer` with the default tuning parameters' configuration to keep the implementation simple. However, the algorithm has a few important parameters, such as the relative tolerance for cost and matrix inversion, that are worth tuning for commercial applications (refer to the *Levenberg-Marquardt* section under *Nonlinear least squares minimization* in the *Appendix A, Basic Concepts*).



The execution of the test produces the following results:

- The **residual mean square** is 0.497
- **Weights** are -0.124 for intercept, 0.453 for ETF volatility, and -0.121 for ETF volume

The last step is the classification of the real-time data.

Classification

As mentioned earlier and despite its name, the binomial logistic regression is actually a binary classifier. The classification method is implemented as an implicit data transformation |>:

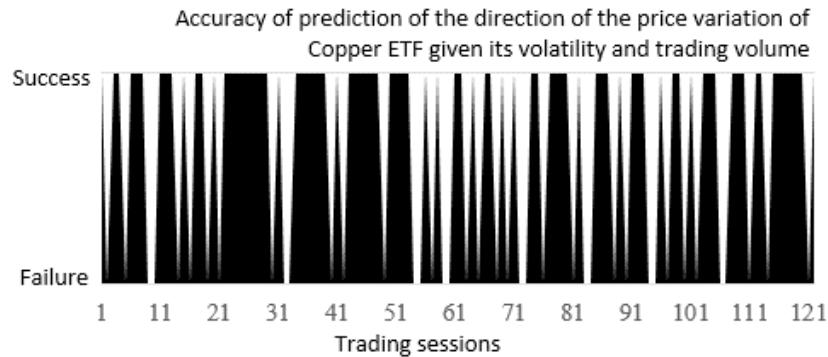
```
val HYPERPLANE = - Math.log(1.0/INITIAL_WEIGHT -1)
def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T] if(isModel &&
    model.size-1 == x.length && isModel) =>
    Try(if(dot(x, model) > HYPERPLANE) 1 else 0 ) //34
}
```

The dot (or inner) product of the observation x with the weights $model$ is evaluated against the hyperplane. The predicted class is 1 if the produce exceeds `HYPERPLANE`, and 0 otherwise (line 34).

Class identification

 The class that the new data x belongs to is determined by the $dot(x, weights) > 0.5$ test, where dot is the product of the features and the regression weights ($w_0 + w_1 \cdot volatility + w_2 \cdot volume$). You may find different classification schemes in the scientific literature.

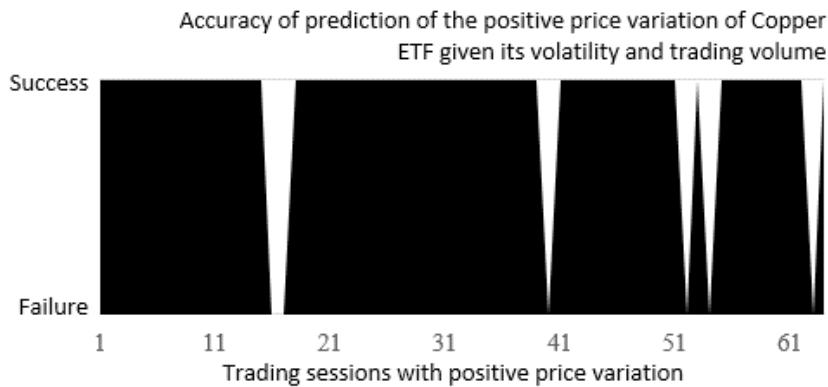
The direction of the price variation of the Copper ETF, $CU\ price(t+1) - price(t)$, is compared to the direction predicted by the logistic regression. The result is plotted with the **success** value if the positive or negative direction is correctly predicted; otherwise, it is plotted with the **failure** value:



The prediction of the direction of the variation of price of the Copper ETF using the logistic regression

The logistic regression was able to classify 78 out of 121 trading sessions (65 percent accuracy).

Now, let's use the logistic regression to predict the positive price variation for the Copper ETF, given its volatility and trading volume. This trading or investment strategy is known as being *long on the market*. This particular use case ignores the trading sessions for which the price was either flat or declined:



The prediction of the direction of the variation of price of the Copper ETF using the logistic regression

The logistic regression was able to correctly predict the positive price variation for 58 out of 64 trading sessions (90.6 percent accuracy). What is the difference between the first and second test cases?

In the first case, the $w_0 + w_1.volatility + w_2.volume$ separating hyperplane equation is used to segregate the features generating either the positive or negative price variation. Therefore, the overall accuracy of the classification is negatively impacted by the overlap of the features from the two classes.

In the second case, the classifier has to consider only the *observations located on the positive side* of the hyperplane equation, without taking into account the false negatives.

Impact of rounding errors



Under some circumstances, the generation of the rounding errors during the computation of the Jacobian matrix has an impact on the accuracy of the $w_0 + w_1.volatility + w_2.volume$ separating hyperplane equation. It reduces the accuracy of the prediction of both the positive and negative price variation.

The accuracy of the binary classifier can be further improved by considering the positive variation of the price using a margin error EPS as $price(t+1) - price(t) > EPS$.

The validation methodology



The validation set is generated by randomly selecting observations from the original labeled dataset. A formal validation requires you to use a K-fold validation methodology to compute the recall, precision, and F1 measure for the logistic regression model.

Summary

This concludes the description and implementation of the linear and logistic regression and the concept of regularization to reduce overfitting. Your first analytical projects using machine learning will (or did) likely involve a regression model of some type. Regression models, along with the Naïve Bayes classification, are the most understood techniques for those without a deep knowledge of statistics or machine learning.

After the completion of this chapter, you will hopefully have a grasp on the following topics:

- The concept of linear and nonlinear least squares-based optimization
- The implementation of ordinary least square regression as well as logistic regression
- The impact of regularization with an implementation of the ridge regression

The logistic regression is also the foundation of the conditional random fields, as described in the *Conditional random fields* section in *Chapter 7, Sequential Data Models*, and multilayer perceptrons, which was introduced in the *The multilayer perceptron* section in *Chapter 9, Artificial Neural Networks*.

Contrary to the Naïve Bayes models (refer to *Chapter 5, Naïve Bayes Classifiers*), the least squares or logistic regression does not impose the condition that the features have to be independent. However, the regression models do not take into account the sequential nature of a time series such as asset pricing. The next chapter, which is dedicated to models for sequential data, introduces two classifiers that take into account the time dependency in a time series: the hidden Markov model and conditional random fields.

7

Sequential Data Models

The universe of Markov models is vast and encompasses computational concepts such as the Markov decision process, discrete Markov, Markov chain Monte Carlo for Bayesian networks, and hidden Markov models.

Markov processes, and more specifically, the **hidden Markov model (HMM)**, are commonly used in speech recognition, language translation, text classification, document tagging, and data compression and decoding.

The first section of this chapter introduces and describes the hidden Markov model with the full implementation of the three canonical forms of the hidden Markov model using Scala. This section covers the different dynamic programming techniques used in the evaluation, decoding, and training of the hidden Markov model. The design of the classifier follows the same pattern as the logistic and linear regression, as described in *Chapter 6, Regression and Regularization*.

The second and last section of this chapter is dedicated to a discriminative (labels conditional to observations) alternative to the hidden Markov model: conditional random fields. The open source CRF Java library authored by Sunita Sarawagi from the Indian Institute of Technology, Bombay, is used to create a predictive model using conditional random fields [7:1].

Markov decision processes

This first section also describes the basic concepts you need to know in order to understand, develop, and apply the hidden Markov model. The foundation of the Markovian universe is the concept known as the **Markov property**.

The Markov property

The Markov property is a characteristic of a stochastic process where the conditional probability distribution of a future state depends on the current state and not on its past states. In this case, the transition between the states occurs at a discrete time, and the Markov property is known as the **discrete Markov chain**.

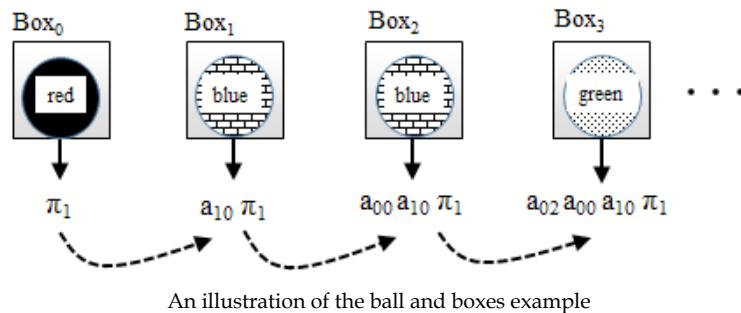
The first order discrete Markov chain

The following example is taken from *Introduction to Machine Learning*, E. Alpaydin [7:2].

Let's consider the following use case. N balls of different colors are hidden in N boxes (one each). The balls can have only three colors (Blue, Red, and Green). The experimenter draws the balls one by one. The state of the discovery process is defined by the color of the latest ball drawn from one of the boxes: $S_0 = \text{Blue}$, $S_1 = \text{Red}$, and $S_2 = \text{Green}$.

Let $\{\pi_0, \pi_1, \pi_2\}$ be the initial probabilities for having an initial set of color in each of the boxes.

Let q_t denote the color of the ball drawn at the time t . The probability of drawing a ball of color S_k at the time k after drawing a ball of the color S_j at the time j is defined as $p(q_t = S_k | q_{t-1} = S_j) = a_{jk}$. The probability of drawing a red ball in the first attempt is $p(q_{t0} = S_1) = \pi_1$. The probability of drawing a blue ball in the second attempt is $p(q_{t1} = S_1 | q_{t0} = S_1) = \pi_1 a_{10}$. The process is repeated to create a sequence of the state $\{S_t\} = \{\text{Red}, \text{Blue}, \text{Blue}, \text{Green}, \dots\}$ with the following probability: $p(q_0 = S_1) \cdot p(q_1 = S_0 | q_0 = S_1) \cdot p(q_2 = S_1 | q_1 = S_0) \cdot p(q_3 = S_2 | q_2 = S_1) \dots = \pi_1 \cdot a_{10} \cdot a_{00} \cdot a_{02} \dots$. The sequence of states/colors can be represented as follows:



Let's estimate the probabilities p using historical data (learning phase):

- The estimation of the probability of drawing a red ball (S_1) in the first attempt is π_{1r} , which is computed as the number of sequences starting with S_1 (red) / total number of balls.
- The estimation of the probability of retrieving a blue ball in the second attempt is a_{10r} , the number of sequences for which a blue ball is drawn after a red ball / total number of sequences, and so on.

Nth-order Markov

The Markov property is popular mainly because of its simplicity. As you will discover while studying the hidden Markov model, having a state solely dependent on the previous state allows us to apply efficient dynamic programming techniques. However, some problems require dependencies between more than two states. These models are known as Markov random fields.

Although the discrete Markov process can be applied to trial and error types of applications, its applicability is limited to solving problems for which the observations do not depend on hidden states. Hidden Markov models are a commonly applied technique to meet such a challenge.

The hidden Markov model

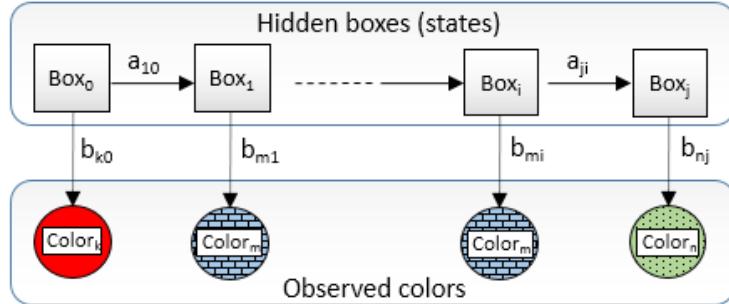
The hidden Markov model has numerous applications related to speech recognition, face identification (biometrics), and pattern recognition in pictures and videos [7:3].

A hidden Markov model consists of a Markov process (also known as a Markov chain) for observations with a discrete time. The main difference with the Markov processes is that the states are not observable. A new observation is emitted with a probability known as the emission probability each time the state of the system or model changes.

There are two sources of randomness, which are as follows:

- Transition between states
- Emission of an observation when a state is given

Let's reuse the boxes and balls example. If the boxes are hidden states (nonobservable), then the user draws the balls whose color is not visible. The emission probability is the probability $b_{ik} = p(o_t = \text{color}_k | q_t = S_i)$ to retrieve a ball of the color k from a hidden box I , as shown in the following diagram:



The hidden Markov model for the balls and boxes example

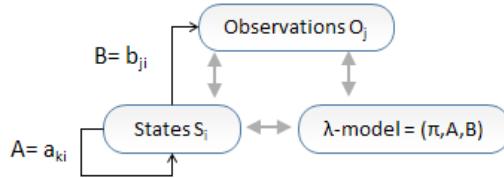
In this example, we do not assume that all the boxes contain balls of different colors. We cannot make any assumptions on the order as defined by the transition a_{ij} . The HMM does not assume that the number of colors (observations) is identical to the number of boxes (states).

 **Time invariance**
Contrary to the Kalman filter, for example, the hidden Markov model requires that the transition elements, a_{ij} , are independent of time. This property is known as stationary or homogeneous restriction.

Keep in mind that the observations, in this case the color of the balls, are the only tangible data available to the experimenter. From this example, we can conclude that a formal HMM has three components:

- A set of observations
- A sequence of hidden states
- A model that maximizes the joint probability of the observations and hidden states, known as the Lambda model

A Lambda model, λ , is composed of initial probabilities π , the probabilities of state transitions as defined by the matrix A , and the probabilities of states emitting one or more observations, as shown in the following diagram:



The visualization of the HMM key components

The preceding diagram illustrates that, given a sequence of observations, the HMM tackles the following three problems known as canonical forms:

- **CF1 (evaluation):** This evaluates the probability of a given sequence of observations O_p given a model $\lambda = (\pi, A, B)$
- **CF2 (training):** This identifies (or learns) a model $\lambda = (\pi, A, B)$, given a set of observations O
- **CF3 (decoding):** This estimates the state sequence Q with the highest probability to generate a given set of observations O and a model λ

The solution to these three problems uses dynamic programming techniques. However, we need to clarify the notations prior to diving into the mathematical foundation of the hidden Markov model.

Notations

One of the challenges of describing the hidden Markov model is the mathematical notation that sometimes differs from author to author. From now on, we will use the following notation:

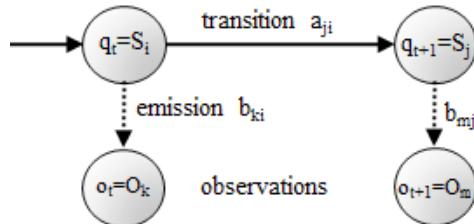
	Description	Formulation
N	The number of hidden states	
S	A finite set of N hidden states	$S = \{S_0, S_1, \dots, S_{N-1}\}$
M	The number of observation symbols	
q_t	The state at time or step t	
Q	A time sequence of states	$Q = \{q_0, q_1, \dots, q_{n-1}\} = Q_{0:n-1}$
T	The number of observations	
o_t	The observation at time t	
O	A finite sequence of T observations	$O = \{o_0, o_1, \dots, o_{T-1}\} = O_{0:T-1}$
A	The state transition probability matrix	$a_{ji} = p(q_{t+1}=S_i \mid q_t=S_j)$
B	The emission probability matrix	$b_{jk} = p(o_t=O_k \mid q_t=S_j)$
π	The initial state probability vector	$\pi_i = p(q_0=S_j)$
λ	The hidden Markov model	$\lambda = (\pi, A, B)$



Variance in the notation

Some authors use the symbol z to represent the hidden states instead of q and x to represent the observations O .

For convenience, let's simplify the notation of the sequence of observations and states using the condensed form: $p(O_{0:T}, q_t | \lambda) = p(O_0, O_1, \dots, O_T, q_t | \lambda)$. It is quite common to visualize a hidden Markov model with a lattice of states and observations, which is similar to our description of the boxes and balls examples, as shown here:



The formal HMM-directed graph

The state S_i is observed as O_k at time t , before being transitioned to the state S_j observed as O_m at the time $t+1$. The first step in the creation of our HMM is the definition of the class that implements the lambda model $\lambda = (\pi, A, B)$ [7:4].

The lambda model

The three canonical forms of the hidden Markov model rely heavily on manipulation and operations on matrices and vectors. For convenience, let's define an `HMMConfig` class that contains the dimensions used in the HMM:

```

class HMMConfig(val numObs: Int, val numStates: Int,
               val numSymbols: Int, val maxIters: Int, val eps: Double)
  extends Config
  
```

The input parameters for the class are as follows:

- `numObs`: This is the number of observations
- `numStates`: This is the number of hidden states
- `numSymbols`: This is the number of observation symbols or features
- `maxIters`: This is the maximum number of iterations required for the HMM training
- `eps`: This is the convergence criteria for the HMM training



Consistency with a mathematical notation

The implementation uses `numObs` (with respect to `numStates` and `numSymbols`) to represent programmatically the number of observations T (with respect to the N hidden states and M features). As a general rule, the implementation reuses the mathematical symbols as much as possible.

The `HMMConfig` companion object defines the operations on ranges of index of matrix rows and columns. The `foreach` (line 1), `foldLeft` (`/:`) (line 2), and `maxBy` (line 3) methods are regularly used in each of the three canonical forms:

```
object HMMConfig {
    def foreach(i: Int, f: Int => Unit): Unit =
        Range(0, i).foreach(f) //1
    def /:(i: Int, f: (Double, Int) => Double, zero: Double) =
        Range(0, i).:/((zero)(f)) //2
    def maxBy(i: Int, f: Int => Double): Int =
        Range(0,i).maxBy(f) //3
    ...
}
```



The λ notation

The λ model in the HMM should not be confused with the regularization factor discussed in the L_n roughness penalty section in *Chapter 6, Regression and Regularization*.

As mentioned earlier, the lambda model is defined as a tuple of the transition probability matrix A , emission probability matrix B , and the initial probability π . It is easily implemented as an `HMMModel` class using the `DMatrix` class, as defined in the *Utility classes* section in the *Appendix A, Basic Concepts*. The simplest constructor for the `HMMModel` class is invoked in the case where the state-transition probability matrix, the emission probability matrix, and the initial states are known, as shown in the following code:

```
class HMMModel( val A: DMatrix, val B: DMatrix, var pi: DblArray,
    val numObs: Int) { //4
    val numStates = A.nRows
    val numSymbols = B.nCols

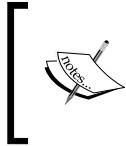
    def setAlpha(obsSeqNum: Vector[Int]): DMatrix
    def getAlphaVal(a: Double, i: Int, obsId: Int): Double
    def getBetaVal(b: Double, i: Int, obsId: Int): Double
    def update(gamma: Gamma, diGamma: DiGamma,
        obsSeq: Vector[Int])
    def normalize: Unit
}
```

The constructor of the `HMMModel` class has the following four arguments (line 4):

- `A`: This is the state transition probabilities matrix
- `B`: This is the omission probabilities matrix
- `pi`: This is the initial probability for the states
- `numObs`: This is the number of observations

The number of states and symbols are extracted from the dimension of the `A` and `B` matrices.

The `HMMModel` class has several methods that will be described in detail whenever they are required for the execution of the model. The probabilities for the `pi` initial states are unknown, and therefore, they are initialized with a random generator of values [0, 1].



Normalization

Input states and observation data may have to be normalized and converted to probabilities before we initialize the `A` and `B` matrices.



The other two components of the HMM are the sequence of observations and the sequence of hidden states.

Design

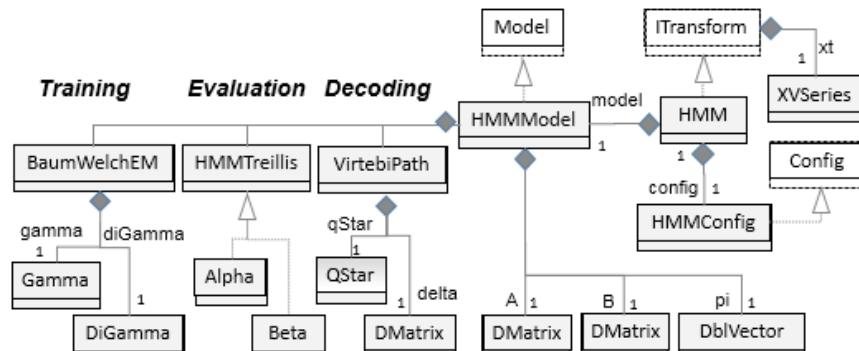
The canonical forms of the HMM are implemented through dynamic programming techniques. These techniques rely on variables that define the state of the execution of the HMM for any of the canonical forms:

- **Alpha** (the forward pass): The probability of observing the first $t < T$ observations for a specific state at S_i for the observation t is $\alpha_t(i) = p(O_{0:t}, q_t = S_i | \lambda)$
- **Beta** (the backward pass): The probability of observing the remainder of the sequence qt for a specific state is $\beta_t(i) = p(O_{t+1:T} | q_t = S_i, \lambda)$
- **Gamma**: The probability of being in a specific state given a sequence of observations and a model is $\gamma_t(i) = p(q_t = S_i | O_{0:T}, \lambda)$
- **Delta**: This is the sequence that has the highest probability path for the first i observations defined for a specific test $\delta_t(i)$
- **Qstar**: This is the optimum sequence q^* of states $Q_{0:T}$

- **DiGamma:** The probability of being in a specific state at t and another defined state at $t + 1$ given the sequence of observations and the model is $\gamma_t(i,j) = p(q_t=S_i, q_{t+1}=S_j | O_{0:T-1}, \lambda)$

Each of the parameters is described mathematically and programmatically in the section related to each specific canonical form. The `Gamma` and `DiGamma` classes are used and described in the evaluation canonical form. The `DiGamma` singleton is described as part of the Viterbi algorithm to extract the sequence of states with the highest probability given a λ model and a set of observations.

The list of dynamic programming-related algorithms used in any of the three canonical forms is visualized through the class hierarchy of our implementation of the HMM:



Scala classes' hierarchy for HMM (the UML class diagram)

The UML diagram omits the utility traits and classes such as `Monitor` or the Apache Commons Math components.

The λ model, the HMM state, and the sequence of observations are all the elements needed to implement the three canonical cases. Each class is described as needed in the description of the three canonical forms of HMM. It is time to dive into the implementation details of each of the canonical forms, starting with the evaluation.

The execution of any of the three canonical forms relies on dynamic programming techniques (refer to the *Overview of dynamic programming* section in the *Appendix A, Basic Concepts*) [7:5]. The simplest of the dynamic programming techniques is a single traversal of the observations/state chain.

Evaluation – CF-1

The objective is to compute the probability (or likelihood) of the observed sequence O_t given a λ model. A dynamic programming technique is used to break down the probability of the sequence of observations into two probabilities (**M1**):

$$p(O_{0:T-1}|\lambda) \propto p(O_{0:t}|\lambda) \cdot p(O_{t+1:T-1}|\lambda)$$

The likelihood is computed by marginalizing over all the hidden states $\{S_i\}$ [7:6] (**M2**):

$$p(O_{0:T-1}|\lambda) = \sum_{i=0}^{N-1} p(O_{0:T-1}, q_t = S_i | \lambda)$$

If we use the notation introduced in the previous chapter for alpha and beta variables, the probability for the observed sequence O_t given a λ model can be expressed as follows (**M3**):

$$p(O_{0:T-1}|\lambda) = \sum_i \alpha_t(i) \cdot \beta_t(i)$$

The product of the α and β probabilities can potentially underflow. Therefore, it is recommended that you use the log of the probabilities instead of the probabilities.

Alpha – the forward pass

The computation of the probability of observing a specific sequence given a sequence of hidden states and a λ model relies on a two-pass algorithm. The alpha algorithm consists of the following steps:

1. Compute the initial alpha value [**M4**]. The value is then normalized by the sum of alpha values across all the hidden states [**M5**].
2. Compute the alpha value iteratively for the time 0 to time t , and then normalize it by the sum of alpha values for all states [**M6**].
3. The final step is to compute of the log of the probability of observing the sequence [**M7**].



Performance consideration

A direct computation of the probability of observing a specific sequence requires $2TN_2$ multiplications. The iterative alpha and beta classes reduce the number of multiplications to N_2T .

For those with some inclination toward mathematics, the computation of the alpha matrix is defined in the following information box.



Alpha (the forward pass)

M4: Initialization is defined as:

$$\alpha_0(i) = \pi_i \cdot b_i(O_0)$$

M5: Normalization of initial values $N - 1$ is defined as:

$$\hat{\alpha}_0(i) = \alpha_0(i) / \sum_{j=0}^{N-1} \alpha_0(j)$$

M6: Normalized summation is defined as:



$$\alpha_t(i) = \sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji} b_i(O_t) \quad c_t = 1 / \sum_{i=0}^{N-1} \alpha_t(i) \quad \hat{\alpha}_t(i) = \alpha_t(i) \cdot c_t$$

M7: The probability of observing a sequence given a lambda model and states is defined as:

$$\log p(O|\lambda) = - \sum_{j=0}^{T-1} \log \left(\frac{1}{\sum_{i=0}^{N-1} \hat{\alpha}_t(i)} \right)$$

Let's take a look at the implementation of the alpha class in Scala, using the referenced number of the mathematical expressions of the alpha class. The alpha and beta values have to be normalized [M3], and therefore, we define an `HMMTreillis` base class for the alpha and beta algorithms that implements the normalization:

```
class HMMTreillis(numObs: Int, numStates: Int) { //5
    var treillis: DMatrix = _ //6
```

```

val ct = Array.fill(numObs) (0.0)

def normalize(t: Int): Unit = { //7
    ct.update(t, /:(numStates, (s, n) => s + treillis(t, n)))
    treillis /= (t, ct(t))
}
def getTreillis: DMatrix = treillis
}

```

The `HMMTreillis` class has two configuration parameters: the number of observations, `numObs`, and the number of states, `numStates` (line 5). The `treillis` variable represents the scaling matrix used in the alpha (or forward) and beta (or backward) passes (line 6).

The normalization method, `normalize`, implements the **M6** formula by recomputing the `ct` scaling factor (line 7).


Computation efficiency

Scala's `reduce`, `fold`, and `foreach` methods are far more efficient than the `for` loop. You need to keep in mind that the main purpose of the `for` loop in Scala is the monadic composition of the `map` and `flatMap` operations.

The computation of the `alpha` variable in the `Alpha` class follows the same computation flow as defined in the **M4**, **M5**, and **M6** mathematical expressions:

```

class Alpha(lambda: HMMModel, obsSeq: Vector[Int]) //8
extends HMMTreillis(lambda.numObs, lambda.numStates) {

    val alpha: Double = Try {
        treillis = lambda.setAlpha(obsSeq) //9
        normalize(0) //10
        sumUp //11
    }.getOrElse(Double.NaN)

    override def isInitialized: Boolean = alpha != Double.NaN

    val last = lambda.numObs-1
    def sumUp: Double = {
        foreach(1, lambda.numObs, t => {
            updateAlpha(t) //12
            normalize(t) //13
        })
    }
}

```

```

    /:(lambda.numStates, (s,k) => s + treillis(last, k))
}

def updateAlpha(t: Int): Unit =
  foreach(lambda.numStates, i => { //14
    val newAlpha = lambda.getAlphaVal(treillis(t-1, i)
      treillis += (t, i, newAlpha, i, obsSeq(t)))
  })

def logProb: Double = /:(lambda.numObs, (s,t) => //15
  s + Math.log(ct(t)), Math.log(alpha))
}

```

The Alpha class has two arguments: the `lambda` model and the `obsSeq` sequence of observations (line 8). The definition of the scaling factor `alpha` initializes the `treillis` scaling matrix using the `HMMModel.setAlpha` method (line 9), normalizes the initial value of the matrix by invoking the `HMMTreillis.normalize` method for the first observation (line 10), and sums the matrix element to return the scaling factor by invoking `sumUp` (line 11).

The `setAlpha` method implements the mathematical expression **M4** as follows:

```

def setAlpha(obsSeq: Array[Int]): DMatrix =
  Range(0, numStates) ./: (DMatrix(numObs, numStates)) ((m, j) =>
    m += (0, j, pi(j)*B(j, obsSeq.head)))
}

```

The fold generates an instance of the `DMatrix` class, as described in the *Utility classes* section in the *Appendix A, Basic Concepts*.

The `sumUp` method implements the mathematical expression **M6** as follows:

- Update the `treillis` matrix of the scaling factor in the `updateAlpha` method (line 12)
- Normalize all the scaling factors for all the remaining observations (line 13)

The `updateAlpha` method updates the `treillis` scaling matrix by computing all the `alpha` factors for all states (line 14). The `logProb` method implements the mathematical expression **M7**. It computes the logarithm of the probability of observing a specific sequence, given the sequence of states and a predefined λ model (line 15).

The log probability

 The `logProb` method computes the logarithm of the probability instead of the probability itself. The summation of the logarithm of probabilities is less likely to cause an underflow than the product of probabilities.

Beta – the backward pass

The computation of beta values is similar to the `Alpha` class except that the iteration executes backward on the sequence of states.

The implementation of `Beta` is similar to the `alpha` class:

1. Compute (**M5**) and normalize (**M6**) the value of beta at $t = 0$ across states.
2. Compute and normalize iteratively the beta value at time $T - 1$ to t , which is updated from its value at $t + 1$ (**M7**).

Beta (the backward pass)

M8: Initialization of beta $\beta_{T-1}(t) = 1$.

M9: Normalization of initial beta values is defined as:



$$\hat{\beta}_{T-1}(i) = \beta_{T-1}(i) / \sum_{j=0}^{N-1} \beta_{T-1}(j)$$

M10: Normalized summation of beta is defined as:

$$\beta_t(i) = \sum_{j=0}^{N-1} \beta_{t+1}(j) \cdot a_{ij} \cdot b_j(O_{t+1}) \quad c_t = 1 / \sum_{j=0}^{N-1} \beta_t(j) \quad \hat{\beta}_t(i) = \beta_t(i) \cdot c_t$$

The definition of the `Beta` class is very similar to the `Alpha` class:

```
class Beta(lambda: HMMModel, obsSeq: Vector[Int])
  extends HMMTreillis(lambda.numObs, lambda.numStates) {

  val initialized: Boolean  //16

  override def isInitialized: Boolean = initialized
  def sumUp: Unit =    //17
    (lambda.numObs-2 to 0 by -1).foreach(t => { //18
```

```

        updateBeta(t)    //19
        normalize(t)
    })

def updateBeta(t: Int): Unit =
    foreach(lambda.numStates, i => {
        val newBeta = lambda.getBetaVal(treillis(t+1, i)
            treillis += (t, i, newBeta, i, obsSeq(t+1))) //20
    })
}

```

Contrary to the Alpha class, the Beta class does not generate an output value. The Beta class has an `initialized` Boolean attribute to indicate whether the constructor has executed successfully (line 16). The constructor updates and normalizes the beta matrix by traversing the sequence of observations backward from before the last observation to the first:

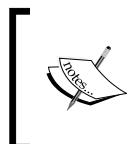
```

val initialized: Boolean = Try {
    treillis = DMatrix(lambda.numObs, lambda.numStates)
    treillis += (lambda.numObs-1, 1.0) //21
    normalize(lambda.numObs-1) //22
    sumUp //23
}.toBoolean("Beta initialization failed")

```

The initialization of the `treillis` beta scaling matrix of the `DMatrix` type assigns the value 1.0 to the last observation (line 21) and normalizes the beta values for the last observation, as defined in **M8** (line 22). It implements the mathematical expressions **M9** and **M10** by invoking the `sumUp` method (line 23).

The `sumUp` method is similar to `Alpha.sumUp` (line 17). It traverses the sequence of observations backward (line 18) and updates the beta scaling matrix, as defined in the mathematical expression **M9** (line 19). The implementation of the mathematical expression **M10** in the `updateBeta` method is similar to the alpha pass: it updates the `treillis` scaling matrix with the `newBeta` values computed in the `lambda` model (line 20).



Constructors and initialization

The alpha and beta values are computed within the constructors of their respective classes. The client code has to validate these instances by invoking `isInitialized`.

What is the value of a model if it cannot be created? The next canonical form CF2 leverages dynamic programming and recursive functions to extract the λ model.

Training – CF-2

The objective of this canonical form is to extract the λ model given a set of observations and a sequence of states. It is similar to the training of a classifier. The simple dependency of a current state on the previous state enables an implementation using an iterative procedure, known as the **Baum-Welch estimator** or **expectation-maximization (EM)**.

The Baum-Welch estimator (EM)

At its core, the algorithm consists of three steps and an iterative method, which is similar to the evaluation canonical form:

1. Compute the probability π (the gamma value at $t = 0$) (**M11**).
2. Compute and normalize the state's transition probabilities matrix A (**M12**).
3. Compute and normalize the matrix of emission probabilities B (**M13**).
4. Repeat steps 2 and 3 until the change of likelihood is insignificant.

The algorithm uses the digamma and summation gamma classes.

The Baum-Welch algorithm

M11: The joint probability of the state q_i at t and q_j at $t+1$ (digamma) is defined as:

$$\gamma_t(i,j) = p(q_t = S_i, q_{t+1} = S_j | O, \lambda)$$

$$\gamma_t(i,j) = \frac{\alpha_t(i)\alpha_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{j=0}^{N-1} \alpha_t(i)\beta_t(j)}$$

M12: The initial probabilities vector $N-1$ and sum of joint probabilities for all the states (gamma) are defined as:



$$\hat{\pi}_i = \gamma_0(i) \quad \gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i,j)$$

M13: The update of the transition probabilities matrix is defined as:

$$\hat{a}_{ij} = \frac{\sum_{t=0}^{T-1} [\gamma_t(i,j)]}{\sum_{t=0}^{T-1} \gamma_t(i)}$$

M14: The update of the emission probabilities matrix is defined as:

$$\hat{b}_{ij} = \frac{\sum_{t=0}^{T-1} \gamma_t(i,j)}{\sum_{t=0}^{T-1} \gamma_t(i)}$$

The Baum-Welch algorithm is implemented in the `BaumWelchEM` class and requires the following two inputs (line 24):

- The λ model, `lambda`, computed from the `config` configuration
- The `obsSeq` sequence (vector) of observations

The code will be as follows:

```
class BaumWelchEM(config: HMMConfig, obsSeq: Vector[Int]) { //24
    val lambda = HMMModel(config)
    val diGamma = new DiGamma(lambda.numObs, lambda.numStates)//25
    val gamma = new Gamma(lambda.numObs, lambda.numStates) //26
    val maxLikelihood: Option[Double] //27
}
```

The `DiGamma` class defines the joint probabilities for any consecutive states (line 25):

```
class DiGamma(numObs: Int, numStates: Int) {
    val diGamma = Array.fill(numObs-1)(DMatrix(numStates))
    def update(alpha: DMatrix, beta: DMatrix, A: DMatrix,
              B: DMatrix, obsSeq: Array[Int]): Try[Int]
}
```

The `diGamma` variable is an array of matrices that represents the joint probabilities of two consecutive states. It is initialized through an invocation of the `update` method, which implements the mathematical expression **M11**.

The `Gamma` class computes the sum of the joint probabilities across all the states (line 26):

```
class Gamma(numObs: Int, numStates: Int) {
    val gamma = DMatrix(numObs, numStates)
    def update(alpha: DMatrix, beta: DMatrix): Unit
}
```

The `update` method of the `Gamma` class implements the mathematical expression **M12**.

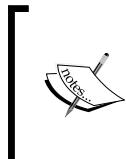
Source code for Gamma and DiGamma

The `Gamma` and `DiGamma` classes implement the mathematical expressions for the Baum-Welch algorithm. The `update` method uses simple linear algebra and is not described; refer to the documented source code for details.

The maximum likelihood, `maxLikelihood`, for the sequence of states given an existing lambda model and a sequence of observations (line 27) is computed using the `getLikelihood` tail recursive method, as follows:

```
val maxLikelihood: Option[Double] = Try {  
  
    @tailrec  
    def getLikelihood(likelihood: Double, index: Int): Double = {  
        lambda.update(gamma, diGamma, obsSeq) //28  
        val _likelihood = frwrdbckwrdLattice //29  
        val diff = likelihood - _likelihood  
  
        if( diff < config.eps ) _likelihood //30  
        else if (index >= config.maxIters) //31  
            throw new IllegalStateException(" ... ")  
        else getLikelihood(_likelihood, index+1)  
    }  
  
    val max = getLikelihood(frwrdbckwrdLattice, 0)  
    lambda.normalize //32  
    max  
}._toOption("BaumWelchEM not initialized", logger)
```

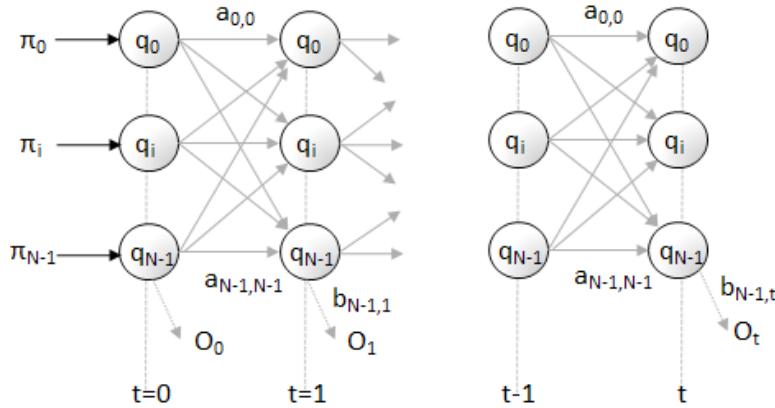
The `maxLikelihood` value implements the mathematical expressions **M13** and **M14**. The `getLikelihood` recursive method updates the lambda model matrices A and B and initial state probabilities p_i (line 28). The likelihood for the sequence of states is recomputed using the forward-backward lattice algorithm implemented in the `frwrdbckwrdLattice` method (line 29).



Update of lambda model

The update method of the `HMMModel` object uses simple linear algebra and is not described; refer to the documented source code for details.

The core of the Baum-Welch expectation maximization is the iterative forward and backward update of the lattice of states and observations between time t and $t + 1$. The lattice-based iterative computation is illustrated in the following diagram:



The visualization of the HMM graph lattice for the Baum-Welch algorithm

The code will be as follows:

```
def frwrdBckwrldLattice: Double = {
    val _alpha = Alpha(lambda, obsSeq) //33
    val beta = Beta(lambda, obsSeq).getTreillis //34
    val alphas = _alpha.getTreillis
    gamma.update(alphas, beta) //35
    diGamma.update(alphas, beta, lambda.A, lambda.B, obsSeq)
    _alpha.alpha
}
```

The forward-backward algorithm uses the `Alpha` class for the computation/update of the `lambda` model in the forward pass (line 33) and the `Beta` class for the update of `lambda` in the backward pass (line 34). The joint probabilities-related `gamma` and `diGamma` matrices are updated at each recursion (line 35), reflecting the iteration of the mathematical expressions **M11** to **M14**.

The recursive computation of `maxLikelihood` exists if the algorithm converges (line 30). It throws an exception if the maximum number of recursions is exceeded (line 31).

Decoding – CF-3

This last canonical form consists of extracting the most likely sequence of states $\{q_t\}$ given a set of observations O_t and a λ model. Solving this problem requires, once again, a recursive algorithm.

The Viterbi algorithm

The extraction of the best state sequence (the sequence of a state that has the highest probability) is very time consuming. An alternative consists of applying a dynamic programming technique to find the best sequence $\{q_t\}$ through iteration. This algorithm is known as the **Viterbi algorithm**. Given a sequence of states $\{q_t\}$ and sequence of observations $\{o_t\}$, the probability $\delta_t(i)$ for any sequence to have the highest probability path for the first T observations is defined for the state S_i [7:7].

The Viterbi algorithm

M12: The definition of the delta function is as follows:

$$\delta_t(i) = \max_{q_j \in \{0, T-1\}} p(q_{0:T-1} = S_i, O_{0:T-1} | \lambda)$$

M13: Initialization of delta is defined as:



$$\delta_0(i) = \pi_i b_i(O_0) \quad \psi_0(i) = 0 \quad \forall i$$

M14: Recursive computation of delta is defined as:

$$\delta_t(j) = \max_i (\delta_{t-1}(i) \cdot a_{ij} \cdot b_j(O_t)) \quad \psi_t(j) = \arg \max_i (\delta_{t-1}(i) \cdot a_{ij})$$

M15: The computation of the optimum state sequence $\{q\}$ is defined as:

$$q^*_{t+1} = \psi_{t+1}(q^*_{t+1}) \quad q^*_T = \arg \max_i \delta_T(i)$$

The `ViterbiPath` class implements the Viterbi algorithm whose purpose is to compute the optimum sequence (or path) of states given a set of observations and a λ model. The optimum sequence or path of states is computed by maximizing the delta function.

The constructors for the `ViterbiPath` class have the same arguments as the forward, backward, and Baum-Welch algorithm: the `lambda` model and the set of observations `obsSeq`:

```
class ViterbiPath(lambda: HMMModel, obsSeq: Vector[Int]) {
    val nObs = lambda.numObs
    val nStates = lambda.numStates
    val psi = Array.fill(nObs)(Array.fill(nStates)(0)) //35
    val qStar = new QStar(nObs, nStates) //36

    val delta = { //37
```

```

Range(0, nStates) ./: (DMatrix(nObs, nStates)) ((m, n) => {
    psi(0)(n) = 0
    m += (0, n, lambda.pi(n) * lambda.B(n, obsSeq.head))
})
val path = HMMPrediction(viterbi(1), qStar()) //38
}

```

As seen in the preceding information box containing the mathematical expressions for the Viterbi algorithm, the following matrices have to be defined:

- `psi`: This is the matrix of indices of `nObs` observations by indices of `nStates` states (line 35).
- `qStar`: This is the optimum sequence of states at each recursion of the Viterbi algorithm (line 36).
- `delta`: This is the sequence that has the highest probability path for the first n observations. It also sets the `psi` values for the first observation to 0 (line 37).

All members of the `ViterbiPath` class are private except `path` that defines the optimum sequence or path of states given the `obsSeq` observations (line 38).

The matrix that defines the maximum probability `delta` of any sequence of states given the `lambda` model and the `obsSeq` observation is initialized using the mathematical expression **M13** (line 37). The predictive model returns the path or optimum sequence of states as an instance of `HMMPrediction`:

```
case class HMMPrediction(likelihood: Double, states: Array[Int])
```

The first argument of `likelihood` is computed by the `viterbi` recursive method. The indices of the states in the `states` optimum sequence is computed by the `QStar` class (line 38).

Let's take a look under the hood of the Viterbi recursive method:

```

@tailrec
def viterbi(t: Int): Double = {
    Range(0, numStates).foreach( updateMaxDelta(t, _) ) //39

    if( t == obsSeq.size-1 ) { //40
        val idxMaxDelta = Range(0, numStates)
            .map(i => (i, delta(t, i))).maxBy(_.value) //41
        qStar.update(t+1, idxMaxDelta._1) //42
        idxMaxDelta._2
    }
    else viterbi(t+1) //43
}

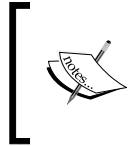
```

The recursion started on the second observation as the `qStar`, `psi`, and `delta` parameters have already been initialized in the constructor. The recursive implementation invokes the `updateMaxDelta` method to update the `psi` indexing matrix and the highest probability for any state, as follows:

```
def updateMaxDelta(t: Int, j: Int): Unit = {  
    val idxDelta = Range(0, nStates)  
        .map(i => (i, delta(t-1, i)*lambda.A(i, j)))  
        .maxBy(_.value) //44  
    psi(t)(j) = idxDelta._1  
    delta += (t, j, idxDelta._2) //45  
}
```

The `updateMaxDelta` method implements the mathematical expression **M14** that extracts the index of the state that maximizes `psi` (line 44). The `delta` probability matrix and the `psi` indexing matrix are updated accordingly (line 45).

The `viterbi` method is called recursively for the remaining observations except the last one (line 43). At the last observation of the `obsSeq.size-1` index, the algorithm executes the mathematical expression **M15**, which is implemented in the `QStar` class (line 42).



The QStar class

The `QStar` class and its `update` method use linear algebra and are not described here; refer to the documented source code and Scaladocs files for details.

This implementation of the decoding form of the hidden Markov model completes the description of the hidden Markov model and its implementation in Scala. Now, let's put this knowledge into practice.

Putting it all together

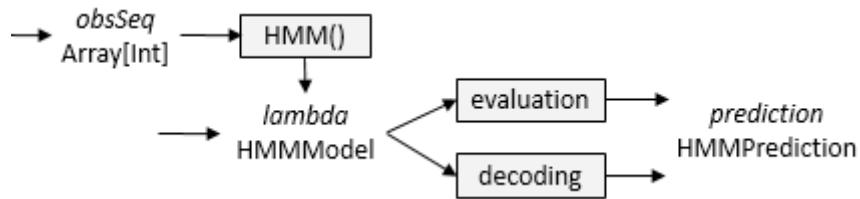
The main `HMM` class implements the three canonical forms. A view bound to an array of integers is used to parameterize the `HMM` class. We assume that a time series of continuous or pseudo-continuous values is quantized into discrete symbol values.

The `@specialized` annotation ensures that the byte code is generated for the `Array[Int]` primitive without executing the conversion implicitly declared by the bound view.

There are two modes that execute any of the three canonical forms of the hidden Markov model:

- The `ViterbiPath` class: The constructor initializes/trains a model similar to any other learning algorithm, as described in the *Design template for immutable classifiers* section of the *Appendix A, Basic Concepts*. The constructor generates the model by executing the Baum-Welch algorithm. Once the model is successfully created, it can be used for decoding or evaluation.
- The `ViterbiPath` object: The companion provides the `decode` and `evaluate` methods for the decoding and evaluation of the sequence of observations using HMM.

The two modes of operations are described in the following diagram:



The computational flow for the hidden Markov model

Let's complete our implementation of the HMM with the definition of its class. The `HMM` class is defined as a data transformation using a model implicitly generated from an `xt` training set, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 46):

```

class HMM[@specialized(Double) T <: AnyVal] (
  config: HMMConfig,
  xt: XVSeries[T],
  form: HMMForm)
  (implicit quantize: Array[T] => Int, f: T => Double)
  extends ITransform[Array[T]](xt) with Monitor[Double] { //46

  type V = HMPrediction //47
  val obsSeq: Vector[Int] = xt.map(quantize(_)) //48

  val model: Option[HMMModel] = train //49
  override def |> : PartialFunction[U, Try[V]] //50
}
  
```

The `HMM` constructor takes the following four arguments (line 46):

- `config`: This is the configuration of the HMM that is the dimension of `lambda` model and execution parameters
- `xt`: This is the multidimensional time series of observations whose features have the `T` type
- `form`: This is the canonical form to be used once the model is generated (evaluation or decoding)
- `quantize`: This is the quantization function that converts an observation of the `Array[T]` type to an `Int` type
- `f`: This is the implicit conversion from the `T` type to `Double`

The constructor has to override the `V` type (`HMMPrediction`) of the output data (line 47) declared in the `ITransform` abstract class. The structure of the `HMMPrediction` class has been defined in the previous section.

The `Monitor` trait is used to collect the profiling information during training (refer to the `Monitor` section under *Utility classes* in the *Appendix A, Basic Concepts*).

The time series of `xt` observations is converted into a vector of `obsSeq` observed states by applying the `quantize` quantization function to each observation (line 48).

As with any supervised learning technique, the model is created through training (line 49). Finally, the `|>` polymorphic predictor invokes either the `decode` method or the `evaluate` method (line 50).

The `train` method consists of the execution of the Baum-Welch algorithm and returns the `lambda` model:

```
def train: Option[HMMModel] = Try {
    BaumWelchEM(config, obsSeq).lambda }.toOption
```

Finally, the `|>` predictor is a simple wrapper to the evaluation form (`evaluate`) and the decoding form (`decode`):

```
override def |>: PartialFunction[U, Try[V]] = {
    case x: Array[T] if(isModel && x.length > 1) =>
        form match {
            case _: EVALUATION =>
                evaluation(model.get, Vector[Int](quantize(x)))
            case _: DECODING =>
                decoding(model.get, Vector[Int](quantize(x)))
        }
}
```

The protected `evaluation` method of the `HMM` companion object is a wrapper around the `Alpha` computation:

```
def evaluation(model: HMMModel,
  obsSeq: Vector[Int]): Try[HMPrediction] = Try {
  HMPrediction(-Alpha(model, obsSeq).logProb, obsSeq.toArray)
}
```

The `evaluate` method of the `HMM` object exposes the evaluation canonical form:

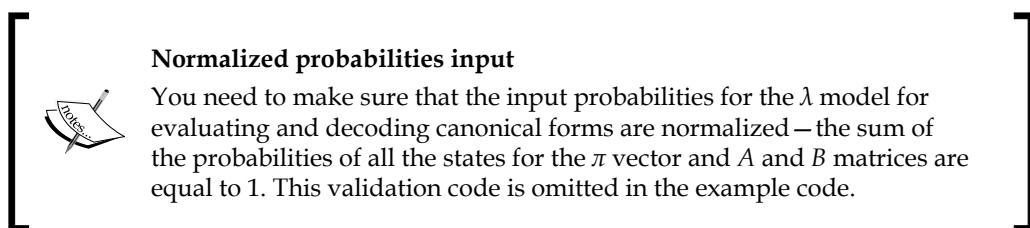
```
def evaluate[T <: AnyVal](model: HMMModel,
  xt: XVSeries[T])(implicit quantize: Array[T] => Int,
  f: T => Double): Option[HMPrediction] =
  evaluation(model, xt.map(quantize(_))).toOption
```

The decoding method wraps the Viterbi algorithm to extract the optimum sequence of states:

```
def decoding(model: HMMModel, obsSeq: Vector[Int]): Try[HMPrediction] = Try {
  ViterbiPath(model, obsSeq).path
}
```

The `decode` method of the `HMM` object exposes the decoding canonical form:

```
def decode[T <: AnyVal](model: HMMModel,
  xt: XVSeries[T])(implicit quantize: Array[T] => Int,
  f: T => Double): Option[HMPrediction] =
  decoding(model, xt.map(quantize(_))).toOption
```

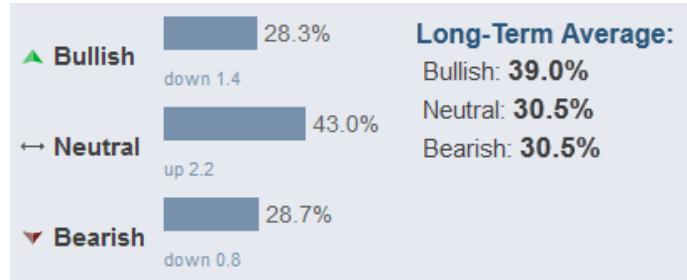


Test case 1 – training

Our first test case is to train an HMM to predict the sentiment of investors as measured by the weekly sentiment survey of the members of the **American Association of Individual Investors (AAII)** [7:8]. The goal is to compute the transition probabilities matrix A , the emission probabilities matrix B , and the steady state probability distribution π , given the observations and hidden states (training canonical forms).

We assume that the change in investor sentiments is independent of time, as required by the hidden Markov model.

The AAII sentiment survey grades the bullishness on the market in terms of percentage:



The weekly AAII market sentiment (reproduced by courtesy from AAII)

The sentiment of investors is known as a contrarian indicator of the future direction of the stock market. Refer to the *Terminology* section in the *Appendix A, Basic Concepts*.

Let's select the ratio of the percentage of investors that are bullish over the percentage of investors that are bearish. The ratio is then normalized.

The following table lists this:

Time	Bullish	Bearish	Neutral	Ratio	Normalized Ratio
t0	0.38	0.15	0.47	2.53	1.0
t1	0.41	0.25	0.34	1.68	0.53
t2	0.25	0.35	0.40	0.71	0.0
...

The sequence of nonnormalized observations (the ratio of bullish sentiments over bearish sentiments) is defined in a CSV file as follows:

```

val OBS_PATH = "resources/data/chap7/obsprob.csv"
val NUM_SYMBOLS = 6
val NUM_STATES = 5
val EPS = 1e-4
val MAX_ITERS = 150
val observations = Vector[Double] (
  0.01, 0.72, 0.78, 0.56, 0.61, 0.56, 0.45, ...
)

val quantize = (x: DblArray) =>
  (x.head * (NUM_STATES+1)).floor.toInt //51
val xt = observations.map(Array[Double](_))

```

```

val config = HMMConfig(zt.size, NUM_STATES, NUM_SYMBOLS,
    MAX_ITERS, EPS)
val hmm = HMM[Array[Int]](config, xt) //52
show(s"Training):\n${hmm.model.toString()}")

```

The constructor for the `HMM` class requires a `T => Array[Int]` implicit conversion, which is implemented by the `quantize` function (line 51). The `hmm.model` model is created by instantiating an `HMM` class with a predefined configuration and an `obsSeq` sequence of observed states (line 52).

The training of the HMM generates the following state transition probabilities matrix:

A	1	2	3	4	5
1	0.090	0.026	0.056	0.046	0.150
2	0.094	0.123	0.074	0.058	0.0
3	0.093	0.169	0.087	0.061	0.056
4	0.033	0.342	0.017	0.031	0.147
5	0.386	0.47	0.314	0.541	0.271

The emission matrix is as follows:

B	1	2	3	4	5	6
1	0.203	0.313	0.511	0.722	0.264	0.307
2	0.149	0.729	0.258	0.389	0.324	0.471
3	0.305	0.617	0.427	0.596	0.189	0.186
4	0.207	0.312	0.351	0.653	0.358	0.442
5	0.674	0.520	0.248	0.294	0.259	0.03

Test case 2 – evaluation

The objective of the evaluation is to compute the probability of the `xt` observed data given a λ model (A_0 , B_0 , and π_0):

```

val A0 = Array[Array[Double]](
    Array[Double](0.21, 0.13, 0.25, 0.06, 0.11, 0.24),
    Array[Double](0.31, 0.17, 0.18, 0.04, 0.19, 0.11),
    ...
)
val B0 = Array[Array[Double]](
    Array[Double](0.61, 0.39),
    Array[Double](0.54, 0.46),
    ...
)

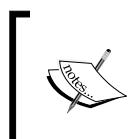
```

```
)  
val PI0 = Array[Double] (  
    0.26, 0.04, 0.11, 0.26, 0.19, 0.14)  
  
val xt = Vector[Double] (  
    0.0, 1.0, 21.0, 1.0, 30.0, 0.0, 1.0, 0.0, ...  
).map(Array[Double](_))  
val max = data.max  
val min = data.min  
implicit val quantize = (x: DblArray) =>  
    ((x.head/(max - min) + min)*(B0.head.length-1)).toInt //55  
val lambda = HMMModel(  
    DMatrix(A0), DMatrix(B0), PI0, xt.length) //53  
evaluation(lambda, xt).map(_.toString).map(show(_)) //54
```

The model is created directly by converting the A_0 state-transition probabilities and B_0 emission probabilities as matrices of the `DMatrix` type (line 53). The evaluation method generates an `HMPrediction` object, which is stringized, and then displays it in the standard output (line 54).

The quantization method consists of normalizing the input data over the number (or range) of symbols associated with the `lambda` model. The number of symbols is the size of the rows of the emission probabilities matrix B . In this case, the range of the input data is [0.0, 3.0]. The range is normalized using the linear transform $f(x) = x / (max - min) + min$, then adjusted for the number of symbols (or values for states) (line 55).

The `quantize` quantization function has to be explicitly defined before invoking the evaluation method.



Test case for decoding

Refer to the source code and the API documents for the test case related to the decoding form.

HMM as a filtering technique

The evaluation form of the hidden Markov model is very suitable for filtering data for discrete states. Contrary to time series filters such as the Kalman filter introduced in the *The discrete Kalman filter* section in *Chapter 3, Data Preprocessing*, the HMM requires data to be stationary in order to create a reliable model. However, the hidden Markov model overcomes some of the limitations of analytical time series analysis. Filters and smoothing techniques assume that the noise (frequency mean, variance, and covariance) is known and usually follows a Gaussian distribution.

The hidden Markov model does not have such a restriction. Filtering techniques, such as moving averaging techniques, discrete Fourier transforms, and Kalman filters apply to both discrete and continuous states while the HMM does not. Moreover, the extended Kalman filter can estimate nonlinear states.

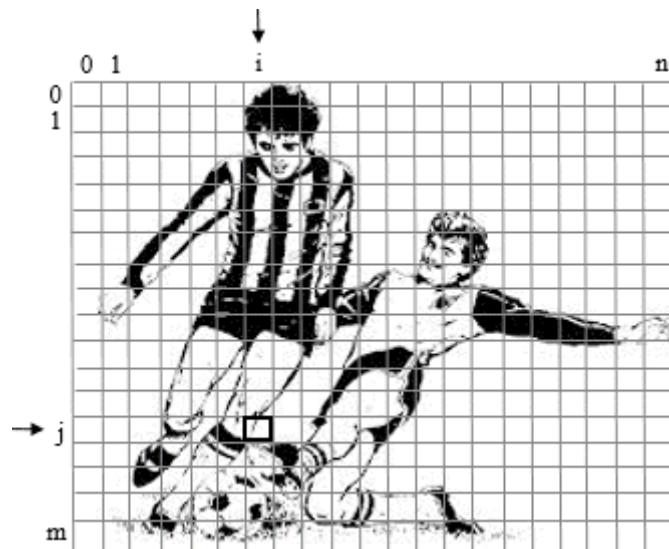
Conditional random fields

The **conditional random field (CRF)** is a discriminative machine learning algorithm introduced by John Lafferty, Andrew McCallum, and Fernando Pereira [7:9] at the turn of the century as an alternative to the HMM. The algorithm was originally developed to assign labels to a set of observation sequences.

Let's consider a concrete example to understand the conditional relation between the observations and the label data.

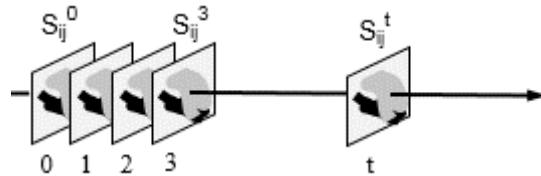
Introduction to CRF

Let's consider the problem of detecting a foul during a soccer game using a combination of video and audio. The objective is to assist the referee and analyze the behavior of the players to determine whether an action on the field is dangerous (red card), inappropriate (yellow card), in doubt to be replayed, or legitimate. The following image is an example of the segmentation of a video frame for image processing:



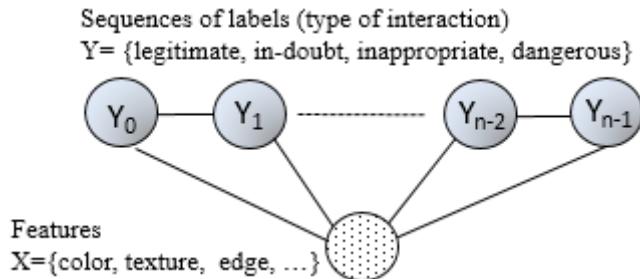
An example of an image processing problem requiring machine learning

The analysis of the video consists of segmenting each video frame and extracting image features such as colors or edges [7:10]. A simple segmentation scheme consists of breaking down each video frame into tiles or groups of pixels indexed by their coordinates on the screen. A time sequence is then created for each tile S_{ij} as represented in the following image:



The learning strategy for pixels in a sequence of video frames

The image segment S_{ij} is one of the labels that is associated with multiple observations. The same features extraction process applies to the audio associated with the video. The relation between the video/image segment and the hidden state of the altercation between the soccer players is illustrated in the following model graph:

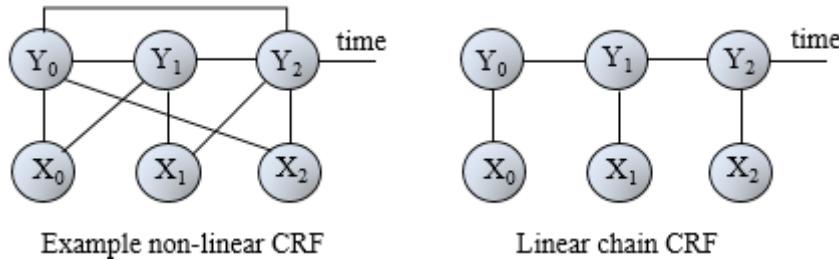


The undirected graph representation of CRF for the soccer infraction detection

CRFs are discriminative models that can be regarded as a structured output extension of the logistic regression. CRFs address the problem of labeling a sequence of data, such as assigning a tag to each word in a sentence. The objective is to estimate the correlation among the output (observed) values Y that are conditional on the input values (features) X .

The correlation between the output and input values is described as a graph (also known as a **graph-structured CRF**). A good example of graph-structured CRFs are cliques. Cliques are sets of connected nodes in a graph for which each vertex has an edge connecting it to every other vertex in the clique.

Such models are complex and their implementation is challenging. Most real-world problems related to time series or ordered sequences of data can be solved as a correlation between a linear sequence of observations and a linear sequence of input data, which is similar to the HMM. Such a model is known as the **linear chain structured CRF** or **linear chain CRF** for short:



An illustration of a nonlinear and linear chain CRF

One main advantage of the linear chain CRF is that the maximum likelihood $p(Y|X, w)$ can be estimated using dynamic programming techniques such as the Viterbi algorithm used in the HMM. From now on, the section focuses exclusively on the linear chain CRF in order to stay consistent with the HMM, as described in the previous section.

Linear chain CRF

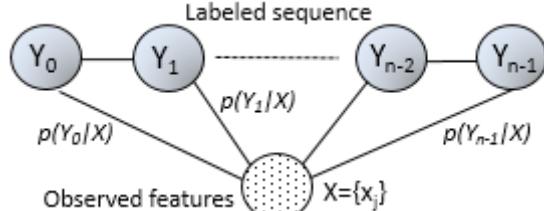
Let's consider a random variable $X=\{x_i\}_{0:n-1}$ representing n observations and a random variable Y representing a corresponding sequence of labels $Y=\{y_j\}_{0:m-1}$. The hidden Markov model estimates the joint probability $p(X, Y)$, as any generative model requires the enumeration of all the sequences of observations.

If each element of Y, y_j obeys the first order of the Markov property, then (Y, X) is a CRF. The likelihood is defined as a conditional probability $p(Y|X, w)$, where w is the model parameters vector.

 **Observation dependencies**
The purpose of CRF models is to estimate the maximum likelihood of $p(Y|X, w)$. Therefore, independence between X observations is not required.

A graphical model is a probabilistic model for which a graph denotes the conditional independence between random variables (vertices). The conditional and joint probabilities of random variables are represented as edges. The graph for generic conditional random fields can indeed be complex. The most common and simplistic graph is the linear chain CRF.

A first order linear chain conditional random field can be visualized as an undirected graphical model, which illustrates the conditional probability of a label Y_j given a set of observations X :



A linear, conditional, random field undirected graph

The Markov property simplifies the conditional probabilities of Y , given X , by considering only the neighbor labels $p(Y_1 | X, Y_j, j \neq 1) = p(Y_1 | X, Y_0, Y_2)$ and $p(Y_i | X, Y_j, j \neq i) = p(Y_i | X, Y_{i-1}, Y_{i+1})$.

The conditional random fields introduce a new set of entities and a new terminology:

- **Potential functions (f):** These strictly positive and real value functions represent a set of constraints on the configurations of random variables. They do not have any obvious probabilistic interpretation.
- **Identity potential functions:** These are potential functions $I(x, t)$ that take 1 if the condition on the feature x at time t is true, and 0 otherwise.
- **Transition feature functions:** Simply known as feature functions, t_i , are potential functions that take a sequence of features $\{X_j\}$, the previous label Y_{i-1} , the current label Y_i , and an index i . The transition feature function outputs a real value function. In a text analysis, a transition feature function would be defined by a sentence as a sequence of observed features, the previous word, the current word, and a position of a word in a sentence. Each transition feature function is assigned a weight that is similar to the weights or parameters in the logistic regression. Transition feature functions play a similar role to the state transition factors a_{ij} in the HMM but without a direct probabilistic interpretation.
- **State feature functions (s):** These are potential functions that take the sequence of features $\{X_j\}$, the current label Y_i , and the index i . They play a similar role to the emission factors in the HMM.

A CRF defines the log probability of a particular label sequence Y , given a sequence of observations X as the normalized product of the transition feature and state feature functions. In other words, the likelihood of a particular sequence Y , given the observed features X , is a logistic regression.

The mathematical notation to compute the conditional probabilities in the case of a first order linear chain CRF is described in the following information box:

The CRF conditional distribution

M1: The log probability of a label's sequence y , given an observation x is defined as:

$$\log f_i(y_{i-1}, y_i, x, i) = w_c + \sum_{i=0}^{K-1} w_i t_i(y_{i-1}, y_i, x, i) + \sum_{j=0}^{K-1} \mu_j s_j(y_i, x, i)$$

M2: Transition feature functions are defined as:

$$t_i(y_{i-1}, y_i, x, i) = I(y_{i-1} = l_1).I(y_i = l_2).I(x = 0)$$

 M3: Using the notation:

$$F_i(y, x) = \sum_{j=0}^{K-1} f_j(y_{j-1}, y_j, x, i) \quad \log p(y|x, \lambda) \propto \sum_{j=0}^{K-1} w_j F_j(x, y)$$

M4: The conditional distribution of labels y , given x , using the Markov property is defined as:

$$p(y|x, w) = \frac{1}{Z(x)} e^{\sum_{j=0}^{K-1} w_j F_j(x, y)} \quad z(x) = \sum_{i=0}^{N-1} \sum_{j=0}^{K-1} w_j F_j(x, y)$$

The weights w_j are sometimes referred as λ in scientific papers, which may confuse the reader; w is used to avoid any confusion with the λ regularization factor.

Now, let's get acquainted with the conditional random fields algorithm and its implementation by Sunita Sarawagi.

Regularized CRFs and text analytics

Most of the examples used to demonstrate the capabilities of conditional random fields are related to text mining, intrusion detection, or bioinformatics. Although these applications have a great commercial merit, they are not suitable for introductory test cases because they usually require a lengthy description of the model and the training process.

The feature functions model

For our example, we will select a simple problem: how to collect and aggregate an analyst's recommendation on any given stock from different sources with different formats.

Analysts at brokerage firms and investment funds routinely publish the list of recommendations or ratings for any stock. These analysts use different rating schemes from buy/hold/sell, A/B/C rating, and stars rating, to market perform/neutral/market underperform rating. For this example, the rating is normalized as follows:

- 0 for a strong sell (F or 1 star rating)
- 1 for sell (D, 2 stars, or market underperform)
- 2 for neutral (C, hold, 3 stars, market perform, and so on)
- 3 for buy (B, 4 stars, market overperform, and so on)
- 4 for strong buy (A, 5 stars, highly recommended, and so on)

Here are examples of recommendations by stock analysts:

- *Macquarie upgraded AUY from Neutral to Outperform rating*
- *Raymond James initiates Ainsworth Lumber as Outperform*
- *BMO Capital Markets upgrades Bear Creek Mining to Outperform*
- *Goldman Sachs adds IBM to its conviction list*

The objective is to extract the name of the financial institution that publishes the recommendation or rating, the stock rated, the previous rating, if available, and the new rating. The output can be inserted into a database for further trend analysis, prediction, or simply the creation of reports.

The scope of the application

Ratings from analysts are updated every day through different protocols (feed, e-mails, blogs, web pages, and so on). The data has to be extracted from HTML, JSON, plain text, or XML format before being processed. In this exercise, we assume that the input has already been converted into plain text (ASCII) using a regular expression or another classifier.

The first step is to define the labels Y representing the categories or semantics of the rating. A segment or sequence is defined as a recommendation sentence. After reviewing the different recommendations, we are able to specify the following seven labels:

- Source of the recommendation (Goldman Sachs and so on)
- Action (upgrades, initiates, and so on)
- Stock (either the company name or the stock ticker symbol)
- From (an optional keyword)
- Rating (an optional previous rating)
- To
- Rating (the new rating for the stock)

The training set is generated from the raw data by *tagging* the different components of the recommendation. The first (or initial) rating for a stock does not have the labels 4 and 5 from the preceding list defined.

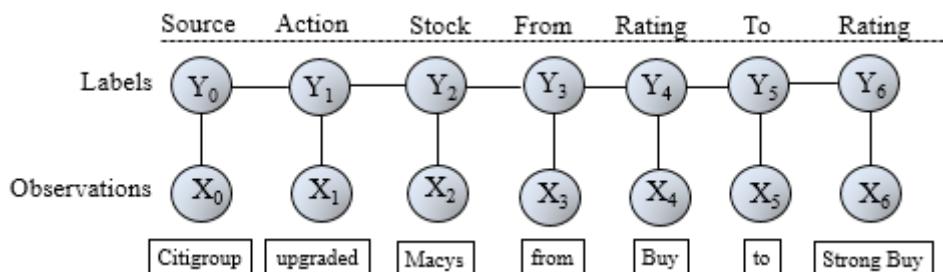
Consider the following example:

```
Citigroup // Y(0) = 1
upgraded // Y(1)
Macys // Y(2)
from // Y(3)
Buy // Y(4)
to // Y(5)
Strong Buy //Y(6) = 7
```

Tagging

Tagging as a word may have a different meaning depending on the context. In **natural language processing (NLP)**, tagging refers to the process of assigning an attribute (an adjective, pronoun, verb, proper name, and so on) to a word in a sentence [7:11].

A training sequence can be visualized in the following undirected graph:



An example of a recommendation as a CRF training sequence

You may wonder why we need to tag the *From* and *To* labels in the creation of the training set. The reason is that these keywords may not always be stated and/or their positions in the recommendation differ from one source to another.

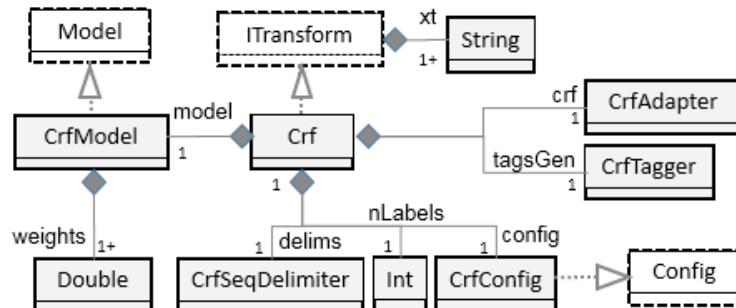
Design

The implementation of the conditional random fields follows the design template for classifiers, which is described in the *Design template for immutable classifiers* section under *Source code considerations* in the *Appendix A, Basic Concepts*.

Its key components are as follows:

- A `CrfModel` model of the `Model` type is initialized through training during the instantiation of the classifier. A model is an array of `weights`.
- The predictive or classification routine is implemented as an implicit data transformation of the `ITransform` type.
- The `Crf` conditional random field classifier has four parameters: the number of labels (or number of features), `nLabels`, configuration of the `CrfConfig` type, the sequence of delimiters of the `CrfSeqDelimiter` type, and a vector of name of files `xt` that contains the tagged observations.
- The `CrfAdapter` class interfaces with the IITB CRF library.
- The `CrfTagger` class extracts features from the tagged files.

The key software components of the conditional random fields are described in the following UML class diagram:



The UML class diagram for the conditional random fields

The UML diagram omits the utility traits and classes such as `Monitor` or the Apache Commons Math components.

Implementation

The test case uses the IIT-B's CRF Java implementation from the Indian Institute of Technology Bombay by Sunita Sarawagi. The JAR files can be downloaded from SourceForge (<http://sourceforge.net/projects/crf/>).

The library is available as JAR files and source code. Some of the functionalities, such as the selection of a training algorithm, is not available through the API. The components (JAR files) of the library are as follows:

- A CRF for the implementation of the CRF algorithm
- LBFGS for limited-memory Broyden-Fletcher-Goldfarb-Shanno nonlinear optimization of convex functions (used in training)
- The CERN Colt library for the manipulation of a matrix
- The GNU generic hash container for indexing

Configuring the CRF classifier

Let's take a look at the `Crf` class that implements the conditional random fields classifier. The `Crf` class is defined as a data transformation of the `ITransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 2):

```
class Crf(nLabels: Int, config: CrfConfig,
         delims: CrfSeqDelimiter, xt: Vector[String])//1
  extends ITransform[String](xt) with Monitor[Double]//2

  type V = Double //3
  val tagsGen = new CrfTagger(nLabels) //4
  val crf = CrfAdapter(nLabels, tagsGen, config.params) //5
  val model: Option[CrfModel] = train //6
  weights: Option[DblArray] = model.map(_.weights)

  override def |> : PartialFunction[String, Try[V]] //7
}
```

The constructor for `Crf` has the following four arguments (line 1):

- `nLabels`: These are the number of labels used for the classification
- `config`: This is the configuration parameter used to train `crf`
- `delims`: These are the delimiters used in raw and tagged files
- `xt`: This is a vector of name of files that contains raw and tagged data

 **Filenames for raw and tagged data**

For the sake of simplicity, the files for the raw observations and the tagged observations have the same name with different extensions: `filename.raw` and `filename.tagged`.

The `Monitor` trait is used to collect the profiling information during training (refer to the `Monitor` section under *Utility classes* in the *Appendix A, Basic Concepts*).

The `v` type of the output of the `|>` predictor is defined as `Double` (line 3).

The execution of the CRF algorithm is controlled by a wide variety of configuration parameters encapsulated in the `CrfConfig` configuration class:

```
class CrfConfig(w0: Double, maxIters: Int, lambda: Double,  
    eps: Double) extends Config { //8  
    val params = s"""initValue $w0 maxIters $maxIters  
        | lambda $lambda scale true eps $eps""".stripMargin  
}
```

For the sake of simplicity, we use the default `CrfConfig` configuration parameters to control the execution of the learning algorithm, with the exception of the following four variables (line 8):

- Initialization of the `w0` weight that uses either a predefined or random value between 0 and 1 (default 0)
- The maximum number of iterations, `maxIters`, that is used in the computation of the weights during the learning phase (default 50)
- The `lambda` scaling factor for the L_2 penalty function that is used to reduce observations with a high value (default 1.0)

- The `eps` convergence criteria that is used to compute the optimum values for the `wj` weights (default 1e-4)

 **The L_2 regularization**

This implementation of the conditional random fields support the L_2 regularization, as described in the *Regularization* section in *Chapter 6, Regression and Regularization*. The regularization is turned off by setting $\lambda = 0$.

The `CrfSeqDelimiter` case class specifies the following regular expressions:

- `obsDelim` to parse each observation in the raw files
- `labelsDelim` to parse each labeled record in the tagged files
- `seqDelim` to extract records from raw and tagged files

The code will be as follows:

```
case class CrfSeqDelimiter(obsDelim: String,
                            labelsDelim: String, seqDelim: String)
```

The `DEFAULT_SEQ_DELIMITER` instance is the default sequence delimiter used in this implementation:

```
val DEFAULT_SEQ_DELIMITER =
  new CrfSeqDelimiter(",\t/ -():.;'?"`&_, "//", "\n")
```

The `CrfTagger` tag or label generator iterates through the tagged file and applies the relevant regular expressions of `CrfSeqDelimiter` to extract the symbols used in training (line 4).

The `CrfAdapter` object defines the different interfaces to the IITB CRF library (line 5). The factory for CRF instances is implemented by the `apply` constructor as follows:

```
object CrfAdapter {
  import iitb.CRF.CRF
  def apply(nLabels: Int, tagger: CrfTagger,
            config: String): CRF = new CRF(nLabels, tagger, config)
  ...
}
```

Adapter classes to the IITB CRF library

The training of the conditional random field for sequences requires to define a few key interfaces:

- `DataSequence` to specify the mechanism to access observations and labels for training and testing data
- `DataIter` to iterate through the sequence of data created using the `DataSequence` interface
- `FeatureGenerator` to aggregate all the feature types

 These interfaces have default implementations bundled in the CRF Java library [7:12]. Each of these interfaces have to be implemented as adapter classes:

```
class CrfTagger(nLabels: Int) extends FeatureGenerator
class CrfDataSeq(nLabels: Int, tags: Vector[String],
delim: String) extends DataSequence
class CrfSeqIter(nLabels: Int, input: String, delim:
CrfSeqDelimiter) extends DataIter
```

Refer to the documented source code and Scaladocs files for the description and implementation of these adapter classes.

Training the CRF model

The objective of the training is to compute the weights w_j that maximize the conditional log-likelihood function without the L_2 penalty function. Maximizing the log-likelihood function is equivalent to minimizing the loss with the L_2 penalty. The function is convex, and therefore, any variant gradient descent (greedy) algorithm can be applied iteratively.

 M5: The conditional log-likelihood for a linear chain CRF training set $D = \{x_i, y_i\}$ is given as follows:

$$\mathcal{L}(w, D) = - \sum_{i=0}^{n-1} \log p(y_i | x_i, w)$$

M6: Maximization of the loss function and L2 penalty is given as follows:

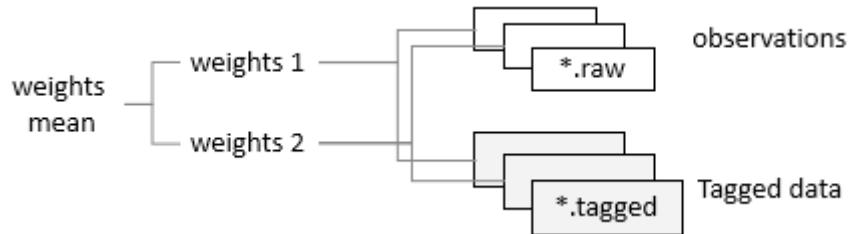
$$w^* = \arg \max_{\lambda} [\mathcal{L}(w, D) + \lambda \|w\|^2] \quad \lambda = \frac{1}{2\sigma^2}$$

The training file consists of a pair of files:

- **Raw datasets:** Recommendations (such as *Raymond James upgrades Gentiva Health Services from Underperform to Market perform*)
- **Tagged datasets:** Tagged recommendations (such as *Raymond James [1] upgrades [2] Gentiva Health Services [3], from [4] Underperform [5] to [6] Market perform [7]*)

 **Tags type**
In this implementation, the tags have the `Int` type. However, alternative types, such as enumerators or even continuous values (that is, `Double`) can be used.

The training or computation of weights can be quite expensive. It is highly recommended that you distribute the observations and tagged observations dataset across multiple files, so they can be processed concurrently:



The distribution of the computation of the weights of the CRF

The `train` method creates the model by computing the weights of the CRF. It is invoked by the constructor of `Crf`:

```

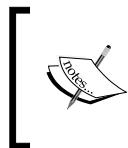
def train: Option[CrfModel] = Try {
  val weights = if(xt.size == 1) //9
    computeWeights(xt.head)
  else {
    val weightsSeries = xt.map( computeWeights(_) )
    statistics(weightsSeries).map(_.mean).toArray //10
  }
  new CrfModel(weights) //11
}._toOption("Crf training failed", logger)
  
```

We cannot assume that there is only one tagged dataset (that is, a single pair of `*.raw` and `*.tagged` files) (line 9). The `computeWeights` method used for computation of weights for the CRF is applied to the first dataset if there is only one pair of raw and tagged file. In the case of multiple datasets, the `train` method computes the mean of all the weights extracted from each tagged dataset (line 10). The mean of the weights are computed using the `statistics` method of the `XTSeries` object, which was introduced in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*. The `train` method returns `CrfModel` if successful, and `None` otherwise (line 11).

For efficiency purpose, the map should be parallelized using the `ParVector` class as follows:

```
val parXt = xt.par
val pool = new ForkJoinPool(nTasks)
v.tasksupport = new ForkJoinTaskSupport(pool)
parXt.map(computeWeights(_))
```

The parallel collections are described in detail in the *Parallel collections* section under *Scala* in *Chapter 12, Scalable Frameworks*.



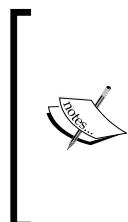
CRF weights computation

It is assumed that input tagged files share the same list of tags or symbols, so each dataset produces the same array of weights.



The `computeWeights` method extracts the weights from each pair of observations and tagged observation files. It invokes the `train` method of the `CrfTagger` tag generator (line 12) to prepare, normalize, and set up the training set, and then invokes the training procedure on the IITB CRF class (line 13):

```
def computeWeights(tagsFile: String): DblArray = {
    val seqIter = CrfSeqIter(nLabels, tagsFile, delims)
    tagsGen.train(seqIter) //12
    crf.train(seqIter) //13
}
```



The scope of the IITB CRF Java library evaluation

The CRF library has been evaluated with three simple text analytics test cases. Although the library is certainly robust enough to illustrate the internal workings of the CRF, I cannot vouch for its scalability or applicability in other fields of interests, such as bioinformatics or process control.



Applying the CRF model

The predictive method implements the `| >` data transformation operator. It takes a new observation (the analyst's recommendation on a stock) and returns the maximum likelihood, as shown here:

```
override def |> : PartialFunction[String, Try[V]] = {
  case obs: String if( !obs.isEmpty && isModel) => {
    val dataSeq = new CrfDataSeq(nLabels,obs,delims.obsDelim)
    Try (crf.apply(dataSeq)) //14
  }
}
```

The `| >` method merely creates a `dataSeq` data sequence and invokes the constructor of the IITB `CRF` class (line 14). The condition on the `obs` input argument to the partial function is rather rudimentary. A more elaborate condition of the observation should be implemented using a regular expression. The code to validate the arguments/parameters of the class and methods are omitted along with the exception handler for the sake of readability.

An advanced CRF configuration

 The CRF model of the **IITB** library is highly configurable. It allows developers to specify a state-label undirected graph with any combination of flat and nested dependencies between states. The source code includes several training algorithms such as the exponential gradient.

Tests

The client code to execute the test consists of defining the number of labels, `NLABELS` (that is, the number of tags for recommendation), the `LAMBDA` L2 penalty factor, the maximum of iterations, `MAX_ITERS`, allowed in the minimization of the loss function, and the `EPS` convergence criteria:

```
val LAMBDA = 0.5
val NLABELS = 9
val MAX_ITERS = 100
val W0 = 0.7
val EPS = 1e-3
val PATH = "resources/data/chap7/rating"
val OBS_DELIM = ",\t/ -():.;'?#`^&_"
```

```

val config = CrfConfig(W0 , MAX_ITERS, LAMBDA, EPS) //15
val delims = CrfSeqDelimiter(DELIM,"//","\n") //16
val crf = Crf(NLABELS, config, delims, PATH) //17
crf.weights.map( display(_) )

```

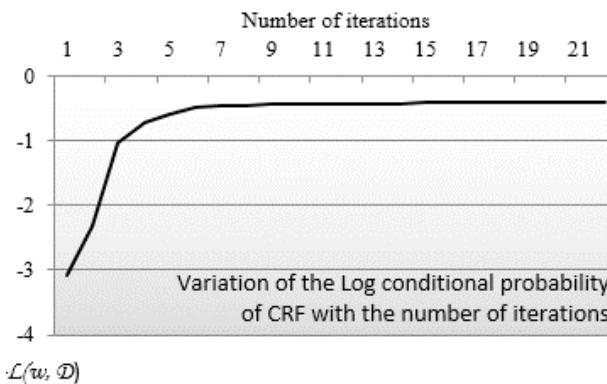
The three simple steps are as follows:

1. Instantiate the `config` configuration for the CRF (line 15)
2. Define the three `delims` delimiters to extract the tagged data (line 16)
3. Instantiate and train the CRF classifier, `crf` (line 17)

For these tests, the initial value of the weights (with respect to the maximum number of iterations for the maximization of the log likelihood and the convergence criteria) are set to 0.7 (with respect to 100 and 1e-3). The delimiters for labels sequence, observed features sequence, and the training set are customized for the format of `rating.raw` and `rating.tagged` input data files.

The training convergence profile

The first training run discovered 136 features from 34 analysts' stock recommendations. The algorithm converged after 21 iterations. The value of the log of the likelihood for each of those iterations is plotted to illustrate the convergence toward a solution of optimum w :



The visualization of the log conditional probability of a CRF during training

The training phase converges quickly toward a solution. It can be explained by the fact that there is little variation in the six-field format of the analyst's recommendations. A loose or free-style format would require a larger number of iterations during training to converge.

Impact of the size of the training set

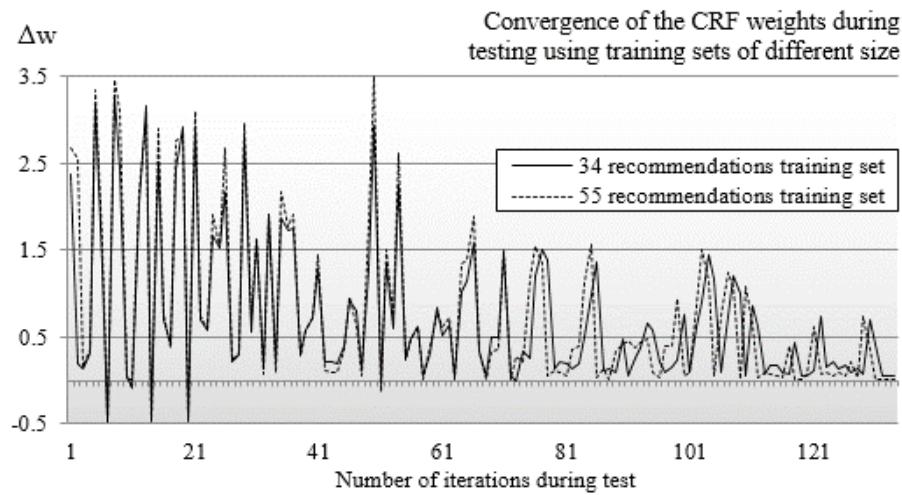
The second test evaluates the impact of the size of the training set on the convergence of the training algorithm. It consists of computing the difference Δw of the model parameters (weights) between two consecutive iterations $\{w_i\}_{t+1}$ and $\{w_i\}_t$:

$$\Delta w = \sum_{i=0}^{D-1} (w_i^{t+1} - w_i^t)$$

The test is run on 163 randomly chosen recommendations using the same model but with two different training sets:

- 34 analysts' stock recommendations
- 55 stock recommendations

The larger training set is a superset of the 34 recommendations' set. The following graph illustrates the comparison of features generated with 34 and 55 CRF training sequences:

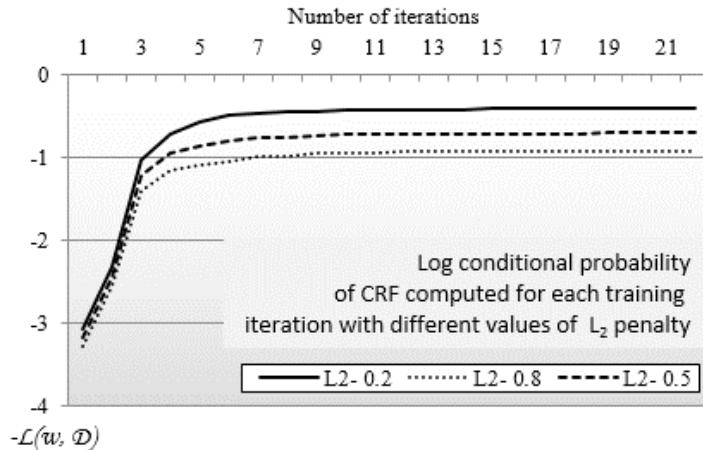


The convergence of the CRF weight using training sets of different sizes

The disparity between the test runs using two different sizes of training sets is very small. This can be easily explained by the fact that there is a small variation in the format between the analyst's recommendations.

Impact of the L_2 regularization factor

The third test evaluates the impact of the L_2 regularization penalty on the convergence toward the optimum weights/features. The test is similar to the first test with a different value of λ . The following chart plots $\log [p(Y|X, w)]$ for different values of $\lambda = 1/\sigma^2$ (0.2, 0.5, and 0.8):



The impact of the L_2 penalty on the convergence of the CRF training algorithm

The log of the conditional probability decreases or the conditional probability increases with the number of iterations. The lower the L_2 regularization factor, the higher the conditional probability.

The variation of the analysts' recommendations within the training set is small, which limits the risk of overfitting. A free-style recommendation format would have been more sensitive to overfitting.

Comparing CRF and HMM

The cost/benefit analysis of discriminative models relative to generative models applies to the comparison of the conditional random field with the hidden Markov model.

Contrary to the hidden Markov model, the conditional random field does not require the observations to be independent (conditional probability). The conditional random field can be regarded as a generalization of the HMM by extending the transition probabilities to arbitrary feature functions that can depend on the input sequence. The HMM assumes the transition probabilities matrix to be constant.

The HMM learns the transition probabilities a_{ij} on its own by processing more training data. The HMM can be regarded as a special case of CRF where the probabilities used in the state transition are constant.

Performance consideration

The time complexity for decoding and evaluating canonical forms of the hidden Markov model for N states and T observations is $O(N_2 T)$. The training of the HMM using the Baum-Welch algorithm is $O(N_2 TM)$, where M is the number of iterations.

There are several options to improve the performance of the HMM:

- Avoid unnecessary multiplication by 0 in the emission probabilities matrix by either using sparse matrices or tracking the null entries.
- Train the HMM on the most *relevant* subset of the training data. This technique can be particularly effective in the case of tagging of words or a bag of words in natural language processing.

The training of the linear chain conditional random fields is implemented using the same dynamic programming techniques as the HMM implementation (Viterbi, forward-backward passes, and so on). Its time complexity for training T data sequences, N labels (or expected outcomes), and M weights/features λ is $O(MTN_2)$.

The time complexity of the training of a CRF can be reduced by distributing the computation of the log likelihood and gradient over multiple nodes using a framework such as Akka or Apache Spark, as described in *Chapter 12, Scalable Frameworks* [7:13].

Summary

In this chapter, we had a closer look at modeling sequences of observations with hidden (or latent) states with the two commonly used algorithms:

- The generative hidden Markov model to maximize $p(X, Y)$
- The discriminative conditional random field to maximize $\log p(Y | X)$

The HMM is a special form of Bayes network. It requires the observations to be independent. Although restrictive, the conditional independence prerequisites make the HMM fairly easy to understand and validate, which is not the case for a CRF.

You learned how to implement three dynamic programming techniques: Viterbi, Baum-Welch, and alpha/beta algorithms in Scala. These algorithms are used to solve diverse type of optimization problems. They should be an essential component of your algorithmic tool box.

The conditional random field relies on the logistic regression to estimate the optimal weights of the model. Such a technique is also used in the multiple layer perceptron, which was introduced in *Chapter 9, Artificial Neural Network*. The next chapter introduces two important alternatives to the logistic regression for discriminating between observations: the Kernel function for nonlinear models and the maximization of the margin between classes of observations.

8

Kernel Models and Support Vector Machines

This chapter introduces kernel functions, binary support vectors classifiers, one-class support vector machines for anomaly detection, and support vector regression.

In the *Binomial classification* section in *Chapter 6, Regression and Regularization*, you learned the concept of hyperplanes to segregate observations from the training set and estimate the linear decision boundary. The logistic regression has at least one limitation: it requires that the datasets be linearly separated using a defined function (sigmoid). This limitation is especially an issue for high-dimension problems (large number of features that are highly nonlinearly dependent). **Support vector machines (SVMs)** overcome this limitation by estimating the optimal separating hyperplane using kernel functions.

In this chapter, we will cover the following topics:

- The impact of some of the SVM configuration parameters and the kernel method on the accuracy of the classification
- How to apply the binary support vector classifier to estimate the risk for a public company to curtail or eliminate its dividend
- How to detect outliers with a one-class support vector classifier
- How the support vector regression is compared to the linear regression

Support vector machines are formulated as a convex optimization problem. The mathematical foundation of the related algorithms is described in this chapter, for reference.

Kernel functions

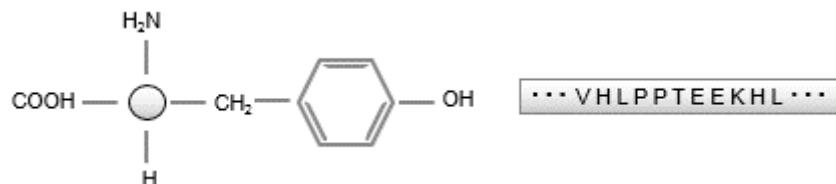
Every machine learning model introduced in this book so far assumes that observations are represented by a feature vector of a fixed size. However, some real-world applications such as text mining or genomics do not lend themselves to this restriction. The critical element of the process of classification is to define a similarity or distance between two observations. Kernel functions allow developers to compute the similarity between observations without the need to encode them in feature vectors [8:1].

An overview

The concept of kernel methods may be a bit odd at first to a novice. Let's consider the example of the classification of proteins. Proteins have different lengths and compositions, but they do not prevent scientists from classifying them [8:2].

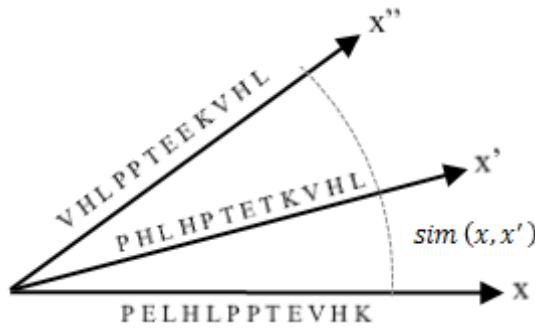
 **Proteins**
Proteins are polymers of amino acids joined together by peptide bonds. They are composed of a carbon atom bonded to a hydrogen atom, another amino acid, or a carboxyl group.

A protein is represented using a traditional molecular notation to which biochemists are familiar. Geneticists describe proteins in terms of a sequence of characters known as the **protein sequence annotation**. The sequence annotation encodes the structure and composition of the protein. The following image illustrates the molecular (left) and encoded (right) representation of a protein:



The sequence annotation of a protein

The classification and the clustering of a set of proteins require the definition of a similarity factor or distance used to evaluate and compare the proteins. For example, the similarity between three proteins can be defined as a normalized dot product of their sequence annotation:



The similarity between the sequence annotations of three proteins

You do not have to represent the entire sequence annotation of the proteins as a feature vector in order to establish that they belong to the same class. You only need to compare each element of each sequence, one by one, and compute the similarity. For the same reason, the estimation of the similarity does not require the two proteins to have the same length.

In this example, we do not have to assign a numerical value to each element of the annotation. Let's consider an element of the protein annotation as its character c and position p (for example, K, 4). The dot product of the two protein annotations x and x' of the respective lengths n and n' are defined as the number of identical elements (character and position) between the two annotations divided by the maximum length between the two annotations (**M1**):

$$\text{sim}(x_{cp}, x'_{c'p'}) = \frac{1}{\max(n, n')} \sum_{i=1}^{\min(n, n')} (c = c') \cap (p = p') \quad \max(n, n') = \max(n, n')$$

The computation of the similarity for the three proteins produces the result as $\text{sim}(x, x') = 6/12 = 0.50$, $\text{sim}(x, x'') = 3/13 = 0.23$, and $\text{sim}(x', x'') = 4/13 = 0.31$.

Another similar aspect is that the similarity of two identical annotations is 1.0 and the similarity of two completely different annotations is 0.0.

The visualization of similarity

It is usually more convenient to use a radial representation to visualize the similarity between features, as in the example of proteins' annotations. The distance $d(x, x') = 1/\text{sim}(x, x')$ is visualized as the angle or cosine between two features. The cosine metric is commonly used in text mining.

In this example, the similarity is known as a kernel function in the space of the sequence annotation of proteins.

Common discriminative kernels

Although the measure of similarity is very useful to understand the concept of a kernel function, kernels have a broader definition. A kernel $K(x, x')$ is a symmetric, nonnegative real function that takes two real arguments (values of two features). There are many different types of kernel functions, among which the most common are as follows:

- **The linear kernel (dot product):** This is useful in the case of very high-dimensional data where problems can be expressed as a linear combination of the original features.
- **The polynomial kernel:** This extends the linear kernel for a combination of features that are not completely linear.
- **The radial basis function (RBF):** This is the most commonly applied kernel. It is used where the labeled or target data is noisy and requires some level of regularization.
- **The sigmoid kernel:** This is used in conjunction with neural networks.
- **The Laplacian kernel:** This is a variant of RBF with a higher regularization impact on training data.
- **The log kernel:** This is used in image processing.

The RBF terminology

In this presentation and the library used in its implementation, the radial basis function is a synonym to the Gaussian kernel function. However, RBF also refers to the family of exponential kernel functions that encompasses Gaussian, Laplacian, and exponential functions.

The simple linear model for regression consists of the dot product of the regression parameters (weights) and the input data (refer to the *Ordinary least squares regression* section in *Chapter 6, Regression and Regularization*).

The model is, in fact, the linear combination of weights and linear combination of inputs. The concept can be extended by defining a general regression model as the linear combination of nonlinear functions, known as basis functions (**M2**):

$$f(x|w) = w_0 + \sum_{d=1}^D w_d \phi_d(x) \quad \phi_d: \mathbb{R} \rightarrow \mathbb{R}$$

The most commonly used basis functions are the power and Gaussian functions. The kernel function is described as the dot product of the two vectors of the basis function $\varphi(x) \cdot \varphi(x')$ of two features vectors x and x' . A partial list of kernel methods is as follows:

M3: The generic kernel function is defined as:

$$K(x, x') = \phi(x) \cdot \phi(x') = \sum_{d=1}^D \phi_d(x) \phi_d(x')$$

M4: The linear kernel is defined as:

$$K(x, x') = x^T x' = \sum_{d=1}^D x_i \cdot x'_i$$

M5: The polynomial kernel with the slope γ , degree n , and constant c is defined as:



$$K(x, x') = (\gamma x^T x' + c)^n \quad \gamma > 0, c \geq 0$$

M6: The sigmoid kernel with the slope γ and constant c is defined as:

$$K(x, x') = \tanh(\gamma x^T x' + c) \quad \gamma > 0, c \geq 0$$

M7: The radial basis function kernel with the slope γ is defined as:

$$K(x, x') = e^{-\gamma \|x - x'\|^2} \quad \gamma > 0$$

M8: The Laplacian kernel with the slope γ is defined as:

$$K(x, x') = e^{-\gamma \|x - x'\|} \quad \gamma > 0$$

M9: The log kernel with the degree n is defined as:

$$K(x, x') = -\log(1 + \|x - x'\|^n)$$

The list of discriminative kernel functions described earlier is just a subset of the kernel methods' universe. The other types of kernels include the following:

- **Probabilistic kernels:** These are kernels derived from generative models. Probabilistic models such as Gaussian processes can be used as a kernel function [8:3].
- **Smoothing kernels:** This is the nonparametric formulation, averaging density with the nearest neighbor observations [8:4].

- **Reproducible kernel Hilbert spaces:** This is the dot product of finite or infinite basis functions [8:5].

The kernel functions play a very important role in support vector machines for nonlinear problems.

Kernel monadic composition

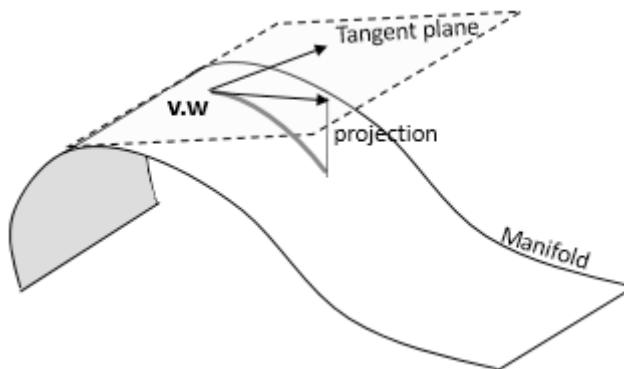
The concept of a kernel function is actually derived from differential geometry and more specifically from manifold, which was introduced in the *Non-linear models* section under *Dimension reduction* in *Chapter 4, Unsupervised Learning*.

A manifold is a low dimension features space embedded in the observation space of higher dimension. The dot (or inner) product of two observations, known as the **Riemann metric**, is computed on a Euclidean tangent space.

The heat kernel function

 The kernel function on a manifold is actually computed by solving the heat equation that uses the Laplace-Beltrami operator. The heat kernel is the solution of the heat differential equation. It associates the dot product with an exponential map.

The kernel function is the composition of the dot product on the tangent space projected on the manifold using an exponential map, as shown in the following diagram:



The visualization of a manifold, Riemann metric, and projection of an inner product

A kernel function is the composition $g \circ f$ of two functions:

- A function h that implements the Riemann metric or similarity between two vectors v and w
- A function g that implements the projection of the similarity $h(v, w)$ to the manifold (exponential map)

The `KF` class implements the kernel function as a composition of the functions g and h :

```
type F1 = Double => Double
type F2 = (Double, Double) => Double

case class KF[G](val g: G, h: F2) {
    def metric(v: DblVector, w: DblVector)
        (implicit gf: G => F1): Double = //1
        g(v.zip(w).map{ case(_v, _w) => h(_v, _w) }.sum) //2
}
```

The `KF` class is parameterized with a `G` type that can be converted to `Function1[Double, Double]`. Therefore, the computation of `metric` (dot product) requires an implicit conversion from `G` to `Function1` (line 1). The `metric` is computed by zipping the two vectors, mapping the `h` similarity function, and summing up the resulting vector (line 2).

Let's define the monadic composition for the `KF` class:

```
val kfMonad = new _Monad[KF] {
    override def map[G, H](kf: KF[G])(f: G => H): KF[H] =
        KF[H](f(kf.g), kf.h) //3
    override def flatMap[G, H](kf: KF[G])(f: G => KF[H]): KF[H] =
        KF[H](f(kf.g).g, kf.h)
}
```

The creation of the `kfMonad` instance overrides the `map` and `flatMap` methods defined in the generic `_Monad` trait, as described in the *Monads* section in *Chapter 1, Getting Started*. The implementation of the `unit` method is not essential to the monadic composition and it is, therefore, omitted.

The function argument of the `map` and `flatMap` methods applies only to the exponential map function g (line 3). The composition of two kernel functions $kf1 = g1 \circ h$ and $kf2 = g2 \circ h$ produces a kernel function $kf3 = g2 \circ (g1 \circ h) = (g2 \circ g1) \circ h = g3 \circ h$.



Interpretation of kernel functions' monadic composition

The visualization of the monadic composition of kernel functions on the manifold is quite intuitive. The composition of two kernel functions consists of composing their respective projections or exponential map functions g . The function g is directly related to the curvature of the manifold around the data point for which the metric is computed. The monadic composition of the kernel functions attempts to adjust the exponential map to fit the curvature of the manifold.

The next step is to define an implicit class to convert a kernel function of the `KF` type to its monadic representation so that it can access the `map` and `flatMap` methods (line 4):

```
implicit class kF2Monad[G](kf: KF[G]) { //4
    def map[H](f: G => H): KF[H] = kfMonad.map(kf)(f)
    def flatMap[H](f: G => KF[H]): KF[H] = kfMonad.flatMap(kf)(f)
}
```

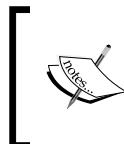
Let's implement the RBF radial basis function and the polynomial kernel function, `Polynomial`, by defining their respective g and h functions. The parameterized type for the kernel function is simply `Function1[Double, Double]`:

```
class RBF(s2: Double) extends KF[F1] {
    (x: Double) => Math.exp(-0.5*x*x/s2),
    (x: Double, y: Double) => x - y
}
class Polynomial(d: Int) extends KF[F1] {
    (x: Double) => Math.pow(1.0+x, d),
    (x: Double, y: Double) => x*y
}
```

Here is an example of the composition of two kernel functions: a `kf1` kernel RBF with a standard deviation of 0.6 (line 5) and a `kf2` polynomial kernel with a degree 3 (line 6):

```
val v = Vector[Double](0.5, 0.2, 0.3)
val w = Vector[Double](0.1, 0.7, 0.2)
val composed = for {
    kf1 <- new RBF(0.6) //5
    kf2 <- new Polynomial(3) //6
} yield kf2
composed.metric(v, w) //7
```

Finally, the `metric` is computed on the composed kernel functions (line 7).



Kernel functions in SVM

Our implementation of the support vector machine uses the kernel function included in the LIBSVM library.



Support vector machines

A support vector machine is a linear discriminative classifier that attempts to maximize the margin between classes during training. This approach is similar to the definition of a hyperplane through the training of the logistic regression (refer to the *Binomial classification* section in *Chapter 6, Regression and Regularization*). The main difference is that the support vector machine computes the optimum separating hyperplane between groups or classes of observations. The hyperplane is indeed the equation that represents the model generated through training.

The quality of the SVM depends on the distance, known as margin, between the different classes of observations. The accuracy of the classifier increases as the margin increases.

The linear SVM

First, let's apply the support vector machine to extract a linear model (classifier or regression) for a labeled set of observations. There are two scenarios for defining a linear model. The labeled observations are as follows:

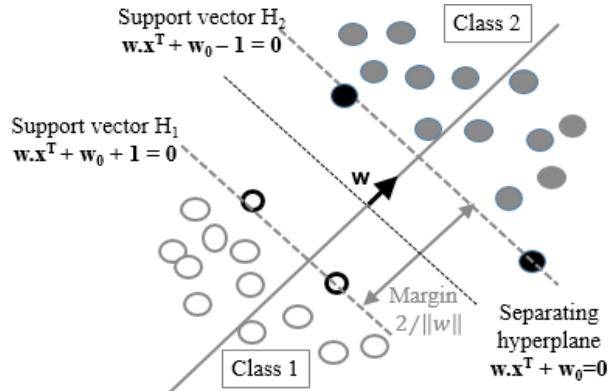
- They are naturally segregated in the features space (the **separable** case)
- They are intermingled with overlap (the **nonseparable** case)

It is easy to understand the concept of an optimal separating hyperplane in cases where the observations are naturally segregated.

The separable case – the hard margin

The concept of separating a training set of observations with a hyperplane is explained in a better way with a two-dimensional (x, y) set of observations with two classes C_1 and C_2 . The label y has the value -1 or +1.

The equation for the separating hyperplane is defined by the linear equation $y=w \cdot x + w_0$, which sits in the midpoint between the boundary data points for the class C_1 ($H_1: w \cdot x^T + w_0 + 1=0$) and class C_2 ($H_2: w \cdot x^T + w_0 - 1=0$). The planes H_1 and H_2 are the support vectors:



The visualization of the hard margin in the support vector machine

In the separable case, the support vectors fully segregate the observations into two distinct classes. The margin between the two support vectors is the same for all the observations and is known as the **hard margin**.

The separable case

M1: The support vectors equation w is represented as:

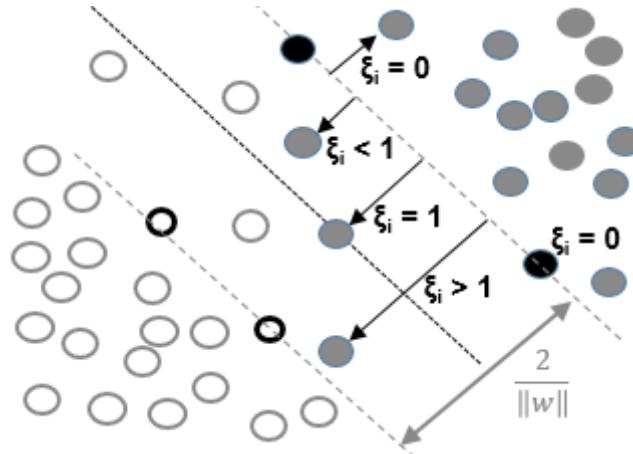
$$y_i(w^T x + w_0) \geq 1 \quad \forall i$$

M2: The hard margin optimization problem is given by:

$$\min_{w, w_0} \left\{ \frac{w^T w}{2} \right\} \text{ subject to } y_i(w^T x + w_0) \geq 1 \quad \forall i$$

The nonseparable case – the soft margin

In the nonseparable case, the support vectors cannot completely segregate observations through training. They merely become linear functions that penalize the few observations or outliers that are located outside (or beyond) their respective support vector H_1 or H_2 . The penalty variable ξ , also known as the slack variable, increases if the outlier is further away from the support vector:



The visualization of the hard margin in the support vector machine

The observations that belong to the appropriate (or own) class do not have to be penalized. The condition is similar to the hard margin, which means that the slack ξ is null. This technique penalizes the observations that belong to the class but are located beyond their support vectors; the slack ξ increases as the observations get closer to the support vector of the other class and beyond. The margin is then known as a soft margin because the separating hyperplane is enforced through a slack variable.

The nonseparable case

M3: The optimization of the soft margin for a linear SVM with C formulation is defined as:



$$\min_{w, \xi} \left\{ \frac{w^T w}{2} + C \sum_{i=0}^{n-1} \xi_i \right\}$$

$$\xi_i \geq 0, \quad y_i (w^T x + w_0) \geq 1 - \xi_i \quad \forall i$$

Here, C is the penalty (or inversed regularization) factor.

You may wonder how the minimization of the margin error is related to the loss function and the penalization factor, introduced for the ridge regression (refer to the *Numerical optimization* section in *Chapter 6, Regression and Regularization*). The second factor in the formula corresponds to the ubiquitous loss function. You will certainly be able to recognize the first term as the L2 regularization penalty with $\lambda = 1/2C$.

The problem can be reformulated as the minimization of a function known as the **primal problem** [8:6].

M4: The primal problem formulation of the support vector classifier using the L_2 regularization is as follows:

$$\min_{w, w_0} \left\{ \frac{w^T w}{2} + C \sum_{i=0}^{n-1} \mathcal{L}_i \right\} \quad \mathcal{L}_i = |1 - y_i(w^T x + w_0)|$$

The C penalty factor is the inverse of the L_2 regularization factor. The loss function L is known as the **hinge loss**. The formulation of the margin using the C penalty (or cost) parameter is known as the **C-SVM** formulation. C-SVM is sometimes called the **C-Epsilon SVM** formulation for the nonseparable case.

The **v-SVM** (or Nu-SVM) is an alternative formulation to C-SVM. The formulation is more descriptive than C-SVM; v represents the upper bound of the training observations that are poorly classified and the lower bound of the observations on the support vectors [8:7].

M5: The **v-SVM** formulation of a linear SVM using the L_2 regularization is defined as:

$$\begin{aligned} & \min_{w, \rho, \xi} \left\{ \frac{w^T w}{2} - \rho + \frac{1}{vn} \sum_{i=0}^{n-1} \xi_i \right\} \\ & \xi_i \geq 0, \quad y_i(w^T x + w_0) \geq \rho - \xi_i \quad \forall i \end{aligned}$$

Here, ρ is a margin factor used as an optimization variable.

The C-SVM formulation is used throughout the chapters for the binary, one class support vector classifier as well as the support vector regression.

Sequential Minimal Optimization

The optimization problem consists of the minimization of a quadratic objective function (w^2) subject to N linear constraints, N being the number of observations. The time complexity of the algorithm is $O(N^3)$. A more efficient algorithm known as **Sequential Minimal Optimization (SMO)** has been introduced to reduce the time complexity to $O(N^2)$.

The nonlinear SVM

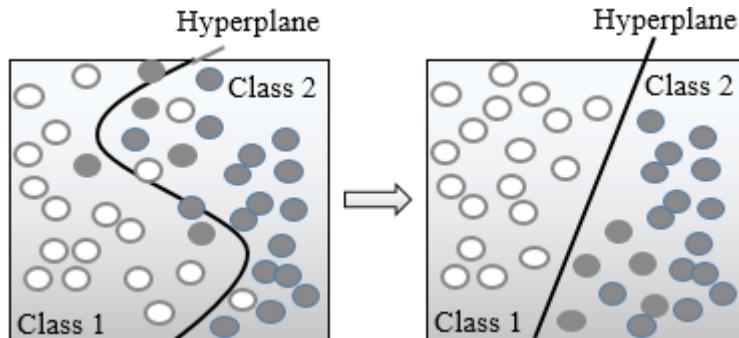
So far, we assumed that the separating hyperplane and therefore the support vectors are linear functions. Unfortunately, such assumptions are not always correct in the real world.

Max-margin classification

Support vector machines are known as large or **maximum margin classifiers**. The objective is to maximize the margin between the support vectors with hard constraints for separable (similarly, soft constraints with slack variables for nonseparable) cases.

The model parameters $\{w_i\}$ are rescaled during optimization to guarantee that the margin is at least 1. Such algorithms are known as maximum (or large) margin classifiers.

The problem of fitting a nonlinear model into the labeled observations using support vectors is not an easy task. A better alternative consists of mapping the problem to a new and higher dimensional space using a nonlinear transformation. The nonlinear separating hyperplane becomes a linear plane in the new space, as illustrated in the following diagram:



An illustration of the kernel trick in the SVM

The nonlinear SVM is implemented using a basis function $\phi(x)$. The formulation of the nonlinear C-SVM is very similar to the linear case. The only difference is the constraint along with the support vector, using the basis function ϕ (**M6**):

$$y_i(w^T \phi(x) + w_0) \geq 1 - \xi_i \quad \xi_i \geq 0 \quad \forall i$$

The minimization of $w^T \phi(x)$ in the preceding equation requires the computation of the inner product $\phi(x)^T \phi(x)$. The inner product of the basis functions is implemented using one of the kernel functions introduced in the first section. The optimization of the preceding convex problem computes the optimal hyperplane w^* as the kernelized linear combination of the training samples $y_i \phi(x_i)$ and **Lagrange multipliers**. This formulation of the optimization problem is known as the **SVM dual problem**. The description of the dual problem is mentioned as a reference and is well beyond the scope of this book [8:8].

M7: The optimal hyperplane for the SVM dual problem is defined as:

$$w^* = \sum_{i=0}^{n-1} \alpha_i y_i \phi(x_i)$$

M8: The hard margin formulation for the SVM dual problem is defined as:

$$y_i (w^T \phi(x) + w_0) = y_i \left(\sum_{i=0}^{n-1} \alpha_i y_i K(x_i, x) + w_0 \right) \geq 1$$

$$K(x_i, x) = \phi(x_i) \phi(x) \quad \forall i$$

The kernel trick

The transformation $(x, x') \Rightarrow K(x, x')$ maps a nonlinear problem into a linear problem in a higher dimensional space. It is known as the **kernel trick**.

Let's consider, for example, the polynomial kernel defined in the first section with a degree $d = 2$ and coefficient of $C0 = 1$ in a two-dimension space. The polynomial kernel function of two vectors, $x = [x_1, x_2]$ and $z = [x'_1, x'_2]$, is decomposed into a linear function in a 6 dimension space:

$$\begin{aligned} K(x, x') &= (1 + x^T x')^2 \\ &= 1 + 2x_1 x'_1 + 2x_2 x'_2 + 2x_1 x'_1 x_2 x'_2 + (x_1 x'_1)^2 + (x_2 x'_2)^2 \\ &= \phi_1(x) \cdot \phi_1(x') + \phi_2(x) \cdot \phi_2(x') + \phi_3(x) \cdot \phi_3(x') + \dots \\ \phi_2(x) &= 1, \phi_2(x) = \sqrt{2}x_1, \phi_3(x) = \sqrt{2}x_2, \phi_4(x) = x_1^2 \dots \end{aligned}$$

Support vector classifiers – SVC

Support vector machines can be applied to classification, anomalies detection, and regression problems. Let's first dive into the support vector classifiers.

The binary SVC

The first classifier to be evaluated is the binary (2-class) support vector classifier. The implementation uses the LIBSVM library created by Chih-Chung Chang and Chih-Jen Lin from the National Taiwan University [8:9].

LIBSVM

The library was originally written in C before being ported to Java. It can be downloaded from <http://www.csie.ntu.edu.tw/~cjlin/libsvm> as a .zip or tar.gz file. The library includes the following classifier modes:

- Support vector classifiers (C-SVC, ν-SVC, and one-class SVC)
- Support vector regression (ν-SVR and ε-SVR)
- RBF, linear, sigmoid, polynomial, and precomputed kernels

LIBSVM has the distinct advantage of using **Sequential Minimal Optimization (SMO)**, which reduces the time complexity of a training of n observations to $O(n^2)$. The LIBSVM documentation covers both the theory and implementation of hard and soft margins and is available at <http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf>.

Why LIBSVM?

There are alternatives to the LIBSVM library for learning and experimenting with SVM. David Soergel from the University of Berkeley refactored and optimized the Java version [8:10]. Thorsten Joachims' **SVMLight** [8:11] Spark/MLLib 1.0 includes two Scala implementations of the SVM using resilient distributed datasets (refer to the *Apache Spark* section in *Chapter 12, Scalable Frameworks*). However, LIBSVM is the most commonly used SVM library.

The implementation of the different support vector classifiers and the support vector regression in LIBSVM is broken down into the following five Java classes:

- `svm_model`: This defines the parameters of the model created during training
- `svm_node`: This models the element of the sparse matrix Q , which is used in the maximization of the margins
- `svm_parameters`: This contains the different models for support vector classifiers and regressions, the five kernels supported in LIBSVM with their parameters, and the `weights` vectors used in cross-validation
- `svm_problem`: This configures the input to any of the SVM algorithm (the number of observations, input vector data x as a matrix, and the vector of labels y)
- `svm`: This implements algorithms used in training, classification, and regression

The library also includes template programs for training, prediction, and normalization of datasets.



The LIBSVM Java code

The Java version of LIBSVM is a direct port of the original C code. It does not support generic types and is not easily configurable (the code uses switch statements instead of polymorphism). For all its limitations, LIBSVM is a fairly well-tested and robust Java library for SVMs.

Let's create a Scala wrapper to the LIBSVM library to improve its flexibility and ease of use.

Design

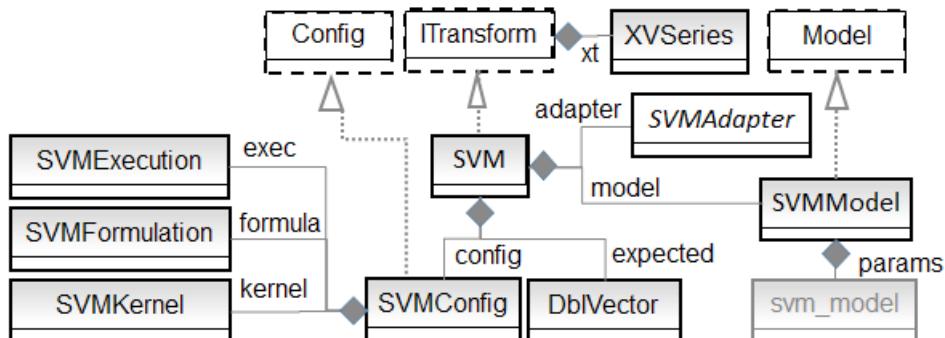
The implementation of the support vector machine algorithm uses the design template for classifiers (refer to the *Design template for classifier* section in the *Appendix A, Basic Concepts*).

The key components of the implementation of a SVM are as follows:

- A model, `SVMModel`, of the `Model` type is initialized through training during the instantiation of the classifier. The model class is an adapter to the `svm_model` structure defined in LIBSVM.
- An `SVMAdapter` object interfaces with the internal LIBSVM data structures and methods.

- The SVM support vector machine class is implemented as an implicit data transformation of the `ITransform` type. It has three parameters: the configuration wrapper of the `SVMConfig` type, the features/time series of the `XVSeries` type, and the target or labeled values, `DblVector`.
- The configuration (the `SVMConfig` type) consists of three distinct elements: `SVMExecution` that defines the execution parameters such as the maximum number of iterations or convergence criteria, `SVMKernel` that specifies the kernel function used during training, and `SVMFormulation` that defines the formula (C , ϵ , or ν) used to compute a nonseparable case for the support vector classifier and regression.

The key software components of the support vector machine are described in the following UML class diagram:



The UML class diagram for the support vector machine

The UML diagram omits the helper traits and classes such as `Monitor` or the Apache Commons Math components.

Configuration parameters

LIBSVM exposes a large number of parameters for the configuration and execution of any of the SVM algorithms. Any SVM algorithm is configured with three categories of parameters, which are as follows:

- Formulation (or type) of the SVM algorithms (the multiclass classifier, one-class classifier, regression, and so on) using the `SVMFormulation` class
- The kernel function used in the algorithm (the RBF kernel, Sigmoid kernel, and so on) using the `SVMKernel` class
- Training and executing parameters (the convergence criteria, number of folds for cross-validation, and so on) using the `SVMExecution` class

The SVM formulation

The instantiation of the configuration consists of initializing the `param LIBSVM` parameter by the SVM type, kernel, and the execution context selected by the user.

Each of the SVM parameters' case class extends the generic `SVMConfigItem` trait:

```
trait SVMConfigItem { def update(param: svm_parameter): Unit }
```

The classes inherited from `SVMConfigItem` are responsible for updating the list of the SVM parameters, `svm_parameter`, defined in LIBSVM. The `update` method encapsulates the configuration of LIBSVM.

The formulation of the SVM algorithm by a class hierarchy with `SVMFormulation` as the base trait is as follows:

```
sealed trait SVMFormulation extends SVMConfigItem {
    def update(param: svm_parameter): Unit
}
```

The list of the formulation for the SVM (`C`, `nu`, and `eps` for regression) is completely defined and known. Therefore, the hierarchy should not be altered and the `SVMFormulation` trait has to be declared sealed. Here is an example of the SVM `CSVCFormulation` formulation class, which defines the C-SVM model:

```
class CSVCFormulation (c: Double) extends SVMFormulation {
    override def update(param: svm_parameter): Unit = {
        param.svm_type = svm_parameter.C_SVC
        param.C = c
    }
}
```

The other SVM `NusVCFormulation`, `OneSVCFormulation`, and `SVRFomulation` formulation classes implement the ν-SVM, 1-SVM, and ε-SVM, respectively for regression models.

The SVM kernel function

Next, you need to specify the kernel functions by defining and implementing the `SVMKernel` trait:

```
sealed trait SVMKernel extends SVMConfigItem {
    override def update(param: svm_parameter): Unit
}
```

Once again, there are a limited number of kernel functions supported in LIBSVM. Therefore, the hierarchy of kernel functions is sealed. The following code snippet configures the radius basis function kernel, `RbfKernel`, as an example of the definition of the kernel definition class:

```
class RbfKernel(gamma: Double) extends SVMKernel {
    override def update(param: svm_parameter): Unit = {
        param.kernel_type = svm_parameter.RBF
        param.gamma = gamma
    }
}
```

The fact that the LIBSVM Java byte code library is not very extensible does not prevent you from defining a new kernel function in the LIBSVM source code. For example, the Laplacian kernel can be added by performing the following steps:

1. Create a new kernel type in `svm_parameter`, such as `svm_parameter.LAPLACE = 5.`
2. Add the kernel function name to `kernel_type_table` in the `svm` class.
3. Add `kernel_type != svm_parameter.LAPLACE` to the `svm_check_parameter` method.
4. Add the implementation of the kernel function to two values in `svm: kernel_function` (Java code):

```
case svm_parameter.LAPLACE:
    double sum = 0.0;
    for(int k = 0; k < x[i].length; k++) {
        final double diff = x[i][k].value - x[j][k].value;
        sum += diff*diff;
    }
    return Math.exp(-gamma*Math.sqrt(sum));
```

5. Add the implementation of the Laplace kernel function in the `svm.k_function` method by modifying the existing implementation of RBF (`distanceSqr`).
6. Rebuild the `libsvm.jar` file

The SVM execution

The `SVMEexecution` class defines the configuration parameters for the execution of the training of the model, namely the `eps` convergence factor for the optimizer (line 2), the size of the cache, `cacheSize` (line 1), and the number of folds, `nFolds`, used during cross-validation:

```
class SVMEexecution(cacheSize: Int, eps: Double, nFolds: Int)
    extends SVMConfigItem {
```

```
    override def update(param: svm_parameter): Unit = {
      param.cache_size = cacheSize //1
      param.eps = eps //2
    }
}
```

The cross-validation is performed only if the `nFolds` value is greater than 1.

We are finally ready to create the `SVMConfig` configuration class, which hides and manages all of the different configuration parameters:

```
class SVMConfig(formula: SVMFormulation, kernel: SVMKernel,
                exec: SVMExecution) {
  val param = new svm_parameter
  formula.update(param) //3
  kernel.update(param) //4
  exec.update(param) //5
}
```

The `SVMConfig` class delegates the selection of the formula to the `SVMFormulation` class (line 3), selection of the kernel function to the `SVMKernel` class (line 4), and the execution of parameters to the `SVMExecution` class (line 5). The sequence of update calls initializes the LIBSVM list of configuration parameters.

Interface to LIBSVM

We need to create an adapter object to encapsulate the invocation to LIBSVM. The `SVMAdapter` object hides the LIBSVM internal data structures: `svm_model` and `svm_node`:

```
object SVMAdapter {
  type SVMNodes = Array[Array[svm_node]]
  class SVMProblem(numObs: Int, expected: DblArray) //6

  def createSVMNode(dim: Int, x: DblArray): Array[svm_node] //7
  def predictSVM(model: SVMModel, x: DblArray): Double //8
  def crossValidateSVM(problem: SVMProblem, //9
                       param: svm_parameter, nFolds: Int, expected: DblArray)
  def trainSVM(problem: SVMProblem, //10
               param: svm_parameter): svm_model
}
```

The `SVMAdapter` object is a single entry point to LIBSVM for training, validating a SVM model, and executing predictions:

- `SVMPProblem` wraps the definition of the training objective or problem in LIBSVM, using the labels or expected values (line 6)
- `createSVMNode` creates a new computation node for each observation x (line 7)
- `predictSVM` predicts the outcome of a new observation x given a model, `svm_model`, generated through training (line 8)
- `crossValidateSVM` validates the model, `svm_model`, with the `nFold` training-validation sets (line 9)
- `trainSVM` executes the `problem` training configuration (line 10)

svm_node

The LIBSVM `svm_node` Java class is defined as a pair of indices of the feature in the observation array and its value:



```
public class svm_node implements java.io.Serializable {
    public int index;
    public double value;
}
```

The `SVMAdapter` methods are described in the next section.

Training

The model for the SVM is defined by the following two components:

- `svm_model`: This is the SVM model parameters defined in LIBSVM
- `accuracy`: This is the accuracy of the model computed during cross-validation

The code will be as follows:

```
case class SVMModel(val svmmode1: svm_node,
                    val accuracy: Double) extends Model {
    lazy val residuals: DblArray = svmmode1.sv_coef(0)
}
```

The `residuals`, that is, $r = y - f(x)$ are computed in the LIBSVM library.



Accuracy in the SVM model

You may wonder why the value of the accuracy is a component of the model. The accuracy component of the model provides the client code with a quality metric associated with the model. Integrating the accuracy into the model, allows the user to make informed decisions in accepting or rejecting the model. The accuracy is stored in the model file for subsequent analysis.

Next, let's create the first support vector classifier for the two-class problems. The SVM class implements the `ITransform` monadic data transformation that implicitly generates a model from a training set, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 11).

The constructor for the SVM follows the template described in the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*:

```
class SVM[T <% Double] (config: SVMConfig, xt: XVSeries[T],
  expected: DblVector) extends ITransform[Array[T]](xt) { //11

  type V = Double    //12
  val normEPS = config.eps*1e-7   //13
  val model: Option[SVMModel] = train //14

  def accuracy: Option[Double] = model.map(_.accuracy) //15
  def mse: Option[Double] //16
  def margin: Option[Double] //17
}
```

The implementation of the `ITransform` abstract class requires the definition of the output value of the predictor as a `Double` (line 12). The `normEPS` is used for rounding errors in the computation of the margin (line 13). The model of the `SVMModel` type is generated through training by the `svm` constructor (line 14). The last four methods are used to compute the parameters of the accuracy model (line 15), the mean square of errors, `mse`, (line 16), and the `margin` (line 17).

Let's take a look at the training method, `train`:

```
def train: Option[SVMModel] = Try {
  val problem = new SVMProblem(xt.size, expected.toArray) //18
  val dim = dimension(xt)

  xt.zipWithIndex.foreach{ case (_x, n) => //19
    problem.update(n, createSVMNode(dim, _x))
  }
```

```
new SVMModel(trainSVM(problem, config.param),  
            accuracy(problem)) //20  
}.toOption("SVM training failed", logger)
```

The `train` method creates `SVMProblem` that provides LIBSVM with the training components (line 18). The purpose of the `SVMProblem` class is to manage the definition of training parameters implemented in LIBSVM, as follows:

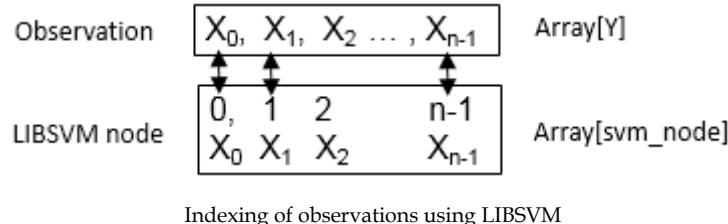
```
class SVMProblem(numObs: Int, expected: DblArray) {  
    val problem = new svm_problem //21  
    problem.l = numObs  
    problem.y = expected  
    problem.x = new SVMNodes(numObs)  
  
    def update(n: Int, node: Array[svm_node]): Unit =  
        problem.x(n) = node //22  
}
```

The arguments of the `SVMProblem` constructor, the number of observations, and the labels or expected values are used to initialize the corresponding `svm_problem` data structure in LIBSVM (line 21). The `update` method maps each observation, which is defined as an array of `svm_node` to the problem (line 22).

The `createSVMNode` method creates an array of `svm_node` from an observation. A `svm_node` in LIBSVM is the pair of the `j` index of a feature in an observation (line 23) and its value, `y` (line 24):

```
def createSVMNode(dim: Int, x: DblArray): Array[svm_node] = {  
    val newNode = new Array[svm_node](dim)  
    x.zipWithIndex.foreach{ case (y, j) => {  
        val node = new svm_node  
        node.index= j //23  
        node.value = y //24  
        newNode(j) = node  
    }}  
    newNode
```

The mapping between an observation and a LIBSVM node is illustrated in the following diagram:



The `trainsvm` method pushes the training request with a well-defined problem and configuration parameters to LIBSVM by invoking the `svm_train` method (line 26):

```
def trainSVM(problem: SVMProblem,
            param: svm_parameter): svm_model =
    svm.svm_train(problem.problem, param) //26
```

The accuracy is the ratio of the true positive plus the true negative over the size of the test sample (refer to the *Key quality metrics* section in *Chapter 2, Hello World!*). It is computed through cross-validation only if the number of folds initialized in the `SVMExecution` configuration class is greater than 1. Practically, the accuracy is computed by invoking the cross-validation method, `svm_cross_validation`, in the LIBSVM package, and then computing the ratio of the number of predicted values that match the labels over the total number of observations:

```
def accuracy(problem: SVMProblem): Double = {
    if( config.isCrossValidation ) {
        val target = new Array[Double](expected.size)
        crossValidateSVM(problem, config.param, //27
                         config.nFolds, target)

        target.zip(expected)
            .filter{case(x, y) =>Math.abs(x- y) < config.eps} //28
            .size.toDouble/expected.size
    }
    else 0.0
}
```

The call to the `crossValidateSVM` method of `SVMAdapter` forwards the configuration and execution of the cross validation with `config.nFolds` (line 27):

```
def crossValidateSVM(problem: SVMProblem, param: svm_parameter,
                     nFolds: Int, expected: DblArray) {
    svm.svm_cross_validation(problem.problem, param,
                            nFolds, expected)
}
```

The Scala `filter` weeds out the observations that were poorly predicted (line 28). This minimalist implementation is good enough to start exploring the support vector classifier.

Classification

The implementation of the `|>` classification method for the `SVM` class follows the same pattern as the other classifiers. It invokes the `predictSVM` method in `SVMAdapter` that forwards the request to LIBSVM (line 29):

```
override def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T] if(x.size == dimension(xt) && isModel) =>
    Try( predictSVM(model.get.svmmodel, x) ) //29
}
```

C-penalty and margin

The first evaluation consists of understanding the impact of the penalty factor C on the margin in the generation of the classes. Let's implement the computation of the margin. The margin is defined as $2/\|w\|$ and implemented as a method of the `SVM` class, as follows:-

```
def margin: Option[Double] =
  if(isModel) {
    val wNorm = model.get.residuals./:(0.0)((s,r) => s + r*r)
    if(wNorm < normEPS) None else Some(2.0/Math.sqrt(wNorm))
  }
  else None
```

The first instruction computes the sum of the squares, `wNorm`, of the residuals $r = y - f(x|w)$. The margin is ultimately computed if the sum of squares is significant enough to avoid rounding errors.

The margin is evaluated using an artificially generated time series and labeled data. First, we define the method to evaluate the margin for a specific value of the penalty (inversed regularization coefficient) factor C :

```
val GAMMA = 0.8
val CACHE_SIZE = 1<<8
val NFOLDS = 1
val EPS = 1e-5

def evalMargin(features: Vector[DblArray],
  expected: DblVector, c: Double): Int = {
  val execEnv = SVMExecution(CACHE_SIZE, EPS, NFOLDS)
```

```

val config = SVMConfig(new CSVCFormulation(c),
    new RbfKernel(GAMMA), execEnv)
val svc = SVM[Double](config, features, expected)
svc.margin.map(_.toString)      //30
}

```

The `evalMargin` method uses the `CACHE_SIZE`, `EPS`, and `NFOLDS` execution parameters. The execution displays the value of the margin for different values of C (line 30). The method is invoked iteratively to evaluate the impact of the penalty factor on the margin extracted from the training of the model. The test uses a synthetic time series to highlight the relation between C and the margin. The synthetic time series created by the `generate` method consists of two training sets of an equal size, N :

- Data points generated as $y = x(1 + r/5)$ for the label 1, r being a randomly generated number over the range $[0,1]$ (line 31)
- A randomly generated data point $y = r$ for the label -1 (line 32)

Consider the following code:

```

def generate: (Vector[DblArray], DblArray) = {
    val z = Vector.tabulate(N)(i => {
        val ri = i*(1.0 + 0.2*Random.nextDouble)
        Array[Double](i, ri) //31
    }) ++
    Vector.tabulate(N)(i => Array[Double](i, i*Random.nextDouble))
    (z, Array.fill(N)(1) ++ Array.fill(N)(-1)) //32
}

```

The `evalMargin` method is executed for different values of C ranging from 0 to 5:

```

generate.map(y =>
    (0.1 until 5.0 by 0.1)
        .flatMap(evalMargin(y._1, y._2, _)).mkString("\n")
)

```

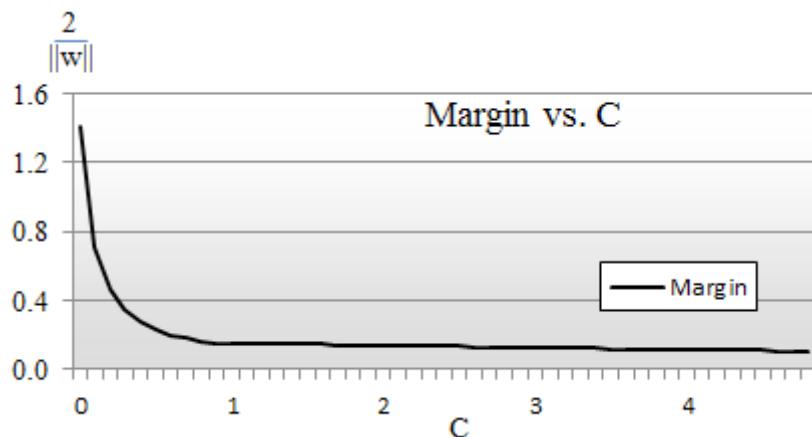
val versus final val

There is a difference between a val and a final val. A nonfinal value can be overridden in a subclass. Overriding a final value produces a compiler error, as follows:



```
class A { val x = 5;  final val y = 8 }
class B extends A {
    override val x = 9 // OK
    override val y = 10 // Error
}
```

The following chart illustrates the relation between the penalty or cost factor C and the margin:



The margin value versus the C -penalty factor for a support vector classifier

As expected, the value of the margin decreases as the penalty term C increases. The C penalty factor is related to the L_2 regularization factor λ as $C \sim 1/\lambda$. A model with a large value of C has a high variance and a low bias, while a small value of C will produce lower variance and a higher bias.

Optimizing C penalty

The optimal value for C is usually evaluated through cross-validation, by varying C in incremental powers of 2: $2^n, 2^{n+1}, \dots$ [8:12].



Kernel evaluation

The next test consists of comparing the impact of the kernel function on the accuracy of the prediction. Once again, a synthetic time series is generated to highlight the contribution of each kernel. The test code uses the runtime prediction or classification method, `|>`, to evaluate the different kernel functions. Let's create a method to evaluate and compare these kernel functions. All we need is the following (line 33):

- An `xt` training set of the `Vector[DblArray]` type
- A test set, `test`, of the `Vector[DblArray]` type
- A set of `labels` for the training set that takes the value 0 or 1
- A `kF` kernel function

Consider the following code:

```
val C = 1.0
def evalKernel(xt: Vector[DblArray], test: Vector[DblArray],
               labels: DblVector, kF: SVMKernel): Double = { //33

    val config = SVMConfig(new CSVCFormulation(C), kF) //34
    val svc = SVM[Double](config, xt, labels)
    val pfnsvc = svc |> //35
    test.zip(labels).count{case(x, y) => pfnsvc(x).get == y}
        .toDouble/test.size //36
}
```

The `config` configuration of the SVM uses the `C` penalty factor 1, the `C`-formulation, and the default execution environment (line 34). The predictive `pfnsvc` partial function (line 35) is used to compute the predictive values for the test set. Finally, the `evalKernel` method counts the number of successes for which the predictive values match the labeled or expected values. The accuracy is computed as the ratio of the successful prediction over the size of the test sample (line 36).

In order to compare the different kernels, let's generate three datasets of the size $2N$ for a binomial classification using the pseudo-random `genData` data generation method:

```
def genData(variance: Double, mean: Double): Vector[DblArray] = {
    val rGen = new Random(System.currentTimeMillis)
    Vector.tabulate(N) (_ => {
        rGen.setSeed(rGen.nextLong)
        Array[Double](rGen.nextDouble, rGen.nextDouble)
            .map(variance*_ - mean) //37
    })
}
```

The random value is computed through a transformation $f(x) = variance * x + mean$ (line 37). The training and test sets consist of the aggregate of two classes of data points:

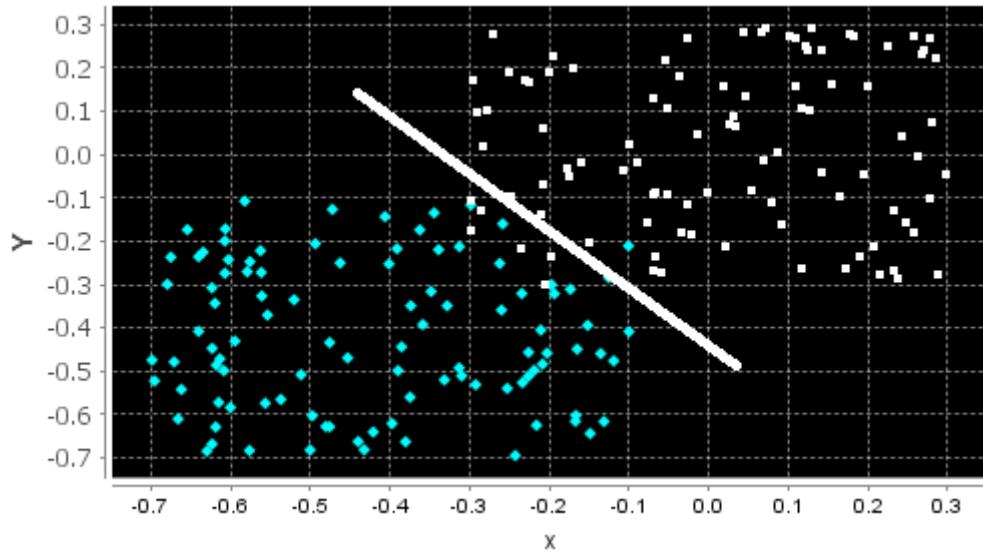
- Random data points with the variance a and mean b associated with the label 0.0
- Random data points with the variance a and mean $1-b$ associated with the label 1.0

Consider the following code for the training set:

```
val trainSet = genData(a, b) ++ genData(a, 1-b)
val testSet = genData(a, b) ++ genData(a, 1-b)
```

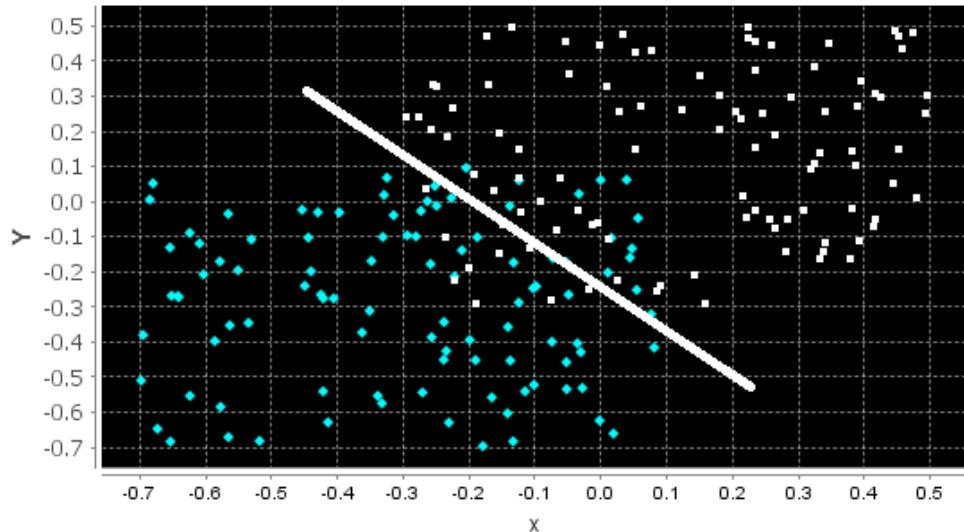
The a and b parameters are selected from two groups of training data points with various degrees of separation to illustrate the separating hyperplane.

The following chart describes the high margin; the first training set generated with the parameters $a = 0.6$ and $b = 0.3$ illustrates the highly separable classes with a clean and distinct hyperplane:



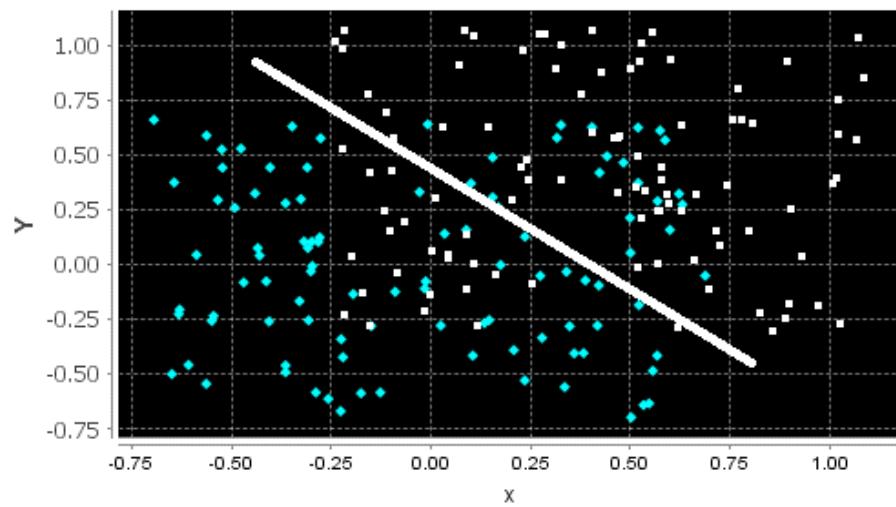
The scatter plot for training and testing sets with $a = 0.6$ and $b = 0.3$

The following chart describes the medium margin; the parameters $a = 0.8$ and $b = 0.3$ generate two groups of observations with some overlap:



The scatter plot for training and testing sets with $a = 0.8$ and $b = 0.3$

The following chart describes the low margin; the two groups of observations in this last training set are generated with $a = 1.4$ and $b = 0.3$ and show a significant overlap:



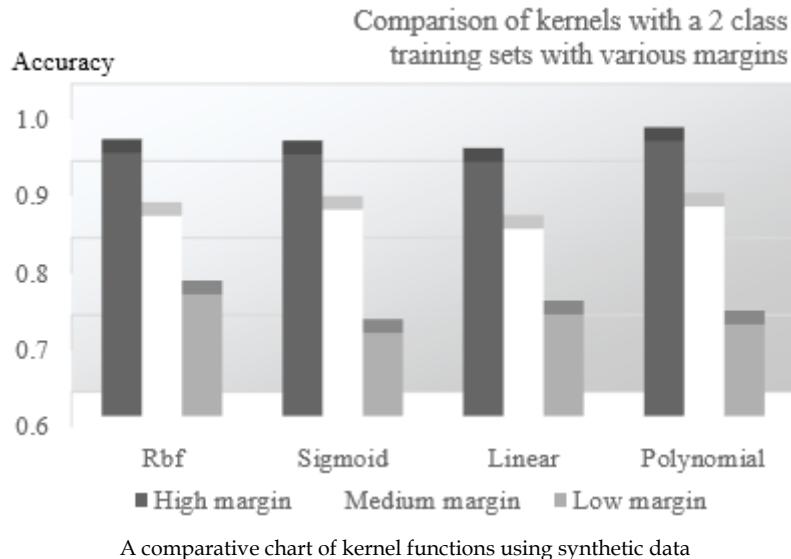
The scatter plot for training and testing sets with $a = 1.4$ and $b = 0.3$

The test set is generated in a similar fashion as the training set, as they are extracted from the same data source:

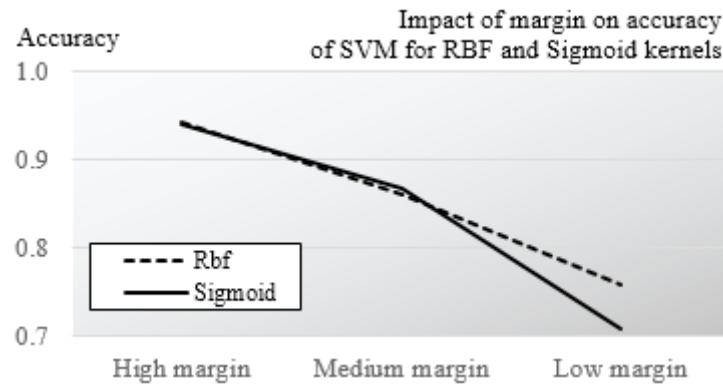
```
val GAMMA = 0.8; val COEF0 = 0.5; val DEGREE = 2 //38
val N = 100

def compareKernel(a: Double, b: Double) {
    val labels = Vector.fill(N)(0.0) ++ Vector.fill(N)(1.0)
    evalKernel(trainSet, testSet, labels, new RbfKernel(GAMMA))
    evalKernel(trainSet, testSet, labels,
               new SigmoidKernel(GAMMA))
    evalKernel(trainSet, testSet, labels, LinearKernel)
    evalKernel(trainSet, testSet, labels,
               new PolynomialKernel(GAMMA, COEF0, DEGREE))
}
```

The parameters for each of the four kernel functions are arbitrary selected from textbooks (line 38). The evalKernel method defined earlier is applied to the three training sets: the high margin ($a = 1.4$), medium margin ($a = 0.8$), and low margin ($a = 0.6$) with each of the four kernels (RBF, sigmoid, linear, and polynomial). The accuracy is assessed by counting the number of observations correctly classified for all of the classes for each invocation of the predictor, |>:



Although the different kernel functions do not differ in terms of the impact on the accuracy of the classifier, you can observe that the RBF and polynomial kernels produce results that are slightly more accurate. As expected, the accuracy decreases as the margin decreases. A decreasing margin indicates that the cases are not easily separable, affecting the accuracy of the classifier:



The impact of the margin value on the accuracy of RBF and Sigmoid kernel functions

A test case design



The test to compare the different kernel methods is highly dependent on the distribution or mixture of data in the training and test sets. The synthetic generation of data in this test case is used for illustrating the margin between classes of observations. Real-world datasets may produce different results.

In summary, there are four steps required to create a SVC-based model:

1. Select a features set.
2. Select the C-penalty (inverse regularization).
3. Select the kernel function.
4. Tune the kernel parameters.

As mentioned earlier, this test case relies on synthetic data to illustrate the concept of the margin and compare kernel methods. Let's use the support vector classifier for a real-world financial application.

Applications in risk analysis

The purpose of the test case is to evaluate the risk for a company to curtail or eliminate its quarterly or yearly dividend. The features selected are financial metrics relevant to a company's ability to generate a cash flow and pay out its dividends over the long term.

We need to select any subset of the following financial technical analysis metrics (refer to *Appendix A, Basic Concepts*):

- Relative change in stock prices over the last 12 months
- Long-term debt-equity ratio
- Dividend coverage ratio
- Annual dividend yield
- Operating profit margin
- Short interest (ratio of shares shorted over the float)
- Cash per share-share price ratio
- Earnings per share trend

The earnings trend has the following values:

- -2 if earnings per share decline by more than 15 percent over the last 12 months.
- -1 if earnings per share decline between 5 percent and 15 percent.
- 0 if earnings per share is maintained within 5 percent.
- +1 if earnings per share increase between 5 percent and 15 percent.
- +2 if earnings per share increase by more than 15 percent. The values are normalized with values 0 and 1.

The labels or expected output (dividend changes) is categorized as follows:

- -1 if the dividend is cut by more than 5 percent
- 0 if the dividend is maintained within 5 percent
- +1 if the dividend is increased by more than 5 percent

Let's combine two of these three labels $\{-1, 0, 1\}$ to generate two classes for the binary SVC:

- Class C1 = stable or decreasing dividends and class C2 = increasing dividends – training set A
- Class C1 = decreasing dividends and class C2 = stable or increasing dividends – training set B

The different tests are performed with a fixed set of C and GAMMA configuration parameters and a 2-fold validation configuration:

```

val path = "resources/data/chap8/dividends2.csv"
val C = 1.0
val GAMMA = 0.5
val EPS = 1e-2
val NFOLDS = 2

val extractor = relPriceChange :: debtToEquity :: dividendCoverage :: cashPerShareToPrice :: epsTrend :: shortInterest :: dividendTrend :: List[Array[String] =>Double] () //39

val pfnSrc = DataSource(path, true, false,1) |> //40
val config = SVMConfig(new CSVCFormulation(C),
    new RbfKernel(GAMMA), SVMExecution(EPS, NFOLDS))

for {
    input <- pfnSrc(extractor) //41
    obs <- getObservations(input) //42
    svc <- SVM[Double](config, obs, input.last.toVector)
} yield {
    show(s"${svc.toString}\naccuracy ${svc.accuracy.get}")
}

```

The first step is to define the `extractor` (which is the list of fields to be retrieved from the `dividends2.csv` file) (line 39). The `pfnSrc` partial function generated by the `DataSource` transformation class (line 40) converts the input file into a set of typed fields (line 41). An observation is an array of fields. The `obs` sequence of observations is generated from the input fields by transposing the matrix `observations x features` (line 42):

```

def getObservations(input: Vector[DblArray]): Try[Vector[DblArray]] = Try {
    transpose( input.dropRight(1).map(_.toArray) ).toVector
}

```

The test computes the model parameters and the accuracy from the cross-validation during the instantiation of the SVM.

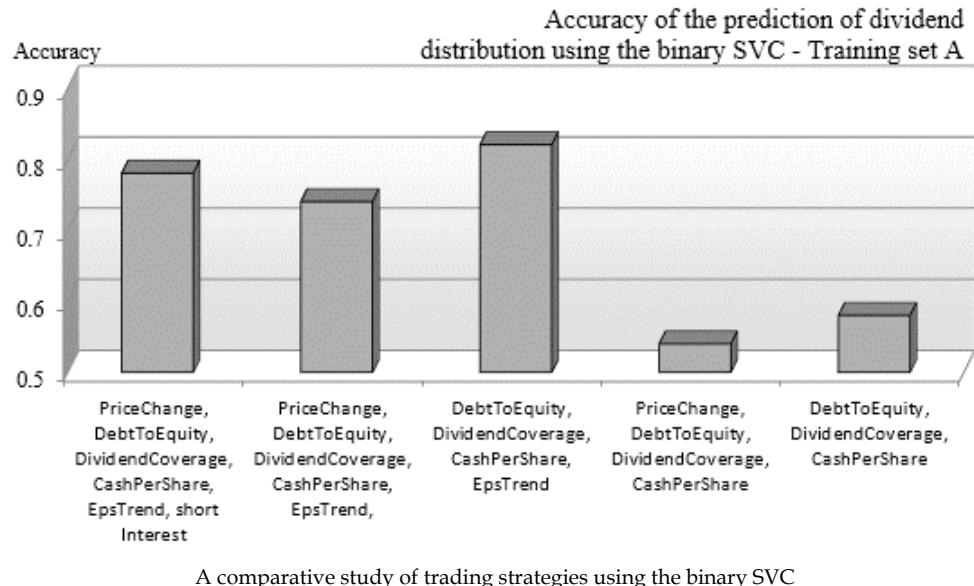
LIBSVM scaling

 LIBSVM supports feature normalization known as scaling, prior to training. The main advantage of scaling is to avoid attributes in greater numeric ranges, dominating those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during the calculation. In our examples, we use the normalization method of the `normalize` time series. Therefore, the scaling flag in LIBSVM is disabled.

The test is repeated with a different set of features and consists of comparing the accuracy of the support vector classifier for different features sets. The features sets are selected from the content of the .csv file by assembling the extractor with different configurations, as follows:

```
val extractor = ... :: dividendTrend :: ...
```

Let's take a look at the following graph:

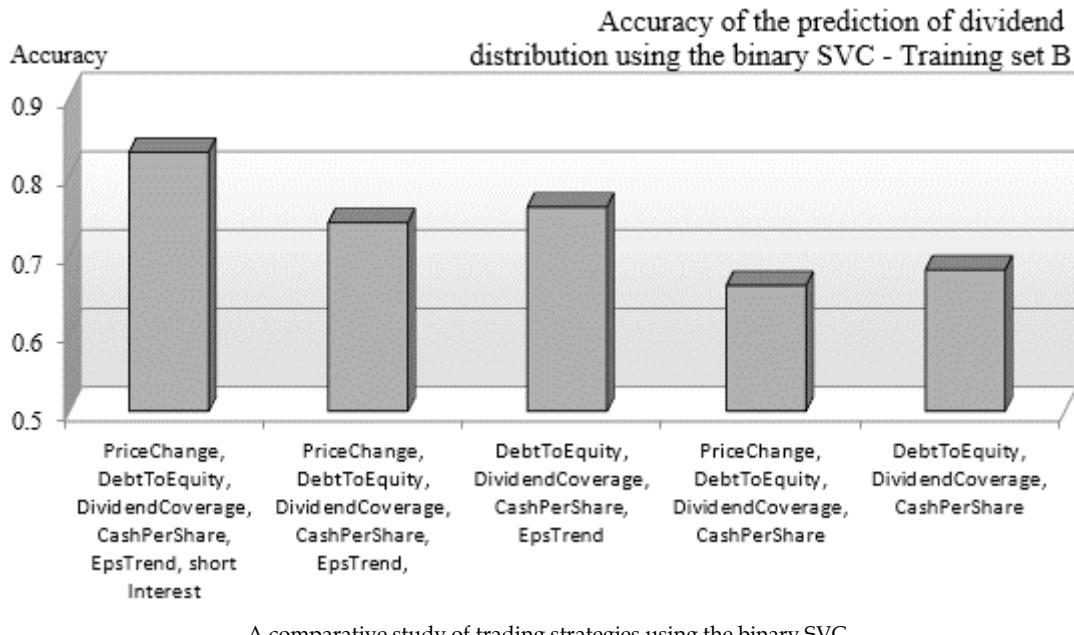


The test demonstrates that the selection of the proper features set is the most critical step in applying the support vector machine, and any other model for that matter, to classification problems. In this particular case, the accuracy is also affected by the small size of the training set. The increase in the number of features also reduces the contribution of each specific feature to the loss function.

The N-fold cross-validation

The cross-validation in this test example uses only two folds because the number of observations is small, and you want to make sure that any class contains at least a few observations.

The same process is repeated for the test B whose purpose is to classify companies with decreasing dividends and companies with stable or increasing dividends, as shown in the following graph:



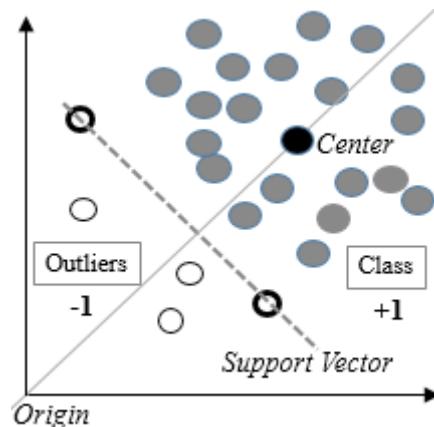
The difference in terms of accuracy of prediction between the first three features set and the last two features set in the preceding graph is more pronounced in test A than test B. In both the tests, the eps feature (earning per share) trend improves the accuracy of the classification. It is a particularly good predictor for companies with increasing dividends.

The problem of predicting the distribution (or not) dividends can be restated as evaluating the risk of a company to dramatically reduce its dividends.

What is the risk if a company eliminates its dividend altogether? Such a scenario is rare, and these cases are actually outliers. A one-class support vector classifier can be used to detect outliers or anomalies [8:13].

Anomaly detection with one-class SVC

The design of the one-class SVC is an extension of the binary SVC. The main difference is that a single class contains most of the baseline (or normal) observations. A reference point, known as the SVC origin, replaces the second class. The outliers (or abnormal) observations reside beyond (or outside) the support vector of the single class:



The visualization of the one-class SVC

The outlier observations have a labeled value of -1, while the remaining training sets are labeled +1. In order to create a relevant test, we add four more companies that have drastically cut their dividends (ticker symbols WLT, RGS, MDC, NOK, and GM). The dataset includes the stock prices and financial metrics recorded prior to the cut in dividends.

The implementation of this test case is very similar to the binary SVC driver code, except for the following:

- The classifier uses the Nu-SVM formulation, `OneSVFormulation`
- The labeled data is generated by assigning -1 to companies that have eliminated their dividends and +1 for all other companies

The test is executed against the `resources/data/chap8/dividends2.csv` dataset. First, we need to define the formulation for the one-class SVM:

```
class OneSVCFormulation(nu: Double) extends SVMFormulation {
    override def update(param: svm_parameter): Unit = {
        param.svm_type = svm_parameter.ONE_CLASS
        param.nu = nu
    }
}
```

The test code is similar to the execution code for the binomial SVC. The only difference is the definition of the output labels; -1 for companies eliminating dividends and +1 for all other companies:

```
val NU = 0.2
val GAMMA = 0.5
val EPS = 1e-3
val NFOLDS = 2

val extractor = relPriceChange :: debtToEquity :::
    dividendCoverage :: cashPerShareToPrice :: epsTrend :::
    dividendTrend :: List[Array[String] =>Double] ()

val filter = (x: Double) => if(x == 0) -1.0 else 1.0 //43
val pfnsr = DataSource(path, true, false, 1) |>
    val config = SVMConfig(new OneSVCFormulation(NU), //44
        new RbfKernel(GAMMA), SVMExecution(EPS, NFOLDS))

for {
    input <- pfnsr(extractor)
    obs <- getObservations(input)
    svc <- SVM[Double](config, obs,
        input.last.map(filter(_)).toVector)
} yield {
    show(s"${svc.toString}\naccuracy ${svc.accuracy.get}"')
}
```

The labels or expected data is generated by applying a binary filter to the last `dividendTrend` field (line 43). The formulation in the configuration has the `OneSVCFormulation` type (line 44).

The model is generated with the accuracy of 0.821. This level of accuracy should not be a surprise; the outliers (companies that eliminated their dividends) are added to the original dividend .csv file. These outliers differ significantly from the baseline observations (companies who have reduced, maintained, or increased their dividends) in the original input file.

In cases where the labeled observations are available, the one-class support vector machine is an excellent alternative to clustering techniques.

The definition of an anomaly

The results generated by a one-class support vector classifier depend heavily on the subjective definition of an outlier. The test case assumes that the companies that eliminate their dividends have unique characteristics that set them apart and are different even from companies who have cut, maintained, or increased their dividends. There is no guarantee that this assumption is indeed always valid.

Support vector regression

Most of the applications using support vector machines are related to classification. However, the same technique can be applied to regression problems. Luckily, as with classification, LIBSVM supports two formulations for support vector regression:

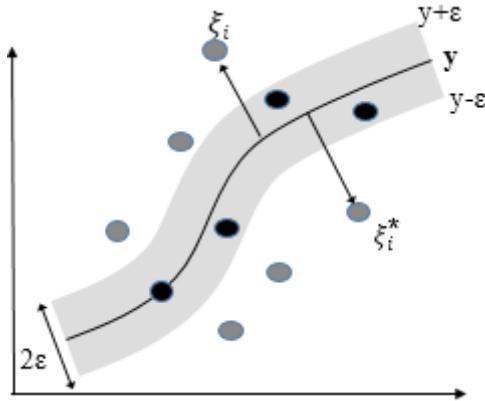
- ϵ -VR (sometimes called C-SVR)
- v-SVR

For the sake of consistency with the two previous cases, the following test uses the ϵ (or C) formulation of the support vector regression.

An overview

The SVR introduces the concept of **error insensitive zone** and insensitive error, ϵ . The insensitive zone defines a range of values around the predictive values, $y(x)$. The penalization component C does not affect the data point $\{x_i, y_i\}$ that belongs to the insensitive zone [8:14].

The following diagram illustrates the concept of an error insensitive zone using a single variable feature x and an output y . In the case of a single variable feature, the error insensitive zone is a band of width 2ϵ (ϵ is known as the insensitive error). The insensitive error plays a similar role to the margin in the SVC.



The visualization of the support vector regression and insensitive error

For the mathematically inclined, the maximization of the margin for nonlinear models introduces a pair of slack variables. As you may remember, the C-support vector classifiers use a single slack variable. The preceding diagram illustrates the minimization formula.

M9: The ϵ -SVR formulation is defined as:

$$\min_{w, \xi, \xi^*} \left\{ \frac{w^T w}{2} + C \sum_{i=0}^{n-1} (\xi_i + \xi_i^*) \right\}$$

$$-\epsilon - \xi_i^* \leq w^T \phi(x_i) + w_0 - y_i \leq \epsilon + \xi_i \quad \forall i$$



Here, ϵ is the insensitive error function.

M10: The ϵ -SVR regression equation is given by:

$$\hat{y}(x) = \sum_{i=0}^{n-1} \alpha_i K(x_i, x) + \hat{w}_0$$

Let's reuse the SVM class to evaluate the capability of the SVR, compared to the linear regression (refer to the *Ordinary least squares regression* section in *Chapter 6, Regression and Regularization*).

SVR versus linear regression

This test consists of reusing the example on single-variate linear regression (refer to the *One-variate linear regression* section in *Chapter 6, Regression and Regularization*). The purpose is to compare the output of the linear regression with the output of the SVR for predicting the value of a stock price or an index. We select the S&P 500 exchange traded fund, SPY, which is a proxy for the S&P 500 index.

The model consists of the following:

- One labeled output: SPY-adjusted daily closing price
- One single variable feature set: the index of the trading session (or index of the values SPY)

The implementation follows a familiar pattern:

1. Define the configuration parameters for the SVR (the `C` cost/penalty function, `GAMMA` coefficient for the RBF kernel, `EPS` for the convergence criteria, and `EPSILON` for the regression insensitive error).
2. Extract the labeled data (the `SPY price`) from the data source (`DataSource`), which is the Yahoo financials CSV-formatted data file.
3. Create the linear regression, `SingleLinearRegression`, with the index of the trading session as the single variable feature and the SPY-adjusted closing price as the labeled output.
4. Create the observations as a time series of indices, `xt`.
5. Instantiate the SVR with the index of trading session as features and the SPY-adjusted closing price as the labeled output.
6. Run the prediction methods for both SVR and the linear regression and compare the results of the linear regression and SVR, `collect`.

The code will be as follows:

```

val path = "resources/data/chap8/SPY.csv"
val C = 12
val GAMMA = 0.3
val EPSILON = 2.5

val config = SVMConfig(new SVRFormulation(C, EPSILON),
                      new RbfKernel(GAMMA)) //45
for {
  price <- DataSource(path, false, true, 1) get close
  (xt, y) <- getLabeledData(price.size) //46
}

```

```

linRg <- SingleLinearRegression[Double](price, y) //47
    svr <- SVM[Double](config, xt, price)
} yield {
    collect(svr, linRg, price)
}

```

The formulation in the configuration has the `SVRFormulation` type (line 45). The `DataSource` class extracts the price of the SPY ETF. The `getLabeledData` method generates the `xt` input features and the `y` labels (or expected values) (line 46):

```

type LabeledData = (Vector[DblArray], DblVector)
def getLabeledData(numObs: Int): Try[LabeledData] = Try {
    val y = Vector.tabulate(numObs)(_.toDouble)
    val xt = Vector.tabulate(numObs)(Array[Double](_))
    (xt, y)
}

```

The single variate linear regression, `SingleLinearRegression`, is instantiated using the `price` input and `y` labels as inputs (line 47).

Finally, the `collect` method executes the two `pfSvr` and `pfLinr` regression partial functions:

```

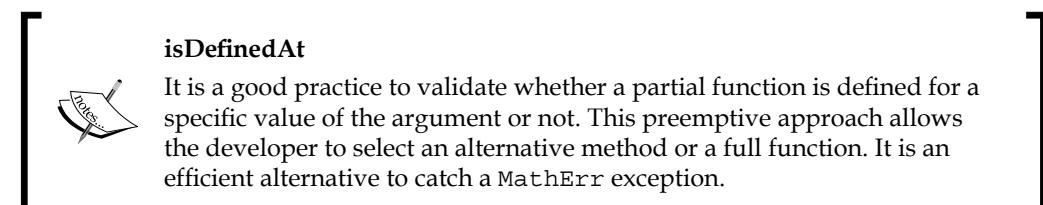
def collect(svr: SVM[Double],
           linr: SingleLinearRegression[Double], price: DblVector) {

    val pfSvr = svr |>
    val pfLinr = linr |>
    for {
        if( pfSvr.isDefinedAt(n.toDouble))
        x <- pfSvr(n.toDouble)
        if( pfLinr.isDefinedAt(n))
        y <- pfLinr(n)
    } yield { ... }
}

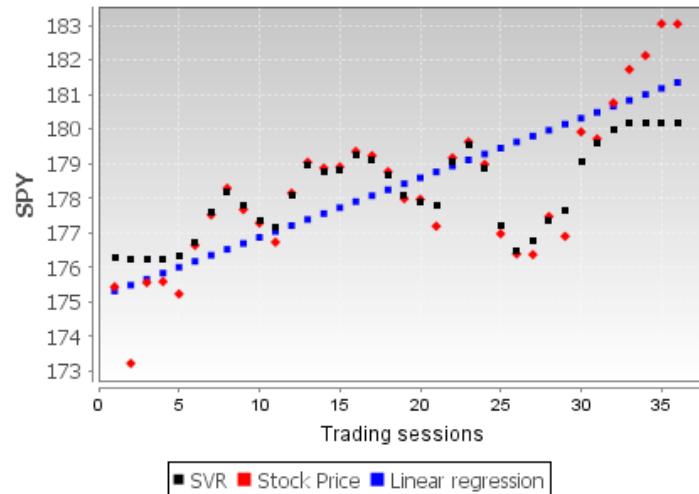
```

isDefinedAt

It is a good practice to validate whether a partial function is defined for a specific value of the argument or not. This preemptive approach allows the developer to select an alternative method or a full function. It is an efficient alternative to catch a `MathErr` exception.

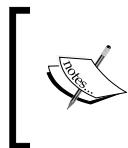


The results are displayed in the following graph, which are generated using the JFreeChart library. The code to plot the data is omitted because it is not essential to the understanding of the application.



A comparative plot of linear regression and SVR

The support vector regression provides a more accurate prediction than the linear regression model. You can also observe that the L_2 regularization term of the SVR penalizes the data points (the SPY price) with a high deviation from the mean of the price. A lower value of C will increase the L_2 -norm penalty factor as $\lambda = 1/C$.



SVR and L_2 regularization

You are invited to run the use case with a different value of C to quantify the impact of the L_2 regularization on the predictive values of the SVR.

There is no need to compare SVR with the logistic regression, as the logistic regression is a classifier. However, the SVM is related to the logistic regression; the hinge loss in the SVM is similar to the loss in the logistic regression [8:15].

Performance considerations

You may have already observed that the training of a model for the support vector regression on a large dataset is time consuming. The performance of the support vector machine depends on the type of optimizer (for example, a sequential minimal optimization) selected to maximize the margin during training:

- A linear model (a SVM without kernel) has an asymptotic time complexity $O(N)$ for training N labeled observations.
- Nonlinear models rely on kernel methods formulated as a quadratic programming problem with an asymptotic time complexity of $O(N^3)$
- An algorithm that uses sequential minimal optimization techniques, such as index caching or elimination of null values (as in LIBSVM), has an asymptotic time complexity of $O(N^2)$ with the worst case scenario (quadratic optimization) of $O(N^3)$
- Sparse problems for very large training sets ($N > 10,000$) also have an asymptotic time of $O(N^2)$

The time and space complexity of the kernelized support vector machine has been receiving a great deal of attention [8:16] [8:17].

Summary

This concludes our investigation of kernel and support vector machines. Support vector machines have become a robust alternative to logistic regression and neural networks for extracting discriminative models from large training sets.

Apart from the unavoidable references to the mathematical foundation of maximum margin classifiers, such as SVMs, you should have developed a basic understanding of the power and complexity of the tuning and configuration parameters of the different variants of SVMs.

As with other discriminative models, the selection of the optimization method for SVMs has a critical impact not only on the quality of the model, but also on the performance (time complexity) of the training and cross-validation process.

The next chapter will describe the third most commonly used discriminative supervised model – artificial neural networks.

9

Artificial Neural Networks

The popularity of neural networks surged in the 90s. They were seen as the silver bullet to a vast number of problems. At its core, a neural network is a nonlinear statistical model that leverages the logistic regression to create a nonlinear distributed model. The concept of artificial neural networks is rooted in biology, with the desire to simulate key functions of the brain and replicate its structure in terms of neurons, activation, and synapses.

In this chapter, you will move beyond the hype and learn the following topics:

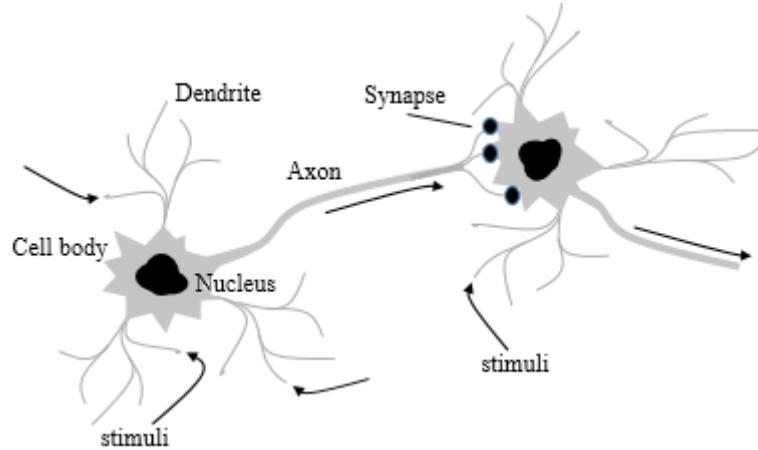
- The concepts and elements of the **multilayer perceptron (MLP)**
- How to train a neural network using error backpropagation
- The evaluation and tuning of MLP configuration parameters
- A full Scala implementation of the MLP classifier
- How to apply MLP to extract correlation models for currency exchange rates
- A brief introduction to **convolutional neural network (CNN)**

Feed-forward neural networks

The idea behind artificial neural networks was to build mathematical and computational models of the natural neural network in the brain. After all, the brain is a very powerful information processing engine that surpasses computers in domains, such as learning, inductive reasoning, prediction and vision, and speech recognition.

The biological background

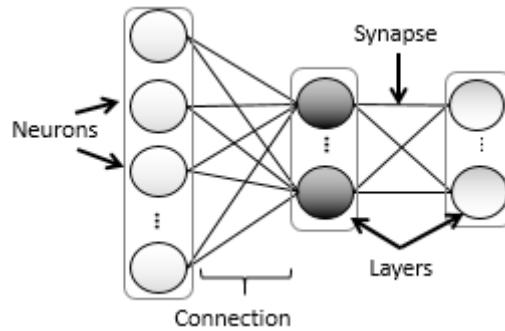
In biology, a neural network is composed of groups of neurons interconnected through synapses [9:1], as shown in the following diagram:



The visualization of biological neurons and synapses

Neuroscientists have been especially interested in understanding how billions of neurons in the brain can interact to provide human beings with parallel processing capabilities. The 60s saw a new field of study emerging, known as **connectionism**. Connectionism marries cognitive psychology, artificial intelligence, and neuroscience. The goal was to create a model for mental phenomena. Although there are many forms of connectionism, the neural network models have become the most popular and the most taught of all connectionism models [9:2].

Biological neurons communicate with electrical charges known as **stimuli**. This network of neurons can be represented as a simple schematic, as follows:



The representation of neuron layers, connections, and synapses

This representation categorizes groups of neurons as layers. The terminology used to describe the natural neural networks has a corresponding nomenclature for the artificial neural network.

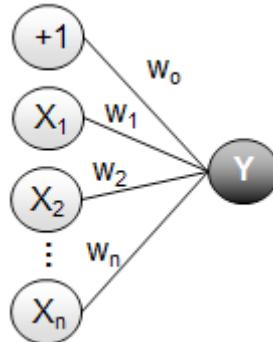
The biological neural network	The artificial neuron network
Axon	Connection
Dendrite	Connection
Synapse	Weight
Potential	Weighted sum
Threshold	Bias weight
Signal, Stimulus	Activation
Group of neurons	Layer of neurons

In the biological world, stimuli do not propagate in any specific direction between neurons. An artificial neural network can have the same degree of freedom. The most commonly used artificial neural networks by data scientists have a predefined direction: from the input layer to output layers. These neural networks are known as a **feed-forward neural network (FFNN)**.

Mathematical background

In the previous chapter, you learned that support vector machines have the ability to formulate the training of a model as a nonlinear optimization for which the objective function is convex. A convex objective function is fairly straightforward to implement. The drawback is that the kernelization of the SVM may result in a large number of basis functions (or model dimensions). Refer to the *The kernel trick* section under *Support vector machines* in *Chapter 8, Kernel Models and Support Vector Machines*. One solution is to reduce the number of basis functions through parameterization, so these functions can adapt to different training sets. Such an approach can be modeled as a FFNN, known as the multilayer perceptron [9:3].

The linear regression can be visualized as a simple connectivity model using neurons and synapses, as follows:



A two-layer neural network

The feature $x_0 = +1$ is known as the **bias input** (or the bias element), which corresponds to the intercept in the classic linear regression.

As with support vector machines, linear regression is appropriate for observations that can be linearly separable. The real world is usually driven by a nonlinear phenomenon. Therefore, the logistic regression is naturally used to compute the output of the perceptron. For a set of input variable $x = \{x_i\}_{0,n}$ and the weights $w = \{w_i\}_{1,n}$, the output y is computed as follows (**M1**):

$$y = \sigma(w_0 + w^T x) = \frac{1}{1 + e^{-(w_0 + w^T x)}}$$

A FFNN can be regarded as a stack of layers of logistic regression with the output layer as a linear regression.

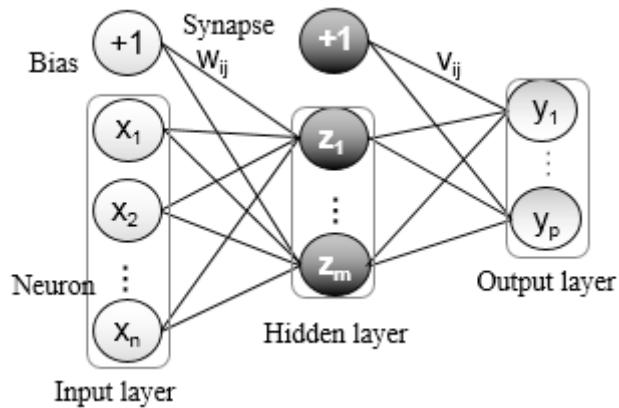
The value of the variables in each hidden layer is computed as the sigmoid of the dot product of the connection weights and the output of the previous layer. Although interesting, the theory behind artificial neural networks is beyond the scope of this book [9:4].

The multilayer perceptron

The perceptron is a basic processing element that performs a binary classification by mapping a scalar or vector to a binary (or XOR) value $\{\text{true}, \text{false}\}$ or $\{-1, +1\}$. The original perceptron algorithm was defined as a single layer of neurons for which each value x_i of the feature vector is processed in parallel and generates a single output y . The perceptron was later extended to encompass the concept of an activation function.

The single layer perceptrons are limited to process a single linear combination of weights and input values. Scientists found out that adding intermediate layers between the input and output layers enable them to solve more complex classification problems. These intermediate layers are known as **hidden layers** because they interface only with other perceptrons. Hidden nodes can be accessed only through the input layer.

From now on, we will use a three-layered perceptron to investigate and illustrate the properties of neural networks, as shown here:



A three-layered perceptron

The three-layered perceptron requires two sets of weights: w_{ij} to process the output of the input layer to the hidden layer and v_{ij} between the hidden layer and the output layer. The intercept value w_0 , in both linear and logistic regression, is represented with +1 in the visualization of the neural network ($w_0 \cdot 1 + w_1 \cdot x_1 + w_2 \cdot x_2 + \dots$).

A FFNN without a hidden layer



A FFNN without a hidden layer is similar to a linear statistical model. The only transformation or connection between the input and output layer is actually a linear regression. A linear regression is a more efficient alternative to the FFNN without a hidden layer.

The description of the MLP components and their implementations rely on the following stages:

1. An overview of the software design.
2. A description of the MLP model components.
3. The implementation of the four-step training cycle.
4. The definition and implementation of the training strategy and the resulting classifier.

Terminology



Artificial neural networks encompass a large variety of learning algorithms, the multilayer perceptron being one of them. Perceptrons are indeed components of a neural network organized as the input, output, and hidden layers. This chapter is dedicated to the multilayer perceptron with hidden layers. The terms "neural network" and "multilayer perceptron" are used interchangeably.

The activation function

The perceptron is represented as a linear combination of weights w_i and input values x_i processed by the output unit activation function h , as shown here (**M2**):

$$\hat{y} = h \left(w_0 + \sum_{i=1}^n w_i x_i \right) = h(w_0 + w^T x)$$

The output activation function h has to be continuous and differentiable for a range of value of the weights. It takes different forms depending on the problems to be solved, as mentioned here:

- An identity for the output layer (linear formula) of the regression mode
- The sigmoid σ for hidden layers and output layers of the binomial classifier

- Softmax for the multinomial classification
- The hyperbolic tangent, $tanh$, for the classification using zero mean

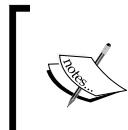
The softmax formula is described in *Step 1 – input forward propagation* under *Training epoch*.

The network topology

The output layers and hidden layers have a computational capability (dot product of weights, inputs, and activation functions). The input layer does not transform data. An n -layer neural network is a network with n computational layers. Its architecture consists of the following components:

- one input layer
- $n-1$ hidden layer
- one output layer

A **fully connected neural network** has all its input nodes connected to hidden layer neurons. Networks are characterized as **partially connected neural networks** if one or more of their input variables are not processed. This chapter deals with a fully connected neural network.



Partially connected networks

Partially connected networks are not as complex as they seem. They can be generated from fully connected networks by setting some of the weights to zero.

The structure of the output layer is highly dependent on the type of problems (regression or classification) you need to solve, also known as the operating mode of the multilayer perceptron. The type of problem at hand defines the number of output nodes [9:5]. Consider the following examples:

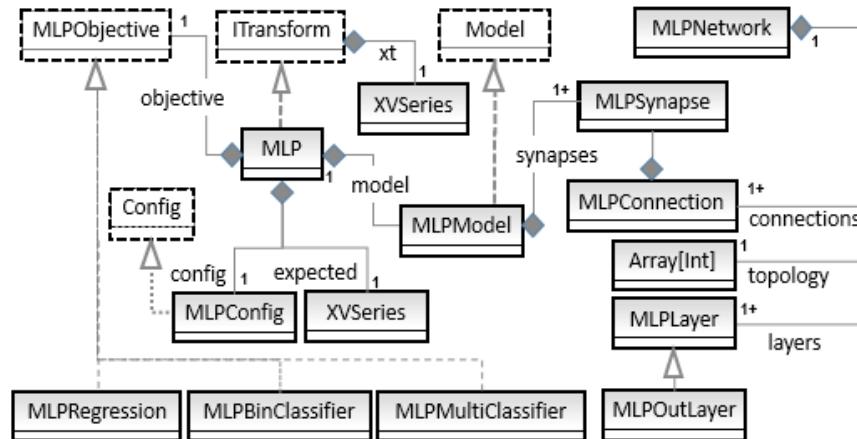
- A one-variate regression has one output node whose value is a real number $[0, 1]$
- A multivariate regression with n variables has n real output nodes
- A binary classification has one binary output node $\{0, 1\}$ or $\{-1, +1\}$
- A multinomial or K-class classification has K binary output nodes

Design

The implementation of the MLP classifier follows the same pattern as previous classifiers (refer to the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*):

- An `MLPNetwork` connectionist network is composed of a layer of neurons of the `MLPLayer` type, connected by synapses of the `MLPSynapse` type contained by a connector of the `MLPConnection` type.
- All of the configuration parameters are encapsulated into a single `MLPConfig` configuration class.
- A model, `MLPModel`, consists of a sequence of connection synapses.
- The `MLP` multilayer perceptron class is implemented as a data transformation, `ITransform`, for which the model is automatically extracted from a training set with labels.
- The `MLP` multilayer perceptron class takes four parameters: a configuration, a features set or time series of the `XVSeries` type, a labeled dataset of the `XVSeries` type, and an activation function of the `Function1[Double, Double]` type.

The software components of the multilayer perceptron are described in the following UML class diagram:



A UML class diagram for the multilayer perceptron

The class diagram is a convenient navigation map used to understand the role and relation of the Scala classes used to build an MLP. Let's start with the implementation of the MLP network and its components. The UML diagram omits the helper traits or classes such as `Monitor` or the Apache Commons Math components.

Configuration

The `MLPConfig` configuration of the multilayer perceptron consists of the definition of the network configuration with its hidden layers, the learning and training parameters, and the activation function:

```
case class MLPConfig(
    val alpha: Double, //1
    val eta: Double,
    val numEpochs: Int,
    val eps: Double,
    val activation: Double => Double) extends Config { //1
}
```

For the sake of readability, the name of the configuration parameters matches the symbols defined in the mathematical formulation (line 1):

- `alpha`: This is the momentum factor α that smoothes the computation of the gradient of the weights for online training. The momentum factor is used in the mathematical expression **M10** in *Step 2 – error backpropagation* under *Training epoch*.
- `eta`: This is the learning rate η used in the gradient descent. The gradient descent updates the weights or parameters of a model by the quantity, $\eta \cdot (predicted - expected).input$, as described in the mathematical formulation **M9** in *Step 2 – error backpropagation* section under *The training epoch*. The gradient descent was introduced in *Let's kick the tires* in *Chapter 1, Getting Started*.
- `numEpochs`: This is the maximum number of epochs (or cycles or episodes) allowed for training the neural network. An epoch is the execution of the error backpropagation across the entire observation set.
- `eps`: This is the convergence criteria used as an exit condition for the training of the neural network when $error < eps$.
- `activation`: This is the activation function used for nonlinear regression applied to hidden layers. The default function is the sigmoid (or the hyperbolic tangent) introduced for the logistic regression (refer to the *Logistic function* section in *Chapter 6, Regression and Regularization*).

Network components

The training and classification of an MLP model relies on the network architecture. The `MLPNetwork` class is responsible for creating and managing the different components and the topology of the network, that is layers, synapses, and connections.

The network topology

The instantiation of the `MLPNetwork` class requires a minimum set of two parameters with an instance of the model as an optional third argument (line 2):

- An MLP execution configuration, `config`, introduced in the previous section
- A topology defined as an array of the number of nodes for each layer: input, hidden, and output layers.
- A model with the `Option[MLPModel]` type if it has already been generated through training, or `None` otherwise
- An implicit reference to the operating mode of the MLP

The code is as follows:

```
class MLPNetwork(config: MLPConfig,  
  topology: Array[Int],  
  model: Option[MLPModel] = None)  
  (implicit mode: MLPMode) { //2  
  
  val layers = topology.zipWithIndex.map { case(t, n) =>  
    if (topology.size != n+1)  
      MLPLayer(n, t+1, config.activation)  
    else MLPOutLayer(n, t)  
  } //3  
  val connections = zipWithShift1(layers, 1).map{case(src, dst) =>  
    new MLPConnection(config, src, dst, model)} //4  
  
  def trainEpoch(x: DblArray, y: DblArray): Double //5  
  def getModel: MLPModel //6  
  def predict(x: DblArray): DblArray //7  
}
```

A MLP network has the following components, which are derived from the topology array:

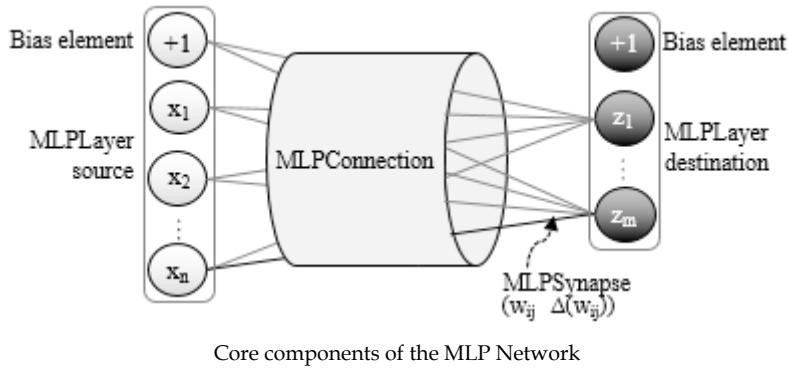
- Multiple layers of the `MLPLayers` class (line 3)
- Multiple connections of the `MLPConnection` class (line 4)

The topology is defined as an array of number of nodes per layer, starting with the input nodes. The array indices follow the forward path within the network. The size of the input layer is automatically generated from the observations as the size of the features vector. The size of the output layer is automatically extracted from the size of the output vector (line 3).

The constructor for `MLPNetwork` creates a sequence of layers by assigning and ordering an `MLPLayer` instance to each entry in the topology (line 3). The constructor creates *number of layers - 1* interlayer connections of the `MLPConnection` type (line 4). The `zipWithShift1` method of the `XTSeries` object zips a time series with its duplicated shift by one element.

The `trainEpoch` method (line 5) implements the training of this network for a single pass of the entire set of observations (refer to the *Putting it all together* section under *The training epoch*). The `getModel` method retrieves the model (synapses) generated through training of the MLP (line 6). The `predict` method computes the output value generated from the network using the forward propagation algorithm (line 7).

The following diagram visualizes the interaction between the different components of a model: `MLPLayer`, `MLPConnection`, and `MLPSynapse`:



Input and hidden layers

First, let's start with the definition of the `MLPLayer` layer class, which is completely specified by its position (or rank) `id` in the network and the number of nodes, `numNodes`, it contains:

```
class MLPLayer(val id: Int, val numNodes: Int,
    val activation: Double => Double) //8
    (implicit mode: MLPMode){ //9

    val output = Array.fill(numNodes)(1.0) //10

    def setOutput(xt: DblArray): Unit =
        xt.copyToArray(output, 1) //11
    def activate(x: Double): Double = activation(x) //12
    def delta(loss: DblArray, srcOut: DblArray,
        synapses: MLPConnSynapses): Delta //13
    def setInput(_x: DblArray): Unit //14
}
```

The `id` parameter is the order of the layer (0 for input, 1 for the first hidden layer, and $n - 1$ for the output layer) in the network. The `numNodes` value is the number of elements or nodes, including the bias element, in this layer. The `activation` function is the last argument of the layer given a user-defined mode or objective (line 8). The `operatingMode` has to be provided implicitly prior to the instantiation of a layer (line 9).

The output vector for the layer is an uninitialized array of values updated during the forward propagation. It initializes the bias value with the value 1.0 (line 9). The matrix of difference of weights, `deltaMatrix`, associated with the output vector (line 10) is updated using the error backpropagation algorithm, as described in the *Step 2 – error back propagation* section under *The training epoch*. The `setOutput` method initializes the output values for the output and hidden layers during the backpropagation of the error on the output of the network (*expected – predicted*) values (line 11).

The `activate` method invokes the activation method (*tanh*, *sigmoid*, ...) defined in the configuration (line 12).

The `delta` method computes the correction to be applied to each weight or synapses, as described in the *Step 2 – error back propagation* section under *The training epoch* (line 13).

The `setInput` method initializes the `output` values for the nodes of the input and hidden layers, except the bias element, with the value `x` (line 14). The method is invoked during the forward propagation of input values:

```
def setInput(x: DblVector): Unit =
  x.copyToArray(output, output.length -x.length)
```

The methods of the `MLPLayer` class for the input and hidden layers are overridden for the output layer of the `MLPOutLayer` type.

The output layer

Contrary to the hidden layers, the output layer does not have either an activation function or a bias element. The `MLPOutLayer` class has the following arguments: the order `id` in the network (as the last layer of the network) and the number, `numNodes`, of the output or nodes (line 15):

```
class MLPOutLayer(id: Int, numNodes: Int)
  (implicit mode: MLP.MLPMode) //15
  extends MLPLayer(id, numNodes, (x: Double) => x) {

  override def numNonBias: Int = numNodes
  override def setOutput(xt: DblArray): Unit =
    obj(xt).copyToArray(output)
```

```
override def delta(loss: DblArray, srcOut: DblArray,
  synapses: MLPConnSynapses): Delta
...
}
```

The `numNonBias` method returns the actual number of output values from the network. The implementation of the `delta` method is described in the *Step 2 – error back propagation* section under *The training epoch*.

Synapses

A synapse is defined as a pair of real (a floating point) values:

- The weight w_{ij} of the connection from the neuron i of the previous layer to the neuron j
- The weights' adjustment (or gradient of weights) Δw_{ij}

Its type is defined as `MLPSynapse`, as shown here:

```
type MLPSynapse = (Double, Double)
```

Connections

The connections are instantiated by selecting two consecutive layers of an index n (with respect to $n + 1$) as a source (with respect to destination). A connection between two consecutive layers implements the matrix of synapses as the $(w_{ij}, \Delta w_{ij})$ pairs. The `MLPConnection` instance is created with the following parameters (line 16):

- Configuration parameters, `config`
- The source layer, sometimes known as the ingress layer, `src`
- The `dst` destination (or egress) layer
- A reference to the `model` if it has already been generated through training or `None` if the model has not been trained
- An implicitly defined operating `mode` or objective `mode`

The `MLPConnection` class is defined as follows:

```
type MLPConnSynapses = Array[Array[MLPSynapse]]

class MLPConnection(config: MLPConfig,
  src: MLPLayer,
  dst: MLPLayer,
  model: Option[MLPModel]) //16
```

```
(implicit mode: MLP.MLPMode) {  
  
    var synapses: MLPConnSynapses //17  
    def connectionForwardPropagation: Unit //18  
    def connectionBackpropagation(delta: Delta): Delta //19  
    ...  
}
```

The last step in the initialization of the MLP algorithm is the selection of the initial (usually random) values of the weights (synapse) (line 17).

The MLPConnection methods implement the forward propagation of weights' computation for this connectionForwardPropagation connection (line 18) and the backward propagation of the delta error during training connectionBackpropagation (line 19). These methods are described in the next section related to the training of the MLP model.

The initialization weights

The initialization values for the weights depends is domain specific. Some problems require a very small range, less than $1e-3$, while others use the probability space $[0, 1]$. The initial values have an impact on the number of epochs required to converge toward an optimal set of weights [9:6].

Our implementation relies on the sigmoid activation function and uses the range $[0, BETA/sqrt(numOutputs + 1)]$ (line 20). However, the user can select a different range for random values, such as $[-r, +r]$ for the *tanh* activation function. The weight of the bias is obviously defined as $w_0=+1$, and its weight adjustment is initialized as $\Delta w_0 = 0$, as shown here (line 20):

```
var synapses: MLPConnSynapses = if(model == None) {  
    val max = BETA/Math.sqrt(src.output.length+1.0) //20  
    Array.fill(dst.numNonBias) (  
        Array.fill(src.numNodes) ((Random.nextDouble*max, 0.00))  
    )  
} else model.get.synapses(src.id) //21
```

The connection derives its weights or synapses from a model (line 21) if it has already been created through training.

The model

The `MLPNetwork` class defines the topological model of the multilayer perceptron. The weights or synapses are the attributes of the model of the `MLPModel` type generated through training:

```
case class MLPModel(
  val synapses: Vector[MLPConnSynapses]) extends Model
```

The model can be stored in a simple key-value pair JSON, CVS, or sequence file.

Encapsulation and the model factory

The network components: connections, layers, and synapses are implemented as top-level classes for the sake of clarity. However, there is no need for the model to expose its inner workings to the client code. These components should be declared as an inner class to the model. A factory design pattern would be perfectly appropriate to instantiate an `MLPNetwork` instance dynamically [9:7].

Once initialized, the MLP model is ready to be trained using a combination of forward propagation, output error back propagation, and iterative adjustment of weights and gradients of weights.

Problem types (modes)

There are three distinct types of problems or operating modes associated with the multilayer perceptron:

- The **binomial classification** (binary) with two classes and one output
- The **multinomial classification** (multiclass) with n classes and output
- Regression

Each operating mode has distinctive error, hidden layer, and output layer activation functions, as illustrated in the following table:

Operating modes	Error function	Hidden layer activation function	Output layer activation function
Binomial classification	Cross-entropy	Sigmoid	Sigmoid
Multinomial classification	Sum of squares error or mean squared error	Sigmoid	Softmax
Regression	Sum of squares error or mean squared error	Sigmoid	Linear

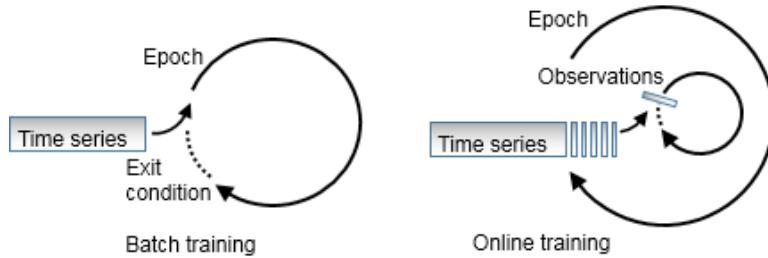
A table for operating modes of the multilayer perceptron

The cross-entropy is described by the mathematical expressions **M6** and **M7** and the softmax uses the formula **M8** in the *Step 1 – input forward propagation* section under *The training epoch*.

Online training versus batch training

One important issue is to find a strategy to conduct the training of a time series as an ordered sequence of data. There are two strategies to create an MLP model for a time series:

- **Batch training:** The entire time series is processed at once as a single input to the neural network. The weights (synapses) are updated at each epoch using the sum of the squared errors on the output of the time series. The training exits once the sum of the squared errors meets the convergence criteria.
- **Online training:** The observations are fed to the neural network one at a time. Once the time series has been processed, the total of the sum of the squared errors (sse) for the time series for all the observations are computed. If the exit condition is not met, the observations are reprocessed by the network.



An illustration on online and batch training

An online training is faster than batch training because the convergence criterion has to be met for each data point, possibly resulting in a smaller number of epochs [9:12]. Techniques such as the momentum factor, which is described earlier, or any adaptive learning scheme improves the performance and accuracy of the online training methodology.

The online training strategy is applied to all the test cases of this chapter.

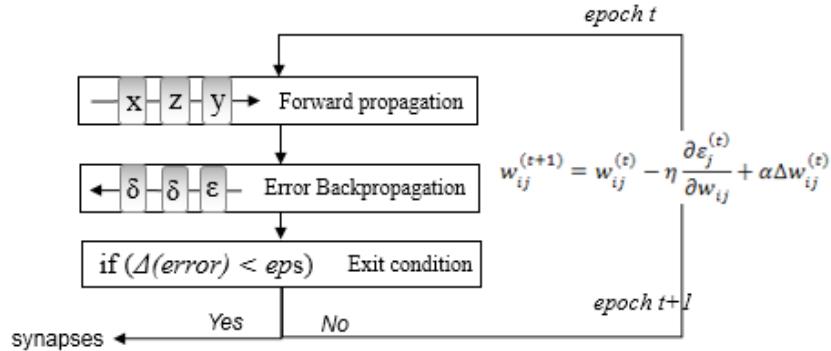
The training epoch

The training of the model processes the training observations iteratively multiple times. A training cycle or iteration is known as an **epoch**. The order of observations is shuffled for each epoch. The three steps of the training cycle are as follows:

1. Forward the propagation of the input value for a specific epoch.
2. Computation and backpropagation of the output error.
3. Evaluate the convergence criteria and exit if the criteria is met.

The computation of the network weights during training can use the difference between labeled data and actual output for each layer. But this solution is not feasible because the output of the hidden layers is actually unknown. The solution is to propagate the error on the output values (predicted values) backward to the input layer through the hidden layers, if an error is defined.

The three steps of the training cycle or training epoch are summarized in the following diagram:



An iterative implementation of the training for MLP

Let's apply the three steps of a training epoch in the `trainEpoch` method of the `MLPNetwork` class using a simple `foreach` Scala higher order function, as shown here:

```

def trainEpoch(x: DblArray, y: DblArray): Double = {
    layers.head.setInput(x) //22
    connections.foreach(_.connectionForwardPropagation) //23

    val err = mode.error(y, layers.last.output)
    val bckIterator = connections.reverseIterator

    var delta = Delta(zipToArray(y, layers.last.output)(diff)) //24
    bckIterator.foreach(iter =>
        delta = iter.connectionBackpropagation(delta)) //25
    err //26
}
    
```

You can certainly recognize the first two stages of the training cycle: the forward propagation of the input and the backpropagation of the error of the online training of a single epoch.

The execution of the training of the network for one epoch, `trainEpoch`, initializes the input layer with observations, `x` (line 22). The input values are propagated through the network by invoking `connectionForwardPropagation` for each connection (line 23). The `delta` error is initialized from the values in the output layer and the expected values, `y` (line 24).

The training method iterates through the connections backward to propagate the error through each connection by invoking the `connectionBackpropagation` method on the backward iterator, `bckIterator` (line 25). Finally, the training method returns the cumulative error, mean square error, or cross entropy, according to the operating mode (line 26).

This approach is not that different than the beta (or backward) pass in the hidden Markov model, which was covered in the *Beta – the backward pass* section in *Chapter 7, Sequential Data Models*.

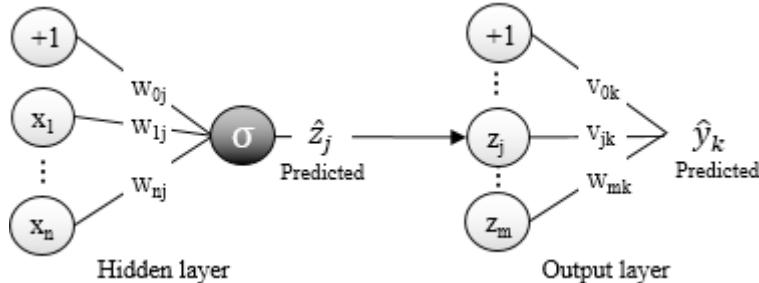
Let's take a look at the implementation of the forward and backward propagation algorithm for each type of connection:

- An input or hidden layer to a hidden layer
- A hidden layer to an output layer

Step 1 – input forward propagation

As mentioned earlier, the output values of a hidden layer are computed as the sigmoid or hyperbolic tangent of the dot product of the weights w_{ij} and the input values x_i .

In the following diagram, the MLP algorithm computes the linear product of the weights w_{ij} and input x_i for the hidden layer. The product is then processed by the activation function σ (the sigmoid or hyperbolic tangent). The output values z_j are then combined with the weights v_{jk} of the output layer that doesn't have an activation function:



The distribution of weights in MLP hidden and output layers

The mathematical formulation of the output of a neuron j is defined as a composition of the activation function and the dot product of the weights w_{ij} and input values x_i :

M3: The computation (or prediction) of the output layer from the output values z_j of the preceding hidden layer and the weights v_{kj} is defined as:

$$\tilde{y}_k = v_{0j} + \sum_{j=1}^m v_{kj} z_j$$



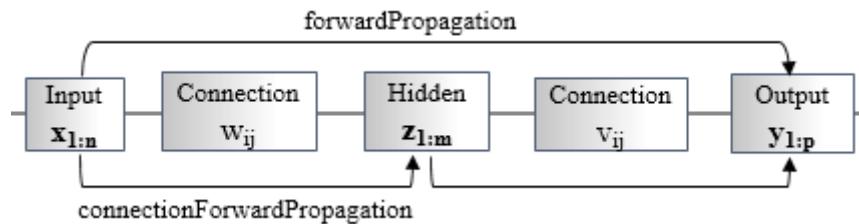
M4: The estimation of the output values for a binary classification with an activation function σ is defined as:

$$\tilde{z}_j = \sigma\left(w_{0j} + \sum_{i=1}^m w_{ij} x_i\right) = \frac{1}{1 + e^{-w_{0j} - \sum_{i=1}^m w_{ij} x_i}}$$

As seen in the network architecture section, the output values for the multinomial (or multiclass) classification with more than two classes are normalized using an exponential function, as described in the following *Softmax* section.

The computational flow

The computation of the output values y from the input x is known as the input forward propagation. For the sake of simplicity, we represent the forward propagation between layers with the following block diagram:



A computation model of the input forward propagation

The preceding diagram conveniently illustrates a computational model for the input forward propagation, as the programmatic relation between the source and destination layers and their connectivity. The input x is propagated forward through each connection.

The `connectionForwardPropagation` method computes the dot product of the weights and the input values and applies the activation function, in the case of hidden layers, for each connection. Therefore, it is a member of the `MLPConnection` class.

The forward propagation of input values across the entire network is managed by the MLP algorithm itself. The forward propagation of the input value is used in the classification or prediction $y = f(x)$. It depends on the value weights w_{ij} and v_{ij} that need to be estimated through training. As you may have guessed, the weights define the model of a neural network similar to the regression models. Let's take a look at the `connectionForwardPropagation` method of the `MLPConnection` class:

```
def connectionForwardPropagation: Unit = {
    val _output = synapses.map(x => {
        val dot = inner(src.output, x.map(_.value)) //27
        dst.activate(dot) //28
    })
    dst.setOutput(_output) //29
}
```

The first step is to compute the linear inner (or dot) product (refer to the *Time series in Scala* section in *Chapter 3, Data Preprocessing*) of the output, `_output`, of the current source layer for this connection and the synapses (weights) (line 27). The activation function is computed by applying the `activate` method of the destination layer to the dot product (line 28). Finally, the computed value, `_output`, is used to initialize the output for the destination layer (line 29).

Error functions

As mentioned in the *Problem types (modes)* section, there are two approaches to compute the error or loss on the output values:

- The sum of the squared errors between expected and predicted output values, as defined in the **M5** mathematical expression
- Cross-entropy of expected and predicted values described in the **M6** and **M7** mathematical formulas

M5: The sum of the squared errors ε and mean square error for predicted values \tilde{y} and expected values y are defined as:

$$\varepsilon = \frac{1}{2} \sum_{i=0}^{n-1} \sum_{j=0}^{k-1} (y_{ij} - \tilde{y}_{ij})^2 \quad \bar{\varepsilon} = \frac{\varepsilon}{n}$$

M6: Cross entropy for a single output value y is defined as:



$$ce = - \sum_{i=0}^{n-1} \{y_i \log(\tilde{y}_i) + (1 - y_i) \cdot \log(1 - \tilde{y}_i)\}$$

M7: Cross entropy for a multivariable output vector y is defined as:

$$ce = - \sum_{i=0}^{n-1} \sum_{j=0}^{k-1} \tilde{y}_{ij} \log(y_{ij})$$

The sum of squared errors and mean squared error functions have been described in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*.

The `crossEntropy` method of the `XTSeries` object for a single variable is implemented as follows:

```
def crossEntropy(x: Double, y: Double): Double =
  -(x * Math.log(y) + (1.0 - x) * Math.log(1.0 - y))
```

The computation of the cross entropy for multiple variable features as a signature is similar to the single variable case:

```
def crossEntropy(xt: DblArray, yt: DblArray): Double =
  yt.zip(xt).aggregate(0.0)({ case (s, (y, x)) =>
    s - y * Math.log(x), _ + _ })
```

Operating modes

In the *network architecture* section, you learned that the structure of the output layer depends on the type of problems that need to be resolved, also known as operating modes. Let's encapsulate the different operating modes (binomial, multinomial classification, and regression) into a class hierarchy, implementing the `MLPMode` trait. The `MLPMode` trait has two methods that are specific to the type of the problem:

- `apply`: This is the transformation applied to the output values
- `error`: This is the computation of the cumulative error for the entire observation set

The code will be as follows:

```
trait MLPMode {
    def apply(output: DblArray): DblArray //30
    def error(labels: DblArray, output: DblArray): Double =
        mse(labels, output) //31
}
```

The `apply` method applies a transformation to the output layer, as described in the last column of the operating modes table (line 30). The `error` function computes the cumulative error or loss in the output layer for all the observations, as described in the first column of the operating modes table (line 31).

The transformation in the output layer of the `MLPBinClassifier` binomial (two-class) classifier consists of applying the `sigmoid` function to each output value (line 32). The cumulative error is computed as the cross entropy of the expected output, labels, and the predicted output (line 33):

```
class MLPBinClassifier extends MLPMode {
    override def apply(output: DblArray): DblArray =
        output.map(sigmoid(_)) //32
    override def error(labels: DblArray,
                      output: DblArray): Double =
        crossEntropy(labels.head, output.head) //33
}
```

The regression mode for the multilayer perceptron is defined according to the operating modes table in the *Problem types (modes)* section:

```
class MLPRegression extends MLPMode {
    override def apply(output: DblArray): DblArray = output
}
```

The multinomial classifier mode is defined by the `MLPMultiClassifier` class. It uses the `softmax` method to boost the output with the highest value, as shown in the following code:

```
class MLPMultiClassifier extends MLPMode {
    override def apply(output: DblArray): DblArray = softmax(output)
}
```

The `softmax` method is applied to the actual output value, not the bias. Therefore, the first node $y(0) = +1$ has to be dropped before applying the `softmax` normalization.

Softmax

In the case of a classification problem with K classes ($K > 2$), the output has to be converted into a probability $[0, 1]$. For problems that require a large number of classes, there is a need to boost the output y_k with the highest value (or probability). This process is known as **exponential normalization** or softmax [9:8].

Note...

M8: The softmax formula for the multinomial ($K > 2$) classification is as follows:

$$\hat{y}_k = \frac{e^{-\hat{y}_k}}{\sum_i e^{-\hat{y}_i}}$$

Here is the simple implementation of the softmax method of the `MLPMultiClassifier` class:

```
def softmax(y: DblArray): DblArray = {
    val softmaxValues = new DblArray(y.size)
    val expY = y.map( Math.exp(_) ) //34
    val expYSum = expY.sum //35

    expY.map( _ /expYSum) .copyToArray(softmaxValues, 1) //36
    softmaxValues
}
```

The softmax method implements the M8 mathematical expression. First, the method computes the `expY` exponential values of the output values (line 34). The exponentially transformed outputs are then normalized by their sum, `expYSum`, (line 35) to generate the array of the `softmaxValues` output (line 36). Once again, there is no need to update the bias element $y(0)$.

The second step in the training phase is to define and initialize the matrix of delta error values to be back propagated between layers from the output layer back to the input layer.

Step 2 – error backpropagation

The error backpropagation is an algorithm that estimates the error for the hidden layer in order to compute the change in weights of the network. It takes the sum of squared errors of the output as the input.



The convention for computing the cumulative error

Some authors refer to the backpropagation as a training methodology for an MLP, which applies the gradient descent to the output error defined as either the sum of squared errors, or the mean squared error for multinomial classification or regression. In this chapter, we keep the narrower definition of the backpropagation as the backward computation of the sum of squared errors.

Weights' adjustment

The connection weights Δv and Δw are adjusted by computing the sum of the derivatives of the error, over the weights scaled with a learning factor. The gradient of weights are then used to compute the error of the output of the source layer [9:9].

The simplest algorithm to update the weights is the gradient descent [9:10]. The batch gradient descent was introduced in *Let's kick the tires in Chapter 1, Getting Started*.

The gradient descent is a very simple and robust algorithm. However, it can be slower in converging toward a global minimum than the conjugate gradient or the quasi-Newton method (refer to the *Summary of optimization techniques* section in the *Appendix A, Basic Concepts*).

There are several methods available to speed up the convergence of the gradient descent toward a minimum, such as the momentum factor and adaptive learning coefficient [9:11].

Large variations of the weights during training increase the number of epochs required for the model (connection weights) to converge. This is particularly true for a training strategy known as online training. The training strategies are discussed in the next section. The momentum factor α is used for the remaining section of the chapter.

M9: The learning rate

The computation of neural network weights using the gradient descent is as follows:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \eta \frac{\partial \varepsilon_j^{(t)}}{\partial w_{ij}}$$



M10: The learning rate and momentum factor

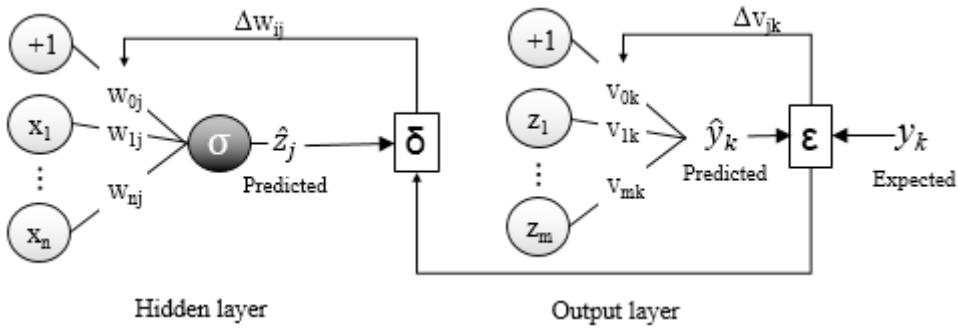
The computation of neural network weights using the gradient descent method with the momentum coefficient α is as follows:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \eta \frac{\partial \varepsilon_j^{(t)}}{\partial w_{ij}} + \alpha \Delta w_{ij}^{(t)}$$

The simplest version of the gradient descent algorithm (**M9**) is selected by simply setting the momentum factor α to zero in the generic (**M10**) mathematical expression.

The error propagation

The objective of the training of a perceptron is to minimize the loss or cumulative error for all the input observations as either the sum of squared errors or the cross entropy as computed at the output layer. The error ε_k for each output neuron y_k is computed as the difference between a predicted output value and label output value. The error cannot be computed on output values of the hidden layers z_j because the label values for those layers are unknown:



An illustration of the back-propagation algorithm

In the case of the sum of squared errors, the partial derivative of the cumulative error over each weight of the output layer is computed as the composition of the derivative of the square function and the derivative of the dot product of weights and the input z .

As mentioned earlier, the computation of the partial derivative of the error over the weights of the hidden layer is a bit tricky. Fortunately, the mathematical expression for the partial derivative can be written as the product of three partial derivatives:

- The derivative of the cumulative error ε over the output value y_k
- The derivative of the output value y_k over the hidden value z_j , knowing that the derivative of a sigmoid σ is $\sigma(1 - \sigma)$
- The derivative of the output of the hidden layer z_j over the weights w_{ij}

The decomposition of the partial derivative produces the following formulas for updating the synapses' weights for the output and hidden neurons by propagating the error (or loss) ε .

Output weights' adjustment

M11: The computation of delta δ and weight adjustment Δv for the output layer with the predicted value \tilde{y} and expected value y , and output z of the hidden layer is as follows:

$$\delta_{ih} = (\tilde{y}_i - y_i) \cdot z_h \quad \Delta v_{ih} = -\eta \cdot \delta_{ih}$$

Hidden weights' adjustment

M12: The computation of delta δ and weight adjustment Δw for the hidden layer with the predicted value \tilde{y} and expected value y , output z of the hidden layer, and the input value x is as follows:

$$\delta_{hi} = \sum_{j=0}^{k-1} \{(\tilde{y}_j - y_j) \cdot v_{jh}\} \cdot z_h (1 - z_h) \cdot x_i \quad \Delta w_{hi} = -\eta \delta_{hi}$$

The matrix δ_{ij} is defined by the `delta` matrix in the `Delta` class. It contains the basic parameters to be passed between layers, traversing the network from the output layer back to the input layer. The parameters are as follows:

- Initial loss or error computed at the output layer
- Matrix of the `delta` values from the current connection
- Weights or synapses of the downstream connection (or connection between the destination layer and the following layer)

The code will be as follows:

```
case class Delta(val loss: DblArray,
    val delta: DblMatrix = Array.empty[DblArray],
    val synapses: MLPConnSynapses = Array.empty[Array[MLPSynapse]] )
```

The first instance of the `Delta` class is generated for the output layer using the expected values y , then propagated to the preceding hidden layer in the `MLPNetwork.trainEpoch` method (line 24):

```
val diff = (x: Double, y: Double) => x - y
Delta(zipToArray(y, layers.last.output)(diff))
```

The **M11** mathematical expression is implemented by the `delta` method of the `MLPOutLayer` class:

```
def delta(error: DblArray, srcOut: DblArray,
    synapses: MLPConnSynapses): Delta = {

    val deltaMatrix = new ArrayBuffer[DblArray] //34
    val deltaValues = error.:(deltaMatrix) ( (m, l) => {
        m.append( srcOut.map( _*l) )
        m
    }) //35
    new Delta(error, deltaValues.toArray, synapses) //36
}
```

The method generates the matrix of delta values associated with the output layer (line 34). The **M11** formula is actually implemented by the fold over the `srcOut` output value (line 35). The new delta instances are returned to the `trainEpoch` method of `MLPNetwork` and backpropagated to the preceding hidden layer (line 36).

The `delta` method of the `MLPLayer` class implements the **M12** mathematical expression:

```
def delta(oldDelta: DblArray, srcOut: DblArray,
    synapses: MLPConnSynapses): Delta = {

    val deltaMatrix = new ArrayBuffer[(Double, DblArray)]
    val weights = synapses.map(_.map(_.-_1))
        .transpose.drop(1) //37

    val deltaValues = output.drop(1)
        .zipWithIndex.:(deltaMatrix){ // 38
        case (m, (zh, n)) => {
```

```

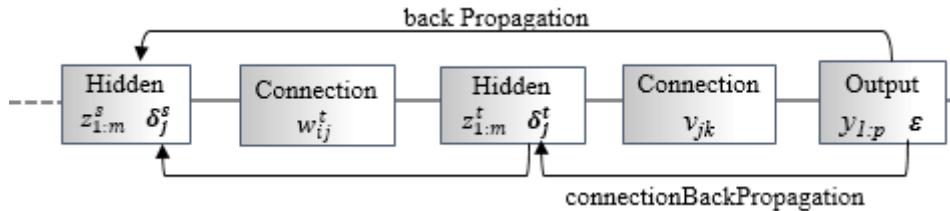
    val newDelta = inner(oldDelta, weights(n))*zh*(1.0 - zh)
    m.append((newDelta, srcOut.map(_ * newdelta) ))
    m
}
}.unzip
new Delta(deltaValues._1.toArray, deltaValues._2.toArray)//39
}

```

The implementation of the `delta` method is similar to the `MLPOutLayer.delta` method. It extracts the weights `v` from the output layer through transposition (line 37). The values of the delta matrix in the hidden connection is computed by applying the **M12** formula (line 38). The new delta instance is returned to the `trainEpoch` method (line 39) to be propagated to the preceding hidden layer if one exists.

The computational model

The computational model for the error backpropagation algorithm is very similar to the forward propagation of the input. The main difference is that the propagation of δ (delta) is performed from the output layer to the input layer. The following diagram illustrates the computational model of the backpropagation in the case of two hidden layers z_s and z_t :



An illustration of the backpropagation of the delta error

The `connectionBackPropagation` method propagates the error back from the output layer or one of the hidden layers to the preceding layer. It is a member of the `MLPConnection` class. The backpropagation of the output error across the entire network is managed by the `MLP` class.

It implements the two set of equations where `synapses(j)(i)._1` are the weights w_{ji} , `dst.delta` is the vector of the error derivative in the destination layer, and `src.delta` is the error derivative of the output in the source layer, as shown here:

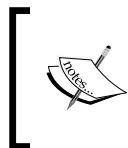
```

def connectionBackpropagation(delta: Delta): Delta = { //40
  val inSynapses = //41
    if( delta.synapses.length > 0) delta.synapses
    else synapses
}

```

```
    val delta = dst.delta(delta.loss, src.output,inSynapses) //42
    synapse = synapses.zipWithIndex.map{ //43
      case (synapsesj, j) => synapsesj.zipWithIndex.map{
        case ((w, dw), i) => {
          val ndw = config.eta*connectionDelta.delta(j)(i)
          (w + ndw - config.alpha*dw, ndw)
        }
      }
    }
    new Delta(connectionDelta.loss,
              connectionDelta.delta, synapses)
  }
```

The `connectionBackPropagation` method takes `delta` associated with the destination (`output`) layer as an argument (line 40). The output layer is the last layer of the network, and therefore, the synapses for the following connection is defined as an empty matrix of length zero (line 41). The method computes the new `delta` matrix for the hidden layer using the `delta.loss` error and output from the source layer, `src.output` (line 42). The weights (`synapses`) are updated using the gradient descent with the momentum factor as in the **M10** mathematical expression (line 43).



The adjustable learning rate

The computation of the new weights of a connection for each new epoch can be further improved by making the learning adjustable.

Step 3 – exit condition

The convergence criterion consists of evaluating the cumulative error (or loss) relevant to the operating mode (or problem) against a predefined `eps` convergence. The cumulative error is computed using either the sum of squares error formula (**M5**) or the cross-entropy formula (**M6** and **M7**). An alternative approach is to compute the difference of the cumulative error between two consecutive epochs and apply the `eps` convergence criteria as the exit condition.

Putting it all together

The `MLP` class is defined as a data transformation of the `ITransform` type using a model implicitly generated from a training set, `xt`, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 44).

The `MLP` algorithm takes the following parameters:

- `config`: This is the configuration of the algorithm
- `hidden`: This is an array of the size of the hidden layers if any
- `xt`: This is the time series of features used to train the model
- `expected`: This is the labeled output values for training purpose
- `mode`: This is the implicit operating mode or objective of the algorithm
- `f`: This is the implicit conversion from feature from type `T` to `Double`

The `V` type of the output of the prediction or classification method `|>` of this implicit transform is `DblArray` (line 45):

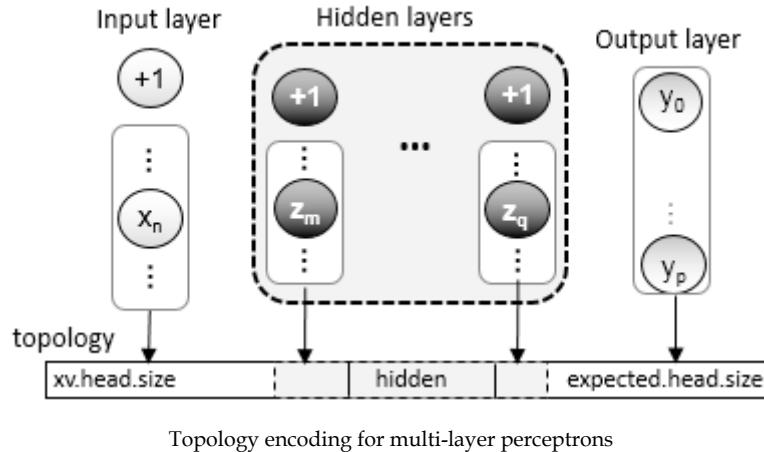
```
class MLP[T <: AnyVal] (config: MLPConfig,
  hidden: Array[Int] = Array.empty[Int],
  xt: XVSeries[T],
  expected: XVSeries[T])
  (implicit mode: MLPMode, f: T => Double)
extends ITransform[Array[T]](xt) with Monitor[Double] { //44

  type V = DblArray //45

  lazy val topology = if(hidden.length == 0)
    Array[Int](xt.head.size, expected.head.size)
  else Array[Int](xt.head.size) ++ hidden ++
    Array[Int](expected.head.size) //46

  val model: Option[MLPModel] = train
  def train: Option[MLPModel] //47
  override def |> : PartialFunction[Array[T], Try[V]]
}
```

The topology is created from the `xt` input variables, the expected values, and the configuration of hidden layers, if any (line 46). The generation of the topology from parameters of the `MLPNetwork` class is illustrated in the following diagram:



For instance, the topology of a neural network with three input variables: one output variable and two hidden layers of three neurons each is specified as `Array[Int] (4, 3, 3, 1)`. The model is generated through training by invoking the `train` method (line 47). Finally, the `|>` operator of the `ITransform` trait is used for classification, prediction, or regression, depending on the selected operating mode (line 48).

Training and classification

Once the training cycle or epoch is defined, it is merely a matter of defining and implementing a strategy to create a model using a sequence of data or time series.

Regularization

There are two approaches to find the most appropriate network architecture for a given classification or regression problem, which are follows:

- **Destructive tuning:** Starting with a large network, and then removing nodes, synapses, and hidden layers that have no impact on the sum of squared errors
- **Constructive tuning:** Starting with a small network, and then incrementally adding the nodes, synapses, and hidden layers that reduce the output error

The destructive tuning strategy removes the synapses by zeroing out their weights. This is commonly accomplished using regularization.

You have seen that regularization is a powerful technique to address overfitting in the case of the linear and logistic regression in the *Ridge regression* section in *Chapter 6, Regression and Regularization*. Neural networks can benefit from adding a regularization term to the sum of squared errors. The larger the regularization factor is, the more likely some weights will be reduced to zero, thus reducing the scale of the network [9:13].

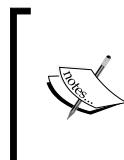
The model generation

The `MLPModel` instance is created (trained) during the instantiation of the multilayer perceptron. The constructor iterates through the training cycles (or epochs) over all the data points of the `xt` time series, until the cumulative is smaller than the `eps` convergence criteria, as shown in the following code:

```
def train: Option[MLPModel] = {
    val network = new MLPNetwork(config, topology) //48
    val zi = xt.toVector.zip(expected.view) // 49

    Range(0, config.numEpochs).find( n => { //50
        val cumulErr = fisherYates(xt.size)
            .map(zi(_))
            .map{ case(x, e) => network.trainEpoch(x, e) }
            .sum/st.size //51
        cumulErr < config.eps //52
    }).map(_ => network.getModel)
}
```

The `train` method instantiates an MLP network using the configuration and topology as the input (line 48). The method executes multiple epochs until either the gradient descent with a momentum converges or the maximum number of allowed iterations is reached (line 50). At each epoch, the method shuffles the input values and labels using the Fisher-Yates algorithm, invokes the `MLPNetwork`.`trainEpoch` method, and computes the `cumulErr` cumulative error (line 51). This particular implementation compares the value of the cumulative error against the `eps` convergence criteria as the exit condition (line 52).



Tail recursive training of MLP

The training of the multilayer is implemented as an iterative process. It can be easily substituted with a tail recursion using weights and the cumulative error as the argument of the recursion.

Lazy views are used to reduce the unnecessary creation of objects (line 49).

The exit condition

In this implementation, the training initializes the model as `None` if it does not converge before the maximum number of epochs are reached. An alternative would be to generate a model even in the case of nonconvergence and add an accuracy metric to the model, as in our implementation of the support vector machine (refer to the *Training* section under *Support vector classifiers – SVC* in *Chapter 8, Kernel Models and Support Vector Machines*).



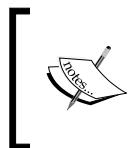
Once the model is created during the instantiation of the multilayer perceptron, it is available to predict or classify the class of a new observation.

The Fast Fisher-Yates shuffle

The *Step 5 – implementing the classifier* section under *Let's kick the tires* in *Chapter 1, Getting Started*, describes a home grown shuffling algorithm as an alternative to the `scala.util.Random.shuffle` method of the Scala standard library. This section describes an alternative shuffling mechanism known as the Fisher-Yates shuffling algorithm:

```
def fisherYates(n: Int): IndexedSeq[Int] = {  
  
    def fisherYates(seq: Seq[Int]): IndexedSeq[Int] = {  
        Random.setSeed(System.currentTimeMillis)  
        (0 until seq.size).map(i => {  
            var randomIdx: Int = i + Random.nextInt(seq.size-i) //53  
            seq(randomIdx) ^= seq(i) //54  
            seq(i) = seq(randomIdx) ^ seq(i)  
            seq(randomIdx) ^= (seq(i))  
            seq(i)  
        })  
    }  
  
    if( n <= 0) Array.empty[Int]  
    else  
        fisherYates(ArrayBuffer.tabulate(n)(n => n)) //55  
    }  
}
```

The Fisher-Yates algorithm creates an ordered sequence of integers (line 55), and swaps each integer with another integer, randomly selected from the remaining of the initial sequences (line 52). This implementation is particularly fast because the integers are swapped in place using the bit operator, also known as **bitwise swap** (line 54).



Tail recursive implementation of Fisher-Yates

The Fisher-Yates shuffling algorithm can be implemented using a tail recursion instead of an iteration.

Prediction

The `| >` data transformation implements the runtime classification/prediction. It returns the predicted value that is normalized as a probability if the model was successfully trained and `None` otherwise. The methods invoke the forward prediction function of `MLPNetwork` (line 53):

```
override def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T] if(isModel && x.size == dimension(xt)) =>
    Try(MLPNetwork(config, topology, model).predict(x)) //56
}
```

The `predict` method of `MLPNetwork` computes the output values from an input `x` using the forward propagation as follows:

```
def predict(x: DblArray): DblArray = {
  layers.head.set(x)
  connections.foreach(_.connectionForwardPropagation)
  layers.last.output
}
```

Model fitness

The fitness of a model measures how well the model fits the training set. A model with a high-degree of fitness will likely overfit. The `fit` fitness method computes the mean squared errors of the predicted values against the labels (or expected values) of the training set. The method returns the percentage of observations for which the prediction value is correct, using the higher order `count` method:

```
def fit(threshold: Double): Option[Double] = model.map(m =>
  xt.map(MLPNetwork(config, topology, Some(m)).predict(_))
    .zip(expected)
```

```
.count{case (y, e) =>mse(y, e.map(_.toDouble))< threshold }  
/xt.size.toDouble  
)
```

Model fitness versus accuracy

The fitness of a model against the training set reflects the degree the model fit the training set. The computation of the fitness does not involve a validation set. Quality parameters such as accuracy, precision, or recall measures the reliability or quality of the model against a validation set.



Our `MLP` class is now ready to tackle some classification challenges.

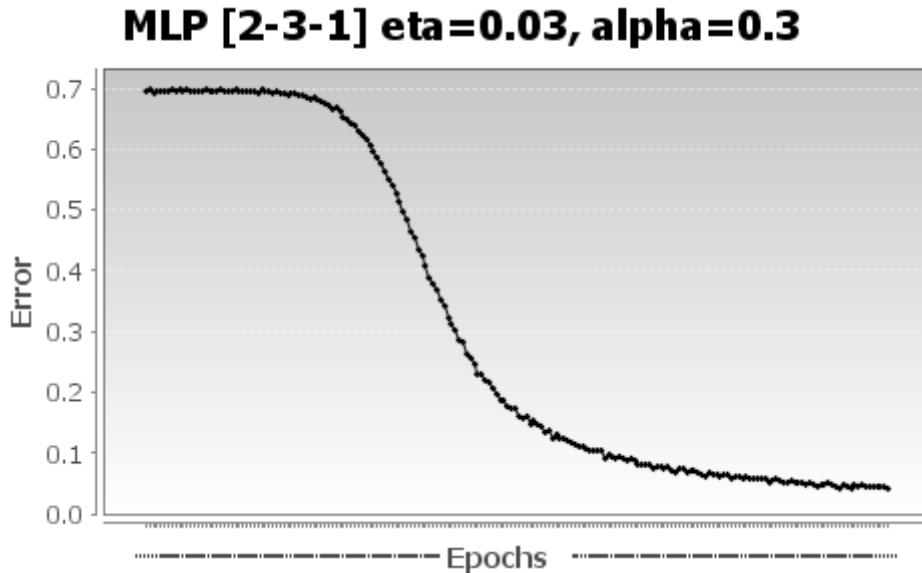
Evaluation

Before applying our multilayer perceptron to understand fluctuations in the currency market exchanges, let's get acquainted with some of the key learning parameters introduced in the first section.

The execution profile

Let's take a look at the convergence of the training of the multiple layer perceptron. The monitor trait (refer to the *Monitor* section under *Utility classes* in the *Appendix A, Basic Concepts*) collects and displays some execution parameters. We select to extract the profile for the convergence of the multiple layer perceptron using the difference of the backpropagation errors between two consecutive episodes (or epochs).

The test profiles the convergence of the MLP using a learning rate of $\eta = 0.03$ and a momentum factor of $\alpha = 0.3$ for a multilayer perceptron with two input values: one hidden layer with three nodes and one output value. The test relies on synthetically generated random values:



The execution profile for the cumulative error for MLP

Impact of the learning rate

The purpose of the first exercise is to evaluate the impact of the learning rate η on the convergence of the training epoch, as measured by the cumulative error of all output variables. The observations xt (with respect to the labeled output yt) are synthetically generated using several noisy patterns such as f_1 (line 57) and f_2 functions (line 58), as follows:

```

def f1(x: Double): DblArray = Array[Double]( //57
  0.1+ 0.5*Random.nextDouble, 0.6*Random.nextDouble)
def f2(x: Double): DblArray = Array[Double]( //58
  0.6 + 0.4*Random.nextDouble, 1.0 - 0.5*Random.nextDouble)

val HALF_TEST_SIZE = (TEST_SIZE>>1)
val xt = Vector.tabulate(TEST_SIZE)(n => //59
  if( n <HALF_TEST_SIZE) f1(n) else f2(n -HALF_TEST_SIZE))
val yt = Vector.tabulate(TEST_SIZE)(n =>
  if( n < HALF_TEST_SIZE) Array[Double](0.0)
  else Array[Double](1.0) ) //60
  
```

The input values, `xt`, are synthetically generated by the `f1` function for half of the dataset and by the `f2` function for the other half (line 59). The data generator for the expected values `yt` assigns the label 0.0 for the input values generated with the `f1` function and 1.0 for the input values created with `f2` (line 60).

The test is run with a sample of size `TEST_SIZE` data points over a maximum of `NUM_EPOCHS` epochs, a single hidden layer of `HIDDEN`.head neurons with no `softmax` transformation, and the following MLP parameters:

```
val ALPHA = 0.3
val ETA = 0.03
val HIDDEN = Array[Int] (3)
val NUM_EPOCHS = 200
val TEST_SIZE = 12000
val EPS = 1e-7

def testEta(eta: Double,
           xt: XVSeries[Double],
           yt: XVSeries[Double]): Option[(ArrayBuffer[Double], String)] = {

  implicit val mode = new MLPBinClassifier //61
  val config = MLPConfig(ALPHA, eta, NUM_EPOCHS, EPS)
  MLP[Double](config, HIDDEN, xt, yt)
    .counters("err").map( (_, s"eta=$eta")) //62
}
```

The `testEta` method generates the profile or errors given different values of `eta`.

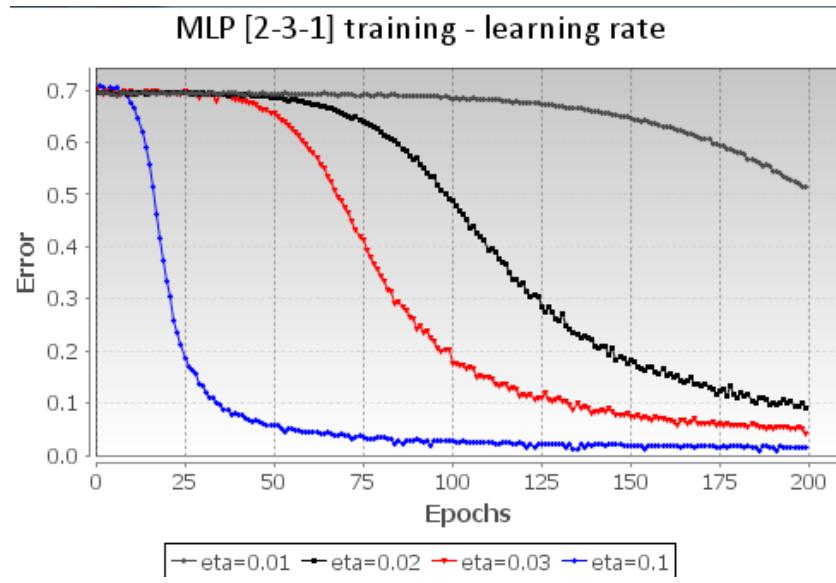
The operating mode has to be implicitly defined prior to the instantiation of the `MLP` class (line 61). It is set as a binomial classifier of the `MLPBinClassifier` type. The execution profile data is collected by the `counters` method of the `Monitor` trait (line 62) (refer to the *Monitor* section under *Utility classes* in the *Appendix A, Basic Concepts*).

The driver code for evaluating the impact of the learning rate on the convergence of the multilayer perceptron is quite simple:

```
val etaValues = List[Double] (0.01, 0.02, 0.03, 0.1)
val data = etaValues.flatMap( testEta(_, xt, yt))
  .map{ case(x, s) => (x.toVector, s) }

val legend = new Legend("Err",
  "MLP [2-3-1] training - learning rate", "Epochs", "Error")
LinePlot.display(data, legend, new LightPlotTheme)
```

The profile is created with the JFreeChart library and displayed in the following chart:



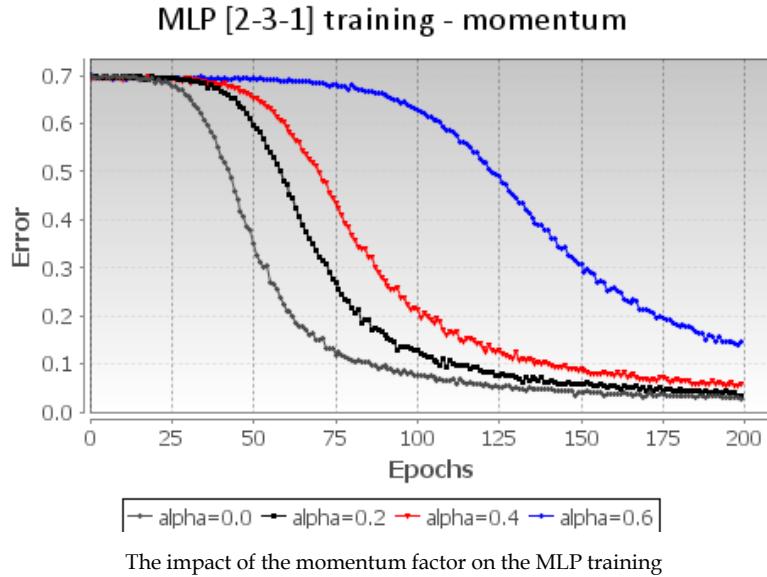
The impact of the learning rate on the MLP training

The chart illustrates that the MLP model training converges a lot faster with a larger value of learning rate. You need to keep in mind, however, that a very steep learning rate may lock the training process into a local minimum for the cumulative error, generating weights with lesser accuracy. The same configuration parameters are used to evaluate the impact of the momentum factor on the convergence of the gradient descent algorithm.

The impact of the momentum factor

Let's quantify the impact of the momentum factor a on the convergence of the training process toward an optimal model (synapse weights). The testing code is very similar to the evaluation of the impact of the learning rate.

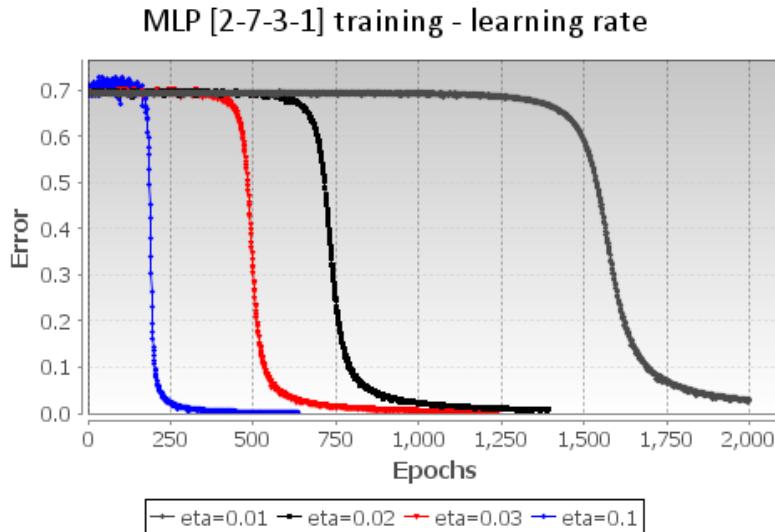
The cumulative error for the entire time series is plotted in the following graph:



The preceding graph shows that the rate of the mean square error decreases as the momentum factor increases. In other words, the momentum factor has a positive although limited impact on the convergence of the gradient descent.

The impact of the number of hidden layers

Let's consider a multilayer perceptron with two hidden layers (7 and 3 neurons). The execution profile for the training shows that the cumulative error of the output converges abruptly after several epochs for which the descent gradient failed to find a direction:



The execution profile of training of an MLP with two hidden layers

Let's apply our newfound knowledge regarding neural networks and the classification of variables that impact the exchange rate of a certain currency.

Test case

Neural networks have been used in financial applications from risk management in mortgage applications and hedging strategies for commodities pricing, to predictive modeling of the financial markets [9:14].

The objective of the test case is to understand the correlation factors between the exchange rate of some currencies, the spot price of gold, and the S&P 500 index. For this exercise, we will use the following **exchange-traded funds (ETFs)** as proxies for the exchange rate of currencies:

- **FXA:** This is the rate of an Australian dollar in US dollar
- **FXB:** This is the rate of a British pound in US dollar
- **FXE:** This is the rate of an Euro in US dollar
- **FXC:** This is the rate of a Canadian dollar in US dollar
- **FXF:** This is the rate of a Swiss franc in US dollar
- **FXY:** This is the rate of a Japanese yen in US dollar
- **CYB:** This is the rate of a Chinese yuan in US dollar
- **SPY:** This is the S&P 500 index
- **GLD:** This is the price of gold in US dollar

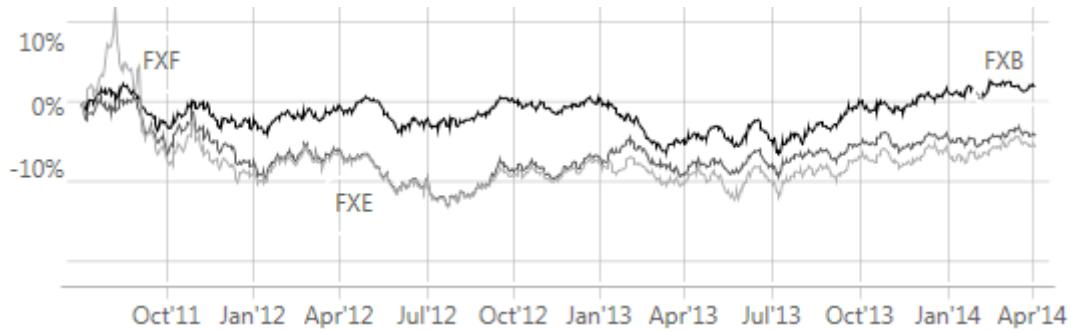
Practically, the problem to solve is to extract one or more regressive models that link one ETFs y with a basket of other ETFs $\{x_i\}$ $y=f(x_i)$. For example, is there a relation between the exchange rate of the Japanese yen (FXY) and a combination of the spot price for gold (GLD), exchange rate of the Euro in US dollar (FXE), the exchange rate of the Australian dollar in US dollar (FXA), and so on? If so, the regression f will be defined as $FXY = f(GLD, FXE, FXA)$.

The following two charts visualize the fluctuation between currencies over a period of two and a half years. The first chart displays an initial group of potentially correlated ETFs:



An example of correlated currency-based ETFs

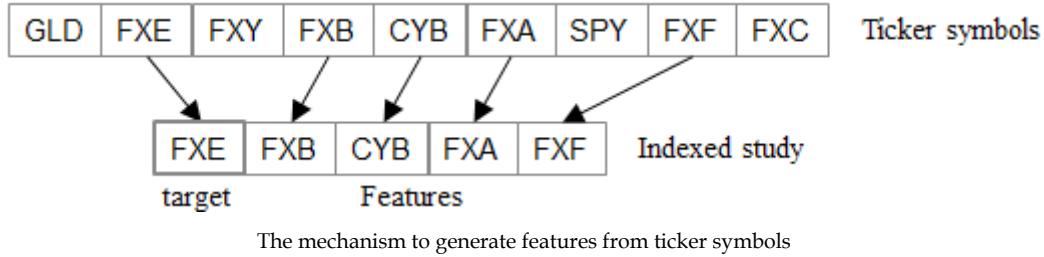
The second chart displays another group of currency-related ETFs that shares a similar price action behavior. Neural networks do not provide any analytical representation of their internal reasoning; therefore, a *visual* correlation can be extremely useful to novice engineers to validate their models:



An example of correlated currency-based ETFs

A very simple approach for finding any correlation between the movement of the currency exchange rates and the gold spot price is to select one ticker symbol as the target and a subset of other currency-based ETFs as features.

Let's consider the following problem: finding the correlation between the price of FXE and a range of currencies FXB, CYB, FXA, and FXC, as illustrated in the following diagram:



Implementation

The first step is to define the configuration parameter for the MLP classifier, as follows:

```
val path = "resources/data/chap9/"
val ALPHA = 0.8;
val ETA = 0.01
val NUM_EPOCHS = 250
val EPS = 1e-3
val THRESHOLD = 0.12
val hiddens = Array[Int](7, 7) //59
```

Besides the learning parameters, the network is initialized with multiple topology configurations (line 59).

Next, let's create the search space of the prices of all the ETFs used in the analysis:

```
val symbols = Array[String](
  "FXE", "FXA", "SPY", "GLD", "FXB", "FXF", "FXC", "FXY", "CYB"
)
val STUDIES = List[Array[String]]( //60
  Array[String]("FXY", "FXC", "GLD", "FXA"),
  Array[String]("FXE", "FXF", "FXB", "CYB"),
  Array[String]("FXE", "FXC", "GLD", "FXA", "FXY", "FXB"),
  Array[String]("FXC", "FXY", "FXA"),
  Array[String]("CYB", "GLD", "FXY"),
  symbols
)
```

The purpose of the test is to evaluate and compare seven different portfolios or studies (line 60). The closing prices of all the ETFs over a period of 3 years are extracted from the Google Financial tables, using the `GoogleFinancials` extractor for a basket of ETFs (line 61):

```
val prices = symbols.map(s => DataSource(s"$path$s.csv"))
    .flatMap(_.get(close).toOption) //61
```

The next step consists of implementing the mechanism to extract the target and the features from a basket of ETFs or studies introduced in the previous paragraph. Let's consider the following study as the list of ETF ticker symbols:

```
val study = Array[String] ("FXE", "FXF", "FXB", "CYB")
```

The first element of the study, `FXE`, is the labeled output; the remaining three elements are observed features. For this study, the network architecture has three input variables (`FXF`, `FXB`, and `CYB`) and one output variable, `FXE`:

```
val obs = symbols.flatMap(index.get(_))
    .map(prices(_).toArray) //62
val xv = obs.drop(1).transpose //63
val expected = Array[DblArray](obs.head).transpose //64
```

The set of observations, `obs`, is built using an index (line 62). By convention, the first observation is selected as the label data and the remaining studies as the features for training. As the observations are loaded as an array of time series, the time features of the series is computed using `transpose` (line 63). The single target output variable has to be converted into a matrix before transposition (line 64).

Ultimately, the model is built through instantiation of the `MLP` class:

```
implicit val mode = new MLPBinClassifier //65
val classifier = MLP[Double](config, hiddenLayers, xv, expected)
classifier.fit(THRESHOLD)
```

The objective or operating `mode` is implicitly defined as an `MLP` binary classifier, `MLPBinClassifier` (line 65). The `MLP`.`fit` method is defined in the *Training and classification* section.

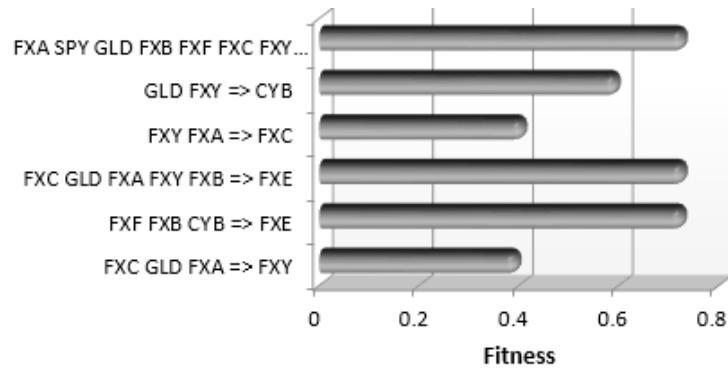
Evaluation of models

The test consists of evaluating six different models to determine which ones provide the most reliable correlation. It is critical to ensure that the result is somewhat independent of the architecture of the neural network. Different architectures are evaluated as part of the test.

The following charts compare the models for two architectures:

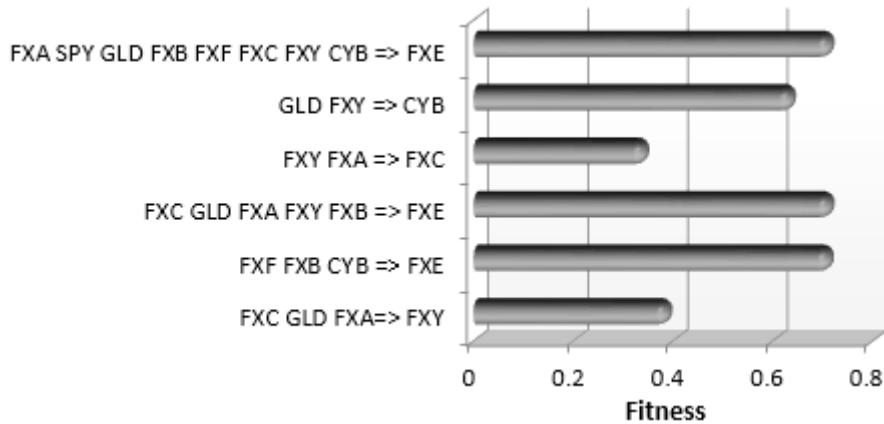
- Two hidden layers with four nodes each
- Three hidden layers with eight (with respect to five and six) nodes

This first chart visualizes the fitness of the six regression models with an architecture consisting of a variable number of inputs (2, 7): one output variable and two hidden layers of four nodes each. The features (ETF symbols) are listed on the left-hand side of the arrow \Rightarrow along the y axis. The symbol on the right-hand side of the arrow is the expected output value:



The accuracy of MLP with two hidden layers of four nodes each

The following chart displays the fitness of the six regression models for an architecture with three hidden layers of eight, five, and six nodes, respectively:



The accuracy of MLP with three hidden layers with 8, 5, and 6 nodes, respectively

The two network architectures shared a lot of similarity; in both cases, the fittest regression models are as follows:

- $FXE = f(FXA, SPY, GLD, FXB, FXF, FXD, FXY, CYB)$
- $FXE = g(FXC, GLD, FXA, FXY, FXB)$
- $FXE = h(FXF, FXB, CYB)$

On the other hand, the prediction of the Canadian dollar to US dollar's exchange rate (FXC) using the exchange rate for the Japanese yen (FXY) and the Australian dollar (FXA) is poor with both the configurations.

The empirical evaluation

These empirical tests use a simple accuracy metric. A formal comparison of the regression models will systematically analyze every combination of input and output variables. The evaluation will also compute the precision, the recall, and the F1 score for each of those models (refer to the *Key quality metrics* section under *Validation* in the *Assessing a model* section in *Chapter 2, Hello World!*!).

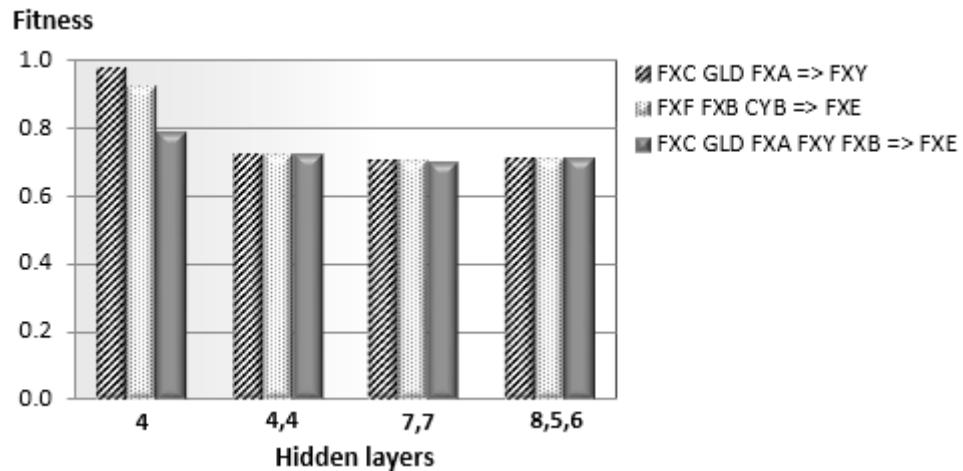
Impact of the hidden layers' architecture

The next test consists of evaluating the impact of the hidden layer(s) of configuration on the accuracy of three models: $FXF, FXB, CYB \Rightarrow FXE$, $FCX, GLD, FXA \Rightarrow FXY$, and $FXC, GLD, FXA, FXY, FXB \Rightarrow FXE$. For this test, the accuracy is computed by selecting a subset of the training data as a test sample, for the sake of convenience. The objective of the test is to compare different network architectures using some metrics, and not to estimate the absolute accuracy of each model.

The four network configurations are as follows:

- A single hidden layer with four nodes
- Two hidden layers with four nodes each
- Two hidden layers with seven nodes each
- Three hidden layers with eight, five, and six nodes

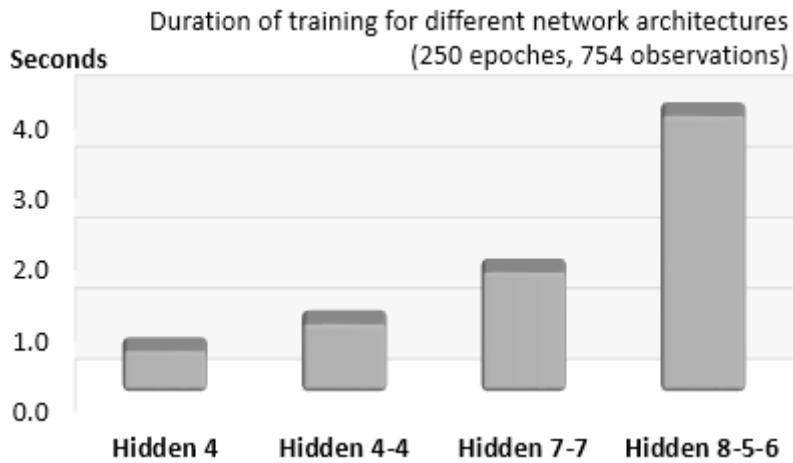
Let's take a look at the following graph:



The impact of the hidden layers' architecture on the MLP accuracy

The complex neural network architecture with two or more hidden layers generates weights with similar accuracy. The four-node single hidden layer architecture generates the highest accuracy. The computation of the accuracy using a formal cross-validation technique would generate a lower accuracy number.

Finally, we take a look at the impact of the complexity of the network on the duration of the training, as shown in the following graph:



The impact of the hidden layers' architecture on the duration of training

Not surprisingly, the time complexity increases significantly with the number of hidden layers and number of nodes.

Convolution neural networks

This section is provided as a brief introduction to convolution neural networks without the Scala implementation.

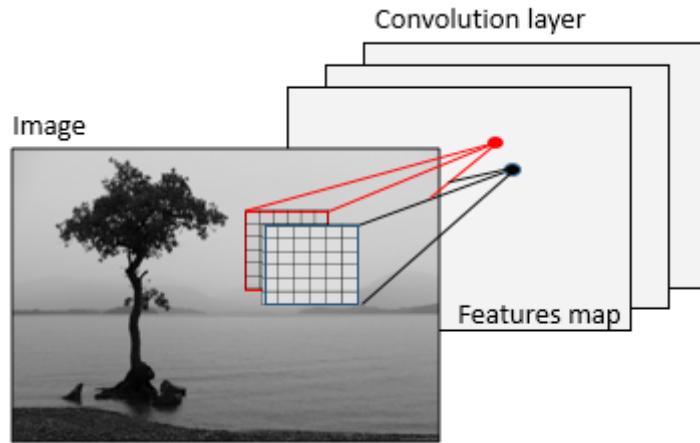
So far, the layers of perceptrons were organized as a fully connected network. It is clear that the number of synapses or weights increases significantly as the number and size of hidden layers increases. For instance, a network for a features set of dimension 6, 3 hidden layers of 64 nodes each, and one output value requires $7*64 + 2*65*64 + 65*1 = 8833$ weights!

Applications such as image or character recognition require very large features set, making training a fully connected layered perceptron very computational intensive. Moreover, these applications need to convey spatial information such as the proximity of pixels as part of the features vector.

A recent approach, known as **convolution neural networks**, consists of limiting the number of nodes in the hidden layers a input node is connected to. In other words, the methodology leverages spatial localization to reduce the complexity of connectivity between the input and the hidden layer [9:15]. The subset of input nodes connected to a single neuron in the hidden layer is known as the **local receptive fields**.

Local receptive fields

The neuron of the hidden layer learns from the local receptive fields or subimage of n by n pixels, each of those pixels being an input value. The next local receptive field, which is shifted by one pixel in any direction, is connected to the next neuron in the first hidden layer. The first hidden layer is known as the **convolution layer**. An illustration of the mapping between the input (image) and the first hidden layer (convolution layer) is as follows:



The generation of a convolution layer from an image

It would make sense that each n by n local receptive field has a bias element (+1) that connects to the hidden neuron. However, the extra complexity does not lead to a more accurate model, and therefore, the bias is shared across the neurons of the convolution layer.

Sharing of weights

The local receptive fields representing a small section of the image are generated by shifting the fields by one pixel (up, down, left, or right). Therefore, the weights associated with the local receptive fields are also shared across the neurons in the hidden layer. As a matter of fact, the same feature such as an image color or edge can be detected in many pixels across the image. The maps between the input features and neurons in the hidden layer, known as **features maps**, share weights across the convolution layer. The output is computed using the activation function.

Tanh versus the sigmoid activation

The sigmoid is predominately used in the examples related to the multilayer perceptron as the activation function for the hidden layer. The hyperbolic tangent function is commonly used for convolution networks.

Convolution layers

The output computed from the features maps is expressed as a convolution that is similar to the convolution used in a discrete Fourier transformed-based filter (refer to **M11** mathematical expression in the *DFT-based filtering* section under *Fourier analysis* in *Chapter 3, Data Preprocessing*). The activation function that computes the output in the hidden layer has to be modified to take into account the local receptive fields.

The activation of a convolution neural network

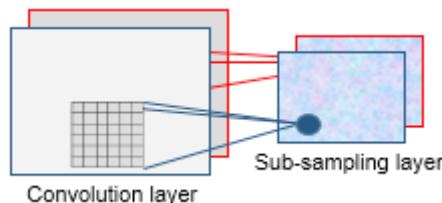
M13: The output value z_j for a shared bias w_0 , an activation function σ , a local receptive field of n by n pixels, input values x_{ij} and weights w_{uv} associated with a features map is given by:

$$\tilde{z}_j = \sigma \left(w_0 + \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} w_{u,v} x_{j+u,i+v} \right)$$

The next step in building the neural network would be to use the output of the convolution layer to a full connected hidden layer. However, the features maps in the convolution layer are usually similar so that they can be reduced to a smaller set of outputs using an intermediate layer known as subsampling layer [9:16].

Subsampling layers

Each features map in the convolution layer is reduced or condensed into a smaller features map. The layer composed of these smaller features map is known as the subsampling layer. The purpose of the sampling is to reduce the sensitivity of the weights to any minute changes in the image between adjacent pixels. The sharing of weights reduces the sensitivity to any nonsignificant changes in the image:

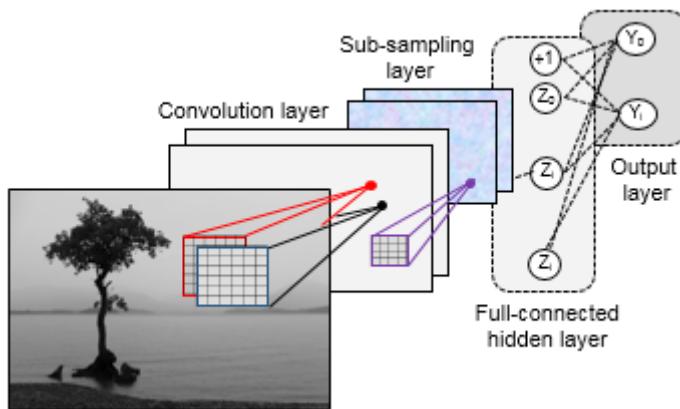


The connectivity between features map from a convolution to a subsampling layer

The subsampling layer is sometimes referred to as the **pooling layer**.

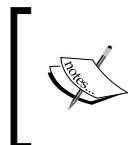
Putting it all together

The last layer of the convolution neural network is the fully connected hidden layer and output layer, subjected to the same transformative formulas as the traditional multilayer perceptron. The output values can be computed using a linear product or a softmax function:



An overview of a convolution neural network

The error backpropagation algorithm described in the *Step 2 – error back propagation* section has to be modified to support the features map [9:17].



The architecture of convolution networks

Deep convolution neural networks have multiple sequences of convolution layers and subsampling layers and may have more than one fully connection hidden layer.



Benefits and limitations

The advantages and disadvantages of neural networks depend on which other machine learning methods they are compared to. However, neural-network-based classifiers, particularly the multilayer perceptron using the error backpropagation, have some obvious advantages, which are as follows:

- The mathematical foundation of a neural network does not require expertise in dynamic programming or linear algebra, beyond the basic gradient descent algorithm.
- A neural network can perform tasks that a linear algorithm cannot.

- An MLP is usually reliable for highly dynamic and nonlinear processes. Contrary to the support vector machines, they do not require us to increase the problem dimension through kernelization.
- An MLP does not make any assumption on linearity, variable independence, or normality.
- The execution of training of an MLP lends itself to concurrent processing quite well for online training. In most architecture, the algorithm can continue even if a node in the network fails (refer to the *Apache Spark* section in *Chapter 12, Scalable Frameworks*).

However, as with any machine learning algorithm, neural networks have their detractors. The most documented limitations are as follows:

- MLP models are black boxes for which the association between features and classes may not be easily described and understood.
- An MLP requires a lengthy training process, especially using the batch training strategy. For example, a two-layer network has a time complexity (number of multiplications) of $O(n.m.p.N.e)$ for n input variables, m hidden neurons, p output values, N observations, and e epochs. It is not uncommon that a solution emerges after thousands of epochs. The online training strategy using a momentum factor tends to converge faster and requires a smaller number of epochs than the batch process.
- Tuning the configuration parameters, such as optimization of the learning rate and momentum factors, selection of the most appropriate activation method, and the cumulative error formula can turn into a lengthy process.
- Estimating the minimum size of the training set required to generate an accurate model and limiting the computation time is not obvious.
- A neural network cannot be incrementally retrained. Any new labeled data requires the execution of several training epochs.

 **Other types of neural networks**

This chapter covers the multilayer perceptron and introduces the concept of a convolution neural network. There are many more types of neural networks, such as recurrent networks and mixture density networks.

Summary

This concludes not only the journey inside the multilayer perceptron, but also the introduction of the supervised learning algorithms. In this chapter, you learned:

- The components and architecture of artificial neural networks
- The stages of the training cycle (or epoch) for the backpropagation multilayer perceptron
- How to implement an MLP from the ground up in Scala
- The numerous configuration parameters and options available to create the MLP classification or regression model.
- To evaluate the impact of the learning rate and the gradient descent momentum factor on the convergence of the training process.
- How to apply a multilayer perceptron to the financial analysis of the fluctuation of currencies
- An overview of the convolution neural network

The next chapter will introduce the concept of genetic algorithms with a complete implementation in Scala. Although, strictly speaking, genetic algorithms do not belong to the family of machine learning algorithms, they play a crucial role in the optimization of nonlinear, nondifferentiable problems, and the selection of strong classifiers within ensembles.

10

Genetic Algorithms

This chapter introduces the concept of evolutionary computing. Algorithms derived from the theory of evolution are particularly efficient in solving large combinatorial or **NP problems**. Evolutionary computing has been pioneered by John Holland [10:1] and David Goldberg [10:2]. Their findings should be of interest to anyone eager to learn about the foundation of **genetic algorithms** (GA) and **artificial life**.

This chapter covers the following topics:

- The origin of evolutionary computing
- The theoretical foundation of genetic algorithms
- Advantages and limitations of genetic algorithms

From a practical perspective, you will learn how to:

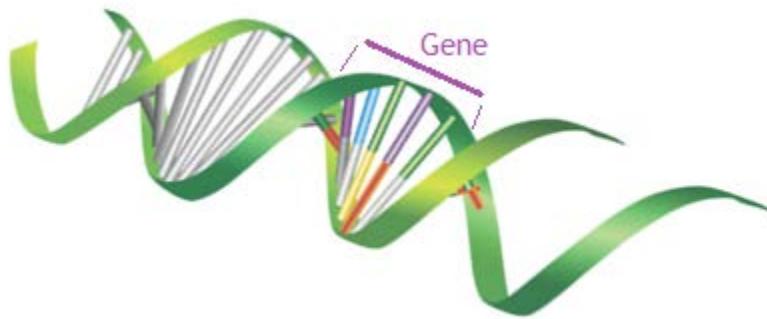
- Apply genetic algorithms to leverage technical analysis of market price and volume movement to predict future returns
- Evaluate or estimate the search space
- Encode solutions in the binary format using either hierarchical or flat addressing
- Tune some of the genetic operators
- Create and evaluate fitness functions

Evolution

The **theory of evolution**, enunciated by Charles Darwin, describes the morphological adaptation of living organisms [10:3].

The origin

The **Darwinian** process consists of optimizing the morphology of organisms to adapt to the harshest environments—hydrodynamic optimization for fishes, aerodynamic for birds, or stealth skills for predators. The following diagram shows a gene:

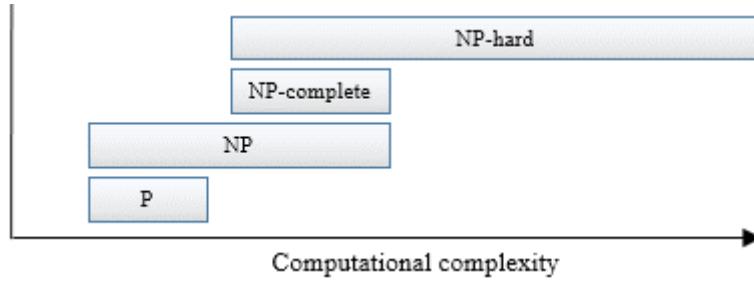


The **population** of organisms varies over time. The number of individuals within a population changes, sometimes dramatically. These variations are usually associated with the abundance or lack of predators and prey as well as the changing environment. Only the fittest organisms within the population can survive over time by adapting quickly to sudden changes in living environments and new constraints.

NP problems

NP stands for nondeterministic polynomial time. The NP problems' concept relates to the theory of computation and more precisely, time and space complexity. The categories of NP problems are as follows:

- **P-problems** (or P decision problems): For these problems, the resolution on a deterministic Turing machine (computer) takes a deterministic polynomial time.
- **NP problems**: These problems can be resolved in a polynomial time on nondeterministic machines.
- **NP-complete problems**: These are NP-hard problems that are reduced to NP problems for which the solution takes a deterministic polynomial time. These types of problems may be difficult to solve but their solutions can be validated.
- **NP-hard problems**: These problems have solutions that may not be found in polynomial time.



The categorization of NP problems using computational complexity

Problems such as the traveling salesman, floor shop scheduling, the computation of a graph K-minimum spanning tree, map coloring, or cyclic ordering have a search execution time that is a nondeterministic polynomial, ranging from $n!$ to 2^n for a population of n elements [10:4].

NP problems cannot always be solved using analytical methods because of the computation overhead—even in the case of a model, it relies on differentiable functions. Genetic algorithms were invented by John Holland in the 1970s, and they derived their properties from the theory of evolution of Darwin to tackle NP and NP-complete problems.

Evolutionary computing

A living organism consists of cells that contain identical chromosomes.

Chromosomes are strands of **DNA** and serve as a model for the whole organism. A chromosome consists of **genes** that are blocks of DNA and encode a specific protein.

Recombination (or crossover) is the first stage of reproduction. Genes from parents generate the whole new chromosome (**offspring**) that can be mutated. During mutation, one or more elements, also known as individual bases of the DNA strand or chromosomes, are changed. These changes are mainly caused by errors that occur when the genes from parents are being passed on to their offspring. The success of an organism in its life measures its fitness [10:5].

Genetic algorithms use reproduction to evolve a population of possible solutions to a problem.

Genetic algorithms and machine learning

The practical purpose of a genetic algorithm as an optimization technique is to solve problems by finding the most relevant or fittest solution among a set or group of solutions. Genetic algorithms have many applications in machine learning, which are as follows:

- **Discrete model parameters:** Genetic algorithms are particularly effective in finding the set of discrete parameters that maximizes the log likelihood. For example, the colorization of a black and white movie relies on a large but finite set of transformations from shades of grey to the RGB color scheme. The search space is composed of the different transformations and the objective function is the quality of the colorized version of the movie.
- **Reinforcement learning:** Systems that select the most appropriate rules or policies to match a given dataset rely on genetic algorithms to evolve the set of rules over time. The search space or population is the set of candidate rules, and the objective function is the credit or reward for an action triggered by these rules (refer to *Chapter 11, Reinforcement Learning*).
- **The neural network architecture:** A genetic algorithm drives the evaluation of different configurations of networks. The search space consists of different combinations of hidden layers and the size of those layers. The fitness or objective function is the sum of the squared errors.
- **Ensemble learning [10:6]:** A genetic algorithm can weed out the weak learners among a set of classifiers in order to improve the quality of the prediction.

Genetic algorithm components

Genetic algorithms have the following three components:

- **Genetic encoding (and decoding):** This is the conversion of a solution candidate and its components into the binary format (an array of bits or a string of 0 and 1 characters)
- **Genetic operations:** This is the application of a set of operators to extract the best (most genetically fit) candidates (chromosomes)
- **Genetic fitness functions:** This is the evaluation of the fittest candidate using an objective function

Encodings and the fitness function are problem dependent. Genetic operators are not.

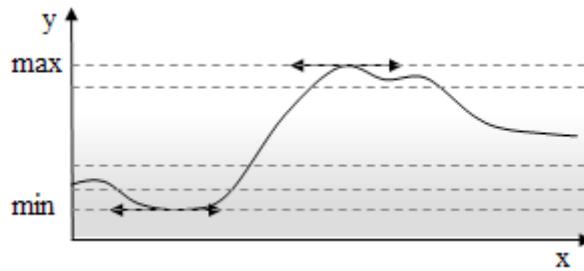
Encoding

Let's consider the optimization problem in machine learning that consists of maximizing the log likelihood or minimizing the loss function. The goal is to compute the parameters or weights, $w=\{w_i\}$, that minimize or maximize a function $f(w)$. In the case of a nonlinear model, variables may depend on other variables, which make the optimization problem particularly challenging.

Value encoding

The genetic algorithm manipulates variables as bits or bit strings. The conversion of a variable into a bit string is known as encoding. In the case where the variable is continuous, the conversion is known as **quantization** or **discretization**. Each type of variable has a unique encoding scheme, as follows:

- Boolean values are easily encoded with 1 bit: 0 for false and 1 for true.
- Continuous variables are quantized or discretized in a fashion similar to the conversion of an analog to a digital signal. Let's consider the function with a maximum **max** (similarly **min** for minimum) over a range of values, encoded with $n = 16$ bits:



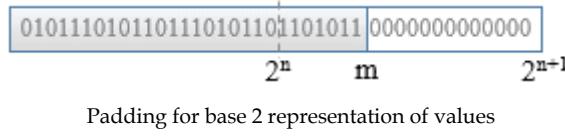
An illustration of quantization of a continuous variable $y = f(x)$

The step size of the discretization is computed as (M1):

$$\text{step} = \frac{\text{max} - \text{min}}{2^n}$$

The step size of the quantization of the \sin $y = \sin(x)$ in 16 bits is 1.524e-5.

Discrete or categorical variables are a bit more challenging to encode to bits. At a minimum, all the discrete values have to be accounted for. However, there is no guarantee that the number of variables will coincide with the bits boundary:



In this case, the next exponent, $n+1$, defines the minimum number of bits required to represent the set of values: $n = \log_2(m).toInt + 1$. A discrete variable with 19 values requires 5 bits. The remaining bits are set to an arbitrary value (0, NaN, and so on) depending on the problem. This procedure is known as **padding**.

Encoding is as much art as it is science. For each encoding function, you need a decoding function to convert the bits representation back to actual values.

Predicate encoding

A predicate for a variable x is a relation defined as a x operator [target]; for instance, *unit cost < [9\$]*, *temperature = [82F]*, or *Movie rating is [3 stars]*.

The simplest encoding scheme for predicates is as follows:

- **Variables** are encoded as a category or type (for example, temperature, barometric pressure, and so on) because there are a finite number of variables in any model
- **Operators** are encoded as discrete types
- **Values** are encoded as either discrete or continuous values



Encoding format for predicates

There are many approaches for encoding a predicate in a bits string. For instance, the format {operator, left-operand, and right-operand} is useful because it allows you to encode a binary tree. The entire rule, *IF predicate THEN action*, can be encoded with the action being represented as a discrete or categorical value.

Solution encoding

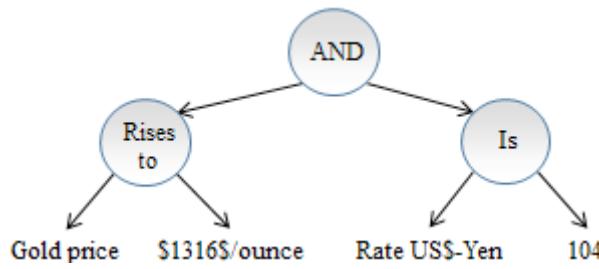
The solution encoding approach describes the solution to a problem as an unordered sequence of predicates. Let's consider the following rule:

```
IF {Gold price rises to [1316$/ounce]} AND
    {US$/Yen rate is [104]}).
THEN {S&P 500 index is [UP]}
```

In this example, the search space is defined by two levels:

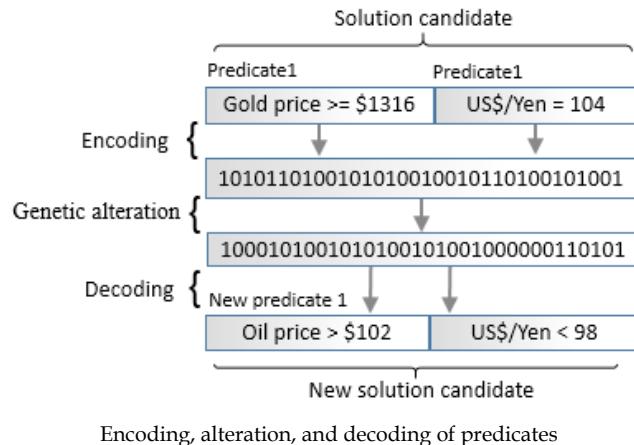
- Boolean operators (for example, AND) and predicates
- Each predicate is defined as a tuple (a variable, operator, target value)

The tree representation for the search space is shown in the following diagram:



A graph representation of encoded rules

The bits string representation is decoded back to its original format for further computation:



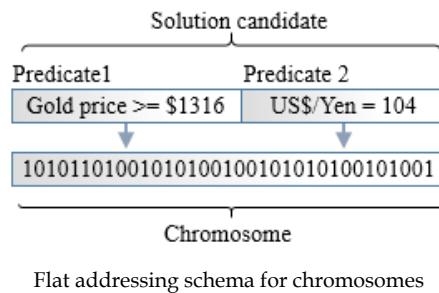
The encoding scheme

There are two approaches to encode such a candidate solution or chain of predicates:

- Flat coding of a chromosome
- Hierarchical coding of a chromosome as a composition of genes

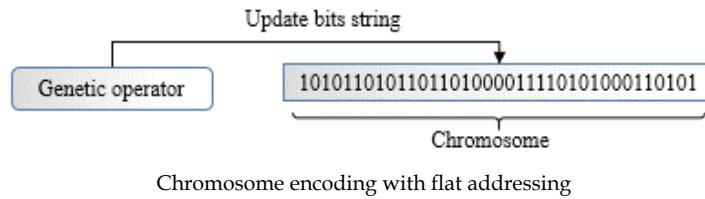
Flat encoding

The flat encoding approach consists of encoding the set of predicates into a single chromosome (bits string), representing a specific solution candidate to the optimization problem. The identity of the predicates is not preserved:



Flat addressing schema for chromosomes

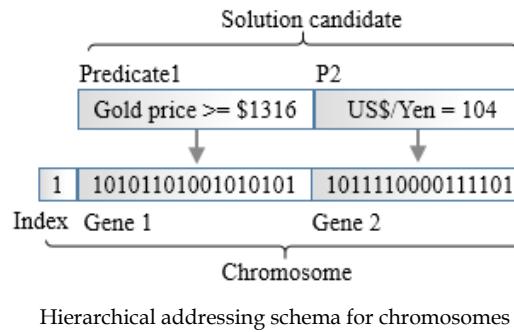
A genetic operator manipulates the bits of the chromosome regardless of whether the bits refer to a particular predicate:



Chromosome encoding with flat addressing

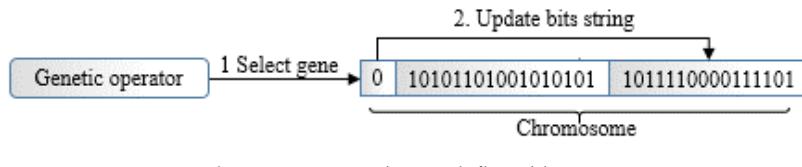
Hierarchical encoding

In this configuration, the characteristic of each predicate is preserved during the encoding process. Each predicate is converted into a gene represented by a bit string. The genes are aggregated to form the chromosome. An extra field is added to the bits string or chromosome for the selection of the gene. This extra field consists of the index or the address of the gene:



Hierarchical addressing schema for chromosomes

A generic operator selects the predicate it needs to first manipulate. Once the target gene is selected, the operator updates the bits string associated with the gene, as follows:



Chromosome encoding with flat addressing

The next step is to define the genetic operators that manipulate or update the bits string representing either a chromosome or individual genes.

Genetic operators

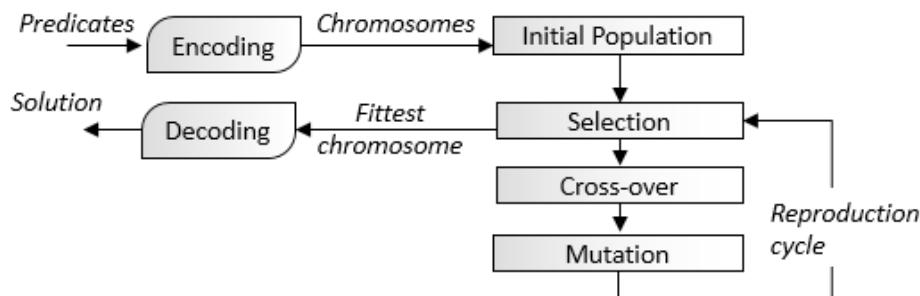
The implementation of the reproduction cycle attempts to replicate the natural reproduction process [10:7]. The reproduction cycle that controls the population of chromosomes consists of three genetic operators:

- **Selection:** This operator ranks chromosomes according to a fitness function or criteria. It eliminates the weakest or less-fit chromosomes and controls the population growth.
- **Crossover:** This operator pairs chromosomes to generate offspring chromosomes. These offspring chromosomes are added to the population along with their parent chromosomes.
- **Mutation:** This operator introduces a minor alteration in the genetic code (bits string representation) to prevent the successive reproduction cycles from electing the same fittest chromosome. In optimization terms, this operator reduces the risk of the genetic algorithm converging quickly toward a local maximum or minimum.

The transposition operator

Some implementations of genetic algorithms use a fourth operator, genetic transposition, in case the fitness function cannot be very well defined and the initial population is very large. Although additional genetic operators could potentially reduce the odds of finding a local maximum or minimum, the inability to describe the fitness criteria or the search space is a sure sign that a genetic algorithm may not be the most suitable tool.

The following diagram gives an overview of the genetic algorithm workflow:



A basic workflow for the execution of genetic algorithms

Initialization

The initialization of the search space (a set of potential solutions to a problem) in any optimization procedure is challenging and genetic algorithms are no exception. In the absence of biases or heuristics, the reproduction initializes the population with randomly generated chromosomes. However, it is worth the effort to extract the characteristics of a population. Any well-founded bias introduced during initialization facilitates the convergence of the reproduction process.

Each of these genetic operators has at least one configurable parameter that has to be estimated and/or tuned. Moreover, you will likely need to experiment with different fitness functions and encoding schemes in order to increase your odds of finding a fittest solution (or chromosome).

Selection

The purpose of the genetic selection phase is to evaluate, rank, and weed out the chromosomes (that is, the solution candidates) that are not a good fit for the problem. The selection procedure relies on a fitness function to score and rank candidate solutions through their chromosomal representation. It is a common practice to constrain the growth of the population of chromosomes by setting a limit to the size of the population.

There are several methodologies to implement the selection process from scaled relative fitness, Holland roulette wheel, and tournament selection to rank-based selection [10:8].

Relative fitness degradation

 As the initial population of chromosomes evolves, the chromosomes tend to get more and more similar to each other. This phenomenon is a healthy sign that the population is actually converging. However, for some problems, you may need to scale or magnify the relative fitness to preserve a meaningful difference in the fitness score between the chromosomes [10:9].

The following implementation relies on rank-based selection.

The selection process consists of the following steps:

1. Apply the fitness function to each chromosome j in the population f_j .
2. Compute the total fitness score for the entire population $\sum f_j$.
3. Normalize the fitness score of each chromosome by the sum of the fitness scores of all the chromosomes $f_j' = f_j / \sum f_j$.
4. Sort the chromosomes by their descending fitness score $f_j' < f_{j-1}'$.
5. Compute the cumulative fitness score for each chromosome j $f_j = f_j + \sum f_k$.
6. Generate the selection probability (for the rank-based formula) as a random value $p \in [0,1]$.
7. Eliminate the chromosome k that has a low unfitness score $f_k < p$ or high fitness cost $f_k > p$.
8. Reduce the size of the population if it exceeds the maximum allowed number of chromosomes.

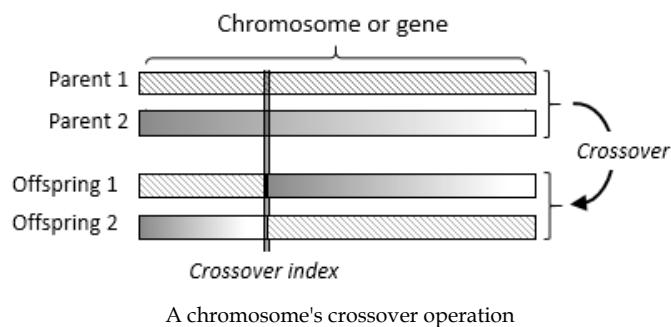
Natural selection



You should not be surprised by the need to control the size of the population of chromosomes. After all, nature does not allow any species to grow beyond a certain point in order to avoid depleting natural resources. The predator-prey process modeled by the **Lotka-Volterra** equation [10:10] keeps the population of each species in check.

Crossover

The purpose of the genetic crossover is to expand the current population of chromosomes in order to intensify the competition among the solution candidates. The crossover phase consists of reprogramming chromosomes from one generation to the next. There are many different variations of crossover techniques. The algorithm for the evolution of the population of chromosomes is independent of the crossover technique. Therefore, the case study uses the simpler one-point crossover. The crossover swaps sections of the two-parent chromosomes to produce two offspring chromosomes, as illustrated in the following diagram:



An important element in the crossover phase is selecting and pairing of parent chromosomes. There are different approaches for selecting and pairing the parent chromosomes that are the most suitable for reproduction:

- Selecting only the n fittest chromosomes for reproduction
- Pairing chromosomes ordered by their fitness (or unfitness) value
- Pairing the fittest chromosome with the least-fit chromosome, the second fittest chromosome with the second least-fit chromosome, and so on

It is a common practice to rely on a specific optimization problem to select the most appropriate selection method as it is highly domain dependent.

The crossover phase that uses hierarchical addressing as the encoding scheme consists of the following steps:

1. Extract pairs of chromosomes from the population.
2. Generate a random probability $p \in [0,1]$.
3. Compute the index r_i of the gene for which the crossover is applied as $r_i = p.\text{num_genes}$, where num_genes are the number of genes in a chromosome.
4. Compute the index of the bit in the selected gene for which the crossover is applied as $x_i = p.\text{gene_length}$, where gene_length is the number of bits in the gene.
5. Generate two offspring chromosomes by interchanging strands between parents.
6. Add the two offspring chromosomes to the population.

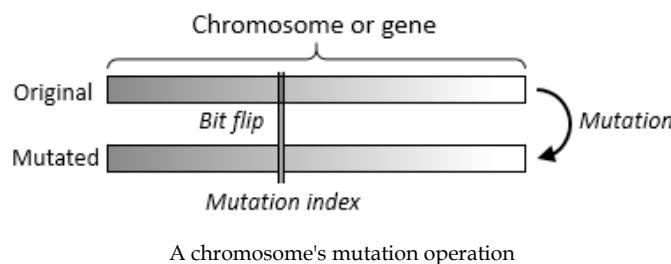


Preserving parent chromosomes

You may wonder why the parents are not removed from the population once the offspring chromosomes are created. This is because there is no guarantee that any of the offspring chromosomes are a better fit.

Mutation

The objective of genetic mutation is to prevent the reproduction cycle from converging toward a local optimum by introducing a pseudo-random alteration to the genetic material. The mutation procedure inserts a small variation in a chromosome to maintain some level of diversity between generations. The methodology consists of flipping one bit in the bits string representation of the chromosome, as illustrated in the following diagram:



The mutation is the simplest of the three phases in the reproduction process. In the case of hierarchical addressing, the steps are as follows:

1. Select the chromosome to be mutated.
2. Generate a random probability $p \in [0,1]$.
3. Compute the index m_i of the gene to be mutated using the formula $m_i = p.\text{num_genes}$.
4. Compute the index of the bit in the gene to be mutated $x_i = p.\text{genes_length}$.
5. Perform a flip XOR operation on the selected bit.

The tuning issue



The tuning of a genetic algorithm can be a daunting task. A plan including a systematic design experiment for measuring the impact of the encoding, fitness function, crossover, and mutation ratio is necessary to avoid lengthy evaluation and self-doubt.

The fitness score

The fitness function is the centerpiece of the selection process. There are three categories of fitness functions, which are as follows:

- **The fixed fitness function:** In this function, the computation of the fitness value does not vary during the reproduction process
- **The evolutionary fitness function:** In this function, the computation of the fitness value morphs between each selection according to predefined criteria
- **An approximate fitness function:** In this function, the fitness value cannot be computed directly using an analytical formula [10:11]

Our implementation of the genetic algorithm uses a fixed fitness function.

Implementation

As mentioned earlier, the genetic operators are independent of the problem to be solved. Let's implement all the components of the reproduction cycle. The fitness function and the encoding scheme are highly domain specific.

In accordance with the principles of object-oriented programming, the software architecture defines the genetic operators using a top-down approach: starting with the population, then each chromosome, and down to each gene.

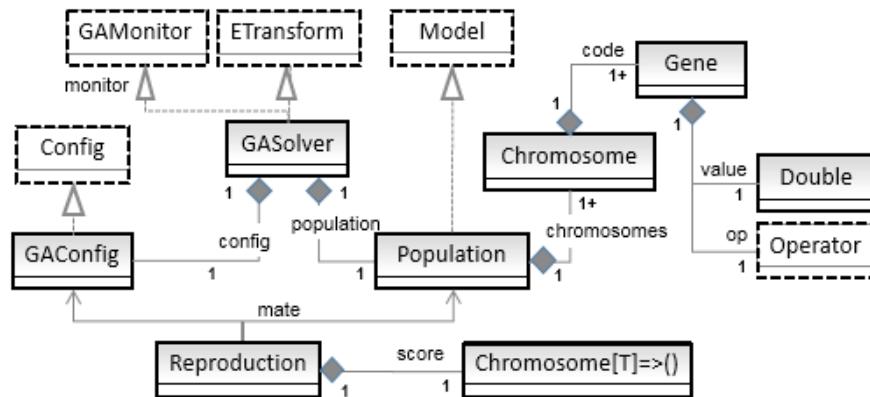
Software design

The implementation of the genetic algorithm uses a design that is similar to the template for classifiers (refer to the *Design template for classifier* section in the *Appendix A, Basic Concepts*).

The key components of the implementation of the genetic algorithm are as follows:

- The `Population` class defines the current set of solution candidates or chromosomes.
- The `GASolver` class implements the GA solver and has two components: a configuration object of the `GAConfig` type and the initial population. This class implements an explicit monadic data transformation of the `ETransform` type.
- The `GAConfig` configuration class consists of the GA execution and reproduction configuration parameters.
- The reproduction (of the `Reproduction` type) controls the reproduction cycle between consecutive generations of chromosomes through the `mate` method.
- The `GAMonitor` monitoring trait tracks the progress of the optimization and evaluates the exit condition for each reproduction cycle.

The following UML class diagram describes the relation between the different components of the genetic algorithm:



The UML class diagram of genetic algorithm components

Let's start with defining the key classes that control the genetic algorithm.

Key components

The `Population` parameterized class (with the `Gene` subtype) contains the set or pool of chromosomes. A population contains chromosomes that are a sequence or list of elements of the type inherited from `Gene`. A `Pool` is a mutable array used in order to avoid excessive duplication of the `Chromosome` instances associated with immutable collections.

The case for mutability

 It is a good Scala programming practice to stay away from mutable collections. However, in this case, the number of chromosomes can be very large. Most implementations of genetic algorithms update the population potentially three times per reproduction cycle, generating a large number of objects and taxing the Java garbage collector.

Population

The `Population` class takes two arguments:

- `limit`: This is the maximum size of the population
- `chromosomes`: This is the pool of chromosomes that define the current population

A reproduction cycle executes the following sequence of three genetic operators on a population: `select` for the selection across all the chromosomes of the population (line 1), `+-` for crossover of all the chromosomes (line 2), and `^` for the mutation of each chromosome (line 3). Consider the following code:

```
type Pool[T <: Gene] = mutable.ArrayBuffer[Chromosome[T]]  
  
class Population[T <: Gene] {  
    limit: Int, val chromosomes: Pool[T]) {  
        def select(score: Chromosome[T] => Unit, cutOff: Double) //1  
        def +- (xOver: Double) //2  
        def ^ (mu: Double) //3  
        ...  
    }  
}
```

The `limit` value specifies the maximum size of the population during optimization. It defines the hard limit or constraints on the population growth.

Chromosomes

The chromosome is the second level of containment in the genotype hierarchy. The Chromosome class takes a list of genes as parameter (code). The signature of the crossover and mutation methods, `+-` and `^`, are similar to their implementation in the Population class except for the fact that the crossover and mutable parameters are passed as indices relative to the list of genes and each gene. The section dedicated to the genetic crossover describes the `GeneticIndices` class:

```
class Chromosome[T <: Gene](val code: List[T]) {
    var cost: Double = Random.nextDouble //4
    def +- (that: Chromosome[T], idx: GeneticIndices):
        (Chromosome[T], Chromosome[T])
    def ^ (idx: GeneticIndices): Chromosome[T]
    ...
}
```

The algorithm assigns the (un)fitting score or a `cost` value to each chromosome to enable the ranking of chromosomes in the population, and ultimately, the selection of the fittest chromosomes (line 4).

Fitness versus cost

 The machine learning algorithms use the loss function or its variant as an objective function to be minimized. This implementation of the GA uses `cost` scores in order to be consistent with the concept of the minimization of the cost, loss, or penalty function.

Genes

Finally, the reproduction process executes the genetic operators on each gene:

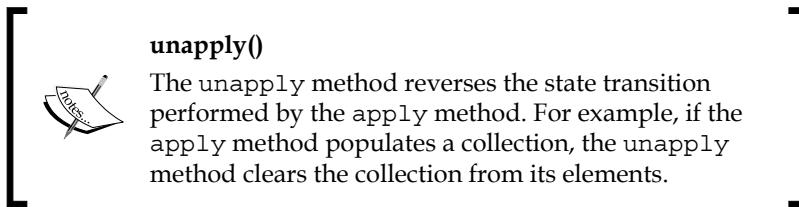
```
class Gene(val id: String,
          val target: Double,
          op: Operator)
          (implicit quantize: Quantization, encoding: Encoding){//5
    lazy val bits: BitSet = apply(target, op)

    def apply(value: Double, op: Operator): BitSet //6
    def unapply(bitSet: BitSet): (Double, Operator) //7
    ...
    def +- (index: Int, that: Gene): Gene //8
    def ^ (index: Int): Unit //9
    ...
}
```

The Gene class takes three arguments and two implicit parameters, which are as follows:

- `id`: This is the identifier of the gene. It is usually the name of the variable represented by the gene.
- `target`: This is the target value or threshold to be converted or discretized into a bit string.
- `op`: This is the operator that is applied to the target value.
- `quantize`: This is the quantization or discretization class that converts a double value to an integer to be converted into bits and vice versa (line 5).
- `encoding`: This is the encoding or bits layout of the gene as a pair of values and operators.

The `apply` method encodes a pair of value and operator into a bit set (line 6). An `unapply` method is the reverse operation of `apply`. In this case, it decodes a bit set into a pair of value and operator (line 7).



The implementation of the crossover (line 8) and mutation (line 9) operators on a gene is similar to the operations on the container chromosome.

The quantization is implemented as a case class:

```
case class Quantization(toInt: Double => Int,  
                     toDouble: Int => Double) {  
  def this(R: Int) = this((x: Double) =>  
    (x*R).floor.toInt, (n: Int) => n/R)  
}
```

The first `toInt` function converts a real value to an integer and `toDouble` converts the integer back to a real value. The `discretization` and `inverse` functions are encapsulated into a class to reduce the risk of inconsistency between the two opposite conversion functions.

The instantiation of a gene converts the predicate representation into a bit string (bits of the `java.util.BitSet` type) using the quantization function, `Quantization.toInt`.

The layout of a gene is defined by the `Encoding` class as follows:

```
class Encoding(nValueBits: Int, nOpBits: Int) {
    val rValue = Range(0, nValueBits)
    val length = nValueBits + nOpBits
    val rOp = Range(nValueBits, length)
}
```

The `Encoding` class specifies the bits layout of the gene as a number of bits, `nValueBits`, to encode the value and the number of bits, `nOpBits`, to encode the operator. The class defines the `rValue` range for the value and the `rOp` range for the operator. The client code has to be supplied to the implicit instance of the `Encoding` class.

The bit set, `bitset`, of the gene (encoding) is implemented by using the `apply` method:

```
def apply(value: Double, op: Operator): BitSet = {
    val bitset = new BitSet(encoding.length)
    encoding.rOp foreach(i => //10
        if((op.id>>i) & 0x01)==0x01) bitset.set(i))
    encoding.rValue foreach(i => //11
        if( ((quant.toInt(value)>>i) & 0x01)==0x01) bitset.set(i))
    bitset
}
```

The bits layout of the gene is created using `java.util.BitSet`. The `op` operator is encoded first through its identifier, `id` (line 10). The `value` is quantized by invoking the `toInt` method and then encoded (line 11).

The `unapply` method decodes the gene from a bit set or bit string to a pair of values and operators. The method uses the quantization instance to cover bits into values and a convert auxiliary function that is described along with its implementation in the source code, accompanying the book (line 12):

```
def unapply(bits: BitSet): (Double, Operator) =
    (quant.toDouble(convert(encoding.rValue, bits)),
     op(convert(encoding.rOp, bits))) //12
```

The `Operator` trait defines the signature of any operator. Each domain-specific problem requires a unique set of operations: Boolean, numeric, or string manipulation:

```
trait Operator {
    def id: Int
    def apply(id: Int): Operator
}
```

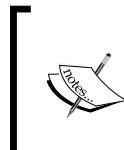
The preceding operator has two methods: an identifier `id` and an `apply` method that converts an index to an operator.

Selection

The first genetic operator of the reproduction cycle is the selection process. The `select` method of the `Population` class implements the steps of the selection phase to the population of chromosomes in the most efficient manner, as follows:

```
def select(score: Chromosome[T] => Unit, cutOff: Double): Unit = {  
    val cumul = chromosomes.map(_.cost).sum/SCALING_FACTOR //13  
    chromosomes foreach(_ /= cumul) //14  
  
    val _chromosomes = chromosomes.sortWith(_.cost < _.cost)//15  
    val cutOffSize = (cutOff*_chromosomes.size).floor.toInt //16  
    val popSize = if(limit < cutOffSize) limit else cutOffSize  
  
    chromosomes.clear //17  
    chromosomes ++= _chromosomes.take(popSize) //18  
}
```

The `select` method computes the `cumul` cumulative sum of the `cost` (line 13) for the entire population. It normalizes the cost of each chromosome (line 14), orders the population by decreasing the value (line 15), and applies a `cutOff` soft limit function on the population growth (line 16). The next step reduces the size of the population to the lowest of the two limits: the hard limit, `limit`, or the soft limit, `cutOffSize`. Finally, the existing chromosomes are cleared (line 17) and updated with the next generation (line 18).



Even population size

The next phase in the reproduction cycle is the crossover, which requires the pairing of parent chromosomes. It makes sense to pad the population so that its size is an even integer.

The `score` scoring function takes a chromosome as a parameter and returns the `cost` value for this chromosome.

Controlling the population growth

The natural selection process controls or manages the growth of the population of species. The genetic algorithm uses the following two mechanisms:

- The absolute maximum size of the population (the hard limit).
- The incentive to reduce the population as the optimization progresses (the soft limit). This incentive (or penalty) on the population growth is defined by the `cutoff` value used during selection (the `select` method).

The `cutoff` value is computed using a `softLimit` user-defined function of the `Int => Double` type, which is provided as a configuration parameter (`softLimit(cycle: Int) => a.cycle + b`).

The GA configuration

The four configurations and tuning parameters required by the genetic algorithm are as follows:

- `xOver`: This is the crossover ratio (or probability) and has a value in the interval $[0, 1]$
- `mu`: This is the mutation ratio
- `maxCycles`: This is the maximum number of reproduction cycles
- `softLimit`: This is the soft constraint on the population growth

Consider the following code:

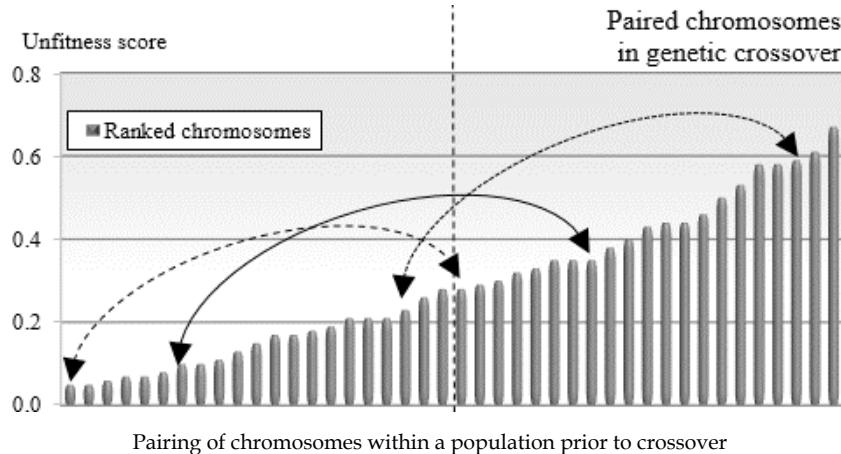
```
class GAConfig(val xover: Double,
               val mu: Double,
               val maxCycles: Int,
               val softLimit: Int => Double) extends Config {
  val mutation = (cycle : Int) => softLimit(cycle)
}
```

Crossover

As mentioned earlier, the genetic crossover operator couples two chromosomes to generate two offspring chromosomes that compete with all the other chromosomes in the population, including their own parents, in the selection phase of the next reproduction cycle.

Population

We use the `+-` notation as the implementation of the crossover operator in Scala. There are several options to select pairs of chromosomes for crossover. This implementation ranks the chromosomes by their *fitness* (or inverse *cost*) value and then divides the population into two halves. Finally, it pairs the chromosomes of identical rank from each half, as illustrated in the following diagram:



The crossover implementation, `+-`, selects the parent chromosome candidates for crossover using the pairing scheme described earlier. Consider the following code:

```
def +- (xOver: Double): Unit =
  if( size > 1) {
    val mid = size>>1
    val bottom = chromosomes.slice(mid, size) //19
    val gIdx = geneticIndices(xOver) //20

    val offSprings = chromosomes.take(mid).zip(bottom)
      .map{ case (t, b) => t +- (b, gIdx) }.unzip //21
    chromosomes ++= offSprings._1 ++ offSprings._2 //22
  }
```

This method splits the population into two subpopulations of equal size (line 19) and applies the Scala `zip` and `unzip` methods to generate the set of pairs of offspring chromosomes (line 20). The `+-` crossover operator is applied to each chromosome pair to produce an array of pairs of `offSprings` (line 21). Finally, the `crossover` method adds offspring chromosomes to the existing population (line 22). The `xOver` crossover value is a probability randomly generated over the interval `[config.xOver, 1]`.

The `GeneticIndices` case class defines two indices of the bit whenever a crossover or a mutation occurs. The first `chOpIdx` index is the absolute index of the bit affected by the genetic operation in the chromosome (line 23). The second `geneOpIdx` index is the index of the bit within the gene subjected to crossover or mutation (line 24):

```
case class GeneticIndices(val chOpIdx: Int, //23
                           val geneOpIdx: Int) //24
```

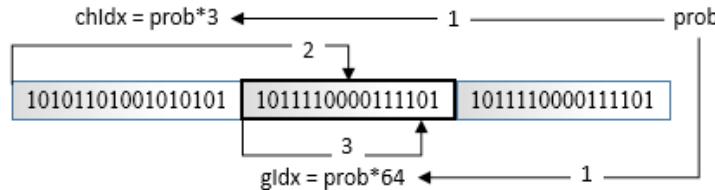
The `geneticIndices` method computes the relative indices of the crossover bit in the chromosomes and genes:

```
def geneticIndices(prob: Double): GeneticIndices = {
  var idx = (prob*chromosomeSize).floor.toInt //25
  val chIdx = if(idx == chromosomeSize) chromosomeSize-1
               else idx //25

  idx = (prob*geneSize).floor.toInt
  val gIdx = if(idx == geneSize) geneSize-1 else idx //26
  GeneticIndices(chIdx, gIdx)
}
```

The first `chIdx` indexer is the index or rank of the gene within the chromosome to be affected by the genetic operator (line 25). The second `gIdx` indexer is the relative index of the bit within the gene (line 26).

Let's consider a chromosome composed of 2 genes with 63 bits/elements each, as illustrated in the following diagram:



The `geneticIndices` method computes the following:

- The `chIdx` index of the gene within the chromosome and the `gIdx` index of the bit within the gene
- The genetic operator selects the gene of the `chIdx` index (that is the second gene) to be altered
- The genetic operator alters the chromosome at the bit of the `gIdx` index (that is $chIdx*64 + gIdx$)

Chromosomes

First, we need to define the Chromosome class, which takes a list of genes, code, (for genetic code) as the parameter:

```
val QUANT = 500
class Chromosome[T <: Gene] (val code: List[T]) {
    var cost: Double = QUANT*(1.0 + Random.nextDouble) //27

    def +- (that: Chromosome[T], indices: GeneticIndices): //28
        (Chromosome[T], Chromosome[T])
    def ^ (indices: GeneticIndices): Chromosome[T] //29
    def /= (normalizeFactor: Double): Unit = //30
        cost /= normalizeFactor
    def decode(implicit d: Gene=>T): List[T] = //31
        code.map( d(_))
    ...
}
```

The cost (or unfitness) of a chromosome is initialized as a random value between QUANT and 2*QUANT (line 27). The genetic +- crossover operator generates a pair of two offspring chromosomes (line 28). The genetic ^ mutation operator creates a slightly modified (1 or 2 bits) clone of this chromosome (line 29). The /= method normalizes the cost of the chromosome (line 30). The decode method converts the gene to a logic predicate or rule using an implicit conversion, d, between a gene and its subclass (line 31).

Cost initialization

There is no absolute rule to initialize the cost of the chromosomes from an initial population. However, it is recommended that you differentiate a chromosome using nonzero random values with a large range as their cost.

The implementation of the crossover for a pair of chromosomes using hierarchical encoding follows two steps:

1. Find the gene on each chromosome that corresponds to the indices chOpIdx crossover index and then swap the remaining genes.
2. Split and splice the gene crossover at xoverIdx.

Consider the following code:

```
def +- (that: Chromosome[T], indices: GeneticIndices):  
    (Chromosome[T], Chromosome[T]) = {  
    val xoverIdx = indices.chOpIdx //32  
    val xGenes = spliceGene(indices, that.code(xoverIdx)) //33  
  
    val offSprng1 = code.slice(0, xoverIdx) :::  
        xGenes._1 :: that.code.drop(xoverIdx+1) //34  
    val offSprng2 = that.code.slice(0, xoverIdx) :::  
        xGenes._2 :: code.drop(xoverIdx+1)  
    (Chromosome[T](offSprng1), Chromosome[T](offSprng2)) //35  
}
```

The crossover method computes the index `xoverIdx` of the bit that defines the crossover in each parent chromosome (line 32). The `this.code(xoverIdx)` and `that.code(xoverIdx)` genes are swapped and spliced by the `spliceGene` method to generate a spliced gene (line 33):

```
def spliceGene(indices: GeneticIndices, thatCode: T): (T,T) ={  
    ((this.code(indices.chOpIdx) +- (thatCode,indices)),  
     (thatCode +- (code(indices.chOpIdx),indices)) )  
}
```

The offspring chromosomes are gathered by collating the first `xOverIdx` genes of the parent chromosome, the crossover gene, and the remaining genes of the other parent (line 34). The method returns the pair of offspring chromosomes (line 35).

Genes

The crossover is applied to a gene using the `+-` method of the `Gene` class. The exchange of bits between the `this` and `that` genes uses the `BitSet` Java class to rearrange the bits after the permutation:

```
def +- (that: Gene, indices: GeneticIndices): Gene = {  
    val clonedBits = cloneBits(bits) //36  
  
    Range(indices.geneOpIdx, bits.size).foreach(n =>  
        if( that.bits.get(n) ) clonedBits.set(n)  
        else clonedBits.clear(n) //37  
    )  
    val valOp = decode(clonedBits) //38  
    new Gene(id, valOp._1, valOp._2)  
}
```

The bits of the gene are cloned (line 36) and then spliced by exchanging their bits along with the `indices.geneOpIdx` crossover point (line 37). The `cloneBits` function duplicates a bit string, which is then converted into a (target value, operator) tuple using the `decode` method (line 38). We omit these two methods because they are not critical to the understanding of the algorithm.

Mutation

The mutation of the population uses the same algorithmic approach as the crossover operation.

Population

The `^` mutation operator invokes the same operator for all the chromosomes in the population and then adds the mutated chromosomes to the existing population, so that they can compete with the original chromosomes. We use the `^` notation to define the mutation operator to remind you that the mutation is implemented by flipping one bit:

```
def ^ (prob: Double): Unit =  
    chromosomes += chromosomes.map(_ ^ geneticIndices(prob))
```

The `prob` mutation parameter is used to compute the absolute index of the mutating gene, `geneticIndices(prob)`.

Chromosomes

The implementation of the `^` mutation operator on a chromosome consists of mutating the gene of the `indices.chOpIdx` index (line 39) and then updating the list of genes in the chromosome (line 40). The method returns a new chromosome (line 41) that will compete with the original chromosome:

```
def ^ (indices: GeneticIndices): Chromosome[T] = { //39  
    val mutated = code(indices.chOpIdx) ^ indices  
    val xs = Range(0, code.size).map(i =>  
        if(i== indices.chOpIdx) mutated  
        else code(i)).toList //40  
    Chromosome[T](xs) //41  
}
```

Genes

Finally, the mutation operator flips (XOR) the bit at the `indices.geneOpIdx` index:

```
def ^ (indices: GeneticIndices): Gene = {
    val idx = indices.geneOpIdx
    val clonedBits = cloneBits(bits) //42

    clonedBits.flip(idx) //43
    val valOp = decode(clonedBits) //44
    new Gene(id, valOp._1, valOp._2) //45
}
```

The `^` method mutates the cloned bit string, `clonedBits`, (line 42) by flipping the bit at the `indices.geneOpIdx` index (line 43). It decodes and converts the mutated bit string by converting it into a (target value, operator) tuple (line 44). The last step creates a new gene from the target-operator tuple (line 45).

Reproduction

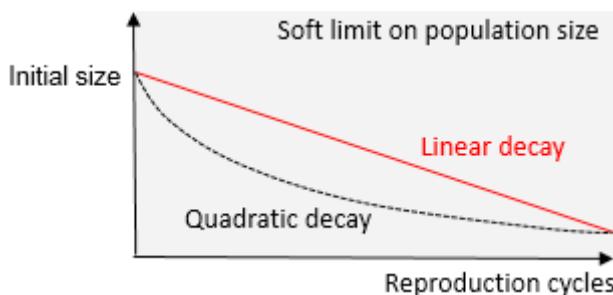
Let's wrap the reproduction cycle into a `Reproduction` class that uses the scoring function, `score`:

```
class Reproduction[T <: Gene](score: Chromosome[T] => Unit)
```

The `mate` reproduction function implements the sequence or workflow of the three genetic operators: `select` for the selection, `+-` (xover) for the crossover, and `^` (mu) for the mutation:

```
def mate(population: Population[T], config: GAConfig,
        cycle: Int): Boolean = (population.size: @switch) match {
    case 0 | 1 | 2 => false //46
    case _ => {
        rand.setSeed(rand.nextInt + System.currentTimeMillis)
        population.select(score, config.softLimit(cycle)) //47
        population +- rand.nextDouble * config.xover //48
        population ^ rand.nextDouble * config.mu //49
        true
    }
}
```

The `mate` method returns false (that is, the reproduction cycle aborts) if the population size is less than 3 (line 46). The chromosomes in the current population are ranked by the increasing cost. The chromosomes with the high cost or low fitness are discarded to comply with the soft limit, `softLimit`, on the population growth (line 47). The randomly generated probability is used as an input to the crossover operation on the entire remaining population (line 48) and as an input to the mutation of the remaining population (line 49):



An illustration of the linear and quadratic soft limit for the population growth

Solver

The `GASolver` class manages the reproduction cycles and the population of chromosomes. The solver is defined as a data transformation of the `ETransform` type using an explicit configuration of the `GAConfig` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 50).

The `GASolver` class implements the `GAMonitor` trait to monitor the population diversity, manage the reproduction cycle, and control the convergence of the optimizer (line 51).

The genetic algorithm-based solver has the following three arguments:

- `config`: This is the configuration of the execution of the genetic algorithm
- `score`: This is the scoring function of a chromosome
- `tracker`: This is the optional tracking function to initialize the monitoring function of `GAMonitor`

The code will be as follows:

```
class GASolver[T <: Gene] (config: GAConfig,  
  score: Chromosome[T] => Unit,
```

```

tracker: Option[Population[T] => Unit] = None)
  extends ETransform[GAConfig](config) //50
    with GAMonitor[T] { //51

  type U = Population[T] //52
  type V = Population[T] //53

  val monitor: Option[Population[T] => Unit] = tracker
  def |>(initialize: => Population[T]): Try[Population[T]] =
    this.|> (initialize()) //54
  override def |> : PartialFunction[U, Try[V]] //55
}

```

This explicit data transformation has to initialize the **U** type of an input element (line 52) and the **V** type of an output element (line 53) for the prediction or optimization method, **|>**. The optimizer takes an initial population as the input and generates a very small population of the fittest chromosomes from which the best solution is extracted (line 55).

The population is generated by the **|>** method (**=> Population[T]**) that takes the constructor of the **Population** class as an argument (line 54).

Let's briefly take a look at the **GAMonitor** monitoring trait assigned to the genetic algorithm. The trait has the following two attributes:

- **monitor**: This is an abstract value to be initialized by classes that implement this trait (line 55).
- **state**: This is the current state of the execution of the genetic algorithm. The initial state of the genetic algorithm is **GA_NOT_RUNNING** (line 56).

The code will be as follows:

```

trait GAMonitor[T <: Gene] extends Monitor {
  self: {
    def |> : PartialFunction[Population[T], Try[Population[T]]]
  } => //55
  val monitor: Option[Population[T] => Unit] //56
  var state: GAState = GA_NOT_RUNNING //57

  def isReady: Boolean = state == GA_NOT_RUNNING
  def start: Unit = state = GA_RUNNING
}

```

```
def isComplete(population: Population[T] ,  
               remainingCycles: Int): Boolean = { ... } //58  
}
```

The state of the genetic algorithm can only be updated in the `|>` method through an instance of the `GAMonitor` class. (line 55).

Here is a subset of the possible state of the execution of the genetic algorithm:

```
sealed abstract class GAState(description: String)  
case class GA_FAILED(description: String)  
  extends GAState(description)  
object GA_RUNNING extends GAState("Running")
```

The solver invokes the `isComplete` method to test the convergence of the optimizer at each reproduction cycle (line 58).

There are two options for estimating that the reproducing cycle is converging:

- **Greedy:** In this approach, the objective is to check whether the n fittest chromosomes have not changed in the last m reproduction cycles
- **Loss function:** This approach is similar to the convergence criteria for the training of supervised learning

Let's consider the following implementation of the genetic algorithm solver:

```
override def |> : PartialFunction[U, Try[V]] = {  
  case population: U if(population.size > 1 && isReady) => {  
    start //59  
    val reproduction = Reproduction[T](score) //60  
  
    @tailrec  
    def reproduce(population: Population[T],  
                 n:Int): Population[T] = { //61  
      if( !reproduction.mate(population, config, n) ||  
          isComplete(population, config.maxCycles -n) )  
        population  
      else  
        reproduce(population, n+1)  
    }  
    reproduce(population, 0)  
    population.select(score, 1.0) //62  
    Try(population)  
  }  
}
```

The optimizing method initializes the state of execution (line 59) and the components of the reproduction cycle (line 60). The reproduction cycle (or an epoch) is implemented as a tail recursion that tests whether the last reproduction cycle has failed or whether the optimization has converged toward a solution (line 61). Finally, the remaining fittest chromosomes are reordered by invoking the `select` method of the `Population` class (line 62).

GA for trading strategies

Let's apply our expertise in genetic algorithms to evaluate different strategies to trade securities using trading signals. Knowledge in trading strategies is not required to understand the implementation of a GA. However, you may want to get familiar with the foundation and terminology of technical analysis of securities and financial markets, as described briefly in the *Technical analysis* section in the *Appendix A, Basic Concepts*.

The problem is to find the best trading strategy to predict the increase or decrease of the price of a security given a set of trading signals. A trading strategy is defined as a set of trading signals ts_j that are triggered or fired when a variable $x = \{x_j\}$, derived from financial metrics such as the price of the security or the daily or weekly trading volume, either exceeds or equals or is below a predefined target value a_j (refer to the *Trading signals and strategy* section in the *Appendix A, Basic Concepts*).

The number of variables that can be derived from price and volume can be very large. Even the most seasoned financial professionals face two challenges, which are as follows:

- Selecting a minimal set of trading signals that are relevant to a given dataset (minimize a cost or unfitness function)
- Turning those trading signals with heuristics derived from personal experience and expertise

Alternative to GA

 The problem described earlier can certainly be solved using one of the machine learning algorithms introduced in the previous chapters. It is just a matter of defining a training set and formulating the problem as minimizing the loss function between the predictor and the training score.

The following table lists the trading classes with their counterpart in the genetic world:

Generic classes	Corresponding securities trading classes
Operator	SOperator
Gene	Signal
Chromosome	Strategy
Population	StrategiesFactory

Definition of trading strategies

A chromosome is the genetic encoding of a trading strategy. A factory class, `StrategyFactory`, assembles the components of a trading strategy: operators, unfitness function, and signals

Trading operators

Let's extend the `Operator` trait with the `SOperator` class to define the operations that we need to trigger the signals. The `SOperator` instance has a single parameter: its identifier, `_id`. The class overrides the `id()` method to retrieve the ID (similarly, the class overrides the `apply` method to convert an ID into an `SOperator` instance):

```
class SOperator(_id: Int) extends Operator {
    override def id: Int = _id
    override def apply(idx: Int): SOperator = new SOperator(idx)
}
```

The operators used by trading signals are the logical operators: < (`LESS_THAN`), > (`GREATER_THAN`), and = (`EQUAL`), as follows:

```
object LESS_THAN extends SOperator(1)
object GREATER_THAN extends SOperator(2)
object EQUAL extends SOperator(3)
```

Each operator of the `SOperator` type is associated with a scoring function by the `operatorFuncMap` map. The scoring function computes the cost (or unfitness) of the signal against a real value or a time series:

```
val operatorFuncMap = Map[Operator, (Double, Double) => Double] (
    LESS_THAN -> ((x: Double, target: Double) => target - x),
    ... )
```

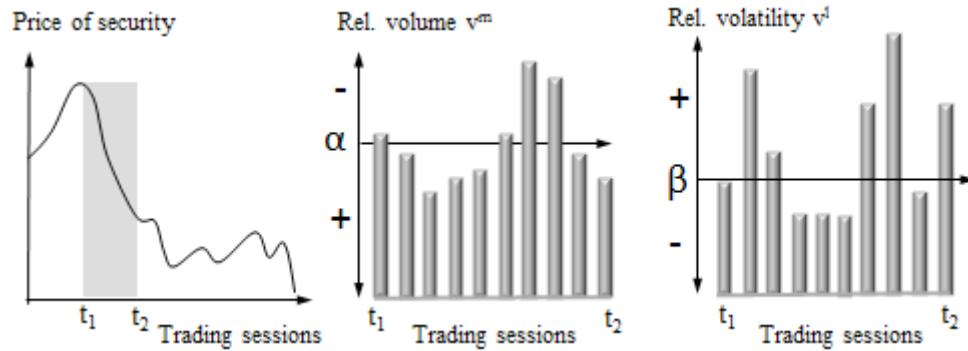
The select method of Population computes the cost value of a signal by quantifying the truthfulness of the predicate. For instance, the unfitness value for a trading signal, $x > 10$, is penalized as $5 - 10 = -5$ for $x = 5$ and credited as $14 - 10 = 4$ if $x = 14$. In this case, the unfitness value is similar to the cost or loss in a discriminative machine learning algorithm.

The cost function

Let's consider the following trading strategy defined as a set of two signals to predict the sudden relative decrease Δp of the price of a security:

- Relative volume v_m with a condition $v_m < \alpha$
- Relative volatility v_l with the condition $v_l > \beta$

Let's take a look at the following graphs:



A chart of the price, relative volume, and relative volatility of a security

As the goal is to model a sudden crash in the stock price, we should reward the trading strategies that predict the steep decrease in the stock price and penalize the strategies that work well only with a small decrease or increase in the stock price. In the case of the trading strategy with two signals, relative volume v_m and relative volatility v_l , n trading sessions, the cost or unfitness function C , and given a relative variation of the stock price and a penalization $w = -\Delta p$ (M2):

$$w_t = -\Delta p_t$$

$$C(p, v^m, v^l | \alpha, \beta) = \sum_{t=0}^{n-1} (\alpha - v_t^m) w_t + (v_t^l - \beta) w_t$$

Trading signals

Let's subclass the `Gene` class to define the trading signal of the `Signal` type as follows:

```
class Signal(id: String, target: Double, op: Operator,
            xt: DblVector, weights: Option[DblVector] = None)
  (implicit quantize: Quantization, encoding: Encoding)
  extends Gene(id, target, op)
```

The `Signal` class requires the following arguments:

- An identifier `id` for the feature
- A target value
- An `op` operator
- An `xt` time series of the `DblVector` type
- The optional `weights` associated with each data point of the time series, `xt`
- An implicit quantization instance, `quantize`
- An implicit encoding scheme

The main purpose of the `Signal` class is to compute its `score` as a chromosome. The chromosome updates its cost by summing the score or weighted score of the signals it contains. The score of the trading signal is simply the summation of the penalty or truthfulness of the signal for each entry of the time series, `ts`:

```
override def score: Double =
  if(!operatorFuncMap.contains(op)) Double.MaxValue
  else {
    val f = operatorFuncMap.get(op).get
    if( weights != None ) xt.zip(weights.get)
      .map{case(x, w) => w*f(x,target)} .sum
    else xt.map( f(_, target)).sum
  }
```

Trading strategies

A trading strategy is an unordered list of trading signals. It makes sense to create a factory class to generate the trading strategies. The `StrategyFactory` class creates strategies of the `List[Signal]` type from an existing pool of signals of the subtype, `Gene`:



A factory pattern for trading signals

The `StrategyFactory` class has two arguments: the number of signals, `nSignals`, in a trading strategy and the implicit Quantization and Encoding instances (line 63):

```
class StrategyFactory(nSignals: Int) //63
  (implicit quantize: Quantization, encoding: Encoding) {
  val signals = new ListBuffer[Signal]
  lazy val strategies: Pool[Signal] //64
  def +=(id: String, target: Double, op: SOperator,
         xt: DblVector, weights: DblVector)
  ...
}
```

The `+=` method takes five arguments: the identifier `id`, the target value, the `op` operation to qualify the class as Gene, the `xt` times series for scoring the signals, and the `weights` associated with the overall cost function. The `StrategyFactory` class generates all possible sequences of signals as trading strategies as lazy values to avoid unnecessary regeneration of the pool on demand (line 64), as follows:

```
lazy val strategies: Pool[Signal] = {
  implicit val ordered = Signal.orderedSignals //70

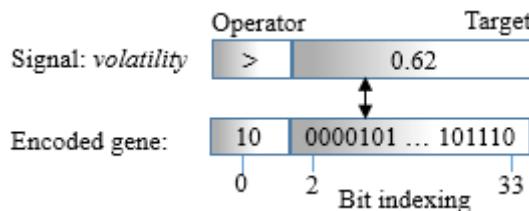
  val XSS = new Pool[Signal] //65
  val treeSet = new TreeSet[Signal] ++= signals.toList //66
  val subsetsIterator = treeSet.subsets(nSignals) //67

  while( subsetsIterator.hasNext) {
    val signalList = subsetsIterator.next.toList //68
    XSS.append(Chromosome[Signal](signalList)) //69
  }
  XSS
}
```

The implementation of the `strategies` value creates a pool of signals `Pool` (line 65) by converting the list of signals to `treeset` (line 66). It breaks down the tree set into unique subtrees of `nSignals` nodes each. It instantiates a `subsetsIterator` iterator to traverse the sequence of subtrees (line 67) and converts them into a list (line 68) as arguments of the new chromosome (trading strategy) (line 69). The procedure to order the signals, `orderedSignals`, in the tree set has to be implicitly defined (line 70) as `val orderedSignals = Ordering.by((signal: Signal) => signal.id)`.

Trading signal encoding

The encoding of trading predicates is the most critical element of the genetic algorithm. In our example, we encode a predicate as a tuple (target value, operator). Let's consider the simple predicate *volatility > 0.62*. The discretization converts the value 0.62 into 32 bits for the instance and a 2-bit representation for the operator:



Encoding of the trading signal: *volatility > 0.62*

IEEE-732 encoding

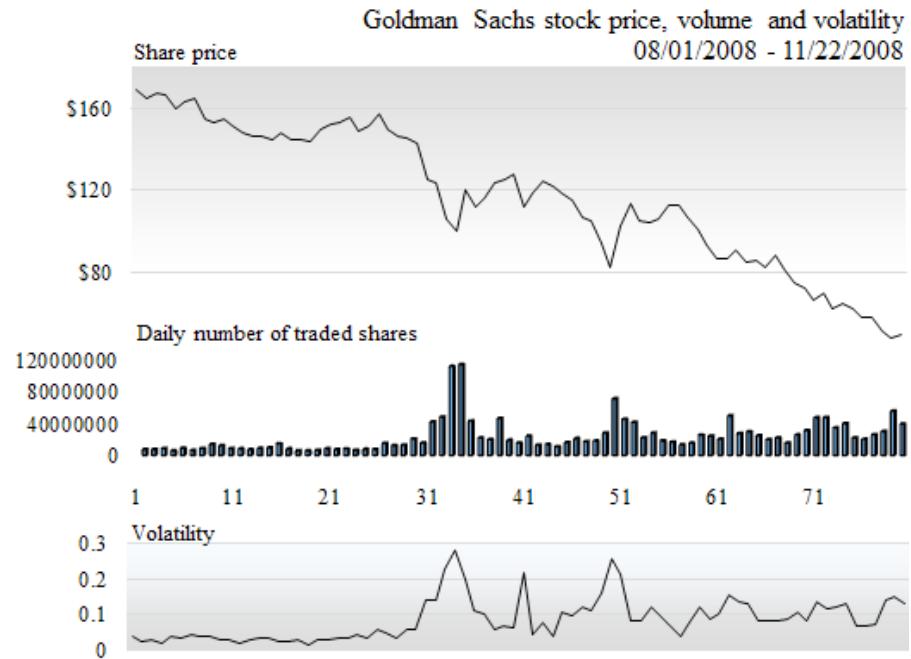
The threshold value for predicates is converted into an integer (the `Int` type or `Long`). The IEEE-732 binary representation of floating point values makes the bit addressing required to apply genetic operators quite challenging. A simple conversion consists of the following:

- encoding e: `(x: Double) => (x*100000).toInt`
- decoding d: `(x: Int) => x*1e-5`

All values are normalized, so there is no risk of overflowing the 32-bit representation.

A test case

The goal is to evaluate which trading strategy was the most relevant (fittest) during the crash of the stock market in fall 2008. Let's consider the stock price of one of the financial institutions, Goldman Sachs, as a proxy of the sudden market decline:



Besides the variation of the price of the stock between two consecutive trading sessions (`dPrice`), the model uses the following parameters (or trading signals):

- `dVolume`: This is the relative variation of the volume between two consecutive trading sessions
- `dVolatility`: This is the relative variation of volatility between two consecutive trading sessions
- `volatility`: This is the relative volatility within a trading session
- `vPrice`: This is the relative difference of the stock opening and closing price

The naming convention for the trading data and metrics is described in the *Trading data* section under *Technical analysis* in the *Appendix A, Basic Concepts*.

The execution of the genetic algorithm requires the following steps:

1. Extraction of model parameters or variables.
2. Generation of the initial population of trading strategies.
3. Setting up the GA configuration parameters with the maximum number of reproduction cycles allowed, the crossover and mutation ratio, and the soft limit function for the population growth.
4. Instantiating the GA algorithm with the scoring/unfitness function.
5. Extracting the fittest trading strategy that can best explain the sharp decline in the price of Goldman Sachs stocks.

Creating trading strategies

The input to the genetic algorithm is the population of trading strategies. Each strategy consists of the combination of three trading signals and each trading signal is a tuple (signal ID, operator, and target value).

The first step is to extract the model parameters as illustrated for the variation of the stock price volume, volatility, and relative volatility between two consecutive trading sessions (line 71):

```
Import YahooFinancials._  
val NUM_SIGNALS = 3  
  
def createStrategies: Try[Pool[Signal]] = {  
    val src = DataSource(path, false, true, 1) //71  
    for { //72  
        price <- src.get(adjClose)  
        dPrice <- delta(price, -1.0)  
        volume <- src.get(volume)  
        dVolume <- delta(volume, 1.0)  
        volatility <- src.get(volatility)  
        dVolatility <- delta(volatility, 1.0)  
        vPrice = src.get(vPrice)  
    } yield { //72  
        val factory = new StrategyFactory(NUM_SIGNALS) //73  
  
        val weights = dPrice //74  
        factory += ("dvolume", 1.1, GREATER_THAN, dVolume, weights)  
        factory += ("volatility", 1.3, GREATER_THAN,  
                    volatility.drop(1), weights)  
        factory += ("vPrice", 0.8, LESS_THAN,  
                    vPrice.drop(1), weights)
```

```

factory += ("dVolatility", 0.9, GREATER_THAN,
            dVolatility, weights)
factory.strategies
}
}

```

The purpose is to generate the initial population of strategies that compete to become relevant to the decline of the price of stocks of Goldman Sachs. The initial population of trading strategies is generated by creating a combination from four trading signals weighted by the variation in the stock price: $\Delta(volume) > 1.1$, $\Delta(volatility) > 1.3$, $\Delta(close-open) < 0.8$, and $volatility > 0.9$.

The delta method computes the variation of a trading variable between consecutive trading sessions. It invokes the `XTSeries.zipWithShift` method, which was introduced in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*:

```

def delta(xt: DblVector, a: Double): Try[DblVector] = Try {
    zipWithShift(xt, 1).map{case (x, y) => a*(y/x - 1.0)}
}

```

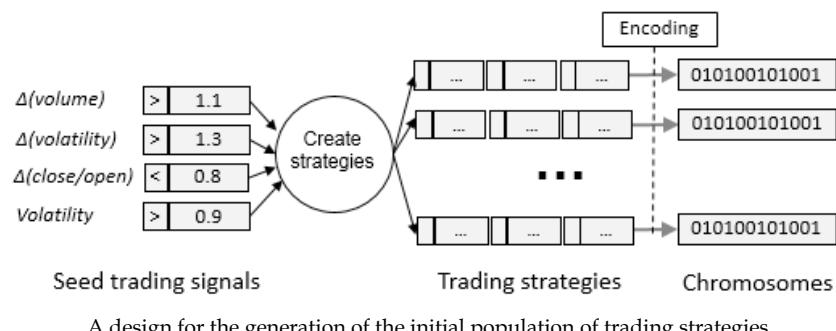
The trading strategies are generated by the `StrategyFactory` class introduced in the previous section (line 73). The weights for the trading strategies are computed as the `dPrice` difference of the price of the stock between two consecutive trading sessions (line 74). The option of unweighted trading strategies is selected by replacing the weights by the average price variation as follows:

```

val avWeights = dPrice.sum/dPrice.size
val weights = Vector.fill(dPrice.size)(avWeights)

```

The generation of the initial population of trading strategies is illustrated in the following diagram:



Configuring the optimizer

The configuration parameters for the execution of the genetic algorithm is categorized as follows:

- Tuning parameters such as crossover, mutation ratio, or soft limit on the population growth
- Data representation parameters such as quantization and encoding
- A scoring scheme

The four configuration parameters for the GA are the maximum number of reproduction cycles (**MAX_CYCLES**) allowed in the execution, the crossover (**XOVER**), the mutation ratio (**MU**), and the soft limit function (**softLimit**) to control the population growth. The soft limit is implemented as a linearly decreasing function of the number of cycles (**n**) to retrain the growth of the population as the execution of the genetic algorithm progresses:

```
val XOVER = 0.8 //Probability(ratio) for cross-over
val MU = 0.4 //Probability(ratio) for mutation
val MAX_CYCLES = 400 //Max. number of optimization cycles

val CUTOFF_SLOPE = -0.003 //Slope linear soft limit
val CUTOFF_INTERCEPT = 1.003 //Intercept linear soft limit
val softLimit = (n: Int) => CUTOFF_SLOPE*n +CUTOFF_INTERCEPT
```

The trading strategies are converted into chromosomes through encoding (line 75). A digitize quantization scheme has to be implicitly defined in order to encode the target value in each trading signal (line 76):

```
implicit val encoding = defaultEncoding //75
val R = 1024 //Quantization ratio
implicit val digitize = new Quantization(R) //76
```

The scoring function computes the cost or unfitness of a trading strategy (chromosome) by applying the score function to each of the three trading signals (genes) it contains (line 77):

```
val scoring = (chr: Chromosome[Signal]) => {
    val signals: List[Gene] = chr.code
    chr.cost = signals.map(_.score).sum //77
}
```

Finding the best trading strategy

The trading strategies generated by the factory in the `createStrategies` method are fed to the genetic algorithm as the initial population (line 79). The upper limit to the population growth is set at eight times the size of the initial population (line 78):

```
createStrategies.map(strategies => {
    val limit = strategies.size <<3 //78
    val initial = Population[Signal](limit, strategies) //79

    val config = GAConfig(XOVER, MU, MAX_CYCLES,softLimit) //80
    val solver = GASolver[Signal](config,scoring,Some(tracker)) //81
        (solver |> initial)
        .map(_.fittest.map(_.symbolic).getOrElse("NA")) match {
            case Success(results) => show(results)
            case Failure(e) => error("GAEval: ", e)
        } //82
    })
})
```

The configuration, `config` (line 80), the scoring function, and optionally a tracker function are all that you need to create and execute the `solver` genetic algorithm (line 81). The partial function generated by the `|>` operator transforms the `initial` population of trading strategies into the two `fittest` strategies (line 82).

The documented source code for the monitoring function, tracker, and miscellaneous methods is available online.

Tests

The cost function C (or unfitness) score of each trading strategy are weighted for the rate of decline of the price of the Goldman Sachs stock. Let's run the following two tests:

- Evaluation of the configuration of the genetic algorithm with the score weighted by the price variation
- Evaluation of the genetic algorithm with an unweighted scoring function

The weighted score

The score is weighted by the variation of the price of the stock GS. The test uses three different sets of crossover and mutation ratios: (0.6, 0.2), (0.3, 0.1), and (0.2, 0.6). The best trading strategy for each scenario is as follows:

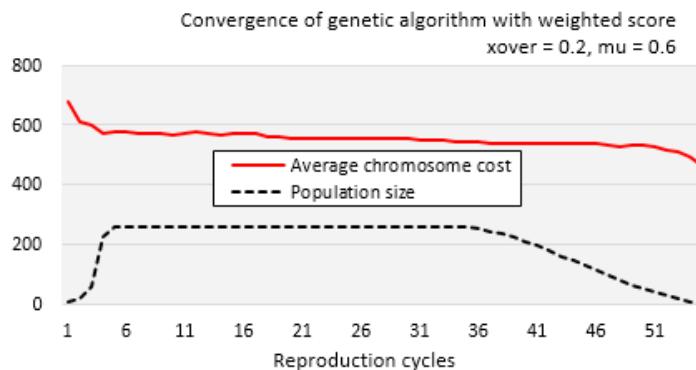
- **0.6-0.2:** $change < 0.82$ $dVolume > 1.17$ $volatility > 1.35$ $cost = 0.0$ $fitness: 1.0E10$
- **0.3-0.1:** $change < 0.42$ $dVolume > 1.61$ $volatility > 1.08$ $cost = 59.18$ $fitness: 0.016$
- **0.2-0.6:** $change < 0.87$ $dVolume < 8.17$ $volatility > 3.91$ $cost = 301.3$ $fitness: 0.003$

The fittest trading strategy for each case does not differ much from the initial population for one or several of the following reasons:

- The initial guess for the trading signals was good
- The size of the initial population is too small to generate genetic diversity
- The test does not take into account the rate of decline of the stock price

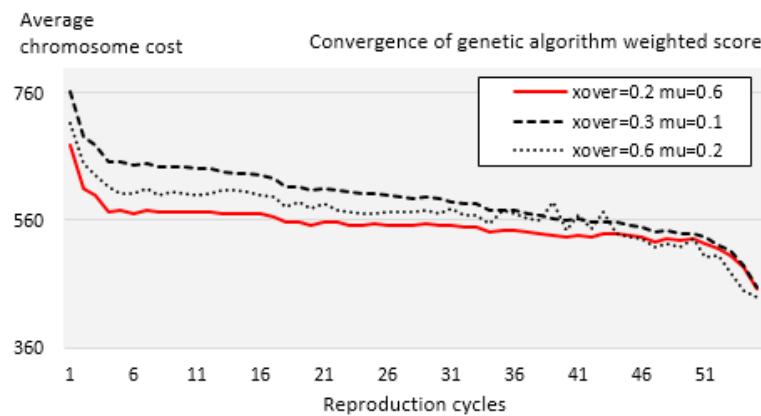
The execution of the genetic algorithm with $crossover = 0.2$ and $mutation = 0.6$ produces a trading strategy that is inconsistent with the first two cases. One possible explanation is the fact that the crossover is applied always to the first of the three genes, forcing the optimizer to converge toward a local minimum.

Let's examine the behavior of the genetic algorithm during execution. We are particularly interested in the convergence of the average chromosome unfitness score. The average chromosome unfitness is the ratio of the total unfitness score for the population over the size of the population. Let's take a look at the following graph:



The convergence of a genetic algorithm for the crossover ratio 0.2 and mutation 0.6 with a weighted score

The GA converges quite quickly and then stabilizes. The size of the population increases through crossover and mutation operations until it reaches the maximum of 256 trading strategies. The soft limit or constraint on the population size kicks in after 23 trading cycles. The test is run again with different values of crossover and mutation ratios, as shown in the following graph:



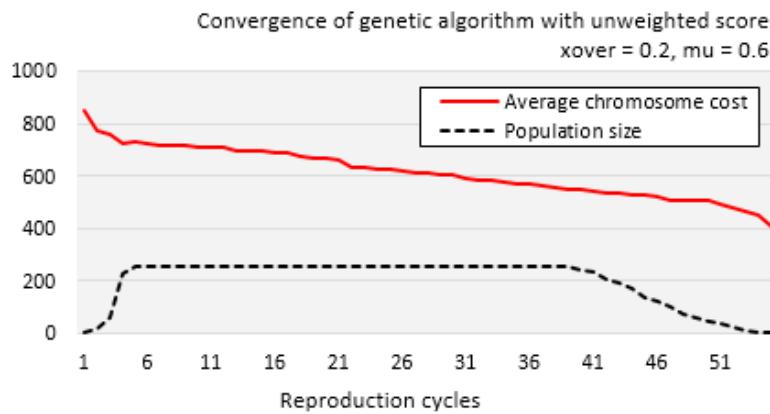
The impact of the crossover and mutation ratio on the convergence of a genetic algorithm with a weighted score

The profile of the execution of the genetic algorithm is not overly affected by the different values of crossover and mutation ratios. The chromosome unfitness score for the high crossover ratio (0.6) oscillates as the execution progresses. In some cases, the unfitness score between chromosomes is so small that the GA recycles the same few trading strategies.

The quick decline in the unfitness of the chromosomes is consistent with the fact that some of the fittest strategies were part of the initial population. It should, however, raise some concerns that the GA locked on a local minimum early on.

The unweighted score

The execution of a test that is similar to the previous one with the unweighted trading strategies (trading strategies that use the average price variation) scoring formula produces some interesting results, as shown in the following graph:



The convergence of a genetic algorithm for the crossover ratio 0.4 and mutation 0.4 with an unweighted score

The profile for the size of the population is similar to the test using weighted scoring. However, the chromosome average cost pattern is somewhat linear. The unweighted (or averaging) adds the rate of decline of the stock price to the score (cost).

The complexity of a scoring function

The complexity of the scoring (or computation of the cost) formula increases the odds of the genetic algorithm not converging properly. The possible solutions to the convergence problem are as follows:

- Make the weighting function additive (less complex)
- Increase the size and diversity of the initial population

Advantages and risks of genetic algorithms

Now, it should be clear that genetic algorithms provide scientists with a powerful toolbox with which to optimize problems that:

- Are poorly understood.
- May have more than one good enough solution.

- Have discrete, discontinuous, and nondifferentiable functions.
- Can be easily integrated with the rules engine and knowledge bases (for example, learning classifiers systems).
- Do not require deep domain knowledge. The genetic algorithm generates new solution candidates through genetic operators. The initial population does not have to contain the fittest solution.
- Do not require knowledge of numerical methods such as the **Newton-Raphson**, **conjugate gradient**, or **BFGS** as optimization techniques, which frighten those with little inclination for mathematics.

However, evolutionary computation is not suitable for problems for which:

- A fitness function cannot be clearly defined
- Finding the global (absolute) minimum or maximum is essential to the problem
- The execution time has to be predictable
- The solution has to be provided in real time or pseudo-real time (streaming data)

Summary

Are you hooked on evolutionary computation, genetic algorithms in particular, and their benefits, limitations as well as some of the common pitfalls? If the answer is yes, then you may find learning classifier systems, introduced in the next chapter, fascinating. This chapter dealt with the following topics:

- Key concepts in evolutionary computing
- The key components and operators of genetic operators
- The pitfalls in defining a fitness or unfitness score using a financial trading strategy as a backdrop
- The challenge of encoding predicates in the case of trading strategies
- Advantages and risks of genetic algorithms
- The process for building a genetic algorithm forecasting tool from the bottom up

The genetic algorithm is an important element of a special class of reinforcement learning, which is introduced in the *Learning classifier systems* section in the next chapter.

11

Reinforcement Learning

This chapter presents the concept of **reinforcement learning**, which is widely used in gaming and robotics. The second part of this chapter is dedicated to **learning classifier systems**, which combine reinforcement learning techniques with evolutionary computing introduced in the previous chapter. Learning classifiers are an interesting breed of algorithms that are not commonly included in literature dedicated to machine learning. I highly recommend that you to read the seminal book on reinforcement learning by R. Sutton and A. Barto [11:1] if you are interested to know about the origin, purpose, and scientific foundation of reinforcement learning.

In this chapter, you will learn the following topics:

- Basic concepts behind reinforcement learning
- A detailed implementation of the Q-learning algorithm
- A simple approach to manage and balance an investment portfolio using reinforcement learning
- An introduction to learning classifier systems
- A simple implementation of extended learning classifiers

The section on **learning classifier systems (LCS)** is mainly informative and does not include a test case.

Reinforcement learning

The need of an alternative to traditional learning techniques arose with the design of the first autonomous systems.

The problem

Autonomous systems are semi-independent systems that perform tasks with a high degree of autonomy. Autonomous systems touch every facet of our life, from robots and self-driving cars to drones. Autonomous devices react to the environment in which they operate. The reaction or action requires the knowledge of not only the current state of the environment but also the previous state(s).

Autonomous systems have specific characteristics that challenge traditional methodologies of machine learning, as listed here:

- Autonomous systems have poorly defined domain knowledge because of the sheer number of possible combinations of states.
- Traditional nonsequential supervised learning is not a practical option because of the following:
 - Training consumes significant computational resources, which are not always available on small autonomous devices
 - Some learning algorithms are not suitable for real-time prediction
 - The models do not capture the sequential nature of the data feed
- Sequential data models such as hidden Markov models require training sets to compute the emission and state transition matrices (as explained in *The hidden Markov model* section in *Chapter 7, Sequential Data Models*), which are not always available. However, a reinforcement learning algorithm benefits from a hidden Markov model if some of the states are unknown. These algorithms are known as behavioral hidden Markov models [11:2].
- Genetic algorithms are an option if the search space can be constrained heuristically. However, genetic algorithms have unpredictable response time, which makes them impractical for real-time processing.

A solution – Q-learning

Reinforcement learning is an algorithmic approach to understanding and ultimately automating goal-based decision making. Reinforcement learning is also known as control learning. It differs from both supervised and unsupervised learning techniques from the knowledge acquisition standpoint: **autonomous**, automated systems, or devices learn from direct and real-time interaction with their environment. There are numerous practical applications of reinforcement learning from robotics, navigation agents, drones, adaptive process control, game playing, and online learning, to scheduling and routing problems.

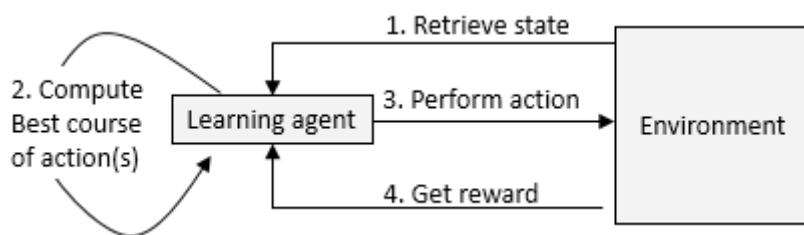
Terminology

Reinforcement learning introduces new terminologies as listed here, which are quite different from that of older machine learning techniques:

- **Environment:** This is any system that has states and mechanisms to transition between states. For example, the environment for a robot is the landscape or facility it operates.
- **Agent:** This is an automated system that interacts with the environment.
- **State:** The state of the environment or system is the set of variables or features that fully describe the environment.
- **Goal or absorbing state or terminal state:** This is the state that provides a higher discounted cumulative reward than any other state. A high cumulative reward prevents the best policy from being dependent on the initial state during training.
- **Action:** This defines the transition between states. The agent is responsible for performing or at least recommending an action. Upon execution of the action, the agent collects a reward (or punishment) from the environment.
- **Policy:** This defines the action to be selected and executed for any state of the environment.
- **Best policy:** This is the policy generated through training. It defines the model in Q-learning and is constantly updated with any new episode.
- **Reward:** This quantifies the positive or negative interaction of the agent with the environment. Rewards are essentially the training set for the learning engine.
- **Episode:** This defines the number of steps necessary to reach the goal state from an initial state. Episodes are also known as trials.
- **Horizon:** This is the number of future steps or actions used in the maximization of the reward. The horizon can be infinite, in which case the future rewards are discounted in order for the value of the policy to converge.

Concepts

The key component in reinforcement learning is a **decision-making agent** that reacts to its environment by selecting and executing the best course of actions and being rewarded or penalized for it [11:3]. You can visualize these agents as robots navigating through an unfamiliar terrain or a maze. Robots use reinforcement learning as part of their reasoning process after all. The following diagram gives the overview architecture of the reinforcement learning agent:



The four state transitions of reinforcement learning

The agent collects the state of the environment, selects, and then executes the most appropriate action. The environment responds to the action by changing its state and rewarding or punishing the agent for the action.

The four steps of an episode or learning cycle are as follows:

1. The learning agent retrieves or is notified of a new state of the environment.
2. The agent evaluates and selects the action that may provide the highest reward.
3. The agent executes the action.
4. The agent collects the reward or penalty and applies it to calibrate the learning algorithm.

 **Reinforcement versus supervision**
The training process in reinforcement learning rewards features that maximize a value or return. Supervised learning rewards features that meet a predefined labeled value. Supervised learning can be regarded as forced learning.

The action of the agent modifies the state of the system, which in turn notifies the agent of the new operational condition. Although not every action will trigger a change in the state of the environment, the agent collects the reward or penalty nevertheless. At its core, the agent has to design and execute a sequence of actions to reach its goal. This sequence of actions is modeled using the ubiquitous Markov decision process (refer to the *Markov decision processes* section in *Chapter 7, Sequential Data Models*).

Dummy actions



It is important to design the agent so that actions may not automatically trigger a new state of the environment. It is easy to think about a scenario in which the agent triggers an action just to evaluate its reward without affecting the environment significantly.

A good metaphor for such a scenario is the *rollback* of the action. However, not all environments support such a *dummy* action, and the agent may have to run Monte-Carlo simulations to try out an action.

Value of a policy

Reinforcement learning is particularly suited to problems for which long-term rewards can be balanced against short-term rewards. A policy enforces the trade-off between short-term and long-term rewards. It guides the behavior of the agent by mapping the state of the environment to its actions. Each policy is evaluated through a variable known as the **value of a policy**.

Intuitively, the value of a policy is the sum of all the rewards collected as a result of the sequence of actions taken by the agent. In practice, an action over the policy farther in the future obviously has a lesser impact than the next action from a state S_t to a state S_{t+1} . In other words, the impact of future actions on the current state has to be discounted by a factor, known as the *discount coefficient for future rewards* < 1 .

Transition and rewards matrices



The transition and emission matrices have been introduced in the *The hidden Markov model* section in *Chapter 7, Sequential Data Models*.

The optimum policy π^* is the agent's sequence of actions that maximizes the future reward discounted to the current time.

The following table introduces the mathematical notation of each component of reinforcement learning:

Notation	Description
$S = \{s_i\}$	These are the states of the environment
$A = \{a_j\}$	These are the actions on the environment
$\Pi_t = p(a_t s_t)$	This is the policy (or strategy) of the agent
$V^\pi(s_t)$	This is the value of the policy at a state
$p_t = p(s_{t+1} s_t, a_t)$	These are the state transition probabilities from the state s_t to the state s_{t+1}
$r_t = p(r_{t+1} s_t, s_{t+1}, a_t)$	This is the reward of an action a_t for a state s_t
R_t	This is the expected discounted long-term return
γ	This is the coefficient to discount the future rewards

The purpose is to compute the maximum expected reward R_t from any starting state s_k as the sum of all discounted rewards to reach the current state s_t . The value V^π of a policy π at the state s_t is the maximum expected reward R_t given the state s_t .

M1: The cumulative reward R_t and value function $V^\pi(st)$ for the state st given a policy π and a discount rate γ is defined as:



$$R_t = \sum_{k=0}^{+\infty} \gamma^k r_{t+1+k}$$

$$V^\pi(s_t) = E\{R_t | s_t\}$$

The Bellman optimality equations

The problem of finding the optimal policies is indeed a nonlinear optimization problem whose solution is iterative (dynamic programming). The expression of the value function V^π of a policy π can be formulated using the Markovian state transition probabilities p_t .

M2: The value function $V^\pi(s_t)$ for a state s_t and future state s_k with a reward r_k using the transition probability $p_{k'}$ given a policy π and a discount rate γ is defined as:

$$V^\pi(s_t) = \sum_{a \in A} \pi_a \sum_k \{p_k(r_k + \gamma \cdot V^\pi(s_k))\}$$

$$V^*(s_t) = \max_\pi V^\pi(s_t)$$

$V^*(s_t)$ is the optimal value of the state s_t across all the policies. The equations are known as the Bellman optimality equations.

The curse of dimensionality

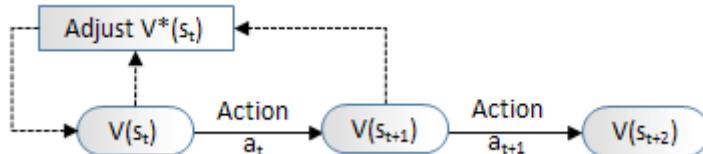
The number of states for a high-dimension problem (large-feature vector) becomes quickly unsolvable. A workaround is to approximate the value function and reduce the number of states by sampling. The application test case introduces a very simple approximation function.

If the environment model, state, action, and rewards, as well as transition between states, are completely defined, the reinforcement learning technique is known as model-based learning. In this case, there is no need to explore a new sequence of actions or state transitions. Model-based learning is similar to playing a board game in which all combinations of steps that are necessary to win are completely known.

However, most practical applications using sequential data do not have a complete, definitive model. Learning techniques that do not depend on a fully defined and available model are known as model-free techniques. These techniques require exploration to find the best policy for any given state. The remaining sections in this chapter deal with model-free learning techniques, and more specifically, the temporal difference algorithm.

Temporal difference for model-free learning

Temporal difference is a model-free learning technique that samples the environment. It is a commonly used approach to solve the Bellman equations iteratively. The absence of a model requires a discovery or **exploration** of the environment. The simplest form of exploration is to use the value of the next state and the reward defined from the action to update the value of the current state, as described in the following diagram:



An illustration of the temporal difference algorithm

The iterative feedback loop used to adjust the value action on the state plays a role similar to the backpropagation of errors in artificial neural networks or minimization of the loss function in supervised learning. The adjustment algorithm has to:

- Discount the estimate value of the next state using the discount rate γ
- Strike a balance between the impact of the current state and the next state on updating the value at time t using the learning rate α

The iterative formulation of the first Bellman equation predicts $V^\pi(s_t)$, the value function of state s_t from the value function of the next state s_{t+1} . The difference between the predicted value and the actual value is known as the temporal difference error abbreviated as δ_t .

M3: The formula for tabular temporal difference δ_t for a value function $V(s_t)$ at state s_t , a learning rate α , a reward r_t , and a discount rate γ is defined as:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

$$\hat{V}^\pi(s_t) = V^\pi(s_t) + \alpha \delta_t$$

An alternative to evaluating a policy using the value of the state $V^\pi(s_t)$ is to use the value of taking an action on a state s_t known as the value of action (or action-value) $Q^\pi(s_t, a_t)$.

 M4: The definition of the value Q of action at a state s_t as the expectation of a reward R_t for an action a_t on a state s_t is defined as:

$$Q_t^\pi = Q^\pi(s_t, a_t) = E(R_t | s_t, a_t)$$

There are two methods to implement the temporal difference algorithm:

- **On-policy:** This is the value for the next best action that uses the policy
- **Off-policy:** This is the value for the next best action that does not use the policy

Let's consider the temporal difference algorithm using an off-policy method and its most commonly used implementation: Q-learning.

Action-value iterative update

Q-learning is a model-free learning technique using an off-policy method. It optimizes the action-selection policy by learning an action-value function. Like any machine learning technique that relies on convex optimization, the Q-learning algorithm iterates through actions and states using the quality function, as described in the following mathematical formulation.

The algorithm predicts and discounts the optimum value of action $\max\{Q_i\}$ for the current state s_t and action a_t on the environment to transition to the state s_{t+1} .

Similar to genetic algorithms that reuse the population of chromosomes in the previous reproduction cycle to produce offspring, the Q-learning technique strikes a balance between the new value of the quality function Q_{t+1} and the old value Q_t using the learning rate α . Q-learning applies temporal difference techniques to the Bellman equation for an off-policy methodology.

 M5: The Q-learning action-value updating formula for a given policy π , set of states $\{s_t\}$, a set of actions $\{a_t\}$ associated with each state s_t , a learning rate α , and a discount rate γ is given by:

$$\tilde{Q}_t^\pi = Q_t^\pi + \alpha \left[r_{t+1} + \gamma \max_{a_{t+1}} Q_{t+1}^\pi - Q_t^\pi \right] \quad Q_t^\pi = Q^\pi(s_t, a_t)$$

- A value 1 for the learning rate α discards the previous state, while a value 0 discards learning
- A value 1 for the discount rate γ uses long-term rewards only, while a value 0 uses the short-term reward only

Q-learning estimates the cumulative reward discounted for future actions.

 **Q-learning as reinforcement learning**
Q-learning qualifies as a reinforcement learning technique because it does not strictly require labeled data and training. Moreover, the Q-value does not have to be a continuous, differentiable function.

Let's apply our hard-earned knowledge of reinforcement learning to management and optimization of a portfolio of exchange-traded funds.

Implementation

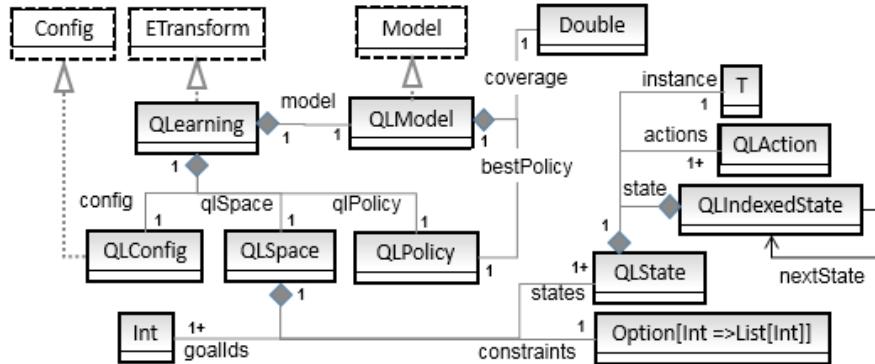
Let's implement the Q-learning algorithm in Scala.

Software design

The key components of the implementation of the Q-learning algorithm are defined as follows:

- The `QLearning` class implements training and prediction methods. It defines a data transformation of the `ETransform` type using an explicit configuration of the `QLConfig` type.
- The `QLSpace` class has two components: a sequence of states of the `QLState` type and the identifier `id` of one or more goal states within the sequence.
- A state, `QLState`, contains a sequence of `QLAction` instances used in its transition to another state and a reference to the object or `instance` for which the state is to be evaluated and predicted.
- An indexed state, `QLIndexedState`, indexes a state in the search toward the goal state.
- An optional constraint function that limits the scope of the search for the next most rewarding action from the current state.
- The model of the `QLModel` type is generated through training. It contains the best policy and the accuracy for a model.

The following diagram shows the key components of the Q-learning algorithm:



The UML components diagram of the Q-learning algorithm

The states and actions

The `QLAction` class specifies the transition of one state with a `from` identifier to another state with the `to` identifier, as shown here:

```
class QLAction(val from: Int, val to: Int)
```

Actions have a Q value (or action-value), a reward, and a probability. The implementation defines these three values in three separate matrices: Q for the action values, R for rewards, and P for probabilities, in order to stay consistent with the mathematical formulation.

A state of the `QLState` type is fully defined by its identifier, `id`, the list of `actions` to transition to some other states, and a `prop` property of the parameterized type, as shown in the following code:

```
class QLState[T](val id: Int,
                 val actions: List[QLAction] = List.empty,
                 val instance: T)
```

The state might not have any actions. This is usually the case of the goal or absorbing state. In this case, the list is empty. The parameterized `instance` is a reference to the object for which the state is computed.

The next step consists of creating the graph or search space.

The search space

The search space is the container responsible for any sequence of states. The `QLSpace` class takes the following parameters:

- The sequence of all the possible states
- The ID of one or several states that have been selected as goals

 **Why multiple goals?**
There is absolutely no requirement that a state space must have a single goal. You can describe a solution to a problem as reaching a threshold or meeting one of the several conditions. Each condition can be defined as a state goal.

The `QLSpace` class is implemented as follows:

```
class QLSpace[T](states: Seq[QLState[T]], goals: Array[Int]) {  
    val statesMap = states.map(st =>(st.id, st)) //1  
    val goalStates = new HashSet[Int] () ++ goals //2  
  
    def maxQ(state: QLState[T],  
             policy: QLPolicy): Double //3  
    def init(state0: Int) //4  
    def nextStates(st: QLState[T]): Seq[QLState[T]] //5  
    ...  
}
```

The constructor of the `QLSpace` class generates a map, `statesMap`. It retrieves the state using its `id` value (line 1) and the array of goals, `goalStates` (line 2). Furthermore, the `maxQ` method computes the maximum action-value, `maxQ`, for a state given a policy (line 3). The implementation of the `maxQ` method is described in the next section.

The `init` method selects an initial state for training episodes (line 4). The state is randomly selected if the `state0` argument is invalid:

```
def init(state0: Int): QLState[T] =  
    if(state0 < 0) {  
        val seed = System.currentTimeMillis+Random.nextLong  
        states((new Random(seed)).nextInt(states.size-1))  
    }  
    else states(state0)
```

Finally, the `nextStates` method retrieves the list of states resulting from the execution of all the actions associated with the `st` state (line 5).

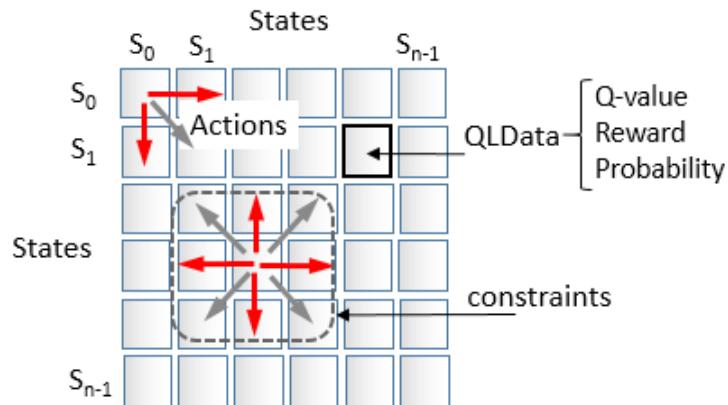
The QLSpace search space is actually created by the apply factory method defined in the QLSpace companion object, as shown here:

```
def apply[T] (goals: Array[Int], instances: Seq[T],
  constraints: Option[Int => List[Int]]): QLSpace[T] = { //6
  val r = Range(n, instances.size)

  val states = instances.zipWithIndex.map{ case(x, n) => {
    val validStates = constraints.map( _(n)).getOrElse(r)
    val actions = validStates.view
      .map(new QLAction(n, _)).filter(n != _.to) //7
    QLState[T](n, actions, x)
  }}
  new QLSpace[T](states, goals)
}
```

The apply method creates a list of states using the instances set, the goals, and the constraints constraining function as inputs (line 6). Each state creates its list of actions. The actions are generated from this state to any other states (line 7).

The search space of states is illustrated in the following diagram:



The state transition matrix with QLData (Q-value, reward, and probability)

The constraints function limits the scope of the actions that can be triggered from any given state, as illustrated in the preceding diagram.

The policy and action-value

Each action has an action-value, a reward, and a potentially probability. The probability variable is introduced to simply model the hindrance or adverse condition for an action to be executed. If the action does not have any external constraint, the probability is 1. If the action is not allowed, the probability is 0.

Dissociating a policy from states

 The action and states are the edges and vertices of the search space or search graph. The policy defined by the action-values, rewards, and probabilities is completely dissociated from the graph. The Q-learning algorithm initializes the reward matrix and updates the action-value matrix independently of the structure of the graph.

The `QLData` class is a container for three values: reward, probability, and a value variable for the Q-value, as shown here:

```
class QLData(val reward: Double, val probability: Double) {  
    var value: Double = 0.0  
    def estimate: Double = value*probability  
}
```

Reward and punishment

 The probability in the `QLData` class represents the hindrance or difficulty to reach one state from another state. Nothing prevents you from using the probability as a negative reward or punishment. However, its proper definition is to create a soft constraint of a state transition for a small subset of a state. For most applications, the overwhelming majority of state transitions have a probability of 1.0, and therefore, rely on the reward to guide the search toward the goal.

The `estimate` method adjusts the Q-value, `value`, with the probability to reflect any external condition that can impede the action.

Mutable data

You might wonder why the `QLData` class defines a value as a variable instead of a value as recommended by the best Scala coding practices [11:4]. The reason being that an instance of an immutable class can be created for each action or state transition that requires you to update the `value` variable.

The training of the Q-learning model entails iterating across several episodes, each episode being defined as a multiple iteration. For instance, the training of a model with 400 states for 10 episodes of 100 iterations can potentially create 160 million instances of `QLData`. Although not quite elegant, mutability reduces the load on the JVM garbage collector.

Next, let's create a simple schema or class, `QLInput`, to initialize the reward and probability associated with each action as follows:

```
class QLInput(val from: Int, val to: Int,
            val reward: Double, val probability: Double = 1.0)
```

The first two arguments are the identifiers for the `from` source state and the `to` target state for this specific action. The last two arguments are the `reward`, collected at the completion of the action, and its `probability`. There is no need to provide an entire matrix. Actions have a reward of 1 and a probability of 1 by default. You only need to create an input for actions that have either a higher reward or a lower probability.

The number of states and a sequence of input define the policy of the `QLPolicy` type. It is merely a data container, as shown here:

```
class QLPolicy(input: Seq[QLInput]) {
    val numStates = Math.sqrt(input.size).toInt //8

    val qlData = input.map(qlIn =>
        new QLData(qlIn.reward, qlIn.prob)) //9

    def setQ(from: Int, to: Int, value: Double): Unit =
        qlData(from*numStates + to).value = value //10

    def Q(from: Int, to: Int): Double =
        qlData(from*numStates + to).value //11
    def EQ(from: Int, to: Int): Double =
        qlData(from*numStates + to).estimate //12
    def R(from: Int, to: Int): Double =
        qlData(from*numStates + to).reward //13
    def P(from: Int, to: Int): Double =
        qlData(from*numStates + to).probability //14
}
```

The number of states, `numStates`, is the square root of the number of elements of the initial input matrix, `input` (line 8). The constructor initializes the `qlData` matrix of the `QLData` type with the input data, `reward`, and `probability` (line 9). The `QLPolicy` class defines the shortcut methods to update (line 10) and retrieve (line 11) the value, the estimate (line 12), the reward (line 13), and the probability (line 14).

The Q-learning components

The `QLearning` class encapsulates the Q-learning algorithm, and more specifically, the action-value updating equation. It is a data transformation of the `ETransform` type with an explicit configuration of the `QLConfig` type (line 16) (refer to the *Monadic data transformation* section in *Chapter 2, Hello World!*):

```
class QLearning[T] (conf: QLConfig,
  qlSpace: QLSpace[T], qlPolicy: QLPolicy) //15
  extends ETransform[QLConfig](conf) { //16

  type U = QLState[T] //17
  type V = QLState[T] //18

  val model: Option[QLModel] = train //19
  def train: Option[QLModel]
  def nextState(iSt: QLIndexedState[T]): QLIndexedState[T]

  override def |> : PartialFunction[U, Try[V]]
  ...
}
```

The constructor takes the following parameters (line 15):

- `config`: This is the configuration of the algorithm
- `qlSpace`: This is the search space
- `qlPolicy`: This is the policy

The `model` is generated or trained during the instantiation of the class (refer to the *Design template for classifier* section in the *Appendix A, Basic Concepts*) (line 19). The Q-learning algorithm is implemented as an explicit data transformation; therefore, the `U` type of the input element and the `V` type of the output element to the `|>` predictor are initialized as `QLState` (lines 17 and 18).

The configuration of the Q-learning algorithm, `QLConfig`, specifies the learning rate, `alpha`, the discount rate, `gamma`, the maximum number of states (or length) of an episode, `episodeLength`, the number of episodes (or epochs) used in training, `numEpisodes`, and the minimum coverage, `minCoverage`, required to select the best policy as follows:

```
case class QLConfig(val alpha: Double,
    val gamma: Double,
    val episodeLength: Int,
    val numEpisodes: Int,
    val minCoverage: Double) extends Config
```

The `QLearning` class has two constructors defined in its companion object that initializes the policy either from an input matrix of states or from a function that compute the reward and probabilities:

- The client code specifies the `input` function to initialize the state of the Q-learning algorithm from the input data
- The client code specifies the functions to generate the `reward` and `probability` for each action or state transition

The first constructor for the `QLearning` class passes the initialization of states => `Seq[QLInput]`, the sequence of references of `instances` associated with the states, and the `constraints` scope constraining function as an argument, besides the configuration and the goals (line 20):

```
def apply[T](config: QLConfig, //20
    goals: Array[Int],
    input: => Seq[QLInput],
    instances: Seq[T],
    constraints: Option[Int => List[Int]] = None): QLearning[T] = {
    new QLearning[T](config,
        QLSpace[T](goals, instances, constraints),
        new QLPolicy(input))
}
```

The second constructor passes the input data, `xt` (line 21), the `reward` function (line 22), and the `probability` function (line 23) as well as the sequence of references of `instances` associated with the states and the `constraints` scope constraining function as arguments:

```
def apply[T](config: QLConfig,
    goals: Array[Int],
    xt: DblVector, //21
```

```
    reward: (Double, Double) => Double, //22
    probability: (Double, Double) => Double, //23
    instances: Seq[T].
    constraints: Option[Int =>List[Int]] =None) : QLearning[T] ={

        val r = Range(0, xt.size)
        val input = new ArrayBuffer[QLInput] //24
        r.foreach(i =>
            r.foreach(j =>
                input.append(QLInput(i, j, reward(xt(i), xt(j)),
                    probability(xt(i), xt(j)))))

        )
    )
    new QLearning[T](config,
        QLSpace[T](goals, instances, constraints),
        new QLPolicy(input))
}
```

The reward and probability matrices are used to initialize the `input` state (line 24).

The Q-learning training

Let's take a look at the computation of the best policy during training. First, we need to define a `QLModel` model class with the `bestPolicy` optimum policy (or path) and its coverage as parameters:

```
class QLModel(val bestPolicy: QLPolicy,
             val coverage: Double) extends Model
```

The creation of `model` consists of executing multiple episodes to extract the best policy. The training is executed in the `train` method: Each episode starts with a randomly selected state, as shown in the following code:

```
def train: Option[QLModel] = Try {
    val completions = Range(0, config.numEpisodes)
        .map(epoch => if(train(-1)) 1 else 0).sum //25
    completions.toDouble/config.numEpisodes //26
}
.map( coverage => {
    if(coverage > config.minCoverage)
        Some(new QLModel(qlPolicy, coverage)) //27
    else None
}).get
```

The `train` method iterates through the generation of the best policy starting from a randomly selected state `config.numEpisodes` times (line 25). The state coverage is calculated as the percentage of times the search ends with the goal state (line 26). The training succeeds only if the coverage exceeds a threshold value, `config.minAccuracy`, specified in the configuration.

The quality of the model

The implementation uses the accuracy to measure the quality of the model or best policy. The F_1 measure (refer to the *Assessing a model* section in *Chapter 2, Hello World!*), is not appropriate because there are no false positives.

The `train(state0: Int)` method does the heavy lifting at each episode (or epoch). It triggers the search by selecting either the `state0` initial state or a `r` random generator with a new seed, if `state0` is `< 0`, as shown in the following code:

```
case class QLIndexedState[T] (val state: QLState[T],
    val iter: Int)
```

The `QLIndexedState` utility class keeps track of the `state` at a specific iteration, `iter`, within an episode or epoch:

```
def train(state0: Int): Boolean = {
    @tailrec
    def search(iSt: QLIndexedState[T]): QLIndexedState[T]

    val finalState = search(
        QLIndexedState(qlSpace.init(state0), 0)
    )
    if( finalState.index == -1) false //28
    else qlSpace.isGoal(finalState.state) //29
}
```

The implementation of `search` for the goal state(s) from a `state0` predefined or random is a textbook implementation of the Scala tail recursion. Either the recursive search ends if there are no more states to consider (line 28) or the goal state is reached (line 29).

Tail recursion to the rescue

Tail recursion is a very effective construct to apply an operation to every item of a collection [11:5]. It optimizes the management of the function stack frame during the recursion. The annotation triggers a validation of the condition necessary for the compiler to optimize the function calls, as shown here:

```
@tailrec
def search(iSt: QLIndexedState[T]): QLIndexedState[T] = {
    val states = qlSpace.nextStates(iSt.state) //30

    if( states.isEmpty || iSt.iter >= config.episodeLength) //31
        QLIndexedState(iSt.state, -1)

    else {
        val state = states.maxBy(s =>
            qlPolicy.R(iSt.state.id, s.id)) //32
        if( qlSpace.isGoal(state) )
            QLIndexedState(state, iSt.iter) //33

        else {
            val fromId = iSt.state.id
            val r = qlPolicy.R(fromId, state.id)
            val q = qlPolicy.Q(fromId, state.id) //34

            val nq = q + config.alpha*(r +
                config.gamma * qlSpace.maxQ(state, qlPolicy)-q) //35
            qlPolicy.setQ(fromId, state.id, nq) //36
            search(QLIndexedState(state, iSt.iter+1))
        }
    }
}
```

Let's dive into the implementation for the Q action-value updating equation. The `search` method implements the **M5** mathematical expression for each recursion.

The recursion uses the `QLIndexedState` utility class (state, iteration number in the episode) as an argument. First, the recursion invokes the `nextStates` method of `QLSpace` (line 30) to retrieve all the states associated with the `st` current state through its actions, as shown here:

```
def nextStates(st: QLState[T]): Seq[QLState[T]] =
    if( st.actions.isEmpty )
        Seq.empty[QLState[T]]
    else
        st.actions.map(ac => statesMap.get(ac.to) )
```

The search completes and returns the current `state` if the length of the episode (maximum number of states visited) is reached or the `goal` is reached or there is no further state to transition to (line 31). Otherwise, the recursion computes the state to which the transition generates the higher reward `R` from the current policy (line 32). The recursion returns the state with the highest reward if it is one of the goal states (line 33). The method retrieves the current `q` action-value (line 34) and `r` reward matrices from the policy, and then applies the equation to update the action-value (line 35). The method updates the action-value `Q` with the new value `nq` (line 36).

The action-value updating equation requires the computation of the maximum action-value associated with the current state, which is performed by the `maxQ` method of the `QLSpace` class:

```
def maxQ(state: QLState[T], policy: QLPolicy): Double = {
    val best = states.filter(_ != state) //37
        .maxBy(st => policy.EQ(state.id, st.id)) //38
    policy.EQ(state.id, best.id)
}
```

The `maxQ` method filters out the current state (line 37) and then extracts the best state, which maximizes the policy (line 38).

Reachable goal

The algorithm does not require the goal state to be reached for every episode. After all, there is no guarantee that the goal will be reached from any randomly selected state. It is a constraint on the algorithm to follow a positive gradient of the rewards when transitioning between states within an episode. The goal of the training is to compute the best possible policy or sequence of states from any given initial state. You are responsible for validating the model or best policy extracted from the training set, independent from the fact that the goal state is reached for every episode.

The validation

A commercial application may require multiple types of validation mechanisms regarding the states transition, reward, probability, and Q-value matrices.

One critical validation is to verify that the user-defined constraints function does not create a dead end in the search or training of Q-learning. The constraints function establishes the list of states that can be accessed from a given state through actions. If the constraints are too tight, some of the possible search paths may not reach the goal state. Here is a simple validation of the constraints function:

```
def validateConstraints(numStates: Int,  
    constraints: Int => List[Int]): Boolean =  
  Range(0, numStates).exists( constraints(_).isEmpty )
```

The prediction

The last functionality of the QLearning class is the prediction using the model created during training. The `|>` method predicts the optimum state transition (or action) from a given state, `state0`:

```
override def |> : PartialFunction[U, Try[V]] = {  
  case state0: U if(isModel) => Try {  
    if(state0.isGoal) state0 //39  
    else nextState(QLIndexedState[T](state0, 0)).state //40  
  }  
}
```

The `|>` data transformation returns itself if the `state0` input state is the goal (line 39) or computes the best outcome, `nextState`, (line 40) using another tail recursion, as follows:

```
@tailrec  
def nextState(iSt: QLIndexedState[T]): QLIndexedState[T] = {  
  val states = qlSpace.nextStates(iSt.state) //41  
  
  if( states.isEmpty || iSt.iter >= config.episodeLength)  
    iSt //42  
  else {  
    val fromId = iSt.state.id  
    val qState = states.maxBy(s => //43  
      model.map(_.bestPolicy.R(fromId, s.id)).getOrElse(-1.0))  
    nextState(QLIndexedState[T](qState, iSt.iter+1)) //44  
  }  
}
```

The `nextState` method executes the following sequence of invocations:

1. Retrieve the eligible states that can be transitioned to from the current state, `ist.state` (line 41).
2. Return the states if there are no more states or if the method does not converge within the maximum number of allowed iterations, `config.episodeLength` (line 42).
3. Extracts the state, `qState`, with the most rewarding policy (line 43).
4. Increment the `ist.iter` iteration counter (line 44).

The exit condition

The prediction ends when no more states are available or the maximum number of iterations within the episode is exceeded. You can define a more sophisticated exit condition. The challenge is that there is no explicit error or loss variable/function that can be used except the temporal difference error.

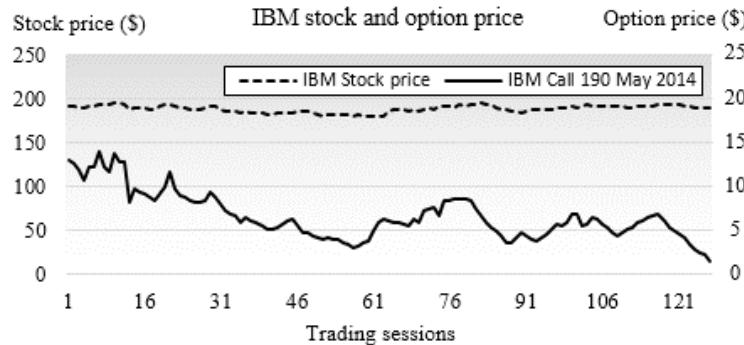
The `|>` prediction method returns either the best possible state or `None` if the model cannot be created during training.

Option trading using Q-learning

The Q-learning algorithm is used in many financial and market trading applications [11:6]. Let's consider the problem of computing the best strategy to trade certain types of options given some market conditions and trading data.

The **Chicago Board Options Exchange (CBOE)** offers an excellent online tutorial on options [11:7]. An option is a contract that gives the buyer the right but not the obligation to buy or sell an underlying asset at a specific price on or before a certain date (refer to the *Options trading* section under *Finances 101* in the *Appendix A, Basic Concepts*.) There are several option pricing models, the Black-Scholes stochastic partial differential equations being the most recognized [11:8].

The purpose of the exercise is to predict the price of an option on a security for N days in the future according to the current set of observed features derived from the time to expiration, price of the security, and volatility. Let's focus on the call options of a given security, IBM. The following chart plots the daily price of the IBM stock and its derivative call option for May 2014 with a strike price of \$190:



The IBM stock and Call \$190 May 2014 pricing in May-Oct 2013

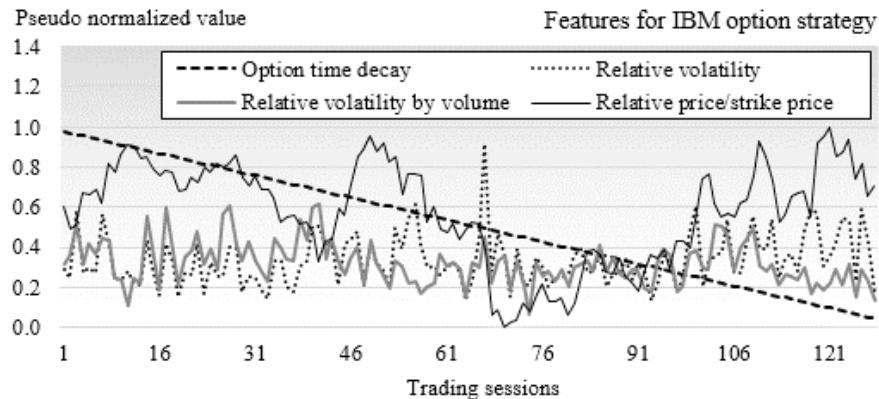
The price of an option depends on the following parameters:

- Time to expiration of the option (time decay)
- The price of the underlying security
- The volatility of returns of the underlying asset

The pricing model usually does not take into account the variation in trading volume of the underlying security. Therefore, it would be quite interesting to include it in our model. Let's define the state of an option using the following four normalized features:

- **Time decay** (`timeToExp`): This is the time to expiration once normalized over $[0, 1]$.
- **Relative volatility** (`volatility`): This is the relative variation of the price of the underlying security within a trading session. It is different from the more complex volatility of returns defined in the Black-Scholes model, for example.
- **Volatility relative to volume** (`vltyByVol`): This is the relative volatility of the price of the security adjusted for its trading volume.
- **Relative difference between the current price and strike price** (`priceToStrike`): This measures the ratio of the difference between price and strike price to the strike price.

The following graph shows the four normalized features for the IBM option strategy:



Normalized relative stock price volatility, volatility relative to trading volume, and price relative to strike price for the IBM stock

The implementation of the option trading strategy using Q-learning consists of the following steps:

1. Describing the property of an option
2. Defining the function approximation
3. Specifying the constraints on the state transition

The OptionProperty class

Let's select $N = 2$ as the number of days in the future for our prediction. Any longer-term prediction is quite unreliable because it falls outside the constraint of the discrete Markov model. Therefore, the price of the option two days in the future is the value of the reward – profit or loss.

The OptionProperty class encapsulates the four attributes of an option (line 45) as follows:

```
class OptionProperty(timeToExp: Double, volatility: Double,
    vltyByVol: Double, priceToStrike: Double) { //45

    val toArray = Array[Double](
        timeToExp, volatility, vltyByVol, priceToStrike
    )
}
```

A modular design



The implementation avoids subclassing the `QLState` class to define the features of our option pricing model. The state of the option is a parameterized prop parameter for the state class.

The OptionModel class

The `OptionModel` class is a container and a factory for the properties of the option. It creates the list of `propsList` option properties by accessing the data source of the four features introduced earlier. It takes the following parameters:

- The symbol of the security.
- The strike price for the `strikePrice` option.
- The source of data, `src`.
- The minimum time decay or time to expiration, `minTDecay`. Out-of-the-money options expire worthless and in-the-money options have very different price behavior as they get closer to the expiration date (refer to the *Options trading* section in the *Appendix A, Basic Concepts*). Therefore, the last `minTDecay` trading sessions prior to the expiration date are not used in the training of the model.
- The number of steps (or buckets), `nSteps`. It is used in approximating the values of each feature. For instance, an approximation of four steps creates four buckets [0, 25], [25, 50], [50, 75], and [75, 100].

The implementation of the `OptionModel` class is as follows:

```
class OptionModel(symbol: String,  strikePrice: Double,
                 src: DataSource, minExpT: Int, nSteps: Int) {

    val propsList = (for {
        price <- src.get(adjClose)
        volatility <- src.get(volatility)
        nVolatility <- normalize(volatility)
        vltyByVol <- src.get(volatilityByVol)
        nvltbyVol <- normalize(vltyByVol)
        priceToStrike <- normalize(price.map(p =>
            (1.0 - strikePrice/p)))
    } yield {
        nVolatility.zipWithIndex./:(List[OptionProperty]()) { //46
            case(xs, (v,n)) => {
                val normDecay = (n + minExpT).toDouble/
                    (price.size + minExpT) //47
                OptionProperty(priceToStrike, v, n, normDecay)
            }
        }
    })
}
```

```

        new OptionProperty(normDecay, v, nVltyByVol(n),
            priceToStrike(n)) :: xs
    }
}.drop(2).reverse //48
})
.getOrElse(List.empty[OptionProperty].)
}

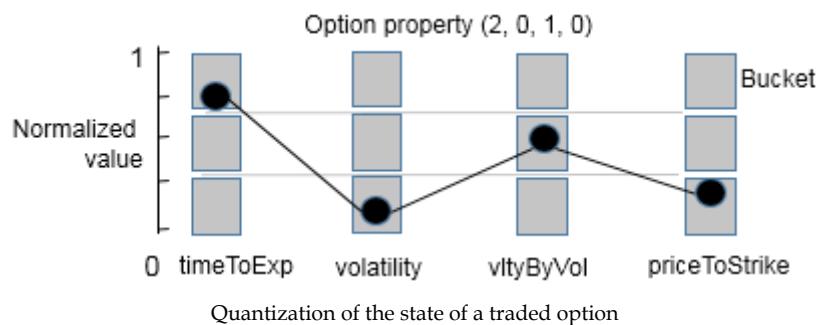
def quantize(o: DblArray): Map[Array[Int], Double]
}

```

The factory uses the `zipWithIndex` Scala method to represent the index of the trading sessions (line 46). All feature values are normalized over the interval [0, 1], including the time decay (or time to expiration) of the `normDecay` option (line 47). The instantiation of the `OptionModel` class generates a list of `OptionProperty` elements if the constructor succeeds (line 48), an empty list otherwise.

Quantization

The four properties of the option are continuous values, normalized as a probability [0, 1]. The states in the Q-learning algorithm are discrete and require a quantization or categorization known as a **function approximation**; although a function approximation scheme can be quite elaborate [11:9]. Let's settle for a simple linear categorization, as illustrated in the following diagram:



The function approximation defines the number of states. In this example, a function approximation that converts a normalized value into three intervals or buckets generates $3^4 = 81$ states or potentially $3^8 \cdot 3^4 = 6480$ actions! The maximum number of states for l buckets function approximation and n features is l^n with a maximum number of $l^n \cdot l^n$ actions.

Quantization or function approximation guidelines

The design of the function to approximate the state of options has to address the following two conflicting requirements:



- Accuracy demands a fine-grained approximation
- Limited computation resources restrict the number of states, and therefore, level of approximation

The `quantize` method of the `OptionModel` class converts the normalized value of each option property of features into an array of bucket indices. It returns a map of profit and loss for each bucket keyed on the array of bucket indices, as shown in the following code:

```
def quantize(o: DblArray): Map[Array[Int], Double] = {
    val mapper = new mutable.HashMap[Int, Array[Int]] //49
    val _acc = new NumericAccumulator[Int] //50

    val acc = propsList.view.map(_.toArray)
        .map( toArrayInt(_)) //51
        .map(ar => {
            val enc = encode(ar) //52
            mapper.put(enc, ar)
            enc
        }).zip(o)
        .:/(_acc) {
            case (acc, (t,y)) => { //53
                acc += (t, y)
                acc
            }
        }
    acc.map {case (k, (v,w)) => (k, v/w)} //54
        .map {case( k,v) => (mapper(k), v)}.toMap
}
```

The method creates a `mapper` instance to index the array of buckets (line 49). An `acc` accumulator of the `NumericAccumulator` type extends `Map[Int, (Int, Double)]` and computes the tuple (number of occurrences of features on each buckets and the sum of increase or decrease of the option price) (line 50). The `toArrayInt` method converts the value of each option property (`timeToExp`, `volatility`, and so on) into the index of the appropriate bucket (line 51). The array of indices is then encoded (line 52) to generate the id or index of a state. The method updates the accumulator with the number of occurrences and the total profit and loss for a trading session for the option (line 53). It finally computes the reward on each action by averaging the profit and loss on each bucket (line 54).

A view is used in the generation of the list of `OptionProperty` to avoid unnecessary object creation.

The source code for the `toArrayInt` and `encode` methods and `NumericAccumulator` is documented and available online.

Putting it all together

The final piece of the puzzle is the code that configures and executes the Q-learning algorithm on one or several options on a security, IBM:

```
val STOCK_PRICES = "resources/data/chap11/IBM.csv"
val OPTION_PRICES = "resources/data/chap11/IBM_O.csv"
val QUANTIZER = 4
val src = DataSource(STOCK_PRICES, false, false, 1) //55

val model = for {
    option <- Try(createOptionModel(src)) //56
    oPrices <- DataSource(OPTION_PRICES, false).extract //57
    _model <- createModel(option, oPrices) //58
} yield _model
```

The preceding implementation creates the Q-learning model with the following steps:

1. Extract the historical prices for the IBM stock by instantiating a data source, `src` (line 55).
2. Create an `option` model (line 56).
3. Extract the historical prices `oPrices` for option call \$190 May 2014 (line 57).
4. Create the model, `_model`, with a `goalStr` predefined goal (line 58).

The code is as follows:

```
val STRIKE_PRICE = 190.0
val MIN_TIME_EXPIRATION = 6

def createOptionModel(src: DataSource): OptionModel =
    new OptionModel("IBM", STRIKE_PRICE, src,
        MIN_TIME_EXPIRATION, QUANTIZER)
```

Let's take a look at the `createModel` method that takes the option pricing model, option, and the historical prices for `oPrices` options as arguments:

```
val LEARNING_RATE = 0.2
val DISCOUNT_RATE = 0.7
val MAX_EPISODE_LEN = 128
val NUM_EPISODES = 80

def createModel(option: OptionModel, oPrices: DblArray,
               alpha: Double, gamma: Double): Try[QLModel] = Try {

    val qPriceMap = option.quantize(oPrices) //59
    val numStates = qPriceMap.size

    val qPrice = qPriceMap.values.toVector //60
    val profit = zipWithShift(qPrice, 1).map{case(x,y) => y - x} //61
    val maxProfitIndex = profit.zipWithIndex.maxBy(_._1)._2 //62

    val reward = (x: Double, y: Double)
                 => Math.exp(30.0*(y - x)) //63
    val probability = (x: Double, y: Double) =>
        if(y < 0.3*x) 0.0 else 1.0 //64

    if( !validateConstraints(profit.size, neighbors)) //65
        throw new IllegalStateException(" ... ")

    val config = QLConfig(alpha, gamma,
                          MAX_EPISODE_LEN, NUM_EPISODES) //66
    val instances = qPriceMap.keySet.toSeq.drop(1)
    QLearning[Array[Int]](config, Array[Int](maxProfitIndex),
                          profit, reward, probability,
                          instances, Some(neighbors)).getModel //67
}
```

The method quantizes the option prices map, `oPrices` (line 59), extracts the historical option prices, `qPrice` (line 60), computes the profit as the difference in the price of the option between two consecutive trading sessions (line 61), and computes the index, `maxProfitIndex`, of the trading session with the highest profit (line 62). The state with the `maxProfitIndex` index is selected as the goal.

The input matrix is automatically generated using the reward and probability functions. The reward function rewards the state transition proportionally to the profit (line 63). The probability function punishes the state transition for which the loss $y - x$ is greater than $0.3*x$ by setting the probability value to 0 (line 64).



Initialization of rewards and probabilities

In our example, the reward and probability matrices are automatically generated through two functions. An alternative approach consists of initializing these two matrices using either historical data or educated guesses.

The validateConstraints method of the QLearning companion object validates the neighbors constraints function, as described in the *The validation* section (line 65).

The last two steps consists of creating a configuration, config, for the Q-learning algorithm (line 66) and training the model by instantiating the QLearning class with the appropriate parameters, including the neighbors method that defines the neighboring states for any given state (line 67). The neighbors method is described in the documented source code available online.

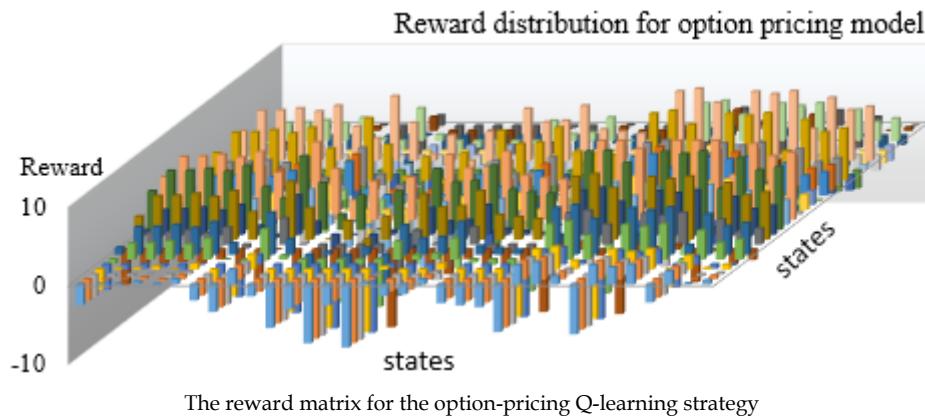


The anti-goal state

The goal state is the state with the highest assigned reward. It is a heuristic to reward a strategy for a good performance. However, it is conceivable and possible to define an anti-goal state with the highest assigned penalty or the lowest assigned reward to guide the search away from some condition.

Evaluation

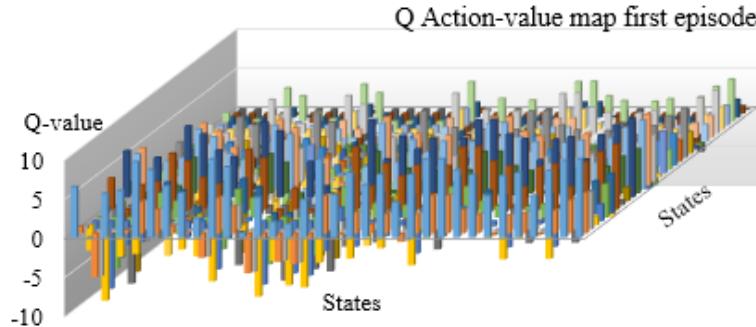
Besides the function approximation, the size of the training set has an impact on the number of states. A well-distributed or large training set provides at least one value for each bucket created by the approximation. In this case, the training set is quite small and only 34 out of 81 buckets have actual values. As result, the number of states is 34. The initialization of the Q-learning model generates the following reward matrix:



The graph visualizes the distribution of the rewards computed from the profit and loss of the option. The xy plane represents the actions between states. The states' IDs are listed on x and y axes. The z axis measures the actual value of the reward associated with each action.

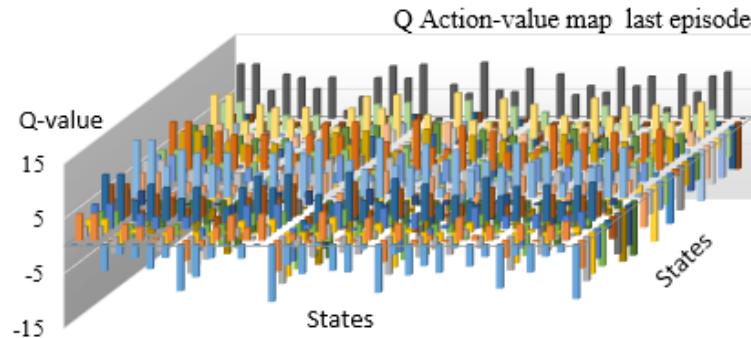
The reward reflects the fluctuation in the price of the option. The price of an option has a higher volatility than the price of the underlying security.

The xy reward matrix R is rather highly distributed. Therefore, we select a small value for the learning rate 0.2 to reduce the impact of the previous state on the new state. The value for the discount rate 0.7 accommodates the fact that the number of states is limited. There is no reason to compute the future discounted reward using a long sequence of states. The training of the policies generates the following action-value matrix Q of 34 states by 34 states after the first episode:



The Q action-value matrix for the first episode (epoch)

The distribution of the action-values between states at the end of the first episode reflects the distribution of the reward across state-to-state action. The first episode consists of a sequence of nine states from an initial randomly selected state to the goal state. The action-value map is compared to the map generated after 20 episodes in the following graph:



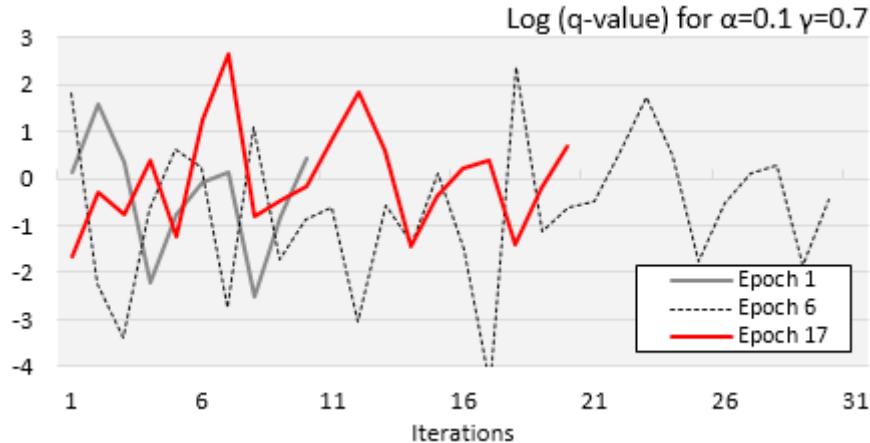
The Q Action-Value matrix for the last episode (epoch)

The action-value map at the end of the last episode shows some clear patterns. Most of the rewarding actions transition from a large number of states (X axis) to a smaller number of states (Y axis). The chart illustrates the following issues with the small training sample:

- The small size of the training set forces us to use an approximate representation of each feature. The purpose is to increase the odds that most buckets have, that is, at least one data point.
- However, a loose function approximation or quantization tends to group quite different states into the same bucket.
- The bucket with a very low number can potentially mischaracterize one property or feature of a state.

Reinforcement Learning

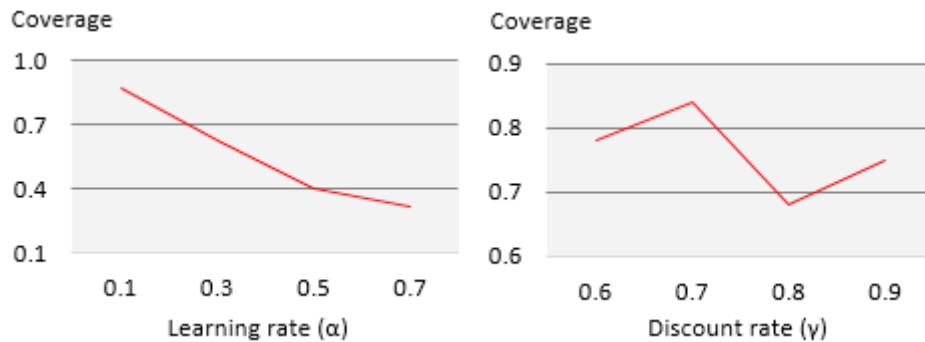
The next test is to display the profile of the log of the Q-value (`QLData.value`) as the recursive search (or training) progress for different episodes or epochs. The test uses a learning rate $\alpha = 0.1$ and a discount rate $\gamma = 0.9$.



The profile of the log (Q-Value) for different epochs during Q-learning training

The preceding chart illustrates the fact that the Q-value for each profile is independent of the order of the epochs during training. However, the length of the profile (or number of iterations to reach the goal state) depends on the initial state, which is selected randomly, in this example.

The last test consists of evaluating the impact of the learning rate and discount rate on the coverage of the training:



Training coverage versus learning rate and discount rate

The coverage (percentage of an episode or epoch for which the goal state is reached) decreases as the learning rate increases. The result confirms the general rule of using learning rate < 0.2 . The similar test to evaluate the impact of the discount rate on the coverage is inconclusive.

Pros and cons of reinforcement learning

Reinforcement learning algorithms are ideal for the following problems:

- Online learning
- The training data is small or nonexistent
- A model is nonexistent or poorly defined
- Computation resources are limited

However, these techniques perform poorly in the following cases:

- The search space (number of possible actions) is large because the maintenance of the states, action graph, and rewards matrix becomes challenging
- The execution is not always predictable in terms of scalability and performance

Learning classifier systems

J. Holland introduced the concept of **learning classifier systems (LCS)** more than 30 years ago as an extension to evolutionary computing [11:10].

Learning classifier systems are a kind of rule-based system with general mechanisms for processing rules in parallel, for adaptive generation of new rules, and for testing the effectiveness of new rules.

However, the concept started to get the attention of computer scientists only a few years ago, with the introduction of several variants of the original concept, including **extended learning classifiers (XCS)**. Learning classifier systems are interesting because they combine rules, reinforcement learning, and genetic algorithms.

Disclaimer

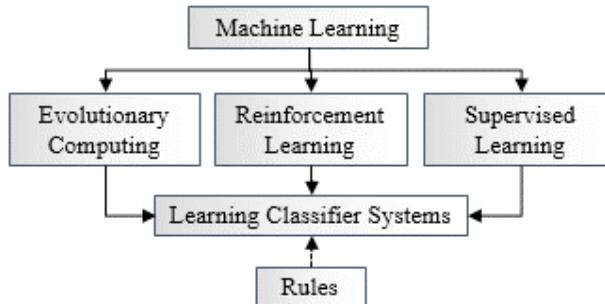
The implementation of the extended learning classifier is presented for informational purposes only. Validating XCS against a known and labeled population of rules is a very significant endeavor. The source code snippet is presented only to illustrate the different components of the XCS algorithm.

Introduction to LCS

Learning classifier systems merge the concepts of reinforcement learning, rule-based policies, and evolutionary computing. This unique class of learning algorithms represents the merger of the following research fields [11:11]:

- Reinforcement learning
- Genetic algorithms and evolutionary computing
- Supervised learning
- Rule-based knowledge encoding

Let's take a look at the following diagram:



A diagram of the scientific disciplines required for learning classifier systems

Learning classifier systems are an example of **complex adaptive systems**. A learning classifier system has the following four components:

- **A population of classifiers or rules:** This evolves over time. In some cases, a domain expert creates a primitive set of rules (core knowledge). In other cases, the rules are randomly generated prior to the execution of the learning classifier system.
- **A genetic algorithm-based discovery engine:** This generates new classifiers or rules from the existing population. This component is also known as the **rules discovery module**. The rules rely on the same pattern of evolution of organisms introduced in the previous chapter. The rules are encoded as strings or bit strings to represent a condition (predicate) and action.
- **A performance or evaluation function:** This measures the positive or negative impact of the actions from the fittest classifiers or policies.

- **A reinforcement learning component:** This rewards or punishes the classifiers that contribute to the action, as seen in the previous section. The rules that contribute to an action that improves the performance of the system are rewarded, while those that degrade the performance of the system are punished. This component is also known as the credit assignment module.

Why LCS?

Learning classifier systems are particularly appropriate to problems in which the environment is constantly changing and are the combinations of a learning strategy and an evolutionary approach to build and maintain a knowledge base [11:12].

Supervised learning methods alone can be effective on large datasets, but they require either a significant amount of labeled data or a reduced set of features to avoid overfitting. Such constraints may not be practical in the case of ever-changing environments.

The last 20 years have seen the introduction of many variants of learning classifier systems that belong to the following two categories:

- Systems for which accuracy is computed from the correct predictions and that apply the discovery to a subset of those correct classes. They incorporate elements of supervised learning to constrain the population of classifiers. These systems are known to follow the **Pittsburgh approach**.
- Systems that explore all the classifiers and apply rule accuracy to the genetic selection of the rules. Each individual classifier is a rule. These systems are known to follow the **Michigan approach**.

The rest of this section is dedicated to the second type of learning classifiers – more specifically, extended learning classifier systems. In a context of LCS, the term *classifier* refers to the predicate or rule generated by the system. From this point on, the term *rule* replaces the term *classifier* to avoid confusion with the more common definition of classification.

Terminology

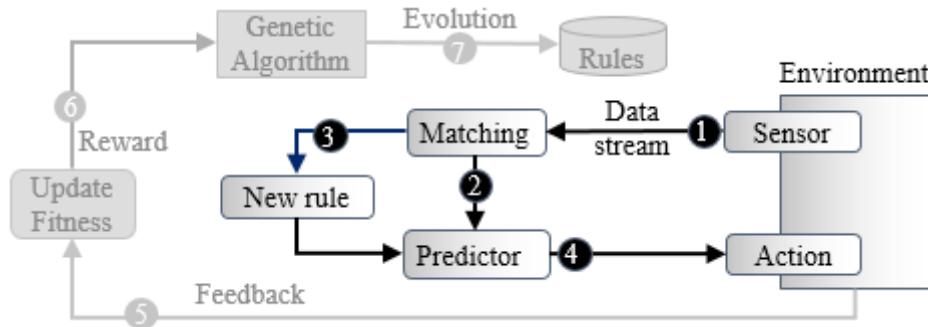
Each domain of research has its own terminology and LCS is no exception. The terminology of LCS consists of the following terms:

- **Environment:** These are the environment variables in the context of reinforcement learning.
- **Agent:** An agent used in reinforcement learning.

- **Predicate:** A clause or fact using the format, *variable-operator-value*, and usually implemented as (operator, variable value); for example, *Temperature-exceeds-87F* or ('Temperature', 87F), *Hard drive-failed* or ('Status hard drive', FAILED), and so on. It is encoded as a gene in order to be processed by the genetic algorithm.
- **Compound predicate:** This is the composition of several predicates and Boolean logic operators, which is usually implemented as a logical tree (for example, ((*predicate1 AND predicate2*) OR *predicate3*) is implemented as OR (AND (*predicate1, predicate2*), *predicate3*)). It uses a chromosome representation.
- **Action:** This is a mechanism that alters the environment by modifying the value of one or several of its parameters using a format (*type of action, target*); for example, *change thermostat settings, replace hard drive*, and so on.
- **Rule:** This is a formal first-order logic formula using the format *IF compound predicate THEN sequence of action*; for example, *IF gold price < \$1140 THEN sell stock of oil and gas producing companies*.
- **Classifier:** This is a rule in the context of an LCS.
- **Rule fitness or score:** This is identical to the definition of the fitness or score in the genetic algorithm. In the context of an LCS, it is the probability of a rule to be invoked and fired in response to the change in environment.
- **Sensors:** These are environment variables monitored by an agent; for example, the temperature and hard drive status.
- **Input data stream:** This is the flow of data generated by sensors. It is usually associated with online training.
- **Rule matching:** This is a mechanism to match a predicate or compound predicate with a sensor.
- **Covering:** This is the process of creating new rules to match a new condition (sensor) in the environment. It generates the rules by either using a random generator or mutating existing rules.
- **Predictor:** This is an algorithm to find the action with the maximum number of occurrences within a set of matching rules.

Extended learning classifier systems

Similar to reinforcement learning, the XCS algorithm has an **exploration** phase and an **exploitation** phase. The exploitation process consists of leveraging the existing rules to influence the target environment in a profitable or rewarding manner:

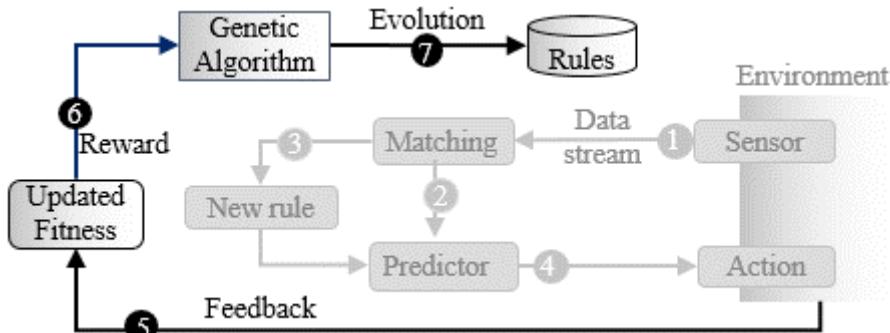


The exploitation component of the XCS algorithm

The following list describes each numbered block:

1. Sensors acquire new data or events from the system.
2. Rules for which the condition matches the input event are extracted from the current population.
3. A new rule is created if no match is found in the existing population. This process is known as covering.
4. The chosen rules are ranked by their fitness values, and the rules with the highest predicted outcome are used to trigger the action.

The purpose of exploration components is to increase the rule base as a population of the chromosomes that encode these rules.



Exploration components of the XCS algorithm

The following list describes each numbered block of the block diagram:

1. Once the action is performed, the system rewards the rules for which the action has been executed. The reinforcement learning module assigns credit to these rules.
2. Rewards are used to update the rule fitness, applying evolutionary constraints to the existing population.
3. The genetic algorithm updates the existing population of classifiers/rules using operators such as crossover and mutation.

XCS components

This section describes the key classes of the XCS. The implementation leverages the existing design of the genetic algorithm and the reinforcement learning. It is easier to understand the inner workings of the XCS algorithm with a concrete application.

Application to portfolio management

Portfolio management and trading have benefited from the application of extended learning classifiers [11:13]. The use case is the management of a portfolio of exchange-traded funds in an ever-changing financial environment. Contrary to stocks, exchange-traded funds are representative of an industry-specific group of stocks or the financial market at large. Therefore, the price of these ETFs is affected by the following macroeconomic changes:

- Gross domestic product
- Inflation
- Geopolitical events
- Interest rates

Let's select the value of the 10-year Treasury yield as a proxy for the macroeconomic conditions, for the sake of simplicity.

The portfolio has to be constantly adjusted in response to any specific change in the environment or market condition that affects the total value of the portfolio, and this can be done by referring to the following table:

XCS component	Portfolio management
Environment	This is the portfolio of securities defined by its composition, total value, and the yield of the 10-year Treasury bond
Action	This is the change in the composition of the portfolio

XCS component	Portfolio management
Reward	This is the profit and loss of the total value of the portfolio
Input data stream	This is the feed of the stock and bond price quotation
Sensor	This is the trading information regarding securities in the portfolio such as price, volume, volatility, yield, and the yield of the-10 year Treasury bond
Predicate	This is the change in the composition of the portfolio
Action	This rebalances a portfolio by buying and selling securities
Rule	This is the association of trading data with the rebalancing of a portfolio

The first step is to create an initial set of rules regarding the portfolio. This initial set can be created randomly, like the initial population of a genetic algorithm or defined by a domain expert.

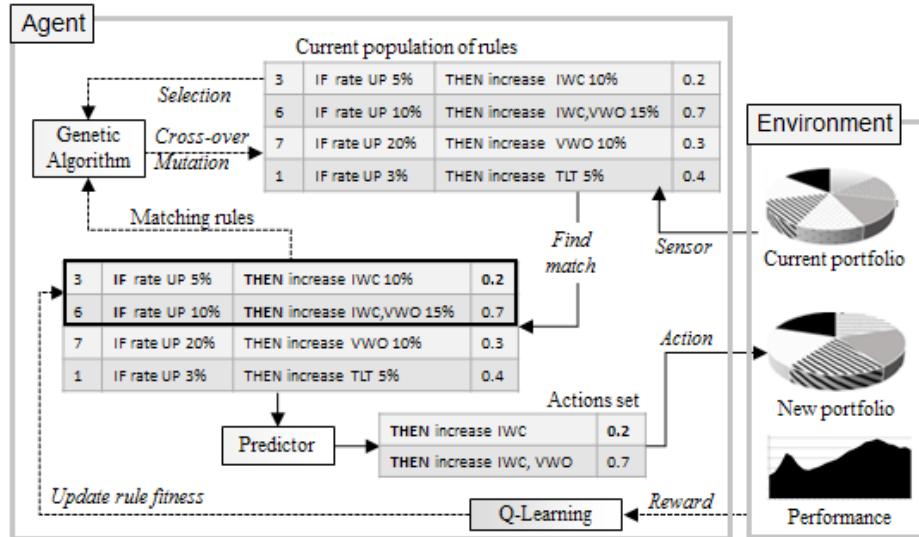
The XCS initial population

Rules or classifiers are defined and/or refined through evolution. Therefore, there is no absolute requirement for the domain expert to set up a comprehensive knowledge base. In fact, rules can be randomly generated at the start of the training phase. However, seeding the XCS initial population with a few relevant rules improves the odds of having the algorithm converge quickly.

You are invited to initialize the population of rules with as many relevant and financially sound trading rules as possible. Over time, the execution of the XCS algorithm will confirm whether or not the initial rules are indeed appropriate. The following diagram describes the application of the XCS algorithm to the composition of a portfolio of ETFs, such as VWO, TLT, IWC, and so on, with the following components:

- The population of trading rules
- An algorithm to match rules and compute the prediction
- An algorithm to extract the actions sets
- The Q-learning module to assign a credit or reward to the selected rules
- The genetic algorithm to evolve the population of rules

Let's take a look at the following diagram:



An overview of the XCS algorithm to optimize the portfolio allocation

The agent responds to the change in the allocation of ETFs in the portfolio by matching one of the existing rules.

Let's build the XCS agent from the ground.

The XCS core data

There are three types of data that are manipulated by the XCS agent:

- **Signal:** This is the trading signal.
- **XcsAction:** This is the action on the environment. It subclasses a Gene defined in the genetic algorithm.
- **XcsSensor:** This is the sensor or data from the environment.

The Gene class was introduced for the evaluation of the genetic algorithm in the *Trading signals* section in *Chapter 10, Genetic Algorithms*. The agent creates, modifies, and deletes actions. It makes sense to define these actions as mutable genes, as follows:

```
class XcsAction(sensorId: String, target: Double)
  (implicit quantize: Quantization, encoding: Encoding) //1
  extends Gene(sensorId, target, EQUAL)
```

The quantization and encoding of the `XcsAction` into a `Gene` has to be explicitly declared (line 1). The `XcsAction` class has the identifier of the `sensorId` sensor and the target value as parameters. For example, the action to increase the number of shares of ETF, VWO in the portfolio to 80 is defined as follows:

```
val vwoTo80 = new XcsAction("VWO", 80.0)
```

The only type of action allowed in this scheme is setting a value using the `EQUAL` operator. You can create actions that support other operators such as `+=` used to increase an existing value. These operators need to implement the `operator` trait, as explained in the *Trading operators* section in *Chapter 10, Genetic Algorithms*.

Finally, the `XcsSensor` class encapsulates the `sensorId` identifier for the variable and value of the sensor, as shown here:

```
case class XcsSensor(val sensorId: String, val value: Double)
val new10ytb = XcsSensor("10yTBYield", 2.76)
```

Setters and getters

In this simplistic scenario, the sensors retrieve a new value from an environment variable. The action sets a new value to an environment variable. You can think of a sensor as a get method of an environment class and an action as a set method with variable/sensor ID and value as arguments.

XCS rules

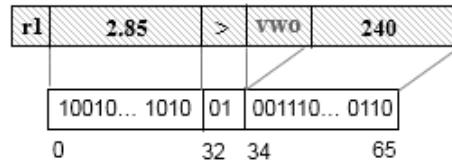
The next step consists of defining a rule of the `XcsRule` type as a pair of two genes: a signal and an action, as shown in the following code:

```
class XcsRule(val signal: Signal, val action: XcsAction)
```

The rule: *r1: IF(yield 10-year TB > 2.84%) THEN reduce VWO shares to 240* is implemented as follows:

```
val signal = new Signal("10ytb", 2.84, GREATER_THAN)
val action = new XcsAction("vwo", 240)
val r1 = new XcsRule(signal, action)
```

The agent encodes the rule as a chromosome using 2 bits to represent the operator and 32 bits for values, as shown in the following diagram:



In this implementation, there is no need to encode the type of action as the agent uses only one type of action—set. A complex action requires encoding of its type.

Knowledge encoding

This example uses very simple rules with a single predicate as the condition. Real-world domain knowledge is usually encoded using complex rules with multiple clauses. It is highly recommended that you break down complex rules into multiple basic rules of classifiers.

Matching a rule to a new sensor consists of matching the sensor to the signal. The algorithm matches the new `new10ytb` sensor (line 2) against the signal in the current population of `s10ytb1` (line 3) and `s10ytb2` (line 4) rules that use the same sensor or the `10ytb` variable as follows:

```
val new10ytb = new XcsSensor("10ytb", 2.76) //2
val s10ytb1 = Signal("10ytb", 2.5, GREATER_THAN) //3
val s10ytb2 = Signal("10ytb", 2.2, LESS_THAN) //4
```

In this case, the agent selects the `r23` rule but not `r34` in the existing population. The agent then adds the `act12` action to the list of possible actions. The agent lists all the rules that match the `r23`, `r11`, and `r46` sensors, as shown in the following code:

```
val r23: XcsRule(s10yTB1, act12) //5
val r11: XcsRule(s10yTB6, act6)
val r46: XcsRule(s10yTB7, act12) //6
```

The action with the most references, `act12`, (lines 5 and 6) is executed. The Q-learning algorithm computes the reward from the profit or loss incurred by the portfolio following the execution of the selected `r23` and `r46` rules. The agent uses the reward to adjust the fitness of `r23` and `r46`, before the genetic selection in the next reproduction cycle. These two rules will reach and stay in the top tier of the rules in the population, until either a new genetic rule modified through crossover and mutation or a rule created through covering, triggers a more rewarding action on the environment.

Covering

The purpose of the covering phase is to generate new rules if no rule matches the input or sensor. The `cover` method of an `XcsCover` singleton generates a new `XcsRule` instance given a sensor and an existing set of actions, as shown here:

```
val MAX_NUM_ACTIONS = 2048

def cover(sensor: XcsSensor, actions: List[XcsAction])
  (implicit quant: Quantization, encoding: Encoding): List[XcsRule] =
  actions./:(List[XcsRule]()) ((xs, act) => {
    val signal = Signal(sensor.id, sensor.value,
      new SOperator(Random.nextInt(Signal.numOperators)))
    new XcsRule(signal, XcsAction(act, Random)) :: xs
  })
}
```

You might wonder why the `cover` method uses a set of actions as arguments knowing that covering consists of creating new actions. The method mutates (^ operator) an existing action to create a new one instead of using a random generator. This is one of the advantages of defining an action as a gene. One of the constructors of `XcsAction` executes the mutation, as follows:

```
def apply(action: XcsAction, r: Random): XcsAction =
  (action ^ r.nextInt(XCSACTION_SIZE))
```

The index of the operator `r` type is a random value in the interval [0, 3] because a signal uses four types of operators: None, >, <, and =.

An implementation example

The `Xcs` class has the following purposes:

- `gaSolver`: This is the selection and generation of genetically modified rules
- `qlLearner`: This is the rewarding and scoring the rules
- `Xcs`: These are the rules for matching, covering, and generation of actions

The extended learning classifier is a data transformation of the `ETransform` type with an explicit configuration of the `XcsConfig` type (line 8) (refer to the *Monadic data transformation* section in *Chapter 2, Hello World!*):

```
class Xcs(config: XcsConfig,
  population: Population[Signal],
  score: Chromosome[Signal] => Unit,
  input: Array[QLInput]) //7
  extends ETransform[XcsConfig](config) { //8

  type U = XcsSensor //9
  type V = List[XcsAction] //10

  val solver = GASolver[Signal](config.gaConfig, score)
  val features = population.chromosomes.toSeq
  val qLearner = QLearning[Chromosome[Signal]]( //11
    config.qlConfig, extractGoals(input), input, features)
  override def |> : PartialFunction[U, Try[V]]
  ...
}
```

The XCS algorithm is initialized with a configuration `config`, an initial set of rules `population`, a fitness function `score`, and an `input` to the Q-learning policy generate reward matrix for `qLearner` (line 7). Being an explicit data transformation, the `U` type of an input element and the `V` type of the output element to the `|>` predictor are initialized as `XcsSensor` (line 9) and `List[XcsAction]` (line 10).

The goals and number of states are extracted from the input to the policy of the Q-learning algorithm.

In this implementation, the `solver` generic algorithm is mutable. It is instantiated along with the `Xcs` container class. The Q-learning algorithm uses the same design, as any classifier, as immutable. The model of Q-learning is the best possible policy to reward rules. Any changes in the number of states or the rewarding scheme require a new instance of the learner.

Benefits and limitations of learning classifier systems

Learning classifier systems and XCS in particular, hold many promises, which are listed as follows:

- They allow nonscientists and domain experts to describe the knowledge using familiar Boolean constructs and inferences such as predicates and rules

- They provide analysts with an overview of the knowledge base and its coverage by distinguishing between the need for exploration and exploitation of the knowledge base

However, the scientific community has been slow to recognize the merits of these techniques. The wider adoption of learning classifier systems is hindered by the following factors:

- The large number of parameters used in both exploration and exploitation phases adds to the sheer complexity of the algorithm.
- There are too many competitive variants of learning classifier systems
- There is no clear unified theory to validate the concept of evolutionary policies or rules. After all, these algorithms are the merger of standalone techniques. The accuracy and performance of the execution of many variants of the learning classifier systems depend on each component as well as the interaction between components.
- An execution that is not always predictable in terms of scalability and performance.

Summary

The software engineering community sometimes overlooks reinforcement learning algorithms. Let's hope that this chapter provides adequate answers to the following questions:

- What is reinforcement learning?
- What are the different types of algorithms that qualify as reinforcement learning?
- How can we implement the Q-learning algorithm in Scala?
- How can we apply Q-learning to the optimization of option trading?
- What are the pros and cons of using reinforcement learning?
- What are learning classifier systems?
- What are the key components of the XCS algorithm?
- What are the potentials and limitations of learning classifier systems?

This concludes the introduction of the last category of learning techniques. The ever-increasing amount of data that surrounds us requires data processing and machine learning algorithms to be highly scalable. This is the subject of the next and the final chapter.

12

Scalable Frameworks

The advent of social networking, interactive media, and deep analysis has caused the amount of data processed daily to skyrocket. For data scientists, it's no longer just a matter of finding the most appropriate and accurate algorithm to mine data; it is also about leveraging multi-core CPU architectures and distributed computing frameworks to solve problems in a timely fashion. After all, how valuable is a data mining application if the model does not scale?

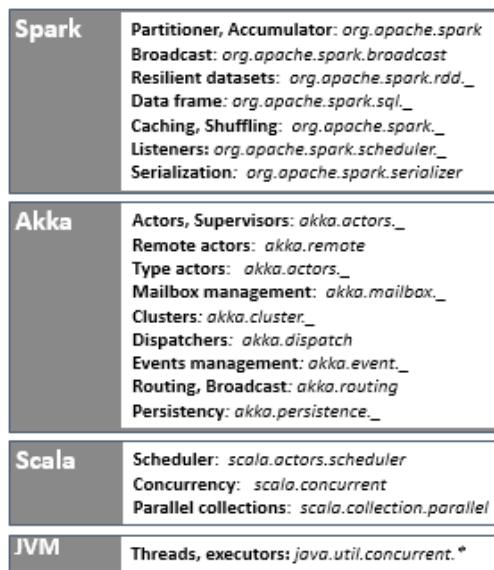
There are many options available to Scala developers to build classification and regression applications for very large datasets. This chapter covers the Scala parallel collections, Actor model, Akka framework, and Apache Spark in-memory clusters. The following topics are covered in this chapter:

- An introduction to Scala parallel collections
- Evaluation of performance of a parallel collection on multi-core CPUs
- The actor model and reactive systems
- Clustered and reliable distributed computing using Akka
- A design of the computational workflow using Akka routers
- An introduction to Apache Spark clustering and its design principles
- Using Spark MLlib for clustering
- Relative performance tuning and evaluation of Spark
- Benefits and limitations of the Apache Spark framework

An overview

The support for distributing and concurrent processing is provided by different stacked frameworks and libraries. Scala concurrent and parallel collections' classes leverage the threading capabilities of the Java virtual machine. **Akka.io** implements a reliable action model originally introduced as part of the Scala standard library. The Akka framework supports remote actors, routing, load balancing protocols, dispatchers, clusters, events, and configurable mailbox management. This framework also provides support for different transport modes, supervisory strategies, and typed actors. Apache Spark's resilient distributed datasets with advanced serialization, caching, and partitioning capabilities leverage Scala and Akka libraries.

The following stack representation illustrates the interdependencies between frameworks:



The Stack representation of scalable frameworks using Scala

Each layer adds a new functionality to the previous one to increase scalability. The Java virtual machine runs as a process within a single host. Scala concurrent classes support effective deployment of an application by leveraging multicore CPU capabilities without the need to write multithreaded applications. Akka extends the Actor paradigm to clusters with advanced messaging and routing options. Finally, Apache Spark leverages Scala higher-order collection methods and the Akka implementation of the Actor model to provide large-scale data processing systems with better performance and reliability, through its resilient distributed datasets and in-memory persistency.

Scala

The Scala standard library offers a rich set of tools, such as parallel collections and concurrent classes to scale number-crunching applications. Although these tools are very effective in processing medium-sized datasets, they are unfortunately quite often discarded by developers in favor of more elaborate frameworks.

Object creation

Although code optimization and memory management is beyond the scope of this chapter, it is worthwhile to remember that a few simple steps can be taken to improve the scalability of an application. One of the most frustrating challenges in using Scala to process large datasets is the creation of a large number of objects and the load on the garbage collector.

A partial list of remedial actions is as follows:

- Limiting unnecessary duplication of objects in an iterated function using a mutable instance
- Using lazy values and **Stream** classes to create objects as needed
- Leveraging efficient collections such as **bloom filters** or **skip lists**
- Running `javap` to decipher the generation of byte code by the JVM

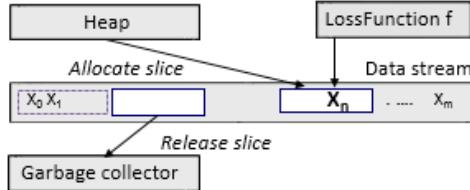
Streams

Some problems require the preprocessing and training of very large datasets, resulting in significant memory consumption by the JVM. Streams are list-like collections in which elements are instantiated or computed lazily. Streams share the same goal of postponing computation and memory allocation as views.

Let's consider the computation of the loss function in machine learning. An observation of the `DataPoint` type is defined as a features vector, `x`, and a labeled or expected value, `y`:

```
case class DataPoint(x: DblVector, y: Double)
```

We can create a loss function, `LossFunction`, that processes a very large dataset on a platform with limited memory. The optimizer responsible for the minimization of the loss or error invokes the loss function at each iteration or recursion, as described in the following diagram:



An illustration of Scala streams allocation and release

The constructor of the `LossFunction` class has the following three arguments (line 2):

- The computation `f` of the loss for each data point
- The weights of the model
- The size of the entire stream `dataSize`

The code is as follows:

```

type StreamLike = WeakReference[Stream[DataPoint]] //1
class LossFunction(
    f: (DblVector, DblVector) => Double,
    weights: DblVector,
    dataSize: Int) { //2

    var nElements = 0
    def compute(stream: () => StreamLike): Double =
        compute(stream().get, 0.0) //3

    def _loss(xs: List[DataPoint]): Double = xs.map(
        dp => dp.y - f(weights, dp.x)).map( sqr(_)).sum //4
}

```

The loss function for the stream is implemented as the `compute` tail recursion (line 3). The recursive method updates the reference of the stream. The type of reference of the stream is `WeakReference` (line 1), so the garbage collection can reclaim the memory associated with the slice for which the loss has been computed. In this example, the loss function is computed as a sum of squared errors (line 4).

The `compute` method manages the allocation and release of slices of stream:

```
@tailrec
def compute(stream: Stream[DataPoint], loss: Double): Double = {
  if( nElements >= dataSize)  loss
  else {
    val step = if(nElements + STEP > dataSize)
      dataSize - nElements else STEP
    nElements += step
    val newLoss = _loss(stream.take(step).toList) //5
    compute( stream.drop(STEP), loss + newLoss ) //6
  }
}
```

The dataset is processed in two steps:

- The driver allocates (that is, `take`) a slice of the stream of observations and then computes the cumulative loss for all the observations in the slice (line 5)
- Once the computation of the loss for the slice is completed, the memory allocated to the weak reference is released (that is, `drop`) (line 6)

An alternative to weak references

There are alternatives to weak references in order for the stream to force the garbage collector to reclaim the memory blocks associated with each slice of observations, which are as follows:



- Define the stream reference as `def`
- Wrap the reference into a method; the reference is then accessible to the garbage collector when the wrapping method returns
- Use a `List` iterator

The average memory allocated during the execution of the loss function for the entire stream is the memory needed to allocate a single slice.

Parallel collections

The Scala standard library includes parallelized collections, whose purpose is to shield developers from the intricacies of concurrent thread execution and race condition. Parallel collections are a very convenient approach to encapsulate concurrency constructs to a higher level of abstraction [12:1].

There are two ways to create parallel collections in Scala, which are as follows:

- Converting an existing collection into a parallel collection of the same semantic using the `par` method; for example, `List[T].par: ParSeq[T]`, `Array[T].par: ParArray[T]`, `Map[K,V].par: ParMap[K,V]`, and so on
- Using the collection classes from the `collection.parallel`, `parallel.immutable`, or `parallel.mutable` packages; for example, `ParArray`, `ParMap`, `ParSeq`, `ParVector`, and so on

Processing a parallel collection

A parallel collection does lend itself to concurrent processing until a pool of threads and a task scheduler are assigned to it. Fortunately, Scala parallel and concurrent packages provide developers with a powerful toolbox to map partitions or segments of collection to tasks running on different CPU cores. The components are as follows:

- `TaskSupport`: This trait inherits the generic `Tasks` trait. It is responsible for scheduling the operation on the parallel collection. There are three concrete implementations of `TaskSupport`.
- `ThreadPoolTaskSupport`: This uses the threads pool in an older version of the JVM.
- `ExecutionContextTaskSupport`: This uses `ExecutorService` that delegates the management of tasks to either a thread pool or the `ForkJoinTasks` pool.
- `ForkJoinTaskSupport`: This uses the fork-join pools of the `java.util.concurrent.ForkJoinPool` type introduced in the Java SDK 1.6. In Java, a **fork-join pool** is an instance of `ExecutorService` that attempts to run not only the current task but also any of its subtasks. It executes the `ForkJoinTask` instances that are lightweight threads.

The following example implements the generation of a random exponential value using a parallel vector and `ForkJoinTaskSupport`:

```
val rand = new ParVector[Float]
Range(0,MAX).foreach(n => rand.updated(n, n*Random.nextFloat()))//1
rand.tasksupport = new ForkJoinTaskSupport(new ForkJoinPool(16))
val randExp = vec.map( Math.exp(_) ) //2
```

The `rand` parallel vector of random probabilities is created and initialized by the main task (line 1), but the conversion to a vector of a `randExp` exponential value is executed by a pool of 16 concurrent tasks (line 2).

Preserving the order of elements

 Operations that traverse a parallel collection using an iterator preserve the original order of the element of the collection. Iterator-less methods such as `foreach` or `map` do not guarantee that the order of the elements that are processed will be preserved.

The benchmark framework

The main purpose of parallel collections is to improve the performance of execution through concurrency. The first step is to either select an existing benchmark or create our own benchmark.

Scala library benchmark

 The Scala standard library has a `testing.Benchmark` trait used to test using the command line [12:2]. All you need to do is insert your function or code in the `run` method:

```
object test with Benchmark { def run { /* ... */ } }
```

Let's create a `ParBenchmark` parameterized class to evaluate the performance of operations on parallel collections:

```
abstract class ParBenchmark[U] (times: Int) {
  def map(f: U => U) (nTasks: Int): Double //1
  def filter(f: U => Boolean) (nTasks: Int): Double //2
  def timing(g: Int => Unit ): Long
}
```

The user has to supply the data transformation `f` for the `map` (line 1) and `filter` (line 2) operations of parallel collections as well as the number of concurrent tasks `nTasks`. The `timing` method collects the duration of the `times` execution of a given operation `g` on a parallel collection:

```
def timing(g: Int => Unit ): Long = {
  var startTime = System.currentTimeMillis
  Range(0, times).foreach(g)
  System.currentTimeMillis - startTime
}
```

Let's define the mapping and reducing operation for the parallel arrays for which the benchmark is defined as follows:

```
class ParArrayBenchmark[U] (u: Array[U], //3
                           v: ParArray[U], //4
                           times:Int) extends ParBenchmark[T] (times)
```

The first argument of the benchmark constructor is the default array of the Scala standard library (line 3). The second argument is the parallel data structure (or class) associated with the array (line 4).

Let's compare the parallelized and default array on the `map` and `reduce` methods of `ParArrayBenchmark` as follows:

```
def map(f: U => U) (nTasks: Int): Unit = {
    val pool = new ForkJoinPool(nTasks)
    v.tasksupport = new ForkJoinTaskSupport(pool)
    val duration = timing(_ => u.map(f)).toDouble //5
    val ratio = timing(_ => v.map(f))/duration //6
    show(s"$numTasks, $ratio")
}
```

The user has to define the mapping function `f` and the number of concurrent tasks `nTasks` available to execute a `map` transformation on the array `u` (line 5) and its parallelized counterpart `v` (line 6). The `reduce` method follows the same design, as shown in the following code:

```
def reduce(f: (U,U) => U) (nTasks: Int): Unit = {
    val pool = new ForkJoinPool(nTasks)
    v.tasksupport = new ForkJoinTaskSupport(pool)
    val duration = timing(_ => u.reduceLeft(f)).toDouble //7
    val ratio = timing(_ => v.reduceLeft(f))/duration //8
    show(s"$numTasks, $ratio")
}
```

The user-defined function `f` is used to execute the `reduce` action on the array `u` (line 7) and its parallelized counterpart `v` (line 8).

The same template can be used for other higher Scala methods, such as `filter`.

The absolute timing of each operation is completely dependent on the environment. It is far more useful to record the ratio of the duration of execution of the operation on the parallelized array, over the single thread array.

The benchmark class `ParMapBenchmark` used to evaluate `ParHashMap` is similar to the benchmark for `ParArray`, as shown in the following code:

```
class ParMapBenchmark[U] (val u: Map[Int, U],
    val v: ParMap[Int, U],
    times: Int) extends ParBenchmark[T] (times)
```

For example, the `filter` method of `ParMapBenchmark` evaluates the performance of the parallel map `v` relative to a single-threaded map `u`. It applies the filtering condition to the values of each map, as follows:

```
def filter(f: U => Boolean) (nTasks: Int): Unit = {
    val pool = new ForkJoinPool(nTasks)
    v.tasksupport = new ForkJoinTaskSupport(pool)
    val duration = timing(_ => u.filter(e => f(e._2))).toDouble
    val ratio = timing(_ => v.filter(e => f(e._2)))/duration
    show(s"$nTasks, $ratio")
}
```

Performance evaluation

The first performance test consists of creating a single-threaded and a parallel array of random values and executing the `map` and `reduce` evaluation methods, on using an increasing number of tasks, as follows:

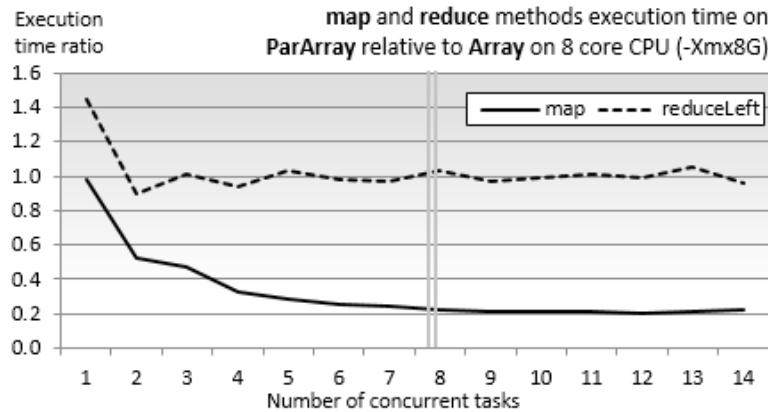
```
val sz = 1000000; val NTASKS = 16
val data = Array.fill(sz) (Random.nextDouble)
val pData = ParArray.fill(sz) (Random.nextDouble)
val times: Int = 50

val bench = new ParArrayBenchmark[Double] (data, pData, times)
val mapper = (x: Double) => Math.sin(x*0.01) + Math.exp(-x)
Range(1, NTASKS).foreach(bench.map(mapper) (_))
val reducer = (x: Double, y: Double) => x+y
Range(1, NTASKS).foreach(bench.reduce(reducer) (_))
```

Measuring performance

The code has to be executed within a loop and the duration has to be averaged over a large number of executions to avoid transient actions such as initialization of the JVM process or collection of unused memory (GC).

The following graph shows the output of the performance test:



The impact of concurrent tasks on the performance on Scala parallelized map and reduce

The test executes the mapper and reducer functions 1 million times on an 8-core CPU with 8 GB of available memory on the JVM.

The results are not surprising in the following respects:

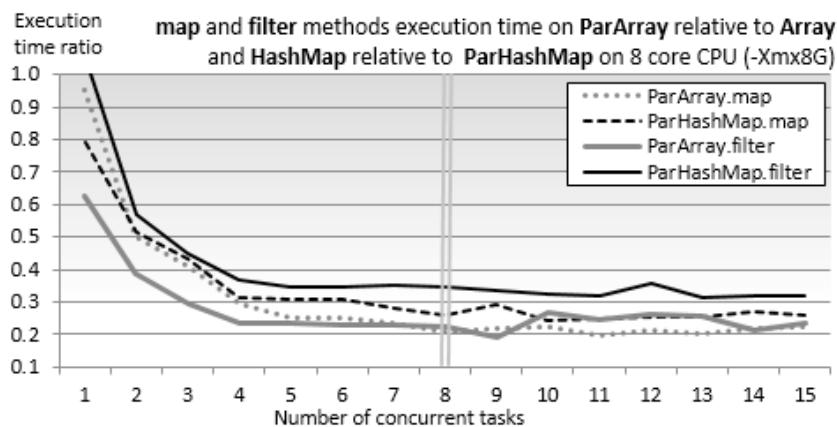
- The reducer doesn't take advantage of the parallelism of the array. The reduction of `ParArray` has a small overhead in the single-task scenario and then matches the performance of `Array`.
- The performance of the `map` function benefits from the parallelization of the array. The performance levels off when the number of tasks allocated equals or exceeds the number of CPU core.

The second test consists of comparing the behavior of the `ParArray` and `ParHashMap` parallel collections, on the `map` and `filter` methods, using a configuration identical to the first test as follows:

```
val sz = 10000000
val mData = new HashMap[Int, Double]
Range(0, sz).foreach( mData.put(_, Random.nextDouble()) ) //9
val mParData = new ParHashMap[Int, Double]
Range(0, sz).foreach( mParData.put(_, Random.nextDouble()) )

val bench = new ParMapBenchmark[Double](mData, mParData, times)
Range(1, NTASKS).foreach(bench.map(mapper)(_)) //10
val filterer = (x: Double) => (x > 0.8)
Range(1, NTASKS).foreach( bench.filter(filterer)(_) ) //11
```

The test initializes a `HashMap` instance and its `ParHashMap` parallel counter with 1 million random values (line 9). The benchmark `bench` processes all the elements of these hash maps with the `mapper` instance introduced in the first test (line 10) and a filtering function `filterer` (line 11) with `NTASKS` equal to 6. The output is shown in the following diagram:



The impact of concurrent tasks on the performance on Scala parallelized array and hash map

The impact of the parallelization of collections is very similar across methods and collections. It's important to notice that the performance of the parallel collections levels off at around four times the single thread collections for five concurrent tasks and above. **Core parking** is partially responsible for this behavior. Core parking disables a few CPU cores in an effort to conserve power, and in the case of a single application, it consumes almost all CPU cycles.

Further performance evaluation

The purpose of the performance test was to highlight the benefits of using Scala parallel collections. You should experiment further with collections other than `ParArray` and `ParHashMap` and other higher-order methods to confirm the pattern.

Clearly, a four times increase in performance is nothing to complain about. Having said that, parallel collections are limited to single-host deployments. If you cannot live with such a restriction and still need a scalable solution, the Actor model provides a blueprint for highly distributed applications.

Scalability with Actors

Traditional multithreaded applications rely on accessing data located in shared memory. The mechanism relies on synchronization monitors such as locks, mutexes, or semaphores to avoid deadlocks and inconsistent mutable states. Even for the most experienced software engineer, debugging multithreaded applications is not a simple endeavor.

The second problem with shared memory threads in Java is the high computation overhead caused by continuous context switches. Context switching consists of saving the current stack frame delimited by the base and stack pointers into the heap memory and loading another stack frame.

These restrictions and complexities can be avoided using a concurrency model that relies on the following key principles:

- Immutable data structures
- Asynchronous communication

The Actor model

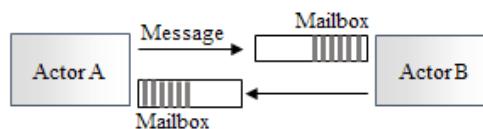
The Actor model, originally introduced in the **Erlang** programming language, addresses these issues [12:3]. The purpose of using the Actor model is twofold as follows:

- It distributes the computation over as many cores and servers as possible
- It reduces or eliminates race conditions and deadlocks, which are very prevalent in the Java development

The model consists of the following components:

- Independent processing units known as Actors. They communicate by exchanging messages asynchronously instead of sharing states.
- Immutable messages are sent to queues, known as mailboxes, before being processed by each actor one at a time.

Let's take a look at the following diagram:



The representation of messaging between actors

There are two message-passing mechanisms, which are as follows:

- **Fire-and-forget or tell:** This sends the immutable message asynchronously to the target or receiving Actor and immediately returns without blocking. The syntax is `targetActorRef ! message`.
- **Send-and-receive or ask:** This sends a message asynchronously, but returns a `Future` instance that defines the expected reply from the `val future = targetActorRef ? message` target actor.

The generic construct for the Actor message handler is somewhat similar to the `Runnable.run()` method in Java, as shown in the following code:

```
while( true ){
    receive { case msg1: MsgType => handler }
}
```

The `receive` keyword is, in fact, a partial function of the `PartialFunction[Any, Unit]` type [12:4]. The purpose is to avoid forcing developers to handle all possible message types. The Actor consuming messages may very well run on a separate component or even application, from the Actor producing these messages. It is not always easy to anticipate the type of messages an Actor has to process in a future version of an application.

A message whose type is not matched is merely ignored. There is no need to throw an exception from within the Actor's routine. Implementations of the Actor model strive to avoid the overhead of context switching and creation of threads [12:5].

I/O blocking operations

 Although it is highly recommended that you do not use Actors to block operations, such as I/O, there are circumstances that require the sender to wait for a response. You need to be keep in mind that blocking the underlying threads might starve other Actors from CPU cycles. It is recommended that you either configure the runtime system to use a large thread pool or allow the thread pool to be resized by setting the `actors.enableForkJoin` property as `false`.

Partitioning

A dataset is defined as a Scala collection, for example, `List`, `Map`, and so on. Concurrent processing requires the following steps:

1. Breaking down a dataset into multiple subdatasets.
2. Processing each dataset independently and concurrently.
3. Aggregating all the resulting datasets.

These steps are defined through a monad associated with a collection in the *Abstraction* section under *Why Scala?* in *Chapter 1, Getting Started*.

1. The `apply` method creates the sub-collection or partitions for the first step, for example, `def apply[T] (a: T): List[T]`.
2. A map-like operation defines the second stage. The last step relies on the monoidal associativity of the Scala collection, for example, `def ++ (a: List[T], b: List[T]): List[T] = a ++ b`.
3. The aggregation, such as `reduce`, `fold`, `sum`, and so on, consists of flattening all the subresults into a single output, for example, `val xs: List(...) = List(List(...), List(...)).flatten`.

The methods that can be parallelized are `map`, `flatMap`, `filter`, `find`, and `filterNot`. The methods that cannot be completely parallelized are `reduce`, `fold`, `sum`, `combine`, `aggregate`, `groupBy`, and `sortWith`.

Beyond actors – reactive programming

The Actor model is an example of the reactive programming paradigm. The concept is that functions and methods are executed in response to events or exceptions. Reactive programming combines concurrency with event-based systems [12:6].

Advanced functional reactive programming constructs rely on composable futures and **continuation-passing style (CPS)**. An example of a Scala reactive library can be found at <https://github.com/ingoem/scala-react>.

Akka

The Akka framework extends the original Actor model in Scala by adding extraction capabilities such as support for typed Actor, message dispatching, routing, load balancing, and partitioning, as well as supervision and configurability [12:7].

The Akka framework can be downloaded from the <http://akka.io/> website or through the Typesafe Activator at <http://www.typesafe.com/platform>.

Akka simplifies the implementation of the Actor model by encapsulating some of the details of Scala Actor in the `akka.actor.Actor` and `akka.actor.ActorSystem` classes.

The three methods you want to override are as follows:

- `prestart`: This is an optional method that is invoked to initialize all the necessary resources such as file or database connection before the Actor is executed
- `receive`: This method defines the Actor's behavior and returns a partial function of the `PartialFunction[Any, Unit]` type
- `postStop`: This is an optional method to clean up resources such as releasing memory, closing database connections, and socket or file handles

Typed and untyped actors

Untyped actors can process messages of any type. If the type of the message is not matched by the receiving actor, it is discarded. Untyped actors can be regarded as contract-less actors. They are the default actors in Scala.



Typed actors are similar to Java remote interfaces. They respond to a method invocation. The invocation is declared publicly, but the execution is delegated asynchronously to the private instance of the target actor [12:8].

Akka offers a variety of functionalities to deploy concurrent applications. Let's create a generic template for a master Actor and worker Actors to transform a dataset using any preprocessing or classification algorithm inherited from an explicit or implicit monadic data transformation, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*

The master Actor manages the worker actors in one of the following ways:

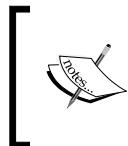
- Individual actors
- Clusters through a **router** or a **dispatcher**

The router is a very simple example of Actor supervision. Supervision strategies in Akka are an essential component to make the application fault-tolerant [12:9]. A supervisor Actor manages the operations, availability, and life cycle of its children, known as **subordinates**. The supervision among actors is organized as a hierarchy. Supervision strategies are categorized as follows:

- **One-for-one strategy:** This is the default strategy. In case of a failure of one of the subordinates, the supervisor executes a recovery, restart, or resume action for that subordinate only.
- **All-for-one strategy:** The supervisor executes a recovery or remedial action on all its subordinates in case one of the Actors fails.

Master-workers

The first model to evaluate is the traditional **master-slaves** or **master-workers** design for the computation workflow. In this design, the worker Actors are initialized and managed by the master Actor, which is responsible for controlling the iterative process, state, and termination condition of the algorithm. The orchestration of the distributed tasks is performed through message passing.



The design principle

It is highly recommended that you segregate the implementation of the computation or domain-specific logic from the actual implementation of the worker and master actors.

Exchange of messages

The first step in implementing the master-worker design is to define the different classes of messages exchanged between the master and each worker in order to control the execution of the iterative procedure. The implementation of the master-worker design is as follows:

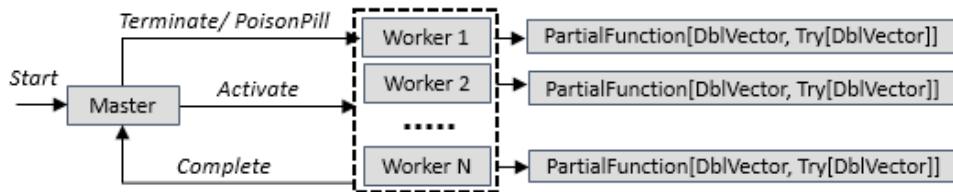
```
sealed abstract class Message(val i: Int)
case class Terminate(i: Int) extends Message(i)
case class Start(i: Int = 0) extends Message(i) //1
case class Activate(i: Int, x: DblVector) extends Message(i) //2
case class Completed(i: Int, x: DblVector) extends Message(i)//3
```

Let's define the messages that control the execution of the algorithm. We need at least the following message types or case classes:

- **Start:** This is sent by the client code to the master to start the computation (line 1).
- **Activate:** This is sent by the master to the workers to activate the computation. This message contains the time series x to be processed by the worker Actors. It also contains the reference to `sender` (master actor). (line 2).
- **Completed:** This is sent by each worker back to `sender`. It contains the variance of the data in the group (line 3).

The master stops a worker using a `PoisonPill` message. The different approaches to terminate an actor are described in the *The master actor* section.

The hierarchy of the `Message` class is sealed to prevent third-party developers from adding another message type. The worker responds to the activate message by executing a data transformation of the `ITransform` type. The messages exchanged between master and worker actors are shown in the following diagram:



A sketch design of the master-slave communication in an actor framework

Messages as case classes

The actor retrieves the messages queued in its mailbox by managing each message instance (copying, matching, and so on). Therefore, the message type has to be defined as a case class. Otherwise, the developer will have to override the `equals` and `hashCode` methods.



Worker actors

The worker actors are responsible for transforming each partitioned datasets created by the master Actor, as follows:

```
type PfnTransform = PartialFunction[DblVector, Try[DblVector]]\n\nclass Worker(id: Int,\n    fct: PfnTransform) extends Actor { //1\n    override def receive = {\n        case msg: Activate => //2\n            sender ! Completed(msg.id+id, fct(msg.xt).get)\n    }\n}
```

The Worker class constructor takes the `fct` (the partial function as an argument) (line 1). The worker launches the processing or transformation of the `msg.xt` data on arrival of the `Activate` message (line 2). It returns the `Completed` message to the master once the `fct` data transformation is completed.

The workflow controller

In the *Scalability* section in *Chapter 1, Getting Started*, we introduced the concepts of workflow and controller to manage the training and classification process as a sequence of transformation on a time series. Let's define an abstract class for all controller actors, `Controller`, with the following three key parameters:

- A time series `xt` to be processed
- A `fct` data transformation implemented as a partial function
- The number of partitions `nPartitions` to break down a time series for concurrent processing

The `Controller` class can be defined as follows:

```
abstract class Controller (\n    val xt: DblVector,\n    val fct: PfnTransform,\n    val nPartitions: Int) extends Actor with Monitor { //3\n\n    def partition: Iterator[DblVector] = { //4\n        val sz = (xt.size.toDouble/nPartitions).ceil.toInt\n        xt.grouped(sz)\n    }\n}
```

The controller is responsible for splitting the time series into several partitions and assigning each partition to a dedicated worker (line 4).

The master actor

Let's define a master actor class `Master`. The three methods to override are as follows:

- `prestart`: This is a method invoked to initialize all the necessary resources such as a file or database connection before the actor executes (line 9)
- `receive`: This is a partial function that dequeues and processes the messages from the mail box
- `postStop`: This cleans up resources such as releasing memory and closing database connections, sockets, or file handles (line 10)

The `Master` class can be defined as follows:

```
abstract class Master( //5
    xt: DblVector,
    fct: PfnTransform,
    nPartitions: Int) extends Controller(xt, fct, nPartitions) {

    val aggregator = new Aggregator(nPartitions) //6
    val workers = List.tabulate(nPartitions)(n =>
        context.actorOf(Props(new Worker(n, fct)),
            name = s"worker_$n")) //7
    workers.foreach( context.watch( _ ) ) //8

    override def preStart: Unit = /* ... */ //9
    override def postStop: Unit = /* ... */ //10
    override def receive
}
```

The `Master` class has the following parameters (line 5):

- `xt`: This is the time series to transform
- `fct`: This is the transformation function
- `nPartitions`: This is the number of partitions

An aggregating class `aggregator` collects and reduces the results from each worker (line 6):

```
class Aggregator(partitions: Int) {
    val state = new ListBuffer[DblVector]

    def += (x: DblVector): Boolean = {
        state.append(x)
        state.size == partitions
    }

    def clear: Unit = state.clear
    def completed: Boolean = state.size == partitions
}
```

The worker actors are created through the `actorOf` factory method of the `ActorSystem` context (line 7). The worker actors are attached to the context of the master actor, so it can be notified when the workers terminate (line 8).

The receive message handler processes only two types of messages: `Start` from the client code and `Completed` from the workers, as shown in the following code:

```
override def receive = {
    case s: Start => start //11

    case msg: Completed => //12
        if( aggregator += msg.xt) //13
            workers.foreach( context.stop(_) ) //14

    case Terminated(sender) => //15
        if( aggregator.completed ) {
            context.stop(self) //16
            context.system.shutdown
        }
}
```

The `Start` message triggers the partitioning of the input time series into partitions (line 11):

```
def start: Unit = workers.zip(partition.toVector)
    .foreach {case (w, s) => w ! Activate(0,s)} //16
```

The partitions are then dispatched to each worker with the `Activate` message (line 16).

Each worker sends a `Completed` message back to master on the completion of their task (line 12). The master aggregates the results from each worker (line 13). Once all the workers have completed their task, they are removed from the master's context (line 14). The master terminates all the workers through a `Terminated` message (line 15), and finally, terminates itself through a request to its `context` to stop it (line 16).

The previous code snippet uses two different approaches to terminate an actor. There are four different methods of shutting down an actor, as mentioned here:

- `actorSystem.shutdown`: This method is used by the client to shut down the parent actor system
- `actor ! PoisonPill`: This method is used by the client to send a poison pill message to the actor
- `context.stop(self)`: This method is used by the Actor to shut itself down within its context
- `context.stop(childActorRef)`: This method is used by the Actor to shut itself down through its reference

Master with routing

The previous design makes sense only if each worker has a unique characteristic that requires direct communication with the master. This is not the case in most applications. The communication and internal management of the worker can be delegated to a router. The implementation of the master routing capabilities is very similar to the previous design, as shown in the following code:

```
class MasterWithRouter(
    xt: DblVector,
    fct: PfnTransform,
    nPartitions: Int) extends Controller(xt, fct, nPartitions) {

    val aggregator = new Aggregator(nPartitions)
    val router = { //17
        val routerConfig = RoundRobinRouter(nPartitions, //18
            supervisorStrategy = this.supervisorStrategy)
        context.actorOf(
            Props(new Worker(0, fct)).withRouter(routerConfig) )
    }
    context.watch(router)

    override def receive
}
```

The only difference is that the `context.actorOf` factory creates an extra actor, `router`, along with the workers (line 17). This particular implementation relies on round-robin assignment of the message by the router to each worker (line 18). Akka supports several routing mechanisms that select a random actor, or the actor with the smallest mailbox, or the first to respond to a broadcast, and so on.

Router supervision



The router actor is a parent of the worker actors. It is by design a supervisor of the worker actors, which are its children actors. Therefore, the router is responsible for the life cycle of the worker actors, which includes their creation, restarting, and termination.

The implementation of the `receive` message handler is almost identical to the message handler in the master without routing capabilities, with the exception of the termination of the workers through the router (line 19):

```
override def receive = {
  case Start => start
  case msg: Completed =>
    if( aggregator += msg.xt) context.stop(router) //19
    ...
}
```

The `start` message handler has to be modified to broadcast the `Activate` message to all the workers through the router:

```
def start: Unit =
  partition.toVector.foreach {router ! Activate(0, _)}
```

Distributed discrete Fourier transform

Let's select the **discrete Fourier transform** (DFT) on a time series `xt` as our data transformation. We discussed this in the *Discrete Fourier transform* section in *Chapter 3, Data Preprocessing*. The testing code is exactly the same, whether the master has routing capabilities or not.

First, let's define a master controller `DFTMaster` dedicated to the execution of the distributed discrete Fourier transform, as follows:

```
type Reducer = List[DblVector] -> immutable.Seq[Double]
class DFTMaster(
  xt: DblVector,
  nPartitions: Int,
  reducer: Reducer) //20
  extends Master(xt, DFT[Double].|>, nPartitions)
```

The reducer method aggregates or reduces the results of the discrete Fourier transform (frequencies distribution) from each worker (line 20). In the case of the discrete Fourier transform, the fReduce reducer method transposes the list of frequencies distribution and then sums up the amplitude for each frequency (line 21):

```
def fReduce(buf: List[DblVector]): immutable.Seq[Double] =
  buf.transpose.map(_.sum).toSeq //21
```

Let's take a look at the test code:

```
val NUM_WORKERS = 4
val NUM_DATAPOINTS = 1000000
val h = (x: Double) => 2.0 * Math.cos(Math.PI * 0.005 * x) +
  Math.cos(Math.PI * 0.05 * x) + 0.5 * Math.cos(Math.PI * 0.2 * x) +
  0.3 * Random.nextDouble //22

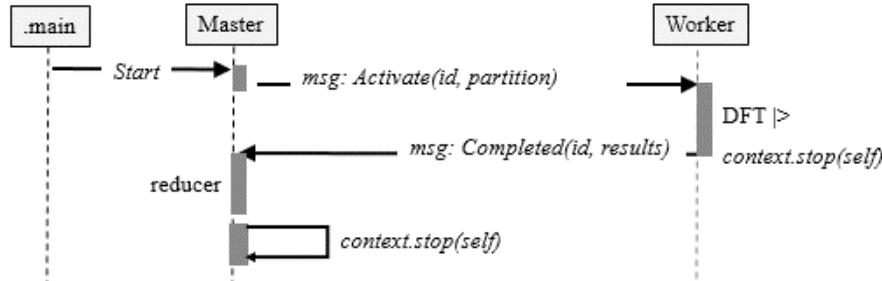
val actorSystem = ActorSystem("System") //23
val xt = Vector.tabulate(NUM_DATA_POINTS)(h(_))
val controller = actorSystem.actorOf(
  Props(new DFTMasterWithRouter(xt, NUM_WORKERS,
    fReduce)), "MasterWithRouter") //24
controller ! Start(1) //25
```

The input time series is synthetically generated by the noisy sinusoidal function h (line 22). The function h has three distinct harmonics: 0.005, 0.05, and 0.2, so the results of the transformation can be easily validated. The Actor system, `ActorSystem`, is instantiated (line 23) and the master Actor is generated through the Akka `ActorSystem.actorOf` factory (line 24). The main program sends a `start` message to the master to trigger the distributed computation of the discrete Fourier transform (line 25).

The action instantiation

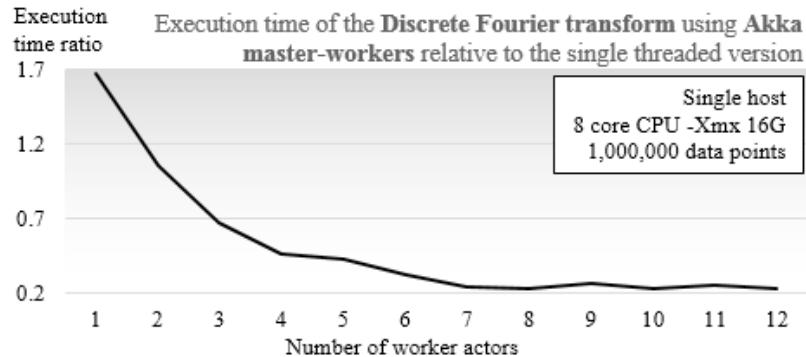
Although the `scala.actor.Actor` class can be instantiated using the constructor, `akka.actor.Actor` is instantiated using an `ActorSystem` context, an `actorOf` factory, and a `Props` configuration object. This second approach has several benefits, including decoupling the deployment of the actor from its functionality and enforcing a default supervisor or parent for the Actor; in this case, `ActorSystem`.

The following sequential diagram illustrates the message exchange between the main program, master, and worker Actors:



A sequential diagram for the normalization of cross-validation groups

The purpose of the test is to evaluate the performance of the computation of the discrete Fourier transform using the Akka framework relative to the original implementation, without actors. As with Scala parallel collections, the absolute timing for the transformation depends on the host and the configuration, as shown in the following graph:



The impact of the number of worker (slave) actors on the performance of the discrete Fourier transform

The single-threaded version of the discrete Fourier transform is significantly faster than the implementation using the Akka master-worker model with a single worker actor. The cost of partitioning and aggregating (or reducing) the results adds a significant overhead to the execution of the Fourier transform. However, the master worker model is far more efficient with three or more worker actors.

Limitations

The master-worker implementation has a few problems, which are as follows:

- In the message handler of the master Actor, there is no guarantee that the poison pill will be consumed by all the workers before the master stops.
- The main program has to sleep for a period of time long enough to allow the master and workers to complete their tasks. There is no guarantee that the computation will be completed when the main program awakes.
- There is no mechanism to handle failure in delivering or processing messages.

The culprit is the exclusive use of the fire-and-forget mechanism to exchange data between master and workers. The send-and-receive protocol and futures are remedies to these problems.

Futures

A future is an object, more specifically a monad, used to retrieve the results of concurrent operations, in a nonblocking fashion. The concept is very similar to a callback supplied to a worker, which invokes it when the task is completed. Futures hold a value that might or might not become available in the future when a task is completed, whether successful or not [12:10].

There are two options to retrieve results from futures:

- Blocking the execution using `scala.concurrent.Await`
- The `onComplete`, `onSuccess`, and `onFailure` callback functions



Which future?

A Scala environment provides developers with two different Future classes: `scala.actor.Future` and `scala.concurrent.Future`.

The `actor.Future` class is used to write continuation-passing style workflows in which the current actor is blocked until the value of the future is available. Instances of the `scala.concurrent.Future` type used in this chapter are the equivalent of `java.concurrent.Future` in Scala.

The Actor life cycle

Let's reimplement the normalization of cross-validation groups by their variance, which we introduced in the previous section, using futures to support concurrency. The first step is to import the appropriate classes for execution of the main actor and futures, as follows:

```
import akka.actor.{Actor, ActorSystem, ActorRef, Props} //26
import akka.util.Timeout    //27
import scala.concurrent.{Await, Future} //28
```

The Actor classes are provided by the `akka.actor` package, instead of the `scala.actor._` package because of Akka's extended actor model (line 26). The future-related classes, `Future` and `Await`, are imported from the `scala.concurrent` package, which is similar to the `java.concurrent` package (line 28). The `akka.util.Timeout` class is used to specify the maximum duration the actor has to wait for the completion of the futures (line 27).

There are two options for a parent actor or the main program to manage the futures it creates, which are as follows:

- **Blocking:** The parent actor or main program stops the execution until all futures have completed their tasks.
- **Callback:** The parent actor or the main program initiates the futures during the execution. The future tasks are performed concurrently with the parent actor, and it is then notified when each future task is completed.

Blocking on futures

The following design consists of blocking the actor that launches the futures until all the futures have been completed, either returning with a result or throwing an exception. Let's modify the master actor into a `TransformFutures` class that manages futures instead of workers or routing actors, as follows:

```
abstract class TransformFutures(
  xt: DblVector,
  fct: PfnTransform,
  nPartitions: Int)
  (implicit timeout: Timeout) //29
  extends Controller(xt, fct, nPartitions) {
  override def receive = {
    case s: Start => compute(transform) //30
  }
}
```

The `TransformFutures` class requires the same parameters as the `Master` actor: a time series, `xt`, a data transformation, `fct`, and the number of partitions, `nPartitions`. The `timeout` parameter is an implicit argument of the `Await.result` method, and therefore, needs to be declared as an argument (line 29). The only message, `Start`, triggers the computation of the data transformation of each future, and then the aggregation of the results (line 30). The `transform` and `compute` methods have the same semantics as those in the master-workers design.

The generic message handler

You may have read or even written examples of actors that have generic `case _ =>` handlers in the message loop for debugging purposes. The message loop takes a partial function as an argument. Therefore, no error or exception is thrown if the message type is not recognized. There is no need for such a handler apart from the one for debugging purposes. Message types should inherit from a sealed abstract class or a sealed trait in order to prevent a new message type from being added by mistake.

Let's take a look at the `transform` method. Its main purpose is to instantiate, launch, and return an array of futures responsible for the transformation of the partitions, as shown in the following code:

```
def transform: Array[Future[DblVector]] = {
    val futures = new Array[Future[DblVector]](nPartitions) //31

    partition.zipWithIndex.foreach { case (x, n) => { //32
        futures(n) = Future[DblVector] { fct(x).get } //33
    }}
    futures
}
```

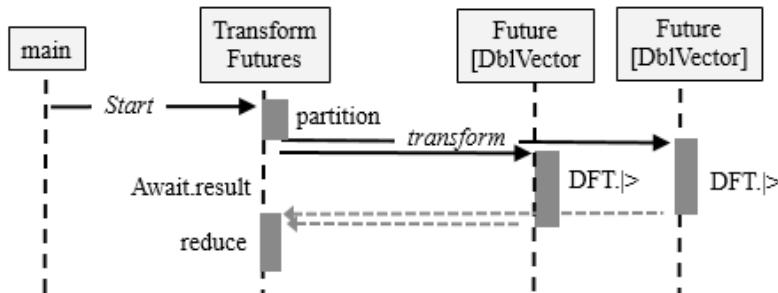
An array of futures (one future per partition) is created (line 31). The `transform` method invokes the partitioning method `partition` (line 32) and then initializes the future with the `fct` partial function (line 33):

```
def compute(futures: Array[Future[DblVector]]): Seq[Double] =
    reduce(futures.map(Await.result(_, timeout.duration))) //34
```

The compute method invokes a user-defined reduce function on the futures. The execution of the Actor is blocked until the `Await` class' `scala.concurrent.Await.result` method (line 34) returns the result of each future computation. In the case of the discrete Fourier transform, the list of frequencies is transposed before the amplitude of each frequency is summed (line 35), as follows:

```
def reduce(data: Array[DblVector]): Seq[Double] =
    data.view.map(_.toArray)
        .transpose.map(_.sum) //35
        .take(SPECTRUM_WIDTH).toSeq
```

The following sequential diagram illustrates the blocking design and the activities performed by the Actor and the futures:



The sequential diagram for actor blocking on future results

Handling future callbacks

Callbacks are an excellent alternative to having the actor blocks on futures, as they can simultaneously execute other functions concurrently with the future execution.

There are two simple ways to implement the callback function, as follows:

- `Future.onComplete`
- `Future.onSuccess` and `Future.onFailure`

The `onComplete` callback function takes a function of the `Try[T] => U` type as an argument with an implicit reference to the execution context, as shown in the following code:

```
val f: Future[T] = future { execute task } f onComplete {
    case Success(s) => { ... }
    case Failure(e) => { ... }
}
```

You can surely recognize the {Try, Success, Failure} monad.

An alternative implementation is to invoke the `onSuccess` and `onFailure` methods that use partial functions as arguments to implement the callbacks, as follows:

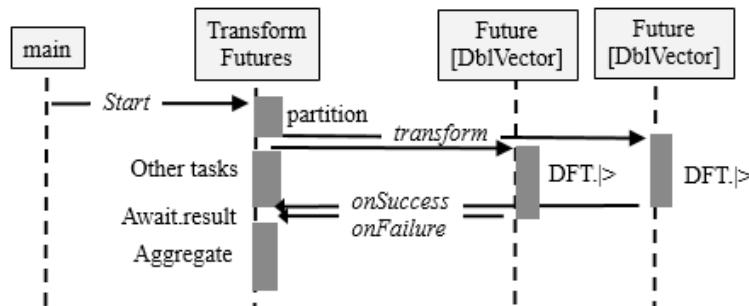
```
f onFailure { case e: Exception => { ... } }
f onSuccess { case t => { ... } }
```

The only difference between blocking one future data transformation and handling callbacks is the implementation of the `compute` method or reducer. The class definition, message handler, and initialization of futures are identical, as shown in the following code:

```
def compute(futures: Array[Future[DblVector]]): Seq[Double] = {
    val buffer = new ArrayBuffer[DblVector]

    futures.foreach( f => {
        f onSuccess { //36
            case data: DblVector => buffer.append(data)
        }
        f onFailure { case e: Exception => /* .. */ } //37
    })
    buffer.find( _.isEmpty).map( _ => reduce(buffer)) //38
}
```

Each future calls the master actor back with either the result of the data transformation, the `onSuccess` message (line 36), or an exception, the `OnFailure` message (line 37). If every future succeeds, the values of all frequencies for all the partitions are summed (line 38). The following sequential diagram illustrates the handling of the callback in the master actor:



A sequential diagram for actor handling future result with callbacks

The execution context

The application of futures requires that the execution context is implicitly provided by the developer. There are three different ways to define the execution context:



- Import the context: `import ExecutionContext.Implicits.global`
- Create an instance of the context within the actor (or actor context): `implicit val ec = ExecutionContext.fromExecutorService(...)`
- Define the context when instantiating the future: `val f = Future[T] = { } (ec)`

Putting it all together

Let's reuse the discrete Fourier transform. The client code uses the same synthetically created time series as in the master-worker test model. The first step is to create a transform future for the discrete Fourier transform, `DFTTransformFuture`, as follows:

```
class DFTTransformFutures(
    xt: DblVector,
    partitions: Int) (implicit timeout: Timeout)
    extends TransformFutures(xt, DFT[Double].|>, partitions) {

    override def reduce(data: Array[DblVector]): Seq[Double] =
        data.map(_.toArray).transpose
            .map(_.sum).take(SPECTRUM_WIDTH).toSeq
}
```

The only purpose of the `DFTTransformFuture` class is to define the `reduce` aggregation method for the discrete Fourier transform. Let's reuse the same test case as in the *Distributed discrete Fourier transform* section under *Master-workers*:

```
import akka.pattern.ask

val duration = Duration(8000, "millis")
implicit val timeout = new Timeout(duration)
val master = actorSystem.actorOf( //39
    Props(new DFTTransformFutures(xt, NUM_WORKERS)),
    "DFTTransform")
val future = master ? Start(0) //40
Await.result(future, timeout.duration) //41
actorSystem.shutdown //42
```

The master actor is initialized as of the `TransformFutures` type with the input time series `xt`, the discrete Fourier transform `DFT`, and the number of workers or partitions `nPartitions` as arguments (line 39). The program creates a future instance by sending (ask) the `Start` message to the master (line 40). The program blocks until the completion of the future (line 41), and then shuts down the Akka actor system (line 42).

Apache Spark

Apache Spark is a fast and general-purpose cluster computing system, initially developed as AMPLab/UC Berkeley as part of the **Berkeley Data Analytics Stack (BDAS)** (http://en.wikipedia.org/wiki/UC_Berkeley). It provides high-level APIs for the following programming languages that make large and concurrent parallel jobs easy to write and deploy [12:11]:

- **Scala:** <http://spark.apache.org/docs/latest/api/scala/index.html>
- **Java:** <http://spark.apache.org/docs/latest/api/java/index.html>
- **Python:** <http://spark.apache.org/docs/latest/api/python/index.html>



The link to the latest information

The URLs as any reference to Apache Spark may change in future versions.



The core element of Spark is a **resilient distributed dataset (RDD)**, which is a collection of elements partitioned across the nodes of a cluster and/or CPU cores of servers. An RDD can be created from a local data structure such as a list, array, or hash table, from the local filesystem or the **Hadoop distributed file system (HDFS)**.

The operations on an RDD in Spark are very similar to the Scala higher-order methods. These operations are performed concurrently over each partition. Operations on RDDs can be classified as follows:

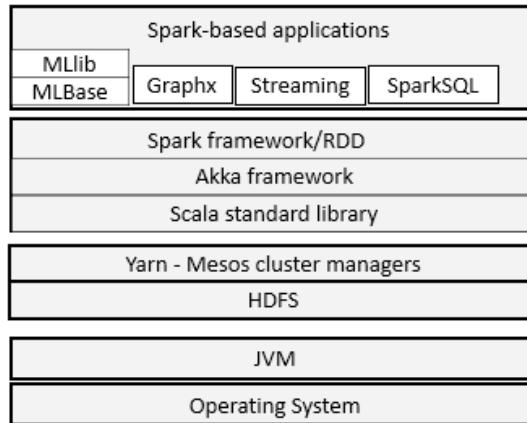
- **Transformation:** This operation converts, manipulates, and filters the elements of an RDD on each partition
- **Action:** This operation aggregates, collects, or reduces the elements of the RDD from all partitions

An RDD can be persisted, serialized, and cached for future computation.

Spark is written in Scala and built on top of Akka libraries. Spark relies on the following mechanisms to distribute and partition RDDs:

- Hadoop/HDFS for the distributed and replicated filesystem
- Mesos or Yarn for the management of a cluster and shared pool of data nodes

The Spark ecosystem can be represented as stacks of technology and framework, as seen in the following diagram:



The Apache Spark framework ecosystem

The Spark ecosystem has grown to support some machine learning algorithms out of the box, such as **MLlib**, a SQL-like interface to manipulate datasets with relational operators, **SparkSQL**, a library for distributed graphs, **GraphX**, and a streaming library [12:12].

Why Spark?

The authors of Spark attempt to address the limitations of Hadoop in terms of performance and real-time processing by implementing in-memory iterative computing, which is critical to most discriminative machine learning algorithms. Numerous benchmark tests have been performed and published to evaluate the performance improvement of Spark relative to Hadoop. In the case of iterative algorithms, the time per iteration can be reduced by a ratio of 1:10 or more.

Spark provides a large array of prebuilt transforms and actions that go well beyond the basic map-reduce paradigm. These methods on RDDs are a natural extension of the Scala collections, making code migration seamless for Scala developers.

Finally, Apache Spark supports fault-tolerant operations by allowing RDDs to persist both in memory and in the filesystem. Persistence enables automatic recovery from node failures. The resiliency of Spark relies on the supervisory strategy of the underlying Akka actors, the persistency of their mailboxes, and the replication schemes of the HDFS.

Design principles

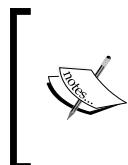
The performance of Spark relies on the following five core design principles [12:13]:

- In-memory persistency
- Laziness in scheduling tasks
- Transform and actions applied to RDDs
- Implementation of shared variables
- Support for data frames (SQL-aware RDDS)

In-memory persistency

The developer can decide to persist and/or cache an RDD for future usage. An RDD may persist in memory only or on disk only—in memory if available, or on disk otherwise as deserialized or serialized Java objects. For instance, an RDD, `rdd`, can be cached through serialization through a simple statement, as shown in the following code:

```
rdd.persist(StorageLevel.MEMORY_ONLY_SER).cache
```



Kryo serialization

Java serialization through the `Serializable` interface is notoriously slow. Fortunately, the Spark framework allows the developer to specify a more efficient serialization mechanism such as the Kryo library.

Laziness

Scala supports lazy values natively. The left-hand side of the assignment, which can either be a value, object reference, or method, is performed once, that is, the first time it is invoked, as shown in the following code:

```
class Pipeline {
    lazy val x = { println("x"); 1.5}
    lazy val m = { println("m"); 3}
    val n = { println("n"); 6}
```

```
def f = (m <<1)
def g(j: Int) = Math.pow(x, j)
}
val pipeline = new Pipeline //1
pipeline.g(pipeline.f) //2
```

The order of the variables printed is `n`, `m`, and then `x`. The instantiation of the `Pipeline` class initializes `n` but not `m` or `x` (line 1). At a later stage, the `g` method is called, which in turn invokes the `f` method. The `f` method initializes the `m` value it needs, and then `g` initializes `x` to compute its power to `m <<1` (line 2).

Spark applies the same principle to RDDs by executing the transformation only when an action is performed. In other words, Spark postpones memory allocation, parallelization, and computation until the driver code gets the result through the execution of an action. The cascading effect of invoking all these transformations backward is performed by the direct acyclic graph scheduler.

Transforms and actions

Spark is implemented in Scala, so you should not be too surprised to know that the most relevant Scala higher methods on collections are supported in Spark. The first table describes the transformation methods using Spark, as well as their counterparts in the Scala standard library. We use the `(K, V)` notation for `(key, value)` pairs:

Spark	Scala	Description
<code>map(f)</code>	<code>map(f)</code>	This transforms an RDD by executing the <code>f</code> function on each element of the collection
<code>filter(f)</code>	<code>filter(f)</code>	This transforms an RDD by selecting the element for which the <code>f</code> function returns <code>true</code>
<code>flatMap(f)</code>	<code>flatMap(f)</code>	This transforms an RDD by mapping each element to a sequence of output items
<code>mapPartitions(f)</code>		This executes the <code>map</code> method separately on each partition
<code>sample</code>		This samples a fraction of the data with or without a replacement using a random generator
<code>groupByKey</code>	<code>groupBy</code>	This is called on <code>(K, V)</code> to generate a new <code>(K, Seq(V))</code> RDD
<code>union</code>	<code>union</code>	This creates a new RDD as an union of this RDD and the argument
<code>distinct</code>	<code>distinct</code>	This eliminates duplicate elements from this RDD
<code>reduceByKey(f)</code>	<code>reduce</code>	This aggregates or reduces the value corresponding to each key using the <code>f</code> function
<code>sortByKey</code>	<code>sortWith</code>	This reorganizes <code>(K, V)</code> in an RDD by ascending, descending, or otherwise specified order of the keys, <code>K</code>
<code>join</code>		This joins an RDD <code>(K, V)</code> with an RDD <code>(K, W)</code> to generate a new RDD <code>(K, (V, W))</code>

Spark	Scala	Description
coGroup		This implements a join operation but generates an RDD ($K, Seq(V), Seq(W)$)

Action methods trigger the collection or the reduction of the datasets from all partitions back to the driver, as listed here:

Spark	Scala	Description
reduce(f)	reduce(f)	This aggregates all the elements of the RDD across all the partitions and returns a Scala object to the driver
collect	collect	This collects and returns all the elements of the RDD across all the partitions as a list in the driver
count	count	This returns the number of elements in the RDD to the driver
first	head	This returns the first element of the RDD to the driver
take(n)	take(n)	This returns the first n elements of the RDD to the driver
takeSample		This returns an array of random elements from the RDD back to the driver
saveAsTextFile		This writes the elements of the RDD as a text file in either the local filesystem or HDFS
countByKey		This generates an (K, Int) RDD with the original keys, K , and the count of values for each key
foreach	foreach	This executes a $T \Rightarrow$ Unit function on each elements of the RDD

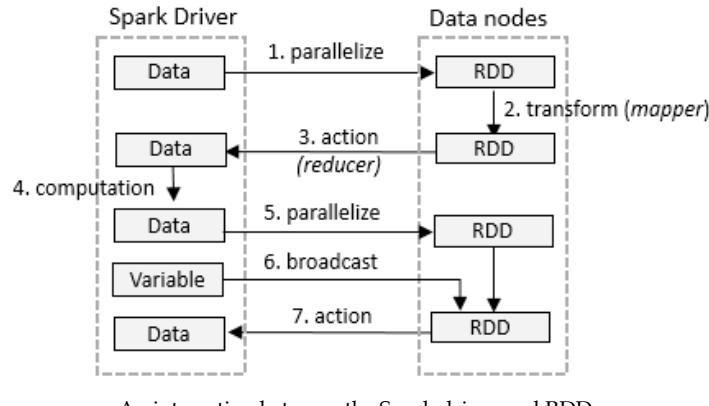
Scala methods such as `fold`, `find`, `drop`, `flatten`, `min`, `max`, and `sum` are not currently implemented in Spark. Other Scala methods such as `zip` have to be used carefully, as there is no guarantee that the order of the two collections in `zip` is maintained between partitions.

Shared variables

In a perfect world, variables are immutable and local to each partition to avoid race conditions. However, there are circumstances where variables have to be shared without breaking the immutability provided by Spark. To this extent, Spark duplicates shared variables and copies them to each partition of the dataset. Spark supports the following types of shared variables:

- **Broadcast values:** These values encapsulate and forward data to all the partitions
- **Accumulator variables:** These variables act as summations or reference counters

The four design principles can be summarized in the following diagram:



An interaction between the Spark driver and RDDs

The preceding diagram illustrates the most common interaction between the Spark driver and its workers, as listed in the following steps:

1. The input data, residing in either the memory as a Scala collection or HDFS as a text file, is parallelized and partitioned into an RDD.
2. A transformation function is applied to each element of the dataset across all the partitions.
3. An action is performed to reduce and collect the data back to the driver.
4. The data is processed locally within the driver.
5. A second parallelization is performed to distribute computation through the RDDs.
6. A variable is broadcast to all the partitions as an external parameter of the last RDD transformation.
7. Finally, the last action aggregates and collects the final result back in the driver.

If you take a look at it closely, the management of datasets and RDDs by the Spark driver is not very different from that by the Akka master and worker actors of futures.

Experimenting with Spark

Spark's in-memory computation for iterative computing makes it an excellent candidate to distribute the training of machine learning models, implemented with dynamic programming or optimization algorithms. Spark runs on Windows, Linux, and Mac OS operating systems. It can be deployed either in local mode for a single host or master mode for a distributed environment. The version of the Spark framework used is 1.3.

JVM and Scala compatible versions

At the time of writing, the version of Spark 1.3.0 required Java 1.7 or higher and Scala 2.10.2 or higher. Spark 1.5.0 supports Scala 2.11 but requires the framework to be reassembled with the flag D-scala2.11.

Deploying Spark

The easiest way to learn Spark is to deploy a localhost in standalone mode. You can either deploy a precompiled version of Spark from the website, or build the JAR files using the **simple build tool (sbt)** or Maven [12:14] as follows:

1. Go to the download page at <http://spark.apache.org/downloads.html>.
2. Choose a package type (Hadoop distribution). The Spark framework relies on the HDFS to run in cluster mode; therefore, you need to select a distribution of Hadoop or an open source distribution such as MapR or Cloudera.
3. Download and decompress the package.
4. If you are interested in the latest functionality added to the framework, check out the newest source code at <http://github.com/apache/spark.git>.
5. Next, you need to build, or assemble, the Apache Spark libraries from the top-level directory using either Maven or sbt:

- **Maven:** Set the following Maven options to support build, deployment, and execution:

```
MAVEN_OPTS="-Xmx4g -XX:MaxPermSize=512M
           -XX:ReservedCodeCacheSize=512m"
mvn [args] -DskipTests clean package
```

The following are some examples:

- Building on Hadoop 2.4 using Yarn clusters manager and Scala 2.10 (default):

```
mvn -Pyarn -Phadoop-2.4 -Dhadoop.version=2.4.0 -  
DskipTests  
    clean package
```

- Building on Hadoop 2.6 using Yarn clusters manager and Scala 2.11:

```
mvn -Pyarn -Phadoop-2.6 -Dhadoop.version=2.6.0 -  
Dscala-2.11  
    -DskipTests clean package
```

- **A simple build tool:** Use the following command:

```
sbt/sbt [args] assembly
```

The following are some examples:

- Building on Hadoop 2.4 using Yarn clusters manager and Scala 2.10 (default):

```
sbt -Pyarn -pHadoop 2.4 assembly
```

- Building on Hadoop 2.6 using Yarn clusters manager and Scala 2.11:

```
sbt -Pyarn -pHadoop 2.6 -Dscala-2.11 assembly
```

Installation instructions

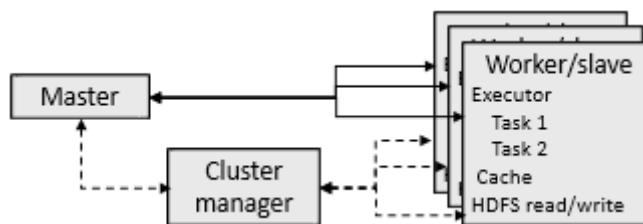
The directory and name of artifacts used in Spark will undoubtedly change over time. You can refer to the documentation and installation guide for the latest version of Spark.

Apache supports multiple deployment modes:

- **Standalone mode:** The drivers and executors run as master and slave Akka actors, bundled with the default spark distribution JAR file.
- **Local mode:** This is a standalone mode running on a single host. The slave actors are deployed across multiple cores within the same host.
- **Yarn clusters manager:** Spark relies on the Yarn resource manager running on Hadoop version 2 and higher. The Spark driver can run either on the same JVM as the client application (client mode) or on the same JVM as the master (cluster mode).

- **Apache Mesos resource manager:** This deployment allows dynamic and scalable partitioning. Apache Mesos is an open source and general-purpose cluster manager that has to be installed separately (refer to <http://mesos.apache.org/>). Mesos manages abstracted the hardware artifacts such as memory or storage.

The communication between a master node (or driver), cluster manager, and set of slave (or worker) nodes is illustrated in the following diagram:



The communication between a master, slave nodes, and a cluster manager

Installation under Windows

 Hadoop relies on some UNIX/Linux utilities that need to be added to the development environment when running on Windows. The `winutils.exe` file has to be installed and added to the `HADOOP_PATH` environment variable.

Using Spark shell

Use any of the following methods to use the Spark shell:

- The shell is an easy way to get your feet wet with Spark-resilient distributed datasets. To launch the shell locally, execute `./bin/spark-shell -master local[8]` to execute the shell on an 8-core localhost.
- To launch a Spark application locally, connect to the shell and execute the following command line:

```

./bin/spark-submit --class application_class --master local[4]
--executor-memory 12G --jars myApplication.jar
--class myApp.class

```

The command launches the application, `myApplication`, with the `myApp.main` method on a 4-core CPU localhost and 12 GB of memory.

- To launch the same Spark application remotely, connect to the shell execute the following command line:

```
./bin/spark-submit --class application_class  
    --master spark://162.198.11.201:7077  
    --total-executor-cores 80  
    --executor-memory 12G  
    --jars myApplication.jar -class myApp.class
```

The output will be as follows:

A partial screenshot of the Spark shell command line output

Potential pitfalls with the Spark shell



Depending on your environment, you might need to disable logging information into the console by reconfiguring `conf/log4j.properties`. The Spark shell might also conflict with the declaration of classpath in the profile or the environment variables' list. In this case, it has to be replaced by `ADD_JARS` as an environment variable such as `ADD_JARS = path1/jar1, path2/jar2`.

MLlib

MLLib is a scalable machine learning library built on top of Spark. As of version 1.0, the library is a work in progress.

The main components of the library are as follows:

- Classification algorithms, including logistic regression, Naïve Bayes, and support vector machines
- Clustering limited to K-means in version 1.0
- L1 and L2 regularization
- Optimization techniques such as gradient descent, logistic gradient and stochastic gradient descent, and L-BFGS
- Linear algebra such as the singular value decomposition
- Data generator for K-means, logistic regression, and support vector machines

The machine learning bytecode is conveniently included in the Spark assembly JAR file built with the simple build tool.

RDD generation

The transformation and actions are performed on RDDs. Therefore, the first step is to create a mechanism to facilitate the generation of RDDs from a time series. Let's create an `RDDSource` singleton with a `convert` method that transforms a time series `xt` into an RDD, as shown here:

```
def convert(
    xt: immutable.Vector[DblArray],
    rddConfig: RDDConfig)
  (implicit sc: SparkContext): RDD[Vector] = {

  val rdd: RDD[Vector] =
    sc.parallelize(xt.toVector.map(new DenseVector(_))) //3
  rdd.persist(rddConfig.persist) //4
  if( rddConfig.cache) rdd.cache //5
  rdd
}
```

The last `rddConfig` argument of the `convert` method specifies the configuration for the RDD. In this example, the configuration of the RDD consists of enabling/disabling cache and selecting the persistency model, as follows:

```
case class RDDConfig(val cache: Boolean,
                     val persist: StorageLevel)
```

It is fair to assume that `SparkContext` has already been implicitly defined in a manner quite similar to `ActorSystem` in the Akka framework.

The generation of the RDD is performed in the following steps:

1. Create an RDD using the `parallelize` method of the context and convert it into a vector (`SparseVector` or `DenseVector`) (line 3).
2. Specify the persistency model or the storage level if the default level needs to be overridden for the RDD (line 3).
3. Specify whether the RDD has to persist in memory (line 5).

 **An alternative for the creation of an RDD**
An RDD can be generated from data loaded from either the local filesystem or HDFS using the `SparkContext.textFile` method that returns an RDD of a string.

Once the RDD is created, it can be used as an input for any algorithm defined as a sequence of transformation and actions. Let's experiment with the implementation of the K-means algorithm in Spark/MLLib.

K-means using Spark

The first step is to create a `SparkKMeansConfig` class to define the configuration of the Apache Spark K-means algorithm, as follows:

```
class SparkKMeansConfig(k: Int, maxIters: Int,  
    numRuns: Int = 1) {  
    val kmeans: KMeans = {  
        (new KMeans).setK(k) //6  
        .setMaxIterations(maxIters) //7  
        .setRuns(numRuns) //8  
    }  
}
```

The minimum set of initialization parameters for MLLib K-means algorithm is as follows:

- The number of clusters, `k` (line 6)
- The maximum number of iterations for the reconstruction of the total errors, `maxIters` (line 7)
- The number of training runs, `numRuns` (line 8)

The `SparkKMeans` class wraps the Spark `KMeans` into a data transformation of the `ITransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* The class follows the design template for a classifier, as explained in the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*:

```
class SparkKMeans(      //9
    kMeansConfig: SparkKMeansConfig,
    rddConfig: RDDConfig,
    xt: Vector[DblArray])
(implicit sc: SparkContext) extends ITransform[DblArray](xt) {

    type V = Int      //10
    val model: Option[KMeansModel] = train //11

    override def |> : PartialFunction[DblArray, Try[V]] //12
    def train: Option[KMeansModel]
}
```

The constructor takes three arguments: the Apache Spark `KMeans` configuration `kMeansConfig`, the RDD configuration `rddConfig`, and the `xt` input time series for clustering (line 9). The return type of the `ITransform` trait's partial function `|>` is defined as an `Int` (line 10).

The generation of `model` merely consists of converting the time series `xt` into an RDD using `rddConfig` and invoking MLlib `KMeans`.`run` (line 11). Once it is created, the model of clusters (`KMeansModel`) is available for predicting a new observation, `x`, (line 12), as follows:

```
override def |> : PartialFunction[DblArray, Try[V]] = {
    case x: DblArray if(x.length > 0 && model != None) =>
        Try[V](model.get.predict(new DenseVector(x)))
}
```

The `|>` prediction method returns the index of the cluster of observations.

Finally, let's write a simple client program to exercise the `SparkKMeans` model using the volatility of the price of a stock and its daily trading volume. The objective is to extract clusters with features (volatility and volume), each cluster representing a specific behavior of the stock:

```
val K = 8
val RUNS = 16
val MAXITERS = 200
val PATH = "resources/data/chap12/CSCO.csv"
val CACHE = true
```

```
val sparkConf = new SparkConf().setMaster("local[8]")
  .setAppName("SparkKMeans")
  .set("spark.executor.memory", "2048m") //13
implicit val sc = new SparkContext(sparkConf) //14

extract.map { case (vty,vol) => { //15
  val vtyVol = zipToSeries(vty, vol)
  val conf = SparkKMeansConfig(K,MAXITERS,RUNS) //16
  val rddConf = RDDConfig(CACHE,
    StorageLevel.MEMORY_ONLY) //17

  val pfnSparkKMeans = SparkKMeans(conf,rddConf,vtyVol) |> //18
  val obs = Array[Double](0.23, 0.67)
  val clusterId = pfnSparkKMeans(obs)
}
```

The first step is to define the minimum configuration for the `sc` context (line 13) and initialize it (line 14). The `vty` and `vol` volatility variables are used as features for K-means and extracted from a CSV file (line 15):

```
def extract: Option[(DblVector, DblVector)] = {
  val extractors = List[Array[String] => Double](
    YahooFinancials.volatility, YahooFinancials.volume
  )
  val pfnSrc = DataSource(PATH, true) |>
  pfnSrc( extractors ) match {
    case Success(x) => Some((x(0).toVector, x(1).toVector))
    case Failure(e) => { error(e.toString); None }
  }
}
```

The execution creates a configuration `conf` for the K-means (line 16) and another configuration for the Spark RDD, `rddConf`, (line 17). The `pfnSparkKMeans` partial function, which implements the K-means algorithm, is created with the K-means, RDD configurations, and the input data `vtyVol` (line 18).

Performance evaluation

Let's execute the normalization of the cross-validation groups on an 8-core CPU machine with 32 GB of RAM. The data is partitioned with a ratio of two partitions per CPU core.

 **A meaningful performance test**

The scalability test should be performed with a large number of data points (normalized volatility, normalized volume), in excess of 1 million, in order to estimate the asymptotic time complexity.

Tuning parameters

The performance of a Spark application depends greatly on the configuration parameters. Selecting the appropriate value for those configuration parameters in Spark can be overwhelming – there are 54 configuration parameters as of the last count. Fortunately, the majority of those parameters have relevant default values. However, there are few parameters that deserve your attention, including the following:

- The number of cores available to execute transformation and actions on RDDs (`config.cores.max`).
- Memory available for the execution of the transformation and actions (`spark.executor.memory`). Setting the value to 60 percent of the maximum JVM heap is a generally a good compromise.
- The number of concurrent tasks to use across all the partitions for shuffle-related operations; they use a key such as `reduceByKey` (`spark.default.parallelism`). The recommended formula is $parallelism = total\ number\ of\ cores \times 2$. The value of the parameter can be overridden with the `spark.reducer.partitions` parameter for specific RDD reducers.
- A flag to compress a serialized RDD partition for `MEMORY_ONLY_SER` (`spark.rdd.compress`). The purpose is to reduce memory footprints at the cost of extra CPU cycles.
- The maximum size of messages containing the results of an action is sent to the `spark.akka.frameSize` driver. This value needs to be increased if a collection may potentially generate a large size array.
- A flag to compress large size broadcasted `spark.broadcast.compress` variables. It is usually recommended.

Tests

The purpose of the test is to evaluate how the execution time is related to the size of the training set. The test executes K-means from the MLlib library on the volatility and trading session volume on the **Bank of America (BAC)** stock over the following periods: 3 months, 6 months, 12 months, 24 months, 48 months, 60 months, 72 months, 96 months, and 120 months.

The following configuration is used to perform the training of K-means: 10 clusters, 30 maximum iterations, and 3 runs. The test is run on a single host with 8-CPU cores and 32 GB of RAM. The test was conducted with the following values of parameters:

- StorageLevel = MEMORY_ONLY
- spark.executor.memory = 12G
- spark.default.parallelism = 48
- spark.akka.frameSize = 20
- spark.broadcast.compress = true
- No serialization

The first step after executing a test for a specific dataset is to log in to the Spark monitoring console at http://host_name:4040/stages:



The average duration of the K-means clustering versus size of training data in months

Obviously, each environment produces somewhat different performance results but confirms that the time complexity of the Spark K-means is a linear function of the training set.

Performance evaluation in a distributed environment

A Spark deployment on multiple hosts will add latency to the overall execution time of the TCP communication. The latency is related to the collection of the results of the clustering back to the Spark driver, which is negligible and independent of the size of the training set.

Performance considerations

This test barely scratches the surface of the capabilities of Apache Spark. The following are the lessons learned from personal experience in order to avoid the most common performance pitfalls when deploying Spark 1.3+:

- Get acquainted with the most common Spark configuration parameters regarding partitioning, storage level, and serialization.
- Avoid serializing complex or nested objects unless you use an effective Java serialization library such as Kryo.
- Look into defining your own partitioning function to reduce large key-value pair datasets. The convenience of `reduceByKey` has its price. The ratio of number of partitions to number of cores has an impact on the performance of a reducer using keys.
- Avoid unnecessary actions such as `collect`, `count`, or `lookup`. An action reduces the data residing in the RDD partitions, and then forwards it to the Spark driver. The Spark driver (or master) program runs on a single JVM with limited resources.
- Rely on shared or broadcast variables whenever necessary. Broadcast variables, for instance, improve the performance of operations on multiple datasets with very different sizes. Let's consider the common case of joining two datasets of very different sizes. Broadcasting the smaller dataset to each partition of the RDD of the larger dataset is far more efficient than converting the smaller dataset into an RDD and executing a join operation between the two datasets.
- Use an accumulator variable for summation as it is faster than using a reduce action on an RDD.

Pros and cons

An increasing number of organizations are adopting Spark as their distributed data processing platform for real-time or pseudo real-time operations. There are several reasons for the fast adoption of Spark:

- It is supported by a large and dedicated community of developers [12:15]
- In-memory persistency is ideal for iterative computation found in machine learning and statistical inference algorithms
- Excellent performance and scalability that can be extended with the Streaming module

- Apache Spark leverages Scala functional capabilities and a large number of open source Java libraries
- Spark can leverage the Mesos or Yarn cluster manager, which reduces the complexity of defining fault-tolerance and load balancing between worker nodes
- Spark needs to be integrated with commercial Hadoop vendors such as Cloudera

However, no platform is perfect and Spark is no exception. The most common complaints or concerns regarding Spark are as follows:

- Creating a Spark application can be intimidating for a developer with no prior knowledge of functional programming.
- The integration with the database has been somewhat lagging, relying heavily on Hive. The Spark development team has started to address these limitations with the introduction of SparkSQL and data frame RDDs.

0xdata Sparkling Water

Sparkling Water is an initiative to integrate **0xdata H2O** with Spark and complement MLlib [12:16]. H2O from 0xdata is a very fast, open source, in-memory platform for machine learning for very large datasets (<http://0xdata.com/product/>). The framework is worth mentioning for the following reasons:

- It has a Scala API
- It is fully dedicated to machine learning and predictive analytics
- It leverages both the frame data representation of H2O and in-memory clustering of Spark

H2O has an extensive implementation of the generalized linear model and gradient boosted classification, among other goodies. Its data representation consists of hierarchical **data frames**. A data frame is a container of vectors potentially shared with other frames. Each vector is composed of **data chunks**, which themselves are containers of **data elements** [12:17]. At the time of writing, Sparkling Water is in beta version.

Summary

This completes the introduction of the most common scalable frameworks built using Scala. It is quite challenging to describe frameworks, such as Akka and Spark, as well as new computing models such as Actors, futures, and RDDs, in a few pages. This chapter should be regarded as an invitation to further explore the capabilities of those frameworks in both a single host and a large deployment environment.

In this last chapter, we learned:

- The benefits of asynchronous concurrency
- The essentials of the actor model and composing futures with blocking or callback modes
- How to implement a simple Akka cluster to squeeze performance of distributed applications
- The ease and blazing performance of Spark's resilient distributed datasets and the in-memory persistency approach

A

Basic Concepts

Machine learning algorithms make significant use of linear algebra and optimization techniques. Describing the concept and the implementation of linear algebra, calculus, and optimization algorithms in detail would have added significant complexity to the book and distracted the reader from the essence of machine learning.

The appendix lists a basic set of elements of linear algebra and optimization mentioned throughout the book. It also summarizes the coding practices and acquaints the reader with basic knowledge of financial analysis.

Scala programming

Here is a partial list of coding practices and design techniques used throughout the book.

List of libraries and tools

The precompiled *Scala for Machine Learning* code is `ScalaML-2.11-0.99.jar` located in the `$ROOT/project/target/scala-2.11` directory. Not all the libraries are needed for every chapter. The list is as follows:

- Java JDK 1.7 or 1.8 is required for all chapters
- Scala 2.10.4 or higher is required for all chapters
- Scala IDE for Eclipse 4.0 or higher
- IntelliJ IDEA Scala plugin 13.0 or higher
- sbt 0.13 or higher

- Apache Commons Math 3.5+ is required for *Chapter 3, Data Preprocessing*, *Chapter 4, Unsupervised Learning*, and *Chapter 6, Regression and Regularization*
- JFChart 1.0.7 is required for *Chapter 1, Getting Started*, *Chapter 2, Hello World!*, *Chapter 5, Naïve Bayes Classifiers*, and *Chapter 9, Artificial Neural Networks*
- Iitb CRF 0.2 (including the LBFGS and Colt libraries) is required for *Chapter 7, Sequential Data Models*
- LIBSVM 0.1.6 is required for *Chapter 8, Kernel Models and Support Vector Machines*
- Akka framework 2.2 or higher is required for *Chapter 12, Scalable Frameworks*
- Apache Spark/MLib 1.3 or higher is required for *Chapter 12, Scalable Frameworks*
- Apache Maven 3.3 or higher (required for Apache Spark 1.4 or higher)

 **A note for Spark developers**
The Scala library and compiler JAR files bundled with the assembly JAR file for Apache Spark contain a version of the Scala standard library and compiler JAR file that may conflict with an existing Scala library (that is, Eclipse default ScalaIDE library).

The `lib` directory contains the following JAR files related to the third-party libraries or frameworks used in the book: colt, CRF, LBFGS and LIBSVM.

Code snippets format

For the sake of readability of the implementation of algorithms, all nonessential code such as error checking, comments, exception, or import have been omitted. The following code elements are discarded in the code snippets presented in the book:

- Comments:

```
/**  
 * This class is defined as ...  
 */  
// The MathRuntime exception has to be caught here!
```

- Validation of class parameters and method arguments:

```
class BaumWelchEM(val lambda: HMMLambda ...) {  
    require( lambda != null, "Lambda model is undefined")
```

- Class qualifiers such as `final` and `private`:

```
final protected class MLP[T <% Double] ...
```

- Method qualifiers and access control (`final`, `private`, and so on):

```
final def inputLayer: MLPLayer
private def recurse: Unit =
```

- Serialization:

```
class Config extends Serializable { ... }
```

- Validation of partial functions:

```
val pfn: PartialFunction[U, V]
pfn.isDefinedAt(u)
```

- Validation of intermediate states:

```
assert( p != None, " ... ")
```

- Java style exceptions:

```
try { ... }
catch { case e: ArrayIndexOutOfBoundsException => ... }
if (y < EPS)
    throw new IllegalStateException( ... )
```

- Scala style exceptions:

```
Try(process(args)) match {
    case Success(results) => ...
    case Failure(e) => ...
}
```

- Nonessential annotations:

```
@inline def mean = { ... }
@implicitNotFound("Conversion $T to Array[Int] undefined")
@throws(classOf[IllegalStateException])
```

- Logging and debugging code:

```
m_logger.debug( ... )
Console.println( ... )
```

- Auxiliary and nonessential methods

Best practices

Encapsulation

One important objective while creating an API is to reduce the access to support a helper class. There are two options to encapsulate helper classes, as follows:

- **A package scope:** The supporting classes are first-level classes with protected access
- **A class or object scope:** The supported classes are nested in the main class

The algorithms presented in this book follow the first encapsulation pattern.

Class constructor template

The constructors of a class are defined in the companion object using `apply` and the class has a package scope (`protected`):

```
protected class A[T] (val x: X, val y: Y,...) { ... }
object A {
  def apply[T] (x: X, y: Y, ...): A[T] = new A(x, y,...)
  def apply[T] (x: , ...): A[T] = new A(x, y0, ...)
}
```

For example, the `SVM` class that implements the support vector machine is defined as follows:

```
final protected class SVM[T <: AnyVal] (
  config: SVMConfig,
  xt: XVSeries[T],
  expected: DblVector)(implicit f: T => Double)
  extends ITransform[Array[T]](xt) {
```

The `SVM` companion object is responsible for defining all the constructors (instance factories) relevant to the `SVM` protected class:

```
def apply[T <: AnyVal] (
  config: SVMConfig,
  xt: XVSeries[T],
  expected: DblVector)(implicit f: T => Double): SVM[T] =
  new SVM[T](config, xt, expected)
```

Companion objects versus case classes

In the preceding example, the constructors are explicitly defined in the companion object. Although the invocation of the constructor is very similar to the instantiation of case classes, there is a major difference; the Scala compiler generates several methods to manipulate an instance as regular data (equals, copy, hash, and so on).

Case classes should be reserved for single state data objects (no methods).

Enumerations versus case classes

It is quite common to read or hear discussions regarding the relative merit of enumerations and pattern matching with case classes in Scala [A:1]. As a very general guideline, enumeration values can be regarded as lightweight case classes or case classes can be considered as heavy weight enumeration values.

Let's take an example of a Scala enumeration that consists of evaluating the uniform distribution of the `scala.util.Random` library:

```
object A extends Enumeration {
    type TA = Value
    val A, B, C = Value
}

import A._
val counters = Array.fill(A.maxId+1)(0)
Range(0, 1000).foreach(_ => Random.nextInt(10) match {
    case 3 => counters(A.id) += 1
    ...
    case _ => {}
})
```

The pattern matching is very similar to the Java's `switch` statement.

Let's consider the following example of pattern matching using case classes that selects a mathematical formula according to the input:

```
package AA {
    sealed abstract class A(val level: Int)
    case class AA extends A(3) { def f = (x:Double) => 23*x}
    ...
}

import AA._
def compute(a: A, x: Double): Double = a match {
    case a: A => a.f(x)
    ...
}
```

The pattern matching is performed using the default equals method, whose byte code is automatically set for each case class. This approach is far more flexible than the simple enumeration at the cost of extra computation cycles.

The advantages of using enumerations over case classes are as follows:

- Enumerations involve less code for a single attribute comparison
- Enumerations are more readable, especially for Java developers.

The advantages of using case classes are as follows:

- Case classes are data objects and support more attributes than enumeration IDs
- Pattern matching is optimized for sealed classes as the Scala compiler is aware of the number of cases

In short, you should use enumeration for single value constants and case classes to match data objects.

Overloading

Contrary to C++, Scala does not actually overload operators. Here is the definition of the very few operators used in code snippets:

- `+=`: This adds an element to a collection or container
- `+`: This sums two elements of the same type

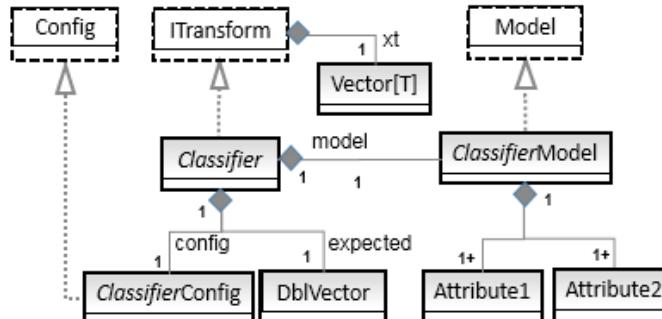
Design template for immutable classifiers

The machine learning algorithms described in this book uses the following design pattern and components:

- The set of configuration and tuning parameters for the classifier is defined in a class inheriting from `Config` (that is, `SVMConfig`).
- The classifier implements a monadic data transformation of the `ITransform` type for which the model is implicitly generated from a training set (that is, `SVM[T]`). The classifier requires at least three parameters, which are as follows:
 - A configuration for the execution of the training and classification tasks
 - An input dataset, `xt`, of the `Vector[T]` type
 - A vector of labels or expected values

- A model of type inherited from `Model`. The constructor is responsible for creating the model through training (that is, `SVMModel`).

Let's take a look at the following diagram:



A generic UML class diagram for classifiers

For example, the key components of the support vector machine package are the classifier SVMs:

```

final protected class SVM[T <: AnyVal] (
  val config: SVMConfig,
  val xt: XTSeries[Array[T]],
  val labels: DblVector)(implicit f: T => Double)
extends ITransform[Array[T]](xt) with Monitor[Double] {

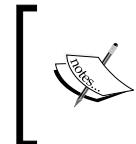
  type V =
  val model: Option[SVMModel] = { ... }
  override def |> PartialFunction[Array[T], V]
  ...
}
  
```

The training set is created by combining or zipping the input dataset `xt` with the labels or expected values `expected`. Once trained and validated, the model is available for prediction or classification.

This design has the main advantage of reducing the life cycle of a classifier: a model is either defined, available for classification, or is not created.

The configuration and model classes are implemented as follows:

```
final class SVMConfig(val formulation: SVMFormulation,  
                      val kernel: SVMKernel,  
                      val svmExec: SVMExecution) extends Config  
  
class SVMModel(val svmmode1: svm_model) extends Model
```



Implementation considerations

The validation phase is omitted in most of the practical examples throughout the book for the sake of readability.



Utility classes

Data extraction

A CSV file is the most common format used to store historical financial data. It is the default format used to import data throughout the book. The data source relies on a `DataSourceConfig` configuration class, as follows:

```
case class DataSourceConfig(pathName: String, normalize: Boolean,  
                           reverseOrder: Boolean, headerLines: Int = 1)
```

The parameters of the `DataSourceConfig` class are as follows:

- `pathName`: This is the relative pathname of a data file to be loaded if the argument is a file or the directory containing multiple input data files. Most of files are CSV files.
- `normalize`: This is the flag that is used to specify whether the data has to be normalized over [0, 1].
- `reverseOrder`: This is the flag that is used to specify whether the order of the data in the file has to be reversed (for example, a time series) if its value is true.
- `headerLines`: This specifies the number of lines for the column headers and comments.

The data source `DataSource` implements data transformation of the `ETransform` type using an explicit configuration `DataSourceConfig`, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*:

```
final class DataSource(config: DataSourceConfig,
    srcFilter: Option[Fields => Boolean] = None)
  extends ETransform[DataSourceConfig](config) {

  type Fields = Array[String]
  type U = List[Fields => Double]
  type V = XVSeries[Double]
  override def |> : PartialFunction[U, Try[V]] =
  ...
}
```

The `srcFilter` argument specifies the filter or condition of some of the row fields to skip the dataset (that is, missing data or incorrect format). Being an explicit data transformation, the constructor for the `DataSource` class has to initialize the `U` input type and the `V` output type of the `|>` extracting method. The method takes the extractor from a row of literal values to double floating point values:

```
override def |> : PartialFunction[U, Try[V]] = {
  case fields: U if(!fields.isEmpty) => load.map(data =>{ //1
    val convert = (f: Fields =>Double) => data._2.map(f(_))
    if( config.normalize) //2
      fields.map(t => new MinMax[Double](convert(t)) //3
        .normalize(0.0, 1.0).toArray ).toVector //4
    else fields.map(convert(_)).toVector
  })
}
```

The data is loaded from the file using the `load` helper method (line 1). The data is normalized if required (line 2) by converting each literal to a floating point value using an instance of the `MinMax` class (line 3). Finally, the `MinMax` instance normalizes the sequence of floating point values (line 4).

The `DataSource` class implements a significant set of methods that are documented in the source code available online.

Data sources

The examples in the book rely on three different sources of financial data using the CSV format:

- `YahooFinancials`: This is for Yahoo schema for the historical stock and ETF price
- `GoogleFinancials`: This is for Google schema for the historical stock and ETF price
- `Fundamentals`: This is for fundamental financial analysis ration (a CSV file)

Let's illustrate the extraction from a data source using `YahooFinancials` as an example:

```
object YahooFinancials extends Enumeration {  
    type YahooFinancials = Value  
    val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME, ADJ_CLOSE = Value  
    val adjClose = ((s:Array[String]) =>  
        s(ADJ_CLOSE.id).toDouble) //5  
    val volume = (s: Fields) => s(VOLUME.id).toDouble  
    ...  
    def toDouble(value: Value): Array[String] => Double =  
        (s: Array[String]) => s(value.id).toDouble  
}
```

Let's take a look at an example of an application of a `DataSource` transformation: loading the historical stock data from the Yahoo finance site. The data is downloaded as a CSV formatted file. Each column is associated with an extractor function (line 5):

```
val symbols = Array[String] ("CSCO", ...) //6  
val prices = symbols  
    .map(s => DataSource(s"$path$s.csv",true,true,1))//7  
    .map(_ |> adjClose) //8
```

The list of stocks for which the historical data has to be downloaded is defined as an array of symbols (line 6). Each symbol is associated with a CSV file (that is, `csc0 => resources/CSCO.csv`) (line 7). Finally, the `YahooFinancials` extractor for the `adjClose` price is invoked (line 8).

The format for the financial data extracted from the Google financial pages are similar to the format used in the Yahoo finances pages:

```
object GoogleFinancials extends Enumeration {  
    type GoogleFinancials = Value  
    val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME = Value  
    val close = ((s:Array[String]) =>s(CLOSE.id).toDouble)//5  
    ...  
}
```

The `YahooFinancials`, `YahooFinancials`, and `Fundamentals` classes implement a significant number of methods that are documented in the source code available online.

Extraction of documents

The `DocumentsSource` class is responsible for extracting the date, title, and content of a list of text documents or text files. The class does not support HTML documents. The `DocumentsSource` class implements a monadic data transformation of the `ETransform` type with an explicit configuration of the `SimpleDateFormat` type:

```
class DocumentsSource(dateFormat: SimpleDateFormat,  
                      val pathName: String)  
extends ETransform[SimpleDateFormat](dateFormat) {  
  
  type U = Option[Long] //2  
  type V = Corpus[Long] //3  
  
  override def |> : PartialFunction[U, Try[V]] = { //4  
    case date: U if (filesList != None) =>  
      Try(if(date == None) getAll else get(date))  
  }  
  def get(t: U): V = getAll.filter(_.date == t.get)  
  def getAll: V //5  
  ...  
}
```

The `DocumentsSource` class takes two arguments: the format of the date associated with the document and the name of the path in which the documents are located (line 1). Being an explicit data transformation, the constructor of the `DocumentsSource` class has to initialize the `U` input type (line 2) as a date and convert it into a `Long` and `V` output type (line 3) as a `Corpus` to extract the `|>` method.

The `|>` extractor generates a corpus associated with a specific date and converts it into a `Long` type (line 4). The `getAll` method does the heavy lifting to extract or sort documents (line 5).

The implementation of the `getAll` method as well as other methods of the `DocumentsSource` class are described in the documented source code available online.

DMatrix class

Some discriminative learning models require operations to be performed on rows and columns of a matrix. The `DMatrix` class facilitates the read and write operations on columns and rows:

```
class DMatrix(val nRows: Int, val nCols: Int,
             val data: DblArray) {
  def apply(i: Int, j: Int): Double = data(i*nCols+j)
  def row(iRow: Int): DblArray = {
    val idx = iRow*nCols
    data.slice(idx, idx + nCols)
  }
  def col(iCol: Int): IndexedSeq[Double] =
    (iCol until data.size by nCols).map( data(_) )
  def diagonal: IndexedSeq[Double] =
    (0 until data.size by nCols+1).map( data(_) )
  def trace: Double = diagonal.sum
  ...
}
```

The `apply` method returns an element of the matrix. The `row` method returns a row array, and the `col` method returns the indexed sequence of column elements. The `diagonal` method returns the indexed sequence of diagonal elements, and the `trace` method sums the diagonal elements.

The `DMatrix` class supports normalization of elements, rows, and columns; transposition; and updation of elements, columns and rows. The `DMatrix` class implements a significant number of methods that are documented in the source code available online.

Counter

The `Counter` class implements a generic mutable counter for which the key is a parameterized type. The number of occurrences of a key is managed by a mutable hash map:

```
class Counter[T] extends mutable.HashMap[T, Int] {
  def += (t: T): type.Counter = super.put(t, getOrElse(t, 0)+1)
  def + (t: T): Counter[T] = {
    super.put(t, getOrElse(t, 0)+1); this
  }
  def ++ (cnt: Counter[T]): type.Counter = {
    cnt./:(this)((c, t) => c + t._1); this
  }
}
```

```
def / (cnt: Counter[T]): mutable.HashMap[T, Double] = map {  
    case(str, n) => (str, if( !cnt.contains(str) )  
        throw new IllegalStateException(" ... ")  
        else n.toDouble/cnt.get(str).get )  
    }  
    ...  
}
```

The `+=` operator updates the counter of the `t` key and returns itself. The `+` operator updates and then duplicates the updated counters. The `++` operator updates this counter with another counter. The `/` operator divides the count for each key by the counts of another counter.

The `Counter` class implements a significant set of methods that are documented in the source code available online.

Monitor

The `Monitor` class has two purposes:

- It stores log information and error messages using the `show` and `error` methods
- It collects and displays variables related to the recursive or iterative execution of an algorithm

The data is collected at each iteration or recursion and then displayed as a time series with iterations as *x* axis values:

```
trait Monitor[T] {  
    protected val logger: Logger  
    lazy val _counters =  
        new mutable.HashMap[String, mutable.ArrayBuffer[T]]()  
  
    def counters(key: String): Option[mutable.ArrayBuffer[T]]  
    def count(key: String, value: T): Unit  
    def display(key: String, legend: Legend)  
        (implicit f: T => Double): Boolean  
    def show(msg: String): Int = DisplayUtils.show(msg, logger)  
    def error(msg: String): Int = DisplayUtils.error(msg, logger)  
    ...  
}
```

The `counters` method returns an array associated with a specific key. The `count` method updates the data associated with a key. The `display` method plots the time series. Finally, the `show` and `error` methods send information and error messages to the standard output.

The documented source code for the implementation of the `Monitor` class is available online.

Mathematics

This section very briefly describes some of the mathematical concepts used in this book.

Linear algebra

Many algorithms used in machine learning such as minimization of a convex loss function, principal component analysis, or least squares regression invariably involves manipulation and transformation of matrices. There are many good books on the subject, from the inexpensive [A:2] to the sophisticated [A:3].

QR decomposition

The QR decomposition (or the QR factorization) is the decomposition of a matrix A into a product of an orthogonal matrix Q and upper triangular matrix R . So, $A=QR$ and $Q^TQ=I$ [A:4].

The decomposition is unique if A is a real, square, and invertible matrix. In the case of a rectangle matrix A , m by n with $m > n$, the decomposition is implemented as the dot product of two vector of matrices: $A = [Q_1, Q_2].[R_1, R_2]^T$, where Q_1 is an m by n matrix, Q_2 is an m by n matrix, R_1 is an n by n upper triangle matrix, and R_2 is an m by n null matrix.

The QR decomposition is a reliable method used to solve a large system of linear equations for which the number of equations (rows) exceeds the number of variables (columns). Its asymptotic computational time complexity for a training set of m dimensions and n observations is $O(mn^2 \cdot n^3 / 3)$.

It is used to minimize the loss function for ordinary least squares regression (refer to the *Ordinary least squares regression* section in *Chapter 6, Regression and Regularization*).

LU factorization

LU factorization is a technique used to solve a matrix equation $A.x = b$, where A is a nonsingular matrix and x and b are two vectors. The technique consists of decomposing the original matrix A as the product of a simple matrix $A = A_1 A_2 \dots A_n$.

- **Basic LU factorization:** This defines A as the product of a lower unit triangular matrix L and an upper triangular matrix U . So, $A = LU$.
- **LU factorization with a pivot:** This defines A as the product of a permutation matrix P , a lower unit triangular matrix L , and an upper triangular matrix U . So, $A = PLU$.

LDL decomposition

The **LDL decomposition** for real matrices defines a real positive matrix A as the product of a lower unit triangular matrix L , a diagonal matrix D , and the transposed matrix of L , that is, L^T . So, $A = LDL^T$.

Cholesky factorization

The **Cholesky factorization** (or the **Cholesky decomposition**) of real matrices is a special case of the LU factorization [A:4]. It decomposes a positive definite matrix A into a product of a lower triangular matrix L and its conjugate transpose L^T . So, $A = LL^T$.

The asymptotic computational time complexity for the Cholesky factorization is $O(mn^2)$, where m is the number of features (model parameters) and n is the number of observations. The Cholesky factorization is used in linear least squares Kalman filter (refer to the *The recursive algorithm* section in *Chapter 3, Data Preprocessing*) and nonlinear Quasi-Newton optimizer.

Singular Value Decomposition

The **singular value decomposition (SVD)** of real matrices defines an m by n real matrix A as the product of an m real square unitary matrix U , an m by n rectangular diagonal matrix Σ , and the transpose matrix V^T of a real matrix. So, $A = U\Sigma V^T$.

The columns of the U and V matrices are the orthogonal bases and the value of the diagonal matrix Σ is a singular value [A:4]. The asymptotic computational time complexity for the singular value decomposition for n observations and m features is $O(mn^2 - n^3)$. The singular value decomposition is used to minimize the total least squares and solve homogeneous linear equations.

Eigenvalue decomposition

The Eigen decomposition of a real square matrix A is the canonical factorization, $Ax = \lambda x$.

λ is the **eigenvalue** (scalar) corresponding to the vector x . The n by n matrix A is then defined as $A = QDQ^T$. Q is the square matrix that contains the eigenvectors and D is the diagonal matrix whose elements are the eigenvalues associated with the eigenvectors [A:5] and [A:6]. The Eigen decomposition is used in Principal Components Analysis (refer to the *Principal components analysis* section in *Chapter 4, Unsupervised Learning*).

Algebraic and numerical libraries

There are many more open source algebraic libraries available to developers as APIs besides Apache Commons Math, which is used in *Chapter 3, Data Preprocessing*, *Chapter 5, Naïve Bayes Classifiers*, *Chapter 6, Regression and Regularization*, and Apache Spark/MLLib in *Chapter 12, Scalable Frameworks*. They are as follows:

- **jBlas 1.2.3** (Java) created by Mikio Braun under the BSD revised license. This library provides Java and Scala developers a high-level Java interface to **BLAS** and **LAPACK** (<https://github.com/mikiobraun/jblas>).
- **Colt 1.2.0** (Java) is a high-performance scientific library developed at CERN under the European Organization for Nuclear Research license (<http://acs.lbl.gov/ACSSoftware/colt/>).
- **AlgeBird 2.10** (Scala) developed at Twitter under Apache Public License 2.0. It defines concepts of abstract linear algebra using monoid and monads. This library is an excellent example of high-level functional programming using Scala (<https://github.com/twitter/algebroid>).
- **Breeze 0.8** (Scala) is a numerical processing library using Apache Public License 2.0 originally created by David Hall. It is a component of the ScalaNLP suite of machine learning and numerical computing libraries (<http://www.scalanlp.org/>).

The Apache Spark/MLLib framework bundles jBlas, Colt, and Breeze. The Iitb framework for conditional random fields uses Colt linear algebra components.

 **An alternative to Java/Scala libraries**

If your application or project needs a high-performance numerical processing tool under limited resources (CPU and RAM memory), then using a C/C++ compiled library is an excellent alternative if portability is not a constraint. The binary functions are accessed through the Java Native Interface (JNI).

First order predicate logic

Propositional logic is the formulation of **axioms** or propositions. There are several formal representations of propositions:

- **Noun-VERB-Adjective:** For example, *Variance of the stock price EXCEEDS 0.76* or *Minimization of the loss function DOES NOT converge*
- **Entity-value = Boolean:** For example, *Variance of the stock price GREATER+THAN 0.76 = true* or *Minimization of the loss function converge = false*
- **Variable op value:** For example, *Variance_stock_price > 0.76* or *Minimization_loss_function != converge*

Propositional logic is subject to the rules of Boolean calculus. Let's consider three propositions: P , Q , and R and the three Boolean operators *NOT*, *AND*, and *OR*:

- $\text{NOT}(\text{NOT } P) = P$
- $P \text{ AND } \text{false} = \text{false}$, $P \text{ AND } \text{true} = P$, $P \text{ or } \text{false} = P$, and $P \text{ or } \text{true} = P$
- $P \text{ AND } Q = Q \text{ AND } P$ and $P \text{ OR } Q = Q \text{ OR } P$
- $P \text{ AND } (Q \text{ AND } R) = (P \text{ AND } Q) \text{ AND } R$

First order predicate logic, also known as **first order predicate calculus**, is the quantification of a propositional logic [A:7]. The most common formulations of the first order logic are as follows:

- Rules (for example, *IF P THEN action*)
- Existential operators

First order logic is used to describe the classifiers in the learning classifier systems (refer to the XCS rules section in *Chapter 11, Reinforcement Learning*).

Jacobian and Hessian matrices

Let's consider a function with n variables x_i and m outputs y_j such that $f: \{x_i\} \rightarrow \{y_j = f_j(x)\}$.

The **Jacobian matrix** [A:8] is the matrix of the first order partial derivatives of the output values of a continuous, differentiable function:

$$J(f) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

The **Hessian matrix** is the square matrix of the second order partial derivatives of a continuously, twice differentiable function:

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \dots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

An example is as follows:

$$f(x, y) = x^2y + e^{-y} \quad J(f) = [2xy, x^2 - e^{-y}] \quad H(f) = \begin{bmatrix} 2y & 2x \\ 2x & e^{-y} \end{bmatrix}$$

Summary of optimization techniques

The same comments regarding linear algebra algorithms apply to optimization. Treating such techniques in depth would have rendered the book impractical. However, optimization is critical to the efficiency and, to a lesser extent, the accuracy of the machine learning algorithms. Some basic knowledge in this field goes a long way to build practical solutions for large datasets.

Gradient descent methods

Steepest descent

The **steepest descent** (or gradient descent) method is one of the simplest techniques used to find a local minimum of any continuous, differentiable function F or the global minimum for any defined, differentiable, and convex function [A:9]. The value of a vector or data point x_{t+1} at iteration $t+1$ is computed from the previous value x_t using the *gradient* ∇F of function F and the slope γ :

$$x_{(t+1)} = x_{(t)} - \gamma \nabla F(a)$$

The steepest gradient algorithm is used to solve systems of nonlinear equations and minimization of the loss function in the logistic regression (refer to the *Numerical optimization* section in *Chapter 6, Regression and Regularization*), in support vector classifiers (refer to the *The nonseparable case – the soft margin* section in *Chapter 8, Kernel Models and Support Vector Machines*), and in multilayer perceptrons (refer to the *Training and classification* section in *Chapter 9, Artificial Neural Networks*).

Conjugate gradient

The **conjugate gradient** solves unconstrained optimization problems and systems of linear equations. It is an alternative to the LU factorization for positive, definite, and symmetric square matrices. The solution x^* of the equation $Ax = b$ is expanded as the weighted summation of n basis orthogonal directions p_i (or **conjugate directions**):

$$Ax = b \rightarrow \sum_{i=0}^{n-1} \alpha_i p_i x^* = b; \quad p_i \cdot p_j = 0$$

The solution x^* is extracted by computing the i^{th} conjugate vector p_i and then computing the coefficients α_i .

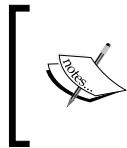
Stochastic gradient descent

The **stochastic gradient** method is a variant of the steepest descent that minimizes the convex function by defining the objective function F as the sum of differentiable, basis function f_i :

$$F(x) = \sum_{i=0}^{n-1} f_i(x), \quad x_{t+1} = x_t - \alpha \sum_{i=0}^{n-1} \nabla f_i(x)$$

The solution x_{t+1} at iteration $t+1$ is computed from the value x_t at iteration t , the step size (or the learning rate) α , and the sum of the gradient of the basis functions [A:10]. The stochastic gradient descent is usually faster than other gradient descents or quasi-Newton methods in converging toward a solution for convex functions. The stochastic gradient descent is used in logistic regression, support vector machines, and backpropagation neural networks.

Stochastic gradient is particularly suitable for discriminative models with large datasets [A:11]. Spark/Mlib makes extensive use of the stochastic gradient method.



The batch gradient descent

The batch gradient descent is introduced and implemented in the *Step 5 – implementing the classifier* section under *Let's kick the tires in Chapter 1, Getting Started*.



Quasi-Newton algorithms

Quasi-Newton algorithms are variations of Newton's method of finding the value of a vector or data point that maximizes or minimizes a function F (first order derivative is null) [A:12].

The Newton's method is a well-known and simple optimization method used to find the solution of equations $F(x) = 0$ for which F is continuous and second order differentiable. It relies on the Taylor series expansion to approximate the function F with a quadratic approximation of variable $\Delta x = x_{t+1} - x_t$ to compute the value at the next iteration using the first order F' and second order F'' derivatives:

$$F(x_t + \Delta x) - F(x_t) \approx F'(x_t) \cdot \Delta x + F''(x_t) \cdot (\Delta x)^2 \rightarrow x_{t+1} = x_t - \frac{F'(x_t)}{F''(x_t)}$$

Contrary to Newton's method, quasi-Newton methods do not require that the second order derivative, Hessian matrix, of the objective function be computed; it just has to be approximated [A:13]. There are several approaches to approximate the computation of the Hessian matrix.

BFGS

The **Broyden-Fletcher-Goldfarb-Shanno (BFGS)** is a quasi-Newton iterative numerical method used to solve unconstrained nonlinear problems. The hessian matrix H_{t+1} at iteration t is approximated using the value of the previous iteration t as $H_{t+1} = H_t + U_t + V_t$ applied to the Newton equation for the direction p_t :

$$H_t p_t = -\nabla F(x_t), \quad x_{t+1} = x_t + \alpha_t p_t$$

The BFGS is used in the minimization of the cost function for the conditional random field and L₁ and L₂ regressions.

L-BFGS

The performance of the BFGS algorithm is related to the caching of the approximation of the Hessian matrix in the memory (U, V) at the cost of high-memory consumption.

The **Limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS)** algorithm is a variant of BFGS that uses a minimum amount of computer RAM. The algorithm maintains the last m incremental updates of the values Δx_t and gradient ΔG_t at iteration t , and then computes these values for the next step $t+1$:

$$x_{t+1} = x_t + \Delta x_t ; \quad \nabla F(x_{t+1}) = \nabla F(x_t) + \Delta G_t \text{ with } \Delta G_t = \Delta(\nabla F(x_t))$$

It is supported by the Apache Commons Math 3.3+, Apache Spark/MLlib 1.0+, Colt 1.0+, and Liitb CRF libraries. L-BFGS is used in the minimization of the loss function in conditional random fields (refer to the *Conditional random fields* section in *Chapter 7, Sequential Data Models*).

Nonlinear least squares minimization

Let's consider the classic minimization of the least squares of a nonlinear function $y = F(x, w)$ with w_i parameters for observations $\{y, x\}$. The objective is to minimize the sum of the squares of residuals r_i , which is as follows:

$$\mathcal{L}(w) = \sum_{i=0}^{m-1} r_i(w)^2 ; \quad r_i = y_i - F(x_i, w)$$

Gauss-Newton

The Gauss-Newton technique is a generalization of Newton's method. The technique solves nonlinear least squares by updating the parameters w_{t+1} at iteration $t+1$ using the first order derivative (or Jacobian):

$$w_{(t+1)} = w_{(t)} - \left\| \frac{\partial r_i(w_{(t)})}{\partial w_i} \right\|_{ij}^{-1} r(w_{(t)})$$

The Gauss-Newton algorithm is used in logistic regression (refer to the *Logistic regression* section in *Chapter 6, Regression and Regularization*).

Levenberg-Marquardt

The Levenberg-Marquardt algorithm is an alternative to the Gauss-Newton technique used to solve nonlinear least squares and curve fitting problems. The method consists of adding the gradient (Jacobian) terms to the residuals r_i to approximate the least squares error:

$$\mathcal{L}(w + \delta) \approx \sum_{i=0}^{m-1} \left(r_i(w) - \frac{\partial F(x_i, w)}{\partial w} \delta \right)^2$$

The Levenberg-Marquardt algorithm is used in the training of logistic regression (refer to the *Logistic regression* section in *Chapter 6, Regression and Regularization*).

Lagrange multipliers

The **Lagrange multipliers** methodology is an optimization technique used to find the local optima of a multivariate function, subject to equality constraints [A:14].

The problem is stated as *maximize $f(x)$ subject to $g(x) = c$, where c is a constant and x is a variable or features vector.*

This methodology introduces a new variable λ to integrate the constraint g into a function, known as the Lagrange function $\mathcal{L}(x, \lambda)$. Let's note $\nabla \mathcal{L}$, which is the gradient of \mathcal{L} over the variables x_i and λ . The Lagrange multipliers are computed by maximizing \mathcal{L} :

$$\begin{aligned}\mathcal{L}(x, \lambda) &= f(x) + \lambda(g(x) - c) \\ \nabla_{x,\lambda} \mathcal{L}(x, \lambda) &= 0 \\ \nabla \mathcal{L} &= \left[\frac{\partial \mathcal{L}}{\partial x_i}, \frac{\partial \mathcal{L}}{\partial \lambda} \right]\end{aligned}$$

An example is as follows:

$$\begin{aligned}f(x, y) &= x^2 + y^2 \text{ subject } x - y = 2 \\ \frac{\partial \mathcal{L}}{\partial x} &= 2x + \lambda, \frac{\partial \mathcal{L}}{\partial y} = 2y - \lambda, \frac{\partial \mathcal{L}}{\partial \lambda} = x - y - 2 \\ x &= 1, y = -1, \lambda = -2\end{aligned}$$

Lagrange multipliers are used to minimize the loss function in the nonseparable case of linear support vector machines (refer to the *The nonseparable case – the soft margin case* section in *Chapter 8, Kernel Models and Support Vector Machines*).

Overview of dynamic programming

The purpose of **dynamic programming** is to break down an optimization problem into a sequence of steps known as **substructures** [A:15]. There are two types of problems for which dynamic programming is suitable.

The solution of a global optimization problem can be broken down into optimal solutions for its subproblems. The solution of the subproblems is known as **optimal substructures**. Greedy algorithms or the computation of the minimum span of a graph are examples of the decomposition into optimal substructures. Such algorithms can be implemented either recursively or iteratively.

The solution of the global problem is applied recursively to the subproblems if the number of subproblems is small. This approach is known as dynamic programming using **overlapping substructures**. Forward-backward passes on hidden Markov models, the Viterbi algorithm (refer to *The Viterbi algorithm* section in *Chapter 7, Sequential Data Models*), or the backpropagation of error in a multilayer perceptron (refer to the *Step 2 – error backpropagation* section in *Chapter 9, Artificial Neural Networks*) are good examples of overlapping substructures.

The mathematical formulation of a dynamic programming solution is specific to the problem it attempts to resolve. Dynamic programming techniques are also commonly used in mathematical puzzles such as *The Tower of Hanoi*.

Finances 101

The exercises presented throughout this book are related to historical financial data and require the reader to have some basic understanding of financial markets and reports.

Fundamental analysis

Fundamental analysis is a set of techniques used to evaluate a security (stock, bond, currency, or commodity) that entails attempting to measure its intrinsic value by examining both macro and micro financial and economy reports. Fundamental analysis is usually applied to estimate the optimal price of a stock using a variety of financial ratios.

Numerous financial metrics are used throughout this book. Here are the definitions of the most commonly used metrics [A:16]:

- **Earnings per share (EPS):** This is the ratio of net earnings to the number of outstanding shares.
- **Price/earnings ratio (PE):** This is the ratio of the market price per share to earnings per share.
- **Price/sales ratio (PS):** This is the ratio of the market price per share to gross sales (or revenue).
- **Price/book value ratio (PB):** This is the ratio of the market price per share to the total balance sheet value per share.
- **Price to earnings/growth (PEG):** This is the ratio of price/earnings per share (PE) to the annual growth of earnings per share.
- **Operating income:** This is the difference between the operating revenue and operating expenses.

- **Net sales:** This is the difference between the revenue or gross sales and cost of goods or cost of sales.
- **Operating profit margin:** This is the ratio of the operating income to the net sales.
- **Net profit margin:** This is the ratio of the net profit to the net sales (or the net revenue).
- **Short interest:** This is the quantity of shares sold short and not yet covered.
- **Short interest ratio:** This is the ratio of the short interest to the total number of shares floated.
- **Cash per share:** This is the ratio of the value of cash per share to the market price per share.
- **Pay-out ratio:** This is the percentage of the primary/basic earnings per share, excluding extraordinary items paid to common stockholders in the form of cash dividends.
- **Annual dividend yield:** This is the ratio of the sum of dividends paid during the previous 12-month rolling period over the current stock price. Regular and extra dividends are included.
- **Dividend coverage ratio:** This is the ratio of the income available to common stockholders, excluding extraordinary items, for the most recent trailing 12 months to gross dividends paid to common shareholders, expressed as percent.
- **Gross Domestic Product (GDP):** This is the aggregate measure of the economic output of a country. It actually measures the sum of values added by the production of goods and delivery of services.
- **Consumer Price Index (CPI):** This is an indicator that measures the change in the price of an arbitrary basket of goods and services used by the Bureau of Labor Statistics to evaluate the inflationary trend.
- **Federal Fund rate:** This is the interest rate at which banks trade balances held at the Federal Reserve. The balances are called Federal Funds.

Technical analysis

Technical analysis is a methodology used to forecast the direction of the price of any given security through the study of the past market information derived from price and volume. In simpler terms, it is the study of price activity and price patterns in order to identify trade opportunities [A:17]. The price of a stock, commodity, bond, or financial future reflects all the information publicly known about that asset as processed by the market participants.

Terminology

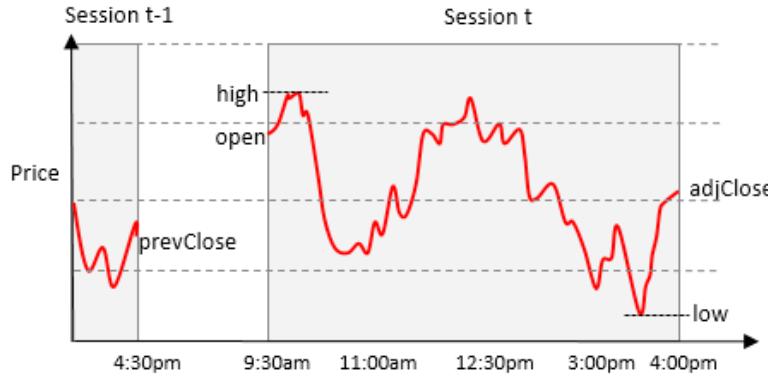
- **Bearish or bearish position:** This attempts to profit by betting that the prices of the security will fall.
- **Bullish or bullish position:** This attempts to profit by betting that the price of the security will rise.
- **Long position:** This is the same as Bullish.
- **Neutral position:** This attempts to profit by betting that the price of the security will not change significantly.
- **Oscillator:** This is a technical indicator that measures the price momentum of a security using some statistical formula.
- **Overbought:** This is a security that is overbought when its price rises too fast as measured by one or several trading signals or indicators.
- **Oversold:** This is a security that is oversold when its price drops too fast as measured by one or several trading signals or indicators.
- **Relative strength index (RSI):** This is an oscillator that computes the average of number of trading sessions for which the closing price is higher than the opening price over the average of number of trading sessions for which the closing price is lower than the opening price. The value is normalized over [0, 1] or [0, 100%].
- **Resistance:** This is the upper limit of the price range of a security. The price falls back as soon as it reaches the resistance level.
- **Short position:** This is the same as Bearish.
- **Support:** This is the lower limit of the price range of a security over a period of time. The price bounces back as soon as it reaches the support level.
- **Technical indicator:** This is a variable derived from the price of a security and possibly its trading volume.
- **Trading range:** The trading range for a security over a period of time is the difference between the highest and lowest price for this period of time.
- **Trading signal:** This is a signal that is triggered when a technical indicator reaches a predefined value, upward or downward.
- **Volatility:** This is the variance or standard deviation of the price of a security over a period of time.

Trading data

The raw trading data extracted from Google or Yahoo financials pages consists of the following:

- **adjClose (or close):** This is the adjusted or nonadjusted price of a security at closing of the trading session
- **open:** This is the price of the security at the opening of the trading session
- **high:** This is the highest price of the security during the trading session
- **low:** This is the lowest price of the security during the trading session

Let's take a look at the following graph:



We can derive the following metrics from the raw trading data:

- Price volatility: $volatility = 1.0 - high/low$
- Price variation: $vPrice = adjClose - open$
- Price difference (or change) between two consecutive sessions: $dPrice = adjClose - prevClose = adjClose(t) - adjClose(t-1)$
- Volume difference between two consecutive sessions: $dVolume = volume(t)/volume(t-1) - 1.0$
- Volatility difference between two consecutive sessions: $dVolatility = volatility(t)/volatility(t-1) - 1.0$
- Relative price variation over the last T trading days: $rPrice = price(t)/average(price over T) - 1.0$
- Relative volume variation over the last T trading days: $rVolume = volume(t)/average(volume over T) - 1.0$
- Relative volatility variation over the last T trading days: $rVolatility = volatility(t)/average(volatility over T) - 1.0$

Trading signals and strategy

The purpose is to create a set variable x , derived from price and volume $x = f(\text{price}, \text{volume})$, and then generate predicates $x \text{ op } c$ for which op is a Boolean operator, such as $>$ or $=$ that compares the value of x to a predetermined threshold c .

Let's consider one of the most common technical indicators derived from price: the relative strength index RSI or the normalized RSI $nRSI$, whose formulation is provided here as a reference:

The relative strength index

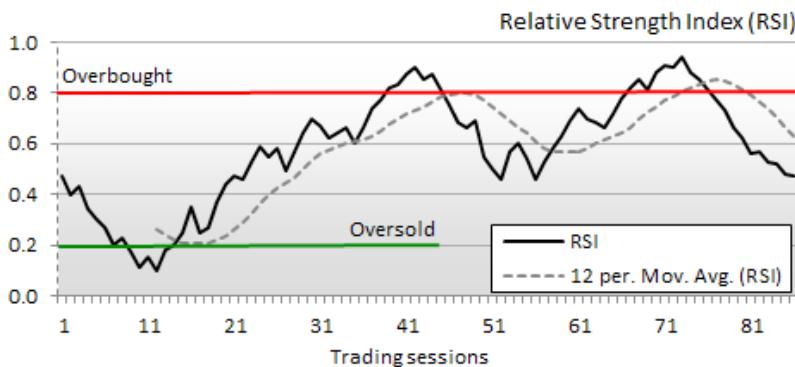
The RSI for a period of T sessions with p_o opening price and p_c closing price is defined as:



$$U_T = \sum_{t=0}^{T-1} (p_c(t) > p_o(t))$$

$$RSI_T = 100 - \frac{100}{1 + \frac{U_T}{T - U_T}} \quad nRSI_T = RSI_T / 100$$

A **trading signal** is a predicate using a technical indicator $nRSI_T(t) < 0.2$. In trading terminology, a signal is emitted for any time period t for which the predicate is true:



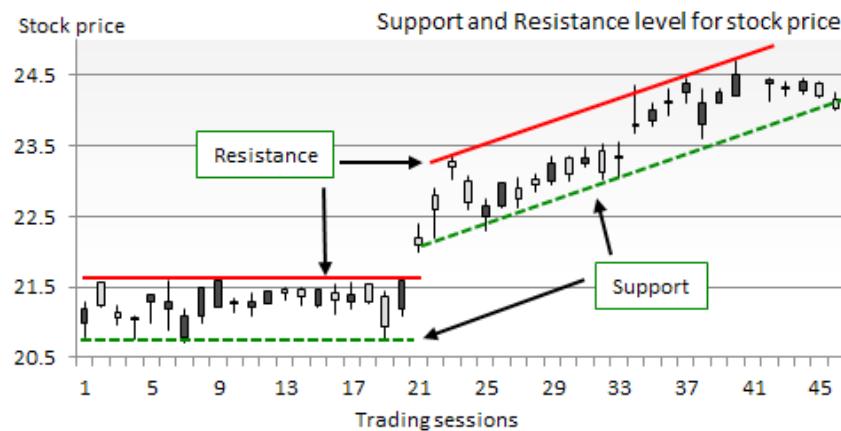
The visualization of oversold and overbought positions using the relative strength index

Traders do not usually rely on a single trading signal to make a rational decision.

For example, if G is the price of gold, I_{10} is the current rate of the 10-year Treasury bond, and RSI_{sp500} is the relative strength index of the S&P 500 index, then we can conclude that the increase in the exchange rate of US\$ to the Japanese Yen is maximized for the following trading strategy: $\{G < \$1170 \text{ and } I_{10} > 3.9\% \text{ and } RSI_{sp500} > 0.6 \text{ and } RSI_{sp500} < 0.8\}$.

Price patterns

Technical analysis assumes that historical prices contains some recurring albeit noisy, patterns that can be discovered using statistical methods. The most common patterns used in this book are the trend, support, and resistance levels [A:18], as illustrated in the following chart:



An illustration of trend, support, and resistance levels in technical analysis

Options trading

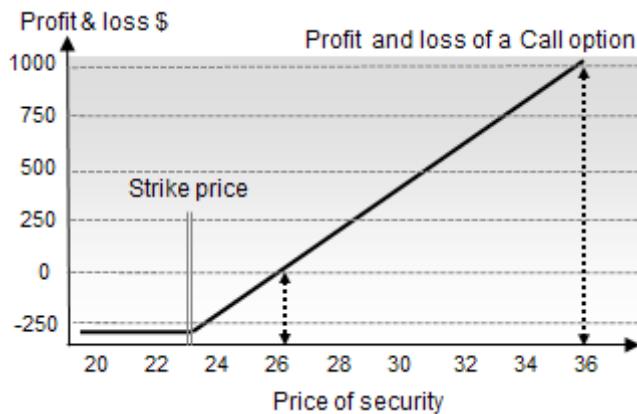
An option is a contract that gives the buyer the right, but not the obligation, to buy or sell a security at a specific price on or before a certain date [A:19].

The two types of options are calls and puts, as described here:

- A call gives the holder the right to buy a security at a certain price within a specific period of time. Buyers of calls expect that the price of the security will increase substantially over the strike price before the option expires.
- A put option gives the holder the right to sell a security at a certain price within a specific period of time. Buyers of puts expect that the price of the stock will fall below the strike price before the option expires.

Let's consider a call option contract of 100 shares at a strike price of \$23 for a total cost of \$270 (\$2.7 per option). The maximum loss the holder of the call can incur is the loss of premium or \$270 when the option expires. However, the profit can be potentially almost unlimited. If the price of the security reaches \$36 when the call option expires, the owner will have a profit of $(\$36 - \$23) * 100 - \$270 = \1030 . The return on the investment is $1030/270 = 380\%$. Buying and then selling the stock would have generated a return on the investment of $36/24 - 1 = 50\%$. This example is simple and does not take into account a transaction fee or margin cost [A:20]:

Let's take a look at the following chart:



An illustration of the pricing of a call option

Financial data sources

There are numerous sources of financial data available to experiment with machine learning and validation models [A:21]:

- Yahoo finances (stocks, ETFs, and indices): <http://finance.yahoo.com>
- Google finances (stocks, ETFs, and indices): <https://www.google.com/finance>
- NASDAQ (stocks, ETFs, and indices): <http://www.nasdaq.com>
- European Central Bank (European bonds and notes): <http://www.ecb.int>
- TrueFx (Forex): <http://www.truefx.com>
- Quandl (Economics and financials statistics): <http://www.quandl.com>
- Dartmouth University (portfolio and simulation): <http://mba.tuck.dartmouth.edu>

Suggested online courses

- *Practical Machine Learning*, J. Leek, R. Peng, B. Caffo, Johns Hopkins University (<https://www.coursera.org/jhu>)
- *Probabilistic Graphical Models*, D. Koller, Stanford University (<https://www.coursera.org/course/pgm>)
- *Machine Learning*, A. Ng, Stanford University (<https://www.coursera.org/course/ml>)

References

- [A:1] *Daily Scala: Enumeration*. J. Eichar. 2009 (<http://daily-scala.blogspot.com/2009/08/enumerations.html>)
- [A:2] *Matrices and Linear Transformations 2nd Edition*. C. Cullen. Dover Books on Mathematics. 1990
- [A:3] *Linear Algebra: A Modern Introduction*. D Poole. BROOKS/COLE CENGAGE Learning. 2010
- [A:4] *Matrix decomposition for regression analysis*. D. Bates. 2007 (<http://www.stat.wisc.edu/courses/st849-bates/lectures/Orthogonal.pdf>)
- [A:5] *Eigenvalues and Eigenvectors of Symmetric Matrices*. I. Mateev. 2013 (<http://www.slideshare.net/vanchizzle/eigenvalues-and-eigenvectors-of-symmetric-matrices>)
- [A:6] *Linear Algebra Done Right 2nd Edition* (§5 Eigenvalues and Eigenvectors) S Axler. Springer. 2000
- [A:7] *First Order Predicate Logic*. S. Kaushik. CSE India Institute of Technology, Delhi (http://www.cse.iitd.ac.in/~saroj/LFP/LFP_2013/L4.pdf)
- [A:8] *Matrix Recipes*. J. Movellan. 2005 (http://www.math.vt.edu/people/dlr/m2k_svbl1_hesian.pdf)
- [A:9] *Gradient descent*. Wikipedia (http://en.wikipedia.org/wiki/Gradient_descent)
- [A:10] *Large Scale Machine Learning: Stochastic Gradient Descent Convergence*. A. Ng. Stanford University (<https://class.coursera.org/ml-003/lecture/107>)
- [A:11] *Large-Scale Machine Learning with Stochastic Gradient Descent*. L Bottou. 2010 (<http://leon.bottou.org/publications/pdf/compstat-2010.pdf>)

- [A:12] *Overview of Quasi-Newton optimization methods.* Dept. Computer Science, University of Washington (<https://homes.cs.washington.edu/~galen/files/quasi-newton-notes.pdf>)
- [A:13] *Lecture 2-3: Gradient and Hessian of Multivariate Function.* M. Zibulevsky. 2013 (<http://www.youtube.com>)
- [A:14] *Introduction to the Lagrange Multiplier.* ediwm.com video (<http://www.noodle.com/learn/details/334954/introduction-to-the-lagrange-multiplier>)
- [A:15] *A brief introduction to Dynamic Programming (DP).* A. Kasibhatla. Nanocad Lab (http://nanocad.ee.ucla.edu/pub/Main/SnippetTutorial/Amar_DP_Intro.pdf)
- [A:16] *Financial ratios.* Wikipedia (http://en.wikipedia.org/wiki/Financial_ratio)
- [A:17] *Getting started in Technical Analysis* (§1 Charts: Forecasting Tool or Folklore?) J Schwager. John Wiley & Sons. 1999
- [A:18] *Getting started in Technical Analysis* (§4 Trading Ranges, Support & Resistance) J Schwager. John Wiley & Sons. 1999
- [A:19] *Options: a personal seminar* (§1 Options: An Introduction, What is an Option) S. Fullman, New York Institute of Finance. Simon Schuster. 1992
- [A:20] *Options: a personal seminar* (§2 Purchasing Options) S. Fullman New York Institute of Finance. Simon Schuster. 1992
- [A:21] *List of financial data feeds.* Wikipedia (http://en.wikipedia.org/wiki/List_of_financial_data_feeds)

Index

Symbols

0xdata H2O 538
0xdata Sparkling Water 538
1-year Treasury bill (1yTB) 172

A

abstraction, Scala
about 4
contravariant functors for co-vectors 6
covariant functors for vectors 5, 6
higher-kind projection 4, 5
monads 7

Actor model
about 502
components 502

actors 7

adaptive modeling 9

Akka framework
about 492, 504, 505
futures 515
master-workers 506
URL 505

Akka.io 492

Algebird 4

algebraic libraries
about 556
AlgeBird 2.10 556

Breeze 0.8 556

Colt 1.2.0 556

jBlas 1.2.3 556

alternative preprocessing techniques
autoregressive models 126
curve-fitting algorithms 126
hidden Markov models 126

nonlinear dynamic systems 126

annual dividend yield 565

Apache Commons Math
about 16
description 16
installation 16, 17
installation, for Mac OS X 17
installation, for Windows 17
licensing 16
URL 15

Apache Spark
about 521
cons 538
design principles 523
deployment modes 528
features 522, 523
performance evaluation 534
pros 537

artificial neural networks
about 343
advantages 393, 394
disadvantages 394
feed-forward neural networks 343

autonomous systems 444

Autoregressive Integrated Moving Average (ARIMA) 126

Autoregressive Moving Average (ARMA) 126

B

batch gradient descent algorithm 34

batch training 358

Baum-Welch estimator 267-269

Bayesian network 170

Berkeley Data Analytics Stack (BDAS)
reference 521

Bernoulli mixture model 192

bias input 346

bias-variance decomposition 77-81

binary SVC
about 313
applications in risk analysis 331-334
classification 323
configuration parameters 315
c-penalty and margin 323-325
design 314, 315
interface to LIBSVM 318, 319
kernel evaluation 326-330
LIBSVM 313, 314
training 319-322

Breeze Scala libraries 4

Broyden-Fletcher-Goldfarb-Shanno (BGFS) 561

C

cake pattern 8, 57

case classes
advantages 546
versus companion objects 545
versus enumerations 545, 546

cash per share 565

categories, NP problems
about 398
NP-complete problems 398
NP-hard problems 398
NP problems 398
P-problems 398

centroid 129

C-Epsilon SVM formulation 310

Cholesky decomposition 555

Cholesky factorization 555

chromosomes 399

class constructor template 544

classification model, evaluation factors
accuracy 69
 F_1 -measure or F_1 -score 69
 F_n score 70
G-measure 70
precision 69
recall 69

classification model, terminology
false negatives (FN) 69
false positives (FP) 69
true negatives (TN) 69
true positives (TP) 69

class prior 174

class prior probability 174

cluster analysis. *See clustering*

cluster assignment, K-means clustering 135

cluster configuration, K-means clustering
about 132
clusters, defining 132
clusters, initializing 134, 135

clustering
about 128, 129
expectation-maximization algorithm 149

clustering algorithms
EM 129
K-means clustering 129

code snippets
format 542, 543

common discriminative kernels 302, 303

companion objects
versus case classes 545

complex adaptive systems 478

components, XCS
about 482
application to portfolio
management 482-485
covering 487
implementation example 487, 488
XCS rules 485, 486

computational workflow
overview 22, 23

conditional dependency 179

conditional independence 46, 169

conditional random field (CRF)
about 279-281
identity potential functions 282
linear chain CRF 281-283
potential functions 282
state feature functions 282
text analytics 283
transition feature functions 282
versus HMM 296, 297

configurability 8

configuration parameters, SVM
 SVM execution 317, 318
 SVM formulation 316
 SVM kernel function 316, 317

confusion matrix 73

conjugate directions 559

conjugate gradient 559

connectionism 344

constructive tuning strategy 374

Consumer Price Index (CPI) 565

continuation-passing style (CPS) 504

control learning 444

convolution neural networks
 about 390
 convolution layers 392
 fully connected hidden layer and output layer 393
 local receptive fields 390, 391
 subsampling layers 392
 weights, sharing 391

core parking 501

Counter class 552, 553

covariant functor 5

co-vector 4

crossover operator, genetic algorithm implementation
 about 417
 chromosomes 420, 421
 genes 421, 422
 population 418, 419

cross-validation, model
 about 75
 K-fold cross validation 77
 one-fold cross validation 75-77

curve fitting 11

D

Darwinian process 398

data chunks 538

data clustering 10

data elements 538

data extraction 548, 549

data frames 538

data partitioning 10

data preprocessing 83

data, profiling
 about 66
 immutable statistics 66, 67
 Z-score 67

data segmentation 10

DataSourceConfig class
 headerLines parameter 548
 normalize parameter 548
 pathName parameter 548
 reverseOrder parameter 548

DBpedia 196

decision boundary 32

decision-making agent 446

decoding, hidden Markov model (HMM)
 about 269
 Viterbi algorithm 270-272

def 59

dependency injection 8

deployment modes, Spark
 Apache Mesos resource manager 529
 local 528
 standalone 528
 Yarn clusters manager 528

descriptive models 9

designing 47

design principles, Spark
 about 523
 actions 524, 525
 in-memory persistency 523
 laziness 523, 524
 shared variables 525, 526
 transforms 524, 525

design template, for classifiers 546, 547

destructive tuning strategy 374

DFT-based filtering 105-107

dimension reduction
 about 11, 157
 non-linear models 165
 principal components analysis 158

directed graphical models 170

discrete Fourier transform (DFT) 98-104, 168

discrete Kalman filter
 about 111, 112
 alternative preprocessing techniques 126
 benefits 125

drawbacks 125
optimal estimator 112
recursive algorithm 112-116
state space estimation 113
discretization 401
dividend coverage ratio 565
DMatrix class 552
DNA 399
Domain Specific Languages (DSL) 9
dynamic programming 563, 564

E

earnings per share (EPS) 564
Eigenvalue decomposition 556
encapsulation
about 544
class or object scope 544
package scope 544
encoding scheme, genetic encoding
about 404
flat encoding 404
hierarchical encoding 404, 405
enumerations
advantages 546
versus case classes 545, 546
epoch 359
Erlang programming language 502
error backpropagation, training epoch
about 366
computational model 371, 372
error propagation 368-371
weights' adjustment 367, 368
error handling, monadic data transformation
about 50
input value 50
output value 50
error insensitive zone 337
evaluation
about 378
execution profile 378
impact of learning rate 379-381
impact of momentum factor 381, 382
impact of number of hidden layers 382, 383
test case 383

evaluation, hidden Markov model (HMM)
about 260
alpha algorithm 260-263
beta algorithm 264, 265
evidence 174
evolution
about 397
evolutionary computing 399
NP problems 398, 399
origin 398
exchange-traded funds (ETFs) 383
ExecutionContextTaskSupport 496
expectation-maximization algorithm
about 149
classification 154
Gaussian mixture models 149
implementation 151-153
online EM algorithm 157
overview 150, 151
testing 154-157
expectation-maximization (EM) 266
experimenting, with Spark
about 527
K-means, using Spark 532-534
MLlib 530, 531
RDD generation 531, 532
Spark, deploying 527-529
Spark shell, using 529, 530
exponential moving average 93-97
exponential normalization 366
Extended Kalman Filters (EKF) 112
extended learning classifier systems
about 480-482
components 482
exploitation phase 480
exploration phase 480

F

Fast Fourier Transform (FFT) 99
features extraction 48
features maps 391
features selection 47
Federal Fund rate 565
feed-forward neural networks
about 343
biological background 344, 345

mathematical background 345, 346
FFNN without a hidden layer 348
finances
 about 101, 564
 financial data sources 570
 fundamental analysis 564
 options trading 569, 570
 technical analysis 565
first order predicate logic 557
fitness functions, genetic algorithms
 about 410
 approximate fitness function 410
 evolutionary fitness function 410
 fixed fitness function 410
fixed lag smoothing 121
fork-join pool 496
ForkJoinTaskSupport 496
Fourier analysis
 about 97
 DFT-based filtering 105-107
 discrete Fourier transform (DFT) 98-104
 market cycles, detecting 108-111
Fourier transform 98
frameworks 15
frequency domain 98
F-score for binomial classification 70-72
F-score for multinomial classification
 about 72-75
 macro method 73
 micro method 73
fully connected neural network 349
function approximation 11, 469
functors 4
fundamental analysis 564
futures, Akka framework
 about 515
 Actor life cycle 516
 blocking on 516, 517
 future callbacks, handling 518, 519

G

Gauss-Newton technique 562
generalized autoregressive conditional heteroskedasticity (GARCH) 126
generic L_p-norm 226
genes 399
genetic algorithms
 about 400
 advantages 440, 441
 components 400
 disadvantages 441
 discrete model parameters 400
 ensemble learning 400
 fitness score 410
 implementation 410
 neural network architecture 400
 reinforcement learning 400
 tests 437
genetic algorithms, for trading strategies
 about 427, 428
 test case 432
 trading strategies, defining 428
genetic encoding
 about 400, 401
 encoding scheme 404
 predicate encoding 402
 solution encoding 403
 value encoding 401, 402
genetic fitness functions 400
genetic operators
 about 400, 405
 crossover 405, 408, 409
 mutation 405, 409, 410
 selection 405-407
 transposition operator 406
GNU Lesser General Public License (LGPL) 17
GoogleFinancials 550
gradient descent 212
gradient descent methods
 about 559
 conjugate gradient 559
 steepest descent 559
 stochastic gradient descent 560
graphical models 169
graph-structured CRF 280
Growth Domestic Product (GDP) 565

H

Hadoop Distributed File System (HDFS) 26
hard margin 308
Hessian matrix 558
hidden layers 347
hidden Markov model (HMM)
 about 251-255
 as filtering technique 278
 canonical forms 255
 canonical forms, implementing 272-275
 components 254
 decoding 269
 design 258, 259
 evaluation 260
 evaluation, test case 2 277
 lambda model 256-258
 notations 255, 256
 performance consideration 297
 training 266
 training, test case 1 275, 276
Hidden Naïve Bayes (HNB) 179
hinge loss 310
HMM constructor
 config 274
 f 274
 form 274
 quantize 274
 xt 274
hyperplane 237

I

implementation, genetic algorithms
 about 410
 crossover operator 417
 GA configuration 417
 key components 412
 mutation operator 422
 population growth, controlling 417
 reproduction 423, 424
 selection operator 416
 software design 411
 solver 424-427
implementation, Q-learning
 about 452
 prediction 464, 465

Q-learning components 458-460
Q-learning training 460, 461
search space 454-458
software design 452
states and actions 453
tail recursion to rescue 462, 463
validation 463, 464

information retrieval and text mining 195

input forward propagation, training epoch
 about 361, 362
 computational flow 362, 363
 error functions 363, 364
 operating nodes 364, 365
 softmax 366

insensitive error 337

J

Jacobian matrix 558

Java 15

JBlas/Linpack

 URL 15
JFreeChart
 about 17
 description 17
 installation 17
 installation, for Mac OSX 17
 installation, for Windows 18
 licensing 17

K

Kalman smoothing 119-121

K-fold cross validation 77

kernel functions

 about 300-302
 common discriminative kernels 302, 303
 kernel monadic composition 304-306
 Laplacian kernel 302
 linear kernel (dot product) 302
 log kernel 302
 polynomial kernel 302
 radial basis function (RBF) 302
 sigmoid kernel 302

kernel trick 312

key components, genetic algorithm

implementation

 chromosomes 413

genes 413-415
 population 412
keyquality metrics 69
K-means clustering
 about 129
 algorithm, defining 131, 132
 classification 140
 cluster assignment 135
 cluster configuration 132
 curse of dimensionality 140, 141
 evaluation, setting up 142, 143
 number of clusters, tuning 145-147
 reconstruction/error minimization 137
 results, evaluating 144
 similarity, measuring 129
 validation 148

L

L1 regularization 226
L2 regularization 226
Lagrange multipliers 563
Laplace 177
lasso regularization 226
Latent Dirichlet allocation (LDA) 171
lazy methods 9
LDL decomposition 555
learning classifier systems (LCS)
 about 477, 478
 benefits 488, 489
 components 478, 479
 features 479
 limitations 489
 terminology 479
learning vector quantization 128
least squares problem 234
lemmatization 195
Levenberg-Marquardt 562
Levenshtein distance 195
libraries 18
libraries directory 542
LIBSVM
 about 313
 benefits 313
 URL, for documentation 313
 URL, for downloading 313

LIBSVM, Java classes
 svm 314
 svm_model 314
 svm_node 314
 svm_parameters 314
 svm_problem 314
Lidstone 177
likelihood 174
Limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) 561
linear algebra
 about 554
 algebraic libraries 556
 Cholesky factorization 555
 Eigenvalue decomposition 556
 LDL decomposition 555
 LU factorization 555
 numerical libraries 556
 QR decomposition 554
 singular value decomposition (SVD) 555
linear chain CRF 281
linear chain structured graph CRF 281
linear least-squares regression 207
linear regression
 about 207
 one-variate linear regression 208
 ordinary least squares regression 211, 212
 versus SVR 339-341

linear SVM
 about 307
 nonseparable case (soft margin) 308-310
 separable case (hard margin) 307, 308

LogBinRegression constructor
 eps 33
 eta 33
 expected 33
 maxIters 33
 obsSet 33

logistic regression
 about 236
 binomial classification 237-239
 classification 247-249
 design 240
 logistic function 236
 training workflow 240

low-band filter 97

LU factorization
about 555
basic LU factorization 555
with pivot 555

M

machine learning
features 2

machine learning algorithms
taxonomy 10

machine learning problems
classification 3
optimization 3
prediction 3
regression 3

maintainability 9

Markov decision processes
about 251
first order discrete Markov chain 252, 253
Markov property 251, 252

master-workers, Akka
about 506
discrete Fourier transform (DFT) 512-514
exchange of messages 506, 507
limitations 515
master actor 509, 510
master with routing 511, 512
worker actors 508
workflow controller 508

mathematical abstractions
about 56
instantiation 57
model definition 56
variable declaration 56

mathematical concepts
about 554
dynamic programming 563, 564
first order predicate logic 557
Hessian matrix 558
Jacobian matrix 558
linear algebra 554
optimization techniques 558

mathematical notation 2

maximum margin classifiers
kernel trick 311

mean squared error (MSE) 208

measurement noise covariance 114

message-passing mechanisms
fire-and-forget or tell 503
send-and-receive or ask 503

metaphor for graphical models 170

methodology
defining 48, 49

Michigan approach 479

mixins 58

mixins, composing for building workflow
about 58
modules, defining 60
problem, understanding 58-60
workflow, instantiating 61, 62

model
about 45, 46
attributes 46
chemistry 46
differential 46
directed graphs 46
features 46
features, extracting 48
features, selecting 47
grammar and lexicon 47
graphical 46
inference logic 47
numerical method 46
parametric 46
probabilistic 46
taxonomy 47
variables 46
versus design 47

model, assessing
about 68
bias-variance decomposition 77
cross-validation 75
overfitting 81
validation 68

model categorization
about 9
adaptive modeling 9
descriptive models 9
predictive models 9

modeling 45-47

monadic composition 7

monadic data transformation

- about 49
- error handling 50, 51
- explicit model 50-53
- implicit model 50-54

monads 4-7

Monitor class 553

morphism 50

moving averages

- about 89
- exponential moving average 93-97
- simple moving average 90, 91
- weighted moving average 92, 93

multilayer perceptron

- about 347, 348
- activation function 348, 349
- configuration 351
- design 350
- model 357
- network components 351
- network topology 349
- online training, versus batch training 358, 359
- problem types (modes) 357, 358
- training and classification 374
- training epoch 359-361
- UML class diagram 350

multinomial Naïve Bayes model

- about 172, 173
- formalism 174, 175
- frequentist perspective 175, 176
- predictive model 176, 177
- zero-frequency problem 177

Multivariate Bernoulli classification

- about 191
- implementation 192
- model 192

mutation operator, genetic algorithm implementation

- about 422
- chromosomes 422
- genes 423
- population 422

N

Naïve Bayes

- applying, to text mining 193-195

Naïve Bayes algorithm

- cons 206
- pros 205

Naïve Bayes classifiers

- about 171
- multinomial Naïve Bayes 172, 173

Naïve Bayes classifiers implementation

- about 178
- classification 185, 186
- design 178, 179
- F1 validation 186, 187
- feature extraction 187-190
- testing 190, 191
- training 179

Naïve Bayes models

- about 171
- mathematical notation 174

natural language processing (NLP) 285

net profit margin 565

net sales 565

network components, multilayer perceptron

- about 351
- connections 355, 356
- initialization weights 356
- input and hidden layers 353, 354
- network topology 352, 353
- output layer 354, 355
- synapses 355

n-grams 195

nonlinear least squares minimization

- about 562
- Gauss-Newton 562
- Levenberg-Marquardt 562

non-linear models, dimension reduction

- about 165
- kernel PCA 165
- manifolds 166

nonlinear SVM

- about 311
- kernel trick 312
- max-margin classification 311

NP problems

about 398
categories 398

numerical optimization

about 234, 235
Newton 235
Quasi-Newton 235

Nu-SVM 310

O

observation 48

one-class SVC

used, for anomaly detection 335, 336

one-variate linear regression

about 208
implementation 208-210
test case 210, 211

online training 358, 359

operating income 564

operating profit margin 565

optimal substructures 563

optimization techniques

about 558
gradient descent methods 559
Lagrange multipliers 563
nonlinear least squares minimization 562
Quasi-Newton algorithms 560

OptionModel class 468

OptionProperty class 467

options trading 569, 570

option trading, with Q-learning

about 465-467
OptionProperty class 467-469
quantization 469, 470

ordinary least squares regression

about 211, 212
design 212-214
feature selection, test case 2 218-224
implementation 215, 216
trending, test case 1 216-218

overfitting 81, 176

overlapping substructures 564

overload operators

+ 546
+= 546
about 546

P

padding 402

parallel collections, Scala

about 496
benchmark framework 497, 498
performance evaluation 499-501

Parallel Colt

URL 15

Partial Least Square Regression (PLSR) 165

partially connected neural networks 349

pay-out ratio 565

penalized least squares regression 227

penalty term 207

performance considerations

about 166
EM 167
K-means 166, 167
PCA 167, 168

performance evaluation, Spark

about 534
parameters, tuning 535
performance considerations 537
tests 535, 536

Pittsburgh approach 479

Pool 412

posterior probability 174

Predicted Residual Error Sum of Squares (PRESS) 165

predictive model 176

price/book value ratio (PB) 564

price/earnings ratio (PE) 564

price patterns 569

price/sales ratio (PS) 564

Price to Earnings/Growth (PEG) 564

primal problem 310

principal components analysis, dimension reduction

about 158
algorithm 159, 160
evaluation 163-165
implementation 160, 161
test case 162

probabilistic graphical models 169

probabilistic kernels 303

probabilistic reasoning 169

propositional logic 557
protein sequence annotation 300

Q

Q-learning
about 444
action-value iterative update 451, 452
Bellman optimality equations 448, 449
evaluation 474-476
for option trading 465-467
implementing 471-473
temporal difference, for model-free learning 450, 451
QR decomposition 212
QStar class 272
quantization 401
Quasi-Newton algorithms
about 560
Broyden-Fletcher-Goldfarb-Shanno (BFGS) 561
Limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) 561

R

real-world Bayesian network
example 171
recombination 399
reconstruction/error minimization, K-means clustering
about 137
iterative implementation 139
K-means components, creating 137, 138
tail recursive implementation 138, 139
recursive algorithm, discrete Kalman filter
about 114-116
correction 118
experimentation 121-125
fixed lag smoothing 121
Kalman smoothing 119-121
prediction phase 117, 118
regression model 214
regression weights 208
regularization
about 207, 225, 226

Ln roughness penalty 226
ridge regression 227
reinforcement learning
about 9, 14, 443
cons 477
problem 444
pros 477
Q-learning 444
terminologies 445
value of a policy 447, 448
reinforcement learning agent
overview architecture 446
reproducible kernel Hilbert spaces 304
residuals mean square (RMS) 245
residual sum of squares (RSS) 207
resilient distributed dataset (RDD)
about 521
action 521
transformation 521
ridge regression
about 207, 226, 227
design 228
implementation 229, 230
test case 231-234
Riemann metric 304

S

Scala
about 3, 15, 493
abstraction 4
computation 9
configurability 8
features 3
maintainability 9
object creation 493
parallel collections 495
scalability 7, 8
streams 493-495
time series 83
scalability 7, 8
scalability, with Actors
about 502
Actor model 502, 503
partitioning 504
reactive programming 504

Scalable frameworks 492
Scala plugin for Eclipse
reference 15
Scala plugin for IntelliJ IDEA
reference 15
Scala programming
about 541
class constructor template 544
code snippets format 542, 543
companion objects, versus case classes 545
Counter class 552, 553
data extraction 548, 549
design template, for classifiers 546, 547
DMatrix class 552
document extraction 551
encapsulation 544
enumerations, versus case classes 545, 546
data sources 550
libraries directory 542
Monitor class 553
overload operators 546
Scalaz 4
semi-supervised learning 14
Sequential Minimal Optimization (SMO) 310, 313
short interest 565
short interest ratio 565
shrinkage 226
Simple Build Tool (SBT) 16
simple moving average 90, 91
simple workflow
classifier, implementing 32-34
data, analyzing 30
data loading 26
data, plotting 30-32
data, preprocessing 27
immutable normalization 27-29
model, evaluating 40-43
model, training 36-39
observations, classifying 40
optimizer, selecting 34, 35
patterns, discovering 30
problem, scoping 24, 25
writing 23
singular value decomposition (SVD) 168, 555
smoothing factor for counters 178
smoothing kernels 303
soft margin 309
source code
about 18
context, versus view bounds 19
immutability 21
implicit conversion 21
presentation 19, 20
primitive types 20
Scala iterators, performance 22
type conversions 21
Spark. *See Apache Spark*
Spark ecosystem 522
Sparkling Water 538
spectral density estimation
purpose 97
stackable trait injection 58
state space estimation, discrete Kalman filter
about 113
measurement equation 114
transition equation 113
steepest descent 559
stemming 195
stimuli 344
stochastic gradient descent 212, 560
substructures 563
sum of squared errors (SSE) 208
supervised learning 11
supervised machine learning algorithms
about 11
discriminative models 12, 13
generative models 11, 12
support vector classifiers. *See SVC*
support vector machines (SVMs)
about 299, 307
components 314, 315
configuration parameters 315
linear SVM 307
nonlinear SVM 311
performance considerations 342
support vector regression. *See SVR*
SVC
about 313
binary SVC 313
one-class SVC 335
SVM dual problem
kernel trick 312

SVMLight 313

SVR

about 337

overview 337, 338

versus linear regression 339-341

T

tagging model 195

TaskSupport 496

taxonomy, machine learning algorithms

about 10

reinforcement learning 14

semi-supervised learning 14

supervised learning 11

unsupervised learning 10

technical analysis

about 565

price patterns 569

trading data 567

trading signal and strategy 568

technical analysis, terminology

bearish or bearish position 566

bullish or bullish position 566

long position 566

neutral position 566

oscillator 566

overbought 566

oversold 566

relative strength index (RSI) 566

resistance 566

short position 566

support 566

technical indicator 566

trading range 566

trading signal 566

volatility 566

temporal difference 450

terminology, LCS

action 480

agent 479

classifier 480

compound predicate 480

covering 480

environment 479

input data stream 480

predicate 480

predictor 480

rule 480

rule fitness or score 480

rule matching 480

sensors 480

terminology, reinforcement learning

absorbing state 445

action 445

agent 445

best policy 445

environment 445

episode 445

goal 445

horizon 445

policy 445

reward 445

state 445

terminal state 445

test case, evaluation

about 383-385

evaluation of models 386-388

impact of the hidden layers'

 architecture 388, 389

 implementation 385, 386

test case, trading strategy

about 432-434

best trading strategy, finding 437

optimizer, configuring 436

trading strategies, creating 434, 435

testing, Naïve Bayes

about 201

text mining classifier, evaluating 203-205

textual information, retrieving 201-203

tests, genetic algorithms

about 437

unweighted score 440

weighted score 438, 439

text analytics, conditional

random field (CRF)

about 283

CRF classifier, configuring 287-289

CRF model, applying 293

CRF model, training 290-292

design 286

feature functions model 284-286

impact, of L2 regularization factor 296

impact, of size of training set 295

implementation 287
tests 293, 294
training convergence profile 294

text mining
about 193
Naïve Bayes, applying to 193-195

text mining methodology
documents, analyzing 197, 198
features, generating 200, 201
frequency of relative terms,
extracting 199, 200
implementing 196, 197

ThreadPoolTaskSupport 496

time series, in Scala
about 83
differential operator 88
lazy views 89
magnet pattern 87
transpose operator 87
types and operations 84-86

tools 15

trading signal 568

trading strategies
about 428
cost function 429
trading operators 428, 429
trading signal encoding 432
trading signals 430
trading strategies 430, 431

training and classification,
multilayer perceptron
about 374
Fast Fisher-Yates shuffle 376, 377
model fitness 377, 378
model generation 375, 376
prediction 377
regularization 374, 375

training epoch, multilayer perceptron
about 359-361
error backpropagation 366
exit condition 372
implementing 373, 374
input forward propagation 361, 362

training, hidden Markov model (HMM)
about 266
Baum-Welch estimator 266-269

training, Naïve Bayes classifiers
implementation
about 179
binomial model 181, 182
classifier components 183-185
class likelihood 180, 181
multinomial model 182, 183

training workflow, logistic regression
about 240-242
binomial multivariate logistic regression,
testing 245, 246
convergence of optimizer, managing 244
Jacobian matrix, computing 243, 244
least squares problem, defining 244, 245
optimizer, configuring 242
sum of square errors, minimizing 245

trending 216

two-step lag smoothing algorithm 121

Typesafe Activator
URL 505

U

unsupervised learning
about 10
data clustering 10
dimension reduction 11

V

validation, model
about 68
F-score for binomial classification 70-72
F-score for multinomial classification 72-74
key quality metrics 69, 70

variance-bias trade-off 80

vector quantization 128

view bounds 19

Viterbi algorithm
about 270-272
delta 271
psi 271
qStar 271

ViterbiPath class 273

ViterbiPath object 273

W

- weighted moving average** 92, 93
- WordNet** 196
- workflow computational model**
 - about 55
 - mathematical abstractions, supporting 56
 - mixins, combining to build workflow 58
 - modularization 63-65

X

- XCS.** *See extended learning classifier systems*

Y

- Yahoo Finances** 24
 - YahooFinancials** 550
- ## **Z**
- zero-frequency problem** 177



Thank you for buying **Scala for Machine Learning**

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 that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

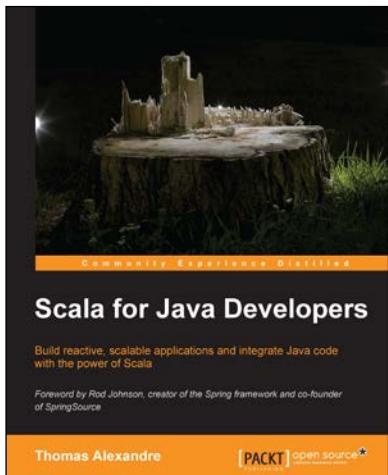
About Packt Open Source

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 licenses, 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, then please 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.

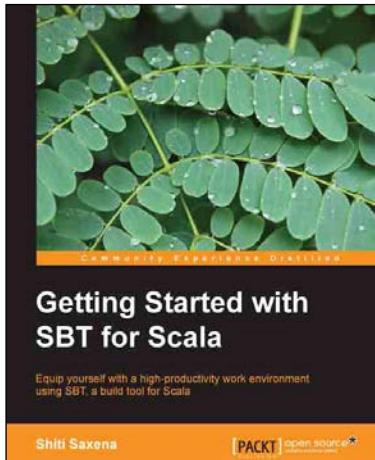


Scala for Java Developers

ISBN: 978-1-78328-363-7 Paperback: 282 pages

Build reactive, scalable applications and integrate Java code with the power of Scala

1. Learn the syntax interactively to smoothly transition to Scala by reusing your Java code.
2. Leverage the full power of modern web programming by building scalable and reactive applications.
3. Easy to follow instructions and real world examples to help you integrate Java code and tackle Big Data challenges.



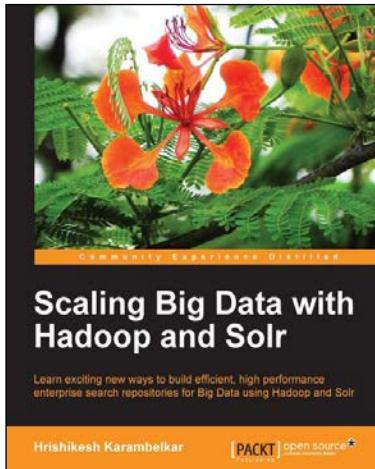
Getting Started with SBT for Scala

ISBN: 978-1-78328-267-8 Paperback: 86 pages

Equip yourself with a high-productivity work environment using SBT, a build tool for Scala

1. Establish simple and complex projects quickly.
2. Employ Scala code to define the build.
3. Write build definitions that are easy to update and maintain.
4. Customize and configure SBT for your project, without changing your project's existing structure.

Please check www.PacktPub.com for information on our titles

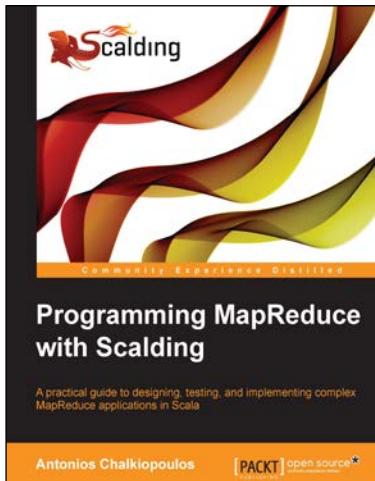


Scaling Big Data with Hadoop and Solr

ISBN: 978-1-78328-137-4 Paperback: 144 pages

Learn exciting new ways to build efficient, high performance enterprise search repositories for Big Data using Hadoop and Solr

1. Understand the different approaches of making Solr work on Big Data as well as their benefits and drawbacks.
2. Learn from interesting, real-life use cases for Big Data search along with sample code.
3. Work with distributed enterprise search without prior knowledge of Hadoop and Solr.



Programming MapReduce with Scalding

ISBN: 978-1-78328-701-7 Paperback: 148 pages

A practical guide to designing, testing, and implementing complex MapReduce applications in Scala

1. Develop MapReduce applications using a functional development language in a lightweight, high-performance, and testable way.
2. Recognize the Scalding capabilities to communicate with external data stores and perform machine learning operations.
3. Full of illustrations and diagrams, practical examples, and tips for deeper understanding of MapReduce application development.

Please check www.PacktPub.com for information on our titles