



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

www.it-ebooks.info

[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 © 2014 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 2014

Production reference: 1121214

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

Indexer

Mariammal Chettiar

Acquisition Editor

Owen Roberts

Graphics

Sheetal Aute
Valentina D'silva
Disha Haria
Abhinash Sahu

Content Development Editor

Mohammed Fahad

Production Coordinator

Arvindkumar Gupta

Technical Editors

Madhuri Das
Taabish Khan

Copy Editors

Janbal Dharmaraj
Vikrant Phadkay

Cover Work

Arvindkumar Gupta

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	1
Chapter 1: Getting Started	9
Mathematical notation for the curious	10
Why machine learning?	10
Classification	10
Prediction	11
Optimization	11
Regression	11
Why Scala?	11
Abstraction	11
Scalability	12
Configurability	13
Maintainability	14
Computation on demand	14
Model categorization	14
Taxonomy of machine learning algorithms	15
Unsupervised learning	15
Clustering	15
Dimension reduction	16
Supervised learning	16
Generative models	16
Discriminative models	17
Reinforcement learning	18
Tools and frameworks	19
Java	19
Scala	20
Apache Commons Math	20
Description	20

Table of Contents

Licensing	20
Installation	21
JFreeChart	21
Description	21
Licensing	21
Installation	22
Other libraries and frameworks	22
Source code	22
Context versus view bounds	23
Presentation	23
Primitives and implicits	24
Primitive types	24
Type conversions	24
Operators	25
Immutability	25
Performance of Scala iterators	26
Let's kick the tires	26
Overview of computational workflows	26
Writing a simple workflow	28
Selecting a dataset	28
Loading the dataset	29
Preprocessing the dataset	30
Creating a model (learning)	34
Classify the data	36
Summary	37
Chapter 2: Hello World!	39
Modeling	39
A model by any other name	39
Model versus design	41
Selecting a model's features	41
Extracting features	42
Designing a workflow	42
The computational framework	44
The pipe operator	44
Monadic data transformation	45
Dependency injection	46
Workflow modules	48
The workflow factory	49
Examples of workflow components	51
The preprocessing module	51
The clustering module	52

Assessing a model	54
Validation	54
Key metrics	54
Implementation	56
K-fold cross-validation	57
Bias-variance decomposition	58
Overfitting	61
Summary	62
Chapter 3: Data Preprocessing	63
Time series	63
Moving averages	66
The simple moving average	67
The weighted moving average	68
The exponential moving average	69
Fourier analysis	73
Discrete Fourier transform (DFT)	73
DFT-based filtering	79
Detection of market cycles	82
The Kalman filter	85
The state space estimation	86
The transition equation	86
The measurement equation	87
The recursive algorithm	87
Prediction	89
Correction	91
Kalman smoothing	92
Experimentation	93
Alternative preprocessing techniques	97
Summary	97
Chapter 4: Unsupervised Learning	99
Clustering	100
K-means clustering	101
Measuring similarity	101
Overview of the K-means algorithm	103
Step 1 – cluster configuration	103
Step 2 – cluster assignment	107
Step 3 – iterative reconstruction	108
Curse of dimensionality	109
Experiment	111
Tuning the number of clusters	114
Validation	117

Table of Contents

Expectation-maximization (EM) algorithm	118
Gaussian mixture model	119
EM overview	120
Implementation	120
Testing	123
Online EM	126
Dimension reduction	126
Principal components analysis (PCA)	127
Algorithm	128
Implementation	129
Test case	130
Evaluation	131
Other dimension reduction techniques	133
Performance considerations	133
K-means	133
EM	134
PCA	134
Summary	135
Chapter 5: Naïve Bayes Classifiers	137
Probabilistic graphical models	137
Naïve Bayes classifiers	139
Introducing the multinomial Naïve Bayes	139
Formalism	141
The frequentist perspective	142
The predictive model	144
The zero-frequency problem	145
Implementation	145
Software design	145
Training	146
Classification	151
Labeling	152
Results	154
Multivariate Bernoulli classification	155
Model	155
Implementation	156
Naïve Bayes and text mining	156
Basics of information retrieval	158
Implementation	159
Extraction of terms	160
Scoring of terms	161
Testing	163
Retrieving textual information	163
Evaluation	166

Pros and cons	168
Summary	168
Chapter 6: Regression and Regularization	169
Linear regression	169
One-variate linear regression	170
Implementation	170
Test case	171
Ordinary least squares (OLS) regression	173
Design	173
Implementation	174
Test case 1 – trending	175
Test case 2 – features selection	178
Regularization	184
L_1 roughness penalty	184
The ridge regression	186
Implementation	186
The test case	188
Numerical optimization	191
The logistic regression	192
The logit function	192
Binomial classification	193
Software design	196
The training workflow	197
Configuring the least squares optimizer	198
Computing the Jacobian matrix	199
Defining the exit conditions	200
Defining the least squares problem	201
Minimizing the loss function	201
Test	202
Classification	203
Summary	205
Chapter 7: Sequential Data Models	207
Markov decision processes	207
The Markov property	208
The first-order discrete Markov chain	208
The hidden Markov model (HMM)	209
Notation	211
The lambda model	212
HMM execution state	214
Evaluation (CF-1)	216
Alpha class (the forward variable)	217
Beta class (the backward variable)	220

Table of Contents

Training (CF-2)	222
Baum-Welch estimator (EM)	222
Decoding (CF-3)	226
The Viterbi algorithm	226
Putting it all together	228
Test case	230
The hidden Markov model for time series analysis	232
Conditional random fields	232
Introduction to CRF	233
Linear chain CRF	235
CRF and text analytics	237
The feature functions model	238
Software design	240
Implementation	241
Building the training set	242
Generating tags	243
Extracting data sequences	244
CRF control parameters	244
Putting it all together	245
Tests	246
The training convergence profile	247
Impact of the size of the training set	247
Impact of the L_2 regularization factor	248
Comparing CRF and HMM	249
Performance consideration	250
Summary	250
Chapter 8: Kernel Models and Support Vector Machines	251
Kernel functions	252
Overview	252
Common discriminative kernels	254
The support vector machine (SVM)	256
The linear SVM	256
The separable case (hard margin)	257
The nonseparable case (soft margin)	258
The nonlinear SVM	260
Max-margin classification	260
The kernel trick	261
Support vector classifier (SVC)	262
The binary SVC	262
LIBSVM	262
Software design	263
Configuration parameters	264
SVM implementation	267

Table of Contents

C-penalty and margin	269
Kernel evaluation	272
Application to risk analysis	277
Anomaly detection with one-class SVC	282
Support vector regression (SVR)	284
Overview	284
SVR versus linear regression	285
Performance considerations	288
Summary	288
Chapter 9: Artificial Neural Networks	289
Feed-forward neural networks (FFNN)	289
The Biological background	290
The mathematical background	291
The multilayer perceptron (MLP)	293
The activation function	294
The network architecture	295
Software design	296
Model definition	297
Layers	298
Synapses	299
Connections	299
Training cycle/epoch	300
Step 1 – input forward propagation	301
Step 2 – sum of squared errors	305
Step 3 – error backpropagation	305
Step 4 – synapse/weights adjustment	308
Step 5 – convergence criteria	309
Configuration	309
Putting all together	310
Training strategies and classification	312
Online versus batch training	312
Regularization	313
Model instantiation	313
Prediction	314
Evaluation	315
Impact of learning rate	315
Impact of the momentum factor	316
Test case	317
Implementation	319
Models evaluation	321
Impact of hidden layers architecture	323
Benefits and limitations	324
Summary	326

Table of Contents

Chapter 10: Genetic Algorithms	327
Evolution	327
The origin	328
NP problems	328
Evolutionary computing	329
Genetic algorithms and machine learning	330
Genetic algorithm components	330
Encodings	331
Value encoding	331
Predicate encoding	332
Solution encoding	333
The encoding scheme	334
Genetic operators	335
Selection	336
Crossover	338
Mutation	339
Fitness score	340
Implementation	340
Software design	340
Key components	341
Selection	344
Controlling population growth	345
GA configuration	345
Crossover	345
Population	346
Chromosomes	347
Genes	348
Mutation	349
Population	349
Chromosomes	349
Genes	349
The reproduction cycle	350
GA for trading strategies	351
Definition of trading strategies	352
Trading operators	353
The cost/unfitness function	353
Trading signals	354
Trading strategies	355
Signal encoding	356
Test case	357
Data extraction	358
Initial population	358
Configuration	359
GA instantiation	359

Table of Contents

GA execution	360
Tests	360
Advantages and risks of genetic algorithms	363
Summary	364
Chapter 11: Reinforcement Learning	365
Introduction	365
The problem	366
A solution – Q-learning	366
Terminology	367
Concept	368
Value of policy	369
Bellman optimality equations	370
Temporal difference for model-free learning	371
Action-value iterative update	372
Implementation	373
Software design	373
States and actions	374
Search space	375
Policy and action-value	376
The Q-learning training	378
Tail recursion to the rescue	380
Prediction	381
Option trading using Q-learning	382
Option property	383
Option model	384
Function approximation	385
Constrained state-transition	386
Putting it all together	387
Evaluation	389
Pros and cons of reinforcement learning	391
Learning classifier systems	391
Introduction to LCS	392
Why LCS	393
Terminology	394
Extended learning classifier systems (XCS)	395
XCS components	396
Application to portfolio management	396
XCS core data	398
XCS rules	399
Covering	401
Example of implementation	401
Benefits and limitation of learning classifier systems	402
Summary	403

Table of Contents

Chapter 12: Scalable Frameworks	405
Overview	406
Scala	407
Controlling object creation	407
Parallel collections	407
Processing a parallel collection	408
Benchmark framework	409
Performance evaluation	410
Scalability with Actors	413
The Actor model	413
Partitioning	415
Beyond actors – reactive programming	415
Akka	415
Master-workers	417
Messages exchange	417
Worker actors	418
The workflow controller	419
The master Actor	419
Master with routing	421
Distributed discrete Fourier transform	422
Limitations	425
Futures	425
The Actor life cycle	426
Blocking on futures	426
Handling future callbacks	428
Putting all together	430
Apache Spark	431
Why Spark	432
Design principles	433
In-memory persistency	433
Laziness	433
Transforms and Actions	434
Shared variables	436
Experimenting with Spark	437
Deploying Spark	437
Using Spark shell	438
MLlib	439
RDD generation	439
K-means using Spark	440
Performance evaluation	442
Tuning parameters	442
Tests	443
Performance considerations	444
Pros and cons	445
Oxdata Sparkling Water	446
Summary	446

Appendix A: Basic Concepts	447
Scala programming	447
List of libraries	447
Format of code snippets	448
Encapsulation	449
Class constructor template	449
Companion objects versus case classes	450
Enumerations versus case classes	450
Overloading	451
Design template for classifiers	452
Data extraction	453
Data sources	454
Extraction of documents	455
Matrix class	456
Mathematics	457
Linear algebra	457
QR Decomposition	458
LU factorization	458
LDL decomposition	458
Cholesky factorization	458
Singular value decomposition	459
Eigenvalue decomposition	459
Algebraic and numerical libraries	459
First order predicate logic	460
Jacobian and Hessian matrices	461
Summary of optimization techniques	462
Gradient descent methods	462
Quasi-Newton algorithms	463
Nonlinear least squares minimization	464
Lagrange multipliers	465
Overview of dynamic programming	466
Finances 101	467
Fundamental analysis	467
Technical analysis	468
Terminology	468
Trading signals and strategy	469
Price patterns	471
Options trading	471
Financial data sources	472
Suggested online courses	473
References	473
Index	475

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 3.3 (*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 1.0.2 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 *Format of code snippets* 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 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. Loading the data.
2. Preprocessing, analyzing, and filtering the input data.
3. Discovering patterns, affinities, clusters, and classes.
4. Selecting the model features and the appropriate machine learning algorithm(s).
5. Refining and validating the model.
6. Improving the computational performance of the implementation.

As we will emphasize throughout this book, each stage of the process is critical to build the *right* model.

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:



Mean value of a variable $X = \{x\}$ is defined as:

$$E(X) = \frac{1}{n} \sum x_j$$

The variance of a variable $X = \{x\}$ is defined as:

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

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 extracted and validated against the past data, it can be used to draw inference from the future data. 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 square), polynomial, and logistic regressions are among the most commonly used techniques to *fit* a parametric model, or function, $y = f(x_j)$, 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 dynamically 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 to machine learning and statistical analysis.

Abstraction

Monoids and monads are important concepts in functional programming.

Monads are derived from the category and group theory allowing developers to create a high-level abstraction as illustrated in **Twitter's Algebird** (<https://github.com/twitter/algebird>) or **Google's Breeze Scala** (<https://github.com/dlwh/breeze>) libraries.

A **monoid** defines a binary operation op on a dataset T with the property of closure, identity operation, and associativity.

Let's consider the + operation is defined for a set T using the following monoidal representation:

```
trait Monoid[T] {  
    def zero: T  
    def op(a: T, b: T): C  
}
```

Monoids are associative operations. For instance, if ts_1, ts_2 , and ts_3 are three time series, then the property $ts_1 + (ts_2 + ts_3) = (ts_1 + ts_2) + ts_3$ is true. The associativity of a monoid operator is critical in regards to parallelization of computational workflows.

Monads are structures that can be seen either as containers by programmers or as a generalization of Monoids. 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:

1. Create the collection.
2. Transform the elements of the collection.
3. Flatten nested collections.

A common categorical representation of a monad in Scala is a trait, `Monad`, parameterized with a container type `M`:

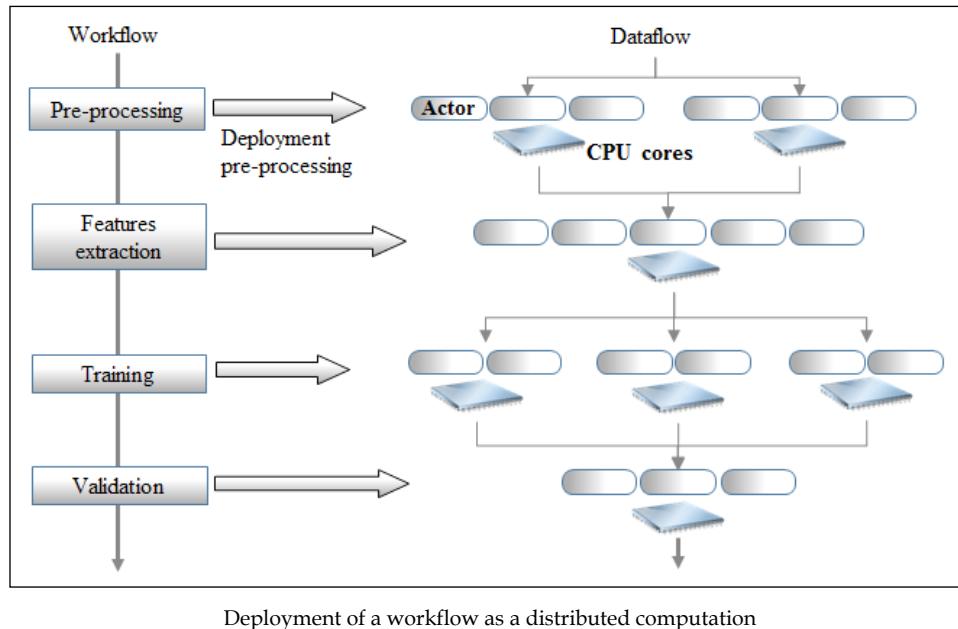
```
trait Monad[M[_]] {  
    def apply[T](a: T): M[T]  
    def flatMap[T, U](m: M[T])(f: T => M[U]): M[U]  
}
```

Monads allow those collections or containers to be chained to generate a workflow. This property is applicable to any scientific computation [1:2].

Scalability

As seen previously, monoids and monads enable parallelization and chaining of data processing functions by leveraging the Scala higher-order methods. In terms of implementation, **Actors** are the core elements that make Scala scalable. Actors act as coroutines, managing the underlying threads pool. Actors communicate through passing asynchronous messages. A distributed computing Scala framework such as Akka and Spark extends the capabilities of the Scala standard library to support computation on very large datasets. Akka and 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. Those tasks consist of high-order Scala methods such as `flatMap`, `map`, `fold`, `reduce`, `collect`, `join`, or `filter` applied to a large collection of observations. Scala allows these observations to be partitioned by executing those tasks through a cluster of actors. Scala also supports message dispatching and routing of messages between local and remote actors. The engineers can decide to execute a workflow either locally or distributed across CPU cores and servers with no code or very little code changes.



In this 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 that are implemented by the Scala actors. The master node exchanges messages with the workers to manage the state of the execution of the workflow as well as its reliability. 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 used throughout this book to create dynamic computation workflows and plots.

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. They are used to predict and classify future events or observations. Predictive analytics is used in a variety of fields such as 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 level in knowledge discovery. They are generated through unsupervised learning.

A third category of models, known as **adaptive modeling**, is generated 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 provide insight into a process, a customer, or an 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 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, and therefore, 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 observations within a cluster and maximizing the observations between clusters. A clustering algorithm consists of the following steps:

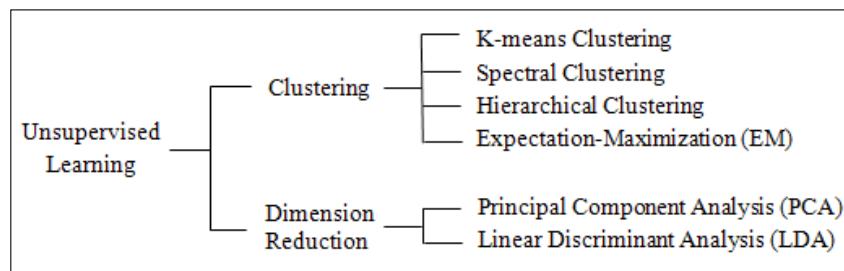
1. Creating a model by making an assumption on the input data.
2. Selecting the objective function or goal of the clustering.
3. Evaluating one or more algorithms to optimize the objective function.

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 that models dataset reliability. 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 extract a relation or function $f: x \rightarrow y$ from a training set $\{x, y\}$. Supervised learning is far more accurate and reliable than any other learning strategy. However, a domain expert may be required to label (tag) data as a training set for certain types of problems.

Supervised machine learning algorithms can be broken into two categories:

- Generative models
- Discriminative models

Generative models

In order to simplify the description of statistics formulas, we adopt the following simplification: the probability of an event X is the same as the probability of the discrete random variable X to have a value x , $p(X) = p(X=x)$. The notation of joint probability (resp. conditional probability) becomes $p(X, Y) = p(X=x, Y=y)$ (resp. $p(X|Y)=p(X=x | Y=y)$.

Generative models attempt to fit a joint probability distribution, $p(X, Y)$, of two events (or random variables), X and Y , representing two sets of observed and hidden (latent) variables x and y . Discriminative models learn 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 an event Y , given an event X , is computed as the product of the conditional probability of the event X , given the event Y , and the probability of the event X normalized by the probability of event Y [1:5].



Join probability (if X and Y are independent):

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

Conditional probability:

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

The Bayes' rule:

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

The Bayes' rule is the foundation of the Naïve Bayes classifier, which is the topic of *Chapter 5, Naïve Bayes Classifiers*.

Discriminative models

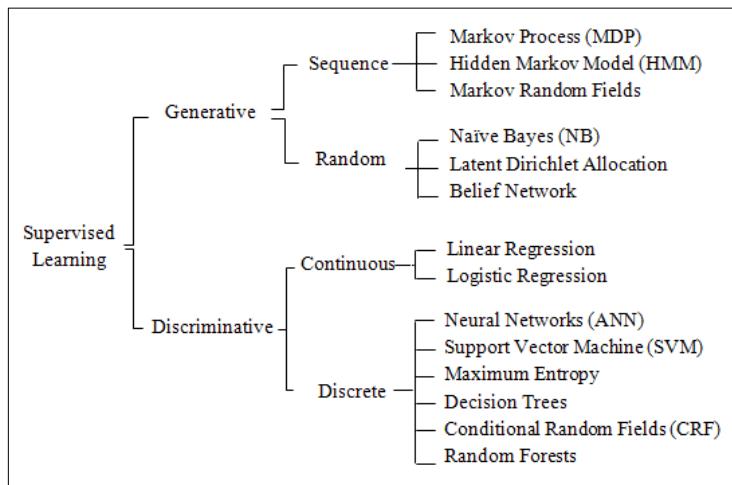
Contrary to generative models, discriminative models compute the conditional probability $p(Y | X)$ directly, using the same algorithm for training and classification.

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

Objective	Generative models	Discriminative models
Accuracy	Highly dependent on the training set.	Probability estimates tend to be more accurate.
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.

Objective	Generative models	Discriminative 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 that introduces 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 between sequential and random variables for generative models and breaking down discriminative methods as applied to continuous processes (regression) and discrete processes (classification):

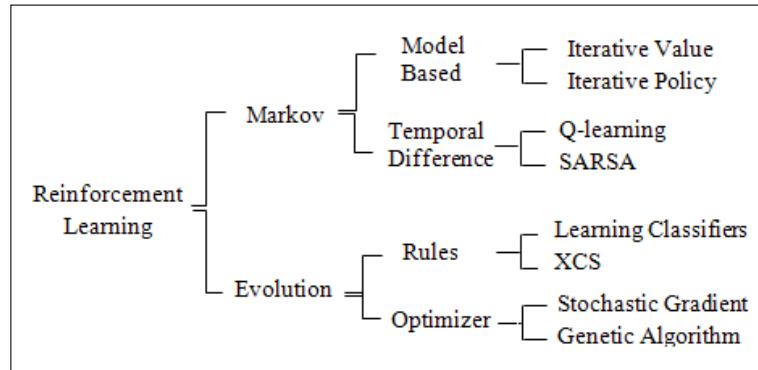


Reinforcement learning

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 compute or estimate the best course of action. Most complex systems based on reinforcement learning establish and update policies that can be vetoed by an expert. The foremost challenge developers of reinforcement learning systems face is that the recommended action or policy may depend on partially observable states and how to deal with uncertainty.

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 rules and policies.

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



This is a brief overview of machine learning algorithms with a suggested 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 and find the documentation appropriate to your level of interest and understanding.

Tools and frameworks

Before getting your hands dirty, you need to download and deploy a minimum set of tools and libraries so as not to reinvent the wheel. 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 the book has been tested with **JDK 1.7.0_45** and **JDK 1.8.0_25** on **Windows x64** and **MacOS X x64**. You need to install the Java Development Kit if you have not already done so. Finally, the environment variables `JAVA_HOME`, `PATH`, and `CLASSPATH` have to be updated accordingly.

Scala

The code has been tested with Scala 2.10.4. We recommend using Scala version 2.10.3 or higher and SBT 0.13 or higher. Let's assume that Scala runtime (REPL) and libraries have been properly installed and environment variables `SCALA_HOME` and `PATH` have been updated. The description and installation instructions of the Scala plugin for Eclipse are available at <http://scala-ide.org/docs/user/gettingstarted.html>.

You can also download the Scala plugin for IntelliJ IDEA 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 version 0.13, and is used to compile and assemble the source code presented throughout this book.

Apache Commons Math

Apache Commons Math is a Java library 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 3.3 or higher.

The main components of Apache Commons Math are:

- Functions, differentiation, and integral and ordinary differential equations
- Statistics distribution
- 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 Commons Math library are quite simple:

1. Go to the download page, http://commons.apache.org/proper/commons-math/download_math.cgi.
2. Download the latest .jar files in the **Binaries** section, commons-math3-3.3-bin.zip (for version 3.3, for instance).
3. Unzip and install the .jar files.
4. Add commons-math3-3.3.jar to classpath as follows:
 - For Mac OS X, use the command `export CLASSPATH=$CLASSPATH:/Commons_Math_path/commons-math3-3.3.jar`
 - For Windows, navigate to **System property | Advanced system settings | Advanced | Environment variables...**, then edit the entry of the CLASSPATH variable
5. Add the commons-math3-3.3.jar file to your IDE environment if needed (that is, for Eclipse, navigate to **Project | Properties | Java Build Path | Libraries | Add External JARs**).

You can also download commons-math3-3.3-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 install the .jar file.
4. Add `jfreechart-1.0.17.jar` (for version 1.0.17) to `classpath` as follows:
 - For Mac OS, update the `classpath` by using `export CLASSPATH=$CLASSPATH:/JFreeChart_path/ jfreechart-1.0.17.jar`
 - For Windows, go to **System property | Advanced system settings | Advanced | Environment variables...** and then edit the entry of 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 the respective chapters.

Why not use Scala algebra and numerical libraries

Libraries such as **Breeze**, **ScalaNLP**, and **Algebird** are great 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 simple common Java libraries [1:8].

Source code

The Scala programming language is used to implement and evaluate the machine learning techniques presented in this book. Only a subset of the source code used to implement the techniques are presented in 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 to 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 MyClassInt[T <: Int]
class MyClassFloat[T <: Double]
```

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 only where they require an implicit conversion to the parameterized type to be defined:

```
Class MyClassFloat[T <% Double]
implicit def T2Double(t : T) : Double
```

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 discarded in the code snippet presented in the book:

- Code comments
- Validation of class parameters and method arguments:

```
class BaumWelchEM(val lambda: HMMLambda ...) {
    require(lambda != null, "Lambda model is undefined")
```

- Exceptions and an exception handler:

```
try { ... }
catch {
    case e: ArrayIndexOutOfBoundsException => println(e.
        toString)
}
```

- Nonessential annotation:
`@inline def mean = ...`
- Logging and debugging code:
`m_logger.debug(...)`
- Private and nonessential methods

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 XY = (Double, Double)
type XYTSeries = Array[(Double, Double)]
type DMatrix[T] = Array[Array[T]]
type DVector[T] = Array[T]
type DblMatrix = DMatrix[Double]
type DblVector = Array[Double]
```

The types have the behavior (methods) of their primitive counterpart (array). However, adding a new functionality to vectors, matrices, and time series requires classes of their own right. These classes will be introduced in the next chapter.

Type conversions

Implicit conversion is an important feature of the Scala programming language because 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 used throughout the book:

```
implicit def int2Double(n: Int): Double = n.toDouble
implicit def vectorT2DblVector[T <% Double](vt: DVector[T]): DblVector
= vt.map( t => t.toDouble)
implicit def double2DblVector(x: Double): DblVector = Array[Double](x)
implicit def dblPair2DblVector(x: (Double, Double)): DblVector =
Array[Double](x._1, x._2)
implicit def dblPairs2DblRows(x: (Double, Double)): DblMatrix =
Array[Array[Double]](Array[Double](x._1, x._2))
...
```

**Library-specific conversion**

The conversion between the primitive type listed here and types introduced in a particular library (such as Apache Commons Math) is declared in future chapters the first time those libraries are used.

Operators

Lastly, some operations are applied by multiple machine learning or preprocessing algorithms. They need to be defined implicitly. The operation on a pair of a vector of arbitrary type and vector of Double is defined as follows:

```
def Op[T <% Double] (v: DVector[T], w: DblVector, op: (T, Double) => Double): DblVector =
    v.zipWithIndex.map(x => op(x._1, w(x._2)))
```

It is also convenient to define the following operators that are included in the Scala standard library:

```
implicit def /(v: DblVector, n: Int): DblVector = v.map(x => x/n)
implicit def /(m: DblMatrix, col: Int, z: Double): DblMatrix = {
    0 until m(n).size).foreach(i => m(n)(i) /= z)
}
```

We won't have to redefine the types, conversions, and operators from now on.

Immutability

It is usually a good idea to reduce the number of states of an object. 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:

- 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` loop construct is to be avoided as a counting iterator except if it is used in conjunction with `yield`. It is designed to implement the `for-comprehension` monad (`map`-`flatMap`). The source code presented in this book uses the `while` and `foreach` constructs.

Scala reducer methods `reduce` and `fold` are also frequently used for their efficiency.

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.

Overview of computational workflows

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

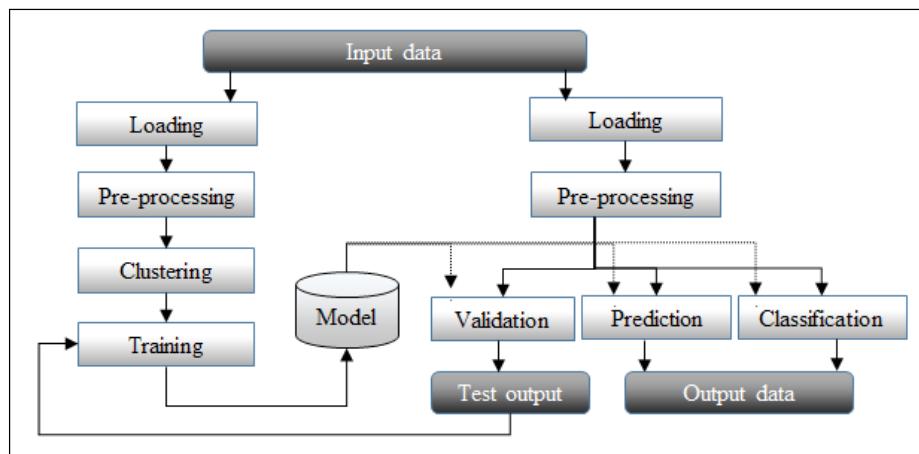
1. Loading the dataset from files, databases, or any streaming devices.
2. Splitting the dataset for parallel data processing.
3. Preprocessing data using filtering techniques, analysis of variance, and applying penalty and normalization functions whenever necessary.
4. Applying the model, either a set 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, 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 criteria.
8. Storing the model into the file or database to be loaded for runtime processing of new observations.

Data clustering and data classification can be performed independent of each other or as part of a workflow that uses clustering techniques as a 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 and classification:



A generic data flow for training and running a model

This 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. In reality, real-world workflows are more complex and require being dynamically configurable 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 of the latent noise in the raw data.

Writing a simple workflow

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



Logistic regression

Logistic regression is treated in depth in *Chapter 6, Regression and Regularization*. The model treated in this example is a simple binary classifier using logistic regression for two-dimensional observations.

The classification of trading sessions according to their volatility is as follows:

- Select a dataset
- Load the dataset
- Preprocess the dataset
- Display data
- Create the model through training
- Classify new data

Selecting a dataset

Throughout the book, we will rely on financial data to evaluate and discuss the merit 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 session
- Lowest price in session
- Price at session close
- Volume
- Adjust price at session close

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 two variables, price and volume, as illustrated by 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:



Price-Volume action for the Cisco stock

Loading the dataset

The first step is loading the dataset from a local file. Typically, large datasets are loaded from a database or distributed filesystem such as **Hadoop Distributed File System (HDFS)**, as shown here:

```
def load(fileName: String): Option[XYTSeries] = {
    val src = Source.fromFile(fileName)
    val fields = src.getLines.map(_.split(CSV_DELIM)).toArray //1
    val cols = fields.drop(1) //2
    val data = transform(cols)
    src.close //3
    Some(data)
}
```

The `transform` method will be described in the next section.

The data file is extracted through an invocation of the `Source.fromFile` static method, and then the fields are extracted through a map (line 1). The header (first) row is removed with a call to `drop` (line 2).

[
]

Data extraction

The `Source.fromFile.getLines.map` invocation pipeline method returns an iterator, which needs to be converted into an array to store the information into memory.

The file has to be closed to avoid leaking of the file handle (line 3).

Code readability

A long pipeline of Scala high-order methods make the code and underlying code quite difficult to read. It is recommended to break down long chains of method calls. The following code is an example of a long chain of method calls:



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

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

```
val lines = Source.fromFile.getLines  
val fields = lines.map(_.split(CSV_DELIM).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].

Preprocessing the dataset

The next step is to normalize the data in the range [-0.5, 0.5] to be trained by the logistic binary classifier. It is time to introduce a non-sense statistics class.

Basic statistics

We select the computation of mean and standard deviation of the two time series as the first step of the preprocessing phase. The computation of these statistics can be implemented by the reduce methods `reduceLeft` and `foldLeft`:

```
val mean = price.reduceLeft(_ + _) / price.size  
val s2 = price.foldLeft(0.0)((s, x) => s + (x - mean) * (x - mean))  
val stdDev = Math.sqrt(s2 / (price.size - 1))
```

However, this implementation has one major drawback: the dataset (price in this example) has to be traversed for each method (`mean`, `stdDev`, `min`, `max`, and so on).

One of the solutions is to create a class that computes the counters and the statistics on demand using, once again, the lazy values:

```
class Stats[T <% Double](private val values: DVector[T]) {  
    class _Stats(var minValue: Double, var maxValue: Double, var sum:  
    Double, var sumSqr: Double)  
    val stats = {  
        val _stats = new _Stats(Double.MaxValue, Double.MinValue, 0.0, 0.0)
```

```

values.foreach(x => {
  if(x < _stats.minValue) x else _stats.minValue
  if(x > _stats.maxValue) x else _stats.maxValue
  _stats.sum + x
  _stats.sumSqr + x*x
})
_stats
}

lazy val mean = _stats.sum/values.size
lazy val variance = (_stats.sumSqr - mean*mean*values.size)/(values.size-1)
lazy val stdDev = if(variance < ZERO_EPS) ZERO_EPS else Math.sqrt(variance)
lazy val min = _stats.minValue
lazy val max = _stats.mazValue
}

```

We made the statistics object generic by using the view bounds `T <% Double`, which assumes a conversion from type `T` to `Double`. By defining the statistics as tuple counters (minimum value, maximum value, sum of values, and sum of square values) and folding these values into a statistics object, we limit the number of invocations of the `foldLeft` reducer method to 1, and therefore, avoid the recomputation of these statistics for the existing dataset each time new data is added.

The code illustrates the use and benefit of lazy values in Scala. The mean is computed only if and when needed.

Normalization and Gauss distribution

Statistics are usually used to normalize data into a probability value [0, 1] as required by most classification or clustering algorithms. It is logical to add the normalization method to the `Stats` class, as we have already extracted the `min` and `max` values:

```

def normalize: DblVector = {
  val range = max - min;  values.map(x => (x - min)/range)
}

```

The same approach is used to compute the multivariate normal distribution:

```

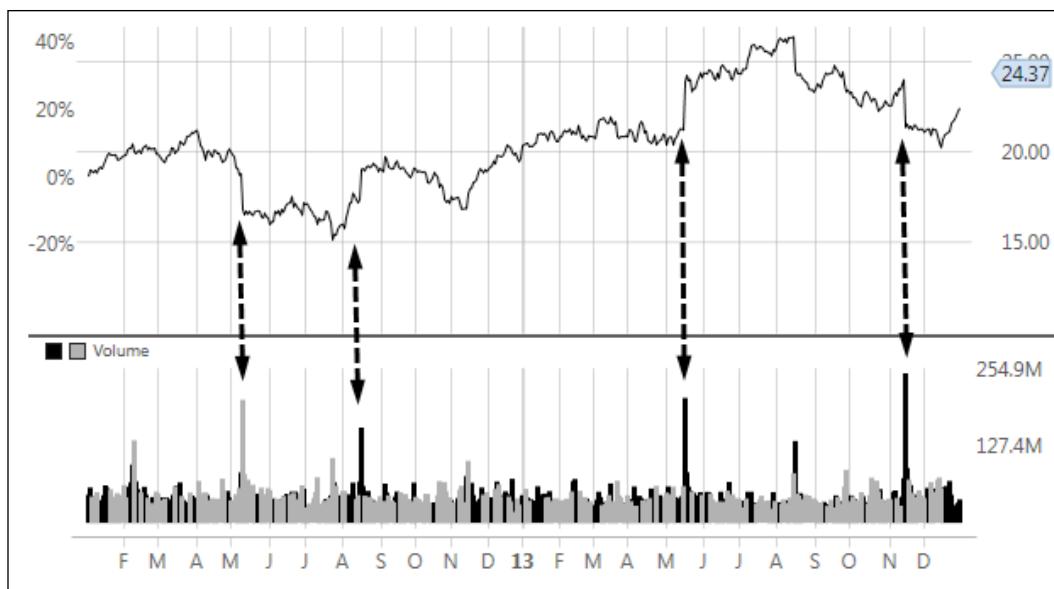
def gauss: DblVector =
  values.map(x =>{
    val y=x-mean
    INV_SQRT_2PI/stdDev*Math.exp(-0.5*y*y/stdDev) })

```

Getting Started

The price action chart has a very interesting characteristic. 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 those price-volume patterns are related to the release of quarterly earnings of Cisco. Such regular but unpredictable patterns can be a source of concern or opportunity if risk can be 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:



Correlation price-volume action for the Cisco stock

Let's try to correlate the volatility of the stock price with volume. For the sake of this exercise, we define the volatility as the maximum variation of the stock price within each trading session: the relative difference between the highest price during the trading session and the lowest price during the session.

The `YahooFinancials` enumeration extracts historical stock prices and session volume from a CSV file. For example, the volatility is extracted from the CSV fields of each line in the CSV file as follows:

```
object YahooFinancials extends Enumeration {
    type YahooFinancials = Value
    val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME, ADJ_CLOSE = Value
```

```
    val volatility = (fs: Array[String]) => fs(HIGH.id).toDouble - fs(LOW.id).toDouble
    ...
}
```

The `transform` method uses the `YahooFinancials` enumeration to generate the input data for the model:

```
def transform(cols: Array[Array[String]]): XYTSeries = {
    val volatility = Stats[Double](cols.map(YahooFinancials.volatility)).normalize
    val volume = Stats[Double](cols.map(YahooFinancials.volume)).normalize
    volatility.zip(volume)
}
```

The `volatility` and `volume` data is normalized using the `Stats.normalize` method defined earlier.

Plotting data

Although charting is not the primary goal of this book, we thought that you will benefit from a brief introduction to JFreeChart. The skeleton code to generate a scatter plot is rather simple. The most relevant code is the transformation of the `XYTSeries` into graphical JFreeChart's `XYSeries`:

```
val xLegend = "Session Volatility"
val yLegend = "Session Volume"
def display(xy: XYTSeries, w: Int, h : Int): Unit = {
    val series = new XYSeries("CSCO 2012-2013 Stock")
    xy.foreach( x => series.add( x._1,x._2))
    val seriesCollection = new XYSeriesCollection
    seriesCollection.addSeries(series)
    ... // plot rendering code
    val chart = ChartFactory.createScatterPlot(xLegend, xLegend,
yLegend, seriesCollection, PlotOrientation.VERTICAL, true, false,
false)
    createFrame("Logistic Regression", chart)
}
```

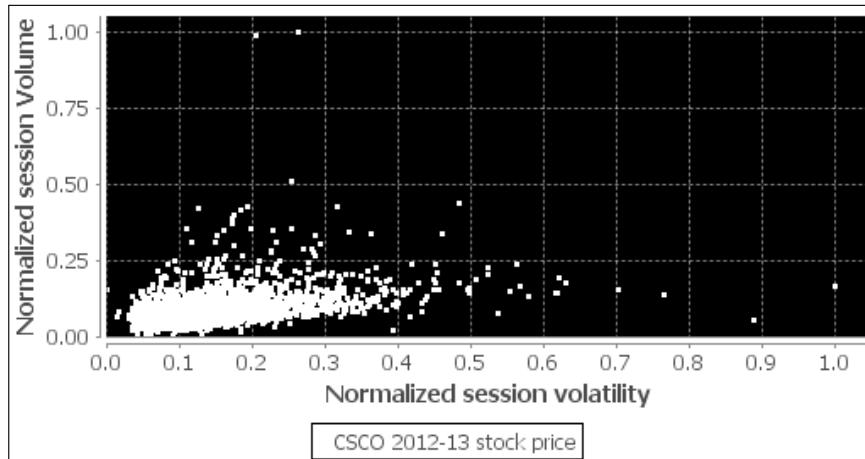
Visualization



The JFreeChart library is introduced as a robust charting tool. The visualization of the results of a computation is beyond the scope of this book. 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. In a few occasions, output data is formatted as a CSV file to be simply imported into a spreadsheet.

Here is an example of a plot using the `ScatterPlot.display` method:

```
val plot = new ScatterPlot(("CSCO 2012-2013", "Session High - Low",
  "Session Volume"), new BlackPlotTheme)
plot.display(volatility_vol.filter(_._1 < 0.5), 250, 340)
```



Scatter plot of volatility and volume for the Cisco stock

There is a level of correlation between session volume and session volatility. We can use this information to classify trading sessions by their volatility.

Creating a model (learning)

The objective of the training is to build a model that can discriminate between volatile and nonvolatile trading sessions. For the sake of the exercise, session volatility has been defined as session price high and session price low coupled with heavy trading volume, which constitute the two parameters of the model.

Logistic regression is commonly used in statistics inference. The following implementation of the binary logistic regression classifier exposes a single method, `classify`, to comply with our desire to reduce the complexity and life cycle of objects. The model parameters, `weights`, 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:

```
class LogBinRegression(val labels: DVector[(XY, Double)], val
maxIters: Int, val eta: Double, val eps: Double) {
    val dim = 3
    val weights = train

    def classify(xy: XY): Option[(Boolean, Double)] = {
        if(weights != None) {
            val likelihood = sigmoid(w(0) + xy._1*w(1) + xy._2*w(2))
            Some(likelihood > 0.5, likelihood)
        }
        else None
    }
}
```

The training method, `train`, consists of iterating through the computation of the weight using a simple descent gradient. The method computes the weights and returns an option, so the model is either trained and ready for runtime classification or nonexistent (`None`):

```
def train: Option[DblVector] = {
    val w = Array.fill(dim)(x=> Random.nextDouble-1.0)

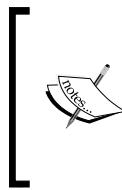
    Range(0, maxIters).find(_ => {
        val deltaW = labels.foldLeft(Array.fill(dim)(0.0))((dw, lbl) => {
            val y = sigmoid(w(0) + w(1)*lbl._1._1 + w(2)*lbl._1._2)
            dw.map(dx => dx + (lbl._2 - y)*(lbl._1._1 + lbl._1._2))
        })
        val nextW = Array.fill(dim)(0.0)
            .zipWithIndex
            .map(nw => w(nw._2)+eta*deltaW(nw._2))
        val diff = Math.abs(nextW.sum - w.sum)
        nextW.copyToArray(w); diff < eps
    }) match {
        case Some(iters) => Some(w)
        case None => { ... }
    }
}
def sigmoid(x: Double):Double = 1.0/(1.0 + Math.exp(-x))
```

The iteration is encapsulated in the Scala `find` method that exists if the algorithm converges (`diff < eps`). The model parameters, `weights`, are set to `None` if the maximum number of iterations is reached.

The training method, `train`, iterates across the set of observations by computing the gradient between the predicted and observed values. In our simplistic approach, the gradient is computed as a linear function of the sigmoid of the sum of the product of the weight and training observations. As for any optimization problem, the initialization of the solution vector, `weights`, is critical. We choose to initialize the weight with random values, although in practice, you would use a more deterministic approach to initialize the model parameters.

In order to train the model, we need to label data. The process consists of tagging every trading session as **volatile** and **non volatile** according to the observations (relative session volatility and session volume). The labeling process is usually quite cumbersome; therefore, let's generate the label automatically. A trading session is considered volatile if a volatility and volume are both greater than 60 percent of the maximum relative volatility and volume:

```
val labels = volatilityVol.zip(volatilityVol.map(x =>if( x._1>0.3 && x._2>0.3) 1.0 else 0.0))
```



Automated labeling

Although quite convenient, automated creation of training labels is not without risk because 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.



The model is created (trained) by a simple instantiation of the logistic binary classifier:

```
val logit = new LogBinRegression(labels, 300, 0.00005, 0.02)
```

The training run is configured with a maximum of 300 iterations, a gradient slope of 0.00005, and convergence criteria of 0.02.

Classify the data

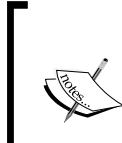
Finally, the model can be tested with a new fresh dataset, not related to the training set:

```
Date,Open,High,Low,Close,Volume,Adj Close  
3/9/2011,14.78,15.08,14.20,14.91,4.79E+08,14.88  
11/17/2009,10.78,10.90,10.62,10.84,3901987,10.85
```

It is just a matter of executing the classification method (exceptions, conditions on method arguments, and returned values are omitted):

```
val testData = load("resources/data/chap1/CSCO2.csv")
logit.classify(testData(0)) match {
  case Some(topCategory) => Display.show(topCategory)
  case None => { ... }
}
logit.classify(testData(1)) match {
  case Some(topCategory) => Display.show(topCategory)
  case None => { ... }
}
```

The result of the classification is `(true, 0.516)` for the first sample and `(false, 0.1180)` for the second sample.



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 metrics and methodology.

Summary

We hope you enjoyed this introduction to machine learning and how to leverage your existing skills in Scala programming to create a simple regression program to predict stock price/volume action. Here are the highlights of this introductory chapter:

- From monadic composition and high-order collection methods for parallelization to configurability to reusability patterns, Scala is the perfect fit to implement and leverage data mining and machine learning algorithms for large-scale projects
- There are many steps to create and apply a machine learning model
- The implementation of the logistic binary 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 traits and dependency injection.

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 functional features, such as dependency injection, 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 . These 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. If it walks like a monoid and quacks like a monoid, then it is a monoid.

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, $dy = 2x \cdot dx$)
- **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, $p(x, y | c) = p(x | c) \cdot p(y | c)$)
- **Directed graphs:** This consists of temporal and spatial relationships (for example, a scheduler)
- **Numerical method:** This consists of finite elements and methods such as Newton-Raphson
- **Chemistry:** This consists of formula and components (for example, H₂O, Fe + C₁₂ = FeC₁₃, 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, Scala programming language)
- **Inference logic:** This consists of a distribution pattern such as IF (stock vol > 1.5 * average) AND rsi > 80 THEN...

Model versus design

The confusion between 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 is describing something you know. A model makes the 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 is manipulating representation for things you don't know. Designing can be seen 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 a model's 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 make the assumption 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.

Selecting features 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 $2n-1$ 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 classes. Using all features from the observation also incurs a high computation cost.

The purpose of **extracting features** 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.

Designing a workflow

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

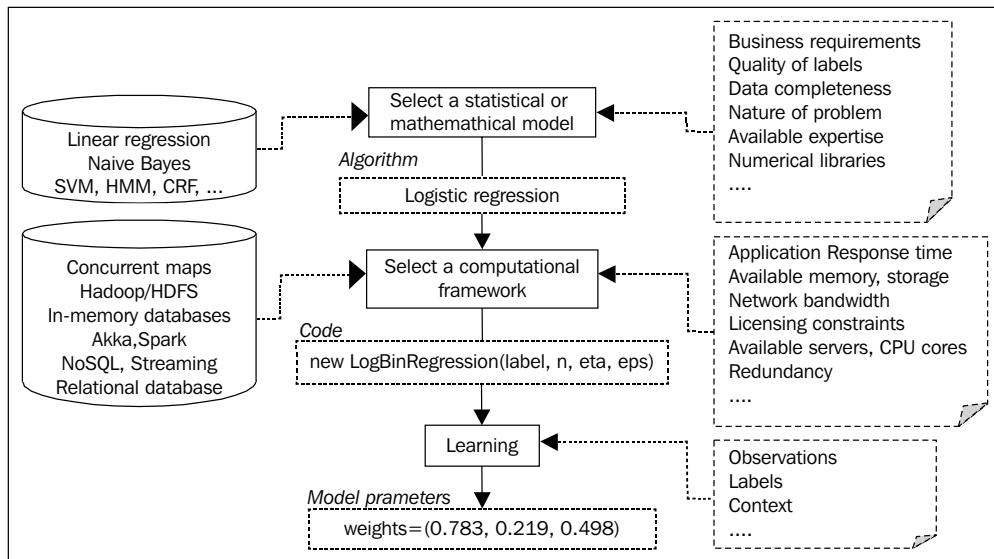
Firstly, a mathematical or statistical model is to be selected to extract knowledge from the raw input data or the output of a data upstream transformation. The selection of the model is constrained by the following parameters:

- Business requirements such as accuracy of results
- Availability of training data and algorithms
- Access to a domain or subject-matter expert

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
- 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

The following diagram illustrates the selection process to define the data transformation for each computation in the 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 creation of software applications, including analysis, design, coding, testing, and deployment.

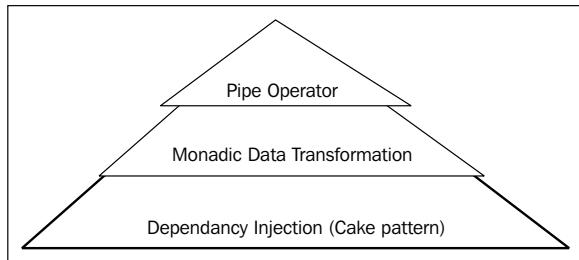


The parameters of a data transformation may need to be reconfigured according to the output of the upstream data transformation. Scala's higher-order functions are particularly suitable for implementing configurable data transformations.

The computational framework

The objective is to create a framework flexible and reusable enough to accommodate different workflows and support all types of machine learning algorithms from preprocessing, data smoothing, and classification to validation.

Scala provides us with a rich toolbox that includes monadic design, design patterns, and dependency injections using traits. The following diagram describes the three levels of complexity for creating the framework:



Hierarchical design of a monadic workflow

The first step is to define a trait and a method that describes the transformation of data by a computation unit (element of the workflow).

The pipe operator

Data transformation is the foundation of any workflow for processing and classifying a dataset, training and validating a model, and displaying results.

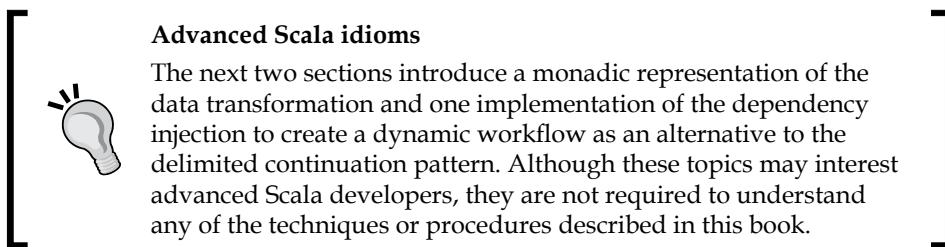
The objective is to define a symbolic representation of the transformation of different types of data without exposing the internal state of the algorithm implementing the data transformation. The pipe operator is used as the signature of a data transformation:

```
trait PipeOperator[-T, +U] {  
    def |> (data: T): Option[U]  
}
```

F# reference

The notation `|>` as the signature of the transform or pipe operator is borrowed from the F# language [2:2]. The data transformation indeed implements a function, and therefore, has the same variance signature as `Function[-T, +R]` of Scala.

The `|>` operator transforms a data of the type `T` into a data of the type `U` and returns an option to handle internal errors and exceptions.



Monadic data transformation

The next step is to create a **monadic design** to implement the pipe operator. Let's use a monadic design to wrap `_fct`, a data transformation function (also known as operator), with the most commonly used Scala higher-order methods:

```
class _FCT[+T] (val _fct: T) {
    def map[U](c: T => U): _FCT[U] = new _FCT[U](c(_fct))
    def flatMap[U](f: T => _FCT[U]): _FCT[U] = f(_fct)
    def filter(p: T => Boolean): _FCT[T] = if( p(_fct) ) new _FCT[T](_fct) else zeroFCT(_fct)
    def reduceLeft[U](f: (U,T) => U)(implicit c: T=> U): U = f(c(_fct), _fct)
    def foldLeft[U](zero: U)(f: (U, T) => U)(implicit c: T=> U): U =
        f(c(_fct), _fct)
    def foreach(p: T => Unit): Unit = p(_fct)
}
```

The methods of the `_FCT` class represent a subset of the traditional Scala higher methods for collections [2:3]. The `_FCT` class is to be used internally. Arguments are validated by subclasses or containers.

Finally, the `Transform` class takes a `PipeOperator` instance as an argument and automatically invokes its operator:

```
class Transform[-T, +U](val op: PipeOperator[T, U]) extends _FCT[Function[T, Option[U]]](op.|>)
    def |>(data: T): Option[U] = _fct(data)
```

You may wonder about the reason behind the monadic representation of a data transformation, `Transform`. You can create any algorithm by just implementing the `PipeOperator` trait, after all. The reason is that `Transform` has a richer protocol (methods) and enables developers to create a complex workflow as an alternative to the delimited continuation. The following code snippet illustrates a generic function composition or data transformation composition using the monadic approach:

```
val op = new PipeOperator[Int, Double] {
  def |> (n: Int): Option[Double] = Some(Math.sin(n.toDouble))
}
def g(f: Int => Option[Double]): (Int=> Long) = {
  (n: Int) => {
    f(n) match {
      case Some(x) => x.toLong
      case None => -1L
    }
  }
}
val gof = new Transform[Int,Double] (op).map(g(_))
```

This code extends `op`, an existing transformation, with another function, `g`. 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 (refer to the *Format of code snippets* section in *Appendix A, Basic Concepts*).

Dependency injection

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. The Cake pattern is an advanced class composition pattern that uses mix-in 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 prefix 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.

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 myApp = 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 myApp = 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 of the application. 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 pattern:

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

The Scala compiler implements the linearization as follows:

`M => B => C => A => 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 can then be simply composed as:

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



Overriding val with def

It is advantageous to override the declaration of a value with a definition of a method with the same signature. Contrary to a value that locks the implementation of the value, a method can return a different value for each invocation:

```
trait PreProcessor { val validation = ... }
trait MyValidator extends Validator { def validation
= ... }
```

In Scala, a value declaration can be overridden by the method definition, not vice versa.

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

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

Workflow modules

The data transformation defined by the `PipeOperator` instance is dynamically injected into the module by initializing the abstract value. Let's define three parameterized modules representing the preprocessing, processing, and post-processing stages of a workflow:

```
trait PreprocModule[-T, +U] { val preProc: PipeOperator[T, U] }
trait ProcModule[-T, +U] { val proc: PipeOperator[T, U] }
trait PostprocModule[-T, +U] { val postProc: PipeOperator[T, U] }
```

The modules (traits) contain only 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, as follows:

```
trait ProcModule[-T, +U] {
  val proc: PipeOperator[T, U]
  class Classification[-T, +U] extends PipeOperator[T, U] { }
}
```

One of the objectives in building the framework is allowing developers to create data transformation (inherited from `PipeOperator`) independently from any workflow. Under these constraints, strict modularity is not an option.

 **Scala traits versus Java packages**

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

The workflow factory

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 paragraph. Here is an implementation of the said `self` reference:

```
class WorkFlow[T, U, V, W] {
    self: PreprocModule[T,U] with ProcModule[U,V] with
PostprocModule[V,W] =>
    def |> (data: T): Option[W] = {
        preProc |> data match {
            case Some(input) => {
                proc |> input match {
                    case Some(output) => postProc |> output
                    case None => { ... }
                }
            }
            case None => { ... }
        }
    }
}
```

Quite simple indeed! If you need only two modules, you can either create a workflow with a stack of two traits or initialize the third with the `PipeOperator` identity:

```
def identity[T] = new PipeOperator[T,T] {
    override def |> (data:T): Option[T] = Some(data)
}
```

Let's test the wiring with the following simple data transformations:

```
class Sampler(val samples: Int) extends PipeOperator[Double => Double,
DblVector] {
    override def |> (f: Double => Double): Option[DblVector] =
```

Hello World!

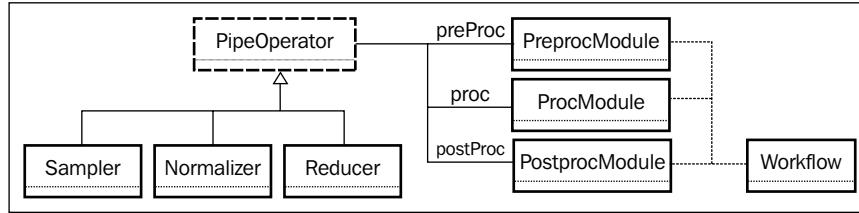
```
Some(Array.tabulate(samples)(n => f(n.toDouble/samples)) )
}

class Normalizer extends PipeOperator[DblVector, DblVector] {
    override def |> (data: DblVector): Option[DblVector] =
        Some(Stats[Double](data).normalize)
}

class Reducer extends PipeOperator[DblVector, Int] {
    override def |> (data: DblVector): Option[Int] =
        Range(0, data.size) find(data(_) == 1.0)
}
```

The first operator, `Sampler`, samples a function, `f`, with a frequency $1/samples$ over the interval $[0, 1]$. The second operator, `Normalizer`, normalizes the data over the range $[0, 1]$ using the `Stats` class introduced in the *Basic statistics* section in *Chapter 1, Getting Started*. The last operator, `Reducer`, extracts the index of the large sample (value 1.0) using the Scala collection method, `find`.

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



Finally, the workflow is instantiated by dynamically initializing the abstract values, `preProc`, `proc`, and `postProc`, with a transformation of the type `PipeOperator` as long as the signature (input and output types) matches the parameterized types defined in each module (lines marked as 1):

```
val dataflow = new Workflow[Double => Double, DblVector, DblVector,
Int]
    with PreprocModule[Double => Double, DblVector]
    with ProcModule[DblVector, DblVector]
    with PostprocModule[DblVector, Int] {

    val preProc: PipeOperator[Double => Double, DblVector] = new
    Sampler(100) //1
```

```

    val proc: PipeOperator[DblVector,DblVector] = new Normalizer //1
    val postProc: PipeOperator[DblVector,Int] = new Reducer//1
}
dataflow |> ((x: Double) => Math.log(x+1.0)+Random.nextDouble) match {
  case Some(index) => ...

```

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.

Examples of workflow components

It is difficult to build an example of workflow using classes and algorithms introduced later in the book. The modularization of the preprocessing and clustering stages is briefly described here to illustrate the encapsulation of algorithms described throughout the book within a workflow.

The preprocessing module

The following examples of a workflow module use the time series class, `XTSeries`, which is used throughout the book:

```
class XTSeries[T] (label: String, arr: Array[T])
```

The `XTSeries` class takes an identifier, a label, and an array of parameterized values, `arr`, as parameters, and is formally described in *Chapter 3, Data Preprocessing*.

The preprocessing algorithms such as moving average or discrete Fourier filters are encapsulated into a preprocessing module using a combination of abstract value and inheritance:

```

trait PreprocessingModule[T] {
  val preprocessor: Preprocessing[T] //1

  abstract class Preprocessing[T] { //2
    def execute(xt: XTSeries[T]): Unit
  }

  abstract class MovingAverage[T] extends Preprocessing[T] with
    PipeOperator[XTSeries[T], XTSeries[Double]] { //3
    override def execute(xt: XTSeries[T]): Unit = this |> xt match {
      case Some(filteredData) => ...
    }
  }
}
```

```
    case None => ...
  }
}

class SimpleMovingAverage[@specialized(Double) T <% Double] (period:
Int)(implicit num: Numeric[T]) extends MovingAverage[T] {
override def |> (xt: XTSeries[T]): Option[XTSeries[Double]] =
...
}
class DFTFir[T <% Double] (g: Double=>Double) extends Preprocessing[T]
extends PreProcessing[T] with PipeOperator[XTSeries[T],
XTSeries[Double]] {
override def execute(xt: XTSeries[T]): Unit = this |> xt match {
  case Some(filteredData) => ...
  case None => ...
}
override def |> (xt: XTSeries[T]): Option[XTSeries[Double]] =
}
}
```

The preprocessing module, `PreprocessingModule`, defines `preprocessor`, an abstract value, that is initialized at runtime (line 1). The `PreProcessing` class is defined as a high-level abstract class with a generic execution function: `execute` (line 2). The preprocessing algorithms; filtering techniques moving average, `MovingAverage`; and discrete Fourier, `DFTFir` in this case, are defined as a class hierarchy with the base type `PreProcessing`. Each filtering class also implements `PipeOperator` so it can be weaved into a simpler data transformation workflow (line 3).

The preprocessing algorithms are described in the next chapter.

The clustering module

The encapsulation of clustering techniques is the second example of a module for dependency-injection-based workflow:

```
trait ClusteringModule[T] {
  type EOutput = List[(Double, DblVector, DblVector)]
  val clustering: Clustering[T]

  abstract class Clustering[T] {
```

```

    def execute(xt: XTSeries[Array[T]]): Unit
}

class KMeans[T <% Double](K: Int, maxIter: Int, distance:
(DblVector, Array[T]) => Double)(implicit order: Ordering[T], m:
Manifest[T]) extends Clustering[T] with PipeOperator[XTSeries[Array[T]]], List[Cluster[T]] {

    override def |> (xt: XTSeries[Array[T]]): Option[List[Cluster[T]]]

    override def execute(xt: XTSeries[Array[T]]): Unit = this |> xt
    match {
        case Some(clusters) => ...
        case None => ...
    }
}

class MultivariateEM[T <% Double](K: Int) extends Clustering[T] with
PipeOperator[XTSeries[Array[T]], EMOOutput] {
    override def |> (xt: XTSeries[Array[T]]): Option[EMOOutput] =
        override def execute(xt: XTSeries[Array[T]]): Unit = this |> xt
    match {
        case Some(emOutput) => ...
        case None => ...
    }
}

```

The `ClusteringModule` clustering module defines an abstract value, `clustering`, which is initialized at runtime (line 1). The two clustering algorithms, `KMeans` and `Expectation-Maximization`, `MultivariateEM`, inherits the `Clustering` base class. The clustering technique can be used in:

- A dependency-injection-based workflow by overriding `execute`
- A simpler data transformation flow by overriding `PipeOperator` (`|>`)

The clustering techniques are described in *Chapter 4, Unsupervised Learning*.

Dependency-injection-based workflow versus data transformation



The data transformation `PipeOperator` trades flexibility for simplicity. The design proposed for preprocessing and clustering techniques allows you to use both approaches. The techniques presented in the book implement the basic data transformation, `PipeOperator`, in order to keep the implementation of these techniques as simple as possible.

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 an N-Fold cross-validation scheme, and selecting the appropriate **labeled data**.

Validation

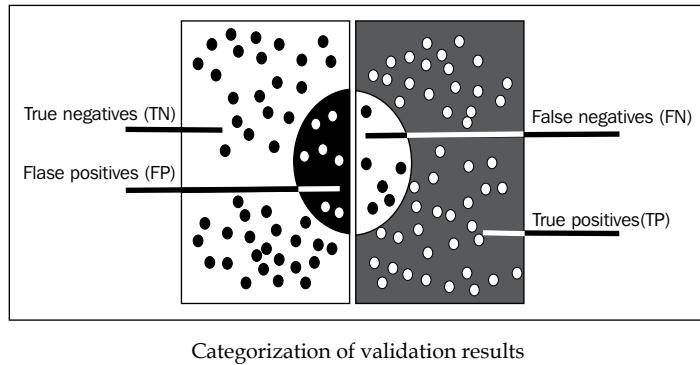
The purpose of this section is to create a Scala class to be used in future chapters for validating models. For starters, the validation process relies on a set of metrics to quantify the fitness of a model generated through training.

Key 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 terminology:

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

- **False negatives (FN)**: These are observations incorrectly labeled as belonging to the negative class (black 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:

- **Accuracy**: Represented as ac , this is the percentage of observations correctly classified.
- **Precision**: Represented as p , this is the percentage of observations correctly classified as positive in the group that the classifier has declared positive.
- **Recall**: Represented as r , this is the percentage of observations labeled as positive that are correctly classified.
- **F-Measure or F-score F1**: This is the score of a test's accuracy that 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).
- **G-measure**: Represented as G , this is similar to the F-measure but is computed as the geometric mean of precision p and recall r .

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

$$F1 = \frac{2pr}{p+r} \quad G = \sqrt{pr}$$

Implementation

Let's implement the validation formula using the same trait-based modular design used in creating the preprocessor and classifier modules. The `Validation` trait defines the signature for the validation of a classification model: the computation of the *F1* statistics and the precision-recall pair:

```
trait Validation {  
    def f1: Double  
    def precisionRecall: (Double, Double)  
}
```

Let's provide a default implementation of the `Validation` trait of the `F1Validation` class. 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 two parameters:

- The array of actual versus expected class: `actualExpected`
- The target class for true positive observations: `tpClass`

```
class F1Validation(actualExpected: Array[(Int, Int)], tpClass: Int) extends Validation {  
    val counts = actualExpected.foldLeft(new Counter[Label])((cnt,  
    oSeries) => cnt + classify(oSeries._1, oSeries._2))  
  
    lazy val accuracy = {  
        val num = counts(TP) + counts(TN)  
        num.toDouble / counts.foldLeft(0)( (s,kv) => s + kv._2)  
    }  
  
    lazy val precision = counts(TP).toDouble / (counts(TP) +  
    counts(FP))  
    lazy val recall = counts(TP).toDouble / (counts(TP) +  
    counters(FN))  
  
    override def f1: Double = 2.0 * precision * recall / (precision +  
    recall)  
    override def precisionRecall: (Double, Double) = (precision,  
    recall)  
  
    def classify(actual: Int, expected: Int): Label = {  
        if(actual == expected) { if(actual == tpClass) TP else TN }  
        else { if (actual == tpClass) FP else FN }  
    }  
}
```

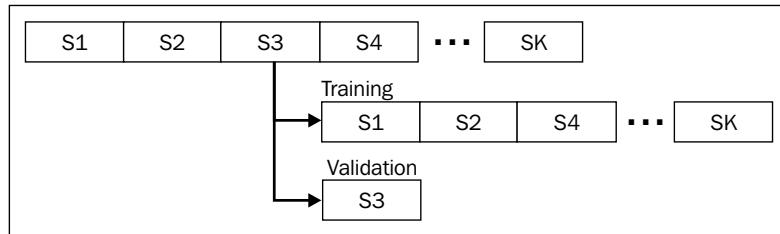
The precision and recall variables are defined as lazy so they are computed only once, when they are either accessed for the first time or the `f1` and `precisionRecall` functions are invoked. The class is independent of the selected machine learning algorithm, the training, the labeling process, and the type of observations.

Contrary to Java, which defines an enumerator as a class of types, Scala requires enumerators to be singletons that inherit the functionality of the `Enumeration` class:

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

K-fold cross-validation

It is quite common that the labeled dataset used for both training and validation is not large enough. The solution is to break the original labeled dataset into K data groups. The data scientist creates K training-validation datasets by selecting one of the groups as a validation set then combining all other remaining groups into a training set as illustrated in the next diagram. The process is known as the K-fold cross validation [2:7].



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

There is an obvious challenge in creating 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 he or she selected for training are representative to 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 squared error (MSE)** of the distribution are defined by the following formulas:


$$\left[\begin{array}{l} \text{Variance and bias for a true model, } \theta: \\ var(\hat{\theta}) = E[(\hat{\theta} - E[\hat{\theta}])^2] \quad bias(\hat{\theta}) = \hat{\theta} - \theta \quad (\hat{\theta}: \theta \text{ estimate}) \\ \\ \text{Mean square error:} \\ MSE = var(\hat{\theta}) + bias(\hat{\theta})^2 \end{array} \right]$$

Let's illustrate the concept of bias, variance, and mean square error with an example. At this stage, most of the machines learning techniques have not been introduced yet. Therefore, the example will emulate a multiple models *fEst: Double => Double* generated from non-overlapping training sets.

These models are evaluated against a test/validation datasets that are emulated by a model, *emul*. The *BiasVarianceEmulator* emulator class takes the emulator function and the size of the *nValues* validation test as parameters. It merely implements the formula to compute the bias and variance for each of the *fEst* models:

```
class BiasVarianceEmulator[T <% Double](emul: Double => Double,
nValues: Int) {

  def fit(fEst: List[Double => Double]): Option[XYTSeries] = {
    val rf = Range(0, fEst.size)
    val meanFEst = Array.tabulate(nValues)(x =>
```

```

rf.foldLeft(0.0)((s, n) => s+fEst(n)(x))/fEst.size) // 1

val r = Range(0, nValues)
Some(fEst.map(fe => {
  r.foldLeft(0.0, 0.0)((s, x) => {
    val diff = (fe(x) - meanFEst(x))/ fEst.size // 2
    (s._1 + diff*diff, s._2 + Math.abs(fe(x)-emul(x))) } )
  )).toArray
})
}
}

```

The `fit` method computes the variance and bias for each of the `fEst` models generated from training. First, the mean of all the models are computed (line 1), and then used in the computation of the variance and bias. The method returns a tuple (variance, bias) for each of the `fEst` model.

Let's apply the emulator to three nonlinear regression models evaluated against validation data:

$$y = \frac{x}{5}, y = 0.0003x^2 + 0.18x \text{ and } y = x \left(1 + \frac{\sin\left(\frac{x}{20}\right)}{5} \right)$$

The client code for the emulator consists of defining the `emul` emulator function, and a list, `fEst`, of three models defined as tuples of (`function, descriptor`) of type `(Double=>Double, String)`. The `fit` method is call on the model functions extracted through a map, as shown in the following code:

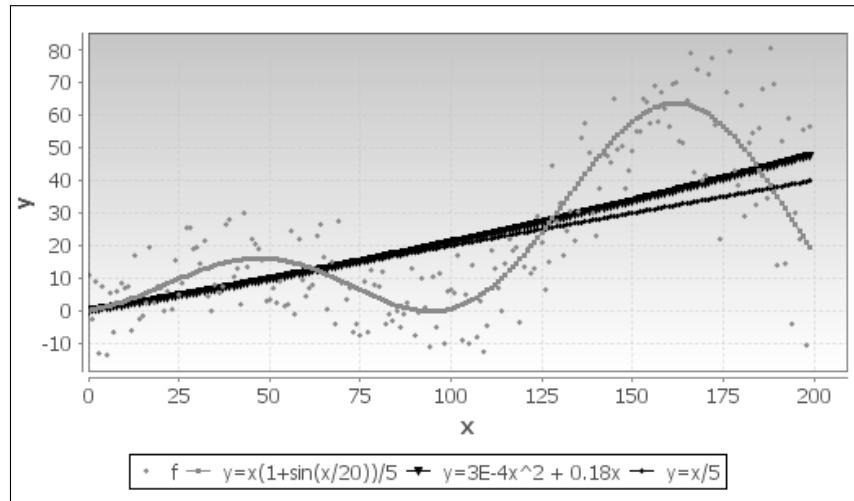
```

val emul = (x: Double) => 0.2*x*(1.0 + Math.sin(x*0.05))
val fEst = List[(Double=>Double, String)] (
  ((x: Double) => 0.2*x, "y=x/5"),
  ((x: Double) => 0.0003*x*x + 0.18*x, "y=3e-4.x^2-0.18x"),
  ((x: Double) =>0.2*x*(1+Math.sin(x*0.05)),
   "y=x(1+sin(x/20))/5"))
val emulator = new BiasVarianceEmulator[Double](emul, 200)
emulator.fit(fEst.map( _.1)) match {
  case Some(varBias) => show(varBias)
  case None => ...
}

```

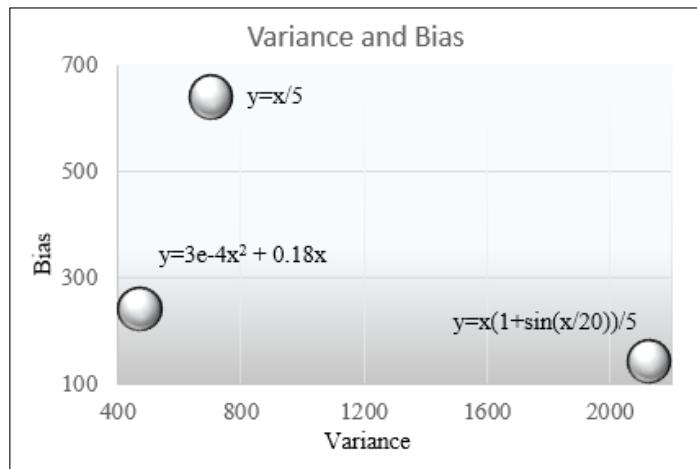
Hello World!

The JFreeChart library is used to display the test dataset and the three model functions.

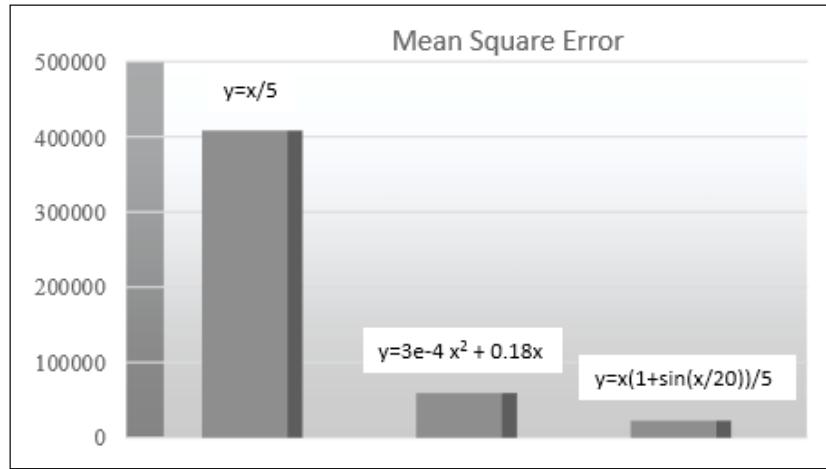


Fitting models to dataset

The variance-bias trade-off is illustrated in the following scatter chart using the absolute value of the bias:



The more complex the function, the lower the bias is. It is usually, but not always related to, a high variance. The most complex function $y=x(1+\sin(x/20))/5$ has by far the highest variance and the lowest bias. The more complex model matches fairly well with the training dataset. As expected, the mean square error reflects the ability of each of the three models to fit the test data.



Mean square error bar chart

The low bias of the complex model reflects in its ability to predict new observations correctly. Its MSE is therefore low, as expected.

Complex models with low bias and high variance are known as **overfitting**. Models with high bias and low variance are characterized as **underfitting**.

Overfitting

The methodology presented in the example can be applied to any classification and regression model. The list of models with low variance includes constant function and models independent of the training set. High degree polynomial, 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 is a sure sign of an overly complex model, which is difficult to debug and consumes computation resources
- It makes the model representing minor fluctuations and noise
- It may discover irrelevant relationships between observed and latent features
- It has 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 and input data through filtering
- Decreasing the number of features using techniques such as principal components analysis
- Modeling observable and latent noised using filtering techniques such as Kalman or autoregressive models
- Reducing inductive bias in a training set by applying cross-validation
- Penalizing extreme values for some of the model's features using regularization techniques

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 on 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 versatility and cleanliness of the Cake pattern in Scala as an effective scaffolding tool for data processing
- The concept of pipe operator for data conversion
- 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 penalizing outliers, modeling, and eliminating 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 unprocessed 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 model, and filter out noise. The purpose of cleansing raw data is twofold:

- Extract some basic knowledge from raw datasets
- 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.

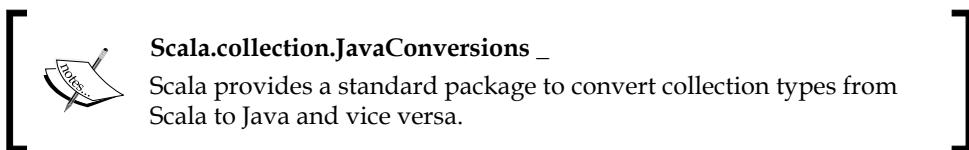
In this chapter, you will learn how to:

- 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 Kalman filter to extract the state of a dynamic system from incomplete and noisy observations

Time series

The overwhelming majority of examples used to illustrate the different machine algorithms in this book process time series or sequential, ordered, or unordered data.

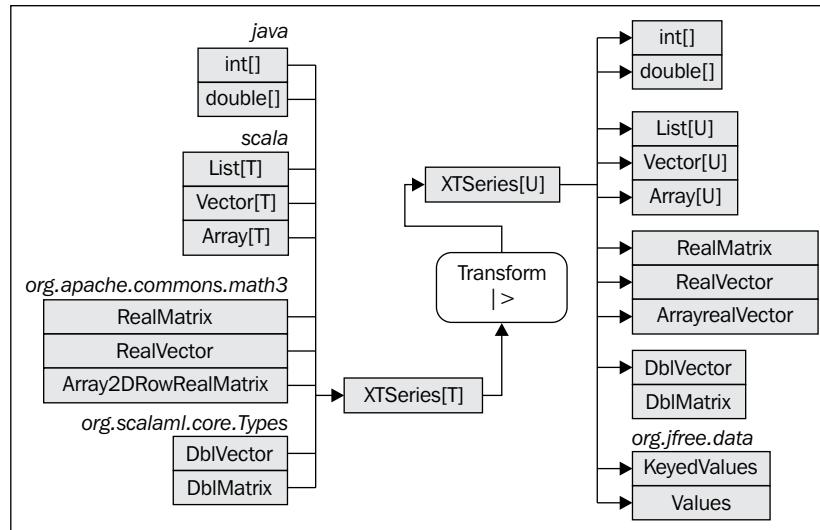
Each library has its own container type to manipulate datasets. The challenge is to define all possible conversions between types from different libraries needed to implement a large variety of machine learning models. Such a strategy may result in a combinatorial explosion of implicit conversion. A solution consists of creating a generic class to manage conversion from and to any type used by a third-party library.



The generic data transformation, `DT`, can be used to transform any `XTSeries` time series:

```
class DT[T,U] extends PipeOperator[XTSeries[T], XTSeries[U]] {  
    override def |> : PartialFunction[XTSeries[T], XTSeries[U]]  
}
```

Let's consider the simple case of using a Java library, the Apache Commons Math framework, and JFreeChart for visualization, and define a parameterized time series class, `XTSeries[T]`. The `\|>` data transformation converts a time series of values of type `T`, `XTSeries[T]`, into a time series of values of type `U`, `XTSeries[U]`. The following diagram provides an overview of type conversion in data transformation:



Let's create the `XTSeries` class. As a container, the class should be an implementation of the Scala higher-order collections functions such as `map`, `foreach`, or `zip`. The class should support at least conversion to `DblVector` and `DblMatrix` types introduced in the first chapter.

Here is a partial implementation of the `XTSeries` class. Comments, exceptions, argument validations, and debugging code are omitted in the code:

```
class XTSeries[T](label: String, arr: Array[T]) { // 1
    def apply(n: Int): T = arr.apply(n)

    @implicitNotFound("Undefined conversion to DblVector") // 2
    def toDblVector(implicit f: T=>Double): DblVector = arr.map(f(_))

    @implicitNotFound("Undefined conversion to DblMatrix") // 2
    def toDblMatrix(implicit fv: T => DblVector): DblMatrix = arr.map(
        fv(_)
    )

    def + (n: Int, t: T)(implicit f: (T,T) => T): T = f(arr(n), t)

    def head: T = arr.head //3
    def drop(n: Int): XTSeries[T] = XTSeries(label, arr.drop(n))
    def map[U: ClassTag](f: T => U): XTSeries[U] = XTSeries[U](label,
        arr.map(x => f(x)))
    def foreach( f: T => Unit) = arr.foreach(f) //3
    def sortWith(lt: (T,T)=>Boolean): XTSeries[T] = XTSeries[T](label,
        arr.sortWith(lt))
    def max(implicit cmp: Ordering[T]): T = arr.max //4
    def min(implicit cmp: Ordering[T]): T = arr.min
    ...
}
```

The class takes an optional label and an invariant array of the parameterized type `T`. The annotation `@specialized` (line 1) instructs the compiler to generate two versions of the class:

- A generic `XTSeries [T]` class that exploits all the implicit conversions required to perform operations on time series of a generic type
- An optimized `XTSeries [Double]` class that bypasses the conversion and offers the client code with a faster implementation

The conversion to `DblVector` (resp. `DblMatrix`) relies on the implicit conversion of elements to type `Double` (resp. `DblVector`) (line 2). The `@implicitNotFound` annotation instructs the compiler to omit an error if no implicit conversion is detected. The conversion methods are used to implement the implicit conversion introduced in the previous section. These methods are defined in the singleton `org.scalaml.core.Types.CommonsMath` library. The following code shows the implementation of the conversion methods:

```
object Types {
    object CommonMath {
        implicit def series2DblVector[T] (xt: XTSeries[T]) (implicit f:
T=>Double) : DblVector = xt.toDblVector(f)
        implicit def series2DblMatrix[T] (xt: XTSeries[T]) (implicit f:
T=>DblVector) : DblMatrix = xt.toDblMatrix(f)
        ...
    }
}
```

This code snippet exposes a subset of the Scala higher-order collections methods (line 3) applied to the time series. The computation of the minimum and maximum values in the time series required that the `cmp` ordering/compare method be defined for the elements of the type `T` (line 4).

Let's put our versatile `XTSeries` class to use in creating a basic preprocessing data transformation starting with the ubiquitous moving average techniques.

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 the technical analysis of financial markets to define a dynamic level of support and resistance for the price of a given security.

Let's consider a time series $x_t = x(t)$ and a function $f(x_{t-p}, x_{t-1})$ that reduces the last p observations into a value or average. The prediction or estimation of the observation at $t+1$ is defined by the following formula:

$$\tilde{x}_{t+1} = f(x_{t-p}, \dots, x_t)$$

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

The simple moving average

Simple moving average, a smoothing method, 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 of a time series $\{x_t\}$ with a period p is computed as the average of the last p observations:

$$\tilde{x}_t = \frac{1}{p} \sum_{j=t-p}^t x_j$$



The computation is implemented iteratively using the following formula (1):

$$\tilde{x}_t = \tilde{x}_{t-1} + \frac{x_t - x_{t-p}}{p} \quad \forall t \geq p; 0 \leq t \leq p$$

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 abstract parameterized class `MovingAverage[T <% Double]` as its root. We use the generic time series class, `XTSeries[T]`, introduced in the first section and the generic pipe operator, `|>`, introduced in the previous chapter:

```
abstract class MovingAverage[T <% Double] extends
  PipeOperator[XTSeries[T], XTSeries[Double]]
```

The pipe operator for the `SimpleMovingAverage` class implements the iterative formula (1) for the computation of the simple moving average. The `override` keyword is omitted:

```
class SimpleMovingAverage[@specialized(Double) T <% Double] (val
  period: Int) (implicit num: Numeric[T]) extends MovingAverage[T] {

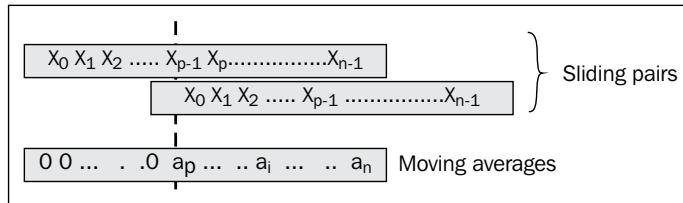
  def |> : PartialFunction[XTSeries[T], XTSeries[Double]] = {
    case xt: XTSeries[T] if(xt != null && xt.size > 0) => {
      val slider = xt.take(data.size-period)
        .zip(data.drop(period)) //1
      val a0 = xt.take(period).toArray.sum/period //2
      var a: Double = a0
      val z = Array[Array[Double]](
```

```
    Array.fill(period) (0.0), a, slider.map(x => {
      a += (x._2 - x._1)/period
      a})
    ).flatten //3
    XTSeries[Double] (z)
}
```

The class is parameterized for the type of elements of the input time series. After all, we do not have control over the source of the input data. The type for the elements of the output time series is Double.

The class has a type T and is specialized for the Double type for faster processing. The implicitly defined num: Numeric[T] is required by the arithmetic operators sum and / (line 2).

The implementation has a few interesting elements. First, the set of observations is duplicated and the index in the clone is shifted by p observations before being zipped with the original to the array of a pair of values: slider (line 1):



The sliding algorithm to compute moving averages

The average value is initialized with the average of the first p data points. The first p values of the trends are initialized as an array of p zero values. It is concatenated with the first average value and the array containing the remaining average values. Finally, the array of three arrays is flattened (flatten) into a single array containing the average values (line 3).

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 a_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 of a series $\{x_j\}$ with a period p and a normalized weights distribution $\{a_j\}$ is given by the following formula (2):



$$\tilde{x}_t = \frac{1}{p} \sum_{j=t-p}^t a_{j-p} x_j; \sum_{j=0}^{p-1} a_j = 1$$

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 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 <% Double] (val
  weights: DblVector) extends MovingAverage[T] {
  def |> : PartialFunction[XTSeries[T], XTSeries[Double]] = {
    case xt: XTSeries[T] if (xt != null && xt.size > 1) => {
      val smoothed = Range(weights.size, xt.size).map(i => {
        xt.toArray.slice(i - weights.size, i)
          .zip(weights)
          .foldLeft(0.0)((s, x) => s + x._1 * x._2) }) //1
      XTSeries[Double](Array.fill(weights.size)(0.0) ++ smoothed) //2
    }
  }
}
```

As with the simple moving average, the array of the initial p moving average with the value 0 is concatenated (line 2) with the first moving average value and the remaining weighted moving average computed using a map (line 1). The period for the weighted moving average is implicitly defined as `weights.size`.

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 on a series $\{x_t\}$ and a smoothing factor α is computed by the following iterative formula:



$$\tilde{x}_t = (1 - \alpha) \tilde{x}_{t-1} + \alpha x_t \quad \forall t > 0; x_0 \text{ if } t = 0$$

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

The implementation of the `ExpMovingAverage` class is rather simple. There are two constructors, one for a user-defined smoothing factor and one for the Nyquist period, p , used to compute the smoothing factor $\alpha = 2/(p+1)$:

```
class ExpMovingAverage[@specialized(Double) T <% Double] (val alpha: Double) extends MovingAverage[T] {
    def |> : PartialFunction[XTSeries[T], XTSeries[Double]] = {
        case xt: XTSeries[T] if(xt != null && xt.size > 1) => {
            val alpha_1 = 1-alpha
            var y: Double = data(0)
            xt.map( x => {
                val z = x*alpha + y*alpha_1; y=z; z })
        }
    }
}
```

The version of the constructor that uses the Nyquist period p is implemented using the Scala `apply` method:

```
def apply[T <% Double] (nyquist: Int): ExpMovingAverage[T] = new
    ExpMovingAverage[T] (2/( nyquist + 1))
```

Let's compare the results generated from these three moving averages methods with the original price. We use a data source (with respect to sink), `DataSource` (with respect to `DataSink`) to load the historical daily closing stock price of **Bank of America (BAC)**. The `DataSource` and `DataSink` classes are defined in the *Data extraction* section in *Appendix A, Basic Concepts*. The comparison of results can be done using the following code:

```
val p_2 = p >>1
val w = Array.tabulate(p) (n =>if(n==p_2) 1.0 else 1.0/(Math.
abs(n-p_2)+1)) //1
val weights = w map { _ / w.sum } //2

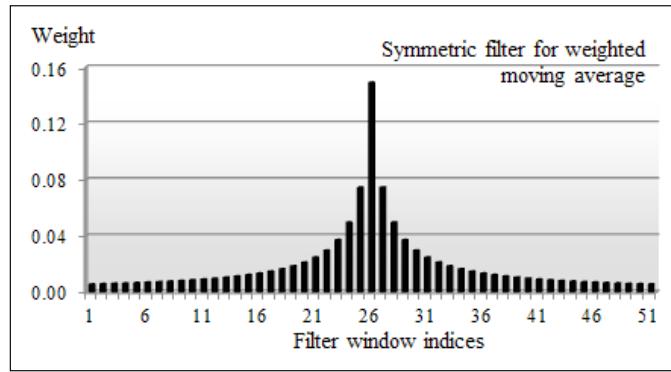
val src = DataSource("resources/data/chap3/BAC.csv, false)//3

val price = src |> YahooFinancials.adjClose //4
val sMvAve = SimpleMovingAverage(p)
val wMvAve = WeightedMovingAverage(weights)
val eMvAve = ExpMovingAverage(p)

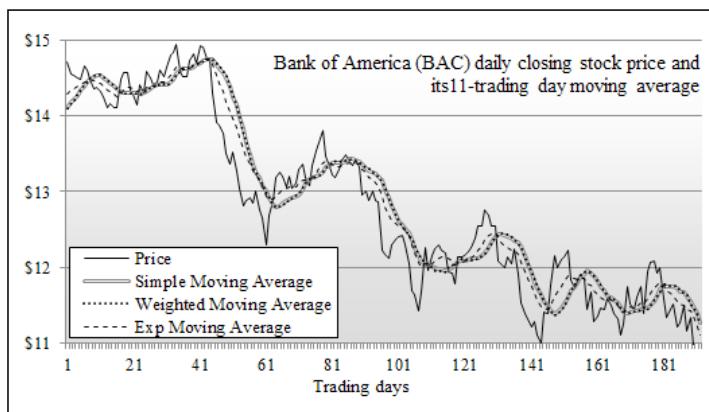
val results = price :: sMvAve.|>(price) :: wMvAve.|>(price) :::
eMvAve.|>(price) :: List[XTSeries[Double]]() //5
Val outFile = "output/chap3/mvaverage" + p.toString + ".csv"
DataSink[Double](outFile) |> results //6
```

The coefficients for the weighted moving average are generated (line 1) and normalized (line 2). The trading data regarding the ticker symbol, BAC, is extracted from the Yahoo! finances CSV file (line 3), `YahooFinancials`, using the `adjClose` extractor (line 4). The smoothed data generated by each of the moving average techniques are concatenated into a list of time series (line 5). Finally, the content is formatted and dumped into a file, `outFile`, using a `DataSink` instance (line 6).

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 generated if a specific moving average cannot be computed. The following graph is an example of a symmetric filter for weighted moving averages:

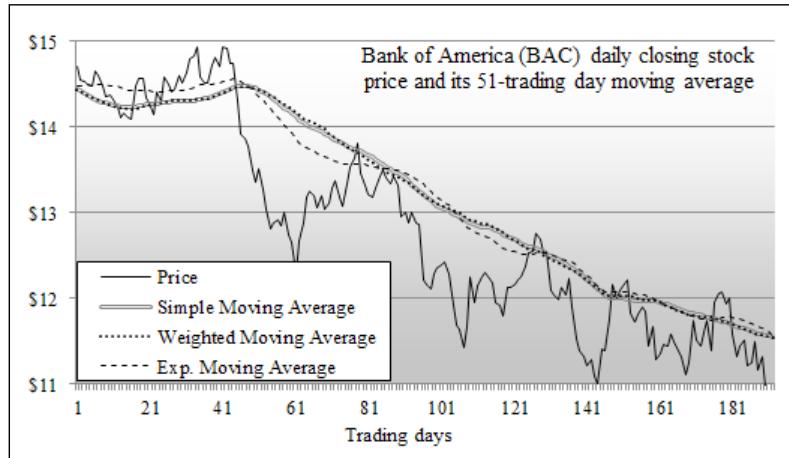


The three moving average techniques are applied to the price of the stock of Bank of America (BAC) over 200 trading days. Both the simple and weighted moving average uses a period of 11 trading days. The exponential moving average method uses a scaling factor of $2/(11+1) = 0.1667$.



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 or two calendar months, the simple and weighted moving averages generate a smoothed time series compared to the exponential moving average with a smoothing factor of $2/(p+1) = 0.038$.



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-band filter that allows only lower frequencies. Fine-tuning the period or smoothing factor is time consuming. Spectral analysis, or more specifically, Fourier analysis, transforms the time series into a sequence of frequencies, which is a time series in the frequency domain.

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 spectral density is estimated 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 time series should not be confused with spectral theory, a subset of linear algebra that studies Eigenfunctions on Hilbert and Banach spaces. Harmonic and Fourier analyses are regarded as a subset of spectral theory.

The **fast Fourier transform (FFT)** is the most commonly used frequency analysis algorithm [3:5]. Let's explore the concept behind the discrete Fourier series and the Fourier transform as well as their benefits as applied to financial markets. The Fourier analysis approximates any generic function as the sum of trigonometric functions, sine and cosine. The decomposition in a basic trigonometric function is known as a **Fourier transform** [3:6].

Discrete Fourier transform (DFT)

A time series $\{x_k\}$ 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 could 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 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 $\omega = a + j \cdot \varphi$ ($j_2 = -1$), for which 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

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

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

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

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

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


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

The Fourier sine transform of a function is defined as:

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

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

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

The computation of the Fourier trigonometric series is time consuming with an asymptotic time complexity of $O(n^2)$. Several attempts have been made to make the computation as effective as possible. The most common numerical algorithm used to compute the Fourier series is the fast Fourier transform created by J. W. Cooley and J. Tukey [3:7]. The algorithm, called Radix-2, 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 deepest-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 $N=2n$ for even functions (sine) and $N = 2n+1$ for cosine. There are two approaches to meet this constraint:

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

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

Let's define a base class, `DTransform[T]`, for all the fast Fourier transforms, parameterized with a view bounded to the `Double` type (`Double`, `Float`, and so on). The first step is to implement the padding method, common to all the Fourier transforms:

```
trait DTransform[T] extends PipeOperator[XTSeries[T], XTSeries[Double]] {
    def padSize(xtSz: Int, even: Boolean=true): Int = {
        val sz = if( even ) xtSz else xtSz-1
        if( (sz & (sz-1)) == 0 ) 0
        else {
            var bitPos = 0
            do {
                bitPos += 1
            } while( (sz >> bitPos) > 0 )
            (if(even) (1<<bitPos) else (1<<bitPos)+1) - xtSz
        }
    }

    def pad(xt: XTSeries[T], even: Boolean=true)
        (implicit f: T => Double): DblVector = {
        val newSize = padSize(xt.size, even)
        val arr: DblVector = xt
        if( newSize > 0) arr ++ Array.fill(newSize)(0.0) else arr
    }
}
```

The while loop

Scala developers prefer Scala higher-order methods on collection to implement iterative computation. However, nothing prevents you from using a traditional while loop if either readability or performance is an issue.

The fast implementation of the padding method, `pad`, consists of detecting the number of observations, N , which is a power of 2 using the bit operator `&` by evaluating whether $N \& (N-1)$ is null. The next highest radix is extracted by computing the number of bits shift in N . The code illustrates the effective use of implicit conversion to make the code readable. The `arr: DblVector = series` conversion triggers a conversion defined in the `XTSeries` companion object.

The next step is to write the `DFT` class for the real discrete transforms, sine and cosine, by subclassing `DTransform`. The purpose of the class is to select the appropriate Fourier series, pad the time series to the next power of 2 if necessary, and invoke the `FastSineTransformer` and `FastCosineTransformer` classes of the Apache Commons Math library [3:8] introduced in the first chapter:

```
class DFT[@specialized(Double) T<%Double] extends DTransform[T] {
    def |> : PartialFunction[XTSeries[T], XTSeries[Double]] = {
        case xt: XTSeries[T] if(xt != null && xt.length > 0) =>
            XTSeries[Double](fwd(xt)._2)
    }
    def fwd(xt: XTSeries[T]): (RealTransformer, DblVector) = {
        val rdt = if(Math.abs(xt.head) < DFT_EPS)
            new FastSineTransformer(DstNormalization.STANDARD_DST_I)
        else new FastCosineTransformer(DctNormalization.STANDARD_DCT_I)

        (rdt, rdt.transform( pad(xt, xt.head==0.0), TransformType.FORWARD ))
    }
}
```

The discrete Fourier sine series requires that the first value of the time series is `0.0`. This implementation automates the selection of the appropriate series by evaluating `series.head`. This example uses the standard formulation of the cosine and sine transformation, defined by the `DctNormalization.STANDARD_DCT_I` argument. The orthogonal normalization, which 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.

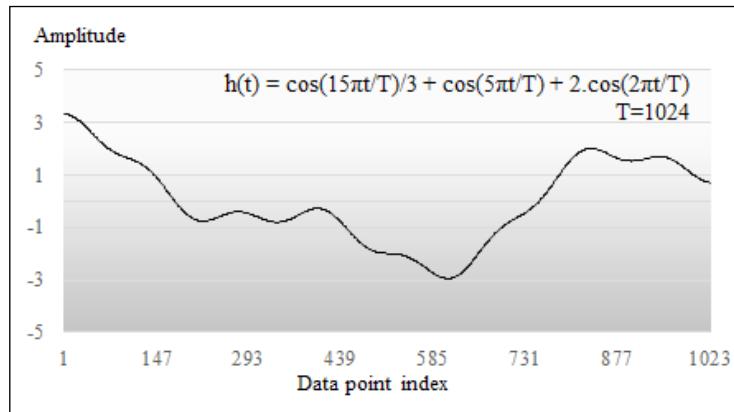
@specialized

The `@specialized(Double)` annotation is used to instruct the Scala compiler to generate a specialized and more efficient version of the class for the type `Double`. The drawback of 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 sequence h of sinusoidal functions:

```
val _T= 1.0/1024
val h = (x:Double) =>2.0*Math.cos(2.0*Math.PI*_T*x) +
    Math.cos(5.0*Math.PI*_T*x) + Math.cos(15.0*Math.PI*_T*x)/3
```

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 $2n+1$. The data generated by the function h is plotted as follows:



Example of the sinusoidal time series

Let's extract the frequencies spectrum for the time series generated by the function h . The data points are created by tabulating the function h . The frequencies spectrum is computed with a simple invocation of the pipe operator on the instance of the DFT class:

```
val rawOut = "output/chap3/raw.csv"
val smoothedOut = "output/chap3/smoothed.csv"
val values = Array.tabulate(1025)(x =>h(x/1025))
DataSink[Double](rawOut) |> values //1

val smoothed = DFT[Double] |> XTSeries[Double](values) //2
DataSink[Double]("output/chap3/smoothed.csv") |> smoothed
```

The first data sink (the type `Datasink`) stores the original time series into a CSV file (line 1). The `DFT` instance extracts the frequencies spectrum and formats it as time series (line 2). Finally, a second sink saves it into another CSV file.

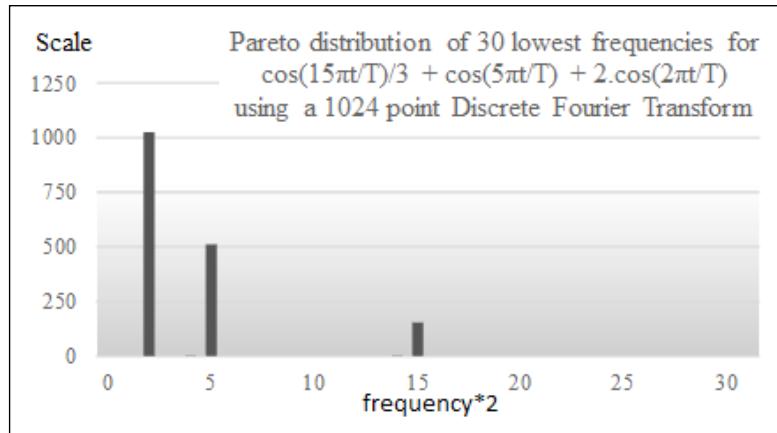
Data sinks and spreadsheets

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 the time series, plotted for the first 32 points, clearly shows three frequencies at $k=2, 5, \text{ and } 15$. This 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:



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 to implement a low or pass band filter in the time domain from autoregressive models to the Butterworth algorithm. However, the fast Fourier transform is still a very popular technique to smooth signals and extract 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 transforms are used sequentially to convert the time series 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

The convolution of two functions f and g is defined as:

$$\langle f, g \rangle = \int_{-\infty}^{\infty} f(t) \cdot g(x-t) dt$$

DFT convolution

One important property of the Fourier transform is that convolution of two signals is implemented as the inner product of their relative spectrums:

$$F(f * g) = F(f)F(g)$$

Let's apply the property to the discrete Fourier transform. If a time series $\{x_i\}$ has a frequency spectrum $\{\omega_f\}$ and a filter f in a frequency domain defined as $\{\omega_g\}$, then the convolution is defined as:

$$F(f * g) = \sum_0^{N-1} \omega_{x,j} \omega_{f,k-j}$$

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.
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.

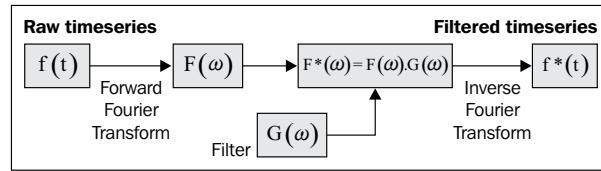


Diagram of a discrete Fourier filter

The most commonly used low-pass filters are known as the `sinc` and `sinc2` functions, defined as a rectangular function and a triangular function, respectively. The simplest low-pass filter is implemented by a `sinc` function that returns 1 for frequencies below a cutoff frequency, `fc`, and 0 if the frequency is higher:

```

def sinc(f: Double, fc: Double): Double = if(Math.abs(f) < fc) 1.0
else 0.0
def sinc2(f: Double, fc: Double): Double = if(f*f < fc) 1.0 else 0.0
  
```

The filtering computation is implemented as a data transformation (pipe operator `|>`). The `DFTFir` class inherits from the `DFT` class in order to reuse the `fwrd` forward transform function. As usual, exception and validation code is omitted. The frequency domain function g is an attribute of the filter. The `g` function takes the frequency cutoff value `fc` as the second argument. The two filters `sinc` and `sinc2` defined in the previous section are examples of filtering functions.

```

class DFTFir[T <% Double] (val g: (Double, Double) => Double, val fc: Double) extends DFT[T]
  
```

The pipe operator implements the filtering functionality:

```

def |> : PartialFunction[XTSeries[T], XTSeries[Double]] = {
  case xt: XTSeries[T] if(xt != null && xt.size > 2) => {
    val spectrum = fwrd(xt) //1
    val cutOff = fc*spectrum._2.size
    
```

```

val filtered = spectrum._2.zipWithIndex.map(x => x._1*g(x._2,
    cutOff)) //2
XTSeries[Double] (spectrum._1.transform(filtered, TransformType.
INVERSE)) //3
}

```

The filtering process follows three steps:

1. Computation of the discrete Fourier forward transformation (sine or cosine), `fwrD`.
2. Apply the filter function through a Scala `map` method.
3. Apply the inverse transform on the frequencies.

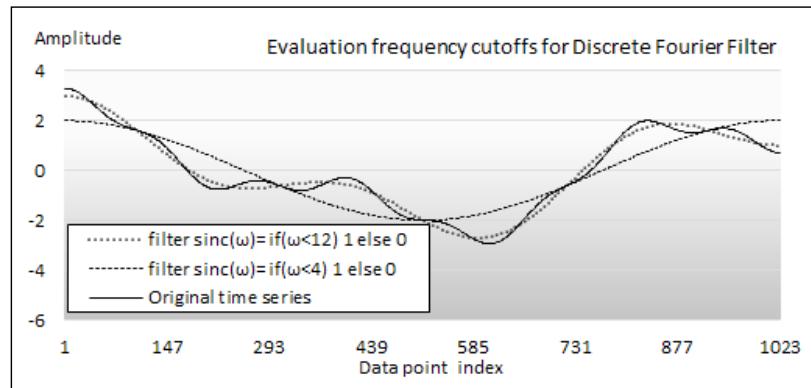
Let's evaluate the impact of the cutoff values on the filtered data. The implementation of the test program consists of invoking the DFT filter pipe operator and writing results into a CSV file. The code reuses the generation function `h` introduced in the previous paragraph:

```

val price = src |> YahooFinancials.adjClose
val filter = new DFTFir[Double] (sinc, 4.0)
val filteredPrice = filter |> price

```

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:



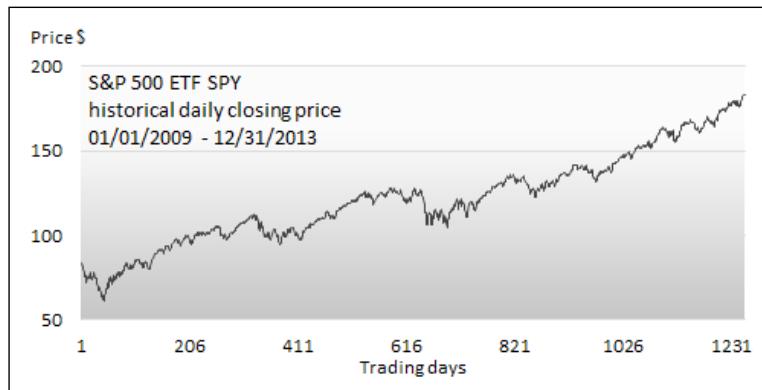
Plotting of the discrete Fourier filter-based smoothing

As you would expect, the low-pass filter with a cutoff value of 12 removes 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

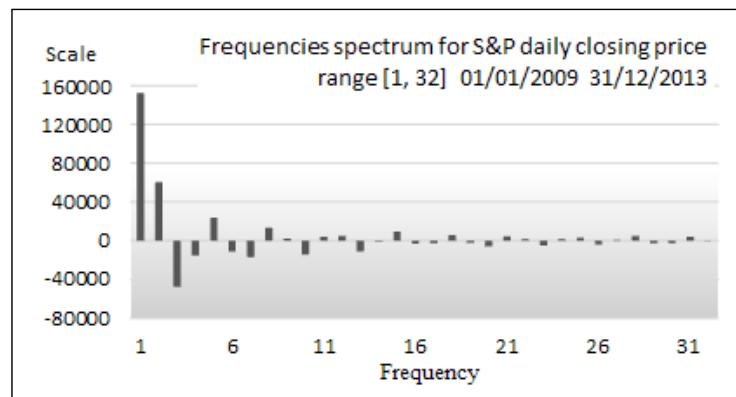
Using the discrete Fourier transform to generate the frequencies spectrum of a periodical time series is easy. However, what about real-world signals such as the time series representing the historical price of a stock?

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:



Historical S&P 500 index prices

The first step is to apply the DFT to extract a spectrum for the S&P 500 historical prices, as shown in the following graph, with the first 32 harmonics:



Frequencies spectrum for historical S&P index

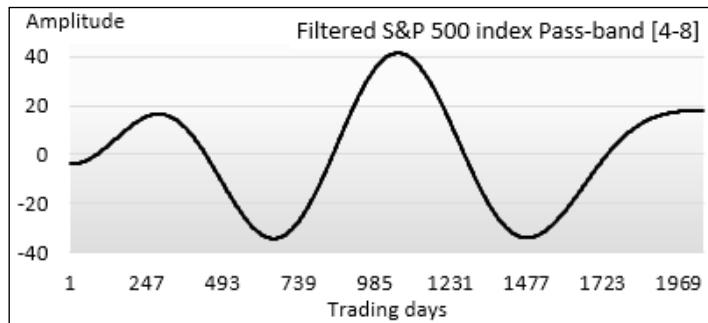
The frequency domain chart highlights some interesting characteristics regarding the S&P 500 historical prices:

- 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. The next step is to apply a pass-band 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 passband 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 introduced in the previous section to implement a low-band filter is modified to enforce the passband within a window, $[w_1, w_2]$, as follows:

```
def sinc(f: Double, w: (Double, Double)): Double = if (Math.abs(f) >
w._1 && Math.abs(f) < w._2) 1.0 else 0.0
```

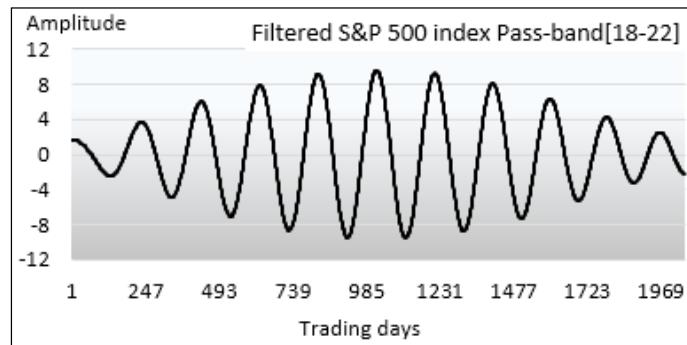
Let's define a DFT-based pass-band filter with a window of width 4, $w=(i, i+4)$, with i ranging between 2 and 20. Applying the window [4, 8] isolates the impact of the second harmonic on the price curve. As we eliminate the main upward trend with frequencies less than 4, all filtered data varies within a short range relative to the main trend. The following graph shows output of this filter:



The output of a pass-band DFT filter range 4-8 on the historical S&P index

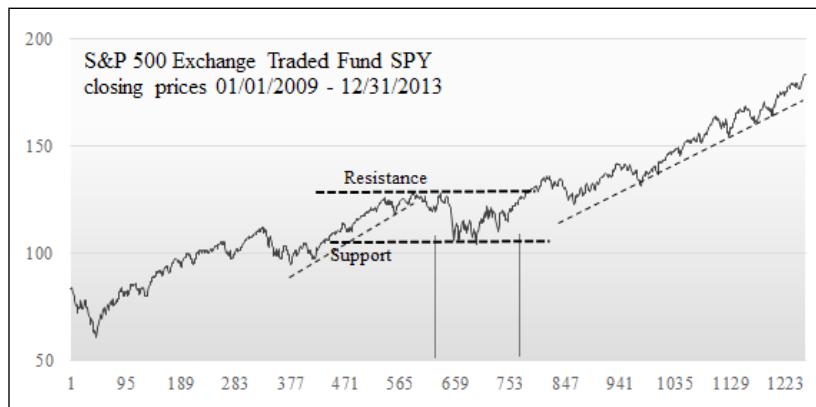
Data Preprocessing

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 pass-band 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 passband with a frequency of 20, as illustrated in the previous plot; 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 and low frequency, as shown in the following graph:



One of the limitations of using the Fourier transform 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 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 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 (HMM)** described in *Chapter 6, Regression and Regularization* [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 smoother that computes 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 model nonlinear-state space 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:

- **Transition equation:** This describes the dynamics of the system including the unobserved variables
- **Measurement equation:** This describes the relationship between the observed and unobserved variables

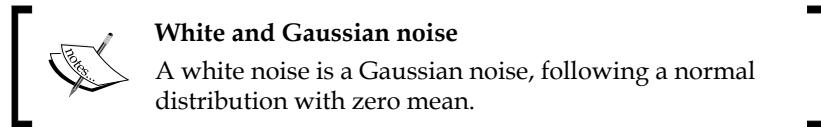
The transition equation

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:

$$x_t = A_t \cdot x_{t-1} + B_t \cdot u_t + w_t$$

- A is the square matrix of dimension n that represents the transition from state x at $t-1$ to state x at t . The matrix is intrinsic to the dynamic system under consideration.
- B is an n by n matrix that describes the control input model (external action on the system or model). It is applied to the control vector u .
- w represents the noise generated by the system or from a probabilistic point of view, 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.



The measurement equation

The measurement of m values z_t of the state of the system is defined by the following equation:

$$z_t = H_t \cdot x_t + v_t$$

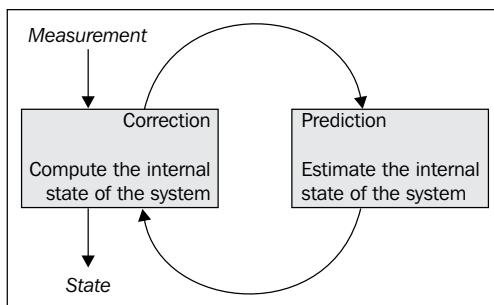
- H is a matrix m by n that models the dependency of the measurement to the state of the system.
- v is the white noise introduced by the measuring devices. Similar to the process noise, v follows a Gaussian distribution with zero mean and a variance R , known as the measurement noise covariance.

The recursive algorithm

The set of equations for the discrete Kalman filter are implemented as 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:



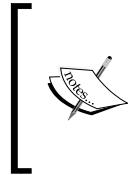
An overview diagram of the recursive Kalman algorithm

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 xt and the previous yield $xt-1$.



State in dynamic systems

The term "state" refers to the state of the dynamic system under consideration. This is a different term for observation, data, or value vector. A state or observation is a set of values, one for each variable of the model.

This implementation of the Kalman filter uses the Apache Commons Math library, which defines and manipulates specific types. The first step is to define the implicit type conversion required to interface with the `KalmanFilter` class:

```
type DblMatrix = Array[Array[Double]]  
type DblVector = Array[Double]  
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 implicit type conversion has to be defined in the scope of the client code.

The Kalman model assumes that process and measurement noise follow a Gaussian distribution, also known as white noise. For the sake of maintainability, the generation or simulation of the white noise is encapsulated in the `QRNoise` class with `qr` as the tuple of scale factors for the process noise matrix `Q` and the measurement noise `R`. The two create methods execute the user-defined noise function `white`:

```
class QRNoise(qr: XY, white: Double=> Double) {
    def q = white(qr._1)
    def r = white(qr._2)
    def noisyQ = Array[Double](q,q)
    def noisyR = Array[Double](r,r)
}
```

The easiest approach to manage the matrices and vectors used in the recursion is to define them as parameters of the main class, `DKalman`:

```
class DKalman(A:DblMatrix, B:DblMatrix, H:DblMatrix, P:DblMatrix)
(implicit val qrNoise: QRNoise) extends PipeOperator[XY,XY] {

    val Q = new DblMatrix(A.size).map(_ => Array.fill(A.size)(qrNoise.
qr.1))

    var x: RealVector =
    var filter: KalmanFilter =
}
```

The matrix used in the prediction and correction phase is defined as an argument of the `DKalman` class. The matrices for the covariance of the process noise `Q` and the measurement noise `R` are also initialized during the instantiation of the Kalman filter class. 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 bond) using the transition equation. We assume that the Federal Reserve has no material effect on the interest rates, making control input matrix B null. The transition equation can be easily resolved using simple operations on matrices.

$$\begin{bmatrix} \hat{x}_t \\ \hat{x}_{t-1} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} \hat{x}_t \\ \hat{x}_{t-1} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} u_t \\ u_{t-1} \end{bmatrix} + \begin{bmatrix} w_t \\ w_{t-1} \end{bmatrix}$$

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. The matrix B would 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 reference to its 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 note describes the two equations for the prediction step.

The prediction step

The prediction of the state at time $t+1$ 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

The mean square error matrix P , which is to be minimized, is updated through 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 `RealMatrix` and `RealVector` classes. The implementation of the equation is as follows:

```
x = A.operate(x).add(qrNoise.noisyQ)
```

The new state is predicted (or estimated), and then used as an input to the correction step.

Correction

The second and last 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 , computing the Kalman gain K , and estimating the matrix of the error covariance P .

Correction step

The state of the system x is estimated from the actual measurement z through the following formula:

$$\hat{x}_t = \hat{x}'_t + K_t (z_t - H_t \cdot \hat{x}'_t) r_t = z_t - H_t \cdot \hat{x}'_t$$

- r is the residual between the predicted measurement and the actual measured values
- K is the Kalman gain for the correction factor Kr



The Kalman gain is computed using the estimated error covariance matrix P'_t :

$$K_t = P'_t \cdot H_t^T (H_t \cdot P'_t \cdot H_t^T + R_t)^{-1}$$

- H^T is the matrix transpose of H

Finally, the estimate of the error covariance matrix P'_t is corrected to the value P_t through the following formula:

$$P'_t = (I_d - K_t \cdot H_t) \cdot P'_t$$

- I_d is the identity matrix.

Kalman smoothing

It is time to put our knowledge of the transition and measurement equations to the test. The Apache Commons Library defines two classes, `DefaultProcessModel` and `DefaultMeasurementModel`, to encapsulate the components of the matrices and vectors. The historical values for the yield of the 10-year Treasury bond is loaded through the `DataSource` method and mapped to the smoothed series that is the output of the filter.

```
def |> : PartialFunction[XTSeries[XY], XTSeries[XY]] = {
  case xt: XTSeries[XY] if(xt.size > 0) => xt.map( y => {
    initialize(Array[Double](y._1, y._2)) //1
    val nState = newState //2
    (nState(0), nState(1)) //3
  })
  ...
}
```

The data transformation for the Kalman filter initializes the process and measurement model for each data point (line 1), updates the state using the transition and correction equations iteratively (line 2), and returns the filtered series (line 3).



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 model is a **2-step lag smoothing** algorithm using a single smoothing factor α with a state, S_t :

$$S_t = \{x_{t+1}, x_t\} \text{ with } x_{t+1} = \alpha \cdot x_t + (1 - \alpha) \cdot x_{t-1} \text{ and } x_t = x_t$$

Following the Scala standard to return errors to the client code, the exceptions thrown by the Commons Math API are caught and processed through the `Option` monad. The iterative prediction and correction of the smoothed yields is implemented by the `newState` method. The method iterates through four steps:

1. Filter an estimate of the state x at time t .
2. The new state is computed using the transition equation.
3. The measured value z of the state is computed using the measurement equation.
4. The original estimate x is corrected with the measured value.

The `newState` method is defined as follows:

```
val PROCESS_NOISE_Q = 0.03
val PROCESS_NOISE_R = 0.1
val MEASUREMENT_NOISE = 0.4

def newState: DblVector = {
    Range(0, maxIters) foreach( _ => {
        filter.predict //1
        val w = qrNoise.create(PROCESS_NOISE_Q, PROCESS_NOISE_R)
        x = A.operate(x).add(qrNoise.noisyQ) //2
        val v = qrNoise.create(MEASUREMENT_NOISE)
        val z = H.operate(x).add(qrNoise.noisyR) //3
        filter.correct(z) // 4
    })
    filter.getStateEstimation
}
```

The `PROCESS_NOISE` factor (with respect to `MEASUREMENT_NOISE`) used in the creation of the process noise `w` and measurement noise `v` are somewhat arbitrary. Their purpose is to simulate the white noise for the model. The `newState` method returns the filtered state as a `DblVector` instance for this particular state.



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.

Experimentation

The objective is to smoothen the yield of the 10-year Treasury bond and quantify the impact of the elements of the state-transition matrix `A` on the smoothing process. 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 represents the yield at time t . This is accomplished by shifting the values of the original time series $\{x_0, \dots, x_n\}$ by 1 using the `drop` method, $X1=\{x_1, \dots, x_n\}$, creating a copy of the original time series without the last element $X2=\{x_0, \dots, x_{n-1}\}$ and zipping $X1$ and $X2$. The resulting sequence of pair $\{(x_k, x_{k-1})\}$ is processed by the Kalman algorithm, as shown in the following code:

```
implicit val qrNoise = QRNoise((0.2, 0.4), (m: Double) => m* (new
Random(System.currentTimeMillis)).nextGaussian) //1
val A: DblMatrix = ((0.9, 0.0), (0.0, 0.1))
```

```
val B: DblMatrix = (0.0, 0.0)
val H: DblMatrix = (1.0, 1.0)
val P0: DblMatrix = ((0.4, 0.5), (0.4, 0.5))
val x0: DblVector = (175.0, 175.0)

val dKalman = new DKalman(A, B, H, P0) //2
val output = "output/chap3/kalman.csv"
val zt_1 = zSeries.drop(1)
val zt = zSeries.take(zSeries.size-1)
val filtered = dKalman |> XTSeries[(Double, Double)](zt_1.zip(zt)) //3
DataSink[Double](output) |> filtered.map(_.1) //4
```

The process and measurement noise `qrNoise` is implicitly initialized with the respective factors, 0.2 and 0.4 (line 1). The Kalman filter is initialized with the prediction-correction equation matrices A , B , H , and $P0$, and the initial state x_0 (line 2). A time series $\{(x_i, x_{i+p})\}_i$ is generated by zipping two copies of the historical 10 Treasury bond yield series, with the second one being shifted by p data. The Kalman filter is applied to the time series of tuples and the result is dumped into an output file using a `DataSink` instance (line 4)

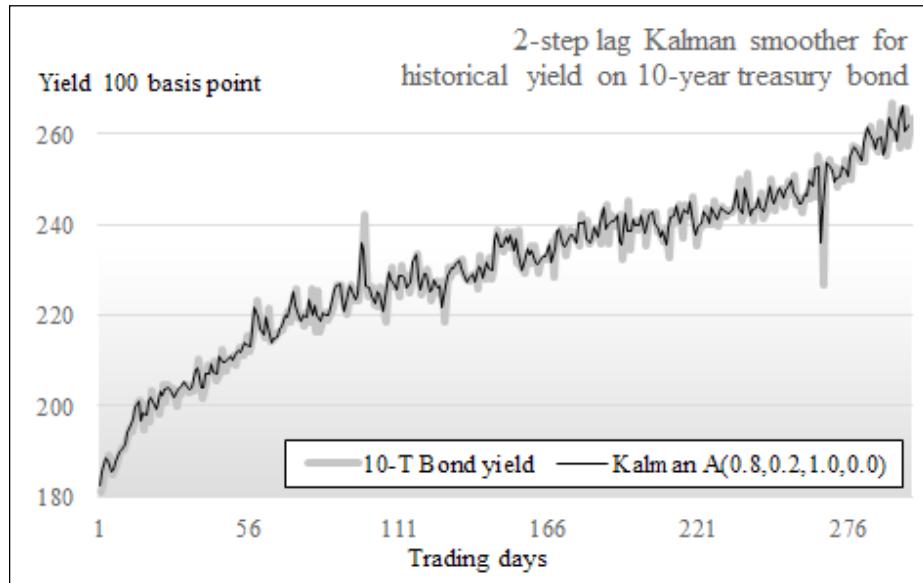
The test is performed over a period of one year, and the results are plotted using a basis point or 100th of a percentage. The quality of the output is evaluated using two different values for the state transition matrix A : [0, 8, 0.2, 1.0, 0.0] and [0.5, 0.5, 1.0, 0.0].



Modeling state transition and noise

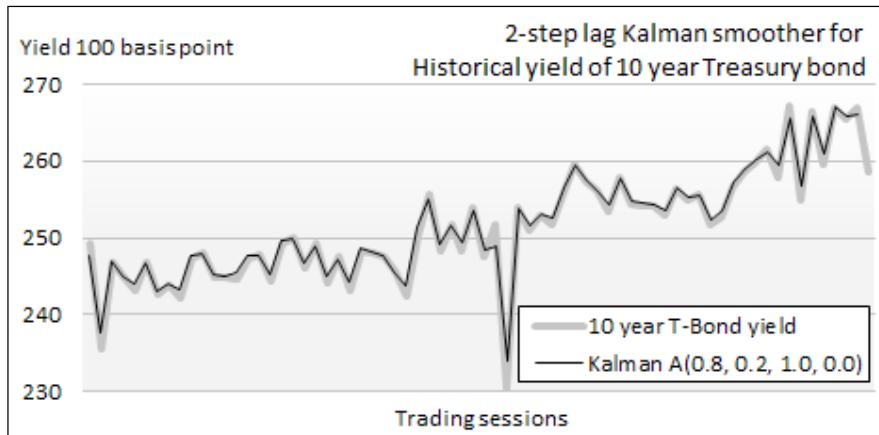
The state transition and the noise related to the process have to be selected carefully. The resolution of the state equations relies on the **QR decomposition**, which requires a non-negative definite matrix. The implementation in the Apache common library throws a `NonPositiveDefiniteMatrixException` if the principle is violated.

The smoothed yield is plotted along the raw data as follows:



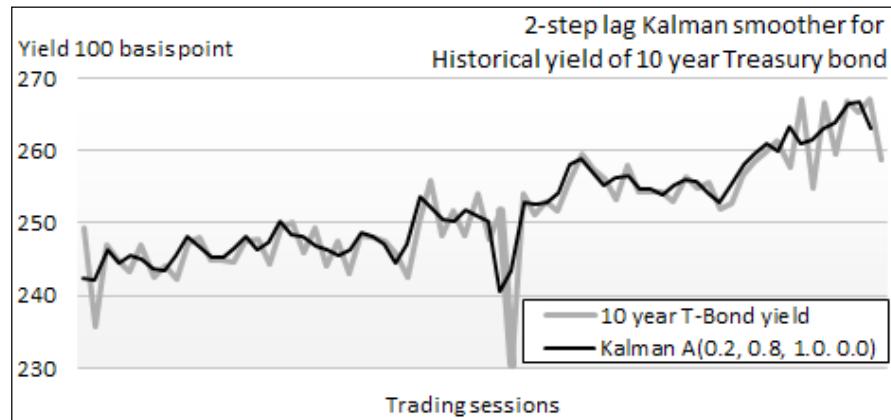
The output of the Kalman filter for the 10-year Treasury bond historical prices

Clearly, the yield time series has been smoothed. However, the amplitude of the underlying trend is significantly higher than any of the noise or the spikes. Consequently, the Kalman filter has a limited impact. 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 0.8-0.2

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

The Kalman filter is a very useful and powerful tool in understanding the distribution of the noise between the process and observation. Contrary to the low or pass-band filters based on the fast Fourier transform, the Kalman filter does not require computation of the frequencies spectrum or assume the range of frequencies of the noise.

However, the linear Kalman filter has its limitations:

- 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. Researchers have been able to formulate extensions to the Kalman filter, known as the **extended Kalman filter (EKF)** to filter signals from non-linear dynamic systems, at the cost of significant computational complexity.

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 two 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, non-linear least squares using the Levenberg-Marquardt optimizer, and probability distribution fitting

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 next 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 not available or not possible to create. In an attempt to extract some hidden association 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 space or feature set similarly to the divide-and-conquer approach commonly used in Computer Science.

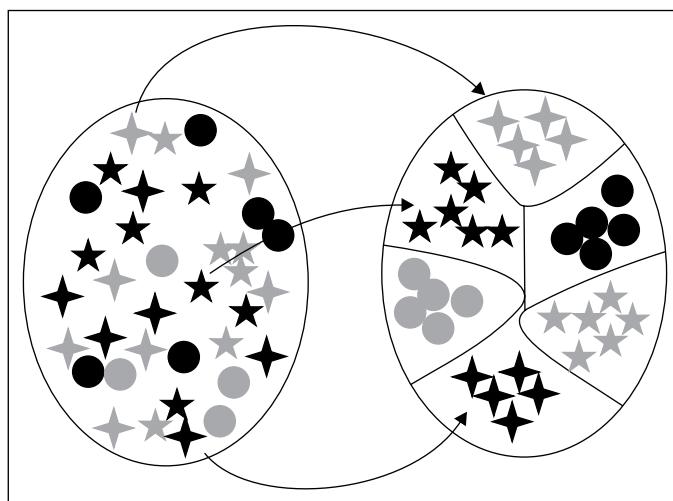
There are numerous unsupervised algorithms; some are more appropriate to handle dependent features while others generate more relevant groups in the case of hidden features [4:1]. In this chapter, you will learn three of the most common unsupervised learning algorithms:

- **K-means:** Clustering observed features
- **Expectation-maximization (EM):** Clustering observed and latent features
- **Principal components analysis (PCA):** Reducing 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 are described in the *Technical analysis* section under *Finances 101* in 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.

Clustering

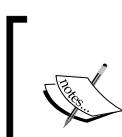
Problems involving a large number of features for large datasets become quickly intractable, and it is quite difficult to evaluate the independence between features. Any computation that requires some level of optimization and, at a minimum, computation of first order derivatives requires a significant 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 continuous, infinite, or very large datasets 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, which 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), which is a two-step probabilistic approach that maximizes the likelihood estimates of a set of parameters. EM is particularly suitable to handle missing data.

K-means clustering

K-means is a popular iterative clustering algorithm. 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 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
- Cosine of value observations

The Manhattan distance is defined by the absolute distance between two variables or vectors, $\{x_i\}$ and $\{y_i\}$, of the same size:

$$d(x, y) = \sum |x_i - y_i|$$

The implementation is generic enough to compute the distance between two arrays of elements of different types as long as an implicit conversion between each of these types to `Double` values is already defined, as shown here:

```
def manhattan[T <% Double, U <% Double](x: Array[T], y: Array[U]): Double = (x, y).zipped.foldLeft(0.0)((s, t) => s + Math.abs(t._1 - t._2))
```

The ubiquitous Euclidean distance is defined as the square of the distance between two vectors, $\{x_i\}$ and $\{y_i\}$, of the same size:

$$d(x, y) = \sum (x_i - y_i)^2$$

```
def euclidean[T <% Double, U <% Double](x: Array[T], y: Array[U]): Double = Math.sqrt((x, y).zipped.foldLeft(0.0)((s, t) => { val d = t._1 - t._2; s + d*d } ))
```

The cosine distance is defined as the cosine of an angle between two vectors, $\{x_i\}$ and $\{y_i\}$, of the same size:

$$d(x, y) = \frac{\sum x_i y_i}{(\sum x_i^2 \sum y_i^2)^{1/2}}$$

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 <% Double, U <% Double](x: Array[T], y: Array[U]): Double = {
    val zeros = (0.0, 0.0, 0.0)
    val norms = (x, y).zipped.foldLeft(zeros)((s, t) =>
        (s._1 + t._1*t._2, s._2 + t._1*t._1, s._3 + t._2*t._2))
    norms._1/Math.sqrt(norms._2*norms._3)
}
```

Performance of zip

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(x => f(x._1, x._2))
```

An functional alternative is to use the `Tuple2.zip` 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.

Overview of the K-means algorithm

The main advantage of the K-means algorithm (and the reason for its popularity) is its simplicity [4:3].



Let's consider K clusters $\{C_k\}$ with means $\{m_k\}$. The K-means algorithm is indeed an optimization problem, the objective of which is to minimize the reconstruction or the total error defined as the total sum of distance.

$$\min_{C_k} \sum_1^K \sum_{x_i \in C_k} d(x_i, m_k)$$

The steps of the iterative algorithm are:

1. Initialize the centroids or means m_k of the K clusters.
2. Assign observations to the nearest cluster given m_k .
3. Iterate until no observations are reassigned to a cluster:
 - Compute centroids m_k that minimize the total error reconstruction for the current assignment
 - Reassign the observations given the new centroids m_k

Step 1 – cluster configuration

The configuration of the K clusters consists of defining the following parameters for the K-means algorithm: number of K clusters, the distance metrics, the maximum number of iterations, and the initial value of the cluster's centroid.

Defining clusters

The first step is to define a cluster. A cluster is defined by the following parameters:

- Centroid: `center`
- The indices of the observations that belong to this cluster: `members`

The following code shows the definition of a cluster:

```
class Cluster[T <% Double] (val center: DblVector) {
  val members = new ListBuffer[Int]
```

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 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 *Appendix A, Basic Concepts*):

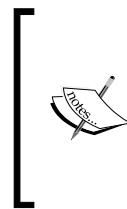
```
object Cluster {  
    def apply[T <% Double] (c:DblVector) :Cluster[T] = new Cluster[T] (c)  
}
```

At a minimum, a cluster should be able to manage its membership of observations, update its center, and compute the variance or standard deviation of all its member observations:

```
def += (n:Int): Unit = members.append(n)  
def moveCenter(xt: XTSeries[Array[T]]): Cluster[T] ={  
    val sums = members.map(xt(_).map(_.toDouble)).toList  
        .transpose  
        .map( _.sum)  
    Cluster[T] (sums.map( _ / members.size).toArray)  
}  
  
def stdDev(xt: XTSeries[Array[T]], distance: (DblVector, Array[T]) =>  
Double): Double = {  
    Stats[Double] (members.map(xt(_)))  
        .map( distance(center, _)).toArray.stdDev  
}
```

The three important methods that define the behavior of a cluster instance are as follows:

- `+=`: Add a member (index of an observation in the original time series).
- `moveCenter`: Create a new cluster with the existing members and a new centroid computed as the mean of all the observations contained in the cluster.
- `stdDev`: Compute the standard deviation (or density) of all the observations contained in the cluster relative to its center. The distance between each member and the centroid is extracted through a map, and then folded to generate the statistics. The function to compute the distance between the center and an observation is an argument of the method. The default distance is Euclidean.



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.

Defining K-means

Let's declare the K-means algorithm class, `KMeans`, with its public methods.

The `KMeans` class takes the number of clusters, `K`, and the maximum number of iterations, `maxIter`, as parameters. The implicit conversion of type `T` to a `Double` is specified by the `T <% Double` view bound. The `Ordering` class has to be passed implicitly as a parameter because it is required by the `sortWith` method in the `initialize` and `maxBy` methods. The `Manifest` method is required to preserve the type erasure for `Array[T]` in the JVM:

```
class KMeans[T <% Double](K: Int, maxIter: Int, distance:
  (DblVector, Array[T]) => Double)(implicit order: Ordering[T],
  m: Manifest[T]) extends PipeOperator[XTSeries[Array[T]],
  List[Cluster[T]]] {
  def |> : PartialFunction[XTSeries[Array[T]], List[Cluster[T]]]
  def initialize(xt: XTSeries[Array[T]]): List[Cluster[T]]
```

As with other data processing units, the extraction of K-means clusters is encapsulated by the pipe operator `|>`, so clustering can be integrated into a workflow using dependency injection described in the *Dependency injection* section in *Chapter 2, Hello World!*. The initialization of the centroids of each of the `K` clusters is performed by the private `initialize` method.

Initializing clusters

The initialization of the cluster centroids is important to ensure fast convergence of K-means. 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. Select the dimension $k \{x_{k'0}, x_{k'1} \dots x_{k'n}\}$ with 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 values, size $(S_m)/2$.
6. Use the corresponding observations as centroids.

The initialization algorithm is implemented by the private `initialize` method:

```
def initialize(xt:XTSeries[Array[T]]): List[Cluster[T]] = {  
    val stats = statistics(xt) //1  
    val maxSDevDim = Range(0,stats.size).maxBy (stats(_).stdDev)//2  
    val rankedObs = xt.zipWithIndex  
        .map(x=> (x._1(maxSDevDim), x._2)) //2  
        .sortWith( _._1 < _._1) //3  
    val halfSegSize = ((rankedObs.size>>1)/K).floor.toInt //4  
    val centroids = rankedObs.filter(isContained( _, halfSegSize,  
    rankedObs.size ) .map(n => xt(n._2)) //6  
        Range(0, K).foldLeft(List[Cluster[T]]())((xs, i) => Cluster[T]  
(centroids(i)) :: xs) //7  
}
```

Let's deconstruct the implementation of the Agha-Ashour algorithm in the `initialize` method.

The `statistics` function is applied to the input time series to extract the standard deviation for each dimension in the observations set (line 1). The dimension with the `maxSDevDim` maximum variance or standard deviation is computed by using the `maxBy` method on a `Stats` instance (line 2). Then, the observations are ranked by the increasing value of the standard deviation, `rankedObs` (line 3).

The ordered sequence of observations is then broken into $xt.size/K$ segments (line 4) and the indices of the `centroids` are selected as the midpoint (or median) observations of those segments using the filtering condition, `isContained`:

```
def isContained(t: (T,Int), hSz: Int, dim: Int): Boolean =  
    ((t._2 % hSz == 0) && (t._2 %(hSz<<1) != 0)
```

The indices of the centroid in the time series are converted to actual observations using a map method (line 6). Finally, the list of clusters is generated using a fold (foldLeft) method on the range of cluster indices (0, K-1) (line 7).

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: XTSeries[Array[T]], clusters: List[Cluster[T]], membership: Array[Int]): Int = {
    xt.toArray
    .zipWithIndex
    .filter(x => { //1
        val nearestCluster = getNearestCluster(clusters, x._1)//2
        val reassigned = nearestCluster != membership(x._2)
        clusters(nearestCluster) += x._2 //3
        membership(x._2) = nearestCluster //4
        reassigned
    }).size
}
```

The core of the assignment of observations to each cluster is the filter on the time series (line 1). The filter computes the index of the closest cluster and checks whether the observation is to be reassigned (line 2). The observation at the index x._2 is added to the nearest cluster, clusters(nearestCluster) (line 3). The current membership of the observations is then updated (line 4).

The cluster closest to an observation data is computed by the `getNearestCluster` method as follows:

```
def getNearestCluster(clusters: List[Cluster[T]], x: Array[T]): Int = {
    clusters.zipWithIndex.foldLeft((Double.MaxValue, 0))((p, c) => {
        val measure = distance(c._1.center, x)
        if(measure < p._1) (measure, c._2) else p
    })._2
}
```

A fold is used to extract from the list of clusters the cluster that is closest to the observation `x` using the `distance` metric defined in the K-means constructor.

Step 3 – iterative reconstruction

The final step is to implement the iterative computation of the reconstruction error. In this implementation, the iteration terminates when no more observations are reassigned to different clusters. As with other data processing units, the extraction of K-means clusters is encapsulated by the pipe operator `|>`, so that clustering can be integrated into a workflow using dependency injection described in the *Dependency Injection* section in *Chapter 2, Hello World!*.

The generation of the K clusters is executed by the data transformation `|>:`

```
def |> :PartialFunction[XTSeries[Array[T]], List[Cluster[T]]] = {
  case xt: XTSeries[Array[T]] if (xt.size > 2 && xt(0).size > 0) => {
    val clusters = initialize(xt) //1

    if( clusters.isEmpty) List.empty
    else {
      val membership = Array.fill(xt.size)(0)
      val reassigned = assignToClusters(xt,clusters,membership)//2
      var newClusters: List[Cluster[T]] = List.empty
      Range(0, maxIters).find( _ => {
        newClusters = clusters.map( c => {
          if( c.size > 0) c.moveCenter(xt, dimension(xt))
          else clusters.filter( _.size > 0)
            .maxBy( _.stdDev(xt, distance))
        }) //3
        assignToClusters(xt, newClusters, membership) == 0
      }) match {
        case Some(index) => newClusters
        case None => { ... }
      } //4
    }
  }
}
```

As described in the algorithm overview section, the main method initializes the membership for all the observations (line 1), creates and initializes the clusters, and assigns the observations to clusters using the `assignToClusters` method (line 2). The iteration updates the content of each cluster using the `moveCenter` method, by assigning new observations to the cluster with the highest standard deviation (line 3). The iterative loop exits when no more reassignment is needed (line 4).

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 to 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 process needs to be executed once again.

The companion object for `KMeans` implements the `apply` constructor and the computation of the `stdDev` standard deviation for each cluster. The default constructor uses the Euclidean distance:

```
def apply[T <% Double](K: Int, maxIters: Int)(implicit order: Ordering[T], m: Manifest[T]): KMeans[T] = new KMeans[T](K, maxIters, euclidean)
def stdDev[T](c: List[Cluster[T]], xt: XTSeries[Array[T]]): List[Double] = c.map(_.stdDev(xt))
```

The `stdDev` method computes the standard deviation of the distances between each data point that belongs to a `c` cluster and its centroid.



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.

Note that ordering a trait and `Manifest` have to be provided in the `apply` constructor because there is no guarantee that such capabilities are provided in runtime by the client code.

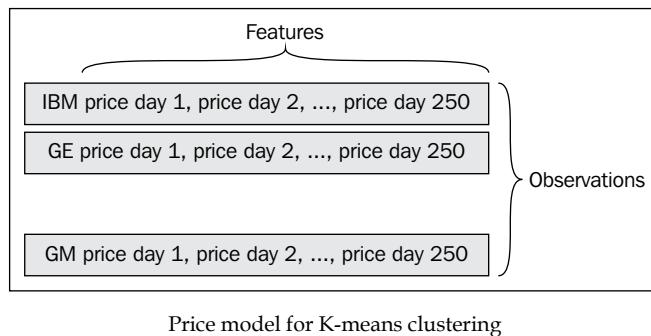
Curse of dimensionality

A model with a significant number of features (high dimensions) requires a larger number of observations in order to extract robust clusters. K-means clustering with very small datasets, of size less than 50, produces models with high bias and a limited number of clusters that 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 versus size of training set

The issue with the dimensionality of models versus the number of observations 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!



Price model for K-means clustering

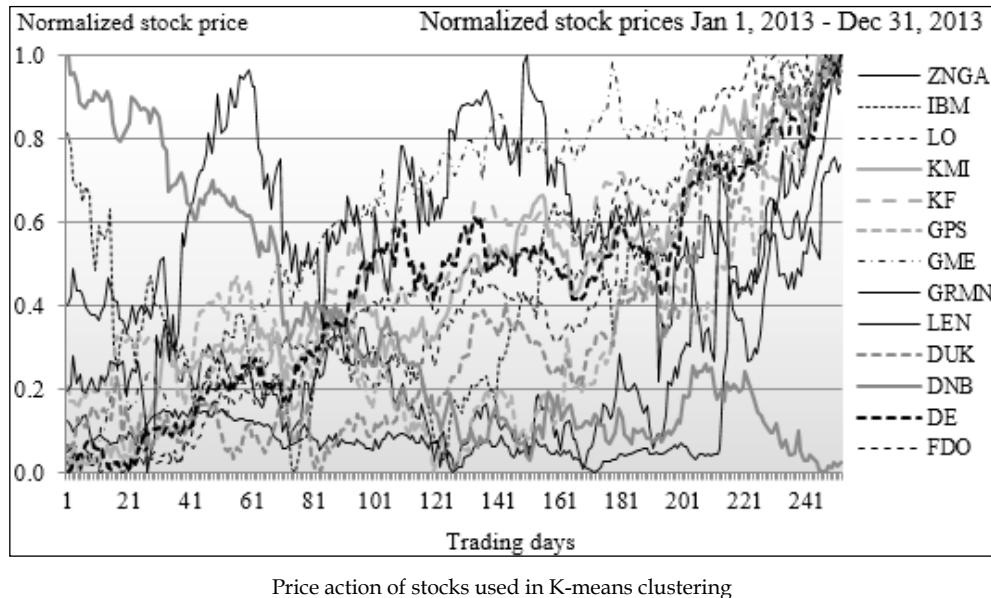
There are options to get around this limitation and shrink the number of observations; among them are:

- Sampling the trading data without losing a significant amount of information from the raw data, assuming 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 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 sampling approach 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, but they are not recommended for actual commercial analytical applications. The principal component analysis introduced in the last section of this chapter is one of the most reliable dimension reduction techniques.

Experiment

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 (SAMPLES = 50) observations are the stock closing price for each trading session between the 80th and 130th days. The adjusted daily closing prices are normalized using the minimum and maximum values.

First, let's create a simple function to execute the K-means algorithm:

```
Val MAX_ITERS = 150
def run(k: Int, obs: DblMatrix): Unit = {
    val kmeans = KMeans[Double](k, MAX_ITERS) //1

    val clusters = kmeans |> XTSeries[DblVector](obs) //2
    clusters.foreach( _.center.foreach( show( _ ) ) ) //3
    clusters.map( _.stdDev(XTSeries[DblVector](obs, euclidean)) .
        foreach( show( _ ) ) //4
    }
}
```

The KMeans class is first initialized with a number of clusters, k , and a maximum number of iterations, MAX_ITERS (line 1). These two parameters are domain and problem specific. The clustering algorithm is executed (line 2) returning a list of clusters. The clusters' centroid information is then displayed (line 3) and the standard deviation is computed for each of the clusters for a given number of clusters, k , and observations, obs (line 4).

Let's load the data from CSV files using the `DataSource` class (refer to the *Data extraction* section in *Appendix A, Basic Concepts*):

```
final val path = "resources/data/chap4/"
val extractor = YahooFinancials.adjClose :: List[Array[String]
=>Double] () // 5
def symbols = DataSource.listSymbols(path) //6

final val START = 80
final val SAMPLES = 50
val normalize=true
val prices = symbols.map(s =>DataSource(s,path,normalize) |>
extractor) //7
prices.find(_.isEmpty) match { //8
  case Some(noPrice) = { ... }
  case None => {
    val values = prices. map(x => x(0))
      .map(_.drop(START).take(SAMPLES))
    args.map(_.toInt) foreach( run(_, values)) //9
  }
}
```

As mentioned earlier, the cluster analysis applies to the closing price in the range between the 80th and 130th trading day. The extractor is defined to extract the adjusted closing price for a stock whose price information is retrieved from `YahooFinancials` (line 5). The list of stock symbols is used to extract price information from CSV files located at the path (line 6). For instance, the ticker symbol for General Electric Corp. is GE and the trading data is located in `GE.csv`.

The 50 daily prices for each stock are extracted by an instance of `DataSource` (line 7). The `run` method introduced earlier is invoked either for each stock or as soon as K-means fails through an exit condition in the `find` method (line 8). The normalized data `values.toArray` for the specific time window is extracted by the combination of calls to `drop` and `take` Scala array methods (line 9).

The first test run is executed with $K=3$ clusters. The mean (or centroid) vector for each cluster is plotted as follows:

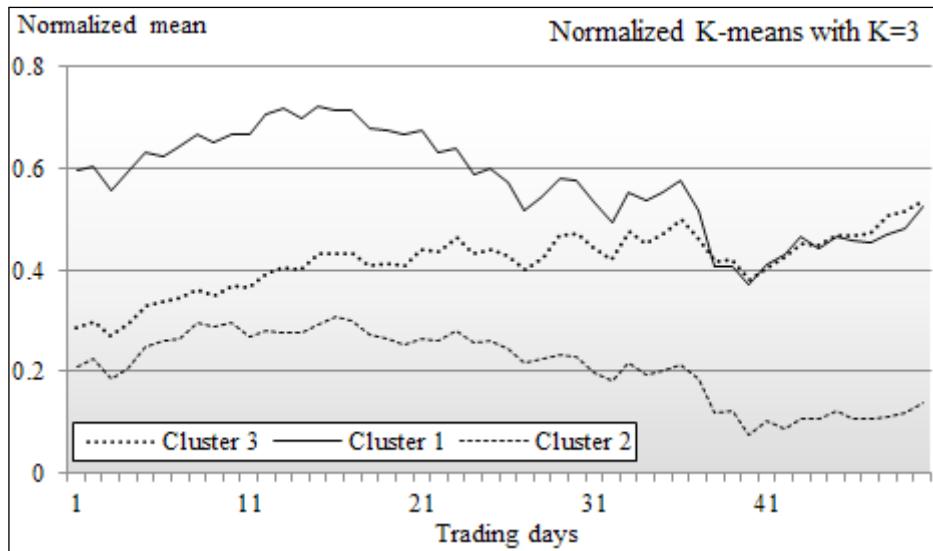


Chart of means of clusters using K-means K=3

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 towards 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 list of stocks for each of the clusters whose centroid values are displayed on the chart:

Cluster	List of stocks
Cluster 1	AET, AHS, BBBY, BRCM, C, CB, CL, CLX, COH, CVX, CYH, DE, DG, DHL, DO, DUK, EA, EBAY, EXC, EXP, FE, GLW, GPS, IBM, JCP, JNJ, JWN, K, KF, KMI, KO, KRFT, LEN, LINC, LRCX, MSFT, NVML, THC, XRT
Cluster 2	AA, AAPL, ADBE, ADSK, AFAM, AMZN, AU, BHI, BTU, CAT, CCL, CCMP, COP, CSC, CU, DOW, EMR, ENTG, ETFC, FCX, FDX, FFIV, FISV, FLIR, FLR, FLS, FTR, GLD, GRMN, GT, JCI, QCOM, QQQ, SIL, SLV, SLW
Cluster 3	ADM, ADP, AXP, BA, BBT, BEN, BK, BSX, CA, CBS, CCE, CELG, CHK, CI, CME, CMG, CSCO, CVS, DAL, DD, DNB, EMC, EXPE, F, FDO, FITB, FMC, GCI, GE, GM, GME, GS, HCA, JNPR, JPM, KLAC, LH, LLL, LM, LMT, LNC, LO, MKSI, MU, NEM, TRW, TXN, UNH, WDC, XLF, XLNX, ZNGA

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 is plotted as follows for $K = 2$:

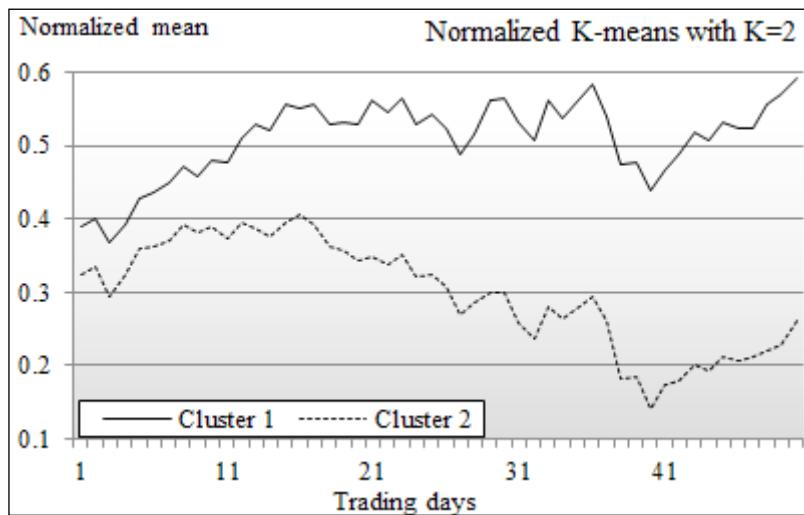


Chart of means of clusters using K-means $K=2$

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 $K = 3$ 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 $K = 3$. 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 is plotted as follows for $K = 5$:

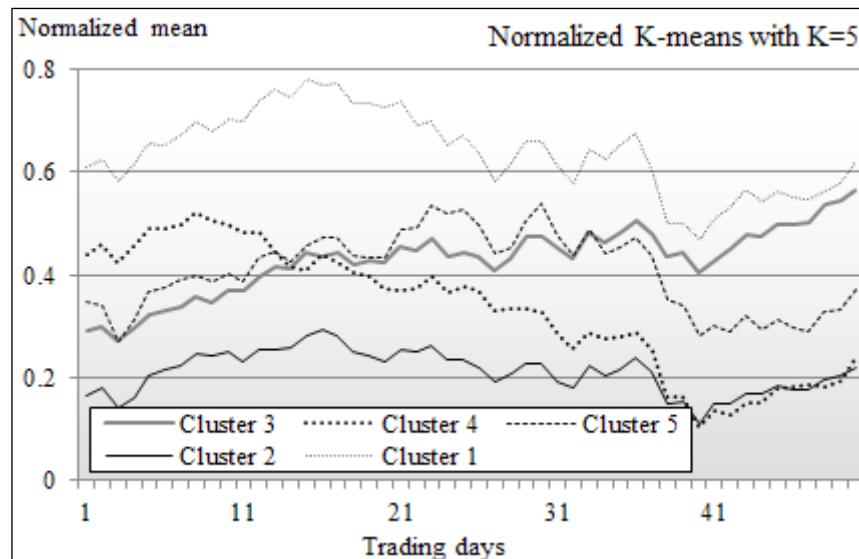


Chart of means of clusters using K-means $K=5$

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 $K = 10$:

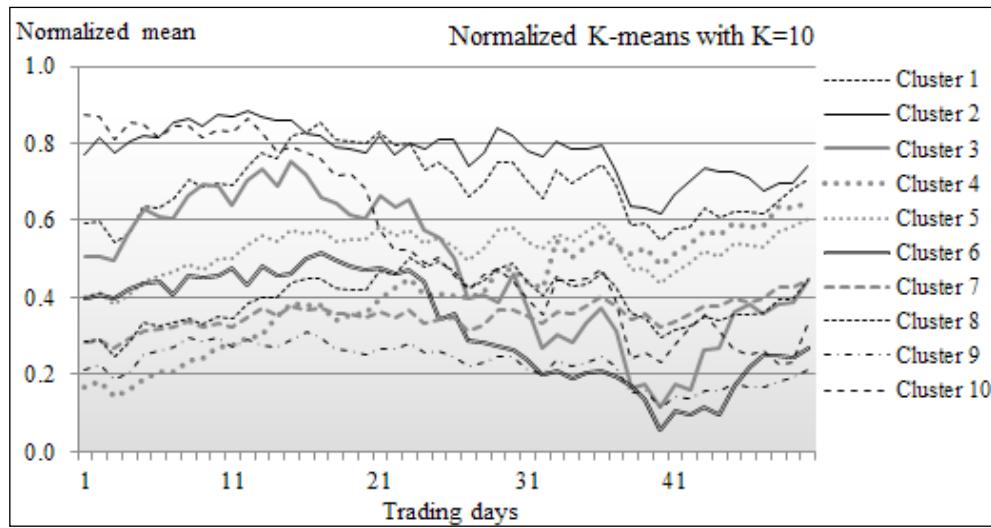


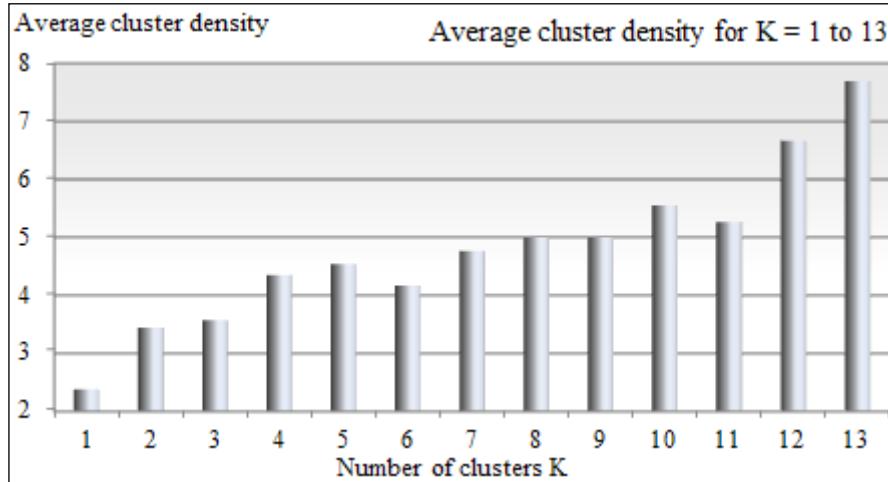
Chart of means of clusters using K-means K=10

The means for clusters 1, 2, and 3 seen in the first chart for the case $K = 3$ 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 members of each cluster and its mean (or centroid):

$$d(C_j) = 1 / \sum_{x \in C_j} (x - c_j)^2$$

The density of the cluster is plotted against the number of clusters with K = 1 to 13:



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 to 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 contained in CC, then the clustering 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 precision, recall, and F1 measures introduced in the *Assessing a model* section of *Chapter 2, Hello World!*, confirms whether the tuning parameters and labels you selected for your cluster are indeed correct.

Validation



The quality of the clusters, as measured by the F1 statistics, depends on the labeling of the cluster and the rule (that is, label a cluster with the industry with the highest relative percentage of stocks in the cluster) used to assign a label. This process is very subjective. The only sure way to validate a validation methodology is to evaluate several labeling schemes and select the one that generates the highest F1 statistics.

We reviewed some of the tuning parameters that impact the quality of the results of the K-means clustering. They are as follows:

- Initial selection of centroid
- Number of K clusters

In some cases, the similarity criterion (that is, Euclidean distance versus cosine value) can have an impact on the *cleanness* 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) variable. The expectation-maximization algorithm described in the next section addresses some of these limitations.

Expectation-maximization (EM) 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.

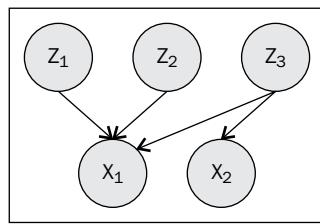
The iterative algorithm consists of computing:

- The 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 model

Latent variables Z 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 in the following information box.

The mixture model

If $\{x_i\}$ is a set of observed features associated with latent features $\{z_k\}$, the probability for the feature x_i given z_k has a value j :

$$p(x_i | Z_k = j)$$

The probability p is called the base distribution. If we extend to the entire model, $\theta = \{x_i, z_k\}$, the conditional probability is defined as follows:



$$p(x_i | \theta) = \sum_{j=1}^J \pi_j p(x_i | Z_k = j)$$

The most widely used mixture model is the Gaussian mixture model that represents the base distribution p as a Normal distribution and the conditional probability as a weighted Normal multivariate distribution:

$$p(x_i | \theta) = \sum_{j=1}^J \pi_j \frac{1}{2\pi^{d/2} \sqrt{|\Sigma_j|}} e^{-\frac{1}{2} (x_i - \mu_j)^T \Sigma_j^{-1} (x_i - \mu_j)}$$

EM overview

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 (LL).
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).

Log likelihood

- **LL:** Let's consider a set of observed variables $X=\{x_i\}$ and latent variables $Z=\{z_j\}$. The log likelihood for X for given Z is:

$$L(\theta) = \sum \log p(x_i, z_j | \theta)$$

- **E-step:** The expectation for the model variable θ at iteration t is computed as:

$$Q(\theta, \theta_t) = E[L(\theta) | X, \theta^t]$$

- **M-step:** The function Q is maximized for the model features θ as:

$$\theta^{t+1} = \arg \max_{\theta} Q(\theta, \theta^t)$$



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 (LL, 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 by using the Apache Commons Math library package [4:10].

 **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.

First, let's define convenient internal types:

```
type EM = MultivariateNormalMixtureExpectationMaximization
type EMOoutput = List[(Double, DblVector, DblVector)]
import scala.collections.JavaConversions._ //1
```

The constructor of the `MultivariateEM` class uses the standard template for machine learning algorithm classes:

- Parameterized view bound type
- Implementation of EM as a data transformation by extending `PipeOperator`

Here is an implementation of the constructor of `MultivariateEM`:

```
class MultivariateEM[T <% Double](K: Int) extends PipeOperator[XTSeries[Array[T]], EMOoutput]
```

The Apache Commons Math Java implementation of the EM uses Java container classes that need to be explicitly converted to Scala collections. Those conversions are defined in the `JavaConversions` package (line 1).

The implementation of the EM algorithm in the data transformation `|>` operator uses the Apache Commons Math `MultivariateNormalMixture` class for the Gaussian mixture model and the `MultivariateNormalMixtureExpectationMaximization` class for the EM algorithm:

```
def |> : PartialFunction[XTSeries[Array[T]], EMOoutput] = {
  case xt: XTSeries[Array[T]] if (xt.size>0 && dimension(xt)>0) =>{
    val data: DblMatrix = xt //2
    val multivariateEM = new EM(data)
    val est = MultivariateNormalMixtureExpectationMaximization
      .estimate(data, K)
    multivariateEM.fit(est) //3

    val newMixture = multivariateEM.getFittedModel //4
    val components = newMixture.getComponents.toList //5
    components.map(p => (p.getKey.toDouble, p.getValue.getMeans,
    p.getValue.getStandardDeviations) ) //6
    ...
  }
}
```

Let's look at the main `|>` method of the `MultivariateEM` wrapper class. The first step is to convert the time series into a primitive matrix of `Double` with observations and historical quotes as rows and the stock symbols as columns (line 2).

The initial mixture of Gaussian distributions can be provided by the user or can be extracted from the dataset as an estimate (line 3). The `getFittedModel` model triggers the M-step (line 4).

The Apache library uses Java primitives that need to be converted to Scala types using the package `import scala.collection.JavaConversions`. An instance of `java.util.List` is converted to `scala.collection.immutable.List` using `toList`, which invokes the `asScalaIterator` method of `wrapAsScala`, one of the base traits of `JavaConversions` (line 5).

The `<Double, MultivariateNormalDistribution>` key-value pair, returned by the call to `getFittedModel` by the Apache `math` method, is to be converted to a tuple containing the mean and standard deviation for each cluster (line 6).

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) => ...  
}
```

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 paragraph related to the curse of dimensionality, 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 stocks and apply a basic 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 quotes 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 `K` of clusters used in the EM algorithm:

```
val extractor = YahooFinancials.adjClose :: List[Array[String]
=>Double] () //1

val period = 8
val samplingRate = 10
val smAv = SimpleMovingAverage[Double](period) //2
val obs = DataSource.listSymbols(path).map(sym => { //3
    val xs = DataSource(sym, path, true) |> extractor //2
    val values : XTSeries[Double] = XTSeries.|>(xs)).head //4
    val filtered = smAv |> values
    filtered.zipWithIndex //5
        .drop(period+1).toArray //6
        .filter(_._2%samplingRate==0)
        .map(_._1)
})
```

The first step is to extract the historical quotes for all the stocks using the same extractor as in the K-means test case (line 1).

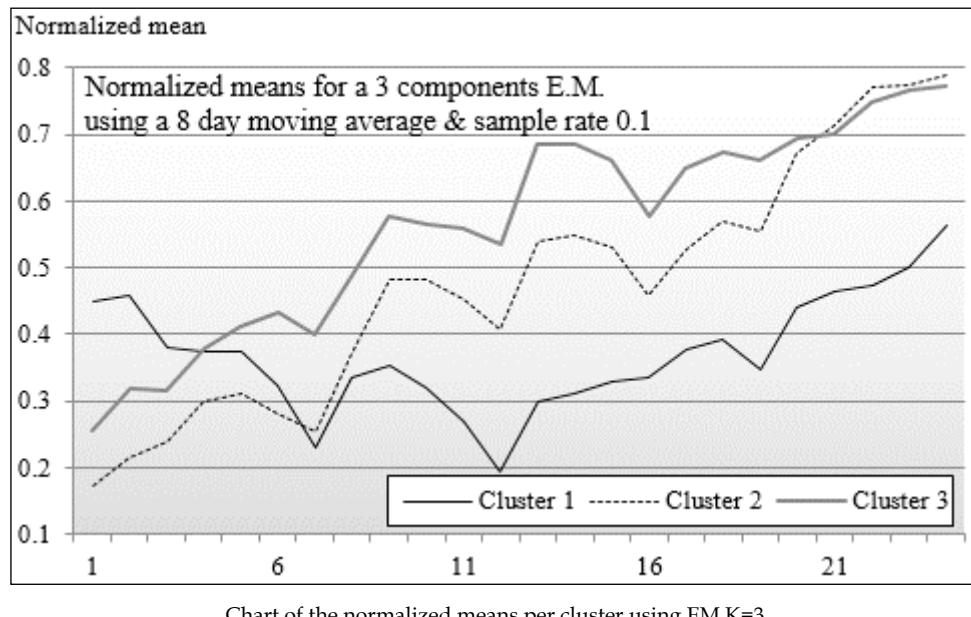
The symbols of the stocks under consideration are extracted from the name of the files in the path directory. The historical data is contained in the CSV file named `path/STOCK_NAME.csv` (line 3). An implicit conversion is triggered by an assignment of values of the type `XTSeries[Double]` (line 4). The simple moving average algorithm zeroed out the first period values in the smoothed data, filtered (line 5). Those null values have to be dropped before applying the sampling (line 6).

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 a period of 8:

```
MultivariateEM[Double](K) |> XTSeries[DblVector](obs) foreach (...)
```

The driver prints the key (line 3), the mean (coordinates of the centroid vector) (line 4), and the standard deviation for each component (cluster).

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 or centroid of each of the 3 clusters:



The mean vectors of clusters 2 and 3 have similar patterns, which may suggest that 2 components or clusters could 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 K = 3:

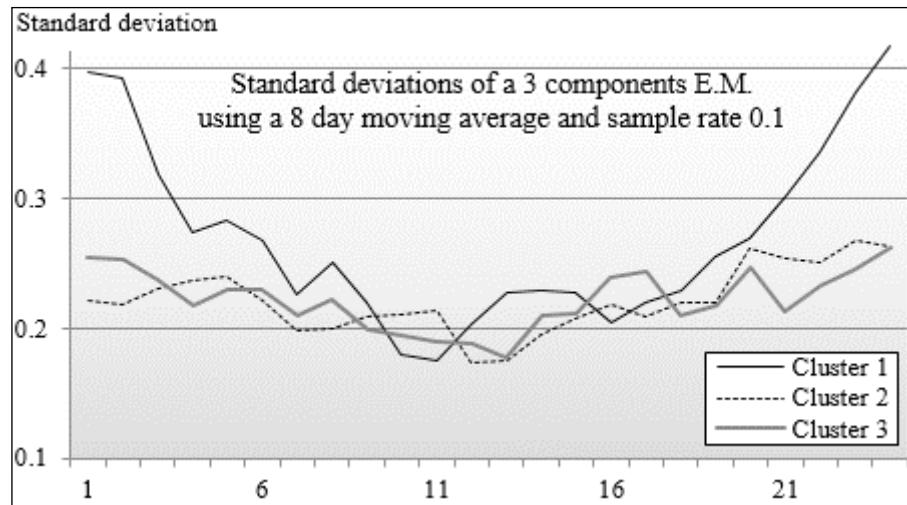


Chart of the normalized standard deviation per cluster using EM K=3

The distribution of the standard deviation along 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 semihomogenous 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 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].

Online EM

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 online EM is beyond the scope of this tutorial. However, you may need to know that there are several algorithms available for online EM if you ever have to deal with large datasets: **batch EM**, **stepwise EM**, **incremental EM**, and **Monte Carlo EM** [4:12].

Dimension reduction

Without prior knowledge of the data 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 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 a 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.

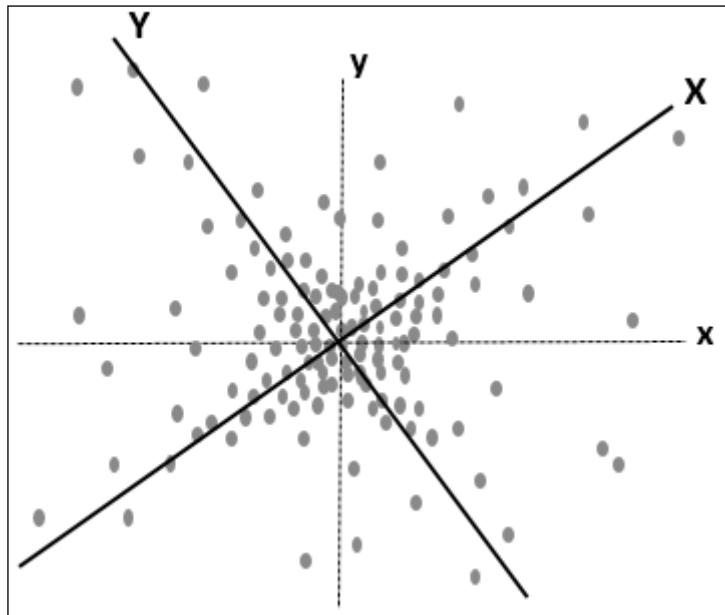
There are three approaches to reduce the number of features in a model:

- Statistical analysis solutions such as ANOVA for smaller feature sets
- Regularization and shrinking techniques, which are introduced in *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 (PCA)

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 $\{xi, yi\}$ plotted on the following chart:



Visualization of principal components for a 2-dimension model

The features x and y are converted into two variables X and Y (that is rotation) to more 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 the tutorial from Lindsay Smith [4:13] that describes the PCA algorithm in a very concrete and simple way using a 2-dimension model.

PCA and covariance matrix

The covariance of two features X and Y with the observations set $\{x_i, y_i\}$ 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 .

The covariance is computed from the *zScore* of each observation:

$$x_i = (x_i - \bar{x}) / \sigma$$

For a model with n features, x_i , the covariance matrix is defined as:



$$\Sigma = \begin{bmatrix} \text{cov}(x_0, x_0) & \cdots & \text{cov}(x_0, x_{n-1}) \\ \vdots & \text{cov}(x_i, x_j) & \vdots \\ \text{cov}(x_{n-1}, x_0) & \cdots & \text{cov}(x_{n-1}, x_{n-1}) \end{bmatrix}$$

The transformation of x to X consists of computing the eigenvalues of the covariance matrix:

$$\Sigma' = W^T \Sigma W = \left\| \text{cov}(X_i, X_j) \right\| \text{ and } X = W^T x$$

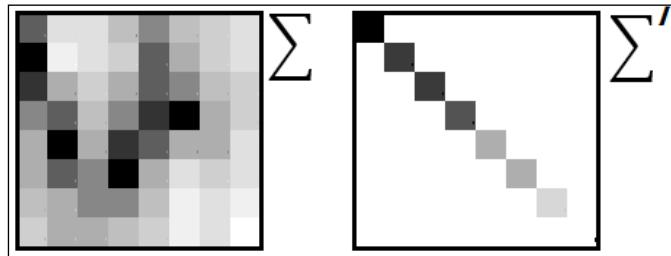
The eigenvalues are ranked by their decreasing order of variance and the cumulative variance for each eigenvalue is computed. Finally, the m top eigenvalues for which the cumulative of variance exceeds a predefined threshold (percentage of the trace of the matrix) are the principal components or reduced feature set.

$$Z = \left\{ X_i : m \mid \sum_{k=1}^m \text{cov}(x_k, x_k) \geq \epsilon \cdot \text{Tr}(\text{cov}) \right\}$$

The algorithm is implemented in five steps:

1. Compute the $zScore$ 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 as 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 color 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

PCA can be easily implemented by using the Apache Commons Math library methods that compute the eigenvalues and eigenvectors. Once again, the main routine is implemented as a pipe operator so that it can be used in a generic workflow as defined in the *The Pipe Operator* section under *Designing a workflow* in *Chapter 2, Hello World!*.

```
import types.ScalaML._, types.CommonMath._, //2

def |> : PartialFunction[XTSeries[Array[T]], (DblMatrix, DblVector)] ={
  case xt: XTSeries[Array[T]] if(xt != null && xt.size>1) => {
```

```
zScoring(xt) match { //1
  case Some(obs) => {
    val covariance = new Covariance(obs).getCovarianceMatrix //3
    val transf = new EigenDecomposition(covariance)
    val eigVectors = transf.getV //4
    val eigValues = new ArrayRealVector(transf.getRealEigenvalues)
    val cov = obs.multiply(eigVectors).getData
    (cov, eigValues.toArray) //5
  ...
}
```

PCA requires that the original set of observations is standardized using the z-score transformation. It is implemented using the `XTSeries.zScoring` function introduced in the *Normalization and Gauss distribution* section in *Chapter 1, Getting Started* (line 1).

The assignment forces the implicit conversion of a time series of features of the type `T` into a matrix of the type `Double`. The implicit conversions between Scala primitives and ScalaML types such as `DblMatrix` (resp. between Apache Commons Math types and Scala ML) are defined in `Types.ScalamML`, as mentioned in the *Type conversions* section in *Chapter 1, Getting Started* (resp. `Types.CommonMath` in the *Time series* section in *Chapter 3, Data Preprocessing*) (line 2). The covariance matrix is computed based on the `zScore` created from the original observations (line 3). The eigenvectors, `eigVectors`, are computed using the `getV` method in the Apache Commons Math `EigenDecomposition` class. The eigenvalues, `eigValues`, are extracted as principal components (line 4).

Finally, the data transformation returns the tuple (*covariance matrix, array of eigenvalues*) (line 5).

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:

- 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 *Appendix A, Basic Concepts*.

The input data is specified with the following format as a tuple: the 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 pca = new PCA[Double] //1
val input = data.map( _._2.take(3))
val cov = pca |> XTSeries[DblVector](input) //2
Display.show(toString(cov), logger) //3
```

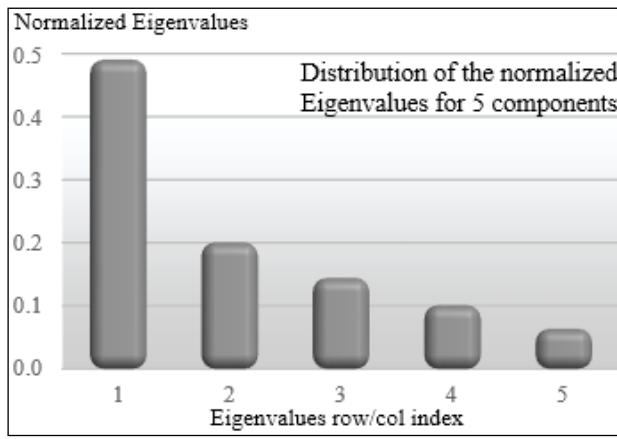
Once the `pca` class is instantiated (line 1), the eigenvalues and covariance matrix, `cov`, are computed (line 2), and then displayed using the utility singleton `Display` that formats messages and appends to the logger (line 3).

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 a bar chart:



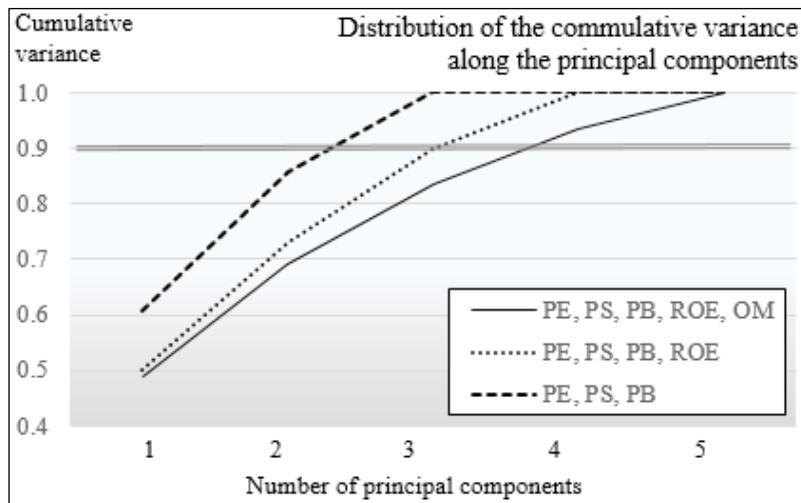
Distribution of eigenvalues in PCA for 5 dimensions

The chart shows that 3 out of 5 features account for 85 percent of 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:

```
Val numFeatures = 4
val ts = XTSeries[DblVector](data.map(_.2.take(numFeatures)))
```

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



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 3-feature model, 25 percent for the 4-feature model, and 40 percent for the 5-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 using a regression estimator such as **partial least square regression (PLSR)** or the **predicted residual error sum of squares (PRESS)**. For those not afraid of statistics, I recommend *Fast Cross-validation in Robust PCA* by S. Engelen and M. Hubert [4:14]. You need to be aware, however, that the implementation of these regression estimators is not simple.

The principal components can be validated through a 1-fold or K-fold cross-validation, by performing some type of regression estimators or EM on the same dataset. The validation of the PCA is beyond the scope and space allocated to this chapter.

Principal components analysis is a special case of the more general factor analysis. The later class of algorithm does not require the transformation of the covariance matrix to be orthogonal.

Other dimension reduction techniques

Although quite popular, the principal components analysis is far from being the only dimension reduction method. Here are some alternative techniques, listed as reference: factor analysis, principal factor analysis, maximum likelihood factor analysis, independent component analysis (ICA), Random projection, nonlinear PCA, nonlinear ICA, Kohonen's self-organizing maps, neural networks, and multidimensional scaling, just to name a few [4:15].

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, K the number of clusters, n the number of observations, and m the number of features. The algorithm can be improved through the use of other techniques by using the following techniques:

- Reducing the average number of iterations by seeding the centroid using an algorithm such as initialization by ranking the variance of the initial cluster as described at 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 possible 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: 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. A second pass is then run on a smaller set of 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 (GMM)** 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 explained in the *Apache Spark* section in *Chapter 12, Scalable Frameworks*
- Using a kernel method 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 the number of observations. The first term represents the computational complexity for computing the covariance matrix. The last term reflects the computational complexity of the eigenvalue decomposition.

The list of potential performance improvements or alternative solutions for PCA includes:

- 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)** [4:16]
- Using the PCA in conjunction with EM (a research)
- Deploying a dataset on a parallel data processing framework such as Apache Spark or Hadoop as explained 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

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

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 a reminder, generative classifiers are supervised learning algorithms that attempt to fit a **joint probability distribution**, $p(X, Y)$, of two events X and Y , 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 build a Naïve Bayes classifier to predict stock price movement, given some prior technical indicators in the analysis of financial markets.

Finally, you will 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 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 [5:1].

There are two categories of graphs, and therefore, graphical models:

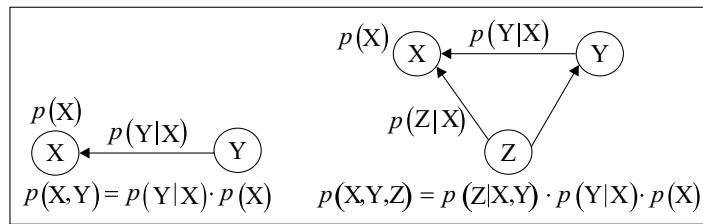
- Directed graphs such as Bayesian networks
- Undirected graphs such as conditional random fields (refer to the *Conditional random fields* section in *Chapter 7, Sequential Data Models*)

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 (or independence) between variables
- Represent statistical inference in terms of graphical manipulation

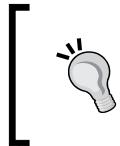
A **Bayesian network** is a directed graphical model defining a joint probability over a set of variables [5:2].

The two joint probabilities, $p(X, Y)$ and $p(X, Y, Z)$, can be graphically modeled using Bayesian networks, as follows:



Examples of probabilistic graphical models

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.



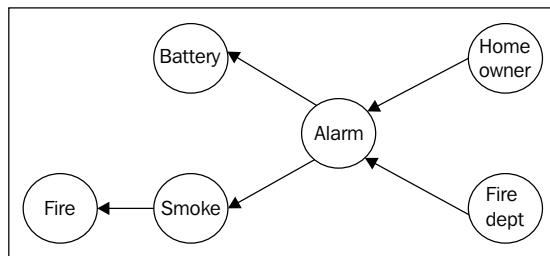
Metaphor for graphical models

From a software engineering perspective, graphical models visualize probabilistic equations the same way the UML class diagram visualizes 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.



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 in *Chapter 1, Getting Started*.

Naïve Bayes classifiers

This 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, concrete examples [5:5].

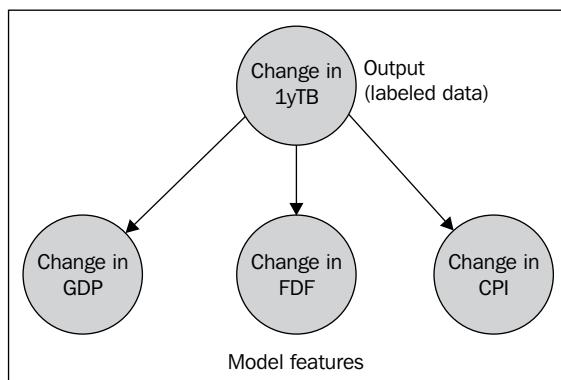
Introducing the multinomial Naïve Bayes

Let's consider the problem of how to predict 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)**, change in the **Federal fund rate (FFR)** and the **gross domestic product (GDP)** as a first set of features. The terminology is described in the *Terminology* section under *Finances 101* in *Appendix A, Basic Concepts*.

The use case is to predict 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 is always increasing 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 Naïve Bayes model for predicting the change in the yield of the 1-year T-bill

The correlation (or cause-effect relationship) is derived from historical data. The methodology consists of counting the number of times each feature either increases (UP) or decreases (DOWN), and recording the corresponding output (or labeled data), as illustrated in the following table:

ID	GDP	FDF	CPI	1yTB
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 occurrence of each change {UP, DOWN} for the three features and the output value (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 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

For this table, we conclude that the yield of the 1-year Treasury bill increases when the GDP is increasing (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 terminology used in the Bayesian model:

- **Class prior probability** or **class prior** is the probability of a class
- **Likelihood** is the probability of an observation given a class, also known as the probability of the predictor given a class
- **Evidence** is the probability of observations occurring, also known as the prior probability of the predictor
- **Posterior probability** is the probability of an observation x being in a given class

No model can be simpler! The log likelihood, $\log(p(x | C))$, is commonly used instead of the likelihood, $p(x | C)$, (probability of an observation given a class) in order to reduce the impact of the features y that have a low likelihood, $p(y | C)$.

The objective of the Naïve Bayes classification of a new observation, is to compute the class that has the higher log likelihood. The mathematical notation for the Naïve Bayes model is also straightforward.

The posterior probability, $p(C_j | x)$:

$$p(C_j | x) = \frac{p(x | C_j) \cdot p(C_j)}{p(x)}$$

- $x = \{x_i\} (0, n-1)$, with a set of n features
- $\{C_j\}$, a set of classes with their class prior $p(C_j)$
- $p(x)$, the evidence of new observation
- $p(x | C_j)$, the likelihood for each feature

Posterior probability, $p(C_j | x)$, with conditional independence:



$$p(C_j | x) = \prod_{i=0}^{n-1} p(x_i | C_j) \cdot p(C_j)$$

- x_i are independent and the probabilities are normalized for evidence $p(x) = 1$

Log-likelihood:

$$\log p(C_j | x) = \sum_{i=0}^{n-1} \log p(x_i | C_j) + \log p(C_j)$$

Naïve Bayes classification:

$$C_m = \arg \max_j (\log p(C_j | x))$$

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 the FDF rate is rather infrequent.

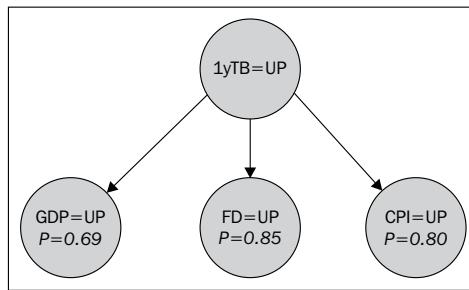
The frequentist perspective

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.

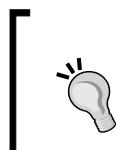
Let's consider the first example that predicts the direction of change in the yield of the 1-year Treasury bill given changes in the GDP, FDF, and CPI.

The results are expressed with simple probabilistic formulas and a directed graphical model:

$$\begin{aligned}
 P(\text{GDP=UP} | \text{1yTB=UP}) &= 110/159 \\
 P(\text{1yTB=UP}) &= \text{num occurrences } (\text{1yTB=UP}) / \text{total num of} \\
 &\quad \text{occurrences} = 159/256 \\
 p(\text{1yTB=UP} | \text{GDP=UP}, \text{FD=UP}, \text{CPI=UP}) &= p(\text{GDP=UP} | \text{1yTB=UP}) \times \\
 &\quad p(\text{FD=UP} | \text{1yTB=UP}) \times \\
 &\quad p(\text{CPI=UP} | \text{1yTB=UP}) \times \\
 p(\text{1yTB=UP}) &= 0.69 \times 0.85 \times \\
 &\quad 0.80 \times 0.625
 \end{aligned}$$



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, in case 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 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 feature with the highest frequency; in this case, the CPI

Let's select another use case for which a large historical data set is available and can be automatically labeled.

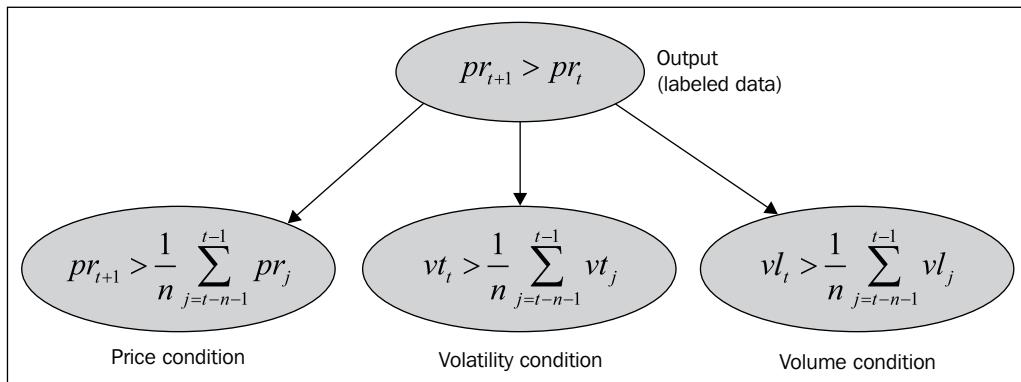
The predictive model

The predictive model is the second use case that consists of predicting the direction of the 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, $pr(i), i=1, t$. The features volume and volatility have been already used in the *Creating a model (learning)* section under *Let's kick the tires in Chapter 1, Getting Started*.

Therefore, the three features under consideration are:

- 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 (price at session $t+1$ is greater than price at session t) and three features: price condition (1), volume condition (2), and volatility condition (3).



A Bayesian model for predicting the future direction of the stock price

This model works under the assumption that there is at least one observation, and ideally few observations for each feature and for each labeled output.

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 formula.

The smoothing factor for counters

Laplace smoothing of the mean k/N out of N observations of features of dimension n :



$$\mu' = \frac{k+1}{N+n}$$

Lidstone smoothing with a factor α :

$$\mu' = \frac{k+\alpha}{N+\alpha \cdot n}$$

The two formulas are commonly used in natural language processing applications, for which 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.

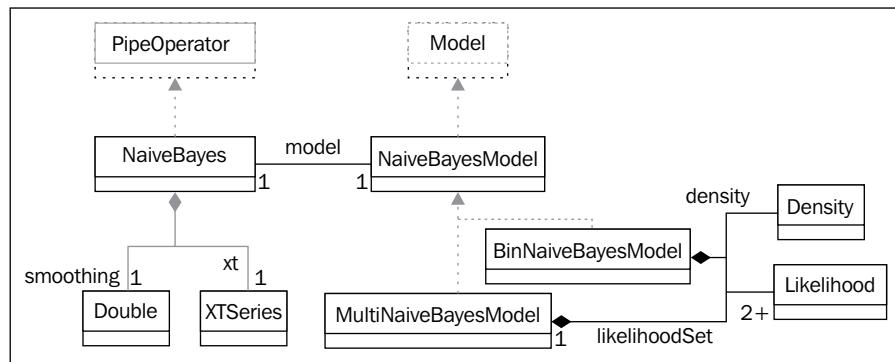
Software design

Our implementation of the Naïve Bayes classifier uses the following components:

- A generic model, `NaiveBayesModel`, of the type `Model`, which is initialized through training during the instantiation of the class.
- A model for the binomial classification, `BinNaiveBayesModel`, which subclasses `NaiveBayesModel`. The model consists of a density function of the type `Density`, and a pair of positive and negative `Likelihood` instances.
- A model for the multinomial classification `MultiNaiveBayesModel`.
- The predictive or classification routine is implemented as a data transformation extending the `PipeOperator` trait.
- The `NaiveBayes` classifier class has two parameters: a smoothing function such as Laplace and a labeled training set of the `XTSeries` type.

The principle of software architecture applied to the implementation of classifiers is described in the *Design template for classifiers* section in *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

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:

- The number of occurrences k of this features for $N > k$ observations in case of binary features or counters
- The mean value for all the observations for this features in the case of numeric or continuous features

It is assumed for the sake of this test case that the features, technical analysis indicators 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 multinomial Naïve Bayes. The first step is to define the likelihood for each feature using historical data. The `Likelihood` class has the following attributes:

- The label for the observation, `label`
- An array of tuple Laplace or Lidstone smoothed mean and standard deviation, `muSigma`
- The prior class `prior` that computes $p(c)$

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. The `Likelihood` class is defined as follows:

```
type Density = (Double*) => Double //1
type XYTSeries = Array[(Double, Double)]
val MINLOGARG = 1e-32
val MINLOGVALUE = -MINLOGARG
class Likelihood[T <% Double] (val label: Int, val muSigma: XYTSeries,
prior: Double) { //2
    def score(obs: Array[T], density: Density): Double =
        (obs, muSigma).zipped
            .foldLeft(0.0)((post, xms) => {
                val mean = xms._2._1
                val stdDev = xms._2._2
                val _obs = xms._1
                val prob = density(mean, stdDev, _obs)
                post + Math.log(if(prob < MINLOGARG) MINLOGVALUE else prob)
            }) + Math.log(prior) //3
}
```

The functions of the `Density` type compute the probability density for the values of a feature (line 1). The method takes an undefined number of arguments: the mean, the standard deviation, and the input value for the Gaussian distribution, the mean and input value {0, 1} for the Bernoulli distribution. The default probability density function is the normal distribution implemented by `Stats.gauss`.

The parameterized, view-bounded class, `Likelihood`, has two purposes:

- Define the model extracted from training (likelihood for each feature and the class prior) in the constructor (line 2)
- Compute the score of a new observation as part of the classification process score (line 3). The computation of the log of the likelihood uses a `density` method of the type `Density`, which is an argument of the `score` method. As seen in the next section, the density can be either a Gaussian or a Bernoulli distribution. The `score` method uses the Scala's `zipped` method to merge the observation values with the labeled output.

The next step is to define the `BinNaiveBayesModel` model for a two-class classification scheme. The two-class model consists of the 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 `NaiveBayesModel`, an abstract class that can be extended as needed to support both the Binomial and Multinomial Naïve Bayes models, as follows:

```
abstract class NaiveBayesModel [T <% Double] (density: Density) {  
    def classify(values: DblVector): Int  
}  
class BinNaiveBayesModel [T <% Double] (positives: Likelihood,  
negatives: Likelihood, density: Density) extends NaiveBayesModel [T] (density) {  
    override def classify(x: Array[T]): Int =  
        if (positives.score(x, density) > negatives.score(x, density)) 1  
        else 0  
}
```

The classification is executed by the `classify` method called by the `|>` operator in the Naïve Bayes classifier. It returns 1 for the class containing the positive cases and 0 for the negative.

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. (Refer to the *Design template for classifiers* section in *Appendix A, Basic Concepts*.)

The multinomial Naïve Bayes model, defined by the `MultiNaiveBayesModel` class is very similar to the `BinNaiveBayesModel` class:

```
class MultiNaiveBayesModel[T <% Double](likelihoodXs: List[Likelihood[T]], density: Density) extends NaiveBayesModel[T]
(density) {
  override def classify(x: Array[T]): Int =
    likelihoodXs.sortWith( (p1,p2) => p1.score(x, density) > p2.score(x, density)).head.label
}
```

The multinomial Naïve Bayes model differs from the binomial version in the following ways:

- The likelihood is defined as a list, `likelihoodXs` (one likelihood per class)
- The runtime classification sorts the class by the log likelihood (`sortWith`), selects the class with the highest `score`, and returns the class ID

Finally, the Naïve Bayes classifier is implemented by the `NaiveBayes` class. It implements the training and runtime classification using the Naïve Bayes formula. Any supervised learning model needs to be validated. 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 validation method, `validate`:

```
trait Supervised[T] {
  def validate(xt: XTSeries[(Array[T], Int)], tpClass:Int): Double
}
```

The `validate` method takes a labeled time series `xt` as an array of tuples (observation, class label) and the `tpClass` index that contains the true positives (that is, increase in the stock price) outcome. The method returns an F1-measure.

Besides inheriting the `Supervised` trait, the `NaiveBayes` class inherits the `PipeOperator` trait so that it can be integrated into a generic workflow as one of the computation units.

The attributes of the multinomial Naïve Bayes are as follows:

- The smoothing formula (Laplace, Lidstone, and so on): `smoothing`
- The labeled training set defined as a time series: `xt`
- The probability density function: `density`

The NaiveBayes class is defined as follows:

```
Class NaiveBayes[T <% Double] (smoothing: Double, xt:  
XTSeries[(Array[T], Int)], density: Density) extends PipeOperator[XTSe  
ries[Array[T]], Array[Int]] with Supervised[T] {  
  
    val model = BinNaiveBayesModel[T](train(1), train(0), density) //1  
    def train(label:Int)(implicit f: Array[T] => DblVector):  
        Likelihood[T] = { //2  
            val xi = xt.toArray  
            val values= xi.filter( _._2 == label).map(x => f(x._1) )  
            val dim = xi(0)._1.size  
            val vt = XTSeries[DblVector](values.toArray) //3  
            val muStdDev = statistics(vt).map(stat =>  
                (stat.lidstoneMean(smoothing, dim), stat.stdDev))  
            Likelihood(label, muStdDev, values.size.toDouble/xi.size) //4  
        }  
        ...  
}
```

The classifier uses the binomial Naïve Bayes model, `BinNaiveBayesModel` (line 1). The training process is implemented in the constructor by invoking the private `train` method (line 2). The method relies on an implicit conversion, `f: Array[T] => DblVector`, because of the `Array` type erasure. The main reason for this is to hide the details of the model and its training from the client code. We cannot assume that the user of the model is the same person as the creator of the model.

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 and trained). It is an elegant way to hide the details of the training of the model from the user.

The `train` method takes the labeled observations (observations or label) as input. The `vt` time series is extracted (line 3) and the likelihoods are calculated by counting the positive and negative labels, computing the mean, corrected with the Lidstone smoothing formula (line 4). The `lidstoneMean` method and standard deviation, `stdDev`, use the `statistics` method of the `XTSeries` singleton instance.

The `NaiveBayes` class also defined the runtime classification method `|>` and the F1-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.

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 likelihood class.

The Naïve Bayes classification using Gaussian distribution is illustrated using two classes, c_1 and c_2 , and a model with two features (x and y):

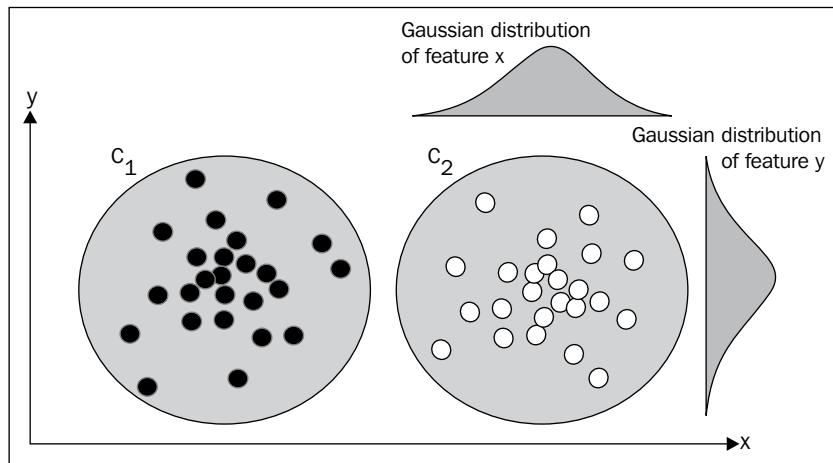


Illustration of the Gaussian Naive Bayes using a 2-dimensional model

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].

Naïve Bayes classification using Gaussian density

For a Lidstone or Laplace smoothed mean μ' and a standard deviation σ , the log likelihood of a posterior probability is defined as:

$$\log p(C_j | x) = \sum_{i=0}^{N-1} \log \left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i - \mu')^2}{2\sigma^2}} \right) + \log p(C_j)$$



In this example, we used the Gaussian distribution as our probability density function as defined in the `Stats` object, which was introduced in *Chapter 2, Hello World!*. The implementation of the computation of the **Gaussian** probability density is quite simple, shown as follows:

```
object Stats {
    final val INV_SQRT_2PI = 1.0/Math.sqrt(2.0*Math.PI)
    def gauss(mu: Double, sigma: Double, x:Double) : Double = {
        val y = x - mu
        INV_SQRT_2PI/sigma * Math.exp(-0.5*y*y/sigma*sigma)
    }
    def gauss(x: Double*): Double = gauss(x(0), x(1), x(2))
    ...
}
```

The second version of the Gaussian density is required to handle the `Density` type: `(Double, Double, Double) => Double`.

Finally, the classification method is implemented as the pipe operator `|>` of the `NaiveBayes` class. The classification model and the density function are provided at runtime as attributes of the class:

```
def |> : PartialFunction[XTSeries[Array[T]], Array[Int]] = {
    case xt: XTSeries[Array[T]] if (xt != null && xt.size > 0 && model != None) => xt.toArray.map( model.classify( _) )}
```

Labeling

The most critical element in the training of a supervised learning algorithm is the creation of labeled data. Fortunately, in this case, the label (or expected class) 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 average price, volume, and volatility over the last n days.

The first step is to extract the average price, volume, and volatility 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.

First, the `extractor` function extracts the closing, high, and low prices, and volume for each trading day, using the `toDouble` and `%` operators described in the *Data extraction* and *Data sources* section in *Appendix A, Basic Concepts*, as follows:

```
val extractor = toDouble(CLOSE)    //stock closing price
      :: ratio(HIGH, LOW) //volatility (HIGH-LOW)/HIGH
      :: toDouble(VOLUME) //daily stock trading volume
      :: List[Array[String] => Double] ()
```

Secondly, the data source extractor outputs the four statistics for each stock (line 1) for which the average for a window period is computed (line 3) using a simple moving average `mv` (line 2):

```
val xs = DataSource(symbol, path, true) |> extractor //1
val mv = SimpleMovingAverage(period) //2

val ratios = xs.map(x => { //3
  val xt = mv get x, toArray
  val zValues = x.drop(period).zip(xt.drop(period))
  zValues.map(z => if(z._1 > z._2) 1 else 0).toArray //4
})
var prev = xs(0)(period)
val label = xs(0).drop(period+1).map(x => { //5
  val y = if(x > prev) 1 else 0
  prev = x; y
}).toArray
ratios.transpose.take(label.size).zip(label) //6
```

The Scala's `drop` method is used to shift the time series to compute the average of the three variables: price, `toDouble(CLOSE)`; volume, `toDouble(VOLUME)`; and volatility, `ratio(HIGH, LOW)` (line 4). The labeled data, direction of the price action for the next trading day, is added to the three ratios (line 5). Finally, the array is transposed to extract the list of tuples (list of UP/DOWN values for each feature and price direction for next trading day/labeled data) (line 6).

The labeled data extracted from the `input` CSV file is used in the training and validation of the time series using the Naïve Bayes classifier:

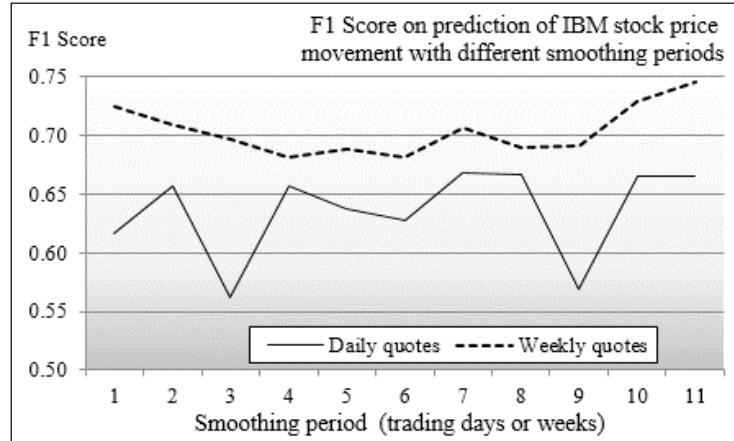
```
val trainValidRatio = 0.8
val period = 10

val labels = XTSeries[(Array[Int], Int)](input.map(x =>
  (x._1.toArray, x._2)).toArray) //7
val numObsToTrain = (trainValidRatio*labels.size).floor.toInt //8
val nb = NaiveBayes[Int](labels.take(numObsToTrain)) //9
validate(labels.drop(numObsForTrains+1), nb) //10
```

The original labeled dataset, `labels`, is split between training and validation labeled data (line 7) using the `trainValidRatio` ratio (line 8). The `NaiveBayes` constructor initializes the model through training (line 9). Finally, the `validate` method returns the F1 measure for the validation test (line 10).

Results

The next chart plots the value of the F1 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 Bayesian prediction, using the value of stock price, volatility, and volume relative to their average over the averaging period.

From this experiment, we conclude that:

- The prediction of the stock movement using the average price, volume, and volatility is not very good. The F1 measure 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.

Multivariate Bernoulli classification

The previous example uses the Gaussian distribution for features that are essentially binary, $\{UP=1, DOWN=0\}$, 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 Naïve Bayes classifier in that it penalizes the features x , which do not have any observations; the Naïve Bayes classifier ignores them [5:10].

The Bernoulli mixture model

For a feature function f_i , with $f_i = 1$ if the feature is observed, and a value of 0 if the feature is not observed:

$$p(f_i | C_j) = \prod_{k=0}^{n-1} \left(f_k \cdot p(x_k | C_j) + (1 - f_k) \cdot (1 - p(x_k | C_j)) \right)$$

Implementation

The implementation of the Bernoulli model consists of modifying the `Likelihood`.
`score` scoring function by using the Bernoulli density 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 mathematical formula. The second version uses the signature of the *Density* (`Double*`) \Rightarrow `Double` type.

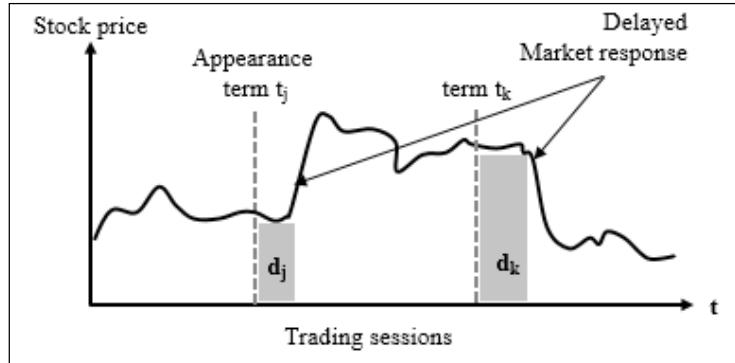
The mean value is the same as in the Gaussian version. The binary feature is implemented as an `Int` type with the value `UP` = 1 (with respect to `DOWN` = 0) for the upward (with respect to downward) direction of the financial technical indicator.

Naïve Bayes and text mining

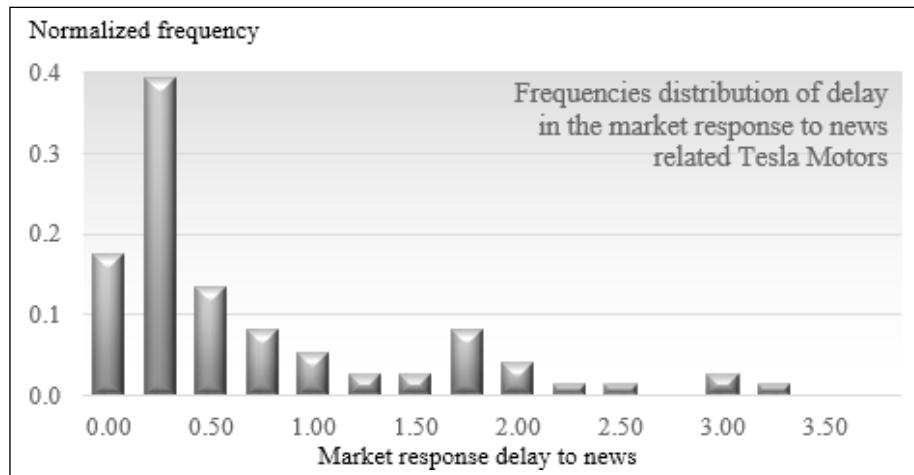
The multinomial Naïve Bayes classifier is particularly suited for **text mining**. Naïve Bayes is used to classify the following entities:

- E-mails as legitimate versus spam
- Business news stories
- Movie reviews and scoring
- Technical papers as per field of expertise

This third use case consists of predicting the direction of a stock, Tesla Motors Inc, (ticker symbol: TSLA) give the financial news. The features are the frequency of occurrence of some specific terms related to the stock. It is unclear how fast the investor or trader reacts to the news and influence, if any, of the value of a stock. Therefore, the delayed response time, as depicted in the following chart, should be a feature of the proposed model:



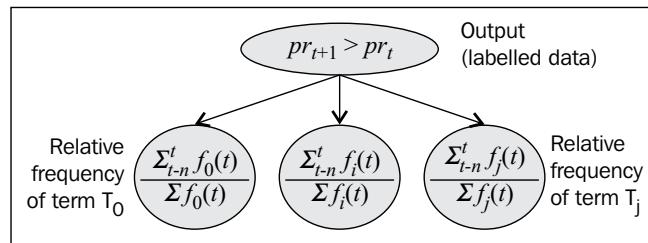
The feature market response delay would play a role in the training, only if the variance of the observations is significant. The distribution of the frequencies of the delay in the market response to any newsworthy articles regarding TSLA shows that the stock prices react within the same day in 82 percent of the case, as seen here:



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 till the following Monday to impact the stock price. The second challenge is to assign any shift of stock price to a specific news release, taking into account that some news can be redundant and simultaneous.

Therefore, the model features for predicting the stock price, pr_{t+1} , are the relative frequency, f_r , of 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 direction of the stock price between two consecutive trading sessions t and $t+1$, given the relative frequency of appearance of some terms in the media:



The Bayesian model for the prediction of 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 model for extracting relevant terms and computing their relative frequency. The following 10-step sequence of actions describe one of numerous methodologies 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 (such as 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 **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 searching, 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 8 simple steps:

1. Extracting all news with a reference to a specific stock or company in the news feed.
2. Extracting the timestamp or date of the article using a regular expression.
3. Grouping all the news articles related to the stock for a specific date t into a document \mathcal{D}_t .
4. Ordering the documents \mathcal{D}_t as per the timestamp.
5. Extracting the terms $\{T_{i,D}\}$ from each sentence of the document \mathcal{D}_t and ranking them by their relative frequency.
6. Aggregating the terms $\{T_{t,i}\}$ for all the documents sharing the same release date t .
7. Computing the relative frequency, rtf , of each term, $\{T_{t,i}\}$, 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.
8. Normalizing the relative frequency for the average number of articles per date, $nrtf$.

The relative term frequency for term t_i with n_{ia} occurrences in article a released on the date D_t is given as:

$$ntrf(t_i) = \frac{\sum_{a \in D_t} n_i^a}{\sum_{a \in Corpus} n_i^a}$$



The relative term frequency normalized by the average number of articles per day, Na/D is given as:

$$ntrf(t_i) = \frac{rnf(t_i)}{Na/D}$$

Extraction of terms

First, let's define the features set for the financial terms as the `NewsArticles` class parameterized for the date type T . For the sake of simplicity, the type of date value is explicitly viewbounded to `Long`. The `NewsArticles` class is a container of the news articles and press releases relevant to a specific stock. At its core, a news article is defined by its release or publication, and the list of tuple of terms and their relative frequency. The `NewsArticles` class is defined as follows:

```
@implicitNotFound("NewsArticles. Ordering not explicitly defined")
class NewsArticles[T <% Long] (implicit val order: Ordering[T]) {
    val articles = new HashMap[T, Map[String, Double]]
    ...
}
```



The `@implicitNotFound` annotation

I recommend using the `implicitNotFound` annotation for every implicit class and method parameter. A declaration may be obvious to one software developer but not obvious to another developer.

The `NewsArticles` class uses the mutable `HashMap` data structure to manage the set of articles. An article is defined by:

- Its release date (type T)
- Its map of tuples {term contained in the article, relative frequency (or weight) of the term}, `wTerms`

The weight of a term is computed as the ratio of the number of occurrences of this term in the article, to the total number of occurrences in the entire corpus of articles related to the stock.

The implicit Ordering class parameter is required for sorting.

The map articles is populated with the overloaded operator `+=`:

```
def += (date: T, wTerms: Map[String, Double]): Unit = { //1
  def merge(m1: Map[String, Double], m2: Map[String, Double]): Map[String, Double] = { //2
    (m1.keySet ++ m2.keySet).foldLeft(new HashMap[String, Double])((m,
    x) => {
      var wt = 0.0
      if (m1.contains(x)) wt += m1(x)
      if (m2.contains(x)) wt += m2(x)
      m.put(x, wt)
    })
  }
  articles.put(date, if (articles.contains(date))
    merge(articles(date), wTerms) else wTerms) //3
}
```

The `+=` method adds new sets (mutable hash map) of pairs (terms, relative frequency), `wTerms`, released at a specific date, to the existing map of news articles (line 1). The terms related to different articles from the same date are merged using the local merge function (line 2). Finally, the list of key-value pairs (term, frequency) is ordered by their timestamp of the type `T`.

The second method, `toOrderedArray`, consists of ordering the articles per their release date:

```
def toOrderedArray: Array[(T, Map[String, Double])] = articles.
  toArray.sortWith(_._1 < _._1)
```

Scoring of terms

The scoring of the terms is actually performed by the `TermssScore` class, parameterized by date and the `score` method:

```
class TermssScore[T <% Long](toDate: String => T, toWords: String =>
  Array[String], lexicon: Map[String, String])(implicit val order:
  Ordering[T]) {
  def score(corpus: Corpus): Option[NewsArticles[T]]
}
```

The TermsScore class parameterized for the type of release date has three parameters:

- A `toDate` function to extract the date from each news article. The function can be implemented as a regular expression or a group of regular expressions.
- A `toWords` function to extract the nonstop terms from the content of the article. The function can be quite elaborate, as described in the previous section. It may require creating classifiers to extract sentences, n-grams, and tags.
- A `lexicon` function that simulates the lemmatization and stemming of the most common terms. The `lexicon` function is implemented as a map that attaches a semantic equivalent to each term as a poor man's lemmatization. For example, "China", "Chinese", and "Shanghai" are semantically associated to the term "China".

The type for date `T` is view bounded by the `Long` type because it is assumed that any date can be potentially converted into time in milliseconds. The `Ordering[T]` class is provided as an implicit attribute to order the news articles as per their release date.

The relative frequency of a term t is computed arbitrarily, as the ratio of the number of occurrences of t for a specific date to the total number of terms.

Let's look at the scoring method:

```
type Corpus = (String, String, String) //1
def score(corpus: Corpus): Option[NewsArticles[T]] = { //2
    val docs = rank(corpus)

    val cnts = docs.map(doc => (doc._1, count(doc._3))) //3
    val totals = cnts
        .map(_._2) //4
        .foldLeft(Counter[String])((s,cnt)=>s ++ cnt)
    val articles = NewsArticles[T]
    cnts.foreach(cnt =>articles +=(cnt._1, (cnt._2/totals).toMap))
    articles
    ...
}
```

The `score` method processes the training set or corpus of the news articles related to a stock and returns a set of `NewsArticles` instances.

The `Corpus` type (line 1) defines the three essential components of a news article: a timestamp, a title, and a body or content. The `rank` method (line 2) extracts the release date from each news article and orders them as per increasing date.

The frequency of terms is computed for each document or group of news articles associated with a date (line 3) using the `count` method. The `count` method matches each term extracted from the news article to the entries of the lexicon map. The counters of the `Counter: Map[String, Int]` type collect the number of occurrences of each term. The next instruction (line 4) aggregates the counts for the entire corpus that is used to compute the relative frequencies (line 5).

The `rank` method uses a sequence of Scala methods `map` and `sortWith` to order the articles as per date (line 6):

```
def rank(corpus: Corpus): Option[CorpusType[T]] = {
    corpus.map(doc => (toDate(doc._1.trim), doc._2, doc._3))
        .sortWith( _._1 < _._1) //6
}
```

The scoring method is protected by a Scala exception handler (line 7). Finally, the `count` method matches a term with an entry in the lexicon and updates the count if a match is found (line 8):

```
def count(term: String): Counter[String] =
    toWords(term).foldLeft(new Counter[String])((cnt, w) =>
        if( lexicon.contains(w) ) cnt + lexicon(w) //8
        else cnt
    )
```

Testing

For testing purpose, let's select the news articles mentioning Tesla Motors and its ticker symbol TSLA over a period of two months.

Retrieving textual information

First, you need to define the three parameters of the scoring `TermssScore` class: `toDate`, `toWords`, and `lexicon`.

The private `toDate` method converts a string into a date defined as a `Long` data type:

```
def toDate(date: String): Long = {
    val idx1 = date.indexOf(".")
    val idx2 = date.lastIndexOf(".")
    if(idx1 != -1 && idx2 != -1)
        (date.substring(0, idx1) + date.substring(idx1+1, idx2)).toLong
    else -1L
}
```

The `toWords` method uses simple regular expressions, `regExpr`, to replace any punctuation into a `.` character (line 1), used as a word delimiter (line 2). All words shorter than three characters are discounted (line 3):

```
def toWords(txt: String): Array[String] = {
    val regExpr = "[',|.|.|?|!|:|\\"]"
    txt.trim.toLowerCase
        .replace(regExpr, "&@") //1
        .split("&@") //2
        .filter(_.length > 2) //3
}
```

Finally, the lexicon contains the terms that need to be monitored. In this particular period of time, the news media were looking for any announcement regarding Tesla Motors' foray into the Chinese market, issues with the batteries, and any plan to deploy electrical vehicle charger stations. The set of terms regarding these issues is limited, and therefore, the lexicon can be built manually:

```
val LEXICON = Map[String, String] (
    "tesla"->"Tesla", "tsla"->"TSLA", "china"->"China", "chinese"->
    "China", ....)
```

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.

The client code to train and validate the model executes the entire workflow, from extracting and scoring the news articles and press releases to generating the normalized labeled data and computing the F1 measure.

The output (or labeled data) `TSLA_QUOTES` consists of the stock price for Tesla Motors:

```
val TSLA_QUOTES = Array[Double] (250.56, 254.84, ... )
```

The first step is to load and clean all the articles (`corpus`) defined in the pathname directory (line 1). This task is performed by the `DocumentsSource` class (described in the *Extraction of documents* section under *Scala programming in Appendix A, Basic Concepts*):

```

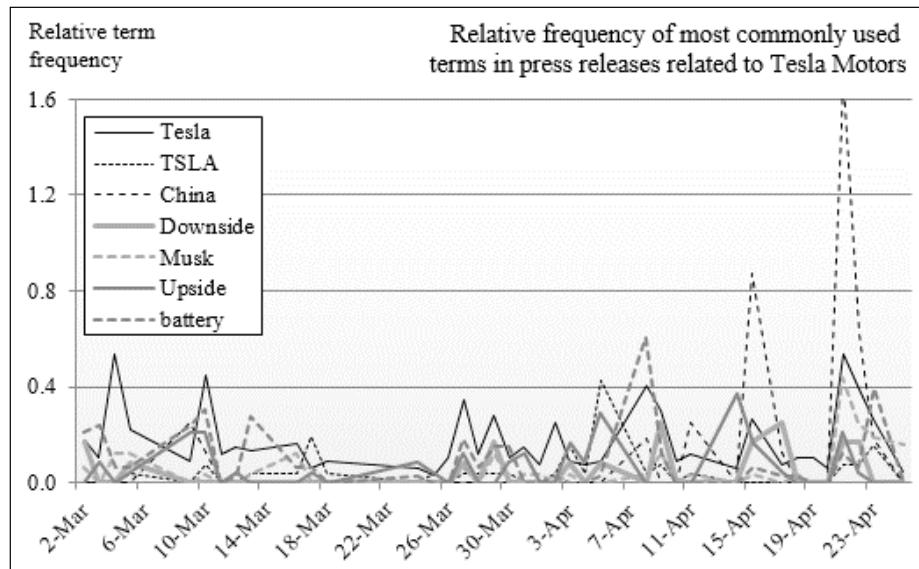
val corpus: Corpus = DocumentsSource(pathName) |> { //1
  val ts = new TermsScore[Long](toDate, toWords, LEXICON)
  ts.score(corpus) match { //2
    case Some(terms) => {
      var prevQ = 0.0
      val diff = TSLA_QUOTES.map( q => {
        val delta = if(q > prevQ) 1 else 0
        prevQ = q; delta
      })
      val columns = LEXICON.values.foldLeft(new HashSet[String])((hs,
        key) => {hs.add(key); hs}).toArray
      val fqLabels = terms.toOrderedArray //3
        .zip(diff) //4
        .map( x => (x._1._2, x._2))
        .map(lbl =>(columns //5
          .map(f =>if( lbl._1.contains(f) ) lbl._1(f)
            else 0.0), lbl._2))
      val xt = XTSeries[(Array[Double], Int)](fqLabels)
      val nb = NaiveBayes[Double](xt) //6
      ...
    }
  }
}

```

Next, the `TermsScore.score` method extracts and scores the more relevant terms from the corpus, using the normalized relative frequency defined in steps 7 and 8 of the information retrieval process (line 2). The `terms` are then ordered by date (line 3) and zipped with the labels (direction of the next trading day's stock price) (line 4). The lexicon is used to generate the final labeled observations (*features = terms relative frequency, label= direction of stock price*) (line 5). Finally, the model is built by invoking the `NaiveBayes.apply` constructor (line 6), which consists of running the algorithm through the training set.

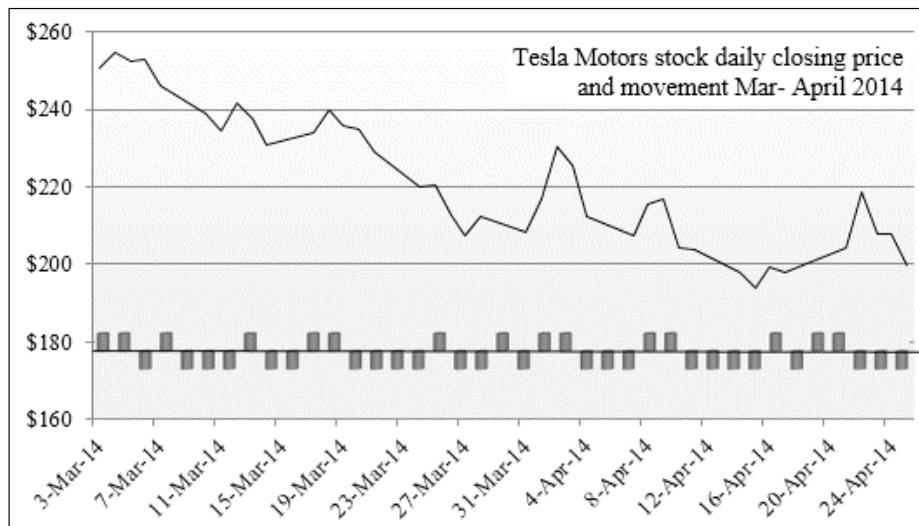
Evaluation

The following chart describes the frequency of occurrences of some of the terms related to either Tesla Motors or its stock ticker TSLA:



Plot of the relative frequency of a partial list of stock-related terms

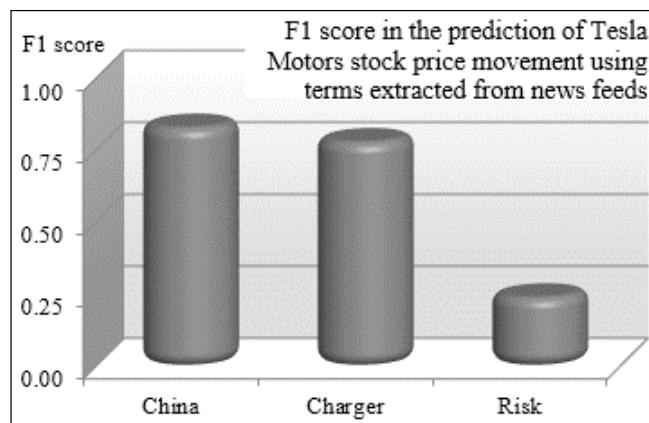
The next chart plots the labeled data, which is the direction of the stock price for the day following the press release(s) or news article(s):



Plot of the stock price and movement for Tesla Motors stock

This chart displays the historical price of the stock TSLA with the direction (UP or DOWN). The classification of 15 percent of the labeled data selected for validation has an F1 measure of 0.71. You need to keep in mind that no preprocessing or clustering was performed to isolate the most relevant features/keywords. The keywords were selected according the frequency of their occurrence 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 F1 score for a validation for which only the observations that have a high number of occurrences of a specific keyword are used, as shown here:



Bar chart representing predominant keywords in predicting TSLA stock movement

The 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 75 percent. The terms under the category **Risk** 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 more elaborate classifiers. However, it should not be regarded as the primary methodology for the features selection, but instead as a by-product of the Naïve Bayes in case a very small number of features (less than 10 percent) are predominant in the model. This result can always be validated by computing the principal components, for which the normalized cumulative variance (eigenvalues) of the most predominant features is 90 percent or more.

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.

Naïve Bayes classifiers are simple and robust generative classifiers that rely on prior conditional probabilities to extract a model from a training dataset. The Naïve Bayes has its benefits, as mentioned here:

- Simple implementation and easy to parallelize
- Very low computational complexity: $O((n+c)*m)$, where m is the number of features, C the number of classes, and n the number of observations
- Handles missing data
- Supports incremental updates, insertions, and deletions

However, Naïve Bayes is not a silver bullet. It has the following disadvantages:

- The assumption of the independence of features is not practical in the real world
- It requires a large training set to achieve reasonable accuracy
- It contains a zero-frequency problem for counters

Summary

There is a reason why the Naïve Bayes model is the first supervised learning technique you learned: 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 document classification, Naïve Bayes assumes incorrectly 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 attempt to address some of the Naïve Bayes's disadvantages [5:14].

However, this chapter does not address 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 the next chapter.

6

Regression and Regularization

In the first chapter, we briefly introduced the binary logistic regression (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 of *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. Linear regression can be implemented using the least squares method. Practically, the least squares method entails the minimization of the sum of the squares of the error between the observed data and the actual model.

The least squares problems fall into 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 $\{x, y\}$.

Single variable linear regression is given by the following formula:

$$\hat{w} = \arg \min_{w,r} \sum_{j=0}^{n-1} (y_j - f(x_j | w))^2 \quad f(x | w) = w_0 + w_1 \cdot x$$

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 parameterized class `SingleLinearRegression[T]` to implement the formula described in the previous section. The class implements the data transformation `PipeOperator` (refer to the *Design template for classifiers* section in *Appendix A, Basic Concepts*).

```
class SingleLinearRegression[T <% Double] (xt: XTSeries[(T, T)])
(implicit g: Double => T) extends PipeOperator[Double, T] {
    type XY = (Double, Double)
    ...
}
```

 **Model instantiation**

The model parameters are computed through training and the value 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. (Refer to the *Design template for classifiers* section *Appendix A, Basic Concepts.*)

The application code must provide an implicit conversion `g` from `Double` to the class type parameter, `T`. The training generates the model defined as the regression weights, the tuple (slope, intercept), in the case of single variable linear regression:

```
val model: Option[XY] = {
    val data = xt.toArray
        .map(x => Array[Double](x._1, x._2)) //1
    val regr = new SimpleRegression(true)
    regr.addData(data) //2
    Some((regr.getSlope, regr.getIntercept)) //3
}
```

The tuple of regression weights or coefficients for the model are computed using the `SimpleRegression` class from the `stats.regression` package of the Apache Commons Math library. The time series is converted to a matrix of double values, `data` (line 1), which is used to initialize the instance of `SimpleRegression` (line 2). The model is initialized with the slope and intercept computed during the training (line 3).

 **private vs. 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 `this` instance. For example, the value `model` can be accessed only by `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 (the ticker symbol: CU) over a period of 6 months (January 1, 2013 to June 30, 2013):

```
val price = DataSource(path, false, true, 1) |> adjClose //1
val xy = price.zipWithIndex
    .map(x => (x._2.toDouble, x._1.toDouble)) //2

val linRegr = SingleLinearRegression(xy) //3
val w1 = linRegr.slope
```

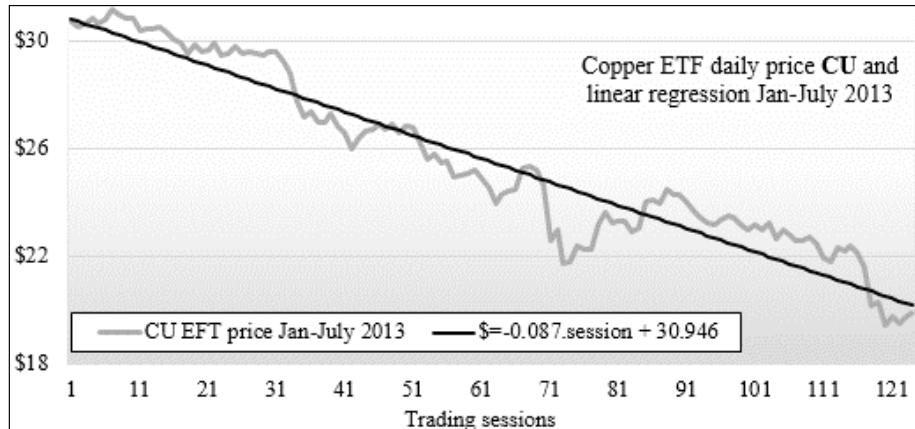
```
val w0 = linRegr.intercept
if( w1 != None ) //4
    Display.show(lsErr(xy.toArray, w1.get, w0.get), logger)
...

```

The closing price for the CU ETF is extracted from a CSV file (line 1) using a `DataSource` instance (refer to the *Data extraction* section *Appendix A, Basic Concepts*). The 2-dimension time series is generated by converting the indexes of the time series into the `x` values using the `zipWithIndex` Scala method (line 2). The regression model, `linRegr`, is trained during instantiation of the `SingleLinearRegression` class (line 3). Once the model is created successfully, the least squared error `lsErr` of the predicted values and the actual values is computed, as follows:

```
def lsErr(xyt: Array[XY], w1: Double, w0: Double): Double =
    Math.sqrt(xyt.foldLeft(0.0)((err, xy) => {
        val diff = xy._2 - w1*xy._1 - w0; err + diff*diff
    })/xyt.size)
```

The original stock price and the linear regression equation are plotted in the following chart:



Single variable linear regression – Copper ETF daily price

Although the single variable linear regression is convenient, it is limited to scalar time series. Let's consider the case of multiple variables.

Ordinary least squares (OLS) 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).

Minimization of the loss function is given by the following formula:

$$\hat{w} = \arg \min_w \sum_{i=0}^{n-1} (y_i - f(x_i | w))^2 \quad f(x | w) = \sum_{d=1}^{D-1} w_d x_d$$

where $w_{j=0,D}$ is the D regression (or model) parameters (or weights), $(x_i, y_i)_{i:0,n-1}$ is n observations of vector x and output value y, and f is the linear multivariate function, $y = f(x_0, x_1, \dots, x_d, \dots)$.

There are several methodologies to minimize the residual sum of squares (RSS) 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 vector of d dimension (d features) with $n > d$ [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 *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 following implementation of the least squares regression leverages the Apache Commons Math library implementation of the ordinary least squares regression [6:6].

Let's create a class, `MultiLinearRegression`, which inherits the implementation of the ordinary least square computation of the Apache Commons Math library `OLSMultipleLinearRegression`. The class is defined as a data transformation implementing the `PipeOperator`, as follows:

```
class MultiLinearRegression[T <% Double] (xt: XTSeries[Array[T]],  
y: DblVector) extends OLSMultipleLinearRegression with  
PipeOperator[Array[T], Double]
```

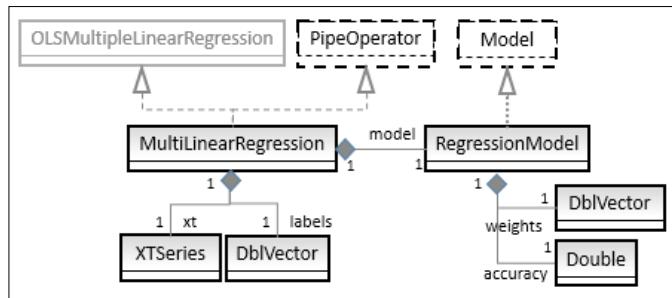
The parameterized class takes the following two parameters:

- The time series of the variables vector `xt` (input matrix)
- The labeled output values, `y`, used in training

The model for the linear regression is defined by its `weights` (or parameters) and its residual sum of squares, `rss`. The RSS is included in the model because it provides the client code with important information regarding the accuracy of the underlying technique used to minimize the loss function:

```
case class RegressionModel(val weights: DblVector, val rss: Double)
```

The relationship between the different components of the least squares regression is described in the following UML class diagram:



Implementation

The training is performed during the instantiation of the class `MultiLinearRegression` (refer to the *Design template for classifiers* section in *Appendix A, Basic Concepts*):

```
val model: Option[RegressionModel] = {  
    newSampleData(labels, xt.toDblMatrix) //1  
    val weights = estimateRegressionParameters  
    val wRss = (weights, calculateResidualSumOfSquares) //2  
    Some(RegressionModel(wRss._1, wRss._2))  
}
```

The least squares algorithm is initialized with the feature observations, `xt`, and the target data, `labels`, using the `newSampleData` method of `OLSMultipleLinearRegression` (line 1).

The model weights are retrieved using `estimateRegressionParameters` (similarly, `rss` using `calculateResidualSumOfSquares`) (line 2).

Exception handling



Wrapping up invocation of methods in a third party with a Scala exception handler matters for a couple of reasons: it makes debugging easier by segregating your code from the third party and it allows your code to recover from the exception by re-executing the same function with alternative third-party library methods, whenever possible.

The predictive algorithm for the ordinary least squares regression is implemented by the data transformation `|>`. The method predicts the output value given `model` and an input value `x`:

```
def |> : PartialFunction[Feature, Double] = {
  case x: Feature if(model!=None && x.size==model.get.size-1) =>{
    val w = model.get.weights
    x.zip(w.drop(1)).foldLeft(w(0))((s, z) => s + z._1*z._2)
  }
}
```

The predictive value is computed by zipping the weight w_1 to w_n with the input vector x and then folding the zipped array.

Test case 1 – trending

Trending consists of extracting the long-term movement in a time series. Trend lines can be identified 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 trading sessions y . The volume, volatility, and price variation for CU between January 1, 2013 and June 30, 2013 are plotted in the following chart:

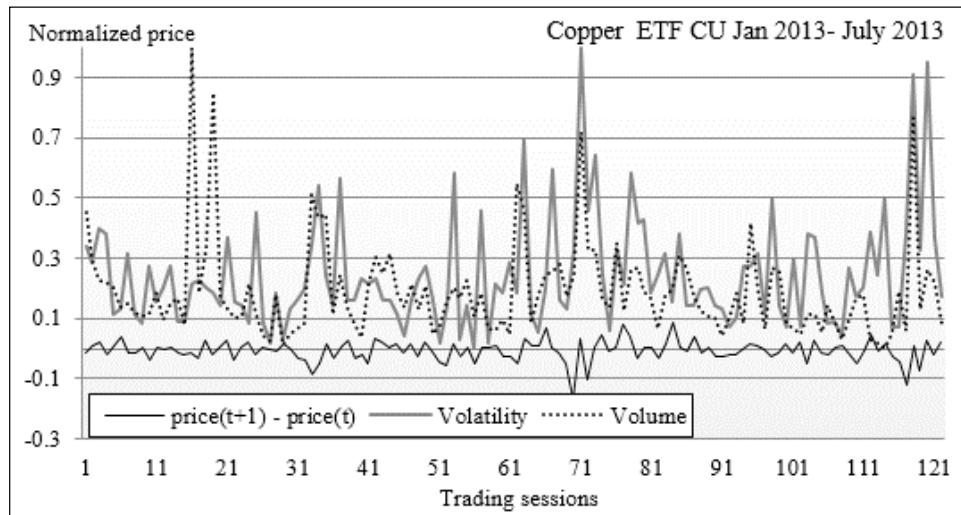


Chart for price variation, volatility, and trading volume for Copper ETF

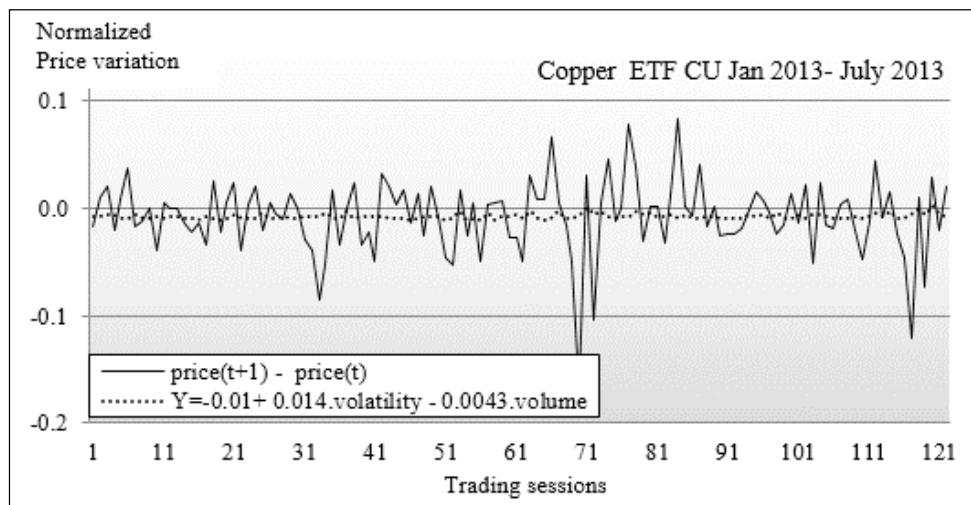
Let's write the client code to compute the multivariate linear regression, $price\ change = w_0 + volatility.w_1 + volume.w_2$:

```
val path = "resources/data/chap6/CU.csv"
val src = DataSource(path, true, true, 1) //2
val price = (src |> YahooFinancials.adjClose).toArray //1
val volatility = src |> YahooFinancials.volatility //1
val volume = src |> YahooFinancials.volume //1

val deltaPrice = price.drop(1)
    .zip(price.take(price.size -1))
    .map(z => z._1 - z._2) //3
val data = volatility.zip(volume)
    .map(z => Array[Double](z._1, z._2))
val features = XTSeries[DblVector](data.dropRight(1)) //4
val regression = MultiLinearRegression[Double](features, deltaPrice)
//5
regression.weights match {
  case Some(w) => Display.show(w, logger)
  ...
}
```

The daily session adjusted closing price, the session volatility, and the session volume for the CU ETF is extracted from a CSV file (line 1) using the `DataSource` transformation (line 2). The array, `priceChange`, which is the daily price change between two consecutive trading sessions is computed by duplicating, shifting, and zipping the session closing prices (line 3). The features are computed by zipping volatility and the volume time series (line 4). The regression model is trained by instantiating the `MultiLinearRegression` class (line 5) and the model weights are displayed using an auxiliary `display` method (to the logger or standard output) (line 6).

The original price change time series and the data predicted by the regression are plotted in the following chart:



Price variation and the least squares regression for 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:

$$\text{price}(t+1) - \text{price}(t) = -0.01 + 0.014 \cdot \text{volatility} - 0.0042 \cdot \text{volume}$$

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 discrete Fourier transform or the Kalman filter for dynamic systems (refer to *Chapter 3, Data Preprocessing*) [6:7].

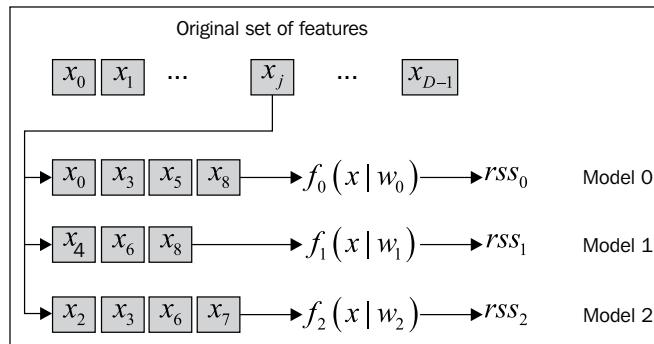
Although trend detection is an interesting application of the least squares regression, the method has limited filtering capabilities for time series [6:8]:

- It is sensitive to outliers
- It put a greater weight to the first and last few observations that need to be discarded
- As a deterministic method, it does not support noise analysis (distribution, frequencies, and so on)

Test case 2 – features selection

The second test case is related to features 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 (RSS) on the training set.

Let's consider an initial set of D features $\{x_i\}$. The objective is to estimate the subset of features $\{x_i^d\}$ that are the most relevant to the set of observations using a least squares regression. Each subset of features is associated to an $f_j(x | w_j)$ model:



The OLS can be used to select the model parameters w if the original set of features is small. Performing the regression of each subset of a large original features set is not practical.

[

The features selection can be expressed mathematically as follows:

$$\check{f} = \arg \min_{f_j} \left\{ \sum_{i=0}^{n-1} (y_j - f(x_j | w))^2 \right\} \quad f(x | w) = w_{j0} + \sum_{d=1}^{D_j-1} w_{jd} x_d$$

]

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

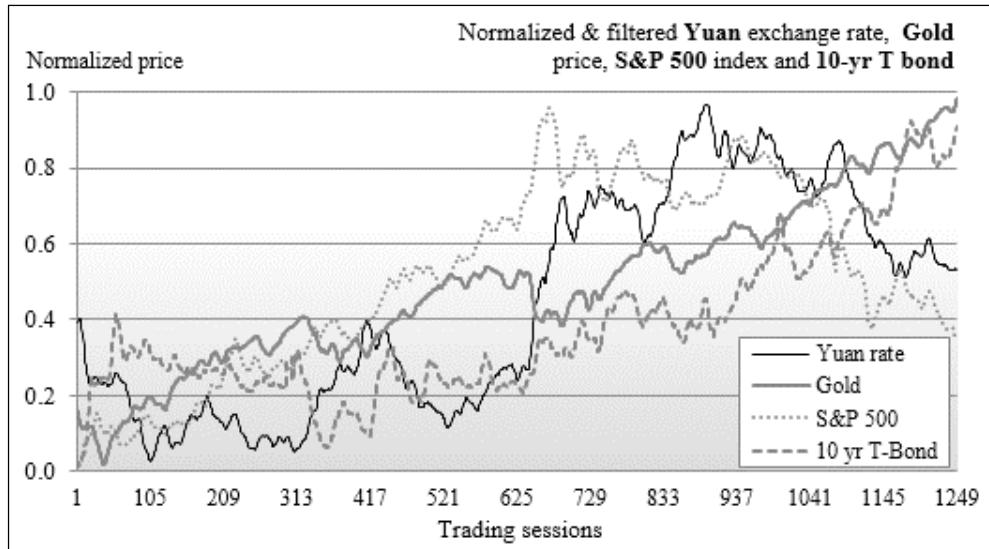
The problem is to estimate which combination of the three variables S&P 500 index, gold price, and 10-year treasury bond price is the most correlated to the exchange rate of the Yuan. For practical reasons, we use the Exchange Trade Funds CYN as the proxy for the Yuan/US dollar exchange rate (similarly, SPY, GLD, and TLT for S&P 500 index, the spot price of gold, and the 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 (gradient descent, genetic algorithm, and so on) using $1/\text{RSS}$ as the objective function to be maximized.

The number of models to evaluate is relatively small, so an ad hoc approach to compute the RSS for each combination is acceptable. Have a look at the following graph:



Graph of the Chinese Yuan exchange rate, gold, 10-year treasury bond price, and S&P 500 index

The `getRSS` method implements the computation of the RSS value given a set of observations `xt` and labeled values `y`:

```
def getRSS(xt: XTSeries[DblVector], y: DblVector): String = {
    val regression = MultiLinearRegression[Double](xt, y) //1
    val buf = new StringBuilder
    regression.weights.get
        .zipWithIndex //2
        .foreach(w => {
            if(w._2 == 0) buf.append(w._1)
            else buf.append(s" + ${w._1}.x${w._2}") //3
        }
    buf.append(s"RSS: ${regression.rss.get}").toString
}
```

The `getRSS` method merely trains the model by instantiating the multilinear regression class (line 1), indexes the array of weights (line 2), and creates a text representation of the linear regression equation (line 3).

Once the regression model is trained during the instantiation of the `MultiLinearRegression` class, the coefficients of the regression weights and the RSS value are printed. The `rss` method is invoked for any combination of the variables `ETF`, `GLD`, `SPY`, and `TLT` against the label `CNY`:

```
val symbols = Array[String] ("CNY", "GLD", "SPY", "TLT")
val smoothingPeriod = 16
val movAvg = SimpleMovingAverage[Double](smoothingPeriod) //4

val input= symbols.map(s=>DataSource(path+s+".csv",true,true, 1))
    .map( _ |> YahooFinancials.adjClose ) //5
    .map(x=> movAvg |> XTSeries[Double](x))

val features = input.drop(1)
val featuresList = List[(String, DblMatrix)](
    ("CNY=f(SPY,GLD,TLT)", features.map( _.toArray ).transpose), //6
    ("CNY=f(GLD,TLT)", features.drop(1).map( _.toArray ).transpose),
    ...
)
featuresList.foreach(x => Display.show(x._1 +
    getRSS(XTSeries[DblVector](x._2), input(0)), logger)) //7
```

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, `movAvg` (line 4). The time series are extracted from CSV files using the `DataSource` class, then smoothed using a sequence of `Array.map` invocations (line 5). The first map extracts the content of the files associated to the stock ticker symbol, assuming that the names of the files are formatted as `path/symbol.csv`.

For the sake of simplicity, the option type returned by the pipe operator is not validated.

The first model using the three variables `SPY`, `GLD`, and `TLT` is created by transposing them by the `xt.size` matrix (line 6). The RSS value is computed by invoking the `rss` method (line 7). The second model using two variables, `SPY` and `TLT`, is created by filtering out the `GLD` time series. The process is repeated for all other models. Have a look at 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

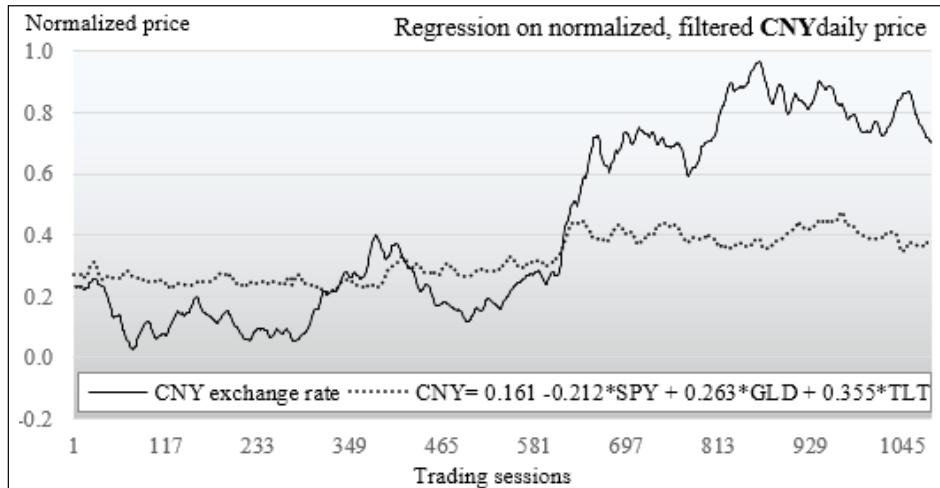
CNY = f(SPY)
0.20238901847764892 + 0.1251591898720694.x1
RSS: 4.7291908591838405

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:



Ordinary least regression on the Chinese Yuan ETF (CNY)

The regression model smoothed the original CNY time series. It weeded out all but the most significant price variation.

However, 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 **r^2 statistics**. The r^2 value is a number that indicates how well data fits a statistical model.

RSS and r^2 statistics are related by the following formulae:

 $r^2 = 1 - \frac{RSS}{TSS}$ $TSS = \sum_{i=0}^{n-1} (y_j - \bar{f}(x|w))^2$ $\bar{f} = \sum_f f_j$

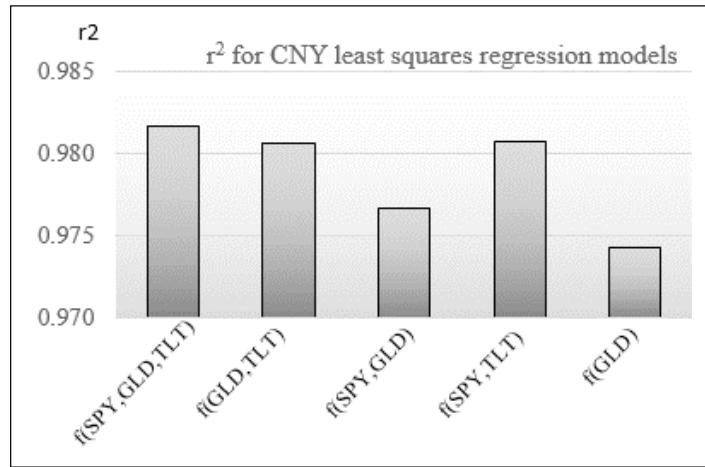
The implementation of the computation of the r_2 statistics is fairly simple. For each model f_j , the `rssSum` method computes the tuple {rss, sum of predicted values}:

```
def rssSum(xt: XTSeries[DblVector], y: DblVector): XY = {
    val regression = MultiLinearRegression[Double](xt, y)
    (regression.rss.get,
     xt.toArray.zip(y).foldLeft(0.0)(s,x) =>
     val d = (x._2 - (regression |> x._1))
     s + d*d
   ))
}
```

Finally, the process is repeated for each model and the sum of the predicted values for each model is summed (line 8), averaged (line 9), and then used in the r_2 formula (line 10):

```
var xsRss = new ListBuffer[Double]()
val tss = featuresList.foldLeft(0.0)((s, x) => { //8
    val _tss = rssSum(XTSeries[DblVector](x._2), input(0))
    xsRss.append(_tss._1)
    s + _tss._2 //9
})/xsRss.size
xsRss.map( 1.0 - _/tss) //10
```

The graph plotting the r^2 value for each model confirms that the three features model is the most accurate:



General linear regression

The concept of 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 also be defined as a linear combination of **basis functions** as $\phi_j: y = w_0 + w_1\phi_1(x) + w_2\phi_2(x) + \dots + w_n\phi_n(x)$ [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 model. You are already aware that regularization is used to reduce model complexity and avoid overfitting as stated in the *Overfitting* section of *Chapter 2, Hello World!*.

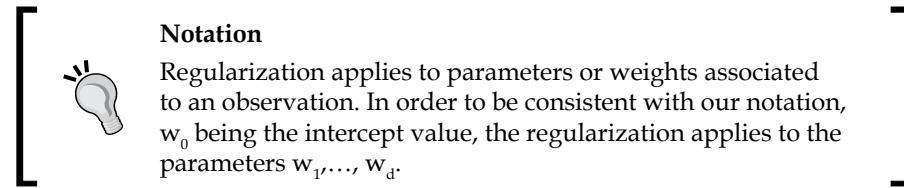
L_n roughness penalty

Regularization consists of adding a penalty function $J(w)$ to the loss function (or RSS in the case of a regressive classifier) in order to prevent the model parameters (or weights) from reaching high values. A model that fits a training set very well tends to have many features variable 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:

$$\hat{w} = \arg \min_{w_d} \left\{ \sum_{i=0}^{n-1} (y_i - f(x_i | w))^2 + \lambda J(w) \right\}$$

The penalty function is completely independent from 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_a . For a model of D dimensions, the generic L_p -norm is defined as follows:

$$J_{pq}(w) = \|w\|_p^q = \left[\sum_{d=1}^{D-1} |w_d|^p \right]^{q/p}$$



The two most commonly used penalty functions for regularization are L_1 and L_2 .

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 L_1 regularization applied to the linear regression is known as the **Lasso regularization**. The **Ridge regression** is a linear regression that uses the L_2 regularization penalty.

You may wonder which regularization makes sense for a given training set. In a nutshell, L_2 and L_1 regularization 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 high 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, it is more appropriate for selecting features.
- **Computation:** L_2 is conducive to a more efficient computation model. The summation of the loss function and the L_2 penalty, \mathbf{w}^2 , is a continuous and differentiable function for which the first and second derivative can be computed (**convex minimization**). The L_1 term is the summation of $|\mathbf{w}_i|$ 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.

The ridge regression

The ridge regression is a multivariate linear regression with an L₂-norm penalty term:

$$\hat{w}_{Ridge} = \arg \min_{w_d} \sum_{j=0}^{n-1} (y - w_0 - w^T x_j)^2 + \lambda \|w\|_2^2 \quad \|w\|_2^2 = \sum_{d=1}^D w_d^2$$

The computation of the ridge regression parameters requires the resolution of a system of linear equations similar to the linear regression.

The matrix representation of the ridge regression closed form is as follows:

$$(X^T X - \lambda I) \hat{w}_{Ridge} = X^T y$$

 I is the identity matrix and uses the QR decomposition:

$$(X^T X - \lambda I) Q \begin{bmatrix} R \\ 0 \end{bmatrix} \quad w_{Ridge} = Q^T \begin{bmatrix} R \\ 0 \end{bmatrix}^{-1}$$

Implementation

The implementation of the ridge regression adds the L2 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. However, the class has to inherit the abstract base class, `AbstractMultipleLinearRegression`, in the Apache Commons Math library and override the generation of the QR decomposition to include the penalty term:

```
class RidgeRegression[T <% Double] (xt: XTSeries[Array[T]], y: DblVector, lambda: Double) extends AbstractMultipleLinearRegression with PipeOperator[Array[T], Double] {
    var qr: QRDecomposition = _
    val model: Option[RegressionModel] = ...
    ...
}
```

Besides the input time series `xt` and the labels `y`, the ridge regression requires the `lambda` factor of the L_2 penalty term. The instantiation of the class trains the model. The steps to create the ridge regression models are as follows:

1. Extract the Q and R matrices for the input values, `newXSampleData` (line 1).
2. Compute the weights using `calculateBeta` defined in the base class (line 2).
3. Return the tuple regression weights, `calculateBeta`, and the residuals, `calculateResiduals`.

Consider the following code:

```
val model: Option[(DblVector, Double)] = {
    this.newXSampleData(xt.toDblMatrix) //1
    newYSampleData(y)
    val _rss = calculateResiduals.toArray.map(x => x*x).sum
    val wRss = (calculateBeta.toArray, _rss) //2
    Some(RegressionModel(wRss._1, wRss._2))
}
```

The QR decomposition in the base class, `AbstractMultipleLinearRegression`, does not include the penalty term (line 3); the identity matrix with the `lambda` factor in the diagonal has to be added to the matrix to be decomposed (line 4):

```
override protected def newXSampleData(x: DblMatrix): Unit = {
    super.newXSampleData(x) //3
    val xtx: RealMatrix = getX
    val nFeatures = xt(0).size
    Range(0, nFeatures)
        .foreach(i =>xtx.setEntry(i,i,xtx.getEntry(i,i)+lambda)) //4
    qr = new QRDecomposition(xtx)
}
```

The regression weights are computed by resolving the system of linear equations using substitution on the Q.R matrices. It overrides `calculateBeta` from the base class:

```
override protected def calculateBeta: RealVector =
    qr.getSolver().solve(getY())
```

The test case

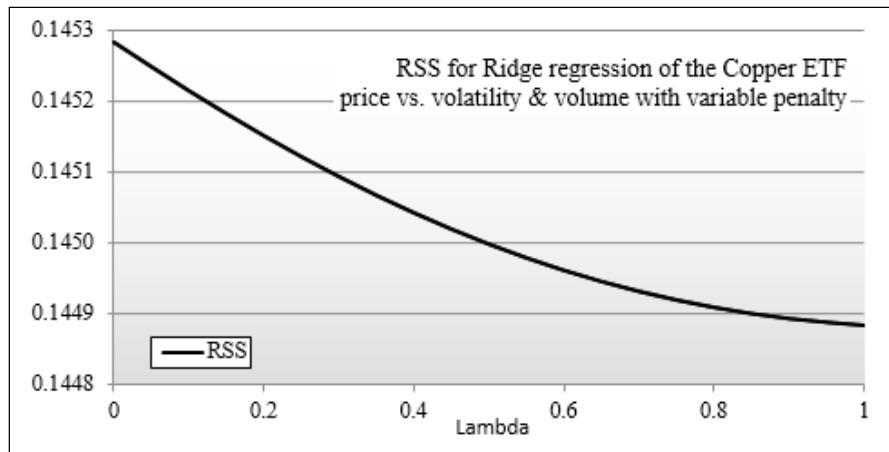
The objective of the test case is to identify the impact of the L_2 penalization on the RSS value and then compare the predicted values with the original values.

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 of the least squares regression:

```
val lambda = 0.5
val src = DataSource(path, true, true, 1)
val price = src |> YahooFinancials.adjClose
val volatility = src |> YahooFinancials.volatility
val volume = src |> YahooFinancials.volume //1
val deltaPrice = XTSeries[Double](price.drop(1)
                                    .zip(price.take(_price.size -1))
                                    .map(z => z._1 - z._2)) //2
val data = volatility.zip(volume)
           .map(z => Array[Double](z._1, z._2)) //3
val features = XTSeries[DblVector](data.dropRight(1))
val regression = new RidgeRegression[Double](features, deltaPrice,
                                             lambda) //4
regression.rss match {
  case Some(rss) => Display.show(rss, logger)
  ...
}
```

The observed data, that is, the ETF daily price and the features (volatility and volume) are extracted from the `src` source (line 1). The daily price change `deltaPrice` is computed using a combination of Scala `take` and `drop` methods (line 2). The features vector is created by zipping `volatility` and `volume` (line 3). The model is created by instantiating the `RidgeRegression` class (line 4). The RSS value, `rss`, is finally displayed (line 5).

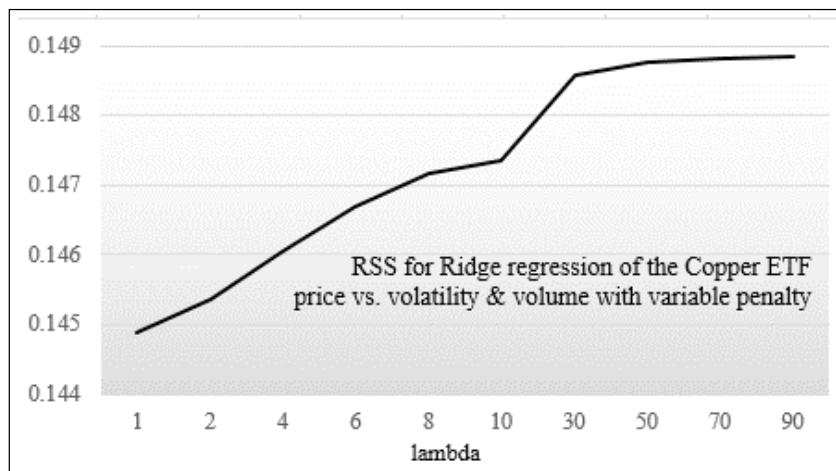
The RSS value, r_{ss} , is plotted for different values of lambda less than 1.0, as shown in the following chart:



Graph of RSS versus Lambda for 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:



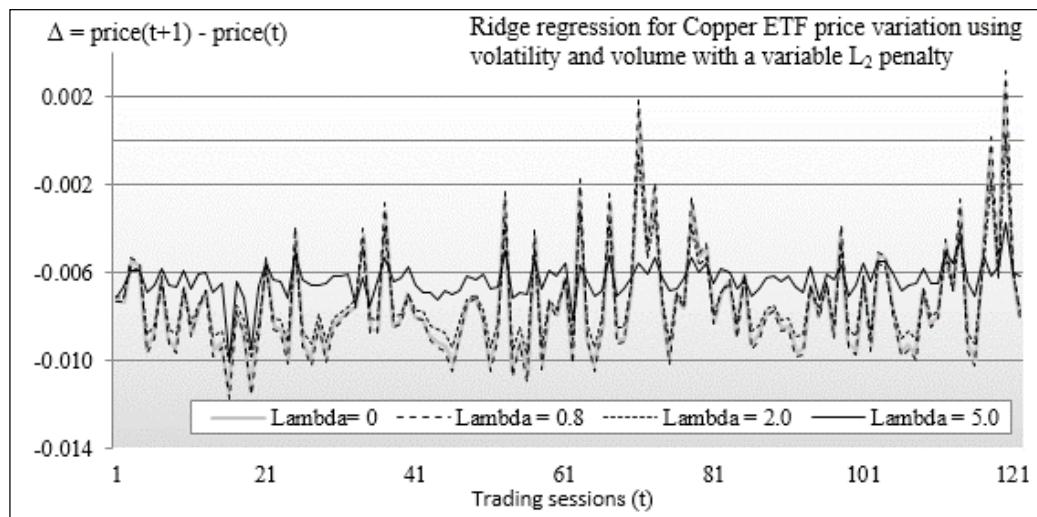
Graph of RSS versus large-value Lambda for Copper ETF

This time around, the value of RSS increases with λ before reaching a maximum of $\lambda > 60$. This behavior is consistent with other findings [6:12]. As λ increases, the overfitting gets more expensive and therefore, the RSS value increases.

The regression weights can be simply outputted as follows:

```
regression.weights.get
```

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 Copper ETF price variation with variable lambda

The original price variation of the Copper ETF, $\Delta = \text{price}(t+1) - \text{price}(t)$, is plotted as $\lambda = 0$. 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 $\lambda = 5$ correspond 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, briefly introduced in the *Let's kick the tires* section of *Chapter 1, Getting Started*, is the next logical regression model to discuss. The logistic regression relies on optimization methods. Let's go through a short refreshment 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 greater detail in the *Summary of optimization techniques* section in *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 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.

The definition of the least squares-based loss function is as follows:

$$L(w) = \sum_{i=0}^{n-1} r_i(w) = y_i - f(x_i | w)$$

The generation of gradient equations with a Jacobian J matrix (refer to the *Jacobian and Hessian matrices* section in *Appendix A, Basic Concepts*) after minimization of the loss function L is described as follows:



$$\sum_{i=0}^{n-1} r_i(w) J_{id} = 0 \quad J_{id}(w) = -\frac{\partial r_i(w)}{\partial w_d}$$

Iterative approximation using the Taylor series is described as follows:

$$f(x_i | w) - f(x_i | w^{(k)}) \sim \sum_{jd=0}^{D-1} \frac{\partial f(x_i | w^{(k)})}{\partial w_d} (w - w^{(k)})$$

Normal equations using the matrix notation and the Jacobian J matrix is described as follows:

$$(J^T J)(w^{(k+1)} - w^{(k)}) = J^T (y_i^{(k+1)} - y_i^{(k)})$$

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 the following 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 the Levenberg-Marquardt methods (refer to the *Nonlinear least squares minimization* section in *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 *Appendix A, Basic Concepts*).

The logistic regression

Despite its name, *the logistic regression is a classifier*. As a matter of fact, the logistic regression is one of the most 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 variable y and a set of variables x that it depends on. Once the model is created (trained), it is used to classify real-time data.

A logistic regression can be either binomial (two classes) or multinomial (three and more classes). In a binomial classification, the observed outcome is defined as {true, false}, {0, 1}, or {-1, +1}.

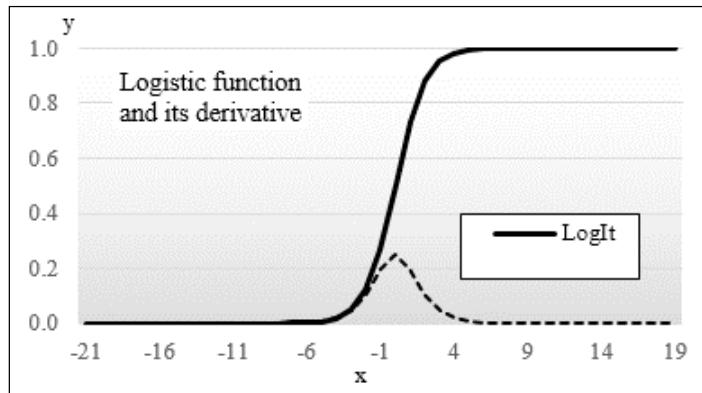
The logit function

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:

$$f(x) = \frac{1}{(1-e^{-x})} \quad \frac{df}{dx} = f(x)(1-f(x))$$

Have a look at 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 a binary classification (two classes).

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 of *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:

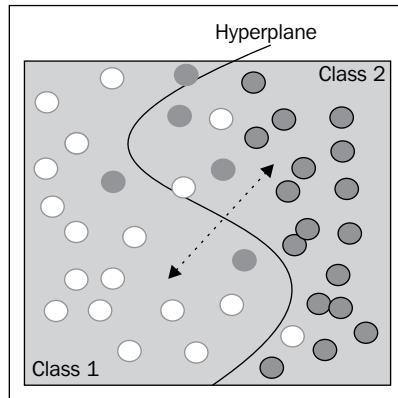


Illustration of the binomial classification for a 2-dimension dataset

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. 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 towards either of the classes.

In the case of two classes, c_1 and c_2 with their respective probabilities, $p(C=c_1 | X=x^i | w) = p(x^i | w)$ and $p(C=c_2 | X=x^i | w) = 1 - p(x^i | w)$, where w is the model parameters set or weights in the case of the logistic regression, the following functions can be defined:

The log likelihood:

$$L(w) = \sum_{i=0}^{N-1} \log p(x_i | w)$$

Conditional probabilities using the logit function:



$$x = [1, x_1, x_2 \dots x_d]^T \quad p(x | w) = \frac{e^{w^T x}}{1 + e^{w^T x}}$$

The log likelihood for the binomial logistic regression:

$$L(w) = \sum_{i=0}^{N-1} y_i w^T x_i - \log(1 + e^{w^T x_i}) \quad y \in \{0, 1\}$$

First order derivative for the log likelihood:

$$\frac{\partial L(w)}{\partial w_j} = \sum_{i=0}^{N-1} x_{ij} (y_i - p(x_i | w))$$

Let's implement the logistic regression without a penalty term using the Apache Commons Math library. The library contains several least squares optimizers, allowing you to specify the minimizing algorithm, optimizer, for the loss function in the logistic regression class LogisticRegression:

```
class LogisticRegression[T <% Double] (xt: XTSeries[Array[T]], 
  labels: Array[Int], optimizer: LogisticRegressionOptimizer) extends 
  PipeOperator[Array[T], Int] {
  val model: Option[RegressionModel] = { ... }
  ...
}
```

The parameters of the logistic regression class are the multivariate time series (features) `xt`, the target or labeled data, `labels`, and the optimizer algorithm used to minimize the loss function or residual sum of squares. In the case of the binomial logistic regression, `labels` are assigned the values of 1 for one class and 0 for the other.

The purpose of the training is to determine the regression coefficient, `model._1`, which minimizes the loss function. The **residual sum of squares (RSS)** is computed as `model._2`.

 **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 {false, 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.

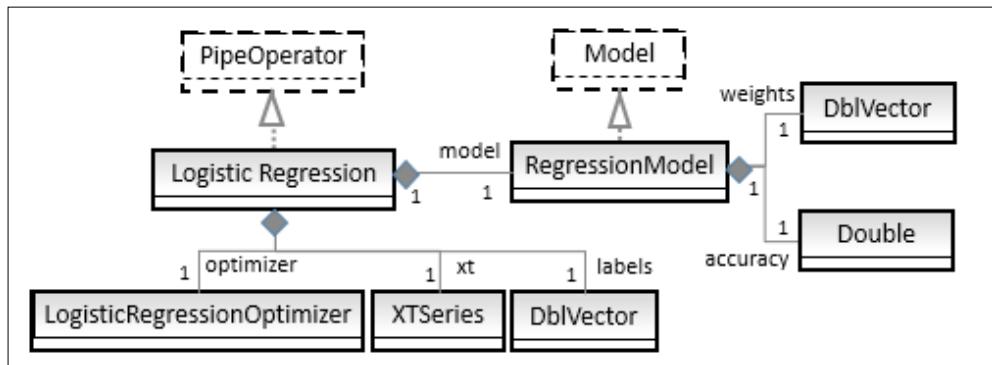
For convenience, the definition and the configuration of the optimizer are encapsulated in the `LogisticRegressionOptimizer` class.

Software design

The implementation of the logistic regression uses the following components:

- `RegressionModel` of the `Model` type, which is initialized through training during the instantiation of the classifier. We reuse the `RegressionModel` type introduced in the *Linear regression* section.
- The predictive or classification routine is implemented as a data transformation `| > extending the PipeOperator trait.`
- The logistic regression class, `LogisticRegression`, has three parameters: the least squares optimizer of the type `LogisticRegressionOptimizer` (used in training), a features set `XTSeries`, and a label vector `DblVector`.

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 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 *Appendix A, Basic Concepts*) packaged with the Apache Commons Math library.

The training of the logistic regression is performed by the `train` method:

```
val model: Option[RegressionModel] = train
```



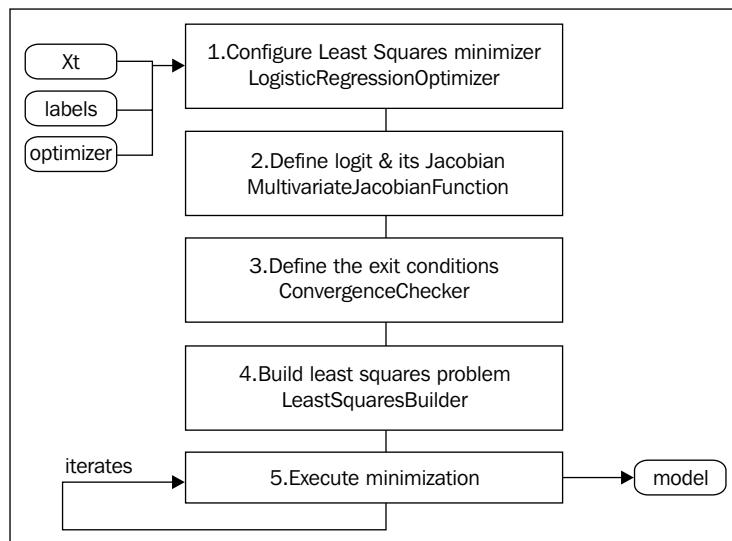
Handling exceptions from the Apache Commons Math library

The training of the logistic regression using the Apache Commons Math library requires handling `ConvergenceException`, `DimensionMismatchException`, `TooManyEvaluationsException`, `TooManyIterationsException`, and `MathRuntimeException`. 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 logit 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 Apache Commons Math

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.

Configuring the least squares optimizer

In this step, you have to specify the algorithm to minimize the residual of the sum of 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

Consider the following code:

```
class LogisticRegressionOptimizer(maxIters: Int, maxEvals: Int, eps: Double, lsOptimizer: LeastSquaresOptimizer) {
    def optimize(lsProblem: LeastSquaresProblem): Optimum = lsOptimizer.optimize(lsProblem)
}
```

The configuration of the logistic regression optimizer is defined using the maximum number of iterations (`maxIters`), the maximum number of evaluations (`maxEval`) for the logistic function and its derivative, the convergence criteria (`eps`) on the residual sum of squares, and the instance of the least squares problem (`org.apache.commons.math3.fitting.leastsquares.LeastSquaresProblem`).

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` interface:

```
final val initWeight = 0.5
val weights0 = Array.fill(xt(0) + 1)(initWeight) //1

val lrJacobian = new MultivariateJacobianFunction {
    override def value(w: RealVector): Pair[RealVector, RealMatrix] = {
        val _w = w.toArray
        val gradient = xt.toArray
            .map(g => { //2
                val expn = g.zip(_w.drop(1))
                    .foldLeft(_w(0))((s, z) => s + z._1 * z._2)
                val logIt = 1.0 / (1.0 + Math.exp(-expn)) //3
                (logIt, logIt * (1 - logIt)) //4
            })
        val jacobian = Array.ofDim[Double](xt.size, weights0.size) //5
        xt.toArray.zipWithIndex.foreach(xi => { //6
            val df: Double = gradient(xi._2)._2
            Range(0, xi._1.size).foreach(j =>
                jacobian(xi._2)(j + 1) = xi._1(j) * df)
            jacobian(xi._2)(0) = 1.0 //7
        })
        (new ArrayRealVector(gradient.map(_.-_1)),
         new Array2DRowRealMatrix(jacobian)) //8
    }
}
```

The regression weights, `weights0`, are initialized with the arbitrary value of 0.5.

The `value` method uses the primitives types `RealVector`, `RealMatrix`, `ArrayRealVector`, and `Array2DRowRealMatrix` defined in the `org.apache.commons.math3.linear` Apache Commons Math package.

It takes the regression weight, `w`, and computes the gradient (line 2) of the logistic function for each data point and returns the value of `logit` (line 3) and its derivative (line 4) as a tuple. The Jacobian matrix is created (line 5), and then initialized with `logit` and its derivative (line 6). The first element of each column of the Jacobian matrix is set to 1.0 to take into account the intercept (line 7). Finally, the vector of the `logit` values for each observation and the Jacobian matrix are returned (line 8) as a tuple to comply with the return type of the function value.

Defining the exit conditions

The third step defines the exit condition for the optimizer. It is accomplished by overriding the `converged` method of the parameterized `org.apache.commons.math3.optim.ConvergenceChecker` interface:

```
val exitCheck = new ConvergenceChecker[PointVectorValuePair] {
    override def converged(iteration: Int, prev: PointVectorValuePair,
                           current: PointVectorValuePair): Boolean = {
        val delta = prev.getValue
            .zip(current.getValue)
            .foldLeft(0.0)((s, z) =>{
                val d = z._1 - z._2
                s + diff*diff
            })
        Math.sqrt(delta) < optimizer.eps && iteration >= optimizer.maxIterations
    }
}
```

This implementation computes the convergence or exit condition as follows:

- Either the L_2 -norm of the difference between the weights of the current iteration and the weights of the previous iteration, `delta`, is smaller than the convergence criteria, `eps`
- Or the iteration exceeds the maximum number of iterations that `maxIterations` allowed

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 `LeastSquareProblem` generated by the factory `LeastSquareBuilder` class:

```
val builder = new LeastSquaresBuilder
val diagWeights0 = Array.fill(xt.size) (1.0) //1
val wMatrix = MatrixUtils.createRealDiagonalMatrix(diagWeights0)
val lsp = builder.model(lrJacobian) //2
    .weight(wMatrix)
    .target(labels) //7
    .checkerPair(exitCheck) //5
    .maxEvaluations(optimizer.maxEvals) //3
    .start(weights0) //6
    .maxIterations(optimizer.maxIters) //4
    .build
```

The diagonal elements of the weights matrix are initialized to 1.0 (line 1). Besides the initialization of the model with the Jacobian matrix, `lrJacobian` (line 2), the maximum number of evaluations (line 3), maximum number of iterations (line 4), and the exit condition (line 5) are also initialized.

The regression weights are initialized as 0.5 (`weights0`) (line 6). Finally, the labeled or target values are initialized (line 7).

Minimizing the loss function

The training is executed with a simple call to the least squares minimizer, `lsp`:

```
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 Copper ETF price variation versus volatility and volume, 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. Therefore, the `deltaPrice` vector used in the linear and ridge regression is to be modified to support the binary outcome:

```
val deltaPrice = prices.drop(1).zip(prices).dropRight(1)  
.map(p => if(p._1 > p._2) 1 else 0)
```

Executing the test case is just a matter of instantiating the `LogisticRegression` class with the appropriate configuration parameters. The implementation reuses the code already defined for the least squares and ridge regression to load data from CSV files (`src`, `price`, `volatility`, and `volume`) and normalize the observations:

```
val MAXITERS = 80; val MAXEVALS = 1000; val EPS = 1e-4  
  
val lsOptimizer = LogisticRegressionOptimizer(MAXITERS, MAXEVALS, EPS,  
new LevenbergMarquardtOptimizer)  
val xt = XTSeries[DblVector](features)  
val regression = new LogisticRegression[Double](xt, deltaPrice,  
lsOptimizer)  
val rms = regression.rms.get  
val weights = regression.weights.get
```

In this example, the Levenberg-Marquardt algorithm is used to minimize the loss function.



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 relative tolerance for cost and matrix inversion, that are worth tuning for commercial applications (refer to the *Levenberg-Marquardt* section in *Appendix A, Basic Concepts*).

The execution of the test produces the following results:

- 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 mention earlier and despite its name, the binomial logistic regression is a binary classifier. The classification method is implemented as a data transformation by overriding the pipe operator:

```
type Feature = Array[T]
final val MARGIN = 0.01
def |> : PartialFunction[Feature, Int] = { //1
    case x: Feature if(model!=None && model.get.size-1==x.size) =>{
        val w = _model.get.weights
        val dot = x.zip(w.drop(1))
            .foldLeft(w(0))((s,xw) => s + xw._1*xw._2)//2
        if(logit(dot) > 0.5 + MARGIN) 1 else 0 //3
    }
}
```

The classification method, `|>`, checks if the number of model parameters (weights) is equal to the number of features plus 1 (line 1) and throws an exception if the test fails. The dot product of the weights and the features is computed using a fold. Finally, the method returns 1 (class 1, which signifies that the price variation of the ETF is positive) if the value of the sigmoid is greater than 0.5. It returns 0 otherwise (class 2, which signifies that the price variation of the ETF is negative) (line 3).

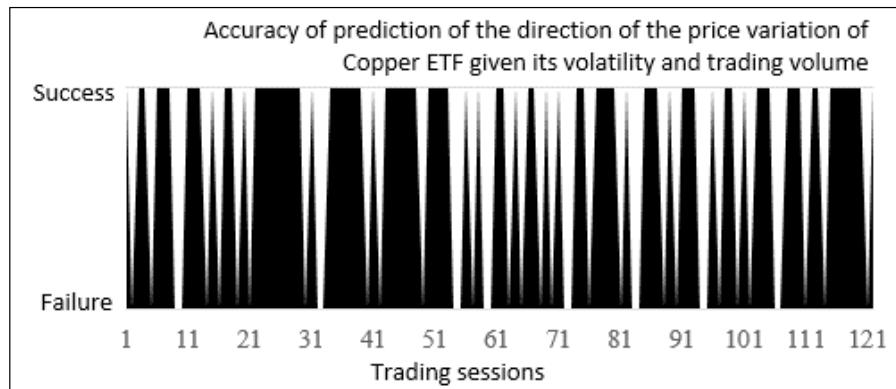
Class identification

The class that the new data x belongs to is determined by the $\text{logit}(\text{dot}) > 0.5$ test, where dot is the product of the features and the regression weights ($w_0 + w_1 \cdot \text{volatility} + w_2 \cdot \text{volume}$). This test is equivalent to $\text{dot} > 0.0$. You may find either condition in the literature.

Let's apply the classification to the original training set, `features`, to validate our model (weights):

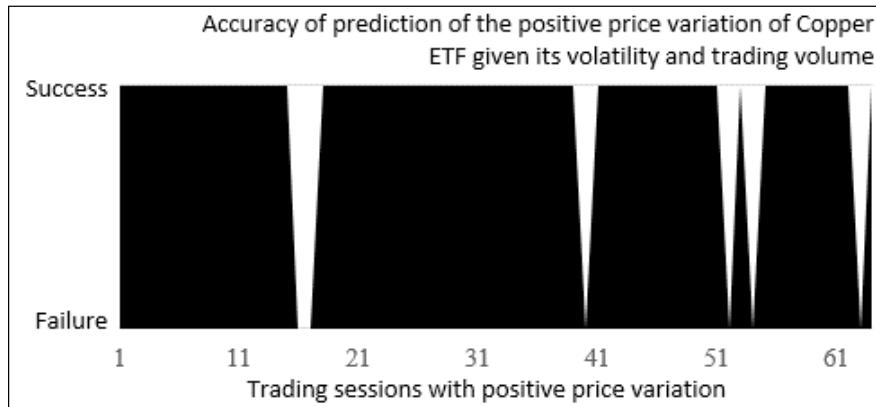
```
val predicted = features.map(x => regression |> x)
```

The direction of the price variation of the Copper ETF, $\text{price}(t+1) - \text{price}(t)$, 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 classified, otherwise, it is plotted with the failure value:



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 use case ignores the trading sessions for which the price was either flat or declined:



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 separating hyperplane equation, $w_0 + w_1 \cdot \text{volatility} + w_2 \cdot \text{volume}$, is used to segregate both the features generating either positive or negative price variation. 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 one 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 separating hyperplane equation: $w_0 + w_1 \cdot \text{volatility} + w_2 \cdot \text{volume}$. This negatively impacts 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 price as $\text{price}(t+1) - \text{price}(t) > EPS$.



Validation methodology

The validation set is generated by randomly selecting data points from the original labeled set. A formal validation requires the use of a K-fold validation methodology to compute the recall, precision, and F_1 measure for the logistic regression model.

Summary

This concludes the description and implementation of 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.

At the completion of this chapter, you hopefully have a grasp on the following:

- 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 introduced in the next chapter and artificial neural networks 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, *Chapter 7, Sequential Data Models*, describes two classifiers that take into account the time dependency in a time series.

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 details 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.

The second and last section of the chapter is dedicated to a discriminative (labels conditional to observation) 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* by 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 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 to draw a red ball in the first attempt is $p(q_0 = S_1) = \pi_1$. The probability to draw a blue ball in the second attempt is $p(q_1 = S_0) = \pi_0$. 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:

$$p(q_0 = S_1) \cdot p(q_1 = S_0 | q_0 = S_1) \cdot p(q_2 = S_0 | q_1 = S_0) \cdot p(q_3 = S_2 | q_2 = S_0) \dots = \pi_1 \cdot a_{10} \cdot a_{00} \cdot a_{02} \dots$$

The sequence of states/colors can be represented as follows:

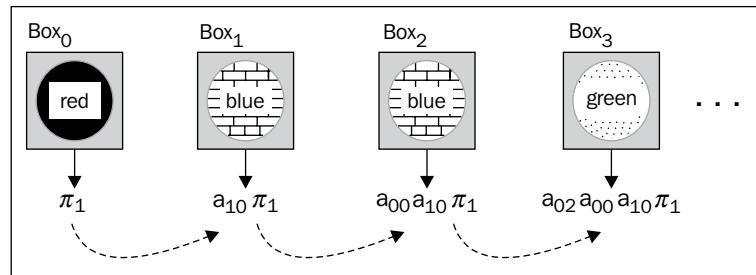


Illustration of the ball and boxes example

Let's estimate the probabilities p using historical data (learning phase):

1. The estimation of the probability to draw a red ball (S_1) in the first attempt is π_1 , which is computed as the number of sequences starting with S_1 (red) / total number of balls.
2. The estimation of the probability of retrieving a blue ball in the second attempt is a_{10} , 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 (HMM)

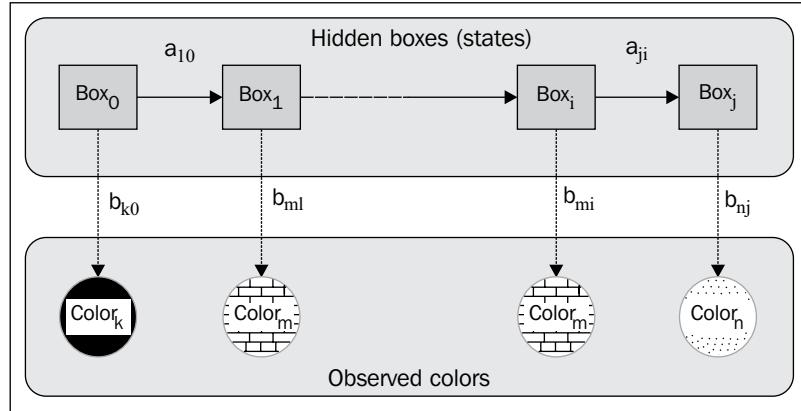
The hidden Markov model has numerous applications related to speech recognition, face identification (biometrics), and pattern recognition in pictures and video [7:3].

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 now two sources of randomness:

- 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 (non-observable), 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 described 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).

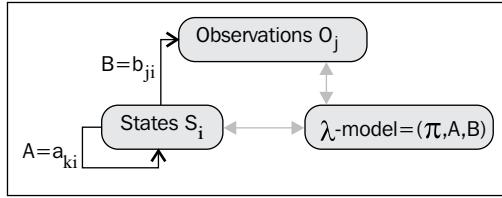
Time invariance

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.

It must be kept 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:



Visualization of the HMM key components

This diagram illustrates that, given a sequence of observations, HMM tackles three problems known as canonical forms:

- **CF1 – evaluation:** Evaluate the probability of a given sequence of observations O_T , given a model $\lambda = (\pi, A, B)$
- **CF2 – training:** Identify (or learn) a model $\lambda = (\pi, A, B)$ given a set of observations O
- **CF3 – decoding:** Estimate 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.

Notation

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	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	$O = \{o_0, o_1, \dots, o_{T-1}\} = O_{0:T-1}$

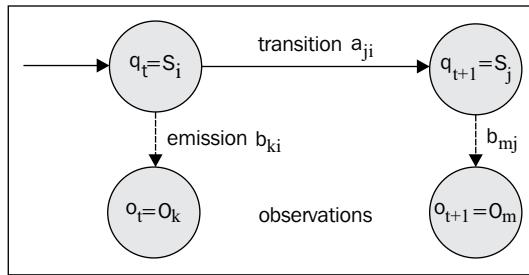
	Description	Formulation
A	The state transition probability matrix	$a_{ji} = p(q_{t+1}=S_i \mid q_t=S_j)$
B	The emission probability matrix	$b_{jk} = p(o_t=O_k \mid q_t=S_j)$
π	The initial state probability vector	$\pi_i = p(q_0=S_i)$
λ	The hidden Markov model	$\lambda = (\pi, A, B)$



Variance in 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 following condensed form: $p(O_{0:T}, q_t \mid \lambda) = p(O_0, O_1, \dots, O_T, q_t \mid \lambda)$. It is quite common to visualize a hidden Markov model with a lattice of states and observations 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 _T: Int, val _N: Int, val _M: Int) extends Config
```

The input parameters for the class are:

- $_T$: The number of observations
- $_N$: The number of hidden states
- $_M$: The number of observation symbols or features

Consistency with mathematical notation



The implementation uses $_T$ (with respect to $_N$, $_M$) to represent programmatically the number of observations T (with respect to hidden states N and features M). As a general rule, the implementation reuses the mathematical symbols as much as possible. Although the practice does not always make the code elegant, it improves its readability.

The `HMMConfig` companion object defines the operations on ranges of index of matrix rows and columns. The `foreach`, `foldLeft`, and `maxBy` methods are regularly used in each of the three canonical forms:

```
object HMMConfig {
    def foreach(i: Int, f: Int => Unit) = Range(0, i).foreach(f)
    def foldLeft(i: Int, f: (Double, Int) => Double, zero:Double) =
        Range(0, i).foldLeft(zero)(f)
    def maxBy(i: Int, f: Int => Double) = Range(0,i).maxBy(f)
    ...
}
```

Notation



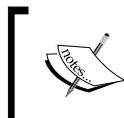
The λ model in HMM should not be confused with the regularization factor discussed in the L_n roughness penalty section in *Chapter 6, Regression and Regularization*.

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 a case class, `HMMLambda`, using the `Matrix` class defined in the *Matrix class* section in *Appendix A, Basic Concepts*. The simplest constructor for the `HMMLambda` class is invoked in the case where the state-transition probability matrix, the emission probability matrix, and the initial states are known, as shown here:

```
class HMMLambda(val A: Matrix[Double], val B: Matrix[Double], var pi: DblVector, val numObs: Int) {
    def getT: Int = numObs
    def getN: Int = A.nRows
    def getM: Int = B.nCols
    val d1 = numObs - 1
    ...
}
```

The implementation reflects the mathematical notation, with π being the initial state probability, A the state transition matrix, and B the emission matrix. The `numObs` value is the number of observations in the sequence. The `getT`, `getN`, and `getM` methods are used to keep the implementation consistent with the initial configuration, `HMMConfig`. The section related to the training of HMM introduces a different constructor for `HMMLambda` using the configuration as a parameter.

The initial probabilities are unknown, and therefore, initialized with a random generator of values $[0, 1]$.



Normalization

Input states and observations data may have to be normalized and converted to probabilities before initializing the matrices A and B .



The two other components of the HMM are the sequence of observations and the sequence of hidden states.

HMM execution state

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 variable): The probability of observing the first $t < T$ observations for a specific state at S_i for the observation t , $\alpha_t(i) = p(O_{0:t}, q_t = S_i | \lambda)$
- Beta (the backward variable): The probability of observing the remainder of the sequence q_t for a specific state $\beta_t(i) = p(O_{t+1:T} | q_t = S_i, \lambda)$
- Gamma: The probability of being in a specific state given a sequence of observations and a model $\gamma_t(i) = p(q_t = S_i | O_{0:T}, \lambda)$
- Delta: The sequence to have the highest probability path for the first i observations defined for a specific test $\delta_t(i)$
- Qstar: 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 $\gamma_t(i,j) = p(q_t = S_i, q_{t+1} = S_j | O_{0:T}, \lambda)$

Each of the parameters is described in the section related to each canonical form. Let's create a class `HMMState` that encapsulates the variables used in the implementation of the three canonical cases.

For convenience, all the parameters related to the three canonical cases and listed in the previous notation section are encapsulated into a single outer class, `HMMState`:

```
class HMMState(lambda: HMMLambda, maxIters:Int) extends Config {
    val delta = Matrix[Double](lambda.getT, lambda.getN) // δt(i)

    object QStar { ... } // q*
    object DiGamma { ... } // γt(i, j)
    object Gamma { ... } // γt(i)
}
```

Once again, we use the same notation as for the configuration of the HMM; `lambda.getT`, being the number of observations, and `lambda.getN`, the number of hidden states. The HMM state parameters have self-descriptive names that strictly follow the notation introduced earlier. The λ model, the HMM state, and the sequence of observations are all the elements needed to implement the three canonical cases.

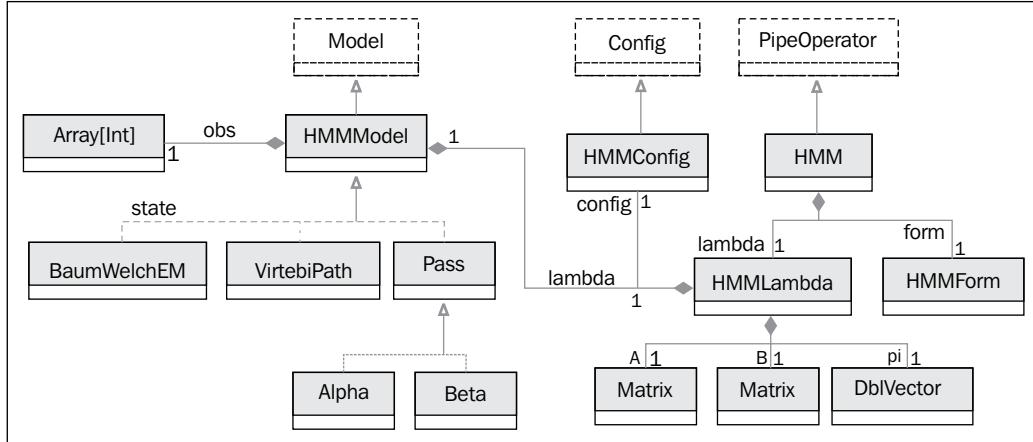
The `Gamma` and `DiGamma` singletons are used and described in the evaluation canonical form. The `DiGamma` singleton is described as part of the Viterbi algorithm to extract the sequence of states with the highest probability given a λ model and a set of observations.

The execution of any of the three canonical forms relies on dynamic programming techniques (refer to the *Overview of dynamic programming* section in *Appendix A, Basic Concepts*) [7:5]. The simplest of the dynamic programming techniques is a single traversal of the observations/state chain.

Therefore, it makes sense to define a base class, `HMMModel`, that has all the algorithms that manipulate the λ model, `lambda`, and the observed states, `obs`:

```
abstract class HMMModel(lambda: HMMLambda, obs: Array[Int])
```

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 (UML class diagram)

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.

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:

$$p(O_{0:T-1} | \lambda) \propto p(O_{0:t} | \lambda) \cdot p(O_{t+1:T-1} | \lambda)$$

The likelihood is computed by marginalizing over all the hidden states [7:6]{ S_i }:

$$p(O_{0:T-1} | \lambda) = \sum_{i=0}^{N-1} p(O_{0:T-1}, q_t = S_i | \lambda)$$

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:

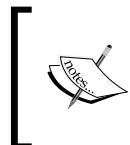
$$p(O_{0:T-1} | \lambda) = \sum_i \alpha_t(i) \cdot \beta_t(i)$$

The product of the probabilities α and β can potentially underflow. Therefore, it is recommended to use the log of the probabilities instead of the probabilities.

Alpha class (the forward variable)

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 [M1]. The value is then normalized by the sum of alpha values across all the hidden states [M2].
2. Compute the alpha value iteratively for the time 0 to time t , then normalize by the sum of alpha values for all states [M3].
3. The final step is the computation of the log of the probability of observing the sequence [M4].



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, computation of the alpha matrix is defined in the following information box. Each formula has an identifier [Mx], which is referenced in the Scala source code implementing it.

Alpha-class (forward variable)

- **M1:** Initialization:

$$\alpha_0(i) = \pi_i \cdot b_i(O_0)$$

- **M2:** Normalization of initial values:

$$\hat{\alpha}_0(i) = \alpha_0(i) / \sum_{j=0}^{N-1} \alpha_0(j)$$

- **M3:** Normalized summation:

$$\alpha_t(i) = \sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji} b_i(O_t) \quad c_t = 1 / \sum_{i=0}^{N-1} \alpha_t(i) \quad \hat{\alpha}_t(i) = \alpha_t(i) \cdot c_t$$

- **M4:** Probability of observing a sequence given a lambda model and states:

$$\log p(O | \lambda) = - \sum_{j=0}^{T-1} \log \left(\frac{1}{\sum_{i=0}^{N-1} \hat{\alpha}_t(i)} \right)$$



Let's 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 a base class, `Pass`, for the alpha and beta algorithms that implements the normalization:

```
class Pass(_lambda: HMMLambda, _obs: Array[Int]) extends HMMModel(_lambda, _obs) { //1
    var alphaBeta: Matrix[Double] = _
    val ct = Array.fill(lambda.getT)(0.0) //2

    def normalize(t: Int): Unit = {
        ct.update(t, foldLeft(lambda.getN, (s, n) => s + alphaBeta(t, n))) //3
        alphaBeta /= (t, ct(t))
    }
}
```

As with any algorithm used in the hidden Markov model, the `Pass` base class of the alpha and beta classes is a composition of the attributes of the model (`HMMLambda`), the computation parameters (`HMMParams`), and the sequence of observations `obs` (line 1). The `alphaBeta` matrix represents either the alpha or beta matrix manipulated in the subclasses (line 2). The scale factor, `ct`, is computed as the summation of the alpha or beta row matrix over all the states using a fold (line 3).



Computation efficiency

Scala's `reduce`, `fold`, and `foreach` methods are far more efficient iterators than the `for` loop. You need to keep in mind that the main purpose of the `for` loop in Scala is the monadic chaining of `map` and `flatMap` operations.

The computation of the `alpha` variable in the `Alpha` class follows the same computation flow as defined in the mathematical expression:

```
class Alpha(lambda: HMLambda, obs: Array[Int]) extends Pass(lambda,
obs)
```

The `alpha` value is initialized [M1] (line 4), then normalized [M2] using the current sequence order (line 5). The value of `alpha` is then updated [M3] by summation of the previous `alpha` value at $t-1$ and the transition from the state j to the state i (line 6), as shown here:

```
Import HMMConfig._

val alpha = {
    alphaBeta = lambda.initAlpha(obs) //4
    normalize(0) //5
    sumUp //6
}

def sumUp: Double = { // [M2]
    foreach(lambda.getT, t => {
        updateAlpha(t) //7
        normalize(t) //8
    })
    foldLeft(lambda.getN, (s, k) => s + alphaBeta(lambda.dim_1, k))
}

def updateAlpha(t: Int): Unit =
    HMMConfig.foreach(lambda.getN, i =>
        alphaBeta += (t, i, lambda.alpha(alphaBeta(t-1, i), i, obs(t)))
    )
}
```

The value of `alpha` is updated by the `updateAlpha` method (line 7) before normalization (line 8). The implementation that relies on the `fold` method is omitted, but can be easily written.

Finally, the computation of the logarithm of the probability to observe a specific sequence, given the sequence of states and a predefined λ model, [M4] can be performed by the following code (line 9):

```
def logProb: Double = foldLeft(lambda.getT, (s, t))  
    => s + Math.log(ct(t)), Math.log(alpha))//9
```

The 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 class (the backward variable)

The recursive 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 at the time $T-1$ to t updated from its value at $t+1$ [M7].

Beta class (the backward variable)

- M5: Initialization of beta: $\beta_{T-1}(t)=1$
- M6: Normalization of initial beta values:

$$\hat{\beta}_{T-1}(i) = \beta_{T-1}(i) / \sum_{j=0}^{N-1} \beta_{T-1}(j)$$

- M7: Normalized summation of beta:

$$\beta_t(i) = \sum_{j=0}^{N-1} \beta_{t+1}(j) \cdot a_{ij} \cdot b_j(O_{t+1}) \quad c_t = 1 / \sum_{j=0}^{N-1} \beta_t(j) \quad \hat{\beta}_t(i) = \beta_t(i) \cdot c_t$$

The definition of the class for the `Beta` class is identical to the `Alpha` class:

```
class Beta(lambdaB: HMMLambda, _obs: Array[Int]) extends Pass(lambdaB,
    _obs)
```

The implementation of the `Beta` class is similar to the `Alpha` class with computation (line 1) and normalization (line 2) of beta at $t=0$. As expected, the summation routine `sumUp` (line 3) is implemented as updating and normalizing beta at the time t , as shown here:

```
val complete = { //4
    alphaBeta = Matrix[Double](lambda.getT, lambda.getN)
    alphaBeta += (lambda.dim_1, 1.0) //1
    normalize(lambda.dim_1) //2
    sumUp; true
}

def sumUp: Unit = //3
    (lambda.getT-2 to 0 by -1).foreach( t =>{
        updateBeta(t)
        normalize(t)
    })

def updateBeta(t: Int): Unit =
    foreach(lambda.config.getN, i => {
        alphaBeta += (t, i, lambda.beta(alphaBeta(t+1, i), i, obs(t+1)))
    })
```

The recursive method updates and normalizes the beta matrix by traversing the sequence of observations backward from before the last observation to the first. Contrary to the `Alpha` class, the `Beta` class does not generate an output value. Therefore, we need to flag the state of the class using a ready Boolean value, which is set to true if the instantiation succeeds and false otherwise.

Constructors

The alpha and beta values are computed within the constructors of their respective class, so no public or protected method needs to verify if these values are already computed. The design pattern reduces the complexity of implementation by ensuring that a class instance has only one state: computation completed.

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).

Baum-Welch estimator (EM)

At its core, the algorithm has three steps and an iterative method, similar to the evaluation canonical form:

1. Compute the probability π (the gamma value at $t=0$) [M9].
2. Compute and normalize the state's transition probabilities matrix A [M10].
3. Compute and normalize the matrix of emission probabilities B [M11].
4. Repeat steps 2 and 3 until the change of likelihood is insignificant.

The algorithm uses the digamma and summation gamma variables defined in the `HMMConfig` class.

The Baum-Welch algorithm

- **M8:** Joint probability of the state q_i at t and q_j at $t+1$:

$$\gamma_t(i, j) = p(q_t = S_i, q_{t+1} = S_j | 0, \lambda)$$

$$\gamma_t(i, j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{j=0}^{N-1} \alpha_t(i) \beta_t(j)}$$

- **M9:** The initial probabilities vector:



$$\hat{\pi}_i = \gamma_0(i) \quad \gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i, j)$$

- **M10:** Update of the transition probabilities matrix:

$$\hat{a}_{ij} = \frac{\sum_{t=0}^{T-1} (\gamma_t(i, j))}{\sum_{t=0}^{T-1} \gamma_t(i)}$$

- **M11:** Update of the emission probabilities matrix :

$$\hat{b}_{ij} = \frac{\sum_{t=0}^{T-1} \gamma_t(i)}{\sum_{t=0}^{T-1} \gamma_t(i)}$$

The Baum-Welch algorithm requires the following three inputs:

- The λ model, `_lambda`, initialized with random values uniformly distributed
- The current state of the training, `state`
- The labeled observed data, `_obsIdx`

The implementation of the Baum-Welch algorithm illustrates the elegance and conciseness of the Scala programming language. The constructor requires a fourth parameter to describe the minimum rate of change of the estimate of the likelihood between iterative calls, as shown in the following code snippet:

```
class BaumWelchEM config: HMMConfig, obs: Array[Int], numIters: Int, eps: Double) extends HMMModel(HMMLambda(config), obs) {  
    val state = HMMState(lambda, numIters)  
}
```

The λ model has to be initialized with the configuration parameters (number of observations, number of states, and number of symbols). The matrices A and B and the initial state probabilities pi are initialized with a uniform random generator [0, 1], Matrix.fillRandom, as shown here:

```
object HMMLambda {  
    def apply(config: HMMConfig): HMMLambda = {  
        val A = Matrix[Double](config._N)  
        A.fillRandom(0.0)  
        val B = Matrix[Double](config._N, config._M)  
        B.fillRandom(0.0)  
        val pi = Array.fill(config._N)(Random.nextDouble)  
        new HMMLambda(A, B, pi, config._T)  
    }  
}
```

The maximum likelihood, maxLikelihood, is computed as part of the constructor to ensure a consistent state:

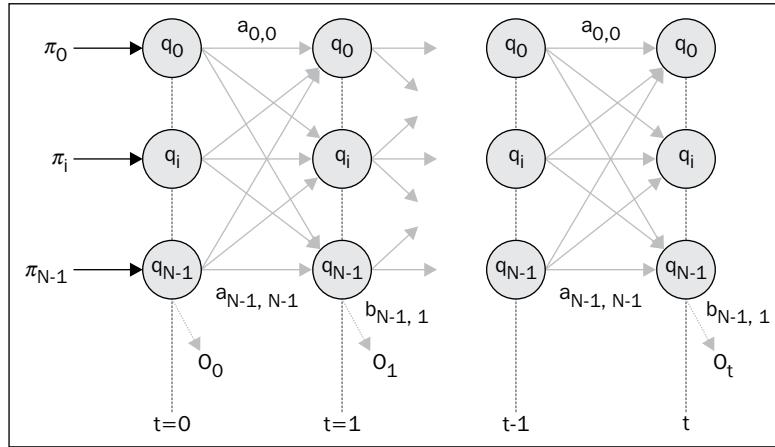
```
var likelihood = frwrdBckwrldLattice  
Range(0, state.maxIters) find(_ => {  
    lambda.estimate(state, obs) //1  
    val _likelihood = frwrdBckwrldLattice //2  
    val diff = likelihood - _likelihood //3  
    likelihood = _likelihood  
    diff < eps //4  
}) match {  
    case Some(index) => maxLikelihood  
    ...  
}
```

The computation of the likelihood requires the estimation of the transition matrix A and emission matrix B (line 1). The training process iterates by traversing the lattice forward and backward until the likelihood reaches a local or global maximum. The λ model is updated using the estimate method (line 1). The method computes the likelihood of the sequence of states (line 2) and then compares it with the likelihood computed in the previous iteration (line 3). The method exits if the difference between two consecutive likelihood values meets the convergence criteria eps (line 4).

The `estimate` method of the `HMMLambda` class updates the λ model (A , B , and π):

```
def estimate(state: HMMState, obsIdx: Array[Int]): Unit = {
    pi = Array.tabulate(config._N)(i => state.Gamma(0, i) )
    HMMConfig.foreach(config._N, i => {
        var denominator = state.Gamma.fold(dim_1, i)
        HMMConfig.foreach(config._N, k =>
            A += (i, k, state.DiGamma.fold(dim_1, i, k)/denominator)
        )
        denominator = state.Gamma.fold(config._T, i)
        HMMConfig.foreach(config._N, k => B += (i, k, state.Gamma.
        fold(config._T, i, k, obsIdx)/denominator))
    })
}
```

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:



Visualization of HMM graph lattice for the Baum-Welch algorithm

The iteration across the lattice is implemented by the `frwrdBckwrdLattice` method (line 2). The lattice is traversed ahead using the `Alpha` instance class (line 1), and backward using the `Beta` instance class (line 2):

```
def frwrdBckwrdLattice: Double = {
    val _alpha = Alpha(lambda, obs).alpha //1
    val _beta = Beta(lambda, obs) //2
    val a = _alpha.alphaBeta
    val b = _beta.alphaBeta
```

```

Gamma.update(a, b) //3
DiGamma.update(a, b, A, B, obs) //4
_alpha.alpha
}

```

The method returns the alpha coefficient and computes the new values for the `Gamma` (line 3) vector and `DiGamma` (line 4) matrix. These `HMMState` methods are omitted for the sake of clarity.

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 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. The algorithm is known as the **Viterbi algorithm**. Given a sequence of states $\{q_t\}$ and sequence of observations $\{o_j\}$, 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: Definition of delta function:

$$\delta_t(i) = \max_{q_j \in \{0, T-1\}} p(q_{0:T-1} = S_i, O_{0:T-1} | \lambda)$$

M13: Initialization of delta:



$$\delta_0(i) = \pi_i b_i(O_0) \psi_0(i) = 0 \forall i$$

M14: Recursive computation of delta:

$$\delta_t(i) = \max_i (\delta_{t-1}(i) \cdot a_{ij} \cdot b_j(O_t)) \quad \psi_t(i) = \arg \max_i (\delta_{t-1}(i) \cdot a_{ij})$$

M15: Computation of the optimum state sequence Q :

$$q_t^* = \psi_{t+1}(q_{t+1}^*) \quad q_t^* = \arg \max_i \delta_T(i)$$

The constructor of the Viterbi algorithm, `ViterbiPath`, is similar to the algorithms of the first two canonical forms, and therefore, inherits `HMMInference`. The purpose of the Viterbi algorithm is to compute the optimum sequence given a set of observations and a λ model by maximizing the delta, `maxDelta`:

```
class ViterbiPath(_lambda: HMMLambda, _state: HMMState, _obs: Array[Int]) extends HMMInference(_lambda, _state, _obs) {
    val maxDelta = recurse(lambda.getT, 0)
    ...
}
```

The recursive method that implements [M14] and [M15] steps is invoked by the constructor:

```
def recurse(t: Int, j: Int): Double = {
    var maxDelta = initial((t, j)) //1
    if( maxDelta == -1.0) {
        if(t != obs.size) {
            maxDelta = maxBy(lambda.getN, //2 [M14]
                s => recurse(t-1, s)* lambda.A(s, j)* lambda.B(j, obs(t)))
        }
        val idx =maxBy(lambda.getT, i =>recurse(t-1 ,i)*lambda.A(i,j))
        //3 [M14]

        state.psi += (t, j, idx) //4
        state.delta += (t, j, maxDelta) //5
    }
    else { //6
        maxDelta = 0.0
        val index =maxBy(lambda.getN, i => {
            val delta = recurse(t-1 ,i)
            if( delta > maxDelta) maxDelta = delta
            delta
        })
        state.QStar.update(t, index) //7
    }
}
maxDelta
}
```

Once initialized (line 1), the maximum value of delta, `maxDelta`, is computed recursively by applying the formula [M14] at each state, `s`, using Scala's `maxBy` method (line 2). Next, the index of the column of the transition matrix `A` corresponding to the maximum of delta is computed (line 3). The last step is to update the matrix `psi` (line 4) (with respect to delta (line 5)). Once the step `t` reaches the maximum number of observation labels (line 6), the optimum sequence of states q^* is computed [M15] (line 7). Ancillary methods are omitted.

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 class `HMM` 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 pseudocontinuous values is converted (or categorized) 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. The HMM can be potentially used as part of a computation workflow, and therefore, has to implement the pipe operator (`PipeOperator`).

There are two different constructors for the `HMM` class. The first constructor uses the λ model as input (evaluation (CF1) and decoding (CF3)):

```
class HMM[@specialized T <% Array[Int]](lambda: HMMLambda, form: HMMForm, maxIters: Int) (implicit f: DblVector => T) extends PipeOperator[T, HMMPredictor] {
    val state = HMMState(lambda, maxIters)
    ...
}
```

The `HMMForm` enumerator is used to specify the canonical form of the HMM solution:

```
object HMMForm extends Enumeration {
    type HMMForm = Value
    val EVALUATION, DECODING = Value
}
```

The conversion of `DblVector` to a type `T` is required only if the evaluation and decoding canonical form uses actual observation values as argument. The `f` function is then used to discretize the double values into a sequence of index of the observations.

The `HMMPredictor` type consists of a tuple log probability (or likelihood) of observations and index of sequence of observations:

```
type HMMPredictor = (Double, Array[Int])
```

The HMM has three canonical forms instead of the two forms of most classifiers.

The second canonical form, training, is implemented by defining a second constructor for the `HMM` class, as follows:

```
object HMM {
    def apply[T <% Array[Int]](config: HMMConfig, obs: Array[Int],
        form: HMMForm, maxIters: Int, eps: Double)
        (implicit f: DblVector => T): HMM[T] = {
        val baumWelchEM = new BaumWelchEM(config, obs, maxIters, eps)
        new HMM[T](baumWelchEM.lambda, form, maxIters)
    }
}
```

The `decode` (with respect to `evaluate`) method implements the third (with respect to the first) canonical form of HMM. Both methods take a sequence of indices for observations as an argument.

```
def decode(obsIdx: Array[Int]): HMMPredictor = (ViterbiPath(lambda,
    state, obsIdx).maxDelta, state.QStar())
def evaluate(obsIdx: Array[Int]): HMMPredictor = (-Alpha(lambda,
    obsIdx).logProb, obsIdx)
```

The data transformation `| >` encapsulates the evaluation and decoding forms in order to preserve its meaning. The observation, `obs`, is automatically converted into a sequence of indices to each observation (line 1) by the `DblVector => T` discretization function, which is an implicit parameter of the `HMM` class.

```
def |> : PartialFunction[DblVector, HMMPredictor] = {
    case obs: DblVector if(obs != null && obs.size > 2) => {
        form match {
            case EVALUATION => evaluate(obs) //1
            case DECODING => decode(obs) //1
        }
    ...
}
```

Normalized probabilities input



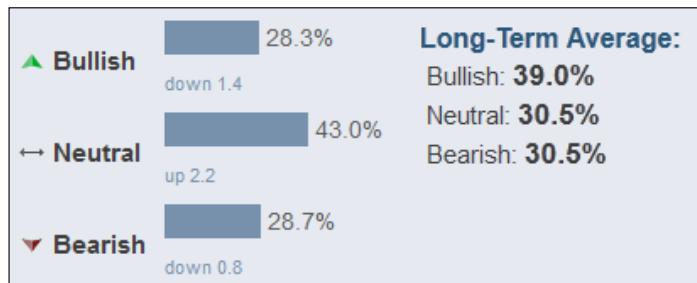
You need to make sure that the input probabilities for the λ model for evaluation and decoding canonical forms are normalized—the sum of the probabilities of all the states for the π vector and A and B matrices are equal to 1. This validation code is omitted in the example code.

Test case

Our 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 form).

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 *Appendix A, Basic Concepts*.

Let's select the ratio of 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
t_0	0.38	0.15	0.47	2.53	1.0
t_1	0.41	0.25	0.34	1.68	0.53
t_2	0.25	0.35	0.40	0.71	0.0
....					

The sequence of non-normalized observations (ratio of bullish sentiment over bearish sentiment) is defined in a CSV file as follows:

```

final val OBS_PATH = "resources/data/chap7/obs.csv"

final val NUM_SYMBOLS = 6
final val NUM_STATES = 5
final val EPS = 1e-3
final val MAX_ITERS = 250

val srcObs = Source.fromFile(OBS_PATH)
val obs = srcObs.getLines.map(_.toDouble).toSeq //1
val config = new HMMConfig(obs.size, NUM_STATES, NUM_SYMBOLS)
val min = obs.min
val delta = obs.max - min
val obsSeq = obs.map( x => (x - min)/delta) //2
    .map(x =>(x*NUM_SYMBOLS).floor.toInt) //3
HMM[Array[Int]](config, obsSeq, EVALUATION, MAX_ITERS, EPS) match {
  case Some( hmm) => //4
    Display.show(s"Lambda: ${hmm.getModel.toString}", logger)
  ...
}

```

The sequence of observations is loaded from the CSV file (line 1) before being normalized (line 2). The discretization converts the normalized bullish sentiment/bearish sentiment ratio in six levels (integers) [0,-5] (line 3). The instantiation of the `HMM` class for the ratio levels (`Array[Int]`) generates the λ model (`A`, `B`, and `pi`) (line 4).

The following is a state-transition 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

The hidden Markov model for time series analysis

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 Kalman filter* section in *Chapter 3, Data Preprocessing*, HMM requires data to be somewhat 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. Moreover, moving averaging techniques, discrete Fourier transforms, and generic Kalman filters require the states to be continuous with linear dependencies, although the extended Kalman filter can approximate nonlinear states.

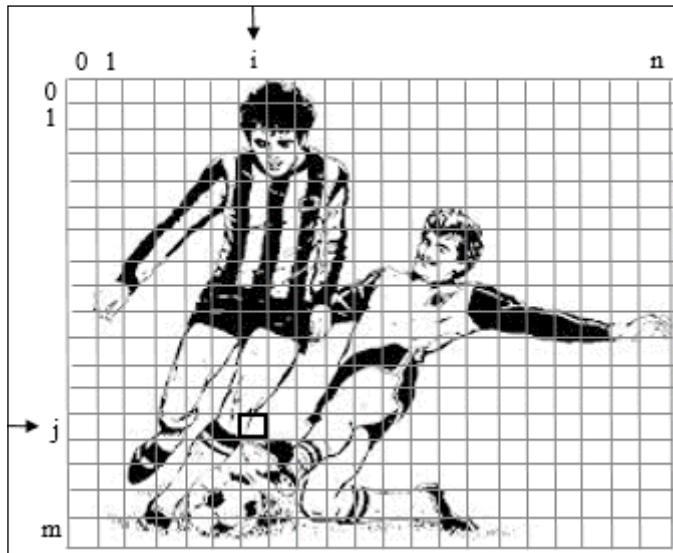
Conditional random fields

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 as found.

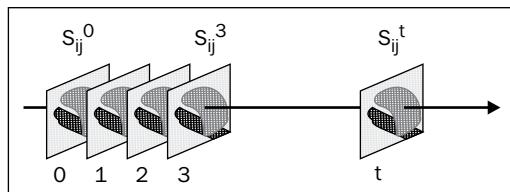
Let's consider a concrete example to understand the conditional relation between the observations and the label data.

Introduction to CRF

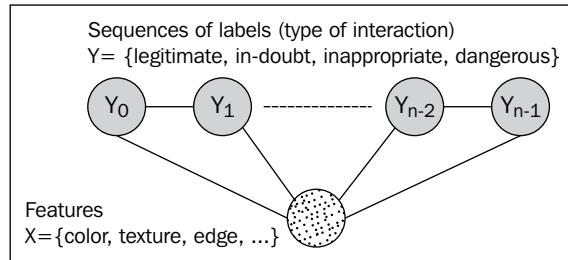
Let's consider the problem of detecting a foul during a soccer game using a combination of video and audio. The objective is to assist the referee and analyze the behavior of the players to determine whether an action on the field is dangerous (red card), inappropriate (yellow card), in doubt to be replayed, or legitimate. The following image is an example of segmentation of a video frame for image processing:



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 image segment S_{ij} is one of the labels that are 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 by the following model graph:

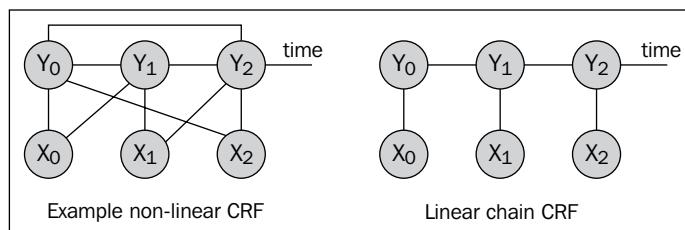


Undirected graph representation of CRF for soccer infraction detection

Conditional random fields (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 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 CRF 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 much like HMM. Such a model is known as the **linear chain structured graph CRF** or **linear chain CRF** for short.



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 to stay consistent with the HMM described in the previous section.

Linear chain CRF

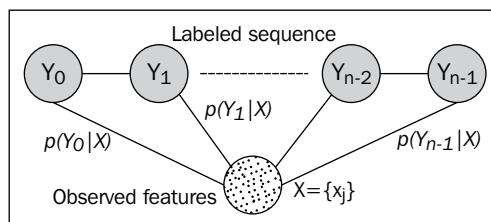
Let's consider a random variable $X=\{x_i\}_{0:n-1}$ representing n observations and a random variable Y representing a corresponding sequence of labels $Y=\{y_j\}_{0:n-1}$. The hidden Markov model estimates the joint probability $p(X, Y)$ as any generative model requires the enumeration of all the sequences 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 observations X 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 :



Linear, conditional, random field undirected graph

The Markov property simplifies the conditional probabilities of Y , given X , by considering only the neighbor labels $p(Y_1 | X, Y_j, j \neq 1) = p(Y_1 | X, Y_{\sigma}, Y_2)$ 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_i):** These are strictly positive, real value functions that 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 $\{X_i\}$, the previous label Y_{t-1} , the current label Y_t 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 as the state transition factors a_{ij} in HMM but without a direct probabilistic interpretation.
- State feature functions s_i are potential functions that take the sequence of features $\{X_i\}$, the current label Y_t , and the index i . They play a similar role as 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.

CRF conditional distribution

- The log probability of a label's sequence y , given an observation x :

$$\log f_i(y_{i-1}, y_i, x, i) = w_c + \sum_{i=0}^{K-1} w_i t_i(y_{i-1}, y_i, x, i) + \sum_{j=0}^{K-1} \mu_j s_j(y_i, x, i)$$

- Transition feature functions with $I(a) = 1$ if a true, 0 otherwise:

$$t_i(y_{i-1}, y_i, x, i) = I(y_{i-1} = l_1) \cdot I(y_i = l_2) \cdot I(x = 0)$$

- Using the notation:

$$F_i(y, x) = \sum_{j=0}^{K-1} f_j(y_{j-1}, y_j, x, i) \quad \log p(y | x, \lambda) \propto \sum_{j=0}^{K-1} w_j F_j(x, y)$$

- Conditional distribution of labels y , given x , using the Markov property:

$$p(y | x, w) = \frac{1}{Z(x)} e^{\sum_{j=0}^{K-1} w_j F_j(x, y)} \quad z(x) = \sum_{i=0}^{N-1} \sum_{j=0}^{K-1} w_j F_j(x, y)$$



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.

CRF 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 as an introductory test case 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 rating for any stock. These analysts used different rating schemes from buy/hold/sell; A, B, C rating; and stars rating to market perform/neutral/Market underperform. For this example, the rating is normalized as follows:

- 0 for a strong sell, (or F or 1 star rating)
- 1 for sell (D, 2 stars, 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 from strong buy (A, 5 stars, highly recommended, and so on)

Here is an example 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.

Scope of the application

Ratings from analysts are updated every day through different protocols (feed, emails, 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 (optional keyword)
- Rating (optional previous rating)
- To
- Rating (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 initiate) rating for a stock does not have the fields 4 and 5 defined.

For example:

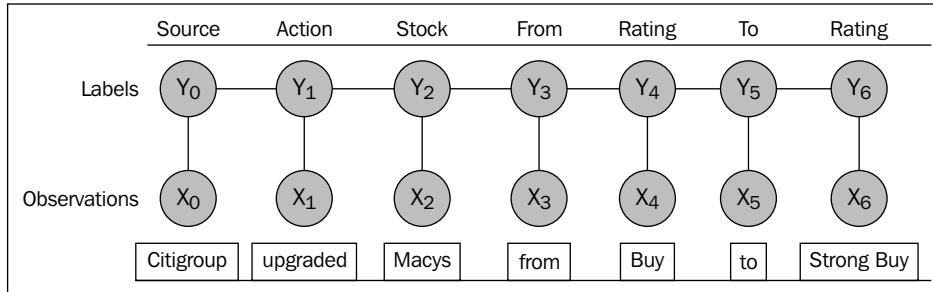
```
Citigroup // Y(0) = 1
upgraded // Y(1)
Macys // Y(2)
from // Y(3)
Buy // Y(4)
to // Y(5)
Strong Buy //Y(6) = 7
```



Tagging

Tagging 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 (adjective, pronoun, verb, proper name, and so on) to a word in a sentence [7:11].

A training sequence can be visualized with 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.

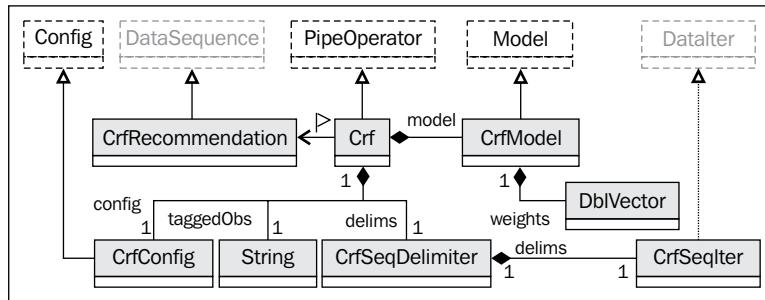
Software design

The implementation of the conditional random fields follows the design template for classifier, as explained in the *Design template for classifiers* section in *Appendix A, Basic Concepts*.

Its key components are as follows:

- A `CrfModel` model of the type `Model` is initialized through training during the instantiation of the classifier.
- The predictive or classification routine is implemented as a data transformation that implements the `PipeOperator` trait.
- The conditional random field classifier, `Crf`, has four parameters: the number of labels (or number of features), `nLabels`; configuration of type `CrfConfig`; the sequence of delimiters of the type `CrfSeqDelimiter`; and the labeled (or tagged) observations `taggedObs`.
- The `CrfRecommendation` class is required by the CRF library to implement the `DataSequence` interface. The class is used to recommend (or estimate) the next label.
- `CrfSeqIter` implements the `DataIter` iteration interface to traverse the labeled data sequence during training, as required by the CRF library.

The key software components of the conditional random fields are described in the following UML class diagram:



UML class diagram for the conditional random fields

The `DataSequence` and `DataIter` interfaces are grayed out to indicate that these are defined in the IITB's CRF Java library.

Implementation

The test case uses the IITB's CRF Java implementation from the Indian Institute of Technology at Bombay by Sunita Sarawagi. The JAR files can be downloaded from Source Forge (<http://sourceforge.net/projects/crf/>).

The library is available as JAR files and source code. Some of the functionality, such as the selection of a training algorithm, is not available through the API. The components (JAR files) of the library are as follows:

- CRF for the implementation of the CRF algorithm
- LBFGS for limited-memory Broyden-Fletcher-Goldfarb-Shanno nonlinear optimization of convex functions (used in training)
- CERN Colt library for manipulation of a matrix
- GNU generic hash container for indexing

The training of the conditional random field for sequences requires defining a few key interfaces:

- `DataSequence` to specify the mechanism to access observations and labels for training and test data
- `DataIter` to iterate through the sequence of data created using the `DataSequence` interface
- `FeatureGenerator` to aggregate all the features types

These interfaces have default implementations bundled in the CRF Java library [7:12].



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 interest such as bioinformatics or process control.

Building the training set

The first step is to implement the structure of the training sequence, which implements the `DataIter` interface. The training file consists of a pair of files:

- Raw recommendations (such as *Raymond James upgrades Gentiva Health Services from Underperform to Market perform*)
- Tagged recommendations (such as Raymond James [1] upgrades [2] Gentiva Health Services [3], from [4] Underperform [5] to [6] Market perform [7])

Let's define the model for the CRF classifier. As mentioned earlier, the model for the CRF is similar to the logistic regression model and consists of the `weights` parameter:

```
class CrfModel(val weights: DblVector) extends Model
```

The tagged recommendations file requires a delimiter class, `CrfSeqDelimiter`. It delineates the sequence of observations using the following parameters:

- `obsDelim` is a regular expression to break down data input into a sequence of observations
- `labelsDelim` generates a sequence of labels from the data input
- `trainingDelim` generates a sequence of training tuples from the training set

The `CrfSeqDelimiter` class is defined as follows:

```
class CrfSeqDelimiter(val obsDelim: String, val labelsDelim: String,  
val trainingDelim: String)
```

The main purpose of the IITB CRF Java library's `DataIter` interface is to define the methods to iterate through a sequence of data, tags, or observations. The three methods are as follows:

- `hasNext` tests if the sequence has another entry
- `next` returns the next data or entry in the sequence and increments the iterator cursor
- `startScan` initializes the `DataIter` iterator

The `CrfSeqIter` sequence iterator uses the `iitb.segment.DataCruncher` class to read a training set from a file (a file with tagged words):

```
class CrfSeqIter(val nLabels: Int, val input: String, val delim: SeqDelimiter) extends DataIter {
    lazy val trainData = DataCruncher.readTagged(nLabels, input, input,
        delim.obsDelim, delim.labelsDelim, delim.trainingDelim, new labelMap)

    override def hasNext: Boolean = trainData.hasNext
    override def next: DataSequence = trainData.next
    override def startScan: Unit = trainData.startScan
}
```

The `trainData` training set is initialized only once when any of the `DataIter` overridden methods is invoked. The class is merely an adapter to the generation of the training set.

Generating tags

The second step consists of selecting the mechanism and class to generate the features observations. The extraction of the features from any data set requires implementation of the `FeatureGenerator` interface in order to access all the features observations from any kind of features.

Our problem is a simple linear tagging of data sequences (recommendations from analysts). Therefore, we can use the `iitb.Model.FeatureGenImpl` default implementation. Our tagging class, `TaggingGenerator` makes `FeatureGenImpl` as a subclass and specifies the model specification as a `CompleteModel`. The IITB CRF library supports both linear chain model of `CompleteModel` with a single edge iterator and the nested chain CRF model of the type `NestedModel` with a nested edge iterator. The complete model does not make any assumption regarding the independence between labels Y :

```
val addFeature = true
class TaggingGenerator (val nLabels: Int) extends FeatureGenImpl(new
    CompleteModel(nLabels), nLabels, addFeature)
```

The class is defined within the scope of the `Crf` class and does not have to be exposed to the client code. The last parameter of `FeatureGenImpl`, `addFeature`, is set as true to allow the tags of dictionary to be built iteratively during the training.

Extracting data sequences

The `CrfTrainingSet` class implements the `DataSequence` interface. It is used to access all the raw analyst's recommendations and rating regarding stocks. The class needs to implement the following methods:

- `set_y` to assign a label index to a position k
- `y` to retrieve a label y at position y
- `x` to retrieve an observed feature vector at position k
- `length` to retrieve the number of entries in the sequence

The `CrfTrainingSet` class can be implemented as follows:

```
class CrfTrainingSet(val nLabels: Int, val entry: String, val delim: String) extends DataSequence {  
    val words = entry.split(delim)  
    val map = new Array[Int](nLabels)  
  
    override def set_y(k: Int, label: Int): Unit = map(k) = label  
    override def y(k: Int): Int = map(k)  
    override def length: Int = words.size  
    override def x(k: Int): Object = words(k)  
}
```

The class takes an analyst's recommendation regarding a stock, `entry`, as an input and breaks it down into words, using the delimiter or regular expression, `delim`.

CRF control parameters

The execution of the CRF algorithm is controlled by a wide variety of configuration parameters. For the sake of simplicity, we use the default configuration parameters, `CrfConfig`, to control the execution of the learning algorithm, with the exception of the following four variables:

- Initialization of the weights, w_i , using either a predefined or a random value between 0 and 1 (default 0)
- Maximum number of iterations used in the computation of the weights during the learning phase `maxIters` (default 50)
- The scaling factor `lambda` for the L_2 penalty function, used to reduce observations with a high value (default 1.0)
- Convergence criteria, `eps`, used in computing the optimum values for the weights w_j (default 1e-4)

Advanced 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.

The test case does not assume any dependence between states:

```
class CrfConfig(w0: Double, maxIters: Int, lambda: Double, eps: Double) extends Config
```

Putting it all together

The objective of the training is to compute the weights w_j that maximize the conditional log-likelihood without the L_2 penalty function.

 Conditional log-likelihood for a linear chain CRF training set, $D = \{(x_i, y_i)\}_{0:n-1}$ is given as follows:

$$\mathcal{L}(w, D) = -\sum_{i=0}^{n-1} \log p(y_i | x_i, w)$$

 Learning: Maximization of loss function and L2 penalty is given as follows:

$$w^* = \arg \max_{\lambda} [\mathcal{L}(w, D) + \lambda \|w\|^2] \quad \lambda = \frac{1}{2\sigma^2}$$

Maximizing the log-likelihood function \mathcal{L} is equivalent to minimizing the loss with L_2 penalty. The function is convex, and therefore, any variant gradient descent (greedy) algorithm can be applied iteratively.

The `Crf` class implements the `learning`, `train`, and `classification` methods. Like any other classifiers, `crf` implements the `PipeOperator` trait; so, the classification can be included in a workflow. The class also implements the `Supervised` trait to force the developer to define a validation routine for the CRF:

```
class Crf(nLabels: Int, config: CrfConfig, delims: SeqDelimiter, taggedObs: String) extends PipeOperator[String, Double] with Supervised[String] {
    val features = new TaggingGenerator(nLabels) //1
    lazy val crf = new CRF(nLabels, features, config.params) //2
    val model: Option[CrfModel] = {
```

```
    features.train(seqIter) //3
    Some(new CrfModel(crf.train(seqIter))) //4
}
...

```

The computation of the CRF weights during training uses either methods defined in IITB's CRF library or methods described in the previous sections.

Once the features have been extracted from the data sequence input file (line 1), the CRF algorithm is instantiated (line 2) with the number of labels, extracted features, and the configuration. The model is trained using the iterator for features `seqIter` (line 3), and then returns a `CrfModel` instance (vector of weights) (line 4) if training succeeds, `None` otherwise.

The predictive method implements the data transformation operator, `|>`. It takes a new observation (analyst's recommendation on a stock) and returns the maximum likelihood, as shown here:

```
def |> : PartialFunction[String, Double] = {
  case obs: String if(obs.length > 1 && model != None) => {
    val dataSeq = new CrfTrainingSet(nLabels,obs,delims.obsDelim)
    crf.apply(dataSeq)
  }
}
```

The data transformation implements the Viterbi algorithm to extract the best sequence of labels for a newly observed recommendation, `obs`. It invokes the `apply` method of the `iitb.crf.CRF` class. The code to validate the arguments/parameters of the class and methods are omitted along with the exception handler for the sake of readability.

Tests

The client code to execute the test consists of defining the number of labels (tags for recommendation), the L_2 penalty factor, `LAMBDA`, and the delimiting string:

```
val LAMBDA = 0.5; val EPS = 1e-3
val NLABELS = 9; val MAX_ITERS = 100; val W0 = 0.7
val PATH = "resources/data/chap7/rating"

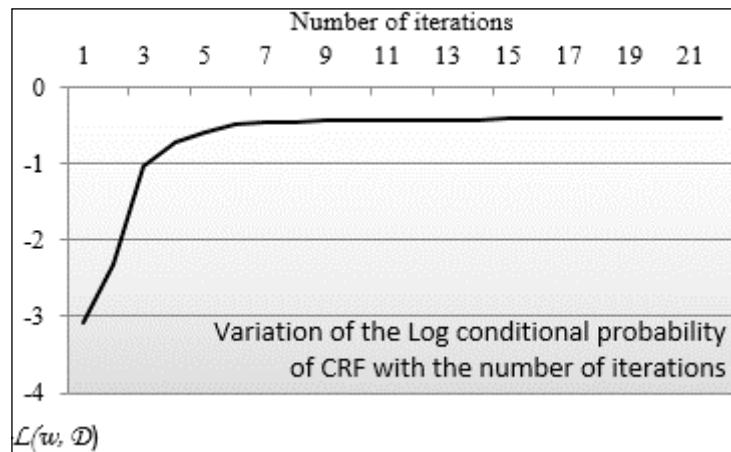
val config = CrfConfig(W0, MAX_ITERS, LAMBDA, EPS)
val delimiters = CrfSeqDelimiter(",\t/-():.;'?#`&_", "//", "\n")

Crf(NLABELS, config, delimiters, PATH).weights match {
  case Some(weights) => weights
  case None => { ... }
}
```

For these tests, the initial value for 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 input data files, rating.raw and rating.tagged.

The training convergence profile

The first training run discovered 136 features from 34 analyst's 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 :



Visualization of the log conditional probability of CRF during training

The training phase converges fairly 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 have required a larger number of iterations during training to converge.

Impact of the size of the training set

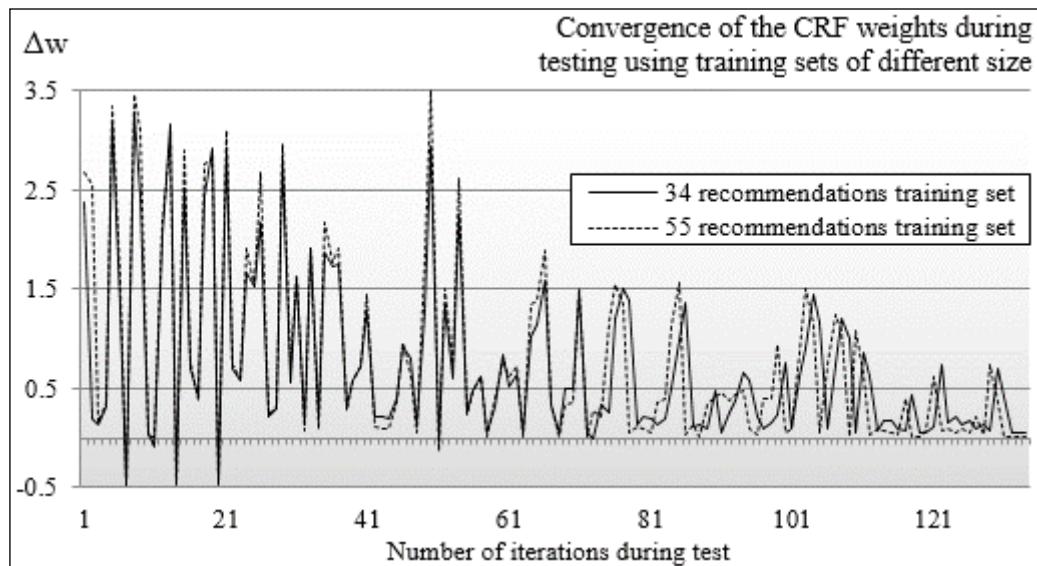
The second test evaluates the impact of the size of the training set on the convergence of the training algorithm. It consists of computing the difference Δw of the model parameters (weights) between two consecutive iterations $\{w_i\}^{t+1}$ and $\{w_i\}^t$:

$$\Delta w = \sum_{i=0}^{D-1} (w_i^{t+1} - w_i^t)$$

The test is run on 163 randomly chosen recommendations using the same model but with two different training sets:

- 34 analyst stock recommendations
- 55 stock recommendations

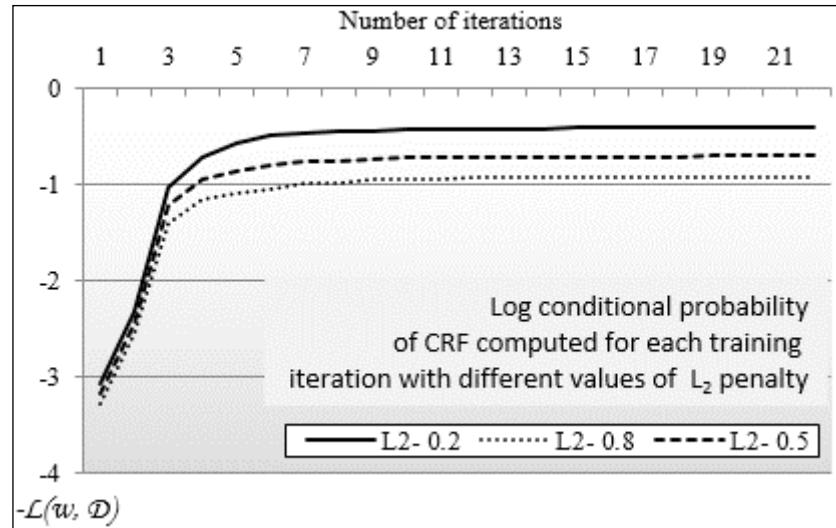
The larger training set is a super set of the 34 recommendations set. The following graph illustrates the comparison of features generated with 34 and 55 CRF training sequences:



The disparity between the test runs using two different size of training set 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 different value of λ . The following charts plot $\log [p(Y | X, w)]$ for different values of $\lambda = 1/\sigma^2$ (0.2, 0.5, and 0.8):

Impact of the L₂ penalty on 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 L₂ regularization factor, the higher the conditional probability.

The variation of the analysts' recommendations within the training set is fairly 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. HMM assumes the transition probabilities matrix to be constant.

HMM learns the transition probabilities a_{ij} on its own by providing 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 $O(N^2T)$. The training of HMM using the Baum-Welch algorithm is $O(N^2TM)$, where M is the number of iterations.

There are several options to improve the performance of HMM:

- Avoid multiplication by 0 in the emission probabilities matrix by using sparse matrices or keeping tab of the null entries
- Try to train HMM on a *relevant* subset of the training data, particularly in the case of tagging

The training of the linear chain conditional random fields is implemented using the same dynamic programming techniques as HMM implementation (Viterbi, forward-backward passes). Its time complexity for training T data sequence, N labels y , 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 [7:13].

Summary

In this chapter, we had a closer look at modeling sequences of observations with hidden states with the two most commonly used algorithms:

- Generative hidden Markov model (HMM) to maximize $p(X, Y)$
- Discriminative conditional random field (CRF) to maximize $\log p(Y | X)$

HMM is a special form of Bayes Network and requires the observations to be independent. Under these circumstances, the HMM is fairly easy to estimate, which is not the case for CRF.

You learned how to implement three dynamic programming techniques, Viterbi, Baum-Welch, and alpha/beta algorithms in Scala. These algorithms are routinely used to solve optimization problems and should be an essential component of your algorithmic toolbox.

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 of *Chapter 6, Regression and Regularization*, you learned the concept of hyperplanes used 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 are 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, you will discover 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 the support vector regression compares to the linear regression

Support vector machines are formulated as a convex optimization problem. Therefore, the mathematical foundation of these algorithms is described for reference.

Kernel functions

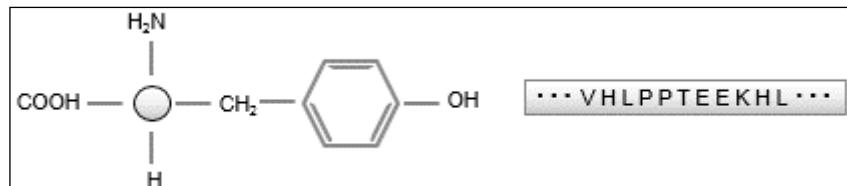
Every machine learning model introduced in this book so far assumes that observations are represented by a feature vector of a fixed size. However, some real-world applications such as text mining or genomics do not lend themselves to this restriction. The critical element of the process of classification is to define a similarity or a 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].

Overview

The concept of kernel methods may be a bit odd at first to a novice. It is usually better understood by using a concrete example. Let's consider the example of the classification of proteins. Proteins have different lengths and composition, but it does not prevent scientists from classifying them [8:2].

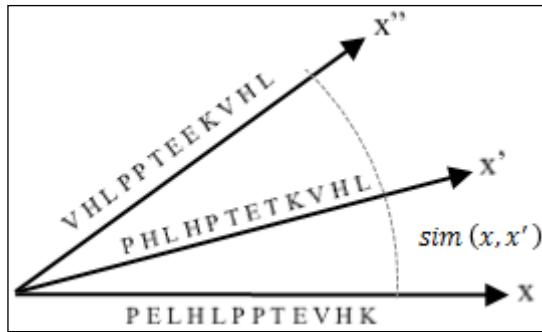
 **Proteins:**
Proteins are polymers of amino acids joined together by peptide bonds. They are composed of a carbon atom bonded to a hydrogen atom, another amino acid, or a carboxyl group.

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 picture illustrates the molecular (left) and encoded (right) representation of a protein:



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:



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 represent 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' can be defined as the number of identical elements (character and position) between the two annotations divided by the maximum length between the two annotations:

$$\text{sim}(x_{cp}, x'_{c'p'}) = \frac{1}{mx} \sum_{i=1}^{mx} (c = c') \cap (p = p') mx = \max(n, n')$$

The computation of the similarity for the three proteins produces the result as $\text{sim}(x, x') = 6/12 = 0.50$, $\text{sim}(x, x'') = 3/13 = 0.23$, $\text{sim}(x', x'') = 4/13 = 0.31$.

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.

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, non-negative 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:

- **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 appropriate 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

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 (OLS)* regression section of *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:

$$f(x|w) = w_0 + \sum_{d=1}^D w_d \phi_d(x) \quad \phi_d : \mathbf{R} \rightarrow \mathbf{R}$$

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 $\phi(x) \cdot \phi(x')$ of two features vector x and x' . A partial list of kernel methods is as follows:

The generic kernel:

$$K(x, x') = \phi(x) \cdot \phi(x') = \sum_{d=1}^D \phi_d(x) \phi_d(x')$$

The linear kernel:

$$K(x, x') = x^T x' = \sum_{d=1}^D x_i \cdot x'_i$$

The polynomial kernel with the slope γ , degree n , and constant c :

$$K(x, x') = (\gamma x^T x' + c)^n \quad \gamma > 0, c \leq 0$$



The sigmoid kernel with the slope γ and constant c :

$$K(x, x') = \tanh(\gamma x^T x' + c) \quad \gamma > 0, c \leq 0$$

The radial basis function kernel with the slope γ :

$$K(x, x') = e^{-\gamma \|x - x'\|^2} \quad \gamma > 0$$

The laplacian kernel with the slope γ :

$$K(x, x') = e^{-\gamma \|x - x'\|} \quad \gamma > 0$$

The log kernel with the degree n :

$$K(x, x') = -\log(1 + \|x - x'\|^n)$$

The list of discriminative kernel functions described earlier is just a subset of the kernel methods universe. Other types of kernels include:

- **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.

The support vector machine (SVM)

A **support vector machine (SVM)** 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 of *Chapter 6, Regularization and Regression*). 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:

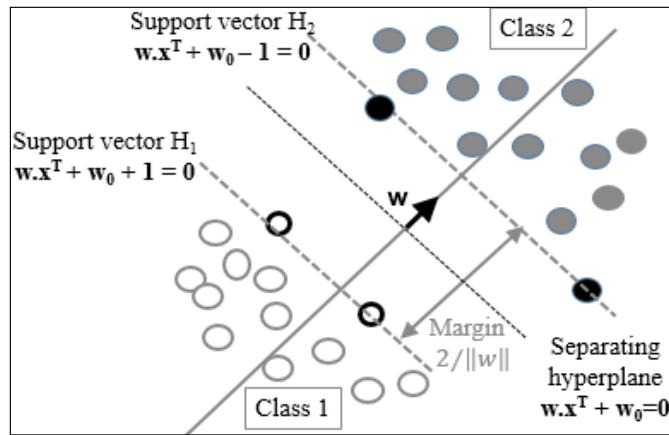
- Naturally segregated in the features space (the **separable** case)
- Intermingled with overlap (the **nonseparable** case)

It is easy to understand the concept of an optimal separating hyperplane in cases the observations are naturally segregated.

The separable case (hard margin)

The concept of separating a training set of observations with a hyperplane is better explained with a 2-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, $y = w \cdot x^T + w_0$, which sits in the midpoint between the boundary data points for 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:



Support vector machine – separable case

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**.

Support vectors equation w is represented as:

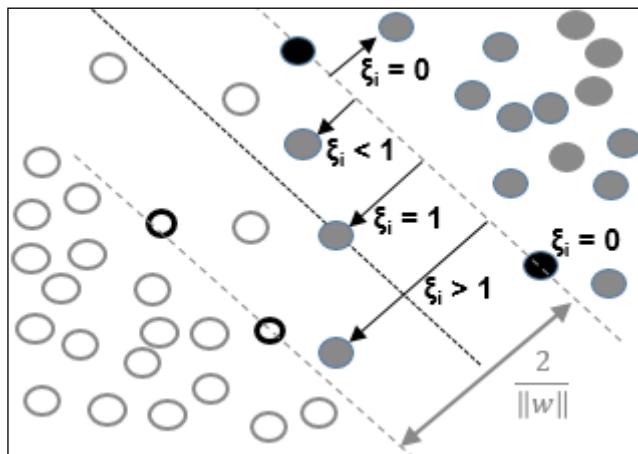
$$y_i(w^T x + w_0) \geq 1 \quad \forall i$$

 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 (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:



A support vector machine – the nonseparable case

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. Observations that belong to the class but located beyond its support vector are penalized; 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.

Optimization of the soft-margin for a linear SVM with C formulation:

$$\begin{aligned} \min_{w, \xi} & \left\{ \frac{w^T w}{2} + c \sum_{i=0}^{n-1} \xi_i \right\} \\ \xi_i & \geq 0, \quad y_i (w^T x + w_0) \geq 1 - \xi_i \quad \forall i \end{aligned}$$

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 of *Chapter 6, Regularization and Regression*). The second factor in the formula corresponds to the ubiquitous loss function. You will certainly recognize the first term as the L_2 regularization penalty with $\lambda=1/2C$.

The problem can be reformulated as the minimization of a function known as the **primal problem** [8:6].



Primal problem formulation of the support vector classifier:

$$\min_{w, w_0} \left\{ \frac{w^T w}{2} + c \sum_{i=0}^{n-1} L_i \right\} L_i = |1 - y_i (w^T x + w_0)|$$

The C penalty factor can be thought of as the inverse of the L_2 regularization factor. The loss function L is then 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 **ν -SVM** (or Nu-SVM) is an alternative formulation to the C-SVM. The formulation is more descriptive than C-SVM; ν 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].



ν -SVM formulation of a linear SVM:

$$\min_{w, p, \xi} \left\{ \frac{w^T w}{2} - p \frac{1}{un} \sum_{i=0}^{n-1} \xi_i \right\}$$

$$\xi_i \geq 0, \quad y_i (w^T x + w_0) \geq p - \xi_i \quad \forall i$$

Here, ρ is a margin factor used as a 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, it has been 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, 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:

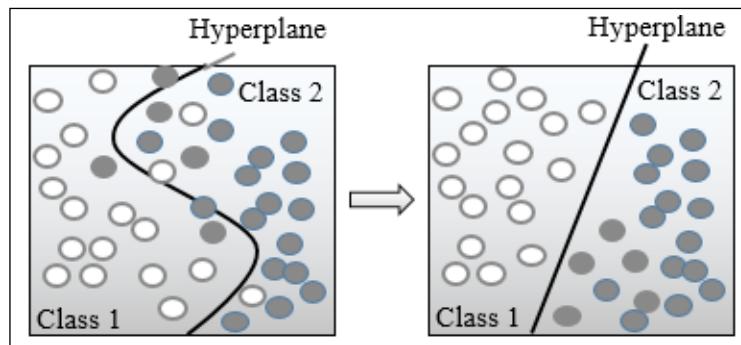


Illustration of the Kernel trick in an SVM

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 the support vector, using the basis function, ϕ :

$$y_i (w^T \phi(x) + w_0) \geq 1 - \xi_i \quad \xi_i \geq 0 \quad \forall i$$

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].

[Optimal hyperplane for the SVM dual problem:

$$w^* = \sum_{i=0}^{n-1} \alpha_i y_i \phi(x_i)$$


] Hard margin formulation for the SVM dual problem:

$$y_i (w^T \cdot \phi(x) + w_0) = y_i \left(\sum_{i=0}^{n-1} \alpha_i y_i K(x_i, x) + w_0 \right) \geq 1$$

$$K(x_i, x) = \phi(x_i) \phi(x) \forall i$$

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 $C_0=1$ in a two-dimension space. The polynomial kernel function on two vectors, $x=[x_1, x_2]$ and $z=[x'_1, x'_2]$, is decomposed into a linear function in a dimension 6 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 classifier (SVC)

Support vector machines can be applied to classification, anomalies detection, and regression problems. Let's dive into the support vector classifiers first.

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 and 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 $O(n^2)$. 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 SVM using resilient distributed datasets (refer to the *Apache Spark* section of *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 , 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 (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 SVM.

Let's create a Scala wrapper to the LIBSVM library to improve its flexibility and ease of use.

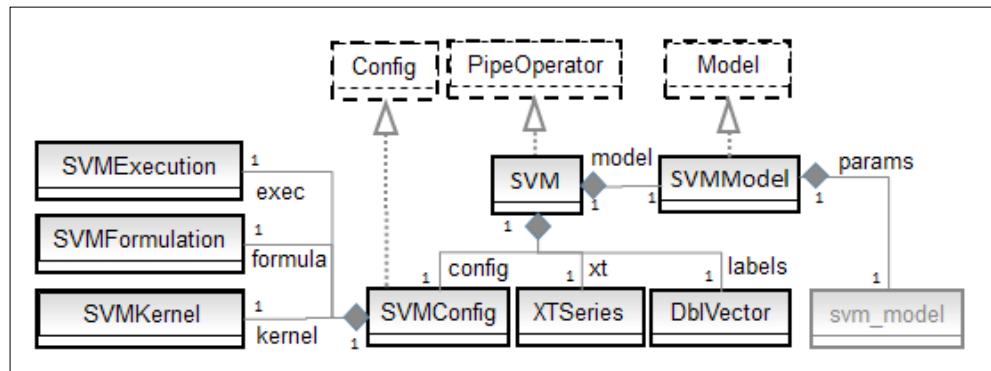
Software design

The implementation of the support vector machine algorithm uses the design template for classifiers (refer to the *Design template for classifier* section in *Appendix A, Basic Concepts*).

The key components of the implementation of an SVM are as follows:

- A model `SVMModel` of the type `Model`, which is initialized through training during the instantiation of the classifier. The model class is an adapter to the `svm_model` structure defined in LIBSVM.
- A predictive or classification routine is implemented as a data transformation extending the `PipeOperator` trait.
- The support vector machine class `SVM` has three parameters: the configuration wrapper of the type `SVMConfig`, the features/time series of the type `XTSeries`, and the target or labeled values `DblVector`.
- The configuration (the type `SVMConfig`) consists of three distinct elements: `SVMExecution` that defines the execution parameters such as 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:



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 (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 execution parameters (convergence criteria, number of folds for cross-validation, and so on) using the **SVMExecution** class

SVM Formulation

The instantiation of the configuration consists of initializing the LIBSVM parameter, **param**, by the SVM type, kernel, and the execution context selected by the user.

Each of the SVM parameters case class extends the generic trait, **SVMConfigItem**:

```
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 the LIBSVM.

The formulation of the SVM algorithm is defined by classes implementing the `SVMFormulation` trait:

```
sealed trait SVMFormulation extends SVMConfigItem {
    def update(param: svm_parameter): Unit
}
```

The list of 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 formulation class, `CSVCFormulation`, 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 formulation classes, `NuSVCFormulation`, `OneSVCFormulation`, and `SVRFormulation`, 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 {
    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 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 with 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 for 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

SVM execution

The `SVMExecution` class defines the configuration parameters for the execution of the training of the model, namely, the convergence factor, `eps` for the optimizer, the size of the cache `cacheSize`, and the number of folds, `nFolds` used during cross-validation:

```
class SVMExecution(cacheSize: Int, eps: Double, nFolds: Int) extends  
SVMConfigItem {  
    override def update(param: svm_parameter): Unit = {  
        param.cache_size = cacheSize  
        param.eps = eps  
    }  
}
```

The cross-validation is performed only if the `nFolds` value is greater than 1.

SVM implementation

We are finally ready to create the configuration class, `SVMConfig`, 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)
    kernel.update(param)
    exec.update(param)
}
```

The instantiation of `SVMConfig` initialized the internal LIBSVM list of configuration parameters through a sequence of update calls.

Next, let's implement the first support vector classifier for the two-class problems. As with any other data transformation, the parameterized class `SVM` implements the `PipeOperator`, as follows:

```
class SVM[T <% Double] (config: SVMConfig, xt: XTSeries[Array[T]], labels: DblVector) extends PipeOperator[Array[T], Double] {
    type Feature = Array[T]
    type SVMNodes = Array[Array[svm_node]]
```

This class has the same parameters as other classifiers presented in the previous chapters: a configuration, `config`, an input time series, `xt`, and labeled data, `labels`. The types are added for convenience. The internal types, `Feature` and `SVMNodes`, are added for convenience.

The LIBSVM type, `svm_node`, is the indexed value of an element of the feature vector in a particular observation:

```
public class svm_node implements java.io.Serializable {
    public int index;
    public double value;
}
```

The type `SVMNodes` defined in the scope of `SVM` class is the representation of a two-dimensional array of features vector elements by observations. The next step is to implement the training procedure. The training is executed during the instantiation of the `svm` class. The SVM model, `svmModel`, is defined as a tuple or pair (`svmModel`, `accuracy`) with the following:

- The `svmModel` is the model defined in LIBSVM
- `accuracy` computed during an N-folds cross-validation if the number of folds, `nFolds`, has been set as one of the parameters of `SVMExecution`

Consider the following code:

```
class SVMModel(val svmmode1: svm_model, val accuracy: Double) extends Model
```

The instantiation of SVC is hidden from the client code. It is executed during the instantiation of the class, so a client code does not have to be aware of the LIBSVM types. Consider the following code:

```
val model: Option[SVMModel] = {  
    val problem = new svm_problem //1  
    problem.l = xt.size;  
    problem.y = labels  
    problem.x = new SVMNodes(xt.size)  
  
    val dim = dimension(xt)  
    xt.zipWithIndex.foreach( xt_i => { //2  
        val svm_col = new Array[svm_node](dim)  
        xt_i._1.zipWithIndex  
            .foreach(xi => {  
                val node = new svm_node  
                node.index= xi._2  
                node.value = xi._1  
                svm_col(xi._2) = node  
            })  
        problem.x(xt_i._2) = svm_col  
    })  
    Some(svm.svm_train(problem, config.param, accuracy(problem)))//3  
}
```

The first step in the creation of the model is to define the SVM problem, `problem`, in the context of LIBSVM (line 1): length of the time series, labeled data, and input observations. The time series has to be converted into the LIBSVM internal class, `svm_nodes` (line 2), to complete the initialization of the problem. The Scala method, `zipWithIndex`, is used to access the index of each observation (time series entry). Finally, the model and the computed accuracy are returned as a tuple (line 3) after processing by the `svm_train` training method.

The accuracy is the ratio of true positive plus the true negative over the size of the test sample (refer to the *Key metrics* section of *Chapter 2, Hello World!*). It is computed through cross-validation only if the number of folds is initialized in the `SVMExecution` configuration class as 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. Here is the essential part of the implementation of accuracy (problem: `svm_problem`):

```
val target = new Array[Double](labels.size)
svm.svm_cross_validation(problem, config.param, config.exec.nFolds,
target)
val rawAccuracy = target.zip(labels)
    .filter(z => Math.abs(z._1-z._2) < config.eps)
rawAccuracy.size.toDouble/labels.size
```

The Scala `filter` weeds out the observations that were poorly predicted. This minimalist implementation is good enough to start exploring the support vector classifier.

C-penalty and margin

The first evaluation consists of understanding the impact of the penalty factor C to 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 `SVC` class, as follows:

```
def margin: Option[Double] = model match {
  case Some(m) => {
    val wNorm = m.svmmode1.sv_coef(0)
      .foldLeft(0.0)((s, r) => s + r*r) //1
    if(wNorm < config.eps) None
    else Some(2.0/Math.sqrt(wNorm)) //2
  }
  ...
}
```

The first instruction (line 1) computes the sum of the squares, `wNorm`, of the residuals $r = y - f(x | w)$. The margin (line 2) 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) factor C:

```
def evalMargin(observations: DblMatrix, labels: DblVector, c: Double):  
  Unit = {  
    val config = SVMConfig(CSVCFormulation(c), RbfKernel(GAMMA)) //3  
    val xt = XTSeries[DblVector](observations)  
    val svc = SVM[Double](config, xt, labels)  
    svc.margin match {  
      case Some(margin) => Display.show("Margin $margin", logger)  
      ...  
    }  
  }
```

This test uses the default execution parameters, `cache_size= 25000` and `eps=1e-15`. Therefore, the 3rd value of `SVMConfig`, `exec`, is not specified in the `SVMConfig`. `apply` constructor (line 3). 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 consists of the following two training sets of an equal size, N:

- **First training set:** data points generated as $y = x(1 + r/5)$ for the label 1, r being a randomly generated number over the range [0,1]
- **Second training set:** randomly generated data point $y = r$ for the label of -1

Consider the following code:

```
def generate: (DblMatrix, DblVector) = {  
  val z = Array.tabulate(N)(i =>  
    Array[Double](i, i*(1.0 + 0.2*Random.nextDouble))  
  ) ++  
  Array.tabulate(N)(i => Array[Double](i, i*Random.nextDouble))  
  (z, Array.fill(N)(1.0) ++ Array.fill(N)(-1.0))  
}
```

The `evalMargin` method is executed for a predefined value of gamma and the value C ranging from 0 to 5:

```
val gamma = 0.8; val N = 100  
val values = generate  
Range(0, 50).foreach(i => evalMargin(values._1, values._2, i*0.1))
```

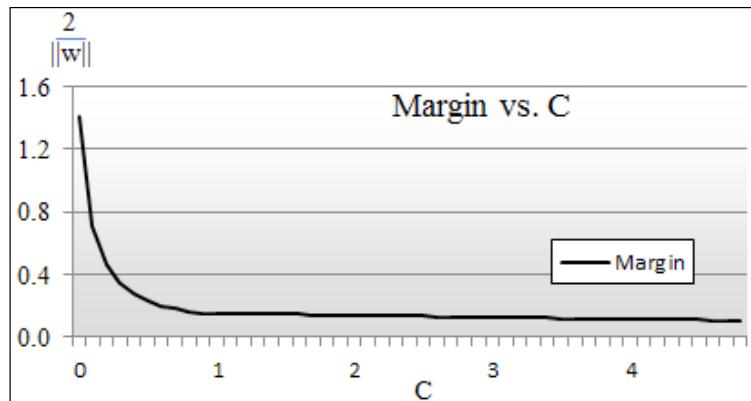
val vs. final val

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 C-penalty for an SVC

As expected, the value of the margin decreases as the penalty term C increases. The C penalty factor is related to the L₂ 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.

First, the prediction method for the SVM class is implemented by overriding the pipe operator data transformation, |>:

```
def |> : PartialFunction[Feature, Double] = {
    case x: Feature if(x != null && x.size==dimension(xt) && model
        != None && model.get.accuracy >= 0.0) =>
        svm.svm_predict(model.get.svmmode, toNodes(x))
}
```

The prediction model relies on the `svm_predict` LIBSVM to compute the output value. It takes two parameters: `svmmode` and an array of `svm_nodes` (line 1). The conversion of a feature from the type `DblVector` to an array of the `svm_nodes` LIBSVM is performed by the `toNodes` method:

```
def toNodes(x: Feature): Array[svm_node] =
    x.zipWithIndex
        .foldLeft(new ArrayBuffer[svm_node])((xs, f) => { //2
            val node = new svm_node
            node.index = f._2
            node.value = f._1
            xs.append(node)
            xs
        }).toArray
```

A fold is used to construct the array of `svm_nodes` from the feature vector, `x`. The nodes (elements of the sparse matrix of the `svm_node` LIBSVM) are generated from the new observation `x` (line 1). The model extracted from the training of the model (instantiation of SVM) and the sparse matrix `nodes` are the input to the LIBSVM predictor, `svm_predict` (line 2).

The predictor is used by the test code for evaluating the different kernel functions. Let's create a method to evaluate and compare these kernel functions. All we need is the following:

- A training set, `observations`, by features of the type `DblMatrix`
- A test set, `test`, of the type `DblMatrix`
- A set of `labels` for the training set, taking the value 0 or 1
- A kernel function `kF`

Consider the following code:

```
def evalKernel(features: DblMatrix, test: DblMatrix, labels: DblVector, kF: SVMKernel): Double = {
    val config = SVMConfig(new CSVCFormulation(C), kF) //3
    val xt = XTSeries[DblVector](features)
    val svc = SVM[Double](config, xt, labels) //4
    val successes = test.zip(labels)
        .count(tl => {
            Try((svc |> tl._1) == tl._2)
            match { case Success(n) => true
                    case Failure(e) => false }
        })
    successes.toDouble/test.size //6
}
```

The support vector classifier, `svc`, is configured with the default execution parameters and the C-formulation (line 3), and trained (instantiated) with the observed features, `xt` and the output, `labels` (line 4).

Once trained, `svc` is used to predict the value for a test sample extracted from the original dataset (line 5). Finally, the number of successful test observations is counted and the accuracy is computed as the ratio of the successful prediction over the size of the test sample (line 6).

In order to compare the different kernels, let's generate three datasets of the size $2N$ for a binomial classification using the following random generator, $y = variance*x - mean$:

```
def genData(variance: Double, mean: Double): DblMatrix =
    val adjVariance1 = variance*Random.nextDouble - mean
    val adjVariance2 = variance*Random.nextDouble - mean
    Array.fill(N)(Array[Double](adjVariance, adjVariance2))
}
```

A training set is then created as the aggregate of two classes of data points:

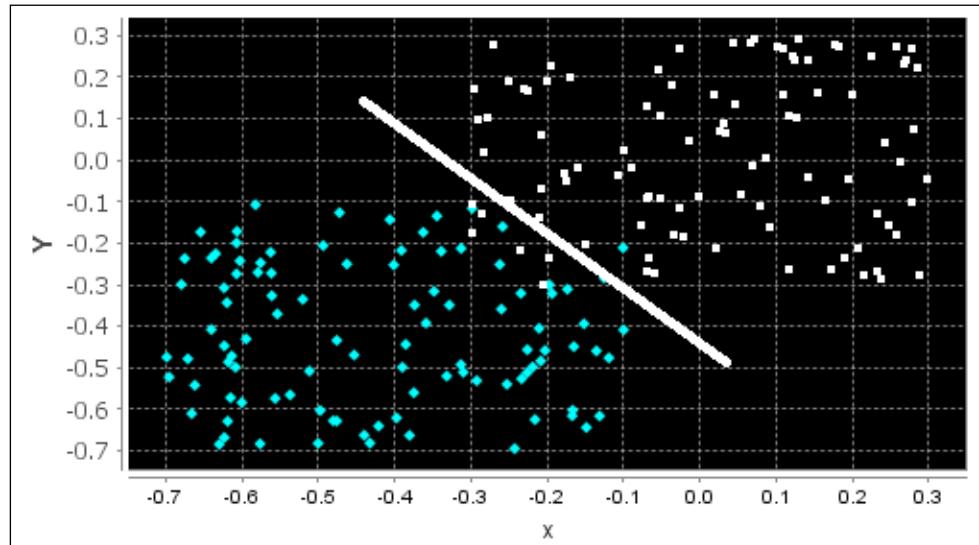
- Random data points (x,y) with variance a and mean $1-b$ with label 0.0
- Random data points with variance a and mean $b-1$ with label 1.0

Consider the following code

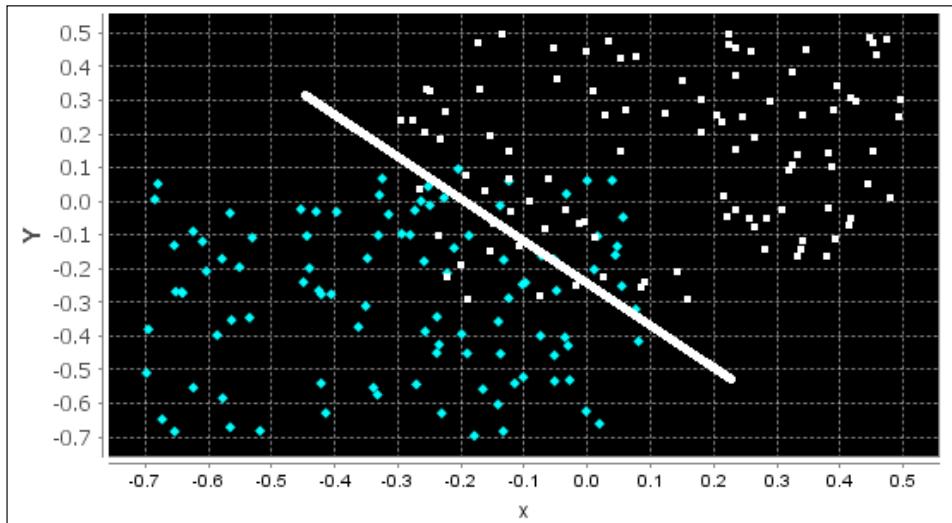
```
val trainingSet = genData(a,b) ++ genData(a,1-b)
val labels = Array.fill(N)(0.0) ++ Array.fill(N)(1.0)
```

The parameters a and b are selected from two groups of training data points with various degree of separation to illustrate the separating hyperplane.

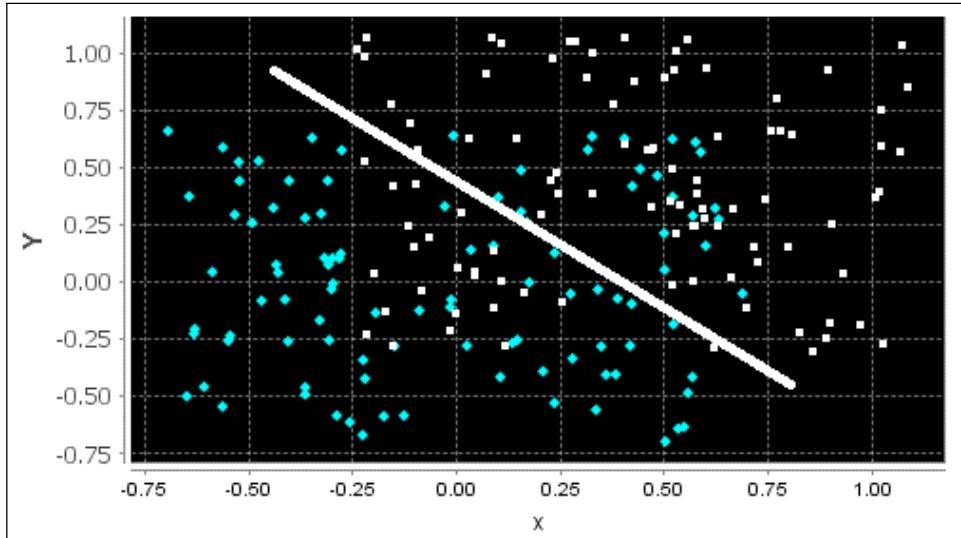
The following chart describes the high margin; the first training set generated with the parameters $a = 0.6$ and $b = 0.3$ illustrates the highly separable classes with a clean and distinct hyperplane:



The following chart describes the medium margin; the parameters $a = 0.8$ and $b = 0.3$ generate two groups of observations with some overlap:



The following chart describes the low margin; the two groups of observations in this last training are generated with $a = 1.4$ and $b = 0.3$ and show a significant overlap:



The test set is generated in a similar fashion as the training set, as they are extracted from the same data source:

```

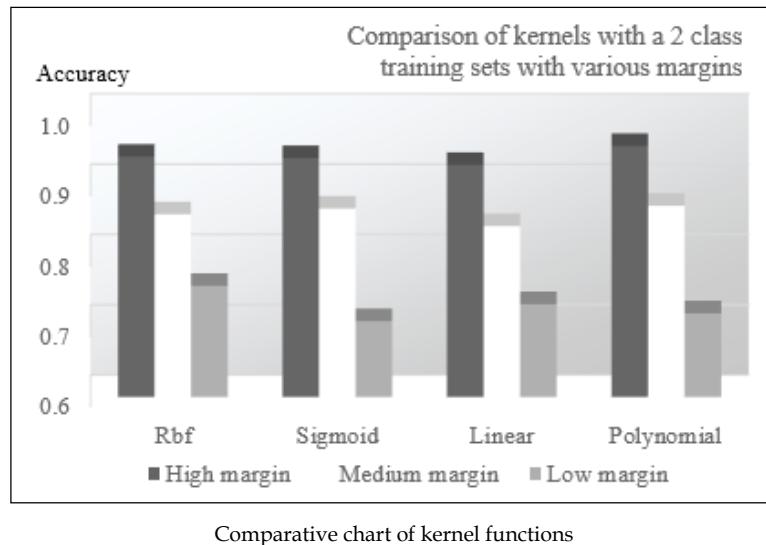
val EPS = 0.0001; val C = 1.0; val GAMMA = 0.8
val N = 100; val COEFO = 0.5; val DEGREE = 2

val a = 1.4; val b = 0.3 //3 sets of values
val trainSet = genData(a, b) ++ genData(a, 1-b)
val testSet = genData(a, b) ++ genData(a, 1-b)
val labels = Array.fill(N)(0.0) ++ Array.fill(N)(1.0)

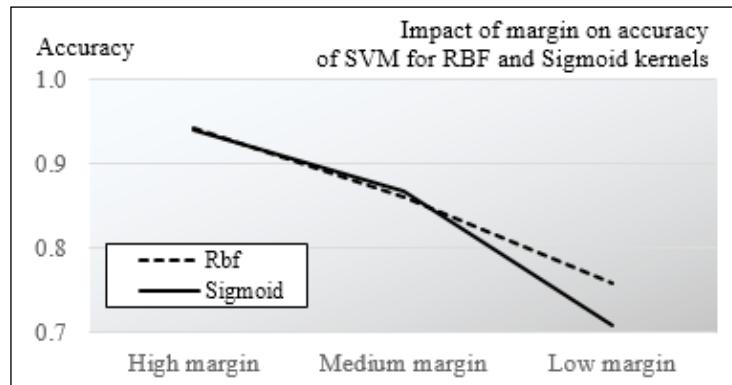
val result =
  evalKernel(trainSet,testSet, labels, RbfKernel(GAMMA)) :::
  evalKernel(trainSet,testSet, labels, SigmoidKernel(GAMMA)) :::
  evalKernel(trainSet,testSet, labels, LinearKernel) :::
  evalKernel(trainSet,testSet, labels, PolynomialKernel(GAMMA, COEFO,
DEGREE)) :: List[Double]()
  
```

The value of the kernel function parameters are arbitrary selected from text books. The evalKernel method defined earlier is applied to the three training sets: 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 slightly more accurate results. As expected, the accuracy decreases as the margin decreases. A decreasing margin is a sign that the cases are not easily separable, affecting the accuracy of the classifier:



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 the purpose of illustrating the margin between classes of observations. Real-world datasets may produce different results.

In summary, there are four steps in creating 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 margin and compare kernel methods. Let's use the support vector classifier for a real-world financial application.

Application to 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 cash flow and pay out its dividends over the long term.

Features and labels

We need to select any subset of the following financial technical analysis metrics (refer to the *Terminology* section in *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 earning 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 features are normalized with values 0 and 1.

The labeled output, dividend changes, is categorized as follows:

- -1, if dividend is cut by more than 5 percent
- 0, if dividend is maintained within 5 percent
- +1, if 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 C_1 = stable or decreasing dividends and class C_2 = increasing dividends; represented by dividendsA
- Class C_1 = decreasing dividends and class C_2 = stable or increasing dividends; represented by dividendsB

The different tests are performed with a fixed set of configuration parameters C and GAMMA and a 2-fold validation configuration:

```
val path = "resources/data/chap8/dividendsA.csv"
val C = 1.0; val GAMMA = 0.5; val EPS = 1e-3; val NFOLDS = 2

val extractor = relPriceChange :: debtToEquity :: dividendCoverage
               :: cashPerShareToPrice :: epsTrend :: dividendTrend
               :: List[Array[String] => Double] () //1
```

The components of the extractor are functions that convert a set of fields in the input .csv file into double floating point values:

```
val xs = DataSource(path, true, false, 1) |> extractor
val config = SVMConfig(new CSVCFormulation(C),
                      RbfKernel(GAMMA),
                      SVMExecution(EPS, NFOLDS))
val features = XTSeries.transpose(xs.take(xs.size-1))//2
val svc = SVM[Double](config, features, xs.last)

svc.accuracy match { //3
  case Some(acc) => Display.show(s"Accuracy: $acc", logger)
  case None => { ... }
}
```

The different fields are extracted from the dividendsA.csv file using the `DataSource` extractor with a filter (line 1). The purpose of the test A is to create a separating hyperplane (the predictive model) for dividendsA, that is, companies that cut or maintained their dividends and the companies that increased their dividends. The last field in the extractor is the labeled output. The observed features time series is created from all the fields extracted from the .csv file except the last. The time series has to be transposed to use the format required by LIBSVM (line 2). Once the support vector classifier is created, you can retrieve the accuracy of the cross-validation (line 3).

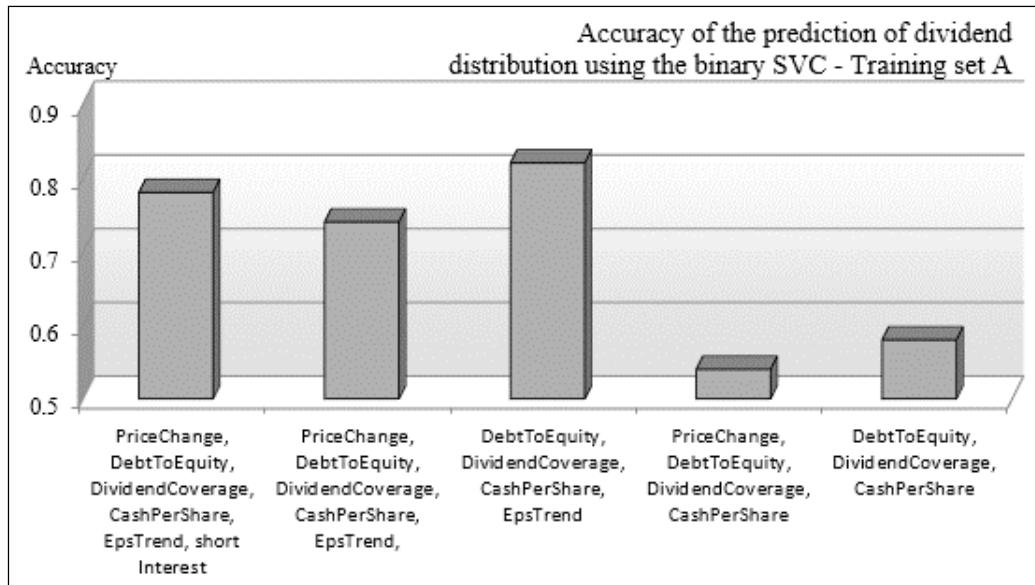


LIBSVM scaling

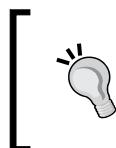
LIBSVM supports feature normalization known as scaling, prior to training. The main advantage of scaling is to avoid attributes in greater numeric ranges dominating those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during the calculation. In our examples, we use the normalization of the time series, `XTSeries.normalize`. 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 :: List[Array[String] => Double] ()
```



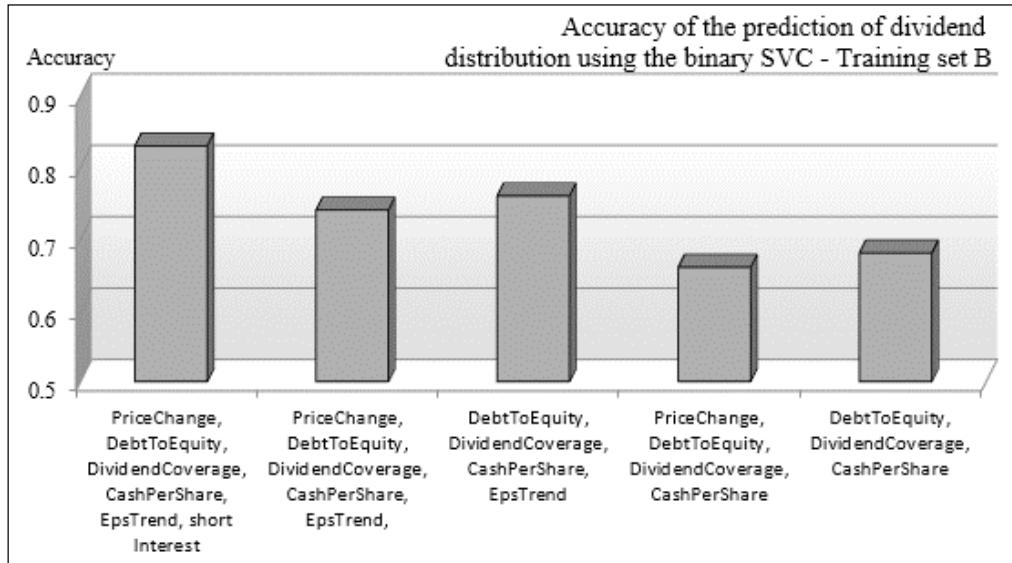
The test demonstrates that the selection of the proper features set is the most critical step in applying the support vector machine, and any other model for that matter, to classification problems. In this particular case, the accuracy is also affected by the small size of the training set. The increase in the number of features also reduces the contribution of each specific feature to the loss function.



N-fold cross-validation

The cross-validation in this test example uses only 2 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 tests, the feature *eps* (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 about the risk a company entails to eliminate its dividend altogether? Such a scenario is rare, and those 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 and the other class is replaced by a reference point known as the **SVC origin**. The outliers (or abnormal) observations reside beyond (or outside) the support vector of the single class:

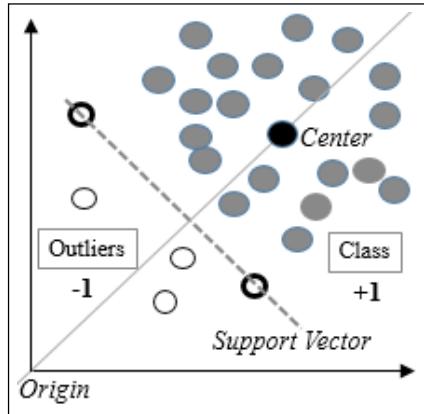


Illustration of the one-class SVC

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 dividend and +1 for all other companies

The test is executed against the dataset `resources/data/chap8/dividends2.csv`. 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 binary 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 NFOLDS = 2
val path = "resources/data/chap8/dividends2.csv"

val xs = DataSource(path, true, false, 1) |> extractor
val config = SVMConfig(new OneSVCFormulation(NU),
                      RbfKernel(GAMMA),
                      SVMExecution(EPS, NFOLDS))
val features = XTSeries.transpose(xs.dropRight(1))
val svc = SVM[Double](config, features, xs.last.map(filter(_)))
svc.accuracy match {
  case Some(acc) => Display.show("Accuracy: $acc", logger)
  case None => { ... }
}
```

The test is executed with the following features: `relPriceChange`, `debtToEquity`, `dividendCoverage`, `cashPerShareToPrice`, and `epsTrend`.

The model is generated with the accuracy of 0.821. This level of accuracy should not be a surprise; the outliers (companies that completely 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 dividend) in the original input file.

Where the labeled observations are available, the one-class support vector machine is an excellent alternative to clustering techniques.

Definition of 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 dividend. There is no guarantee that this assumption is indeed always valid.

Support vector regression (SVR)

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:

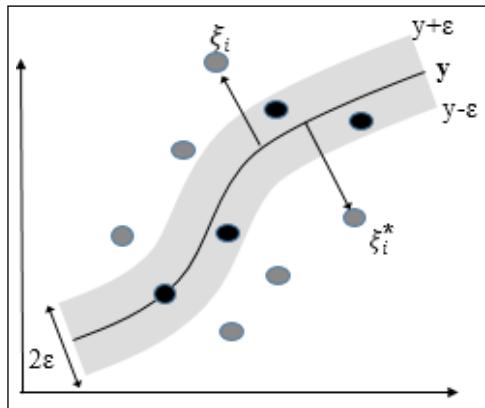
- ϵ -VR (sometimes called C-SVR)
- v-SVR

For the sake of consistency with the two previous cases, the following test uses the ϵ (or C) formulation of the support vector regression.

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.



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.

$$\varepsilon \text{ SVR:}$$

$$\min_{w, \xi, \xi^*} \left\{ \frac{w^T w}{2} + c \sum_{i=0}^{n-1} (\xi_i + \xi_i^*) \right\}$$

$$-\in -\xi_i^* \leq w^T \phi(x_i) + w_0 - y_i \leq \varepsilon + \xi_i \quad \forall i$$


Here, ε is the insensitive error function.
The ε -SVR regression equation:

$$\hat{y}(x) = \sum_{i=0}^{n-1} \alpha_i K(x_i, x) + \hat{w}_0$$

Let's reuse the SVM class to evaluate the capability of the SVR, compared to the linear regression (refer to the *Ordinary least squares (OLS) regression* section of *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 of *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 indexes, `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

```

val path = "resources/data/chap8/SPY.csv"
val C = 1; val GAMMA = 0.8; val EPS = 1e-3; val EPSILON = 0.1 //1

val price = DataSource(path, false, true, 1) |> adjClose //2
val priceIdx = price.zipWithIndex
    .map(x => (x._1.toDouble, x._2.toDouble))
val linRg = SingleLinearRegression(priceIdx) //3
val config = SVMConfig(new SVRFormulation(C, EPSILON),
RbfKernel(GAMMA)) //3
val labels = price.toArray

val xt = XTSeries[DblVector](
    Array.tabulate(labels.size)(Array[Double](_))) //4
val svr = SVM[Double](config, xt, labels) //5
collect(svr, linRg, price) //6

```

The `collect` method invokes the predictive method for the support vector regression (line 7) and the linear regression model (line 8), and then buffers the results along with the original observation, `price` (line 9).

```

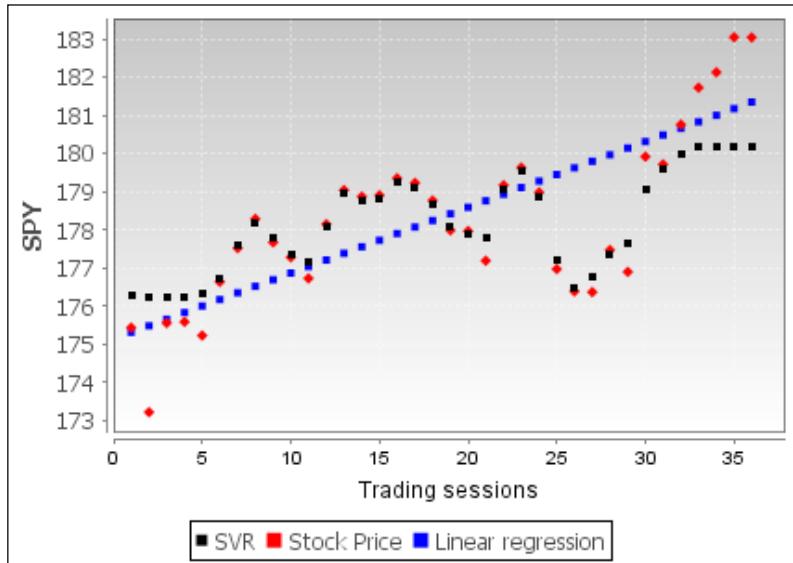
def collect(svr: SVM_Double,
            lin: SingleLinearRegression[Double],
            price: DblVector): Array[XYTSeries] = {

    val collector = Array.fill(3)(new ArrayBuffer[XY])
    Range(1, price.size-2).foldLeft(collector)((xs, n) => {
        xs(0).append((n, (svr |> n.toDouble).get)) //7
        xs(1).append((n, (lin |> n).get)) //8
        xs(2).append((n, price(n))) //9
        xs
    }).map(_.toArray)
}

```

The types `XY=(Double, Double)` and `XYTSeries=Array[(Double, Double)]` have already been defined in the *Primitive types* section of *Chapter 1, Getting Started*.

The results are displayed in the following graph, generated using the JFreeChart library. The code to plot the data is omitted because it is not essential to the understanding of the application.



Comparative plot 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, SVM is related to the logistic regression; the hinge loss in SVM is similar to the loss in the logistic regression [8:15].

Performance considerations

You may have already observed that the training of a support vector regression model on a large data set is time consuming. The performance of the support vector machine depends on the type of optimizer (for example, sequential minimal optimization) selected to maximize the margin during training.

- A linear model (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 SVM, you should have developed a basic understanding of the power and complexity of the tuning and configuration parameters of the different variants of SVM.

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:

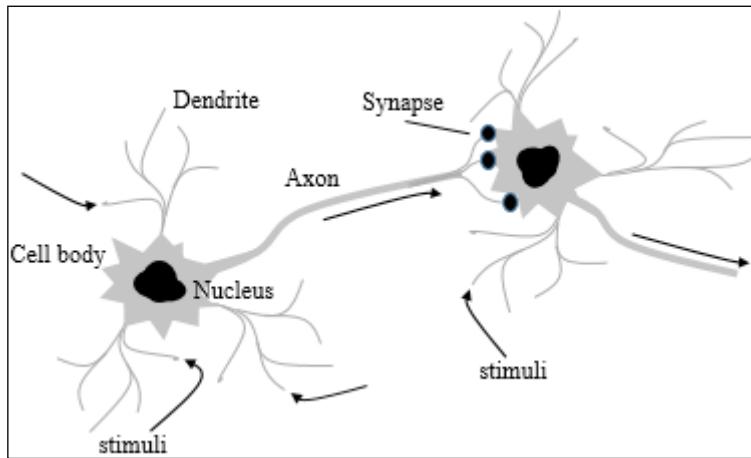
- The concept and elements of the **multilayer perceptron (MLP)**
- How to train a neural network using error backpropagation
- The evaluation and tuning of MLP configuration parameters
- Full Scala implementation of the MLP classifier
- How to apply MLP to extract correlation models for currency exchange rates

Feed-forward neural networks (FFNN)

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.

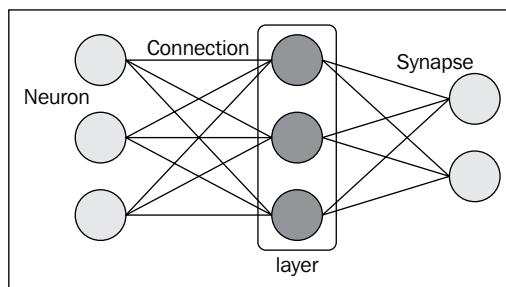
The Biological background

In biology, a neural network is composed of groups of neurons interconnected through synapses [9:1], as shown in the following image:



Neuroscientists have been especially interested in understanding how the 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 through electrical charges known as **stimuli**. This network of neurons can be represented as a simple schematic, as follows:



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

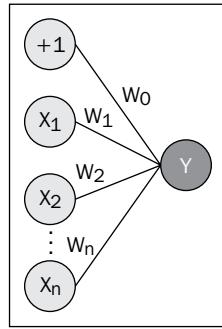
In the biological world, stimuli do not propagate in any specific direction between neurons. An artificial neural network can have the same degree of freedom. The artificial neural networks most commonly used by data scientists, have a predefined direction: from the input layer to output layers. These neural networks are known as FFNN.

The 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 *The support vector machine (SVM)* 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 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 phenomena. 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:

$$y = \sigma(w_0 + w^T x) = \frac{1}{1 + e^{-(w_0 + w^T x)}}$$

An FFNN can be regarded as a stack of layers of logistic regression with the output layer as a linear regression.

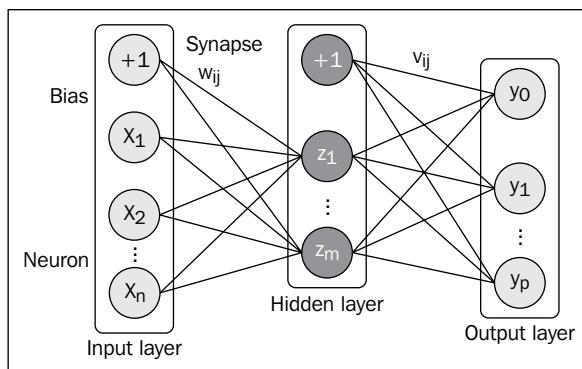
The value of the variables in each hidden layer is computed as the sigmoid of the dot product of the connection weights and the output of the previous layer. Although interesting, the theory behind artificial neural networks is beyond the scope of this book [9:4].

The multilayer perceptron (MLP)

The perceptron is a basic processing element that performs 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_o , in both linear and logistic regression, is represented with +1 in the visualization of the neural network ($w_o \cdot 1 + w_1 \cdot x_1 + w_2 \cdot x_2 + \dots$).

FFNN with no 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. Overview of the software design.
2. Description of the MLP model components.
3. Implementation of the four-step training cycle.
4. 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 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_j , and input values, x_i , processed by the **output unit activation** function h , as shown here:

$$\hat{y} = h\left(w_0 + \sum_{i=1}^n w_i x_i\right) = h(w_0 + w^T x)$$

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:

- Identity for the output layer of the binary classification or regression problem
- Sigmoid, σ , for hidden layers

- Softmax for the multinomial classification
- Hyperbolic tangent, $tanh$, for classification using zero mean

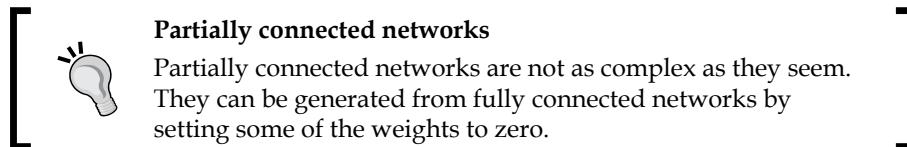
The Softmax formula is described in the next section.

The network architecture

The output layer 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:

- 1 input layer
- $(n-1)$ hidden layer
- 1 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.



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 **objective of the neural network**. The type of problem at hand defines the number of output nodes [9:5], for example:

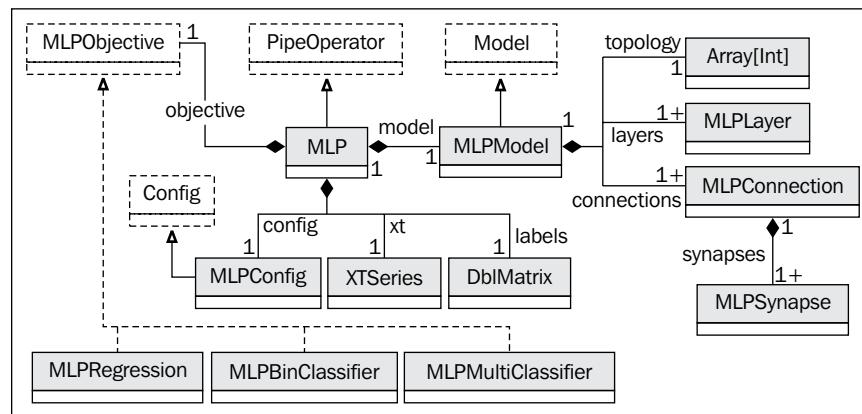
- A one-variate regression has one output node whose value is a real number $[0, 1]$
- A multivariate regression with n variables has n real output nodes
- A binary classification has one binary output node $\{0, 1\}$ or $\{-1, +1\}$
- A multinomial or K-class classification has K binary output nodes

Software design

The implementation of the MLP classifier follows the same pattern as previous classifiers (refer to the *Design template for classifiers* section in *Appendix A, Basic Concepts*):

- A model `MLPModel` of the type `Model` is initialized through training during the initialization of the classifier. The model is composed of a layer of neurons of the type `MLPLayer`, connected by synapses of the type `MLPSynapse` contained by a connector of the type `MLPConnection`.
- All of the configuration parameters are encapsulated into a single configuration class, `MLPConfig`.
- The predictive or classification routine is implemented as a data transformation, extending the `PipeOperator` trait.
- The multilayer perceptron class, `MLP`, takes three parameters: configuration instance, a features set or time series of the `XTSeries` class, and a labeled dataset of the type `DblMatrix`.

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 to understand the role and relation of the Scala classes used to build an MLP. Let's start with the implementation of the MLP model and its components.

Model definition

The purpose of the model is to completely define the network architecture. It is implemented by the `MLPModel` parameterized class, which is responsible for creating and managing the different components of the network, layers, and connections as well as the topology.

Let's establish a simple naming convention for the layers of neurons as follows:

- The input layer, `inLayer`, consists of `nInputs` neurons
- A hidden layer, `hidLayer`, has `nHiddens` neurons
- The output layer, `outLayer`, has `nOutputs` neurons

The instantiation of the class requires a minimum set of three parameters:

```
class MLPModel[T <% Double] (config: MLPConfig, nInputs: Int, nOutputs: Int) extends Model {
    val layers: Array[MLPLayer]
    val connections: Array[MLPConnection]
    val topology: Array[Int]
}
```

Besides the `config` configuration, the model class has two parameters: the number of input features, `{x}`, `nInputs`; and the number of output values, `{y}`, `nOutputs`. These three parameters are all you need to initialize the topology of the network. A model has the following attributes:

- Multiple layers of the type `MLPLayers`
- Multiple connections of the type `MLPConnection`
- A `topology` array that wires these layers and connections

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:

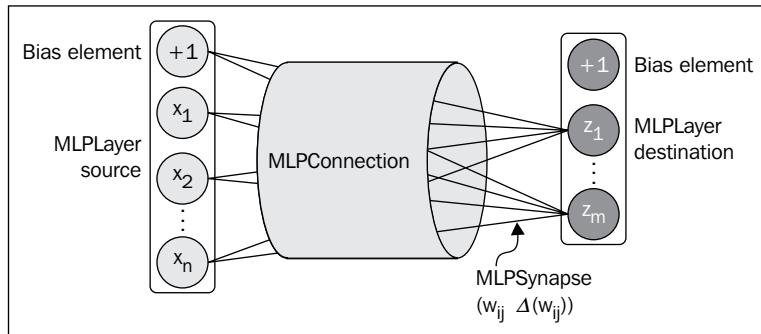
```
val topology = Array[Int](nInputs) ++ config.hidLayers ++
    Array[Int](nOutputs)
```

The sequence of hidden layers, `hidLayers`, is defined as an array of number of neurons (or nodes) per hidden layers:

```
val hidLayers: Array[Int]
```

This is an attribute of the `MLPConfiguration` class described in the next section. 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 following diagram visualizes the interaction between the different components of a model: `MLPLayer`, `MLPConnection`, and `MLPSynapse`:



Components of the MLP model

Layers

First, let's start with the definition of the layer class, `MLPLayer`, which is completely specified by its position in the network and the number of nodes it contains:

```
class MLPLayer(val id: Int, val len: Int) {
    val output = new DblVector(len) //1
    val delta = new DblVector(len) //2
    ...output.update(0, 1.0) //3
```

The `id` parameter is the order of the layer (0 for input, 1 for the first hidden layer, ..., $n-1$ for the output layer) in the network. The `len` value is the number of elements or nodes, including the bias element, in this layer. The `output` vector for the layer (line 1) is an uninitialized vector of values updated during the forward propagation, except for the first value (bias element), which is set to 1 (line 3). The `delta` vector associated to the `output` vector (line 2) is updated through the error backpropagation algorithm, described in the next section.

The output values, except the bias element, is initialized using the `set` method:

```
def set(x: DblVector): Unit = x.copyOf(output, 1)
```

Synapses

A synapse is defined as a pair of real values:

- The weight of the connection from the neuron i of the previous layer to the neuron j , w_{ij}
- The weights adjustment (or gradient of weights), Δw_{ij}

Its type is defined as `MLPSynapse`, as shown here:

```
type MLPSynapse = (Double, Double)
```

Connections

A connection between two consecutive layers implements the matrix of synapses, the $(w_{ij}, \Delta w_{ij})$ pairs. The `MLPConnection` instance is created with the following parameters:

- Configuration parameters, `config`
- The source layer, sometimes known as the ingress layer, `src`
- The destination (or egress) layer, `dst`

The `MLPConnection` class is defined as follows:

```
class MLPConnection(config: MLPConfig, src: MLPLayer, dst: MLPLayer)
```

The last step in the initialization of the MLP algorithm is the selection of the initial (usually random) values of the weights (synapse). The following code snippet initializes the weights for non-bias neurons as random values in the range $[0, beta]$ with $beta \leq 1.0$.

The weight for the bias is obviously defined as $w_0=+1$, and its weight adjustment is initialized as $\Delta w_0 = 0$, as shown here:

```
Val beta = 0.1
val synapses = Array.tabulate(dst.len) (n =>
  if(n > 0) Array.fill(src.len)((beta*Random.nextDouble, 0.0))
  else Array.fill(src.len)((1.0, 0.0))
)
```

Random initialization of weights

The range $[0, beta]$ of initial random values 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 impact the number of epochs required to converge toward an optimal set of weights. [9:6]



Once the topology, synapses, layers, and connections of the MLP algorithm are defined, the initialization of the `MLPModel` model is straightforward:

```
val layers = topology.zipWithIndex
            .map(t => MLPLayer(t._2, t._1+1))
val connections = Range(0, layers.size-1).map(n =>
    new MLPConnection(config, layers(n), layers(n+1))).toArray
```

The layers are created by traversing the network topology and instantiating each layer with its proper index and number of elements. The connections are instantiated by selecting two consecutive layers of index n (with respect to $n+1$) as source (with respect to destination).

Encapsulation and the model factory

The model components: connections, layers, and synapses are implemented as top-level classes for clarity sake. 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.

Moreover, the model is responsible for creating its topology. A factory design pattern would be perfectly appropriate to instantiate an `MLPModel` 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.

Training cycle/epoch

The training of the model processes the training observations multiple times. A training cycle or iteration is known as an **epoch**. The five steps of the training cycle are as follows:

1. Forward propagation of the input value for a specific epoch.
2. Compute the sum of squared errors.
3. Backpropagation of the output error.
4. Recomputation of the synapse weight and gradient of weight.
5. Evaluate the convergence criteria and exit if criteria is met

The computation of the network weights during training could 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 unknown. The solution is to propagate the error on the output values backward through the hidden layers. This approach is not that different than the beta (or backward) pass in the hidden Markov model, covered in the *Beta class (the backward variable)* section in *Chapter 7, Sequential Data Models*.

The error at the output layer for p neurons can be computed in either of the following ways:

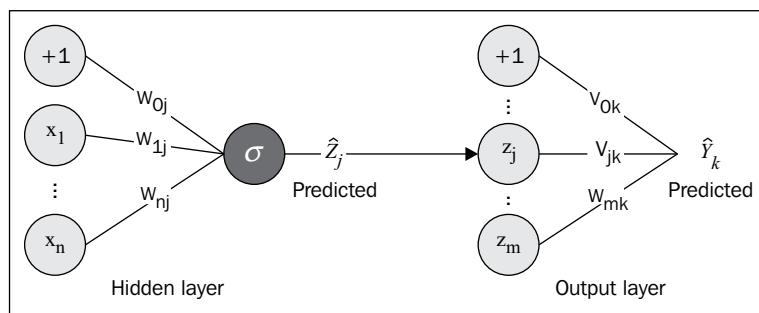
- **Sum of the squared of errors (SSE):** Calculated for each output, y_k
- **Mean squared error (MSE):** Calculated as $MSE = SSE/p$

We select the sum of the squared errors to initialize the error back-propagation algorithm.

Step 1 – input forward propagation

As mentioned earlier, the output values of a hidden layer are computed as a logistic function (the activation function) 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 (sigmoid or hyperbolic tangent). The output values z_j are then combined with the weights v_{jk} of the output layer. The output layer doesn't have an activation function.



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 .

Computation of the output y for the output layer:

$$\hat{y}_k = v_0 + \sum_{j=1}^m v_{kj} z_j$$



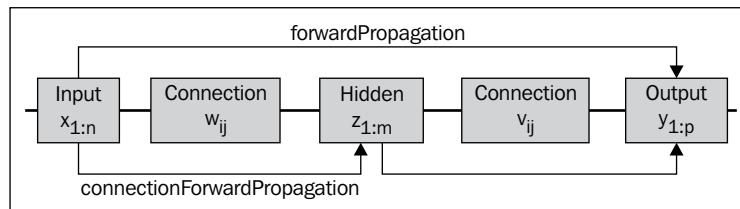
Estimation of the output values for binary classification with an activation function σ :

$$\hat{z}_j = \sigma\left(w_0 + \sum_{i=1}^n w_{ij} x_i\right) = \frac{1}{1 + e^{-w_0 - \sum_{i=1}^n w_{ij} x_i}}$$

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 (softmax).

The computational model

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. Such a representation will be quite convenient for the design and implementation of the MLP.



A computation model of input forward propagation

This diagram 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 look at the `connectionForwardPropagation` method of the `MLPConnection` class:

```
def connectionForwardPropagation: Unit = {
    val synps= synapses.drop(1)
    val _output = synps.map(x => { //1
        val sum = x.zip(src.output)
            .foldLeft(0.0)((s, xy) => s + xy._1._1*xy._2)
        if(!isOutLayer) config.activation(sum) //2
        else sum
    })
    val out = if(isOutLayer) mlpObjective(_output) else _output //3
    out.copyToArray(dst.output, 1)
}
```

The first step is to compute the linear dot product of the `_output` output of the current source layer, `src`, for this connection, and the weights, `w` (line 1). The `activation` method, the implementation of which is described in the next paragraph, is applied to the dot product, `dot` (line 2). If the destination layer of the connection is the output layer, then the output values are processed according to the `mlpObjective` objective of the algorithm (line 3).

Objective

In the *The network architecture* section, you learned that the structure of the output layer depends on the type of problems that need to be resolved, or objective of the algorithm. Let's encapsulate the different objectives (binary, multiclass classifiers, and regression) into an `MLPOjective` hierarchy (nested in `MLP` companion object) and the transformation of the output values, `y`, using a simple `apply` method:

```
trait MLPOjective { def apply(y: DblVector): DblVector }
```

The output of the `apply` method is used to compute the sum of squared errors during training, after the forward propagation of features. The binary (2 class) classifier requires a single output without any transformation because the values are either 0 or 1.

```
class MLPBinClassifier extends MLPOjective {
    override def apply(y: DblVector): DblVector = output
}
```

The `MLPMultiClassifier` multiclass classifier objective class used the `softmax` method to boost the output with the highest value, as shown here:

```
class MLPMultiClassifier extends MLPOjective {  
    override def apply(y:DblVector) :DblVector = softmax(y.drop(1))  
    def softmax(y: DblVector) : DblVector = { ...}  
}
```

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 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 the **exponential normalization** or softmax [9:8].

Softmax formula for multinomial ($K > 2$) classification is as follows:

$$\hat{y}_k \stackrel{\triangle}{=} \frac{e^{-\hat{y}_k}}{\sum_i e^{-\hat{y}_i}}$$

Here is the simple implementation of the `softmax` method of the `MLPMultiClassifier` class:

```
def softmax(y: DblVector) : DblVector = {  
    val softmaxValues = new DblVector(y.size)  
    val expY = y.map( Math.exp(_))/1  
    val expYSum = expY.sum  
    expY.map( _ /expYSum).copyToArray(softmaxValues, 1) //2  
    softmaxValues  
}
```

First, the output values are transformed to exponential, `expY` (line 1). The exponentially transformed outputs are then normalized by their sum, `expYSum`, to generate the array of `softmaxValues` output (line 2). Once again, you do not have to update the bias element $y(0)$.

The second step in the training phase is the back propagation of the output error.

Step 2 – sum of squared errors

Once the input features are propagated across the neural network, the sum of squared errors, `sse`, for the output layer of the `MPLayer` type is computed at each epoch, as follows:

```
def sse(labels: DblVector): Double = {
    var _sse = 0.0
    output.drop(1) //1
        .zipWithIndex
        .foreach(on => {
            val err = labels(on._2) - on._1 //2
            delta.update(on._2+1, on._1* (1.0- on._1)*err) //3
            _sse += err*err
        })
    _sse*0.5 //4
}
```

As expected, the computation of the sum of squared errors requires the labeled values, `labels`, and the `objective` method as arguments. The vector output values, `output`, stripped of the bias node (line 1) is used to compute the difference, `err`, between the `label` and the actual output (line 2). The `delta` value (line 3), described in the next section, is used in the back-propagation algorithm to adjust the weights of the output and hidden layers. Note that the sum of squares, `_sse`, is divided by 2 (line 4), so its derivative is `err`.

Step 3 – error backpropagation

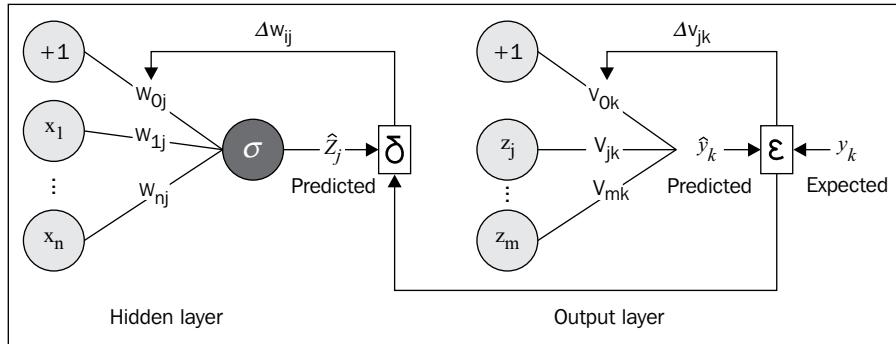
The error backpropagation is an algorithm that estimates the error for the hidden layer in order to compute the change in weights of the network. It takes the sum of squared errors on the output as input.

Terminology

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. In this chapter, we keep the narrower definition of backpropagation as the backward computation of the sum of squared errors.

Error propagation

The objective of the training of a perceptron is to minimize the sum of squared errors 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. This approach does not work for the hidden layers z_j because the label value is unknown.



The partial derivative of the sum of squared output 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 .

Derivative of the output SSE over the weights of the output layer:

$$\varepsilon_k = y_k - \hat{y}_k = y_k - v_0 - v^T z$$

$$\varepsilon = \frac{1}{2} \sum_{k=1}^p \varepsilon_k^2$$

$$\frac{\partial \varepsilon}{\partial v_k} = - \sum_{j=1}^m \varepsilon z_{jk}$$

As mentioned earlier, the computation of the partial derivative of the sum of squared error over the weights of the hidden layer is a bit tricky. Fortunately, the partial derivative can be broken down into the following three pieces using the output layer values and the output of the hidden layer:

- Derivative of sum of squared error ε over the output value y_k
- Derivative of the output value y_k over the hidden value z_j knowing that the derivative of a sigmoid σ is $\sigma(1 - \sigma)$
- Derivative of the output of the hidden layer z_j over the weights w_{ij}

Derivative of error over the weights of the hidden layer:

[

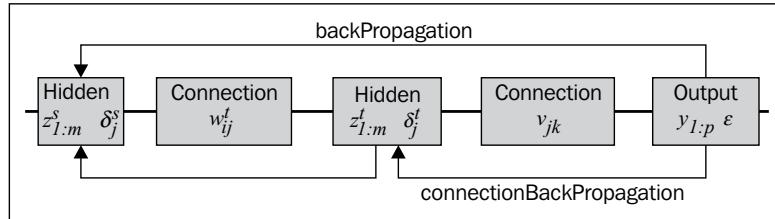
 *Notebook*

$$\frac{\partial \epsilon}{\partial w_{ij}} = \sum_{k=1}^p \frac{\partial \epsilon}{\partial y_k} \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}} = - \sum_{k=1}^p \delta_j x_j$$

$$\delta_j = \epsilon_k z_j (1 - z_j)$$
]

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 the derivative delta δ 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 :



The `connectionBackPropagation` method propagates the error back to the previous 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 error derivative in the destination or next layer, and `src.delta` is the error derivative on the outputs in the source (or antecedent) layer, as shown here:

```

def connectionBackpropagation: Unit =
    Range(1, src.len).foreach(i => {
        val dot = Range(1, dst.len).foldLeft(0.0)((s, j) =>
            s + synapses(j)(i)._1*dst.delta(j)) //1
        src.delta(i) = src.output(i)*(1.0 - src.output(i))*dot//2
    })

```

The dot product of the synapse weights and the errors of the destination layers (line 1) is used to compute the delta on the source (or previous layer) layers (line 2).

Step 4 – synapse/weights adjustment

The connection weights Δv and Δw are adjusted by computing the sum of the derivative 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].

Momentum factor for gradient descent

The simplest algorithm to update the weights is the gradient descent [9:10].

The gradient descent is a very simple and robust algorithm. However, it is 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 *Appendix A, Basic Concepts*).

There are several methods available to speed up the convergence of the gradient descent toward a minimum: momentum factor and adaptive learning coefficient [9:11].

Large variations of the weights (or large value of the gradient of weights) cause the gradient descent to require more training iteration in order 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.

The computation of neural network weights using gradient descent is as follows:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \eta \frac{\partial \varepsilon_j^{(t)}}{\partial w_{ij}}$$



The computation of neural network weights using gradient descent method with 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 basic gradient descent algorithm is selected by setting the momentum factor α to zero.

Implementation

The fourth step of the training phase is to adjust each connection's synapses ($w, \Delta w$). This task is performed by the `connectionUpdate` method of the `MLPConnection` class:

```
def connectionUpdate: Unit =
  Range(1, dst.len).foreach(i => {
    val delta = dst.delta(i) //1

    Range(0, src.len).foreach(j => {
      val _output = src.output(j) //2
      val oldSynapse = synapses(i)(j)
      val grad = config.eta*delta*_output //3
      val deltaWeight = grad + config.alpha* oldSynapse._2 //4
      synapses(i)(j) = (oldSynapse._1 + deltaWeight, grad) //5
    })
  })
```

The `connectionUpdate` method computes the error of each destination neuron (line 1). The `_output` output of each neuron source (line 2) is used in the computation of the `grad` gradient (line 3). The weight is then adjusted for a momentum (line 4) as per the mathematical formulation. Finally, the synapses for source and destination layers are updated (line 5).

 [**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 5 – convergence criteria

The convergence criterion consists of evaluating the sum of squared errors against a predetermined threshold `eps`. It is common to normalize the sum of squared errors by the number of observations.

Configuration

The `MLPConfig` configuration of the multilayer perceptron consists of the definition of the network configuration with hidden layers, the learning parameters, the training parameters, and the activation function:

```
Class MLPConfig(val alpha: Double, val eta: Double, val hidLayers:
  Array[Int], val numEpochs: Int, val eps: Double, val activation:
  Double=>Double) extends Config
```

For the sake of readability, the name of the configuration parameters matches the symbols defined in the mathematical formulation:

- `alpha`: This is the momentum factor.
- `eta`: This is the learning rate (fixed or adaptive).
- `hidLayers`: This is an array of size of hidden layers (for example, two hidden layers of two and four elements are specified as `Array[Int](2, 4)`).
- `numEpochs`: This is the maximum number of epochs allowed for training the neural network.
- `eps`: This is the convergence criteria used as an exit condition for the training of the neural network, `error < eps`.
- `activation`: This is the activation function used for nonlinear regression applied to hidden layers. The default function is the sigmoid.

Putting all together

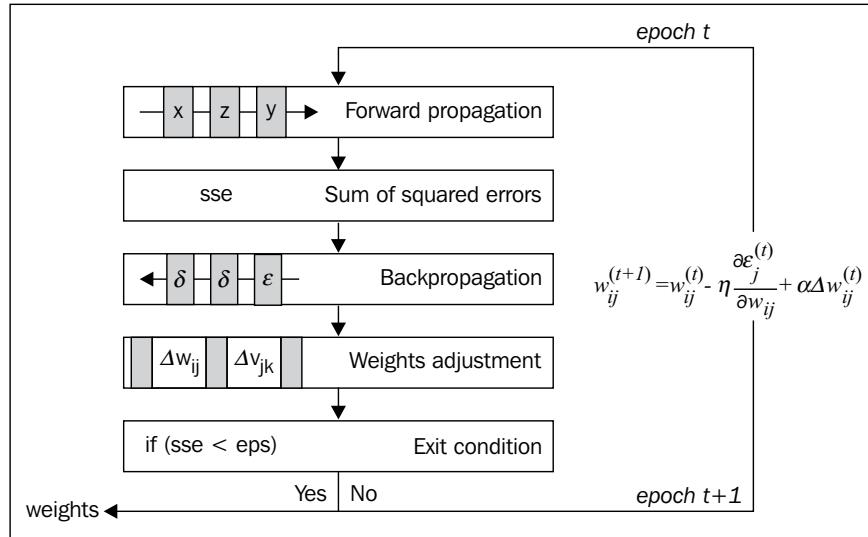
The five steps of the training cycle have been implemented for each connection or matrix of synapses (weights, gradient of weights). The management of the cycle is performed by the algorithm defined by the `MLP` class, as shown here:

```
class MLP[T <% Double] (config: MLPConfig, xt: XTSeries[Array[T]],  
labels: DblMatrix) (implicit val mlpObjective: MLP.MLPOjective)  
extends PipeOperator[Array[T], DblVector] {  
    val model: Option[MLPModel]  
    def |> : PartialFunction[Array[T], DblVector]  
}
```

The `MLP` algorithm takes the following parameters:

- `config`: The configuration of the algorithm
- `xt`: The time series of features used to train the model
- `labels`: The labeled output values for training purpose
- `mlpObjective`: The implicit objective of the algorithm (a type of problem)

The five steps of the training cycle or epoch is summarized in the following diagram:



Let's apply the five steps of a training epoch in a `trainEpoch` method of the `MLPModel` class using a simple the `foreach` Scala iterator, as shown here:

```

def trainEpoch(x: DblVector, y: DblVector): Double = {
    inLayer.set(x)

    connections.foreach(_.connectionForwardPropagation) //1
    val _sse = sse(y) //2
    val bckIterator = connections.reverseIterator
    bckIterator.foreach(_.connectionBackpropagation) //3
    connections.foreach(_.connectionUpdate) //4
    _sse
}

```

You can certainly recognize the first four stages of the training cycle: forward propagation of the input, x (line 1), computation of the sum of squared errors, $_sse$ (line 2), the back propagation of the error (line 3), and the recomputation of the weight and gradient of weight associated with each synapse (line 4).

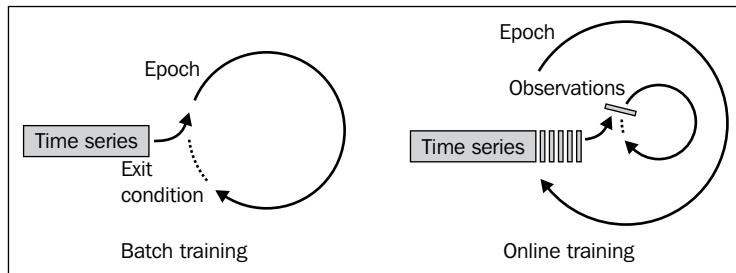
Training strategies and classification

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.

Online versus batch training

One important remaining issue is finding a strategy to conduct the training of time series, as ordered sequences of data. There are two strategies to create an MLP model for 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 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 error (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 improve the performance of the online training process further.

The online training strategy is applied to a financial time series for the remainder of this chapter.

Regularization

There are two approaches to find the most appropriate network architecture for a given classification or regression problem; they are:

- **Destructive tuning:** Starting with a large network, then removing nodes, synapses, and hidden layers that have no impact on the sum of squared errors
- **Constructive tuning:** Starting with a small network, 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 by 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 *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].

Model instantiation

The `model` instance is created (trained) during the instantiation of the multilayer perceptron. The model is created by iterating the training cycle over all the data points of the time series `xt`, and through multiple epochs until the total sum of squared errors is smaller than the threshold `eps`, as in the following code:

```
var converged = false
val model: Option[MLPModel] = {
    val _model = new MLPModel(config, xt(0).size, labels(0).size)
    (mlpObjective) //1
    val errScale = 1.0/(labels(0).size*xt.size) //4

    converged = Range(0, config.numEpochs).find(_ => {
        xt.toArray.zip(labels)
            .foldLeft(0.0)((s, xtbl) =>
                s + _model.trainEpoch(xtbl._1, xtbl._2) //2
            )*errScale < config.eps //3
    }) != None
    _model
}
```

The model is first initialized (line 1). The first four stages of the MLP training cycle are executed by the `MLPModel.trainEpoch` method described in the previous section (line 2). The method returns the sum of squared errors for each observation in the time series. The sum of squared errors for the observations are summed, then evaluated against the convergence criterion, `eps` (line 3). The sum of squared errors is normalized for the size of the time series and the size of the output vector (line 5). The implementation uses the Scala method, `find`, to exit from the iterative loop before the maximum number of epochs, `config.numEpochs`, is reached.

The exit condition



In this implementation a flag, `converged`, is set to indicate that the execution of the training has not converged before the maximum number of epochs has been reached; however, the model is still instantiated nevertheless. It allows the client code to evaluate the pattern of the sum of squared errors in regard to a local minimum.

Once the model is created during the instantiation of the multilayer perceptron, it is available to predict the class of a new observation.

Prediction

The prediction method of the `MLPModel` class, `getOutput`, takes a new observation (feature vector) as argument and returns the output by using the forward propagation algorithm:

```
def getOutput(x: DblVector): DblVector = {  
    inLayer.set(x)  
    connections.foreach(_.connectionForwardPropagation)  
    outLayer.output  
}
```

The classification method is implemented as the data transformation `|>`. It returns the predicted value, normalized as a probability if the model was successfully trained; `None`, otherwise:

```
def |> : PartialFunction[Array[T], DblVector] = {  
    case x: Array[T] if(model!=None && x.size == dimension(xt)) =>  
        model.get.getOutput(x)  
}
```

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.

Impact of 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 sum of the squared errors of all output variables. The observations x (with respect to the labeled output, y) are synthetically generated using several noisy patterns: functions f_1 , f_2 , and `noise`, as follows:

```
val noise = () => NOISE_RATIO*Random.nextDouble
val f1 = (x: Double) => x*(1.0 + noise())
val f2 = (x: Double) => x*x*(1.0 + noise())

def vec1(x: Double): DblVector = Array[Double](f1(x), noise(), f2(x),
noise())
def vec2(x: Double): DblVector = Array[Double](noise(), noise())
val x = XTSeries[DblVector](Array.tabulate(TEST_SIZE)(vec1(_)))
val y = XTSeries[DblVector](Array.tabulate(TEST_SIZE)(vec2(_)))
```

The x and y values are normalized [0, 1]. The test is run with a sample of size `TEST_SIZE` data points over a maximum of 250 epochs, a single hidden layer of five neurons with no softmax transformation and the following MLP parameters:

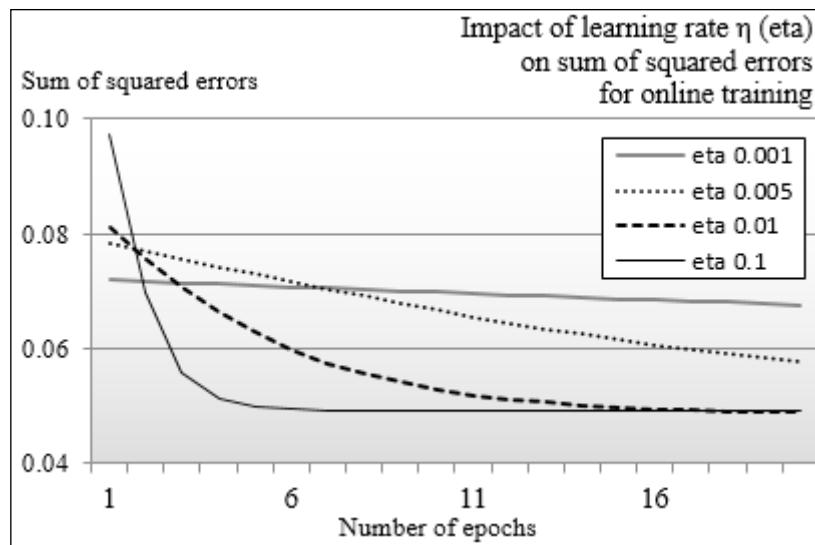
```
val NUM_EPOCHS = 250; val EPS = 1.0e-4
val HIDDENLAYER = Array[Int](5)
val ALPHA = 0.9; val TEST_SIZE = 40

val features = XTSeries.normalize(x).get
val labels = XTSeries.normalize(y).get.toArray
val config = MLPConfig(ALPHA, _eta, SIZE_HIDDEN_LAYER, NUM_EPOCHS,
EPS)

implicit val mlpObjective = new MLP.MLPBinClassifier
val mlp = MLP[Double](config, features, labels)
```

The objective of the algorithm, `mlpObjective`, has to be implicitly defined prior to the instantiation of the `MLP` class.

The test is performed with a different learning rate, `eta`. For clarity's sake, the graph displays the sum of squared errors for the first 22 epochs.

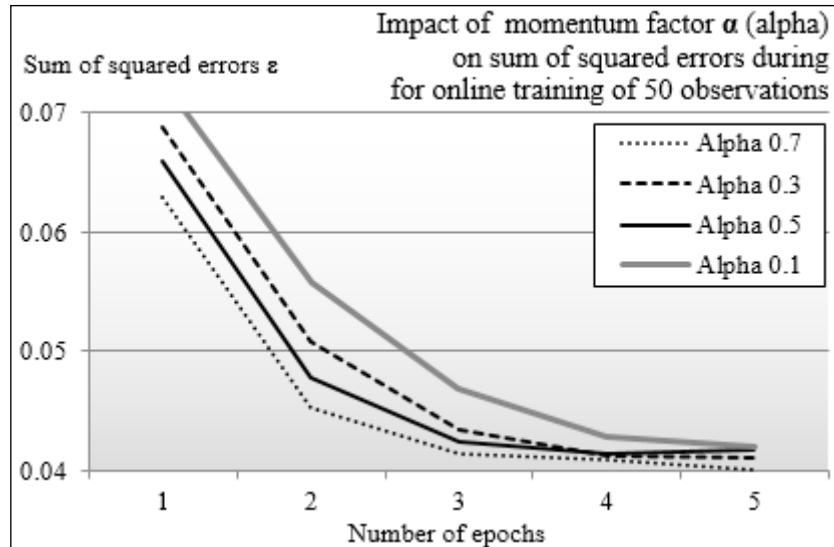


Impact of the learning rate on the MLP training

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 sum of squared errors 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.

Impact of the momentum factor

Let's quantify the impact of the momentum factor, α , on the convergence of the training process toward an optimal model (synapse weights). The total sum of squared errors for the entire time series is plotted for the first five epochs in the following graph:



Impact of the momentum factor on the MLP training

The graph shows that the rate the sum of squared errors declines as the momentum factor increases. In other words, the momentum factor has a positive although limited impact on the convergence of the gradient descent.

Let's apply our newfound knowledge regarding neural networks and the classification of variables, that impact the exchange rate of 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 exchange rate of currencies:

- FXA: Rate of an Australian dollar in US dollar
- FXB: Rate of a British pound in US dollar
- FXE: Rate of an Euro in US dollar

- **FXC:** Rate of a Canadian dollar in US dollar
- **FXF:** Rate of a Swiss franc in US dollar
- **FXY:** Rate of a Japanese yen in US dollar
- **CYB:** Rate of a Chinese yuan in US dollar
- **SPY:** S&P 500 index
- **GLD:** The price of gold in US dollar

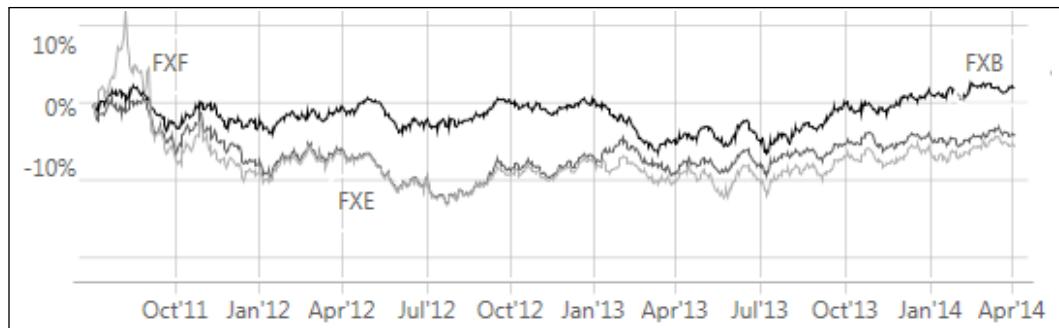
Practically, the problem to solve is to extract one or more regressive models that link one ETFs y with a basket of other ETFs $\{x_i\}$ $y=f(x_i)$. For example, is there a relation between the exchange rate of the Japanese yen (FXY) and a combination of the spot price for gold (GLD), exchange rate of the Euro in US dollar (FXE) and 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

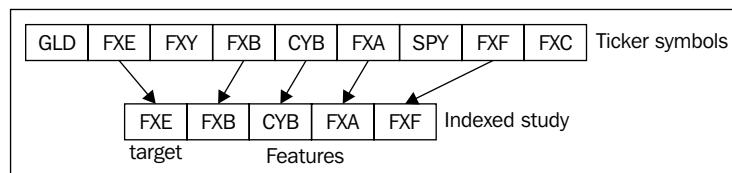
The second chart displays another group of currency-related ETFs that shares a similar price action behavior. Neural networks do not provide any analytical representation of their internal reasoning; therefore, a *visual* correlation can be extremely useful to novice engineers in validating their models.



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:



The mechanism to generate features from ticker symbols

Implementation

The first step is to define the configuration parameter for the MLP classifier, is as follows:

```

val path = "resources/data/chap9/"
val ALPHA = 0.5; val ETA = 0.03
val NUM_EPOCHS = 250; val EPS = 1.0e-6
var hidLayers = Array[Int](7, 7) //1

var config = MLPConfig(ALPHA, ETA, hidLayers, NUM_EPOCHS, EPS)
  
```

Besides the learning parameters, the network is initialized with two configurations:

- One hidden layer with four nodes
- Two hidden layers of four neurons each (line 1)

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") //2
```

The closing prices of all the ETFs over a period of three years are extracted from the Google Financial tables, using the `GoogleFinancials` extractor (line 3) for a basket of ETFs (line 2):

```
val prices = symbols.map(s =>DataSource(s"$path$s.csv",true))
    .map( _ |> GoogleFinancials.close) //3
    .map( _.toArray)
```

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 (line 4):

```
val study = Array[String] ("FXE", "FXF", "FXB", "CYB") //4
```

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, CYB} and one output variable FXE:

```
val obs = study.map(s =>index.get(s).get).map( prices( _ )) //5
val features = obs.drop(1).transpose //6
val target = Array[DblVector] (obs(0)).transpose //7
```

The set of observations is built using an index (line 5). 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 series is computed through transpose (line 6). The single output variable, target, has to be converted into a matrix before transposition (line 7).

Ultimately, the model is built through instantiation of the `MLP` class:

```
val THRESHOLD = 0.08
implicit val mlpObjective = new MLP. MLPBinClassifier
val mlp = MLP[Double](config, features, target)
mlp.accuracy(THRESHOLD)
```

The objective type, `m1pObjective`, is implicitly defined as an MLP binary classifier, `MLPBinClassifier`. The square root of the sum of squares of the difference between the predicted output generated by the MLP and the target value is computed and compared to a predefined threshold. The accuracy value is computed as the percentage of data points, whose prediction matches the target value within a range `< THRESHOLD`:

```
val nCorrects = xt.toArray.zip(labels).foldLeft(0)((s, xt1) => {
    val output = model.get.getOutput(xt1._1) //8
    val _sse = xt1._2.zip(output.drop(1))
        .foldLeft(0.0)((err, tp) => {
            val diff= tp._1 - tp._2
            err + diff*diff
        }) //9
    val error = Math.sqrt(_sse)/(output.size-1) //10
    if(error < threshold) s + 1
    else s
})
nCorrects.toDouble/xt.size
```

The implementation of the computation of the accuracy, as in the previous code snippet, retrieves the values of the output layer (line 8). The `error` value (line 10) is computed as the square root of the sum of squared errors, `_sse` (line 9). Finally, a prediction is considered correct if it is equal to the labeled output, within the margin error, `threshold`.

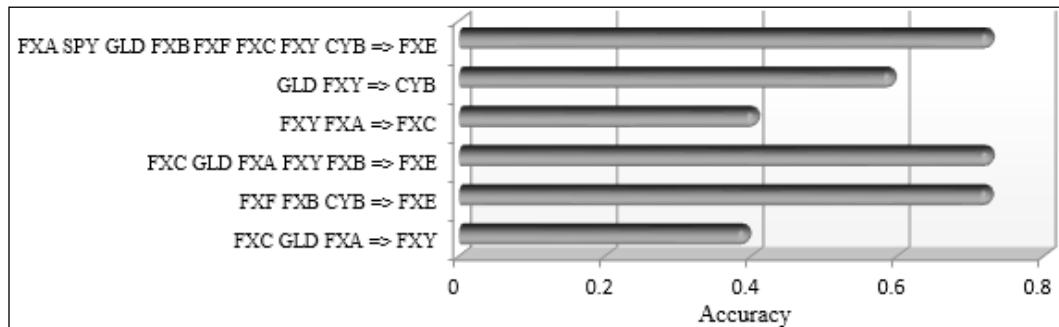
Models evaluation

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:

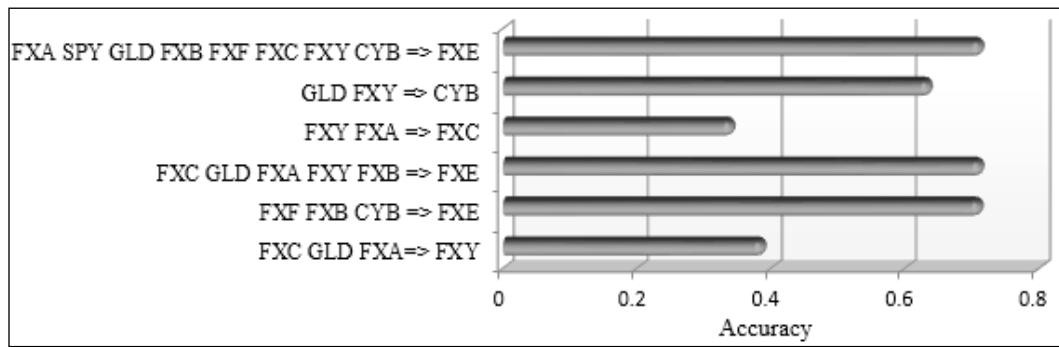
- Two hidden layers with four nodes each
- Three hidden layers with eight (with respect to five and six) nodes

This first chart visualizes the accuracy 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:



Accuracy of MLP with two hidden layers of four nodes each

The next chart displays the accuracy of the six regression models for an architecture with three hidden layers of eight, five, and six nodes, respectively:



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 most accurate regression models are as follows:

- $\text{FXE} = f(\text{FXA}, \text{SPY}, \text{GLD}, \text{FXB}, \text{FXF}, \text{FXD}, \text{FXY}, \text{CYB})$
- $\text{FXE} = g(\text{FXC}, \text{GLD}, \text{FXA}, \text{FXY}, \text{FXB})$
- $\text{FXE} = h(\text{FXF}, \text{FXB}, \text{CYB})$

On the other hand, the prediction 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 configuration.

The empirical evaluation



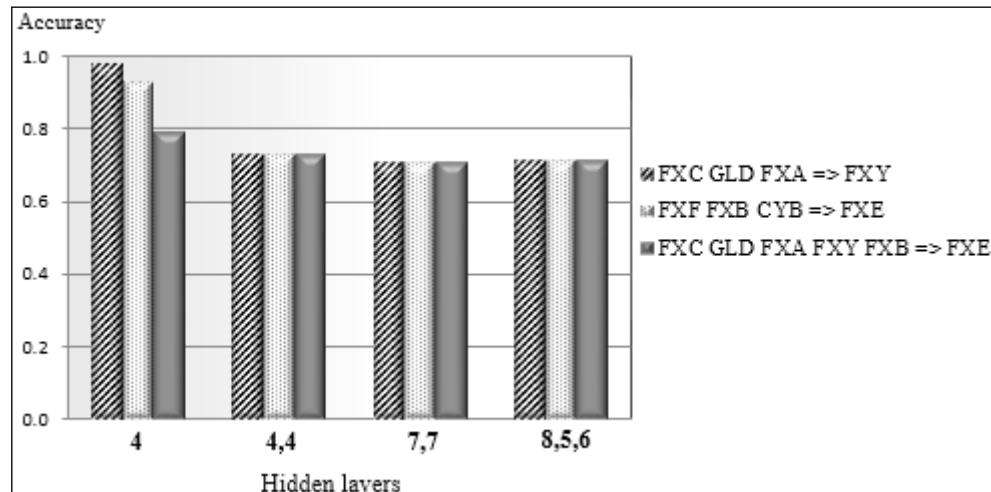
These empirical tests use a simple accuracy metric. A formal comparison of the regression models would systematically analyze every combination of input and output variables. The evaluation would also compute the precision, the recall, and the F1 score for each of those models (refer to the *Key metrics* section under *Validation* in the *Assessing a model* section in *Chapter 2, Hello World!*).

Impact of 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 => FXE*), (*FCX, GLD, FXA => FXY*), and (*FXC, GLD, FXA, FXY, FXB => 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, not to estimate the absolute accuracy of each model.

The four network configurations are as follows:

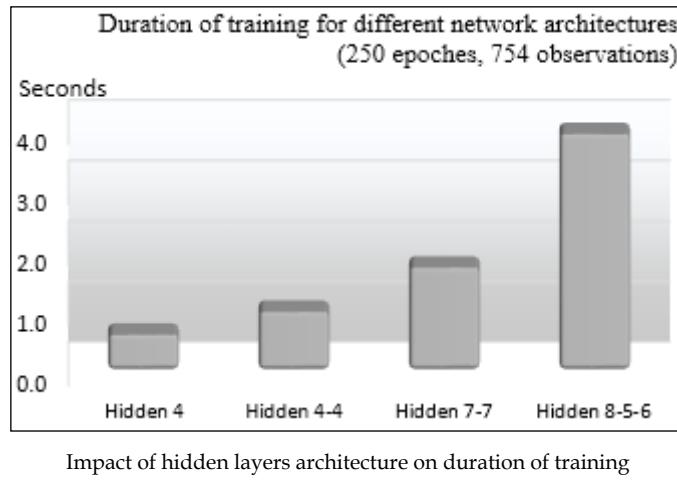
- A single hidden layer with four nodes
- Two hidden layers with four nodes each
- Two hidden layers with seven nodes each
- Three hidden layer with eight, five, and six nodes



Impact of 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 look at the impact of the complexity of the network on the duration of the training, as in the following graph:



Not surprisingly, the time complexity increases significantly with the number of hidden layers and number of nodes.

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 error backpropagation, have some obvious advantages, such as:

- 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.

- 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.
- MLP does not make any assumption on linearity, variable independence, or normality.
- The execution of training of the 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.

However, as with any machine learning algorithm, neural networks have their detractors. Among 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.
- MLP requires a lengthy training process, especially using the batch 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 momentum factor tends to converge faster and require a smaller number of epochs than the batch process.
- Tuning the configuration parameters, such as learning rate, selection of the activation method, application of softmax transformation, or momentum factor, can turn into a lengthy process.
- Estimating the minimum size of the training set to get accurate results is not obvious.
- A neural network cannot be incrementally retrained. Any new labeled data requires an entirely new training cycle.

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 a neural networks
- The stages of the training cycle of a backpropagation multilayer perceptron
- How to implement an MLP from the ground up in Scala
- The numerous configuration parameters and options to use MLP as a classifier and regression
- To evaluate the impact of the learning rate and the gradient descent momentum factor on the convergence of the sum of squared errors during training
- How to apply a multilayer perceptron to the financial analysis of the fluctuation of currencies

The next chapter will introduce the concept of genetic algorithms with a full 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:

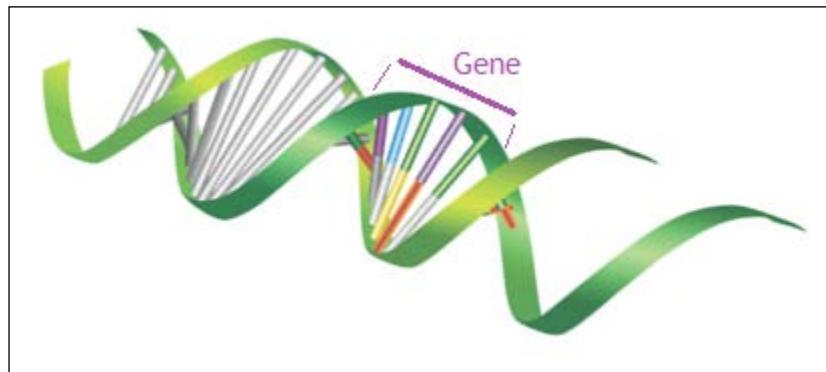
- Apply genetic algorithms to leverage technical analysis of market price and volume movement to predict future returns
- Evaluate or estimate the search space
- Encode solutions in the binary format using either hierarchical or flat addressing
- Tune some of the genetic operators
- Create and evaluate 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:



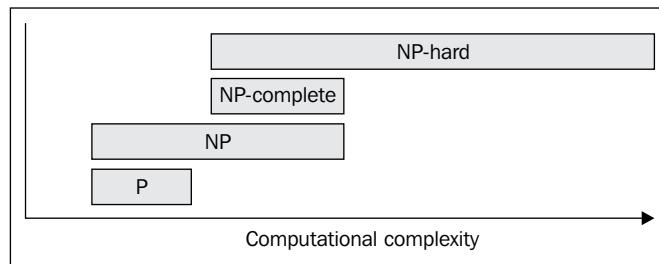
The **population** of organisms varies over time. The number of individuals within a population changes, sometimes dramatically. These variations are usually associated with the abundance or lack of predators and prey as well as the changing environment. Only the fittest organisms within the population 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 solution can be validated.

- **NP-hard problems:** These problems have solutions that may not be found in polynomial time.



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 solution for a problem that is similar to unsupervised learning, for which a class or clusters are identified through an iterative or optimization methodology.

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, 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 data set 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 the *Introduction* section of *Chapter 11, Reinforcement Learning*).
- **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 function:** This is the evaluation of the fittest candidate using an objective function

Encodings and the fitness function are problem dependent. Genetic operators are not.

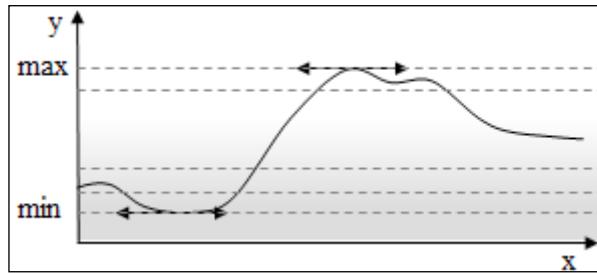
Encodings

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 **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 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:

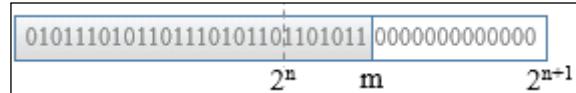


The step size of the discretization is computed as:

$$step = \frac{\max - \min}{2^n}$$

The step size of the discretization of the sine $y = \sin(x)$ 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, $n+1$, defined 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,...) 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 $x \operatorname{operator} [target]$, for instance, $\text{unit cost} < [9\$]$, $\text{temperature} = [82F]$, or $\text{Movie rating is } [3 \text{ stars}]$.

The simplest encoding scheme for predicates is as follows:

- Variables are encoded as category or type (for example, temperature, barometric pressure, and so on) because there is a finite number of variables in any model
- Operators are encoded as discrete type
- 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 $\{\operatorname{operator}, \operatorname{left-operand}, \operatorname{right-operand}\}$ 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

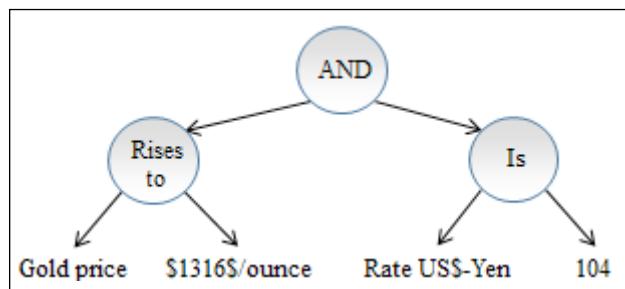
The solution encoding approach describes the solution to a problem as an unordered sequence of predicates. Let's consider the following rule:

```
IF {Gold price rises to [1316$/ounce]} AND
    {US$/Yen rate is [104]}).
THEN {S&P 500 index is [UP]}
```

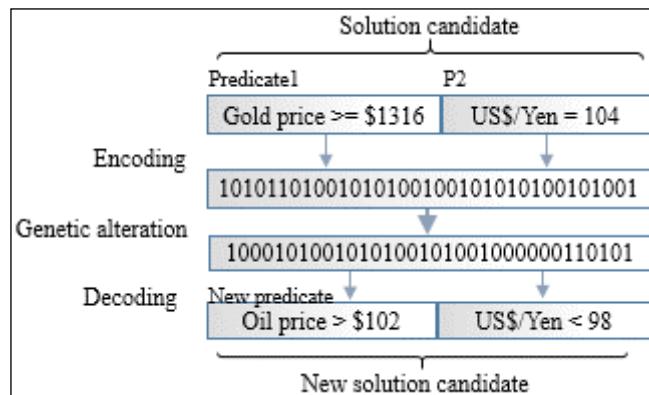
In this example, the search space is defined by two levels:

- Boolean operators (for example, AND) and predicates
- Each predicate is defined as a tuple $\{variable, operator, target value\}$

The tree representation for the search space is shown in the following diagram:



The bits string representation is decoded back to its original format for further computation:



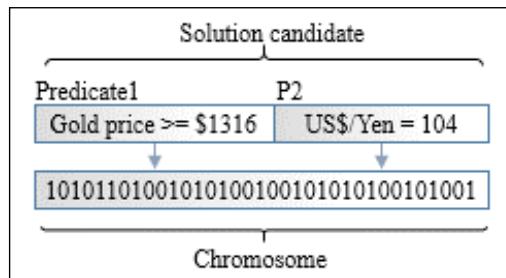
The encoding scheme

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

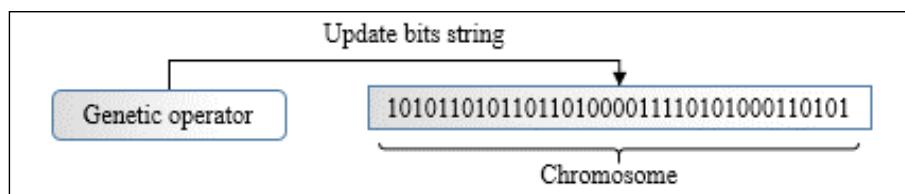
Flat encoding

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:



An overview of flat addressing

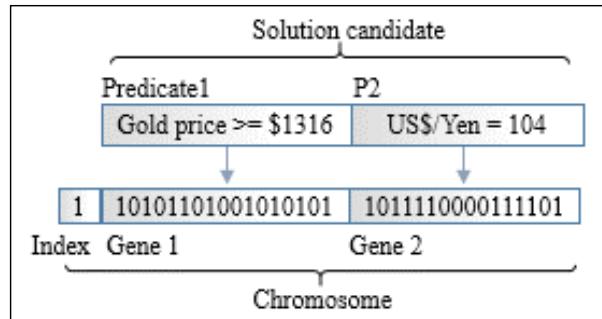
A genetic operator manipulates the bits of the chromosome regardless of whether the bits refer to a particular predicate:



Chromosome encoding with flat addressing

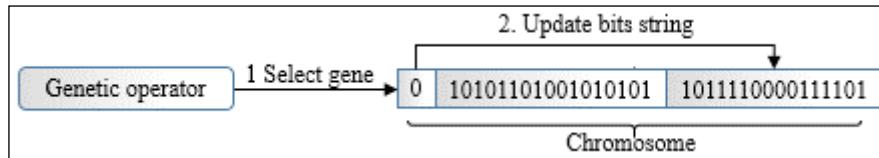
Hierarchical encoding

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:



An overview of hierarchical addressing

A generic operator selects the predicate it needs to manipulate first. Once the target gene is selected, the operator updates the bits string associated to the gene, as follows:



A chromosome with hierarchical 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

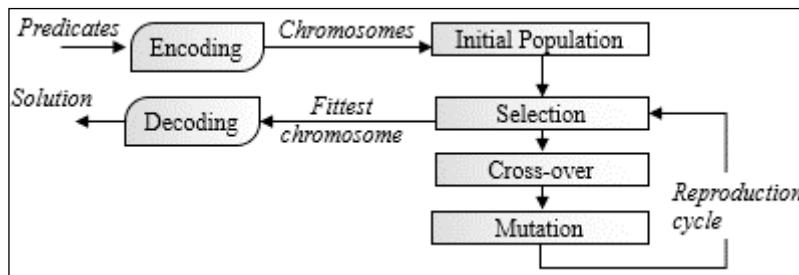
The implementation of the reproduction cycle attempts to replicate the natural reproduction process [10:7]. The reproduction cycle that controls the population of chromosomes consists of three genetic operators:

- **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 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 towards a local maximum or minimum.

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:



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 bias 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 using either a fitness or unfitness function to score chromosomes.

The selection process consists of the following steps:

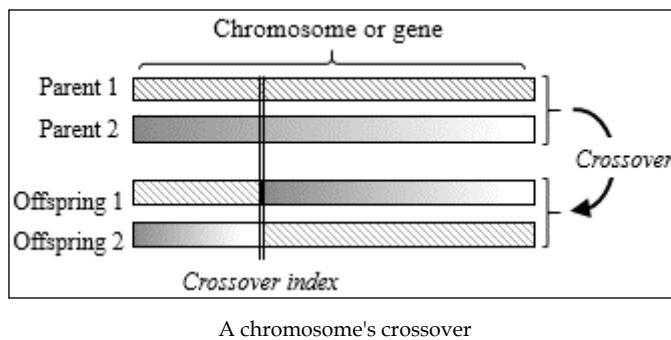
1. Apply the fitness/unfitness function to each chromosome j in the population, f_j
2. Compute the total fitness/unfitness score for the entire population, $\sum f_j$
3. Normalize the fitness/unfitness score of each chromosome by the sum of the fitness/unfitness scores of all the chromosomes, $f_j = f_j / \sum f_j$
4. Sort the chromosomes by their descending fitness score or ascending unfitness score
5. Compute the cumulative fitness/unfitness score for each chromosome, $j f_j = f_j + \sum_{k < j} f_k$
6. Generate the selection probability (for the rank-based formula) as a random value, $p \in [0,1]$
7. Eliminate the chromosome, k , having a low fitness score $f_k < p$ or high unfitness cost, $f_k > p$
8. Reduce the size of the population further if it exceeds the maximum allowed number of chromosomes.

Natural selection

You should not be surprised by the need to control the size of 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:



A chromosome's crossover

An important element in the crossover phase is the selection 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 ri of the gene for which the crossover is applied as $ri = p \cdot num_genes$, 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 $xi=p.gene_length$, 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.

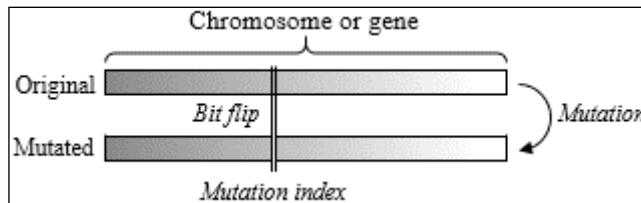
Preserving parent chromosomes



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 preventing the reproduction cycle from converging towards 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 chromosome mutation

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 mi of the gene to be mutated using the formula $mi = p.num_genes$.
4. Compute the index of the bit in the gene to be mutated $xi=p.genes_length$.
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.

Fitness score

The fitness function is the centerpiece of the selection process. There are three categories of fitness functions:

- **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, down to each gene.

Software design

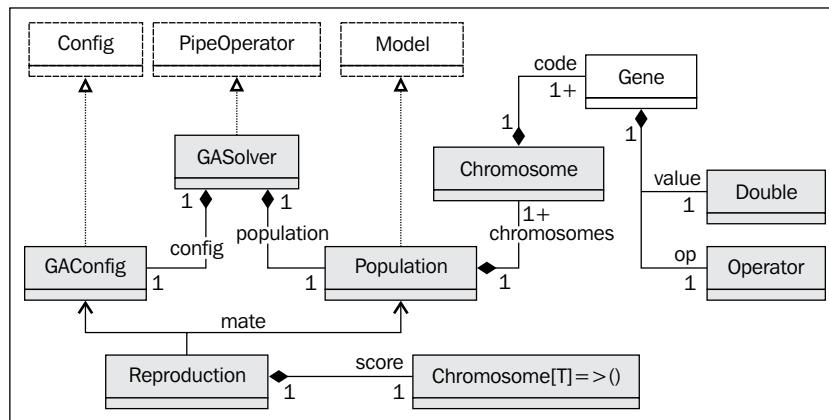
The implementation of the genetic algorithm uses a design that is similar to the template for classifiers (refer to the *Design template for classifier* section in *Appendix A, Basic Concepts*).

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 type `GAConfig` and the initial population. This class defines a data transformation by implementing the `PipeOperator` trait.
- The configuration class `GAConfig` consists of the GA execution and reproduction configuration parameters.
- The reproduction (of the type `Reproduction`) controls the reproduction cycle between consecutive generations of chromosomes through the `mate` method.

The following UML class diagram describes the relation between the different components of the genetic algorithm:



UML class diagram of genetic algorithm components

Let's start by defining the key classes that control the genetic algorithm.

Key components

The parameterized class `Population` (with the subtype `Gene`) contains the set or pool of chromosomes. A population contains chromosomes that are a sequence or list of element of the type inherited from `Gene`. A `Pool` is a mutable array in order to avoid excessive duplication of the `Chromosome` instances associated with immutable collections.

A 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.

The Population class takes two parameters:

- **limit**: This is the maximum size of the population
- **chromosomes**: This is the pool of chromosomes defining the current population

A reproduction cycle executes the following sequence of three genetic operators on a population: `select` for selection across all the chromosomes of the population, `+-` for crossover of all the chromosomes, and `^` for the mutation of each chromosome.

Consider the following code:

```
type Pool[T <: Gene] = ArrayBuffer[Chromosome[T]]  
class Population[T <: Gene](limit: Int, val chromosomes: Pool[T]) {  
    def select(score: Chromosome[T] => Unit, cutoff: Double)  
    def +- (xOver: Double)  
    def ^ (mu: Double)  
    ...  
}
```

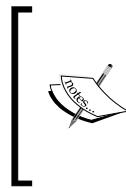
The `limit` value specifies the maximum size of the population during optimization. It defines the hard limit or constraints on the population growth.

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 unfitness: Double = 1e+5 * (1 + Random.nextDouble)  
    def +- (that: Chromosome[T], idx: GeneticIndices):  
        (Chromosome[T], Chromosome[T])  
    def ^ (idx: GeneticIndices): Chromosome[T]  
    ...  
}
```

The algorithm assigns the fitting score an `unfitness` value in this implementation to enable the ranking of the population and ultimately the selection of the fittest chromosomes.



Fitness vs. unfitness

The machine learning algorithms used the loss function or its variant as an objective function to be minimized. This implementation of the GA uses unfitness scores to be consistent with the concept of minimization of cost, loss, or penalty function.

Finally, the reproduction process executes the genetic operators on each gene:

```
class Gene(val id: String, val target: Double, op: Operator)(implicit
discr: Discretization) {
    val bits: BitSet
    ...
    def +- (index: Int, that: Gene): Gene
    def ^ (index: Int): Unit
}
```

The Gene class takes four parameters:

- `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.
- `discr`: This is the discretization class that converts a double value to an integer to be converted into bits and vice versa.

The discretization is implemented as a case class:

```
case class Discretization(toInt: Double => Int,toDouble: Int =>
Double) {
    def this(R: Int) =
        this((x: Double) => (x*R).floor.toInt, (n: Int) => n/R)
}
```

The first function, `toInt`, 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 type `java.util.BitSet`) using the discretization function `Discretization.toInt`. The bit string is decoded by the `decode` method of the `Gene` companion object.

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.

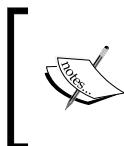
Selection

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) = {
    val cumul = chromosomes.foldLeft(0.0)((s,x) =>{
        score(xy); s + xy.unfitness} ) //1
    chromosomes foreach( _ /= cumul) //2
    val newChromosomes = chromosomes.sortWith(_.unfitness < _.unfitness)
    //3

    val cutOffSize = (cutOff*newChromosomes.size).floor.toInt //4
    val newPopSize = if(limit<cutOffSize) limit else cutOffSize //5
    chromosomes.clear //6
    chromosomes ++= newChromosomes.take(newPopSize) //7
}
```

The `select` method computes the cumulative sum of an unfitness value, `cumul`, for the entire population (line 1). It normalizes the unfitness of each chromosome (line 2), orders the population by decreasing value (line 3), and applies a soft limit function on population growth, `cutOff` (line 4). 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` (line 5). Finally, the current population is cleared (line 6) and updated with the next generation (line 7).



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 scoring function `score` takes a chromosome as parameter and updates its `unfitness` value for this chromosome.

Controlling population growth

The natural selection process controls or manages the growth of the population of species. The genetic algorithm uses two mechanisms:

- The absolute maximum size of the population (hard limit).
- The incentive to reduce the population as the optimization progresses (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 through a user-defined function, `softLimit`, of the type `Int => Double`, provided as a configuration parameter (`softLimit(cycle: Int) => a.cycle +b)`.

GA configuration

The four configurations and tuning parameters required by the genetic algorithm are:

- `xover`: This is the crossover ratio (or probability) and has a value in the interval $[0, 1]$.
- `mu`: This is the mutation ratio with a value in the interval $[0, 1]$.
- `maxCycles`: This is the maximum number of reproduction cycles.
- `softLimit`: This is the soft constraint on the population growth. The constraint function takes the number of iterations as argument and returns the maximum number of chromosomes allowed in the population.

Consider the following code:

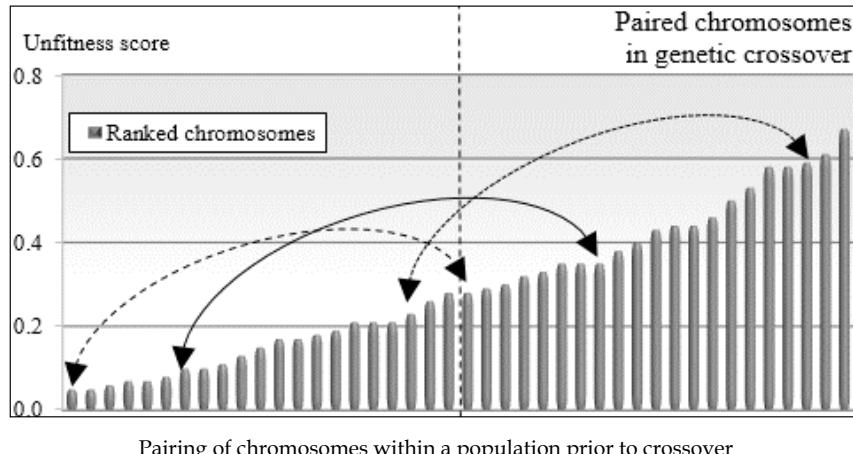
```
class GAConfig(val xover: Double, val mu: Double, val maxCycles: Int, val  
  softLimit: Int => Double) extends Config
```

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 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) //1
        val gIdx = geneticIndices(xOver) //5
        val offSprings = chromosomes.take(mid)
            .zip(bottom) //2
            .map(p => p._1 +- (p._2, gIdx))
            .unzip //3
        chromosomes +=+ offSprings._1 ++ offSprings._2 //4
    }
}
```

This method splits the population into two subpopulations of equal size (line 1) and applies the Scala `zip` method (line 2) to generate the set of pairs of offspring chromosomes (line 3). The crossover operator, `+-`, is applied to each chromosome pair to produce an array of pairs of offspring. Finally, the crossover method adds offspring chromosomes to the existing population (line 4). The crossover value, `xOver`, is a probability randomly generated over the interval `[config.xOver, 1]`.

The `geneticIndices` method (line 5) computes the relative indices of the crossover bit in the chromosomes and genes:

```
case class GeneticIndices(val chOpIdx: Int, val geneOpIdx: Int)
def geneticIndices(prob: Double): GeneticIndices = {
    var idx = (prob*chromosomeSize).floor.toInt
    val chIdx = if(idx==0) 1
                else if(idx == chromosomeSize) chromosomeSize-1 else idx

    idx = (prob*geneSize).floor.toInt
    val gIdx = if(idx == 0) 1
               else if(idx == geneSize) geneSize-1 else idx
    GeneticIndices(chIdx, gIdx)
}
```

The `GeneticIndices` case class defines two indices of the bit whenever a crossover or a mutation occurs. The first index, `chOpIdx`, is the absolute index of the bit affected by the genetic operation in the chromosome. The second index, `geneOpIdx`, is the index of the bit within the gene subjected to crossover or mutation. The `geneticIndices` method of the `Population` class computes the two indices from a randomly generated value, `prob`, selected over the interval `[config.xover, 1]` for crossover and `[config.mu, 1]` for mutation.

Chromosomes

First, we need to define the `Chromosome` class, which takes a list of genes, `code` (for genetic code), as the parameter:

```
class Chromosome[T <: Gene] (val code: List[T])
```

The implementation of the crossover for a pair of chromosomes using hierarchical encoding follows two steps:

- Find the gene on each chromosome that corresponds to the crossover index, `gIdx.chOpIdx`, and then swap the remaining genes
- Split and splice the gene crossover at `xoverIdx`

Consider the following code:

```
def +- (that: Chromosome[T], gIdx: GeneticIndices): (Chromosome[T], Chromosome[T]) = {
    val xoverIdx = gIdx.chOpIdx //6
    val xGenes = spliceGene(gIdx, that.code(xoverIdx)) //7

    val offSprng1 = code.slice(0, xoverIdx) ::: xGenes._1 :: that.code.
drop(xoverIdx+1) //8
```

```
    val offSprng2 = that.code.slice(0, xoverIdx) ::: xGenes._2 :: code.  
    drop(xoverIdx+1)  
    (Chromosome[T](offSprng1), Chromosome[T](offSprng2)//9  
}
```

The crossover method computes the index of the bit that defines the crossover (`xoverIdx`) in each parent chromosome (line 6). The genes `this.code(xoverIdx)` and `that.code(xoverIdx)` are swapped and spliced by the `spliceGene` method to generate a spliced gene (line 7).

```
def spliceGene(gIdx: GeneticIndices, thatCode: T): (T, T) = {  
  ((this.code(gIdx.chOpIdx) +- (thatCode, gIdx)),  
   (thatCode +- (code(gIdx.chOpIdx), gIdx)) )  
}
```

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 8). The method returns the pair of offspring chromosomes (line 9).

Genes

The crossover is applied to a gene through the `+-`-method of the `Gene` class. The exchange of bits between the two genes `this` and `that` uses the `BitSet` Java class to rearrange the bits after the permutation:

```
def +- (that: Gene, idx: GeneticIndices): Gene = {  
  val clonedBits = cloneBits(bits) //10  
  
  Range(gIdx.geneOpIdx, bits.size).foreach(n =>  
    if( that.bits.get(n) ) clonedBits.set(n)  
    else clonedBits.clear(n)  
  ) //11  
  
  val valOp = decode(clonedBits) //12  
  Gene(id, valOp._1, valOp._2)  
}
```

The bits of the gene are cloned (line 10) and then spliced by exchanging their bits along the crossover point `xOverIdx` (line 11). The `cloneBits` function duplicates a bit string, which is then converted into a (target value, operator) tuple using the `decode` method (line 12). 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 the reader that the mutation is implemented by flipping one bit:

```
def ^ (mu: Double): Unit =
    chromosomes += chromosomes.map(_ ^ geneticIndices(mu))
```

The mutation parameter `mu` is used to compute the absolute index of the mutating gene, `geneticIndices(mu)`.

Chromosomes

The implementation of the mutation operator `^` on a chromosome consists of mutating the gene of the index `gIdx.chOpIdx` (line 1) and then updating the list of genes in the chromosome (line 2). The method returns a new chromosome (line 3) that will compete with the original chromosome:

```
def ^ (gIdx: GeneticIndices): Chromosome[T] = {
    val mutated = code(gIdx.chOpIdx) ^ gIdx
    val xs = Range(0, code.size).map(i =>
        if(i==gIdx.chOpIdx) mutated else code(i)).toList
    Chromosome[T](xs)
}
```

Genes

Finally, the mutation operator flips (XOR) the bit at the index `gIdx.geneOpIdx`:

```
def ^ (gIdx: GeneticIndices): Gene = {
    val clonedBits = cloneBits(bits) //4
    clonedBits.flip(idx.geneOpIdx) //5

    val valOp = decode(clonedBits) //6
    Gene(id, valOp._1, valOp._2) //7
}
```

The `^` method mutates the cloned bit string, `clonedBits` (line 4) by flipping the bit at the index `gIdx.geneOpIdx` (line 5). It decodes and converts the mutated bit string by converting it into a (target value, operator) tuple (line 6). The last step creates a new gene from the target-operator tuple (line 7).

The reproduction cycle

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 reproduction function, `mate`, 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 match {
    case 0 | 1 | 2 => false
    case _ => {
        population.select(score, config.softLimit(cycle))
        population +- (1.0 - Random.nextDouble * config.xover)
        population ^ (1.0 - Random.nextDouble * config.mu)
        true
    }
}
```

This method returns `true` if the size of the population is larger than 2. The last element of the puzzle is the exit condition. There are two options for estimating that the reproducing cycle is converging:

- **Greedy:** In this approach, the objective is to evaluate 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

A simple exit condition describes the state, of the type `GASTate`, of the genetic algorithm at each reproduction cycle:

```
def converge(population: Population[T], cycle: Int): GASTate = {
    if (population == null) GA_FAILED
    else if (iters >= config.cycles)
        GA_NO_CONVERGENCE(s"failed after $cycle cycles")
    ...
}
```

Let's define the state of the genetic algorithm as a case class of the super type `GASTate`:

```
sealed abstract class GASTate(val description: String)
case class GA_FAILED(val _description: String) extends GASTate(_description)
object GA_RUNNING extends GASTate("Running")
case class GA_NO_CONVERGENCE(val _desc: String) extends GASTate(_desc)
...
```

The last class `GASolver` manages the reproduction cycle and evaluates the exit condition or the convergence criteria:

```
class GASolver[T <: Gene](config: GAConfig, score: Chromosome[T] => Unit) extends PipeOperator[Population[T], Population[T]] {
    var state: GASTate = GA_NOT_RUNNING
```

This class implements the data transformation `|>`, which transforms a population to another one, given a configuration, `config` and a scoring method, `score`, as follows:

```
def |> : PartialFunction[Population[T], Population[T]] = {
    case population: Population[T] if (population.size > 1) => {
        val reproduction = Reproduction[T](score)
        state = GA_RUNNING

        Range(0, config.maxCycles).find(n => { //1
            reproduction.mate(population, config, n) match { //2
                case true => converge(population, n) != GA_RUNNING //3
                case false => { .... }
            }
        }) match {
            case Some(n) => population
            ...
        }
    }
}
```

The reproduction cycle is controlled by the `find` function (line 1) that tests whether an error occurs during the reproduction, `mate` (line 2), before the convergence criteria (line 3) are applied.

GA for trading strategies

Let's apply our fresh 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, described briefly in the *Technical analysis* section in *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 $x = \{x_i\}$, 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 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:

- Selecting a minimal set of trading signals that are relevant to a given data set (minimize a cost or unfitness function)
- Tuning 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 counter part 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 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(val _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: `<`, `>`, and `=`, as follows:

```
object LESS_THAN extends SOperator(1)
object GREATER_THAN extends SOperator(2)
...
```

Each operator is associated with a scoring function by the map `operatorFuncMap`. The function computes the 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),
    GREATER_THAN -> ((x: Double, target: Double) => x - target),
    ...
)
```

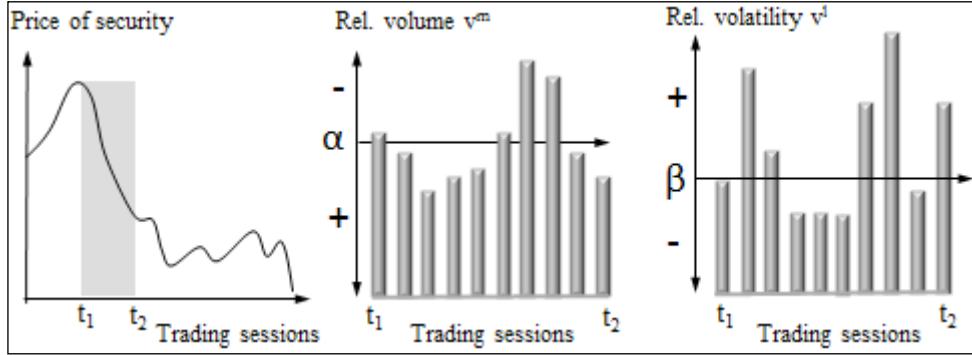
The `select` method of `Population` computes the unfitness value of a signal by quantifying the truthfulness of the predicate. For instance, the unfitness value for a trading signal, $x > 10$, is penalized as $5 - 10 = -5$ for $x = 5$ and credited as $14 - 10 = 4$ if $x = 14$. In this regard, the unfitness value is similar to the cost or loss in a discriminative machine learning algorithm.

The cost/unfitness 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 $v^l > \beta$

Have a look at the following graphs:



As the goal is to model a sudden crash in 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 stock price. For 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 stock price and a penalization $w = -\Delta p$:

$$w_i = -\Delta p_i$$

$$C(p, v^m, v^l | \alpha, \beta) = \sum_{i=0}^{n-1} (\alpha - v_i^m) w_i + (v_i^l - \beta) w_i$$

Trading signals

Let's subclass the `Gene` class to define the trading signal:

```
class Signal(_id: String, _target: Double, _op: Operator, xt: DblVector, weights: DblVector) (implicit discr: Discretization) extends Gene(_id, _target, _op)
```

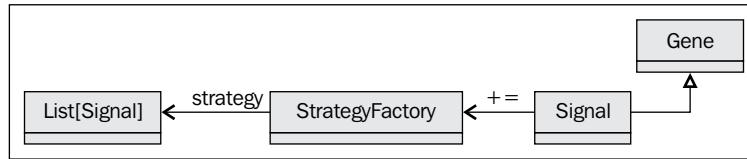
The `Signal` class takes the identifier for the feature, the `target` value, an operator `op`, the time series `xt` of the type `DblVector`, and the weights associated to each data point of the time series `xt`. The main purpose of the `Signal` class is to compute its score. The chromosome updates its unfitness 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`:

```
def score: Double = sumScore(operatorFuncMap.get(op).get)
def sumScore(f: (Double, Double) => Double): Double = xt.foldLeft(0.0)
  ((s, x) => s + f(x, target))
```

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 type `List[Signal]` from an existing pool of signals of the subtype `Gene`:



The `StrategyFactory` class has two arguments: the number of signals, `nSignals`, in a trading strategy and the implicit discretization instance:

```

class StrategyFactory(nSignals: Int) (implicit descr: Discretization) {
  val signals = new ListBuffer[Signal]
  lazy val strategies: Pool[Signal]
  ...
}
  
```

The `+=` method adds the trading signals to the factor. The `StrategyFactory` class generates all possible sequences of signals as trading strategies. The `+=` method takes five arguments: the identifier (`id`), target, operation (`op`) to qualify the class as a Gene, the times series `xt` for scoring the signals, and the `weights` associated to the overall cost function:

```

def += (id: String, target: Double, op: Operator, xt: XTSeries[Double], weights: DblVector): Unit =
  signals.append(Signal(id, target, op, xt.toArray, weights))
  
```

The `StrategyFactory` class defines `strategies` as lazy values to avoid unnecessary regeneration of the pool on demand:

```

lazy val strategies: Pool[Signal] = {
  implicit val ordered = Signal.orderedSignals //1

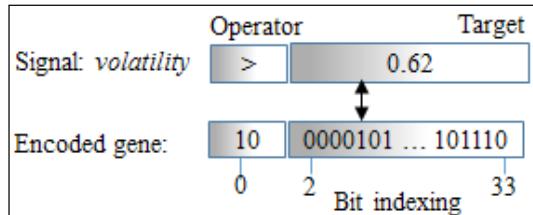
  val xss = new Pool[Signal] //2
  val treeSet = new TreeSet[Signal] ++= signals.toList //3
  val subsetsIterator = treeSet.subsets(nSignals) //4
  while( subsetsIterator.hasNext) { //5
    val subset = subsetsIterator.next
    val signalList: List[Signal] = subset.toList //6
    xss.append(Chromosome[Signal](signalList)) //7
  } xss
}
  
```

The implementation of the `strategies` value creates `Pool` (line 1) by converting the list of signals to a `treeset` (line 2). It breaks down the tree set into unique subtrees of `nSignals` nodes each (line 3). It instantiates a `subsetsIterator` iterator (line 3) to traverse the sequence of subtrees (line 4) and converts them into a list (line 5) as arguments of the new chromosome (trading strategy) (line 6). The procedure to order the signals, `orderedSignals`, in the tree set has to be implicitly defined (line 7):

```
val orderedSignals = Ordering.by((signal: Signal) => signal.id)
```

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 price volatility as a gene

IEEE-732 encoding

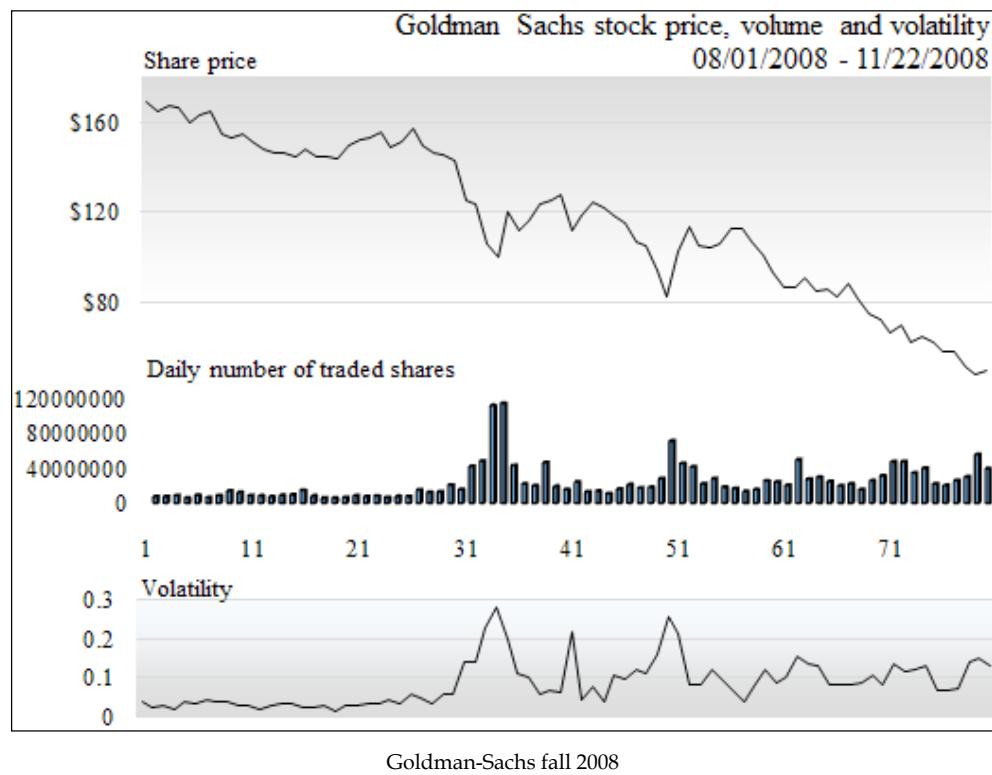
The threshold value for predicates is converted into an integer (the type `Int` or `Long`). The IEEE-732 binary representation of floating point values makes the bit addressing required to apply genetic operators quite challenging. A simple conversion consists of the following:

```
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.

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 (`deltaPrice`), the model uses the following parameters:

- `deltaVolume`: This is the relative variation of the volume between two consecutive trading sessions
- `deltaVolatility`: This is the relative variation of volatility between two consecutive trading sessions
- `relVolatility`: This is the relative volatility within a trading session
- `relCloseOpen`: This is the relative difference of the stock opening and closing price

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 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.

Data extraction

The first step is to extract the model parameters as illustrated for the variation of the stock price between two consecutive trading sessions:

```
val path = "resources/data/chap10/GS.csv"
val src = DataSource(path, false, true, 1)
val price = src |> YahooFinancials.adjClose
val deltaPrice = price.drop(1)
    .zip(price.dropRight(1))
    .map(p => (1.0 - p._2/p._1))
```

The extraction of relative variation in volume and volatility is similar to the extraction of the relative variation of the stock price.

Initial population

The next step consists of generating the initial population of strategies that compete to become relevant to the decline of the price of stocks of Goldman Sachs. The factory is initialized with a set of signals:

```
val NUM_SIGNALS_PER_STRATEGY = 3
val factory = new StrategyFactory(NUM_SIGNALS_PER_STRATEGY)
factory += ("Delta_volume", 1.1, GREATER_THAN, deltaVolume,
deltaPrice)
factory += ("Rel_volatility", 1.3, GREATER_THAN, relVolatility.
drop(1), deltaPrice)
...

```

The test code generates population by retrieving the pool of strategies:

```
val limit = factory.strategies.size // 1 <<4
val population = Population[Signal](limit, factory.strategies)
```

The maximum size of the population (hard limit) is arbitrarily set as 16 times the number of the initial trading strategies (line 1).

At this stage, we need to instantiate a `Discretization` instance:

```
val R=1024.0
implicit val digitize = new Discretization(R)
```

Configuration

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:

```
val XOVER = 0.2; val MU = 0.6; val MAX_CYCLES = 250
val CUTOFF_SLOPE = -0.003; val CUTOFF_INTERCEPT = 1.003

val softLimit = (n: Int) => CUTOFF_SLOPE*n + CUTOFF_INTERCEPT
val config = GAConfig(XOVER, MUTATE, MAX_NUM_ITERS, softLimit)
```

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.

GA instantiation

Let's implement the chromosome scoring function using the formula introduced in the cost/unfitness section. The trading strategy/chromosome scoring function sums up the score for each gene and updates it:

```
val scoring = (chr: Chromosome[Signal]) => {
    val signals: List[Gene] = chr.code
    chr.unfitness = signals.foldLeft(0.0)((s, x) => s + x.score)
}
```

The configuration `config` and the scoring function, `scoring`, are all you need to create and execute the solver `gaSolver`:

```
val gaSolver = GASolver[Signal](config, scoring)
```

GA execution

The execution of the genetic algorithm transforms an initial population to a very small group of the NFIITS fittest trading strategies:

```
val NFIITS = 2
val best = gaSolver |> population
best.fittest(NFIITS)
    .getOrElse(ArrayBuffer.empty)
    .foreach(ch => Display.show(s"Best: ${ch.toString(" ")}", logger))
...

```

Tests

The cost function C and the unfitness score of each trading strategy are weighted for the rate of decline of the price of the Goldman Sachs stock. Let's run two tests:

- Evaluation of the genetic algorithm with an unweighted score function
- Evaluation of the configuration of the genetic algorithm with the weighted score

The unweighted score

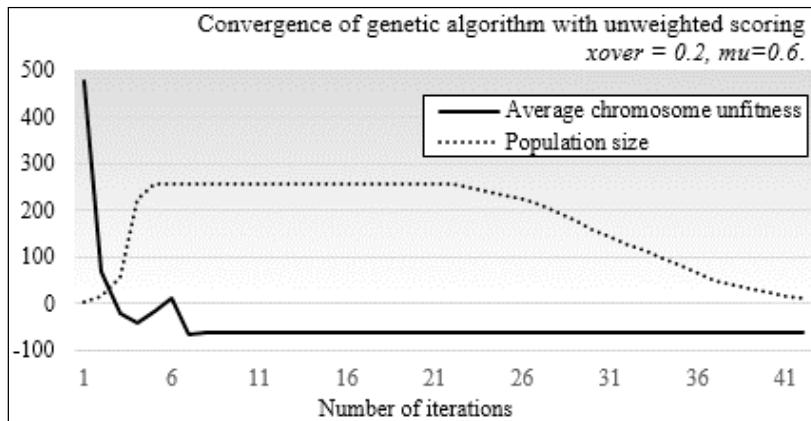
The test uses three different sets of crossover and mutation ratios: (0.6, 0.2), (0.3, 0.1), and (0.2, 0.6). The best trading strategy for each scenario are as follows:

- **0.6-0.2:** For this, $\text{Delta_volume} > 1.10$, $\text{Rel_close-Open} > 0.75$, and $\text{Rel_volatility} > 0.97$ with average chromosome unfitness = 0.025
- **0.3-0.1:** For this, $\text{Delta_volatility} > 0.9$, $\text{Rel_close-Open} < 0.8$, and $\text{Rel_volatility} > 1.77$ with unfitness = 0.100
- **0.2-0.6:** For this, $\text{Delta_volatility} > 0.9$ $\text{Delta_volume} > 33.09$, and $\text{Rel_volatility} > 1.09$ with unfitness = 0.099

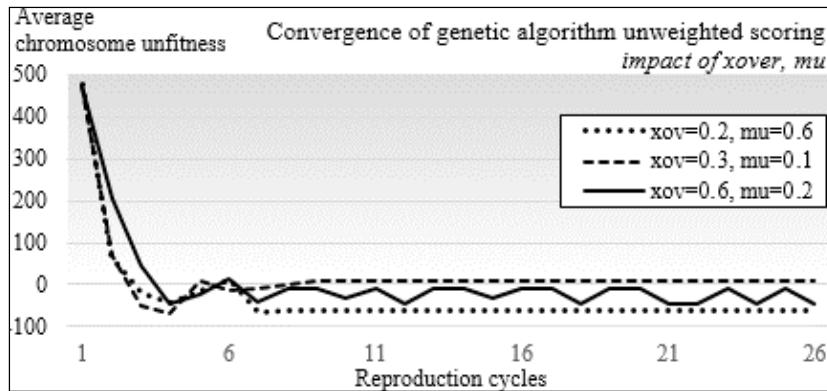
The fittest trading strategy for each case does not differ much from the initial population for one or several of the following reasons:

- 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

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: Have a look at the following graph:



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 a different values of crossover and mutation ratio, as shown in the following graph:

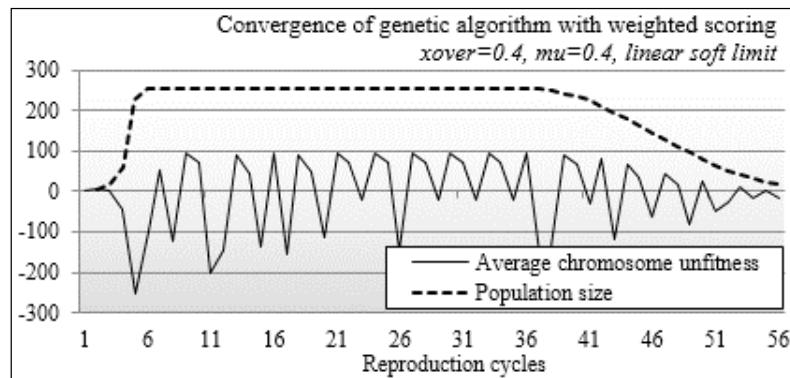


The profile of the execution of the genetic algorithm is not overly affected by the different values of crossover and mutation ratios. The chromosome unfitness score for the high crossover ratio, 0.6, oscillates as the execution progresses. In some cases, the unfitness score between chromosomes is so small that the GA recycles the same few trading strategies.

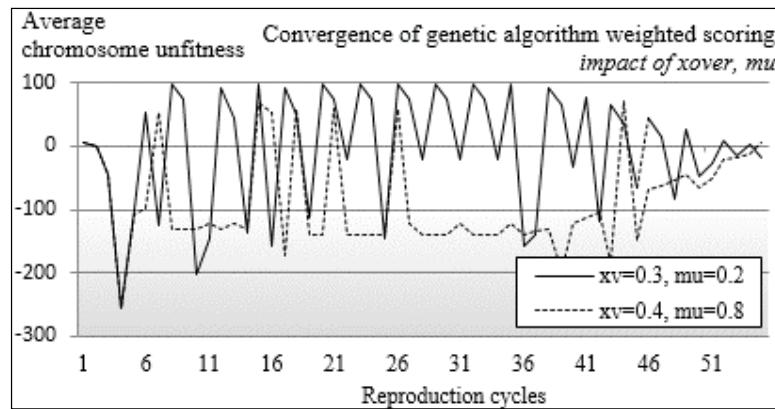
The quick decline in the unfitness of the chromosomes is consistent with the fact that some of the fittest strategies were part of the initial population. It should, however, raise some concerns that the GA locked on a local minimum early on.

The weighted score

The execution of a test that is similar to the previous one with the weighted unfitness scoring formula produces some interesting results, as shown in the following graph:



The profile for the size of the population is similar to the test using unweighted unfitness. However, the average chromosome unfitness does not stabilize as the optimization goes on until the size of the population is reduced by the soft limit function. This phenomenon is confirmed by running the test using different configurations, as shown in the following graph:



The weighting function adds the rate of decline of the stock price into the scoring of the unfitness. The formula to compute the cost/unfitness of a trading strategy is not a linear function; its complexity increases the odds of the genetic algorithm not converging properly, which is confirmed with extra runs with different values of the crossover and mutation ratios.

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

It should be clear by now 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 solutions.
- Have discrete, discontinuous, and non-differentiable 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 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

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 introduced in the *Learning classifier systems* section of 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 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:

- Basic concepts behind reinforcement learning
- 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.

Introduction

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 non-sequential 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 *The hidden Markov model (HMM)* section in *Chapter 7, Sequential Data Models*), which are not always available. However, a reinforcement learning algorithm benefits from a hidden Markov model in case 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, 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 schedule and routing problems.

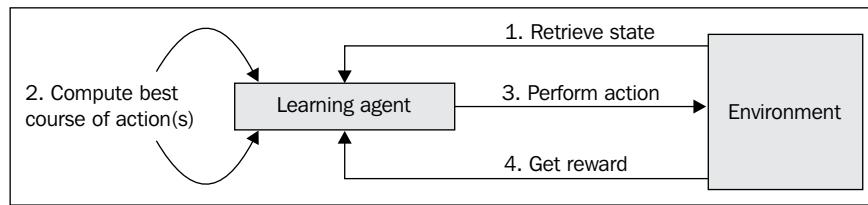
Terminology

Reinforcement learning introduces a new terminology as listed here, quite different from that of older machine learning techniques:

- **Environment:** The environment 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 in.
- **Agent:** The agent 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:** A goal state is the state that provides a higher discounted cumulative rewards than any other state. It is a constraint on the training process that prevents the best policy from being dependent on the initial state.
- **Action:** An action 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:** The policy 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:** A reward 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:** The horizon 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.

Concept

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 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 either 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 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 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 state S_t to 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**.



State transition matrix

The state transition matrix has 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
$S = \{s_i\}$	States of the environment
$A = \{a_j\}$	Actions on the environment
$\Pi_t = p(a_t s_t)$	Policy (or strategy) of the agent
$V^\pi(s_j)$	Value of the policy at the state
$p_{t,i} = p(s_{t+1} s_t, a_i)$	State transition probabilities from state s_t to state s_{t+1}

Notation	Description
$r_t = p(r_{t+1} s_t, s_{t+1}, a_t)$	Reward of an action a_t for a state s_t
R_t	Expected discounted long term return
γ	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 state s_t is the maximum expected reward R_t given the state s_t .

The value of a policy π at state s_t with reward r_j in previous state s_j :



$$R_t = \sum_{k=0}^{+\infty} \gamma^k r_{t+k}$$

$$V^\pi(s_t) = E\{R_t | s_t\}$$

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 .

Value of state s_t using the transition probability



$$V^\pi(s_t) = \sum_{a \in A} \pi_t(a) \sum_k \left\{ p_k \left(r_k + \gamma \cdot V^\pi(s_k) \right) \right\}$$

$$V^*(s_t) = \max_\pi V^\pi(s_t)$$

$V^*(s_i)$ is the optimal value of state s_i 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 insolvable. 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 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:

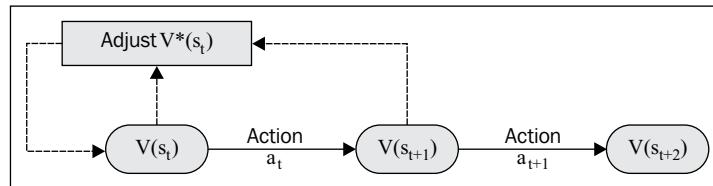


Illustration of the temporal difference algorithm

The iterative feedback loop used to adjust the value action on the state plays a role similar to back propagation 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 impacts 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 .



Formula for tabular temporal difference:

$$\left[\begin{array}{c} \delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \\ \hat{V}^\pi(s_t) = V^\pi(s_t) + \alpha \delta_t \end{array} \right]$$

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) $Q^\pi(s_t, a_t)$.



Value of action at state s_t

$$\left[\begin{array}{c} Q_t^\pi = Q^\pi(s_t, a_t) = E(R_t | s_t, a_t) \end{array} \right]$$

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, $\max\{Q_t\}$, for the current state s_t and action a_t on the environment to transition to 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.



Q-learning action-value updating formula:

$$\hat{Q}_t^\pi = Q_t^\pi + \alpha \left[r_{t+1} + \gamma \max_{a_{t+1}} Q^\pi(s_{t+1}, a_{t+1}) \right] - Q_t^\pi$$

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 us implement the Q-learning algorithm in Scala.

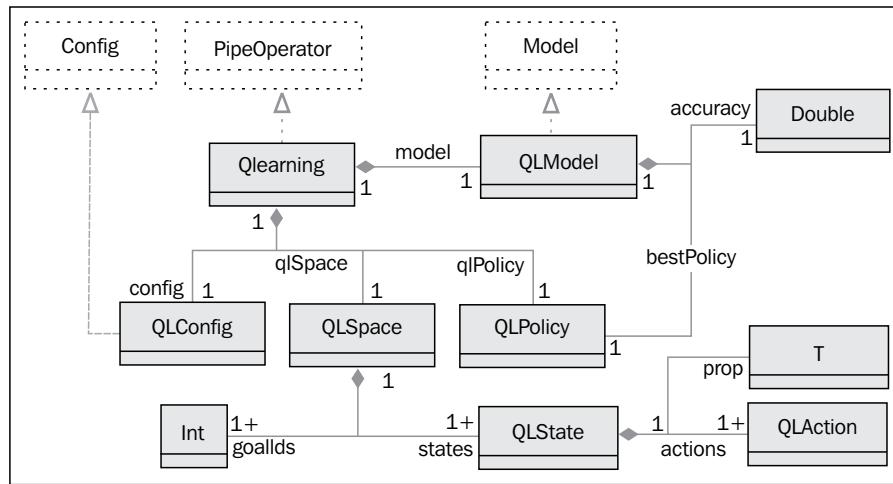
Software design

The key components of the implementation of the Q-learning algorithm are as follows:

- The `QLearning` class implements training and prediction methods. It defines a data transformation by implementing the `PipeOperator` trait. The constructor has three arguments: a configuration of type `QLConfig`, a search space of type `QLSpace`, and a mutable policy of type `QLPolicy`.

- The `QLSpace` class has two components: a sequence of states of type `QLState` and the 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.
- The model of type `QLModel` is generated through training. It contains the best policy and the accuracy for a model.

The following diagram shows the flow of the Q-learning algorithm:



States and actions

The `QLAction` class specifies the transition of one state with ID `from` to another state with ID `to`, as shown here:

```
class QLAction[T <% Double] (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 type `QLState` is fully defined by its ID, the list of actions to transition to some other states, and a property `prop` of parameterized type, as shown in the following code:

```
class QLState[T] (val id: Int, val actions: List[QLAction[T]]=List.empty, val prop: T)
```

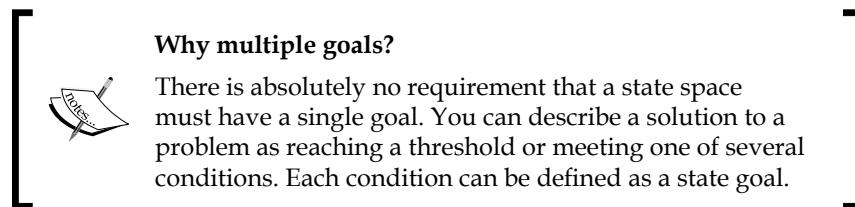
The state might not have any actions. This is usually the case of the goal or absorbing state. In this case, the list is empty. The parameterized `prop` property is a placeholder for any information, heuristic about the state, or any action performed by the state.

The next step consists of creating the graph or search space.

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



The `QLSpace` class can be implemented as follows:

```
class QLSpace[T] (states: Seq[QLState[T]], goals: Array[Int]) {
    val statesMap = states.map(st => (st.id, st)).toMap //1
    val goalStates = new HashSet[Int]() ++ goals //2

    def maxQ(state: QLState[T], policy: QLPolicy[T]): Double //3
    def init(r: Random) = states(r.nextInt(states.size-1)) //4
    def nextStates(st: QLState[T]): List[QLState[T]] //5
    ...
}
```

The instantiation of the `QLSpace` class generates a map, `statesMap`, to retrieve the state using its `id` (line 1) and the set of goals, `goalStates` (line 2). Furthermore, the `maxQ` method computes the maximum action-value, `maxQ`, for a state given a policy (line 3), the `init` method selects an initial state for training episodes (line 4), and finally, the `nextStates` method retrieves the list of states resulting from the execution of all the actions associated to the `st` state (line 5).

The search space is actually created by the instance factory defined in the `QLSpace` companion object, as shown here:

```
def apply[T](numStates: Int, goals: Array[Int], features: Set[T],  
neighbors: (Int, Int) => List[Int]): QLSpace[T] = {  
    val states = features  
        .zipWithIndex  
        .map(x => {  
            val actions = neighbors(x._2, numStates)  
                .map(j => QLAction[T](x._2, j))  
                .filter(x._2 != _.to)  
            QLState[T](x._2, actions, x._1  
        })  
    new QLSpace[T](states.toArray, goals)  
}
```

The `apply` method creates a list of states using the `features` set as input. Each state creates its list of actions. The user-defined function, `neighbors`, constrains the number of actions assigned to each state. The test case describes a very simple implementation of the `neighbors` function, which is defined in the configuration.

Policy and action-value

Each action has an action-value, a reward, and a potentially probability. The probability variable is introduced to model simply 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 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-value, 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 variables: `reward`, `probability`, and `value` for the Q-value, as shown here:

```
class QLData(var reward: Double = 1.0, var probability: Double = 1.0  
var value: Double = 0.0) {  
    def estimate: Double = value*probability  
}
```

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 uses variables instead of values as recommended by the best Scala coding practices [11:4]. An instance of an immutable class would be created for each action or state transition. 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 us 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 = 1.0, val probability: Double = 1.0)
```

The first two arguments are the identifiers for the source state, `from`, and target state, `to`, 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 type `QLPolicy`. It is merely a data container, as shown here:

```
class QLPolicy[T](numStates: Int, input: Array[QLInput]) {
    val qlData = {
        val data = Array.tabulate(numStates)(v =>
            Array.fill(numStates)(new QLData[T]()))
        input.foreach(i => {
            data(i.from)(i.to).reward = i.reward //1
            data(i.from)(i.to).probability = i.probability //2
        })
        data
    }
    ...
}
```

The constructor initializes the `qlData` matrix of type `QLData` with the input data, reward (line 1) and probability (line 2). The `QLPolicy` class defines the methods of the element in the reward (line 3), probability, and Q-learning action-value (line 4) matrices as follows:

```
def R(from: Int, to: Int): Double = qlData(from)(to).reward //3
def Q(from: Int, to: Int): Double = qlData(from)(to).value //4
```

The Q-learning training

The `QLearning` class encapsulates the Q-learning algorithm, and more specifically the action-value updating equation. It implements `PipeOperator` to the prediction used as a transformation between states, as shown here:

```
class QLearning[T](config: QLConfig, qlSpace: QLSpace[T], qlPolicy: QLPolicy[T]) extends PipeOperator[QLState[T], QLState[T]]
```

The constructor takes the following parameters:

- Configuration of the algorithm, `config`
- Search space, `qlSpace`
- Policy, `qlPolicy`

The model is generated or trained during the instantiation of the class (refer to the *Design template for classifier* section in *Appendix A, Basic Concepts.*)

The configuration defines the learning rate, `alpha`; the discount rate, `gamma` the maximum number of states (or length) of an episode, `episodeLength`; the number of episodes used in training, `numEpisodes`; the minimum coverage of the state transition/actions during training to select the best policy, `minCoverage`; and the search constraint function, `neighbors`, as shown here:

```
class QLConfig(val alpha: Double, val gamma: Double, val
  episodeLength: Int, val numEpisodes: Int, val minCoverage: Double, val
  neighbors: (Int, Int) => List[Int]) extends Config
```

Let us look at the computation of the best policy during training. First, we need to define a model class, `QLModel`, with the best policy and its state-transition coverage of training as parameters:

```
class QLModel[T](val bestPolicy: QLPolicy[T], val coverage: Double)
```

The creation of `model` consists of executing multiple episodes to extract the best policy. Each episode starts with a randomly selected state, as shown in the following code:

```
val model: Option[QLModel[T]] = {
    val r = new Random(System.currentTimeMillis) //1
    val rg = Range(0, config.numEpisodes)
    val cnt = rg.foldLeft(0)((s, _) => s+(if(train(r)) 1 else 0))/2

    val accuracy = cnt.toDouble/config.numEpisodes
    if( accuracy > config.minCoverage )
        Some(new QLModel[T](qlPolicy, coverage)) //3
    else None
}
```

The model initialization code creates a random number generator (line 1), and iterates the generation of the best policy starting from a randomly selected state `config.numEpisodes` times (line 2). The transition coverage is computed as the percentage of times the search ends with the goal state (line 3). The initialization succeeds only if the accuracy exceeds a threshold value, `config.minCoverage`, specified in the configuration.



Quality of the model

The implementation uses the coverage to measure the quality of the model or best policy. The F1 measure (refer to the *Assessing a model* section in *Chapter 2, Hello World!*), is not appropriate because there are no false positives.

The `train` method does the heavy lifting at each episode. It triggers the search by selecting the initial state using a random generator `r` with a new seed, as shown in the following code:

```
def train(r: Random): Boolean = {
    r.setSeed(System.currentTimeMillis*Random.nextInt)
    qlSpace.isGoal(search((qlSpace.init(r), 0))._1)
}
```

The implementation of `search` for the goal state(s) from any random states is a textbook implementation of the Scala tail recursion.

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:

```
@scala.annotation.tailrec
def search(st: (QLState[T], Int)): (QLState[T], Int) = {
    val states = qlSpace.nextStates(st._1) //1

    if( states.isEmpty || st._2 >= config.episodeLength ) st //2
    else {
        val state = states.maxBy(s => qlPolicy.R(st._1.id,s.id ))//3

        if( qlSpace.isGoal(state) ) (state, st._2) //4
        else {
            val r = qlPolicy.R(st._1.id, state.id)
            val q = qlPolicy.Q(st._1.id, state) //5
            val nq = q + config.alpha*(r + config.gamma *
                qlSpace.maxQ(state, qlPolicy) - q)//6
            qlPolicy.setQ(st._1.id, state.id, nq) //7
            search((state, st._2))
        }
    }
}
```

Let us dive into the implementation for the Q action-value updating equation. The recursion uses the tuple (state, iteration number in the episode) as argument. First, the recursion invokes the `nextStates` method of `QLSpace` to retrieve all the states associated with the current state, `st`, through its actions, as shown here:

```
def nextStates(st: QLState[T]): List[QLState[T]] =
    st.actions.map(ac => statesMap.get(ac.to).get)
```

The search completes and returns the current state if either 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 2). Otherwise the recursion computes the state to which the transition generates the higher reward R from the current policy (line 3). The recursion returns the state with the highest reward if it is one of the goal states (line 4). The method retrieves the current q action value and r reward matrices from the policy, and then applies the equation to update the action-value (line 6). The method updates the action-value Q with the new value nq (line 7).

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[T]): Double = {
    val best = states.filter(_ != state)
        .maxBy(st => policy.EQ(state.id, st.id))
    policy.EQ(state.id, best.id)
}
```

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.

Prediction

The last functionality of the `QLearning` class is the prediction using the model created during training. The method predicts a state from an existing state.

```
def |> : PartialFunction[QLState[T], QLState[T]] = {
    case state: QLState[T] if(state != null && model != None)
        => nextState(state, 0)._1
}
```

The data transformation `|>` computes the best outcome, `nextState`, given a state using another tail recursion, as follows:

```
@scala.annotation.tailrec
def nextState(st: (QLState[T], Int)): (QLState[T], Int) = {
    val states = qlSpace.nextStates(st._1)

    if( states.isEmpty || st._2 >= config.episodeLength) st
    else nextState( (states.maxBy(s =>
        model.get.bestPolicy.R(st._1.id, s.id)), st._2+1))
}
```

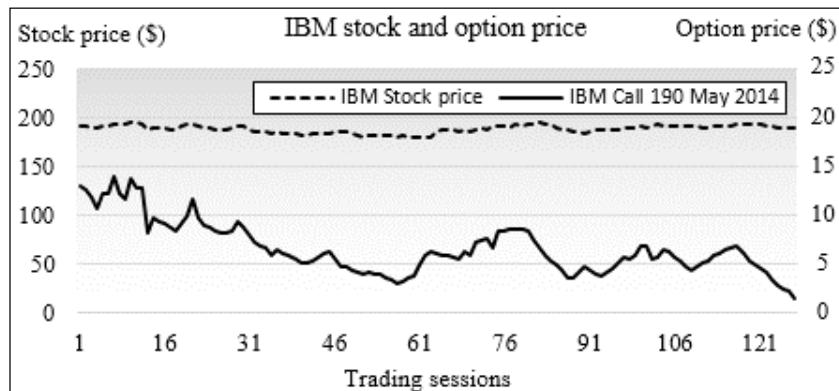
The prediction ends when no more states are available or the maximum number of iterations within the episode is exceeded. You can define a more sophisticated exit condition. The challenge is that there is no explicit error or loss variable/function that can be used except the temporal difference error. The prediction 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 us 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 giving 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 *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 IBM stock and its derivative call option for May 2014 with a strike price of \$190:



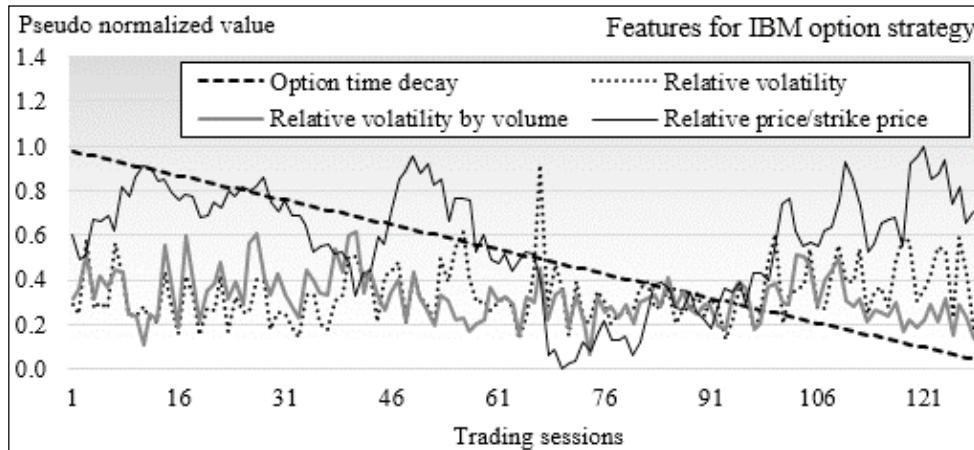
The price of an option depends on the following parameters:

- Time to expiration of the option (time decay)
- The price of the underlying security
- The volatility of returns of the underlying asset

Pricing model usually does not take into account the variation in trading volume of the underlying security. So it would be quite interesting to include it in our model. Let us define the state of an option using the following four normalized features:

- **Time decay:** This is the time to expiration once normalized over [0, 1].
- **Relative 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:** 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:** 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 IBM option strategy:



The implementation of the option trading strategy using Q-learning consists of the following steps:

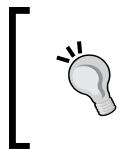
1. Describing the property of an option
2. Defining the function approximation
3. Specifying the constraints on the state transition

Option property

Let us 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 as follows:

```
class OptionProperty(timeToExp: Double, relVolatility: Double,  
volatilityByVol: Double, relPriceToStrike: Double) {  
    val toArray = Array[Double](timeToExp, relVolatility,  
    volatilityByVol, relPriceToStrike)  
}
```



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.

Option model

The OptionModel class is a container and a factory for the properties of the option. It creates the list of option properties, propsList, 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 option, `strikePrice`.
- 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 *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`, 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 = {  
        val volatility = normalize((src |> relVolatility).get.toArray)  
        val rVolByVol = normalize((src |> volatilityByVol).get.toArray)  
        val priceToStrike = normalize(price.map(p => 1.0-strikePrice/p)  
  
        volatility.zipWithIndex //1  
            .foldLeft(List[OptionProperty]())((xs, e) => {
```

```

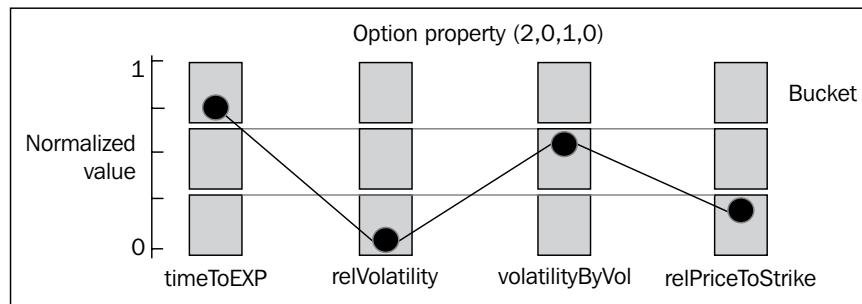
val normDecay = (e._2+minExpT).toDouble/(price.size+minExpT) //2
new OptionProperty(normDecay, e._1, volByVol(e._2), priceToStrik
e(e._2)) :: xs
}).drop(2).reverse
}

```

The factory uses the `zipWithIndex` Scala method to model the index of the trading sessions (line 1). All feature values are normalized over the interval [0, 1], including the time decay (or time to expiration) of the `normDecay` option (line 2).

Function approximation

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 discretization or categorization known as a **function approximation**, although a function approximation scheme can be quite elaborate [11:9]. Let us 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 $l^{2n} - l^n$ actions.

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 `approximate` 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 approximate(y: DblVector): Map[Array[Int], Double] = {
    val mapper = new HashMap[Int, Array[Int]] //1

    val acc = new NumericAccumulator //2
    propsList.map(_.toArray)
        .map(toArrayInt(_)) //3
        .map(ar => {
            val enc = encode(ar) //4
            mapper.put(enc, ar)
            enc })
    .zip(y)
    .foldLeft(acc)((acc, t) => {acc += (t._1, t._2); acc}) //5
    acc.map(kv => (kv._1, kv._2._2 / kv._2._1)) //6
    .map(kv => (mapper(kv._1), kv._2)).toMap
}
```

The method creates a `mapper` instance to index the array of buckets (line 1). An accumulator, `acc`, of type `NumericAccumulator` extends `Map[Int, (Int, Double)]` and computes the tuple (number of occurrences of features on each buckets, the sum of increase or decrease of the option price) (line 2). The `toArrayInt` method converts the value of each option property (`timeToExp`, `relVolatility`, and so on) into the index of the appropriate bucket (line 3). The array of indices is then encoded (line 4) 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 5). It finally computes the reward on each action by averaging the profit and loss on each bucket (line 6).

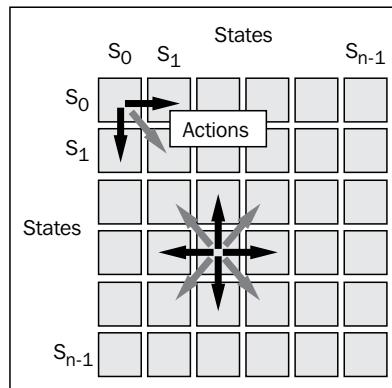
```
def toArrayInt(feature: DblVector): Array[Int] =
    feature.map(x => (nSteps*x).floor.toInt)
```

Constrained state-transition

Each state is potentially connected to any other state through actions. There are two methodologies to reduce search space or number of actions/transitions:

- Static constraint defines the actions/transition when the model is instantiated. The state transition map is fixed for the entire life cycle of the model.

- Dynamic constraint relies on the probability of an action to prevent or hinder state transitions.



The implementation of the static constraint avoids the unnecessary creation of a large number of `QLAction` object at the expense of the inability to modify the search space during training. The test case uses the static constraint as defined in the `neighbors` function passed as a parameter of the `QLSpace` class:

```
val RADIUS = 4
val neighbors = (idx: Int, numStates: Int) => {

  def getProximity(idx: Int, radius: Int): List[Int] = {
    val idx_max = if(idx + radius >= numStates) numStates-1 else idx+radius
    val idx_min = if(idx < radius) 0 else idx - radius
    Range(idx_min, idx_max+1).filter(_ != idx)
      .foldLeft(List[Int]())((xs, n) => n :: xs)
  }
  getProximity(idx, RADIUS).toList
}
```

The `neighbors` function restrains the number of actions to up to `RADIUS*2` states, depending on the ID, `idx`, of the state. The function is implemented as a closure: it requires the value `numStates` to be defined within the function or in its outer scope.

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 stockPricePath = "resources/data/chap11/IBM.csv"
val optionPricePath = "resources/data/chap11/IBM_O.csv"
```

```
val MIN_TIME_EXP = 6; val APPROX_STEP = 3; val NUM_FEATURES = 4
val ALPHA = 0.4; val DISCOUNT = 0.6; val NUM_EPISODES = 202520

val src = DataSource(stockPricePath, false, false, 1) //1
val ibmOption = new OptionModel("IBM", 190.0, src, MIN_TIME_EXP,
APPROX_STEP) //2

DataSource(optionPricePath, false, false, 1) extract match {
case Some(v) => initializeModel(ibmOption, v)
...
}
```

The client code instantiates the option model, `ibmOption`, for the IBM stock (line 1). It invokes the `initializeModel` method once the historical price of the option is downloaded through the appropriate data source (line 2). The `initializeModel` method does all the work as shown in the following code:

```
def initializeModel(ibmOption: OptionModel, oPrice: DblVector): QLearning[Array[Int]] {
    val fMap = ibmOption.approximate(oPrice) //3
    val input = new ArrayBuffer[QLInput]

    val profits = fMap.values.zipWithIndex
    profits.foreach(v1 =>
        profits.foreach( v2 =>
            input.append(new QLInput(v1._2, v2._2, v2._1-v1._1))) //4

    val goal = input.maxBy( _.reward).to
    val config = new QLConfig(ALPHA, DISCOUNT, EPISODE_LEN, NUM_
    EPISODES, MIN_ACCURACY, getNeighbors)
    QLearning[Array[Int]](config, fMap, goal, input.toArray, fMap.
    keySet)
}
```

The `initializeModel` method generates the approximation map, `fMap` (line 3), which contains the profit and loss for each state. Next, the method initializes the input to the policy by computing the reward as the difference of the profit/loss of the source `v1` and the destination `v2` of each action (line 4). The goal is initialized as the action with the highest reward (line 5). The last step is the instantiation of the `QLearning` class that executes the training.



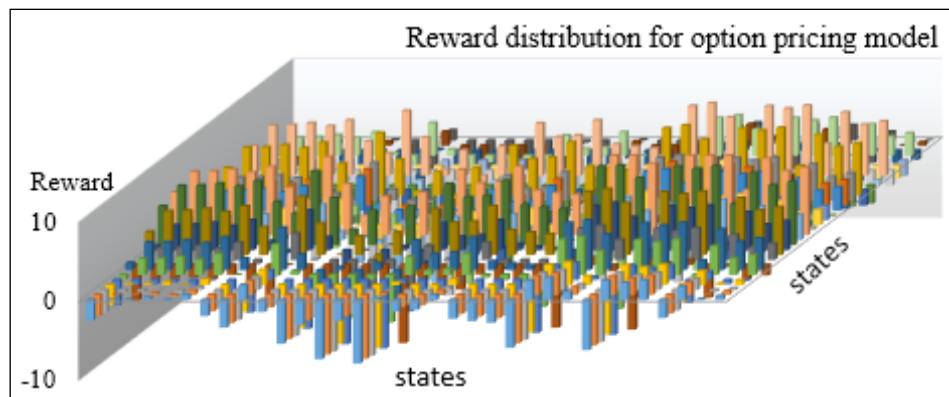
The anti-goal state

The goal state is the state with the highest assigned reward. It is a heuristic to reward a strategy for 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.

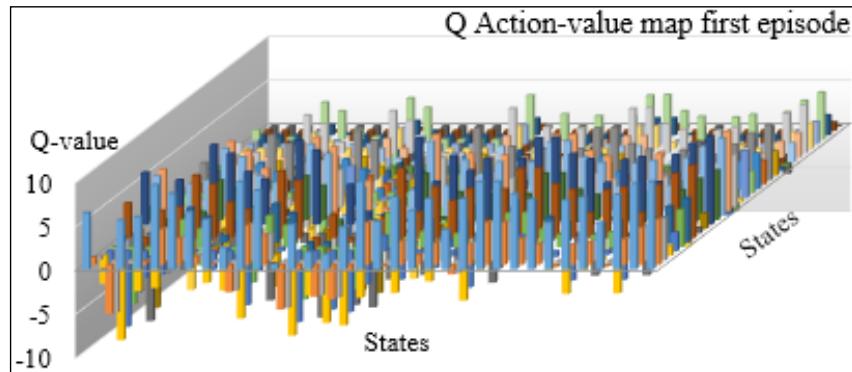
The initialization of the Q-learning model generates the following reward matrix:



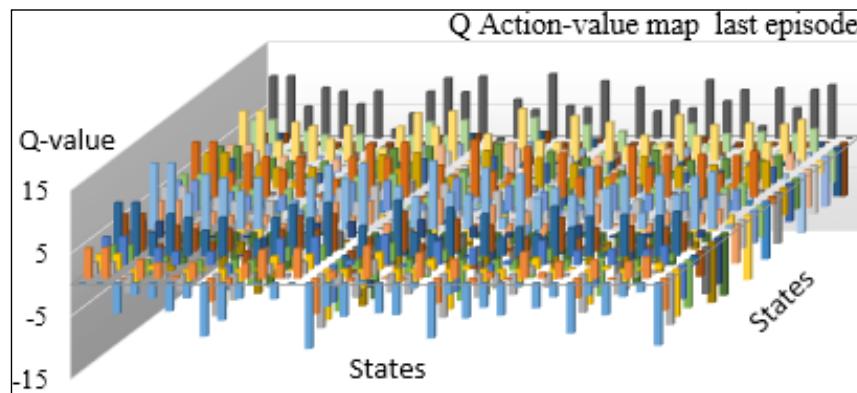
The graph visualizes the distribution of the rewards computed from the profit and loss of the option. The xy plane represents the actions between states. The states' IDs are listed on x and y axes. The z-axis measures the actual value of the reward associated with each action.

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, 0.4, to reduce the impact of the previous state on the new state. The value for the discount rate, 0.6, 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 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 with the map generated after 20 episodes in the following graph:



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 at least one data point.
- However, a loose function approximation 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.

Pros and cons of reinforcement learning

Reinforcement learning algorithms are ideal for the following problems:

- Online learning
- The training data is small or non-existent
- A model is non-existent 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 causing the maintenance of the states, action graph, and rewards matrix become 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 classifier systems (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

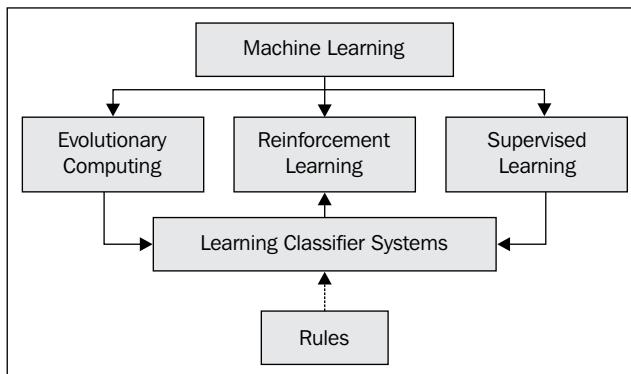


Diagram of the scientific disciplines required for learning classifier systems

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 that 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 that 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 that measures the positive or negative impact of the actions from the fittest classifiers or policies.
- A reinforcement learning component that 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 combination of 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 in 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

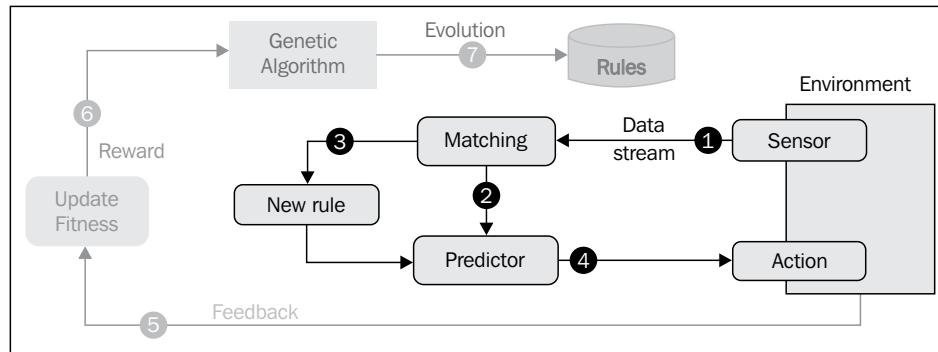
Each domain of research has its own terminology and LCS is no exception.

The terminology of LCS consists of the following terms:

- **Environment:** 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:** 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:** 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:** 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:** 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 of change in environment.
- **Sensors:** Environment variables monitored by agent, for example, temperature and hard drive status.
- **Input data stream:** Flow of data generated by sensors. It is usually associated with online training.
- **Rule matching:** Mechanism to match a predicate or compound predicate with a sensor.
- **Covering:** 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:** An algorithm to find the action with the maximum number of occurrences within a set of matching rules.

Extended learning classifier systems (XCS)

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.

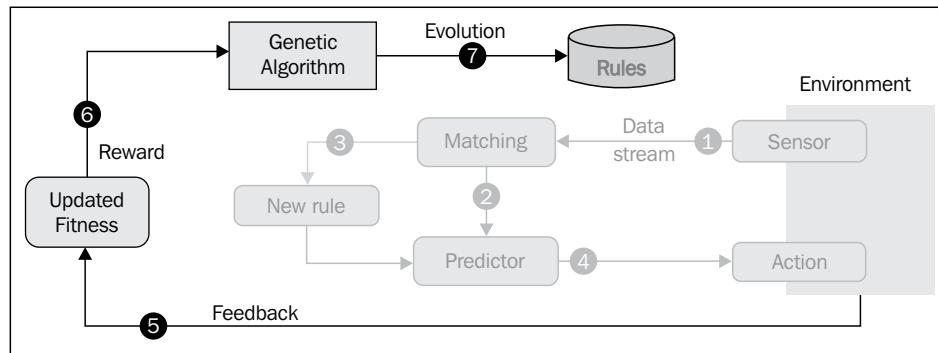


Exploitation component of the XCS algorithm

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 searched and 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:

- 5: 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.
- 6: Rewards are used to update the rule fitness, applying evolutionary constraints to the existing population.
- 7: 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 (ETFs)** 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 can be done referring to the following table:

XCS component	Portfolio management
Environment	Portfolio of securities defined by its composition, total value, and the yield of the 10-year Treasury bond
Action	Change in the composition of the portfolio
Reward	Profit and loss of the total value of the portfolio
Input data stream	Feed of stock and bond price quotation
Sensor	Trading information regarding securities in the portfolio such as price, volume, volatility, or yield, and the yield on the-10 year Treasury bond
Predicate	Change in composition of the portfolio
Action	Rebalancing a portfolio by buying and selling securities
Rule	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, much like the initial population of a genetic algorithm, or be defined by a domain expert.

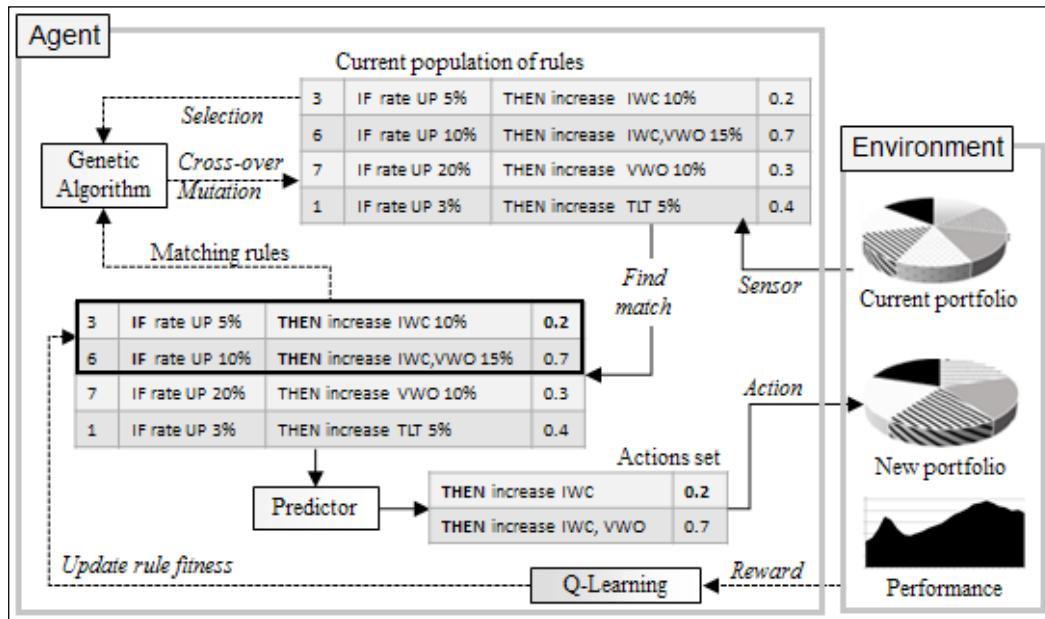
The XCS initial population

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.

The reader is 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 credit or reward to the selected rules
- The genetic algorithm to evolve the population of rules



Overview of XCS algorithm to optimize 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.

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
- **XcsSensor:** This is the sensor or data from the environment

The **XcsAction** 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(val sensorid: String, val target: Double)(implicit val
discr: Discretization) extends Gene(sensorid, target, EQUAL)
```

The `XcsAction` class has the identifier of the sensor, `sensorId`, 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 defines 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, explained in the *Trading operators* section in *Chapter 10, Genetic Algorithms*.

A discretization instance has to be implicitly defined in order to encode the target value.

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 = new 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

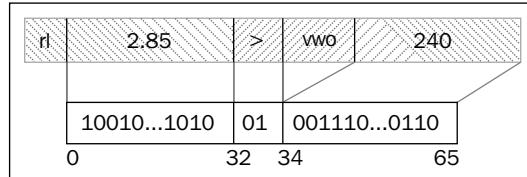
The next step consists of defining a rule as a pair of two genes: a signal and an action, as shown in the following code:

```
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 against the signal in the current population of `s10ytb1` and `s10ytb2` rules that uses the same sensor or variable `10ytb`, as follows:

```
val new10ytb = new XcsSensor("10ytb", 2.76)

val s10ytb1 = Signal("10ytb", 2.5, GREATER_THAN)
val s10ytb2 = Signal("10ytb", 2.2, LESS_THAN)

val r23: XcsRule(s10ytb1, act12)
val r34: XcsRule(s10ytb2, act17)
...
```

In this case, the agent selects the rule `r23` 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 sensor: `r23`, `r11`, and `r46`, as shown in the following code:

```
val r23: XcsRule(s10yTB1, act12)
val r11: XcsRule(s10yTB6, act6)
val r46: XcsRule(s10yTB7, act12)
```

The action with the most references, `act12`, is executed. The Q-learning algorithm computes the reward from the profit or loss incurred by the portfolio following the execution of the selected rules `r23` and `r46`. 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:

```
def cover(sensor: XcsSensor, actions: List[XcsAction]) (implicit
discr: Discretization): List[XcsRule] = {
    actions.foldLeft(List[XcsRule]()) ((xs, act) => {
        val rIdx = Random.nextInt(Signal.numOperators)
        val signal = new Signal(sensor.id, sensor.value, new
SOperator(rIdx))
        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. The mutation is executed by one of the constructors of `XcsAction`, as follows:

```
def apply(action: XcsAction, r: Random): XcsAction =
    (action ^ r.nextInt(XCSACTION_SIZE))
```

The index of the operator type, `rIdx`, is a random value in the interval [0, 3] because a signal uses four types of operators: `None`, `>`, `<`, and `=`.

Example of implementation

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 action

```
class Xcs(config: XcsConfig, population: Population[Signal],  
score: Chromosome[Signal] => Unit, input: Array[QLInput]) extends  
PipeOperator[XcsSensor, List[XcsAction]] {  
  
    val gaSolver = GASolver[Signal](config.gaConfig, score)  
    val featuresSet: Set[Chromosome[Signal]] = population.  
chromosomes.toSet  
    val qLearner = QLearning[Chromosome[Signal]](config.qlConfig,  
computeNumStates(input), extractGoals(input), input, featuresSet)  
    ...  
}
```

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. The goals and number of states are extracted from the input to the policy of the Q-learning algorithm.

In this implementation, the generic algorithm, gaSolver, 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 limitation of learning classifier systems

Learning classifier systems and XCS in particular, hold many promises, which are as follows:

- They allow non-scientists 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:

- Sheer complexity of the configuration of the algorithm because of the large number of parameters for exploration and exploitation.
- Lack of a 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 LCSes depend on each component as well as the interaction between components.
- An execution that is not always predictable in terms of scalability and performance.
- Too many variants of LCS.

Summary

Reinforcement learning algorithms are sometimes overlooked by the software engineering community. 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?

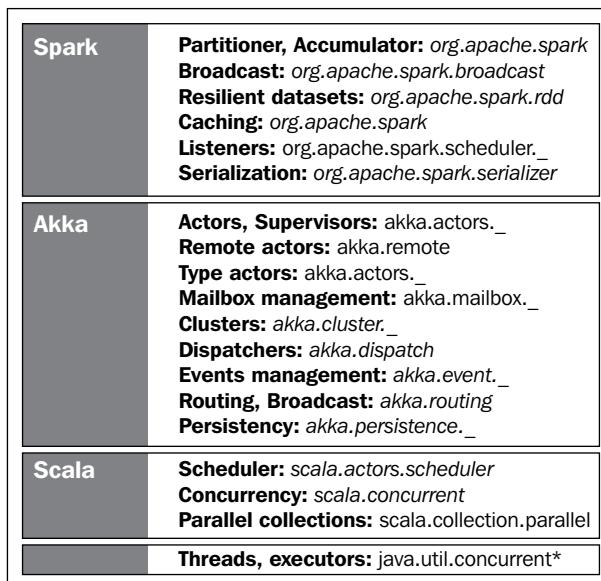
There are many options available to Scala developers to build classification and regression applications for very large datasets. This chapter covers the Scala parallel collections, Actor model, Akka framework, and Apache Spark in-memory clusters. The following are the topics addressed in this chapter:

- Introduction to Scala parallel collections
- Evaluation of performance of a parallel collection on multicore CPU
- The actor model and reactive systems.
- Clustered and reliable distributed computing using Akka
- Design of computational workflow using Akka routers
- 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

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, and load balancing protocol; dispatchers, clusters, events, and configurable mailboxes management; and 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:



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.

Controlling 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 by 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

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:

- 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 collections 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 tasks 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`, which delegates the management of tasks to either a thread pool or the `ForkJoinTasks` pool.
- **ForkJoinTaskSupport**: This uses the fork-join pools of type `java.util.concurrent.ForkJoinPool` introduced in 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 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 parallel vector of random probabilities, `rand`, is created and initialized by the main task (line 1), but the conversion to a vector of exponential value, `randExp`, is executed by a pool of 16 concurrent tasks (line 2).

Preserving 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.

Benchmark framework

Scala library benchmark

The Scala standard library has a trait, `testing.Benchmark`, for testing using the command line [12:2]. All you need to do is to insert your function or code in the `run` method:

```
object test with Benchmark { def run { /* fill
the blank */ }}
```

The main purpose of parallel collections is to improve the performance of execution through concurrency. First, let us create a parameterized class, `Benchmark`, to evaluate the performance of operations on a parallel array, `v`, relative to an array, `u`, as follows:

```
class ParArrayBenchmark[U] (u: Array[U], v: ParArray[U], times:Int)
```

Next, you need to create a method, `timing`, that computes the ratio of the duration of a given operation on a parallel collection over the duration of the same operation on a single threaded collection, as shown here:

```
def timing(g: Int => Unit) : Long = {
  var startTime = System.currentTimeMillis
  Range(0, times).foreach(g)
  System.currentTimeMillis - startTime
}
```

This method measures the time it takes to process a user-defined function, `g`, `times` times.

Let's compare the parallelized and default array on the `map` and `reduce` methods of `Benchmark` 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 //3
  val ratio = timing(_ => v.map(f))/duration //4
  Display.show(s"$nTasks, $ratio", logger)
}
```

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 3) and its parallelized counterpart `v` (line 4). 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  
    val ratio = timing(_ => v.reduceLeft(f)) / duration  
    Display.show(s"$nTasks, $ratio", logger)  
}
```

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 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)
```

For example, the `filter` method of `ParMapBenchmark` evaluates the performance of the parallel map `v` relative to 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  
    Display.show(s"$nTasks, $ratio", logger)  
}
```

Performance evaluation

The first performance test consists of creating a single-threaded and a parallel array of random values and executing the evaluation methods, `map` and `reduce`, on using an increasing number of tasks, as follows:

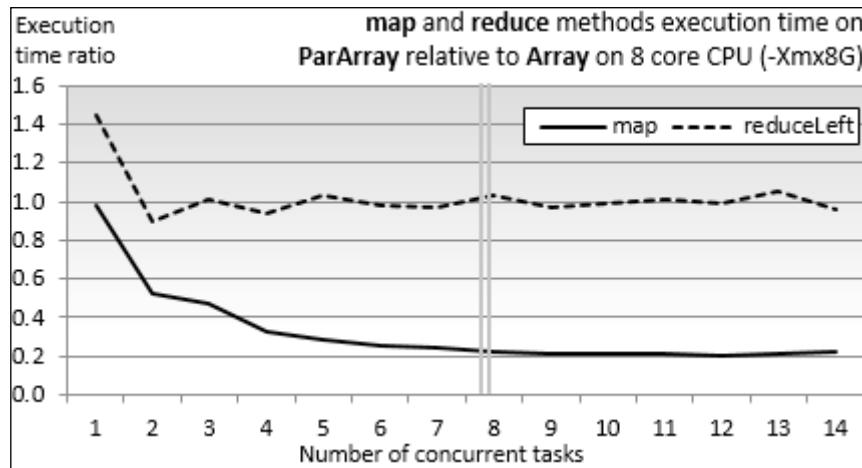
```
val sz = 1000000  
val data = Array.fill(sz)(Random.nextDouble)  
val pData = ParArray.fill(sz)(Random.nextDouble)  
val times: Int = 50
```

```

val bench1 = new ParArrayBenchmark[Double] (data, pData, times)
val mapper = (x: Double) => Math.sin(x*0.01) + Math.exp(-x)
Range(1, 16).foreach(n => bench1.map (mapper) (n))
val reducer = (x: Double, y: Double) => x+y
Range(1, 16).foreach(n => bench1.reduce (reducer) (n))

```

The following graph shows the output of the performance test:



The test executes the mapper and reducer functions 1 million times on an 8-core CPU with 8 GB of available memory on 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 two parallel collections, `ParArray` and `ParHashMap`, on two methods, `map` and `filter`, using a configuration identical to the first test as follows:

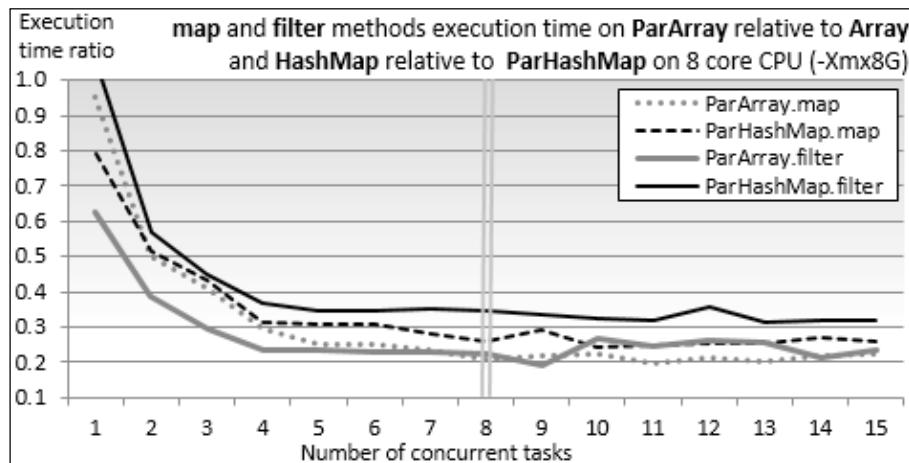
```

val sz = 1000000
val mData = new HashMap[Int, Double]
Range(0, sz).foreach(n => mData.put(n, Random.nextDouble)) //1
val mParData = new ParHashMap[Int, Double]
Range(0, sz).foreach(n => mParData.put(n, Random.nextDouble))

```

```
val bench2 = new ParMapBenchmark[Double] (mData, mParData, times)
Range(1, 16).foreach(n => bench2.map(mapper)(n)) //2
val filterer = (x: Double) => (x > 0.8)
Range(1, 16).foreach(n => bench2.filter(filterer)(n)) //3
```

The test initializes a `HashMap` instance and its parallel counter `ParHashMap` with 1 million random values (line 1). The benchmark, `bench2`, processes all the elements of these hash maps with the `mapper` instance introduced in the first test (line 2) and a filtering function, `filterer` (line 3), with 16 tasks. The output is as shown here:



The impact of the parallelization of collections is very similar across methods and across 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 singe application, 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. That being said, parallel collections are limited to single host deployment. 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 by using a concurrency model that relies on the following key principles:

- Immutable data structures
- Asynchronous communication

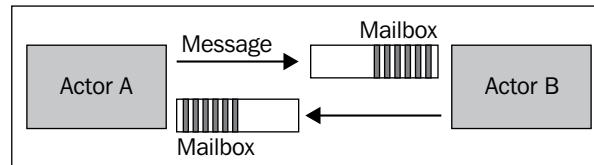
The Actor model

The Actor model, originally introduced in the **Erlang** programming language, addresses these issues [12:3]. The purpose of using the Actor model is twofold:

- 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 Java development

The model consists of the following components:

- Independent processing units known as Actors. Actors 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.



Representation of messaging between actors

There are two message-passing mechanisms:

- **Fire-and-forget or tell:** Sends the immutable message asynchronously to the target or receiving actor, and returns immediately without blocking. The syntax is as follows:

```
targetActorRef ! message
```

- **Send-and-receive or ask:** Sends a message asynchronously, but returns a Future instance that defines the expected reply from the target actor

```
val future = targetActorRef ? message
```

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 type PartialFunction[Any, Unit] [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, than 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 not to use Actors for blocking operations such as I/O, there are circumstances that require the sender to wait for a response. The reader needs to be mindful that blocking an underlying thread inside the Actor might starve other Actors from CPU cycles. It is recommended to either configure the runtime system to use a large thread pool, or to 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 subcollection 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]) (a: 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 www.akka.io website, or through the Typesafe Activator at <http://www.typesafe.com/platform>.

Akka simplifies the implementation of Actor 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, 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 type `PartialFunction[Any, Unit]`
- `postStop`: This is an optional method to clean up resources such as releasing memory, closing database connections, and socket or file handles

Typed versus 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 us create a generic template for a master Actor and worker Actors to transform a dataset using any preprocessing or classification algorithm inherited from the `PipeOperator` trait, as explained in the *The pipe operator* section under *Designing a workflow* in *Chapter 2, Hello World!*. The master Actor manages the worker actors in one of the following ways:

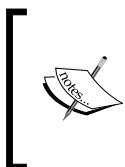
- Individual actors
- Clusters through a **router** or a **dispatcher**

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 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.

Messages exchange

The first step in implementing the master-worker design is to define the different classes of messages exchanged between the master and each worker, to control the execution of the iterative procedure. The implementation of the master-worker design is as follows:

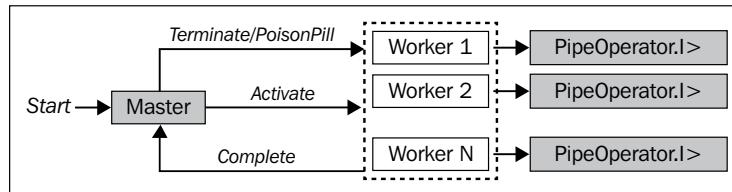
```
type DblSeries = XTSeries[Double]

sealed abstract class Message(val id: Int)
case class Start(i: Int = 0) extends Message(i) //1
case class Activate(i: Int, xt: DblSeries extends Message(i) //2
case class Completed(i: Int, xt: DblSeries) 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:

1. Start is sent by the client code to the master to start the computation.
2. Activate is sent by the master to the workers to activate the computation. This message contains the time series, *xt*, to be processed by the worker Actors.
3. Completed is sent by each worker back to sender. It contains the variance of the data in the group.
4. 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 type inherited from `PipeOperator`. The messages exchanged between master and worker actors are shown in the following diagram:



Messages as case classes



The actor retrieves the messages queued in its mailbox by managing each message instance (copy, 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 partition created by the master Actor, as follows:

```

class Worker(id: Int, fct: PipeOperator[DblSeries, DblSeries]) extends Actor { //1
  override def receive = {
    case msg: Activate => {
      msg.sender ! Completed(msg.id+id, transform(msg.xt)) //2
      context.stop(self)
    }
    case _ => Display.show("Unknown message", logger)
  }
  def transform(xt: DblSeries): DblSeries = fct |>
}
  
```

The `Worker` class constructor takes the `fct` data transformation as an argument (line 1). The worker launches the processing or transformation of the `msg.xt` data upon arrival of the `Activate` message (line 2). It returns the `Completed` message to the master once the data transformation, `transform`, is completed.

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.

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 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 a process
- A data transformation, `fct`, of type `PipeOperator`
- A partitioning method, `partitioner`, to break down a time series for concurrent processing

The `Controller` class can be defined as follows:

```
abstract class Controller(val xt: DblSeries, val fct: PipeOperator[DblSeries, DblSeries], val partitioner: Partitioner)
extends Actor
```

The workflow controller is responsible for splitting the time series into several partitions and assigning each partition to a dedicated worker Actor. A helper class, `Partitioner`, implements the partitioning of the dataset as follows:

```
class Partitioner(val numPartitions: Int) {
  def split(xt: DblSeries): Array[Int] = {
    val sz = (xt.size.toDouble/numPartitions).floor.toInt
    val indices = Array.tabulate(numPartitions)(i=>(i+1)*sz)
    indices.update(numPartitions - 1, xt.size)
    indices
  }
}
```

The `split` method breaks down a time series, `xt`, into `numPartitions` partitions, and returns the index of each partition relative to the original time series.

The master Actor

Let's define a master Actor class, `Master`. The three methods to override are as follows:

- `preStart` is a method invoked to initialize all the necessary resources such as file or database connection before the actor executes
- `receive` is a partial function that dequeues and processes the messages from the mail box
- `postStop` cleans up resources such as releasing memory and closing database connections, sockets, or file handles

The Master class can be defined as follows:

```
abstract class Master(xt: DblSeries, fct: PipeOperator[DblSeries,  
DblSeries], partitioner: Partitioner) extends Controller(xt, fct,  
partitioner) {  
    val workers = List.tabulate(partitioner.numPartitions)(n =>  
        context.actorOf(Props(new Worker(n, fct)))) //4  
    val aggregator = new ListBuffer[DblVector] //5  
  
    override def preStart: Unit = {} //6  
    override def postStop: Unit = {} //7  
    override def receive
```

The Master class has the following parameters:

- xt: This is the time series to transform
- fct: This is the transformation function
- partitioner: This is the instance of time series partitioning

The worker actors are created through the actorOf factory method of the ActorSystem context (line 4). A list buffer, aggregator, collects and reduces the results from each worker (line 5). The preStart method implements any initialization required to process the messages (line 6). The postStop method releases all the resources allocated to process the messages (line 7).

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 Start => split //8  
    case msg: Completed => { //10  
        if(aggregator.size >= partitioner.numPartitions-1) { //12  
            aggregate //14  
            //13 workers.foreach(_ ! PoisonPill)  
            context.stop(self) //15  
        }  
        aggregator.append(msg.xt.toArray) //11  
    }  
}  
  
def aggregate: Seq[Double]  
  
def split: Unit = {  
    val partIdx = partitioner.split(xt)
```

```

workers.zip(partIdx).foreach(w =>
    w._1 ! Activate(0, xt.slice(w._2-partIdx(0), w._2))) //9
}

```

The `Start` message triggers the split of the input time series into partitions (line 8), which are then dispatched to each worker with the `Activate` message (line 9). Each worker sends a `Completed` message back to master upon the completion of their task (line 10). The master aggregates the results from the each worker (line 11). Once every worker has completed its task (line 12), the master terminates all the workers, through a `PoisonPill` message in case the worker actors do not terminate themselves (line 13). The master aggregate the results (line 14) before it terminates itself through a request to its context to stop it (line 15).

The `aggregate` method can be defined as a parameter either of the `Master` class or of one of its subclasses.

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

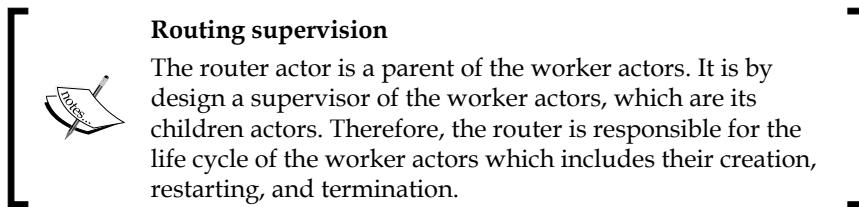
The previous design makes sense only if each worker has a unique characteristic that requires direct communication with the master. This is not the case in most applications. The communication and internal management of the worker can be delegated to a router. The implementation of the master routing capabilities is very similar to the previous design, as shown in the following code:

```

abstract class MasterWithRouter(xt: DblSeries, fct:
  PipeOperator[DblSeries, DblSeries], partitioner: Partitioner) extends
Controller(xt, fct, partitioner) {
  val router = context.actorOf(Props(new Worker(0, fct)))
  .withRouter(RoundRobinPool(partitioner.numPartitions,
    supervisorStrategy = this.supervisorStrategy))
  ...
}

```

The only difference is that the `context.actorOf` factory creates an extra actor, `router`, along with the workers. This particular implementation relies on round-robin assignment of the message by the router to each worker. 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.



The implementation of the `receive` message handler is almost identical to the message handler in the master without routing capabilities, except that the partitioning (line 1) is delegated to the router instead of being applied to each individual worker, as follows:

```
override def receive = {
    case msg: Start => split
    case msg: Completed => {
        if(aggregator.size >= partitioner.numPartitions-1) {
            aggregate
            context.stop(self)      //2
        }
        aggregator.append(msg.xt.toArray)
    }
}
def split: Unit = {
    val indices = partitioner.split(xt)
    indices.foreach(n =>
        router ! Activate(xt.slice(n - indices(0), n))) //1
}
```

The supervising router terminates itself automatically once all its child actors are terminated (line 2).

Distributed discrete Fourier transform

Let's select the **discrete Fourier transform (DFT)** on a time series, `xt`, as our data transformation. We discussed it in the *Discrete Fourier transform (DFT)* 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:

```
class DFTMaster(xt: XTSeries[Double], partitioner: Partitioner)
  extends Master(xt, DFT[Double], partitioner) {
  override def aggregate: Seq[Double] =
    aggregator.transpose.map(_.sum).toSeq
}
```

The `aggregate` method aggregates or reduces the results of the discrete Fourier transform (frequencies distribution) from each worker. In the case of the discrete Fourier transform, the `aggregate` method transposes the list of frequencies distribution then summed the amplitude for each frequency, as shown here:

```
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 //1
val xt = XTSeries[Double](Array.tabulate(NUM_DATAPOINTS)(h(_)))
val partitioner = new Partitioner(NUM_WORKERS) //2

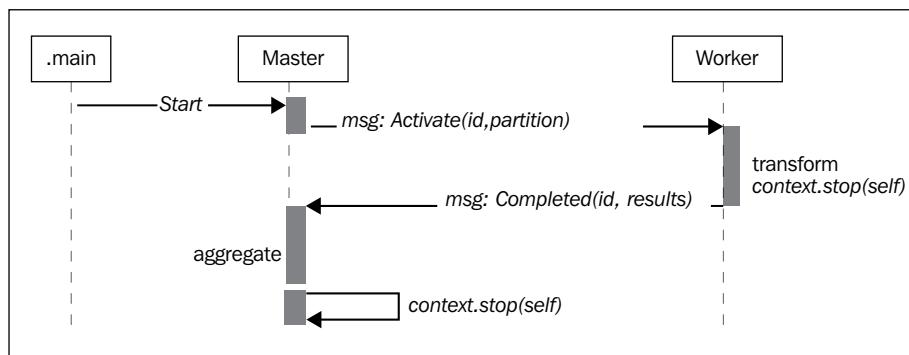
implicit val actorSystem = ActorSystem("system") //3
val master = actorSystem.actorOf(Props(new DFTMaster(xt,
partitioner)), "DFTMaster") //4
master ! Start //5
Thread.sleep(15000)
actorSystem.shutdown //6
```

The input time series is synthetically generated by the noisy function, `h` (line 1). The function `h` has three distinct harmonics, 0.005, 0.05, and 0.2, so the results of the transformation can be easily validated. A `partitioner` instance is created for `NUM_WORKERS` worker Actors (line 2). The Actor system, `ActorSystem`, is instantiated (line 3) and the `master` Actor is generated through the Akka `ActorSystem.actorOf` factory. The main program sends a `Start` message to the master to trigger the distributed computation of the discrete Fourier transform. The main program has to sleep for a period of time long enough to allow the master to complete its task. Finally, the main program shuts down the actor system (line 6).

Actor instantiation

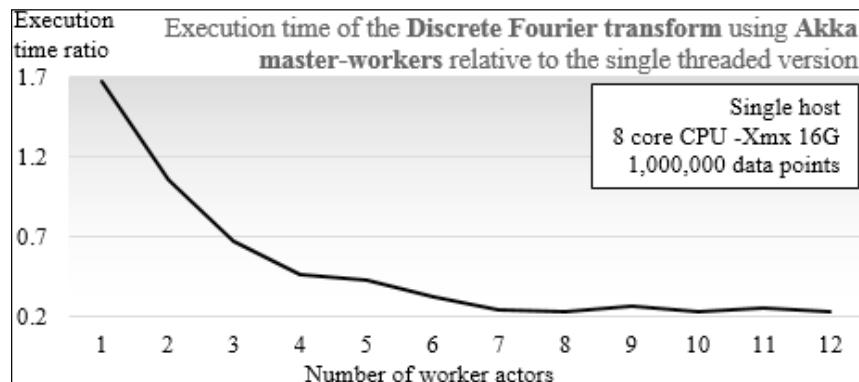
Although the `scala.actor.Actor` class can be instantiated using the constructor, `akka.actor.Actor` is instantiated using a context, `ActorSystem`; a factory, `actorOf`; and a configuration object, `Props`. This second approach has several benefits, including decoupling the deployment of the actor from 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:



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 the Scala parallel collections, the absolute timing for the transformation depends on the host and the configuration, as shown in the following graph:



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 the aggregating (or reducing) the results adds a significant overhead to 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:

- 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 non-blocking 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, successful or not [12:10].

There are two options to retrieve results from futures:

- Blocking execution using `scala.concurrent.Await`
- Callback functions, `onComplete`, `onSuccess`, and `onFailure`

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 type `scala.concurrent.Future` 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}
import akka.util.Timeout
import scala.concurrent.{Await, Future}
```

The Actor classes are provided by the package `akka.actor`, instead of the `scala.actor._` package because of Akka's extended actor model. The future-related classes, `Future` and `Await`, are imported from the `scala.concurrent` package, which is similar to the `java.concurrent` package. The `akka.util.Timeout` class is used to specify the maximum duration the actor has to wait for the completion of the futures.

There are two options for a parent actor or the main program to manage the futures it creates:

- **Blocking:** The parent actor or main program stops execution until all futures have completed their tasks.
- **Callback:** The parent actor or the main program initiates the futures during execution. The future tasks are performed concurrently with the parent actor, that 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 class, `TransformFutures`, that manages futures instead of workers or routing actors, as follows:

```
abstract class TransformFutures(zt: DblSeries,
    fct: PipeOperator[DblSeries, DblSeries],
    partitioner: Partitioner)(implicit timeout: TimeOut)
extends Controller(zt, fct, partitioner) { //1

  override def receive = {
    case Start => compute(transform) //2
    case _ => Display.error("Message not recognized", logger)
  }
  def aggregate(results: Array[DblSeries]): Seq[Double]
  ...
}
```

The `TransformFutures` class requires the same parameters as the `Master` actor: a time series, `xt`; a data transformation, `fct`; and `partitioner`. The `timeout` parameter is an implicit argument of the `Await.result` method, and therefore, needs to be declared as an argument (line 1). The only message, `Start`, triggers the computation of the data transformation of each future, and then the aggregation of the results (line 2). 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 argument. Therefore, no error or exception is thrown in case the message type is not recognized. There is no need for such a handler aside from 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 have 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[DblSeries]] = {
    val partIdx = partitioner.split(xt)
    val partitions = partIdx.map(n =>
        XTSeries[Double](xt.slice(n - partIdx(0), n).toArray)) //3

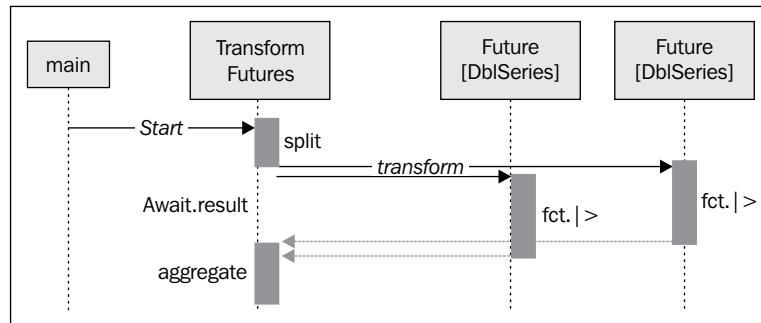
    val futures = new Array[Future[DblSeries]](partIdx.size) //4
    partitions.zipWithIndex.foreach(pi => {
        futures(pi._2) = Future[DblSeries] { fct |> pi._1 }
    })
    futures
}
```

First, the `transform` method splits the input time series into several partitions (line 3), similar to the master Actor in the previous section. An array of futures (one future per partition) is created (line 4). Each future executes the data transformation, `fct`, to the partition assigned to the future (line 5) as the worker Actor did in the previous section.

The `compute` method has the same purpose as the `aggregate` method in the master-workers design. The execution of the Actor is blocked until the `Await` class method (line 6) `scala.concurrent.Await.result` 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 (7), as follows:

```
def compute(futures: Array[Future[DblSeries]]): Seq[Double] = {
    val results = futures.map(Await.result(_, timeout.duration))
    aggregate(results)
}
```

The following sequential diagram illustrates the blocking design and the activities performed by the Actor and the futures:



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:

- `Future.onComplete`
- `Future.onSuccess` and `Future.onFailure`

The `onComplete` callback function takes a function of type `Try[T] => U` as argument with an implicit reference to the execution context, as shown in the following code:

```
val f: Future[T] = future { executeSomeTask }
f onComplete {
  case Success(s) => { ... }
  case Failure(e) => { ... }
}
```

You can surely recognize the {Try, Success, Failure} monad.

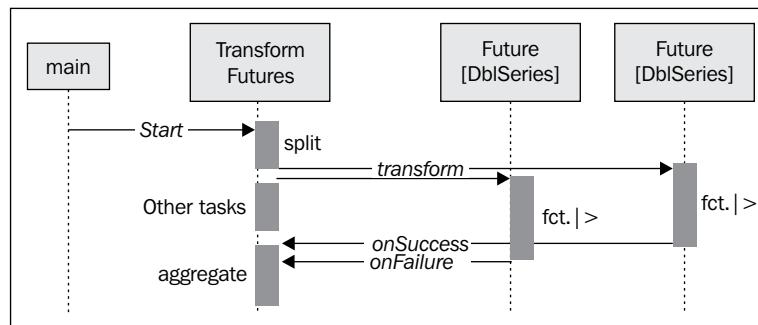
An alternative implementation is to invoke the `onSuccess` and `onFailure` methods that use partial functions as arguments to implement the callbacks, as follows:

```
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[DblSeries]]): Seq[Double] = {
    val aggregation = new ArrayBuffer[DblSeries]
    futures.foreach(f => {
        f onSuccess { //1
            case data: DblSeries => aggregation.append(data)
        }
        f onFailure { //2
            case e: Exception => aggregation.append(XTSeries.empty)
        }
    })
    if( aggregation.find(_.isEmpty) == None) //3
        aggregate(aggregation.toArray)//4
    else Seq.empty
}
```

Each future calls the master Actor back with either the result of the data transformation, the `onSuccess` message (line 1), or an exception, the `OnFailure` message (line 2). If every future succeeds (line 3), the values of every frequency for all the partitions are summed (line 4). The following sequential diagram illustrates the handling of the callback in the master Actor:



Sequential diagram for actor handling future result with Callbacks

Execution context

The Futures method requires that the execution context be 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 all together

Let's reuse the discrete Fourier transform. The client code uses the same synthetically created time series as with 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: DblSeries, partitioner: Partitioner)
  (implicit timeout: Timeout)
  extends TransformFutures(xt, DFT[Double], partitioner) {

  override def aggregate(xt: Array[DblSeries]): Seq[Double] =
    xt.map(_.toArray).transpose.map(_.sum).toSeq
}
```

The only purpose of the `DFTTransformFuture` class is to define the aggregation method, `aggregate`, for the discrete Fourier transform, as follows:

```
import akka.pattern.ask
val duration = Duration(10000, "millis")
implicit val timeout = new Timeout(duration)
implicit val actorSystem = ActorSystem("system")

val xt = XTSeries[Double](Array.tabulate(NUM_DATAPOINTS)(h(_)))
val partitioner = new Partitioner(NUM_WORKERS)

val master = actorSystem.actorOf(Props(new DFTTransformFutures(xt,
partitioner)), "DFTTransform") //1
```

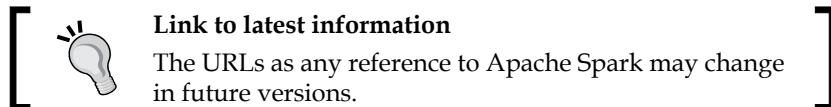
```
val future = master ? Start //2
Await.result(future, timeout.duration) //3
actorSystem.shutdown //4
```

The master Actor is initialized as of the `TransformFutures` type with the input time series, `xt`; discrete Fourier transform, `DFT`; and `partitioner` as arguments (line 1). The program creates a future instance, by sending (ask) the `Start` message to `master`. The program blocks until the completion of the future (line 3), and then shuts down the Akka actor system (line 4).

Apache Spark

Apache Spark is a fast and general-purpose cluster computing system, initially developed as **AMPLab / UC Berkley** 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, 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 core element of Spark is **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 list, array, or hash table, from the local file system 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 RDD can be classified as follows:

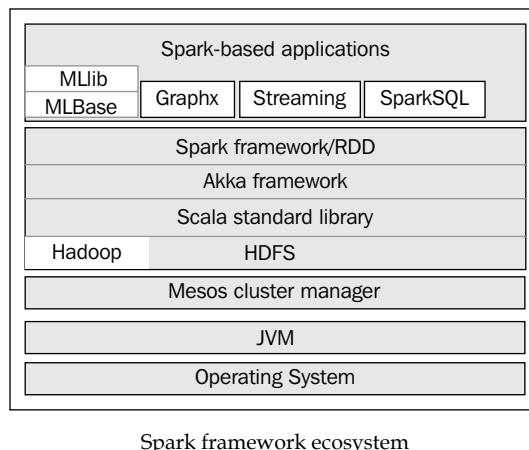
- **Transformation:** This operation converts, manipulates, and filters the elements of an RDD on each partition
- **Action:** This operation aggregates, collects, or reduces the elements of the RDD from all partitions

An RDD can persist, be serialized, and be 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 files system
- Mesos for management of 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 Spark ecosystem has grown to support some machine-learning algorithms out of the box, **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. Those 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 filesystems. Persistency 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 HDFS.

Design principles

The performance of Spark relies on four core design principles [12:13]:

- In-memory persistency
- Laziness in scheduling tasks
- Transform and actions applied to RDDs
- Implementation of shared variables

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 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`. At a later stage, the `g` method is called, which in turn invokes the `f` method. The `f` method initializes the value `m` it needs, then `g` initializes `x` to compute its power to `m<<1`.

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 backwards is performed by the direct acyclic graph scheduler.

Transforms and Actions

Spark is implemented in Scala, so you should not be too surprised 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>	Transforms an RDD by executing the <code>f</code> function on each element of the collection.
<code>filter(f)</code>	<code>filter(f)</code>	Transforms an RDD by selecting the element for which the <code>f</code> function returns true.
<code>flatMap(f)</code>	<code>flatMap(f)</code>	Transforms an RDD by mapping each element to a sequence of output items.
<code>mapPartitions(f)</code>		Executes the <code>map</code> method separately on each partition.
<code>sample</code>		Samples a fraction of the data with or without a replacement using a random generator.
<code>groupByKey</code>	<code>groupBy</code>	Called on <code>(K, V)</code> to generate a new <code>(K, Seq(V))</code> RDD.
<code>union</code>	<code>union</code>	Creates a new RDD as union of this RDD and the argument.
<code>distinct</code>	<code>distinct</code>	Eliminates duplicate elements from this RDD.
<code>reduceByKey(f)</code>	<code>reduce</code>	Aggregates or reduces the value corresponding to each key using the <code>f</code> function.

Spark	Scala	Description
sortByKey	sortWith	Reorganizes (K, V) in an RDD by the ascending, descending, or otherwise specified order of the keys, K .
join		Joins an RDD (K, V) with an RDD (K, W) to generate a new RDD $(K, (V, W))$.
coGroup		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)	Aggregates all the elements of the RDD across all the partitions and returns a Scala object to the driver.
collect	collect	Collects and returns all the elements of the RDD across all the partitions as a list in the driver.
count	count	Returns the number of elements in the RDD to the driver.
first	head	Returns the first element of the RDD to the driver.
take(n)	take(n)	Returns the first n elements of the RDD to the driver .
takeSample		Returns an array of random elements from the RDD back to the driver.
saveAsTextFile		Writes the elements of the RDD as a text file in either the local files system or HDFS.
countByKey		Generates a (K, Int) RDD with the original keys, K , and the count of values for each key.
foreach	foreach	Executes a $T \Rightarrow Unit$ function on each elements of the RDD.

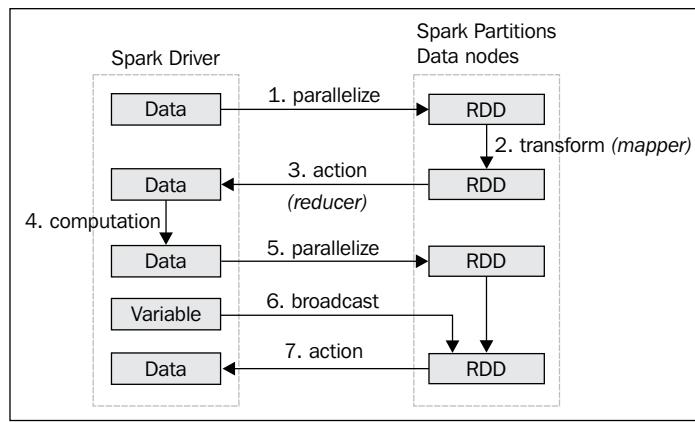
Scala methods such as `fold`, `find`, `drop`, `flatten`, `min`, `max`, and `sum` are not currently implemented in Spark. Other Scala methods such as `zip` have to be used carefully, as there is no guarantee that the order of the two collections in `zip` is maintained between partitions.

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:



Interaction between 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:

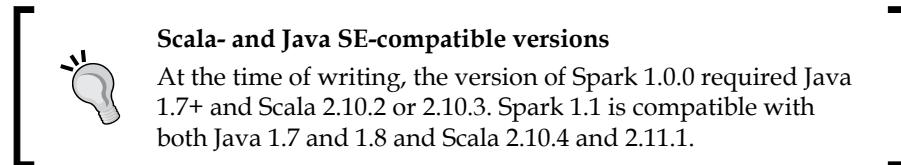
1. The input data, residing in either memory as a Scala collection or HDFS as a text file, is parallelized and partitioned into an RDD
2. A transformation function is applied on each element of the dataset across all the partitions
3. An action is performed to reduce and collect the data back to the 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 look closely, the management of datasets and RDDs by the Spark driver is not very different from that by 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 1.1.



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 HDFS to run in cluster mode; therefore, you need to select a distribution of Hadoop, or an open source distribution, 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 <https://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 -DskipTests clean package
```

- **Simple build tool:** Use the following command:

```
sbt/sbt assembly
```

Installation instructions



The directory and name of artifacts used in Spark will undoubtedly change over time. Please refer to the documentation and installation guide for the latest version of Spark.

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 (RDD). 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 with 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 main method, `myApp.main`, on a 4-core CPU local host, and 12 GB of memory.

- To launch the same Spark application remotely, connect with the shell and 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
```

```
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
14/10/13 14:12:36 INFO SecurityManager: SecurityManager: authentication disabled;
14/10/13 14:12:36 INFO HttpServer: Starting HTTP Server
14/10/13 14:12:36 INFO Utils: Successfully started service 'HTTP class server'
Welcome to


$$\sqrt{X} = \sqrt{\frac{X}{Y}} \cdot \sqrt{Y}$$
 version 1.1.0

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_45)
Type in expressions to have them evaluated.
Type :help for more information.

14/10/13 14:12:40 INFO SparkILoop: Created spark context..
Spark context available as sc.
```

Partial screenshot of Spark shell



Potential pitfalls with 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 environment variable 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 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: XTSeries[DblVector], rddConfig: RDDConfig)(implicit
  sc: SparkContext): RDD[DblVector] = {
  val rdd: = sc.parallelize(xt.toArray
    .map(new DenseVector(_))) //1
  rdd.persist(rddConfig.persist) //2
  if( rddConfig.cache) rdd.cache //3
  rdd
}
```

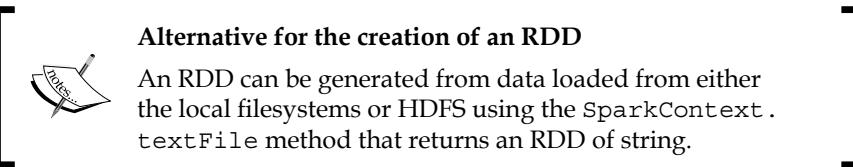
The last parameter, `rddConfig`, specifies the configuration for the RDD. In this example, the configuration of the RDD consists of enabling/disabling cache and selecting the persistency model, as follows:

```
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 by using the `parallelize` method of the context and converting into a vector (`SparseVector` or `DenseVector`) (line 1)
2. Specify the persistency model or the storage level if the default level needs to be overridden for the RDD (line 2)
3. Specify whether the RDD has to persist in memory (line 3)



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, maxIter: Int, numRuns: Int = 1) {  
    val kmeans: KMeans = {  
        val kmeans = new KMeans  
        kmeans.setK(k) //4  
        kmeans.setMaxIterations(maxIter) //5  
        kmeans.setRuns(numRuns) //6  
        kmeans  
    }  
}
```

The minimum set of initialization parameters for MLlib K-means algorithm is as follows:

- Number of clusters, `K` (line 4)
- Maximum number iterations for the reconstruction of the total error, `maxIter` (line 5)
- The number of training runs, `numRuns` (line 6)

The `SparkKMeans` class wraps the Spark `KMeans` into a data transformation of type `PipeOperator` so that it can be used in a computation workflow. The class follows the design template for classifier as explained in the *Design template for classifiers* section in *Appendix A, Basic Concepts*.

```
class SparkKMeans(config: SparkKMeansConfig, rddConfig: RDDConfig, xt: XTSeries[DblVector])(implicit sc: SparkContext)
  extends PipeOperator[DblVector, Int] {
  val model = config.kmeans.run(RDDSource.convert(xt, rddConfig))
  ...
}
```

The constructor takes three arguments: the Apache Spark `KMeans` configuration, `config`; the RDD configuration, `rddConfig`; and the input time series to clustering, `xt`. The generation of `model` merely consists of converting the time series `xt` into an RDD using `rddConfig` and invoking MLlib `KMeans`.`run`. Once created, the clusters (`KMeansModel`) are available for predicting new observation, `obs`, as follows:

```
def |> : PartialFunction[DblVector, Int] = {
  case x: DblVector if(x != null && x.size>0 && model != null) =>
    model.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 trading volume of each trading session, and the volatility of the price of the stock during the session:

```
val K = 8; val MAXITERS = 100; val NRUNS = 16
val PATH = "resources/data/chap12/CSCO.csv"
val CACHE = true
val extractors = List[Array[String] => Double](
  YahooFinancials.volatility, YahooFinancials.volume) //7
)
val input = DataSource(PATH, true) |> extractors //8

val volatilityVol = input(0).zip(input(1)) //9
```

```
.map(x => Array[Double](x._1, x._2))

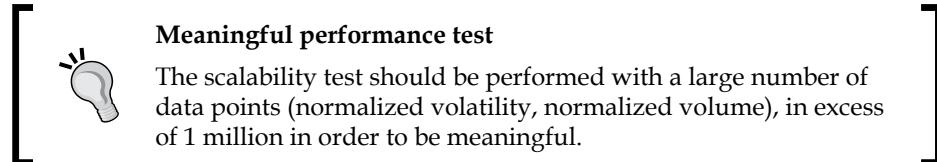
implicit val sc = new SparkContext("Local", "SparkKMeans") //10
val config = new SparkKMeansConfig(K, MAXITERS, NRUNS)
val rddConfig = RDDConfig(CACHE, StorageLevel.MEMORY_ONLY)
val xt = XTSeries[DblVector](volatilityVol)

val sparkKMeans = SparkKMeans(config, rddConfig, xt) //11
val obs = Array[Double](0.23, 0.67)
val clusterId = sparkKMeans |> obs//12
Display.show(s"cluster = $clusterId", logger)
```

The first step is to define the variable to be extracted from the CSV file (line 7). The spark context is created (line 10) once the volatility and volume are extracted (line 8) and zipped (line 9). The K-means wrapper, sparkKMeans, is initialized (line 11). The final step consists of correctly predicting the cluster for a new observation (line 12).

Performance evaluation

Let's execute the normalization of the cross-validation group on an 8-core CPU machine with 32 GB of RAM. The data is partitioned with a ratio of two partitions per CPU core.



The actual values of the data points have no bearing on the overall performance of the Spark cluster.

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:

- 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 as 60 percent of the maximum JVM heap is a generally a good compromise.
- Number of concurrent tasks to use across all the partitions for shuffle-related operations, they use key such as `reduceByKey: spark.default.parallelism`. The recommended formula is $parallelism = \text{total number of cores} \times 2$. The value of the parameter can be overridden with the `spark.reduceby.partitions` parameter for specific RDD reducers.
- Flag to compress serialized RDD partition for `MEMORY_ONLY_SER: spark.rdd.compress`. The purpose is to reduce memory footprints at the cost of extra CPU cycles.
- Maximum size of message containing the results of an action sent to the `spark.akka.frameSize` driver. This value has to be increased if a collection may potentially generate a large size array.
- Flag to compress large size broadcasted `spark.broadcast.compress` variables. It is usually recommended.

Tests

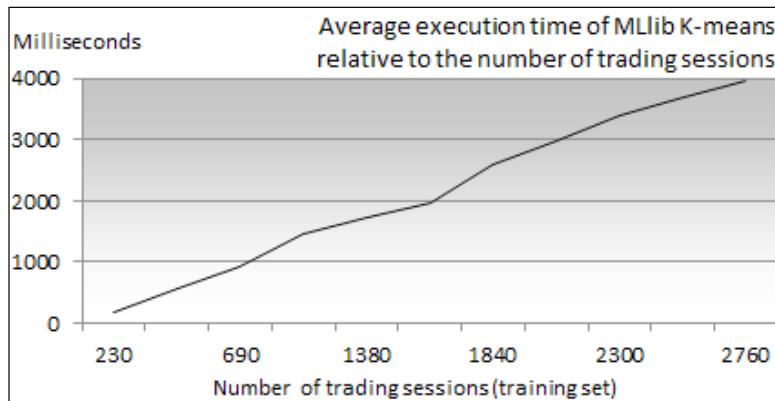
The purpose of the test is to evaluate how the execution time is related to the size of the training set. The test executes K-means from MLlib library on the volatility and trading session volume on **Bank of America (BAC)** stock over the following periods: 3 months, 6 months, 12 months, 24 months, 48 months, 60 month, 72 month, 96 months, and 120 months.

The following configuration is used to perform the training of the 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 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:



Average duration of K-means clustering versus size of trading 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.

Evaluation in distributed environment



A Spark deployment on multiple hosts would add latency of the TCP communication to the overall execution time. 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 1.1:

- 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 key.

- 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.
- Relies on shared or broadcast variables whenever necessary. Broadcast variables, for instance, improve the performance of operations on multiple datasets with very different sizes. Let us 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:

- 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 cluster manager, which reduces the complexity of defining fault-tolerance and load balancing between worker nodes
- Spark is 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:

- 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.

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, 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 concepts 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.

This appendix lists a set of basic elements of linear algebra and optimization mentioned throughout the book. It also summarizes the coding practices that have been covered, and acquaints the reader with basic knowledge of financial analysis.

Scala programming

The following is a partial list of coding practices and design techniques used throughout the book.

List of libraries

The `libraries` directory contains the JAR files related to the third-party libraries or frameworks used in this book. Not all libraries are needed for every chapter. The list is as follows:

- Apache Commons Math 3.3 in *Chapter 3, Data Preprocessing; Chapter 4, Unsupervised Learning*; and *Chapter 6, Regression and Regularization*
- JFChart 1.0.1 in *Chapter 1, Getting Started; Chapter 2, Hello World!; Chapter 5, Naïve Bayes Classifiers*; and *Chapter 9, Artificial Neural Networks*
- Iitb CRF 0.2 (including L-BFGS and Colt libraries) in *Chapter 7, Sequential Data Models*
- LIBSVM 0.1.6 in *Chapter 8, Kernel Models and Support Vector Machines*
- Akka framework 2.2.4 in *Chapter 12, Scalable Frameworks*
- Apache Spark/MLlib 1.1 in *Chapter 12, Scalable Frameworks*



Note for Spark developers

The Scala library and compiler JAR files bundled with the assembly JAR file of Apache Spark contain a version of the Scala standard library and compiler JAR file that may conflict with an existing Scala library (for example, Eclipse default ScalaIDE library).

Format of code snippets

For the sake of readability of the implementation of algorithms, all non-essential pieces of code such as error checking, comments, exceptions, or imports have been omitted. *The following code elements have been discarded in the code snippets presented in the book:*

- Comments:

```
// 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, private, and so on:

```
final protected class MLP[T <% Double] ...
```

- Method qualifiers and access controls (final, private, and so on):

```
final def inputLayer: MLPLayer  
private def recurse: Unit =
```

- 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) => ...  
}
```

- Non-essential annotations:

```
@inline def mean = { ... }
```

- Logging and debugging code:
`m_logger.debug(...)
Console.println(...)`
- Auxiliary methods not essential to the understanding of an algorithm

Encapsulation

One important objective while creating an API is reducing access to supporting or helper classes. There are two options to encapsulate helper classes, as follows:

- **Package scope:** In this, the supporting classes are first-level classes with protected access
- **Class or object scope:** In this, 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 package scope (`protected`):

```
protected class MyClass[T] (val x: X, val y: Y,...) { ... }  
object MyClass {  
    def apply[T] (x: X, y:Y, ...): MyClass[T] = new MyClass(x,y,...)  
    final val y0 = ...  
    def apply[T] (x: , ...): MyClass[T] = new MyClass(x, y0, ...)  
}
```

For example, the configuration of the support vector machine classifier is defined as follows:

```
protected class SVMConfig(val formulation: SVMFormulation, val kernel: SVMKernel, val svmExec: SVMExecution) extends Config
```

Its constructors are defined as follows:

```
object SVMConfig {  
    val DEFAULT_CACHE = 25000  
    val DEFAULT_EPS = 1e-15  
    ...  
    def apply(svmType: SVMFormulation, kernel: SVMKernel, svmExec: SVMExecution): SVMConfig = new SVMConfig(svmType, kernel, svmExec)
```

```
def apply(svmType: SVMFormulation, kernel: SVMKernel): SVMConfig  
= new SVMConfig(svmType, kernel, new SVMExecution(DEFAULT_CACHE,  
DEFAULT_EPS, -1))  
{}
```

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, that is, objects with no methods.

Enumerations versus case classes

It is not uncommon 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 heavyweight enumeration values.

Let's take an example of a Scala enumeration that consists of evaluating the uniform distribution of `scala.util.Random`:

```
object MyEnum extends Enumeration {  
    type TMyEnum = Value  
    val A, B, C = Value  
}  
  
import MyEnum._  
val counters = Array.fill(MyEnum.maxId+1)(0)  
Range(0, 1000).foreach(_ => Random.nextInt(10) match {  
    case 3 => counters(A.id) += 1  
    ...  
    case _ => {}  
})
```

The previous pattern matching is very similar to the `switch` statement of Java.

Let's consider the following example of pattern matching using case classes that selects a mathematical formula according to the input:

```
package MyPackage {  
    sealed abstract class MyEnum(val level: Int)  
    case class A extends MyEnum(3) { def f = (x:Double) => 23*x}  
    ...  
}  
  
import MyPackage._  
def compute(myEnum: MyEnum, x: Double): Double = myEnum match {  
    case a: A => a.f(x)  
    ...  
}
```

The previous 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 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 a nutshell, 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 meaning of the operators used in code snippets:

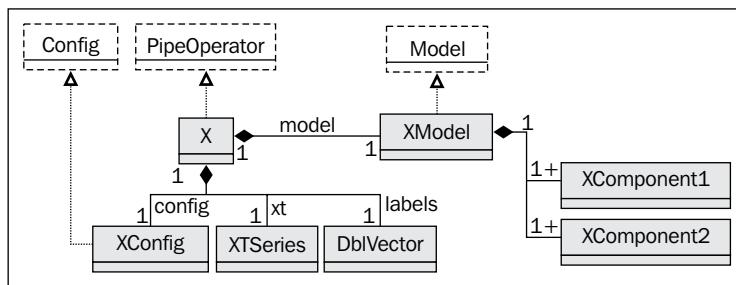
- `+=`: This adds an element to a collection or container.
- `+`: This sums two elements of the same type.
- `|>`: This transforms a collection of data. It is also known as pipe operator. The type of output collections and elements can be different from that of the input.

Design template for classifiers

The machine learning algorithms described in this book use the following design pattern:

- A model instance that implements the `Model` trait is created through training during the initialization of the classifier
- All configuration parameters are encapsulated into a single configuration class inheriting the `Config` trait
- The predictive or classification routine is implemented as a data transformation extending the `PipeOperator` trait
- The classifier takes at least three parameters: configuration instance, a features set or time series, and a labeled dataset

Have a look at the following diagram:



A generic UML class diagram for classifiers

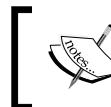
For example, the key components of the support vector machine package are as follows:

```
final protected class SVM[T <% Double](val config: SVMConfig, val xt: XTSeries[Array[T]], val labels: DblVector)
    extends PipeOperator[Array[T], Double] {
    val model: Option[SVMModel] = { ... }
    override def |> (x: Feature): Option[Double] = { prediction }
    ...
}

final protected class SVMConfig(val formulation: SVMFormulation, val kernel: SVMKernel, val svmExec: SVMExecution) extends Config
protected class SVMModel(val params: (svm_model, Double)) extends Model
```

The two data inputs required to train a model are the configuration of the classifier (`config`) and the training set (`xt` and `labels`). Once trained and validated, the model is available for prediction or classification.

This design has the main advantage of reducing the life cycle of the classifier; a model is either defined, available for classification, or is not created.



Implementation considerations

The validation phase is omitted in most of the practical examples throughout this book for the sake of readability.



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 this book:

```
type Fields = Array[String]
class DataSource(pathName: String,
                 normalize: Boolean,
                 reverseOrder: Boolean,
                 headerLines: Int = 1,
                 srcFilter: Option[Fields=>Boolean])
  extends PipeOperator[List[Fields =>Double], List[DblVector]]
```

The parameters for the `DataSource` 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 the files are CSV files.
- `normalize`: This is a flag to specify if the data has to be normalized over $[0, 1]$.
- `reverseOrder`: This is a flag to specify whether the order of the data in the file has to be reversed—for example, time series—if its value is `true`.
- `headerLines`: This specifies the number of lines for column headers and comments.
- `srcFilter`: This is a filter or condition for some of the row fields to skip the data set, for example, missing data or incorrect format.

The most important method of `DataSource` is the following data transformation from a file to a typed time series (`XTSeries[T]`) implemented as the pipe operator method. The method takes the extractor from a row of literal values to `Double` floating-point values:

```
def |> : PartialFunction[List[Fields=>Double], List[DblVector]] ={
  case extr: List[Fields=>Double] if(extr!=null && extr.size>0) =>
    load match { //1
      case Some(data) => {
        if( normalize) // 2
          extr.map(t=>Stats[Double](data._2.map(t(_)))) //3
            .normalize() //4
        else extr.map(t => data._2.map(t(_)))
      }
      ...
    }
}
```

The data is loaded from the file and converted into a list of vectors using the extractor, `extr` (line 1). The data is normalized if required (line 2) by converting each literal to a floating point value and a `Stats` object is created (line 3). Finally, the `Stats` instance normalizes the sequence of floating-point values (line 4).

A second data transformation consists of transforming a single literal per row to create a time series of single variables:

```
def |> (extr: Fields => Double): Option[XTSeries[Double]]
```

Data sources

The examples in this book rely on three different sources of financial data using CSV format:

- `YahooFinancials` for Yahoo schema for historical stock and ETF price
- `GoogleFinancials` for Google schema for historical stock and ETF price
- `Fundamentals` for fundamental financial analysis ratio (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: Fields) => s(ADJ_CLOSE.id).toDouble)
  ...
}
```

```
deftoDouble(v: Value): Fields => Double =
  (s: Fields) => s(v.id).toDouble

def vol: Fields => Double = (s: Fields) => {
  s(HIGH.id).toDouble/s(LOW.id).toDouble -1.0) * s(VOLUME.id).
  toDouble)
}
...
}
```

Let's look at an example of application of a `DataSource` transformation: loading historical stock data from the Google finance website. The data is downloaded as a CSV-formatted file. The first step is to specify the column name using an enumeration singleton, `YahooFinancials`:

```
object GoogleFinancials extends Enumeration {
  type GoogleFinancials = Value
  val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME = Value
  val close = ((s: Fields) => s(CLOSE.id).toDouble)//5
  ...
}
```

Each column is associated with an extractor function (line 5). Consider the following code:

```
val symbols = Array[String] ("CSCO", ...) //6
val prices = symbols
  .map(s => DataSource(path+s+".csv",true,true,1))//7
  .map(_ |> YahooFinancials.close) //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 (for example, CSCO is associated with `resources/CSCO.csv`) (line 7). Finally, the `YahooFinancials` extractor for the close price is invoked (line 8).

Extraction of documents

The `DocumentsSource` class is responsible for extracting the date, title, and content of a list of text documents or text files. This class does not support HTML documents:

```
class DocumentsSource(val pathName: String)
```

The extraction of terms is performed by the data transformation |>, as follows:

```
def |> : Corpus = {
    filesList.map( fName => {
        val src = Source.fromFile(pathName + fName) //1
        val fieldIter = src.getLines //2

        val date = nextField(fieldIter)
        val title = nextField (fieldIter)
        val content = fieldIter.foldLeft(new StringBuilder)((b, str)
            => b.append(str.trim)) //3
        src.close //4
        if(date == None || title == None)
            throw new IllegalStateException( ... ) //6
        (date.get, title.get, content.toString) //5
    })
}
```

This method loads the text files for each filename in the list, `filesList` (line 1). It gets a reference to the document lines iterator, `fieldIter` (line 2). The iterator is used to extract (line 3) and return the tuple (document date, document title, document content) (line 5) once the file handle is closed (line 4). An `IllegalStateException` is thrown and caught if the text file is malformed. The `nextField` method moves the iterator forward to the next non-null line:

```
def nextField(iter: Iterator[String]): Option[String] =
    iter.find(s=> (s != null && s.length > 1)
```

Matrix class

Some discriminative learning models require operations performed on rows and columns of the matrix. The parameterized `Matrix` class facilitates the read/write operations on columns and rows:

```
class Matrix[@specialized(Double, Int) T: ClassTag](val nRows: Int,
    val nCols: Int, val data:Array[T])(implicit f: T => Double){
    def apply(i: Int, j: Int): T = data(i*nCols+j)
    def cols(i: Int): Array[T] = {
        (i until data.size by nCols)
            .map(data(_)).toArray
    }
    ...
    def += (i: Int, j : Int, t: T): Unit = data(i*nCols +j) = t
    def += (iRow: Int, t: T): Unit = {
        val i = iRow*nCols
```

```
    Range(0, nCols).foreach(k => data(i + k) =t)
  }
def /= (iRow: Int, t: T)(implicit g: Double => T): Unit = {
  val i = iRow*nCols
  Range(0, nCols).foreach(k => data(i + k) /= t)
}
}
```

The `apply` method returns an element. Similarly, the `cols` method returns a column. The write methods consist of updating an element or a column of elements (`+=`) with a value and dividing the elements of a column by a value (`/=`). The matrix is specialized with the type `Double` in order to generate a dedicated byte code for this type.

The generation of the transpose matrix is performed by the `transpose` method. It is an alternative to the Scala methods `Array.transpose` and `List.transpose`:

```
def transpose: Matrix[T] = {
  val m = Matrix[T](nCols, nRows)
  Range(0, nRows).foreach(i => {
    val col = i*nCols
    Range(0, nCols).foreach(j => m += (j, i, data(col+j)))
  })
  m
}
```

The constructors of the `Matrix` class are defined by its companion object:

```
def apply[T: ClassTag](nR: Int, nC: Int, data: Array[T])
  (implicit f: T => Double): Matrix[T] =
  new Matrix(nRows, nCols, data)
```

Mathematics

This section describes very briefly 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 involve 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

QR decomposition (or 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 vectors of matrix $A = [Q_1 \ Q_2].[R_1 \ R_2]^T$, 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 triangular matrix, and R_2 is an m by n null matrix.

QR decomposition is a reliable method of solving a large system of linear equations in 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 $O(mn^2-n^3/3)$.

It is used to minimize the loss function for ordinary least squares regression (refer to the *Ordinary least squares (OLS) regression* section of *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 non-singular matrix and x and b are two vectors. The technique consists of decomposing the original matrix A as the product of simple matrices $A = A_1A_2\dots A_n$. It is of two types as follows:

- **Basic LU factorization:** This defines A as the product of a unit lower triangular matrix L and a upper triangular matrix U . So, $A = LU$.
- **LU factorization with pivot:** This defines A as the product of a permutation matrix P , a unit lower triangular matrix L , and an upper triangular matrix U . So, $A = PLU$.

LDL decomposition

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 Cholesky decomposition of real matrices is a special case of 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. Cholesky factorization is used in the linear least squares Kalman filter (refer to the *The recursive algorithm* section of *Chapter 3, Data Preprocessing*).

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 square real unitary matrix U , an m by n rectangular diagonal matrix Σ , and the transpose V^T matrix of a real matrix. So, $A=U\Sigma V^T$.

The columns of the matrices U and V 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 $O(mn^2-n^3)$. 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 as $Ax = \lambda x$.

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 to the eigenvectors [A:5], [A:6]. Eigen decomposition is used in Principal Components Analysis (refer to the *Principal components analysis (PCA)* section of *Chapter 4, Unsupervised Learning*).

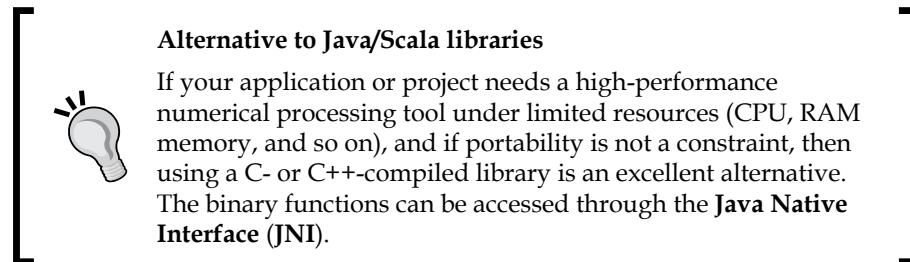
Algebraic and numerical libraries

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*, and *Chapter 6, Regression and Regularization*, and Apache Spark/MLlib used 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**. It is available at <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. It is available at <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 monoids and monads. This library is an excellent example of high-level functional programming using Scala. It is available at <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, and it is available at <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.



First order predicate logic

Propositional logic is the formulation of axioms or propositions. There are several formal representations of propositions:

- Noun-VERB-Adjective: "Variance of the stock price EXCEEDS 0.76" or "Minimization of the loss function DOES NOT converge"
- Entity-value = Boolean: " Variance of the stock price GREATER+THAN 0.76 = true" or "Minimization of the loss function converge = false"
- Variable *op* value: "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 three Boolean operators NOT, AND, OR. So the following rules apply:

- NOT (NOT *P*) = *P*
- *P* AND false = false, *P* AND true = *P*, *P* OR false = *P*, *P* OR true = *P*
- *P* AND *Q* = *Q* AND *P*, *P* OR *Q* = *Q* OR *P*
- *P* AND (*Q* AND *R*) = (*P* AND *Q*) AND *R*

First-order predicate logic, also known as first-order predicate calculus, is the quantification of propositional logic [A:7]. The most common formulations of the first order logic are as follows:

- IF P THEN action rules
- Existential operators

First order logic is used to describe the classifiers in learning classifier systems. Refer to the XCS *rules* section of *Chapter 11, Reinforcement Learning* for more information.

Jacobian and Hessian matrices

Let's consider a function with n variables x_i and m outputs y_j such that $f: \{x_i\} \rightarrow \{y_j = f(x)\}$.

The **Jacobian matrix** [A:8] is the matrix of the first order partial derivatives of the output values of a continuous, differential 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 of partial derivatives of a continuous, 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:

$$f(x, y) = x^2 y + e^{-y} \quad J(f) = \begin{bmatrix} 2xy, x^2 - e^{-y} \end{bmatrix} \quad H(f) = \begin{bmatrix} 2y & 2x \\ 2x & e^{-y} \end{bmatrix}$$

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 data sets.

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 of any defined, differentiable, and convex function [A:9]. The value of a vector or data point x_{t+1} at the 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 for solving systems of non-linear equations and minimization of the loss function in the logistic regression (refer to the *Numerical optimization* section of *Chapter 6, Regression and Regularization*), in support vector classifiers (refer to the *The nonlinear SVM* section of *Chapter 8, Kernel Models and Support Vector Machines*), and in multilayer perceptrons (refer to the *The multilayer perceptron (MLP)* section of *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^* to 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 method that minimizes the convex function by defining the objective function F as the sum of differentiable, basis functions 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 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 descent or quasi-Newton methods in converging towards a solution for convex functions.

The stochastic gradient descent method is used in logistic regression, support vector machines, and back-propagation neural networks.

Stochastic gradient is particularly suitable for discriminative models with large datasets [A:11]. Spark/MLlib makes extensive use of the stochastic gradient method.

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 whose first order derivative is null [A:12].

Newton's method is a well-known and simple optimization method used to find the solution to equations $F(x) = 0$ for which F is continuous and differentiable up to the second order. It relies on the Taylor series expansion to approximate the function F with a quadratic approximation on the 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)** method is a quasi-Newton iterative numerical method to solve unconstrained nonlinear problems. The Hessian matrix H_{t+1} at an iteration $t+1$ 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 method is used in minimization of the cost function for the conditional random field, and L₁ and L₂ regression.

L-BFGS

The performance of the BFGS algorithm can be improved by caching the intermediate computation in memory in order to approximate the Hessian matrix. The obvious drawback is that the memory becomes a limiting factor in the scalability of the optimizer.

The **Limited memory Broyden-Fletcher-Goldfarb-Shanno** algorithm or **L-BFGS** 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 the gradient ΔG_t at iteration t , and then computes those values for the next step $t+1$:

$$x_{t+1} = x_t + \Delta x_t; \quad \nabla F(x_{t+1}) = \nabla F(x_t) + \Delta G_t \quad \text{with} \quad \Delta G_t = \Delta(\nabla F(x_t))$$

It is supported by the Apache Commons Math 3.3 and above, Apache Spark/MLlib 1.0 and above, Colt 1.0 and above, and Iiitb CRF libraries. L-BFGS is used in minimization of the loss function in conditional random fields. For more information, refer to the *Conditional random fields* section of *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 $\{y, x_i\}$. The objective is to minimize the sum of the squares of residuals r_i :

$$\mathcal{L}(w) = \sum_{i=0}^{m-1} r_i(w)^2; \quad r_i = y_i - F(x_i, w)$$

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. For more information, refer to the *The logistic regression* section of *Chapter 6, Regression and Regularization*.

Levenberg-Marquardt

The **Levenberg-Marquardt** algorithm is an alternative to the Gauss-Newton technique for solving nonlinear least squares and curve fitting problems. The method consists of adding the gradient or 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. For more information, refer to the *The logistic regression* section of *Chapter 6, Regression and Regularization*.

Lagrange multipliers

The **Lagrange multipliers** methodology is an optimization technique 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 in minimizing the loss function in the non-separable case of linear support vector machines. For more information, refer to *The nonseparable case (soft margin)* section of *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 solutions of the subproblems are known as **optimal substructures**. Greedy algorithms or the computation of the minimum span of a graph are examples of 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 *The Viterbi algorithm* section of *Chapter 7, Sequential Data Models*), or the back-propagation of error in a multilayer perceptron (refer to the *Step 3 – error backpropagation* section of *Chapter 9, Artificial Neural Networks*) are good examples of overlapping substructures.

The mathematical formulation of dynamic programming solutions is specific to the problem it attempts to solve. 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 to evaluate a security – stock, bond, currency, or commodity – that entails attempting to measure its intrinsic value by examination related to 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 market price per share over gross sales or revenue.
- **Price/Book value ratio (PB):** This is the ratio of market price per share over total balance sheet value per share.
- **Price to Earnings/Growth (PEG):** This is the ratio of price/earnings per share (PE) over 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 over net sales.
- **Net profit margin:** This is the ratio of net profit over net sales (or 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 over total number of shares floated.
- **Cash per share:** This is the ratio of the value of cash per share over 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 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 income available to common stockholders, excluding extraordinary items, for the most recent trailing twelve 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 value 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 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:** A bear position attempts to profit by betting that the prices of the security will fall.
- **Bullish or bullish position:** A bull position attempts to profit by betting that the price of the security will rise.
- **Long position:** This is the same as bullish.

- **Neutral position:** A neutral position attempts to profit by betting the price of the security will not change significantly.
- **Oscillator:** An oscillator is a technical indicator that measures the price momentum of a security using some statistical formulae.
- **Overbought:** A security is overbought when its price rises too fast as measured by one or several trading signals or indicators.
- **Oversold:** A security is oversold when its price drops too fast as measured by one or several trading signals or indicators.
- **Relative strength index (RSI):** The RSI 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:** A resistance level 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:** A support level 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:** A technical indicator 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:** A signal is triggered when a technical indicator reaches a predefined value, upwards or downwards.
- **Volatility:** This is the variance or standard deviation of the price of a security over a period of time.

Trading signals and strategy

The purpose is to create a set variable x , derived from price and volume; $x = f(\text{price}, \text{volume})$ then generate predicates, $x \text{ op } c$, where op is a Boolean operator, such as $>$ or $=$. The op operator 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 for reference:

[

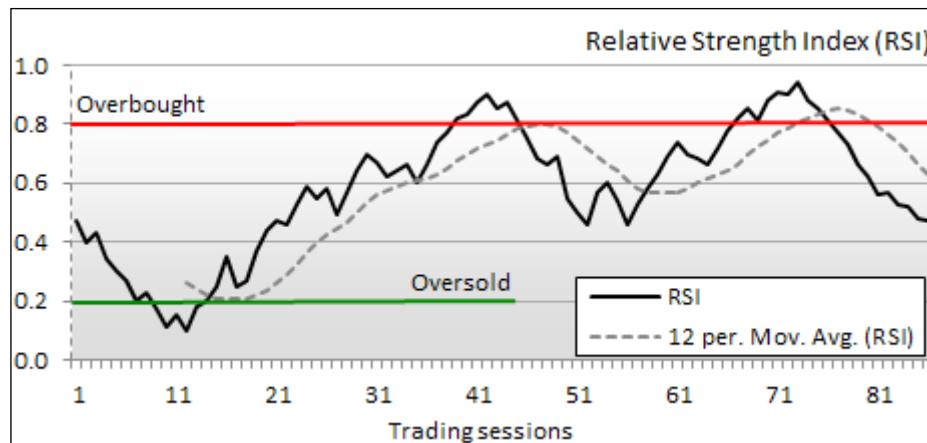
The RSI for a period of T sessions, with opening price p_o , and closing price p_c is given by:

$$U_T = \sum_{t=0}^{T-1} (p_c(t) > p_o(t))$$

$$RSI_T = 100 - \frac{100}{1 + \frac{U_T}{T - U_T}} \quad nRSI_T = RSI_T / 100$$

]

A **trading signal** is a predicate using a technical indicator $nRSI(t) < 0.2$. In trading terminology, a signal is emitted for any time period, t , for which the predicate is true. Have a look at the following graph:



Traders do not usually rely on a single trading signal to make a rational decision.

As an example, if G is the price of gold, $I10$ 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 the US\$ to the Japanese Yen maximizes for the trading strategy: $\{G < \$1170 \text{ and } I10 > 3.9\% \text{ and } RSI_{sp500} > 0.6 \text{ and } RSI_{sp500} < 0.8\}$.

Price patterns

Technical analysis assumes that historical prices contain some recurring albeit noisy patterns that can be discovered using the statistical method. The most common patterns used in the book are the trend, support, and resistance levels [A:18], as illustrated in the following chart:

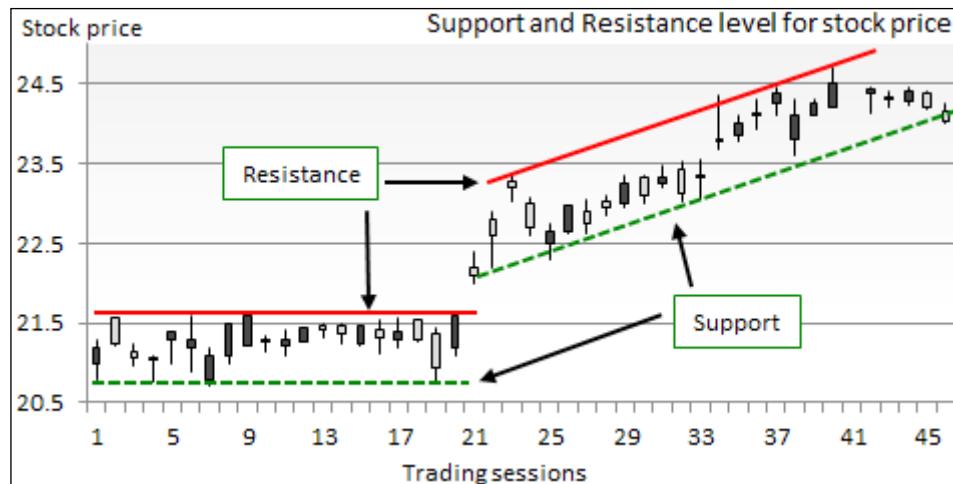


Illustration of trend, support, and resistance levels in technical analysis

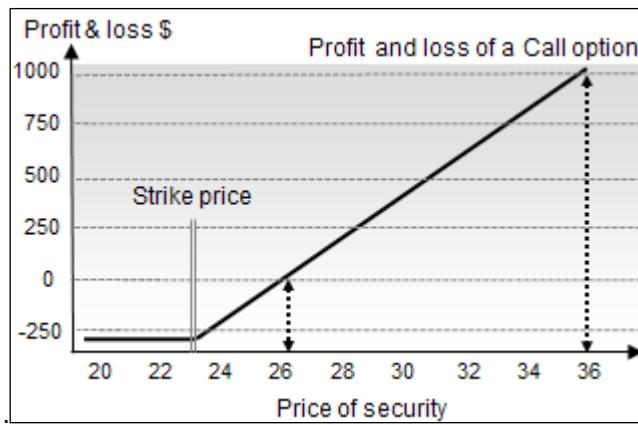
Options trading

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:

- 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 on 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 investment is $1030/270 = 380\text{ percent}$. Buying and then selling the stock would have generated a return on investment of $36/24 - 1 = 50\text{ percent}$. This example is simple and does not take into account transaction fee or margin cost [A:20]. Have a look at the following graph:



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) available at <http://finance.yahoo.com>
- Google finances (stocks, ETFs, and indices) available at <https://www.google.com/finance>
- NASDAQ (stocks, ETFs, and indices) available at <http://www.nasdaq.com>
- European Central Bank (European bonds and notes) available at <http://www.ecb.int>
- TrueFx (forex) available at <http://www.truefx.com>
- Quandl (economics and financials statistics) available at <http://www.quandl.com>
- Dartmouth University (portfolio and simulation) available at <http://mba.tuck.dartmouth.edu>

Suggested online courses

- *Practical Machine Learning.* J. Leek, R. Peng, B. Caffo. Johns Hopkins University, available at <https://www.coursera.org/jhu>
- *Probabilistic Graphical Models.* D. Koller. Stanford University, available at <https://www.coursera.org/course/pgm>
- *Machine Learning.* A. Ng. Stanford University, available at <https://www.coursera.org/course/ml>

References

- [A:1] *Daily scala: Enumeration.* J. Eichar. 2009, available at <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, available at <http://www.stat.wisc.edu/courses/st849-bates/lectures/Orthogonal.pdf>
- [A:5] *Eigenvalues and Eigenvectors of Symmetric Matrices.* I. Mateev. 2013, available at <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, available at http://www.cse.iitd.ac.in/~saroj/LFP/LFP_2013/L4.pdf
- [A:8] *Matrix Recipes.* J. Movellan. 2005, available at http://www.math.vt.edu/people/dlr/m2k_svb11_hesian.pdf
- [A:9] *Gradient descent.* Wikipedia: the free encyclopedia. Wikimedia foundation, available at http://en.wikipedia.org/wiki/Gradient_descent
- [A:10] *Large Scale Machine Learning: Stochastic Gradient Descent Convergence.* A. Ng. Stanford University, available at <https://class.coursera.org/ml-003/lecture/107>

- [A:11] *Large-Scale Machine Learning with Stochastic Gradient Descent*. L Bottou. 2010, available at <http://leon.bottou.org/publications/pdf/compstat-2010.pdf>
- [A:12] *Overview of Quasi-Newton optimization methods*. Dept. Computer Science. University of Washington, available at <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, available at <http://www.youtube.com>
- [A:14] *Introduction to the Lagrange Multiplier*. ediwm.com, video available at <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, available at http://nanocad.ee.ucla.edu/pub/Main/SnippetTutorial/Amar_DP_Intro.pdf
- [A:16] *Financial ratios*. Wikipedia, available at 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, the free encyclopedia. Wikimedia foundation, available at http://en.wikipedia.org/wiki/List_of_financial_data_feeds

Index

Symbols

0xdata H2O 446
2-step lag smoothing algorithm 92
|> operator 45
@specialized annotation 76
v-SVM 259

A

action methods, Apache Spark
 collect 435
 count 435
 countByKey 435
 first 435
 foreach 435
 reduce(f) 435
 saveAsTextFile 435
 take(n) 435
 takeSample 435
activation function 294
Actor model
 about 413, 414
 components 413
actor ! PoisonPill method 421
Actors
 about 12, 413
 Actor model 413, 414
 partitioning 415
 reactive programming 415
actorSystem.shutdown method 421
adaptive model 14
aggregation effect 115
Akka
 about 415, 416

futures 425
master-workers 417
URL 415
Akka, supervision strategies
 all-for-one strategy 416
 one-for-one strategy 416
Algebird
 about 22
 URL 11
AlgeBird 2.10 460
algebraic libraries
 about 459
 AlgeBird 2.10 460
 Breeze 0.8 460
 Colt 1.2.0 459
 jBlas 1.2.3 459
alternative techniques, data preprocessing
 autoregressive models 97
 curve-fitting algorithms 97
American Association of Individual Investors (AAII) 230
annual dividend yield 468
anti-goal state 389
Apache Commons Math
 about 20
 components 20
 exception handling 197
 installation 21
 licensing 20
 URL 20
Apache Public License 2.0
 URL 20
Apache Spark
 about 431, 432

benefits 432, 433
cons 445
deploying 437
design principles 433
experimenting 437
K-means 440-442
MLlib 439
performance evaluation 442
pros 445
RDD, generating 439, 440
Spark shell, using 438
URL 437

Apache Spark performance
considerations 444, 445
parameters, tuning 442
testing 443, 444

apply method 457

artificial neural networks 294

attributes 40

autonomous systems

about 366
characteristics 366

autoregressive integrated moving average (ARIMA) 97

autoregressive models 97

autoregressive moving average (ARMA) 97

B

batch EM 126

batch training 312

Baum-Welch estimator (EM) 222-225

Bayesian network 138

behavioral hidden Markov models 366

Bellman optimality equations 370, 371

benchmark framework 409, 410

Berkeley Data Analytics Stack (BDAS) 431

Bernoulli model

about 155
implementation 156

bias 58

bias input 292

bias-variance decomposition 58-61

binary SVC

about 262

C-penalty 269-271
kernel functions, evaluating 272-277

LIBSVM 262, 263

margin 269-271

risk analysis 277

software design 263, 264

SVM implementation 267-269

binomial classification, logistic regression 193-196

blocking, futures 426-428

bloom filters 407

Breeze 0.8 460

Broyden-Fletcher-Goldfarb-Shanno (BFGS) method 464

C

cake pattern 13

callbacks, futures

handling 426-430

case classes

advantages 451
versus companion objects 450
versus enumerations 450, 451

cash per share 468

centroid

about 101
versus mean 109

characteristics, Kalman filter

optimal 85
recursive 85

Chicago Board Options Exchange (CBOE) 382

chromosomes 329

class constructor template 449

classification model, evaluation factors

accuracy 55
F-measure 55
F-score F1 55
G-measure 55
precision 55
recall 55

classification model, terminology

false negatives (FN) 55
false positives (FP) 54

true negatives (TN) 54
 true positives (TP) 54
class prior 141
class prior probability 141
clustering (cluster analysis) 15, 100, 101
clustering algorithms
 EM 101
 K-means 101
cluster, K-means
 assignment 107
 configuration 103
 defining 103, 104
 initializing 105, 106
 iterative reconstruction 108, 109
 K-means, defining 105
 selecting 105
 tuning 114-117
code snippets
 format 448
Colt 1.2.0 459
common discriminative kernels 254-256
companion objects
 versus case classes 450
components, parallel collections
 ExecutionContextTaskSupport 408
 ForkJoinTaskSupport 408
 TaskSupport 408
 ThreadPoolTaskSupport 408
components, reinforcement learning
 mathematical notation 369
components, XCS
 action 397
 data 396
 environment 397
 input data stream 397
 predicate 397
 reward 397
 rule 397
 sensor 397
computational workflow
 basic statistics, computing 30, 31
 creating 28
 data, classifying 36, 37
 data, plotting 33, 34
 dataset, loading 29, 30
 dataset, preprocessing 30
 dataset, selecting 28, 29
 Gauss distribution 31, 32
 model, creating 34-36
 normalization 31, 32
 overview 26, 27
conditional dependency 146
conditional independence 137
conditional random field. *See* CRF
configuration, GA
 maxCycles 345
 mu 345
 softLimit 345
 xover 345
configuration parameters, SVM
 formulation 264
 kernel function, specifying 265
 SVMExecution class 266
conjugate gradient method 462
connectionism 290
constrained state-transition 386, 387
constructive tuning 313
consumer price index (CPI) 139
context bound
 about 23
 versus view bound 23
context.stop(childActorRef) method 421
context.stop(self) method 421
continuation-passing style (CPS) 415
control learning. *See* reinforcement learning
convex minimization 185
convolution 79
Cooley-Tukey algorithm 74
core parking 412
correction, recursive algorithm 91
cost/unfitness function 353, 354
count method 163
C-penalty, binary SVC 269-271
CRF
 about 232
 comparing, with HMM 249
 example 233, 234
 feature functions model 238-240
 identity potential functions 236
 implementation 241-246

potential functions (fi) 236
 software design 240, 241
 tests 246
 text analytics 237
 transition feature functions 236

CRF implementation

- control parameters 244, 245
- data sequences, extracting 244
- tags, generating 243
- training set, building 242, 243

CRF, tests

- L_2 regularization factor,
evaluating 248, 249
- training convergence profile 247
- training set size, evaluating 247, 248

crossover implementation

- about 335-345
- chromosomes 347, 348
- genes 348
- population 346, 347

C-SVM 259

curve-fitting algorithms 97

D

Darwinian process 328

data chunks 446

data elements 446

data extraction 453

data frames 446

data mining

- workflow 9

data preprocessing

- about 63
- alternative techniques 97
- purpose 63

data scientist 43

DataSource class

- headerLines parameter 453
- normalize parameter 453
- pathName parameter 453
- reverseOrder parameter 453
- srcFilter parameter 453

data sources 454, 455

data, XCS

- about 398, 399
- Signal 398
- XcsAction 398
- XcsSensor 398

DBpedia 159

decision-making agent 368

decoding (CF-3), HMM

- about 226
- Viterbi algorithm 226-228

dependency injection 13, 46, 47

descriptive model 14

design

- versus model 41

design principles, Apache Spark

- action methods 434, 435
- in-memory persistency 433
- lazy values, using 433, 434
- shared variables 436, 437
- transformation methods 434

design template, for classifiers 452

destructive tuning 313

DFT

- about 73, 78, 422, 425
- limitations 85
- used, for detecting market cycles 82-85

DFT-based filtering 79-81

DFT convolution 79

dimension reduction

- about 16, 126
- other techniques 133
- PCA 127

directed graphical models 138

discount coefficient for future rewards 369

discrete Fourier transform. *See DFT*

discrete Markov chain 208

discrete model parameters 330

discretization 331

discriminative models

- about 17, 18
- versus generative models 17

dividend coverage ratio 468

documents

- extracting 455, 456

DocumentsSource class 455, 456
domain 43
Domain Specific Languages (DSL) 14
dynamic programming
 about 466
 overview 466

E

earnings per share (EPS) 467
eigenvalue
 about 459
 decomposition 459

EM
 about 99, 118, 119
 considerations 134
 filtering 123
 GMM 119
 implementation 120-122
 online EM 126
 overview 120
 performance considerations 134
 relating, with K-means 125
 sampling 123
 testing 123-125
 third-party library exceptions 122

encapsulation
 about 449
 class scope 449
 object scope 449
 package scope 449

ensemble learning 330

enumerations
 advantages 451
 versus case classes 450, 451

epoch 300

Erlang programming language 413

error backpropagation, MLP training cycle
 about 305
 computational model 307
 error propagation 306

error insensitive zone 284

evaluation (CF-1), HMM
 about 216, 217
 Alpha class (forward variable) 217-219

Beta class (backward variable) 220, 221
constructors 222

evidence 141

evolution
 about 327
 evolutionary computing 329
 NP problems 328, 329
 origin 328

evolutionary computing 327, 329

exception handling 175

exchange-traded funds (ETFs)
 about 317
 CYB 318
 FXA 317
 FXB 317
 FXC 318
 FXE 317
 FXF 318
 FXY 318
 GLD 318
 SPY 318

execution state, HMM 214-216

expectation-maximization. *See* EM

experimentation, recursive algorithm 93-96

exploitation phase, XCS 395

exploration phase, XCS 395

exponential moving average 69-72

exponential normalization. *See* softmax

extended Kalman filter (EKF) 96

extended learning classifier systems. *See* XCS

F

fast Fourier transform (FFT) 73

feature functions. *See* transition feature functions

features, model
 attributes 40
 extracting 42
 selecting 41
 variables 40

features, Scala
 abstraction 11, 12
 computation on-demand 14
 configurability 13

maintainability 14
scalability 12, 13

Federal fund rate (FFR) 139

feed-forward neural networks (FFNN)
about 289
biological background 290, 291
mathematical background 291, 292
without hidden layers 294

filtering
versus smoothing 85

final val
versus val 271

finances 101
about 467
financial data sources 472
fundamental analysis 467
options, trading 471, 472
technical analysis 468

first-order discrete Markov chain 208, 209

first-order predicate logic 461

fitness function
about 330, 340
approximate fitness function 340
evolutionary fitness function 340
fixed fitness function 340
versus unfitness 342

flat encoding approach 334

fork-join pool 408

Fourier analysis
about 73
DFT 73-79
DFT-based filtering 79-81

Fourier transform 73

frequency domain 73

fully connected neural network 295

function approximation
about 385, 386
guidelines 385

fundamental analysis 467

futures
about 425
Actor life cycle 426
blocking 426-428
callbacks, handling 428-430
implementing 430, 431

G

GA

about 327
advantages 363
configuration 345
implementation 340
machine learning 330
risks 363
trading strategies 351, 352

GA implementation

about 340
configuration 345
crossover 345
key components 341-343
mutation 349
population growth, controlling 345
reproduction cycle 350, 351
selection 344
software design 340, 341

GA, applications

discrete model parameters 330
ensemble learning 330
neural network architecture 330
reinforcement learning 330

GA, components

genetic decoding 330
genetic encoding 330
genetic fitness function 330
genetic operations 330

Gaussian mixture 151

Gaussian mixture model (GMM) 134

Gaussian noise 87

Gaussian probability density 152

Gauss-Newton technique 465

Gene class

discr parameter 343
id parameter 343
op parameter 343
target parameter 343

generalized autoregressive conditional heteroskedasticity (GARCH) 97

generative models

about 16
versus discriminative models 17

generic message handler 427

genes 329

genetic algorithms. *See* GA

genetic decoding 330

genetic encoding

about 330

predicate encoding 332

solution encoding 333

value encoding 331, 332

genetic operators

about 335, 336

crossover 335, 338

mutation 335, 339

selection 335-337

gradient descent methods

about 462

conjugate gradient method 462

steepest descent method 462

stochastic gradient method 463

graph structured CRF 234

GraphX 432

gross domestic product (GDP) 139, 468

H

Hadoop Distributed File System (HDFS) 29

hard margin 257

Hessian matrix 461

hidden layers 293

hidden Markov model. *See* HMM

Hidden Naïve Bayes (HNB) 146

hierarchical encoding 334, 335

hinge loss 259

HMM

about 207-209

comparing, with CRF 249

components 210

decoding (CF-3) 211, 226

evaluation (CF-1) 211-217

execution state 214-216

implementation 228-230

lambda model 212-214

notation 211, 212

performance consideration 250

stationary or homogeneous restriction 210

test case 230, 231

time series analysis 232

training (CF-2) 211, 222

hyperplane 194

I

IEEE-732 encoding 356

implicit conversion, Scala 24

incremental EM 126

input forward propagation, MLP training

cycle

about 301, 302

computational model 302

objective 303, 304

softmax 304

installation, Apache Commons Math 21

I/O blocking operations 414

J

Jacobian matrix 461

Java 19

Java Native Interface (JNI) 460

Java packages

versus Scala traits 49

jBlas 1.2.3 459

JFreeChart

about 21

installation 22

licensing 21

URL 22

joint probability distribution 137

K

Kalman filter

about 85

characteristics 85

exception handling 92

recursive algorithm 87-89

state space estimation 86

usage 85

Kalman smoothing, recursive algorithm 92, 93

kernel functions
about 252
common discriminative kernels 254-256
evaluating 272-277
overview 252-254

kernel functions, types
laplacian kernel 254
linear kernel 254
log kernel 254
polynomial kernel 254
probabilistic kernels 256
RBF 254
reproducible Kernel Hilbert Spaces 256
sigmoid kernel 254
smoothing kernels 256

kernel trick 261

K-fold cross-validation 57

K-means
about 101
advantage 103
cluster assignment 107
cluster configuration 103
clusters, tuning 114-117
considerations, K-means 133, 134
dimensionality issue, of models 109, 110
exit condition 109
experiment 111-114
iterative reconstruction 108, 109
overview 103
performance considerations 133, 134
relating, with EM 125
similarity, measuring 101, 102
using 440-442
validation 117, 118

Kryo serialization 433

L

L₁ regularization
versus L₂ regularization 185

L₂ regularization
versus L₁ regularization 185

labeled data 54

Lagrange multipliers 261, 465

lambda model 212-214

laplacian kernel 254

Lasso regularization 185

latent Dirichlet allocation (LDA) 139

LCS
about 365, 391
benefits 393
complex adaptive systems 392
components 392
limitation 402, 403
XCS 395, 396

LCS, categories
Michigan approach 393
Pittsburgh approach 393

LCS, terminology
action 394
agent 394
classifier 394
compound predicate 394
covering 394
environment 394
input data stream 394
predicate 394
predictor 394
rule 394
rule fitness or score 394
rule matching 394
sensors 394

LDL decomposition 458

learning classifier systems. *See* LCS

learning vector quantization (LVQ) 100

least squares problem 191

Levenberg-Marquardt algorithm 465

Levenberg-Marquardt parameters 202

lexicon function 162

libraries
about 22
Algebird 22
Breeze 22
ScalaNLP 22

LIBSVM
about 262
benefits 262
Java code 263
scaling 279
URL 262

LIBSVM, Java classes

svm 263
svm_model 262
svm_node 262
svm_parameters 263
svm_problem 263

likelihood 141**Limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm 464****linear algebra**

about 457
algebraic libraries 459
Cholesky factorization 458
eigenvalue decomposition 459
LDL decomposition 458
LU factorization 458
QR decomposition 458
singular value decomposition (SVD) 459

linear chain CRF (linear chain structured graph CRF)

about 234-237
advantages 235

linear Kalman filter

limitations 96

linear kernel 254**linear regression**

about 169
OLS regression 173
one-variate linear regression 170
versus SVR 285-287

linear SVM

about 256
nonseparable case (soft margin) 258, 259
separable case (hard margin) 257

 L_n roughness penalty 184, 185**logistic regression**

about 192
binomial classification 193-196
classification 203-205
errors, rounding 205
logit function 192, 193
software design 196
training workflow 197, 198
validation methodology 205

logit function 192, 193**log kernel 254****Lotka-Volterra equation 337** **L_p -norm 184****LU factorization**

about 458
basic LU factorization 458
LU factorization with pivot 458

M**machine learning**

about 10, 330
regularization 185

machine learning algorithms

reinforcement learning 18, 19
supervised learning 16
taxonomy 15
unsupervised learning 15

machine learning, problems

classification 10
optimization 11
prediction 11
regression 11

Markov decision process

about 207
first-order discrete Markov chain 208, 209

Markov property 207**Markov random fields 209****master-workers (master-slaves)**

about 417
design principle 417
DFT 422-425
limitations 425
master Actor 419-421
master routing, implementing 421
messages exchange 417, 418
worker actors 418
workflow controller 419

mathematical concepts

about 457
dynamic programming 466
first-order predicate logic 461
Hessian matrix 461
Jacobian matrix 461

linear algebra 457
 optimization techniques, summary 462
mathematical notation 10
Matrix class 456, 457
max-margin classification 260, 261
mean
 versus centroid 109
mean squared error (MSE) 170, 301
measurement equation 86, 87
message-passing mechanisms, Actor model
 fire-and-forget (tell) 414
 send-and-receive (ask) 414
Michigan approach 393
MLlib
 about 432, 439
 components 439
MLP
 about 289-294
 activation function 294
 classification 312
 evaluation 315
 model 297
 network architecture 295
 software design 296
 training cycle 300
 training strategies 312
MLP algorithm, parameters
 config 310
 labels 310
 mlpObjective 310
 xt 310
MLPConfig configuration, parameters
 activation 310
 alpha 310
 eps 310
 eta 310
 hidLayers 310
 numEpochs 310
MLP, evaluation
 impact of learning rate 315, 316
 impact of momentum factor 316, 317
 test case 317-319
model
 about 14, 39-41
 adaptive model 14
 assessing 54
 bias-variance decomposition 58-61
 descriptive model 14
 instantiation 171
 overfitting 61
 predictive model 14
 validation 54
 versus design 41
model, forms
 chemistry 40
 differential 40
 directed graphs 40
 grammar 41
 graphical 40
 inference logic
 lexicon 41
 numerical method 40
 parametric 40
 probabilistic 40
 taxonomy 41
modeling 41
model, MLP
 about 297
 connections 299, 300
 layers 298
 synapses 299
monadic data transformation 45, 46
monads 11, 12
monoids 11, 12
Monte Carlo EM 126
moving averages
 about 66
 exponential moving average 69-72
 simple moving average 67, 68
 weighted moving average 68, 69
multilayer perceptron. *See* **MLP**
multinomial Naïve Bayes model
 about 139, 141
 attributes 149
 formalism 141, 142
 frequentist perspective 142, 143
 missing data, handling 151
 NaiveBayes class 150
 predictive model 144
 zero-frequency problem 145

multivariate Bernoulli classification
about 155
implementation 156
model 155

mutation implementation
about 335, 339, 349
chromosomes 349
genes 349
population 349

N

Naïve Bayes
about 139
applying, to text mining 156-158
benefits 168
disadvantages 168
mathematical notation 142
testing 163
using 156

Naïve Bayes classification
Gaussian density, using 152

Naïve Bayes classifiers
about 139
implementing 145
multinomial Naïve Bayes 139
UML class diagram 146

Naïve Bayes classifiers, implementing
about 145
classification 151, 152
labeling 152-154
results 154, 155
software design 145, 146
training phase 146-150

natural language processing (NLP) 239

net profit margin 467

net sales 467

network architecture, MLP 295

neural networks
about 289
advantages 324, 325
limitations 325

newState method
about 92
exit condition 93

N-fold cross-validation 280

nonlinear least squares minimization
about 464
Gauss-Newton technique 465
Levenberg-Marquardt algorithm 465

nonlinear SVM
about 260
kernel trick 261
max-margin classification 260, 261

notation, HMM
about 211, 212
variance 212

NP problems
about 327-329
NP-complete problems 328
NP-hard problems 329
P-problems 328

numerical optimization
about 191, 192
Newton (2nd order techniques) 192
Quasi-Newton (1st order techniques) 192

O

object creation
controlling 407

observation 42

OLS regression
about 173
design 173, 174
features selection test case 178-183
implementation 174
trending test case 175-177

one-class SVC
used, for anomaly detection 282, 283

one-variate linear regression
about 170
implementation 170, 171
test case 171, 172

online training 312

operating income 467

operating profit margin 467

operators, Scala 25

optimal substructures 466

optimization techniques
 gradient descent methods 462
 Lagrange multipliers 465
 nonlinear least squares minimization 464
 Quasi-Newton algorithms 463
 summary 462

OptionModel class
 implementing 384

OptionProperty class
 implementing 383

option trading, with Q-learning
 about 382, 383, 471, 472
 constrained state-transition 386, 387
 defining 383
 function approximation 385, 386
 implementing 387, 388
 normalized features 383
 OptionModel class, implementing 384
 OptionProperty class, implementing 383

ordinary least squares regression. *See OLS regression*

output unit activation function 294

overfitting
 about 61, 143
 solutions 62

overlapping substructures 466

overload operators
 $+$ 451
 $+=$ 451
 $|>$ 451
 about 451

P

padding 332

parallel collections, Scala
 about 407
 benchmark framework 409, 410
 creating 407
 performance evaluation 410-412
 processing 408

parent chromosomes
 preserving 339

partially connected neural networks 295

pay-out ratio 468

PCA
 about 99
 algorithm 128, 129
 considerations 134
 cross-validation 133
 evaluation 131-133
 implementation 129
 performance considerations 134
 purpose 127
 test case 130

penalized least squares regression. *See ridge regression*

penalty term 169

Pittsburgh approach 393

polynomial kernel 254

Population class
 chromosomes parameter 342
 limit parameter 342

population growth
 controlling 345

portfolio management
 with XCS 396-398

posterior probability 141

predicates
 encoding format 332

prediction phase, recursive algorithm 89, 90

predictive model 14, 144

prestart method 416

price/book value ratio (PB) 467

price/earnings ratio (PE) 467

price patterns 471

price/sales ratio (PS) 467

price to earnings/growth (PEG) 467

primal problem 259

primitive types, Scala 24

principal components analysis. *See PCA*

private value
 versus private[this] value 171

probabilistic graphical models 137

probabilistic kernels 256

probabilistic reasoning 137

propositional logic 460

proteins 252

protein sequence annotation 252

Q

Q-learning

about 366
actions, implementing 374, 375
action-value, implementing 376, 377
evaluation 389-391
implementation 373
key components, implementing 373, 374
model quality, measuring 379
policy, implementing 376, 377
prediction 381
search space, implementing 375, 376
software design 373, 374
states, implementing 374, 375
tail recursion 380, 381
training 378, 379
used, for option trading 382, 383

QR decomposition 94, 458

Quasi-Newton algorithms

about 463
Broyden-Fletcher-Goldfarb-Shanno (BFGS)
method 464
L-BFGS 464

R

r² statistics 182

radial basis function (RBF)

about 254
terminology 254

RDD

generating 439, 440

RDD, operations

action 431
transformation 431

real-world Bayesian network

example 138

receive method 416

recombination 329

recursive algorithm

about 87-89
correction 91
experimentation 93-96
Kalman smoothing 92, 93

prediction 89, 90

regression weights 170

regularization

about 169, 184
 L_n roughness penalty 184, 185
notation 184
ridge regression 186

reinforcement learning

about 14, 18, 19, 330, 365
Bellman optimality equations 370, 371
concept 368
pros and cons 391
Q-learning 366, 372, 373
temporal difference 371, 372
value-action iterative update 372, 373
value of policy 369, 370
versus supervised learning 368

reinforcement learning, terminology

absorbing state 367
action 367
agent 367
best policy 367
environment 367
episode 367
goal state 367
horizon 367
policy 367
reward 367
state 367
terminal state 367

reproducible Kernel Hilbert Spaces 256

reproduction cycle

implementation 350, 351

residual sum of squares (RSS)

about 169, 196
minimization techniques 173

Resilient Distributed Datasets (RDD) 14

ridge regression

about 169, 186
implementation 186, 187
test case 188-190

risk analysis, binary SVC

features 277-281
labels 277-281

router 416
rules discovery module 393
rules, XCS
defining 399-401

S

Scala
about 11, 20, 407
object creation, controlling 407
parallel collections 407
used, for building scalable frameworks 406
Scala plugin
for Eclipse, URL 20
for IntelliJ IDEA, URL 20
Scala programming
about 447
class constructor template 449
code snippet format 448
companion objects, versus case classes 450
data extraction 453, 454
data sources 454, 455
design template, for classifiers 452, 453
documents, extracting 455, 456
encapsulation 449
enumerations, versus case classes 450, 451
libraries directory 447
Matrix class 456
overload operators 451
Scala traits
versus Java packages 49
scheme, genetic encoding
flat encoding 334
hierarchical encoding 334, 335
score method 162
selection
about 335-337
implementation 344
Sequential Minimal Optimization (SMO) 259, 262
shared variables
about 436, 437
accumulator variables 436
broadcast values 436
short interest 467
short interest ratio 468
shrinkage 184
sigmoid kernel 254
signal encoding 356
simple build tool (sbt) 437
simple moving average 67, 68
singular value decomposition (SVD) 135
skip lists 407
smoothing
versus filtering 85
smoothing factor for counters 145
smoothing kernels 256
softmax 304
software design, MLP 296
software developer 43
solution encoding approach 333
source code, Scala
about 22, 23
context bound 23
immutability 25
implicit conversion 24, 25
iterator performance, evaluating 26
operators 25
presentation 23, 24
primitive types 24
view bound 23
Spark. See **Apache Spark**
Spark/MLLib 1.0 262
Spark shell
pitfalls 439
using 438
SparkSQL 432
spectral analysis 73
spectral density estimation 73
spreadsheets
using 78
state, dynamic systems 88
state space estimation
about 86
measurement equation 86, 87
transition equation 86, 87
stdDev() method 104
steepest descent method 462
stimuli 290

- stochastic gradient method** 463
Stream classes 407
subject-matter expert 43
subordinates 416
substructures 466
sum of squared errors (SSE) 170
supervised learning
 about 16
 autonomous systems, design problem 366
 discriminative models 17, 18
 generative models 16
 versus reinforcement learning 368
support vector classifier (SVC)
 about 262
 binary SVC 262
 one-class SVC 282, 283
support vector machines (SVM)
 about 251, 256
 components 263
 configuration parameters 264
 implementation 267-269
 linear SVM 256
 nonlinear SVM 260
 performance considerations 288
support vector regression (SVR)
 about 284
 overview 284, 285
 versus linear regression 285-287
SVC origin 282
SVM dual problem 261
SVMLight 262
synapse/weights adjustment, MLP training cycle
 about 308
 gradient descent 308
 implementation 309
- T**
- tagging model** 159
technical analysis
 about 468
 price patterns 471
technical analysis, terminology
 bearish position 468
 bullish position 468
 long position 468
 neutral position 469
 oscillator 469
 overbought 469
 oversold 469
 relative strength index (RSI) 469
 resistance 469
 short position 469
 support 469
 technical indicator 469
 trading range 469
 trading signals 469
 volatility 469
temporal difference
 about 371, 372
 exploration 371
 off-policy implementation 372
 on-policy implementation 372
TermsScore class
 about 162
 lexicon function 162
 toDate function 162
 toWords function 162
TermsScore.score method 165
test case, MLP
 about 317-319
 hidden layers architecture impact 323, 324
 implementation 319-321
 models evaluation 321, 322
test case, trading strategies
 about 357, 358
 configuration 359
 data extraction 358
 evaluation 360
 GA execution 360
 GA instantiation 359
 initial population, generating 358, 359
 unweighted score, evaluating 360, 361
 weighted score, evaluating 362, 363
testing, Naïve Bayes
 about 163
 evaluation 166, 167
 textual information, retrieving 163, 165

text mining
about 156
extraction of terms 160, 161
implementing 159
Naïve Bayes, applying to 156-158
scoring of terms 161-163

time series
about 63, 64
analysis, with HMM 232
implementation 65, 66

toDate function 162

tools
about 19
Apache Commons Math 20
Java 19
JFreeChart 21
Scala 20

toOrderedArray method 161

toWords function 162

trading operators 353

trading signals 354

trading strategies, GA
about 351-355
cost/unfitness function 353, 354
defining 353
signal encoding 356
test case 357, 358
trading operators 353
trading signals 354

training cycle, MLP
about 300
configuration 309
convergence criteria 309
error backpropagation 305
implementation 310, 311
input forward propagation 301, 302
sum of squared errors 305
synapse/weights adjustment 308

training strategies, MLP
batch training 312
model instantiation 313, 314
online training 312

prediction 314
regularization 313

training workflow
exit conditions, defining 200
Jacobian matrix, computing 199
least squares optimizer, configuring 198
least squares problem, defining 201
loss function, minimizing 201
testing 202

train method 150

transformation methods, Apache Spark
coGroup 435
distinct 434
filter(f) 434
flatMap(f) 434
groupByKey 434
join 435
map(f) 434
mapPartitions(f) 434
reduceByKey(f) 434
sample 434
sortByKey 435
union 434

transition equation 86

transition feature functions 236

transposition operator 336

tuning, GA 340

typed actors
versus untyped actors 416

U

underfitting 61

unsupervised learning
about 15
clustering 15
dimension reduction 16
EM 99
goal 99
K-means 99
PCA 99

untyped actors
versus typed actors 416

V

val

versus final val 271

validation, model

implementation 56, 57

key metrics 54, 55

K-fold cross-validation 57

value encoding 331, 332

value of policy 369, 370

variables, HMM execution state

Alpha 214

Beta 214

Delta 214

DiGamma 214

Gamma 214

Qstar 214

variance 58

vector quantization 100

view bound

about 23

versus context bound 23

Viterbi algorithm 226-228

W

weighted moving average 68, 69

while loop 75

WordNet 159

workflow

computational framework 44

dependency injection 46-48

designing 42, 43

example 51

modules 48

monadic data transformation 45, 46

pipe operator 44

workflow factory 49-51

workflow, example

clustering module 52, 53

preprocessing module 51, 52

workflow factory 49-51

X

XCS

about 395, 396

components 396

core data 398, 399

covering 401

example 401

exploitation phase 395

exploration phase 395

rules, defining 399-401

used, for portfolio management 396-398

Z

zero-frequency problem 145

zip method 102



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.

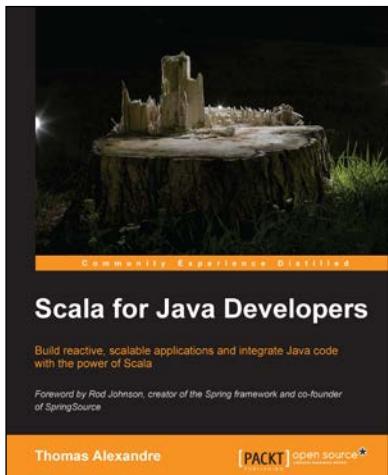
About Packt Open Source

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.

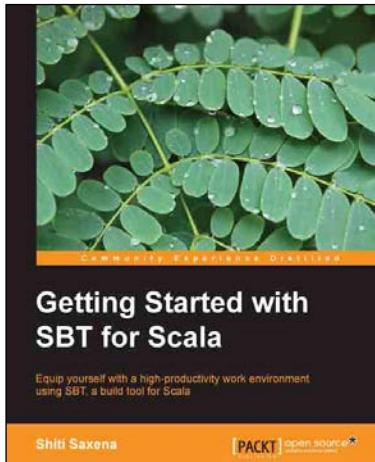


Scala for Java Developers

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.



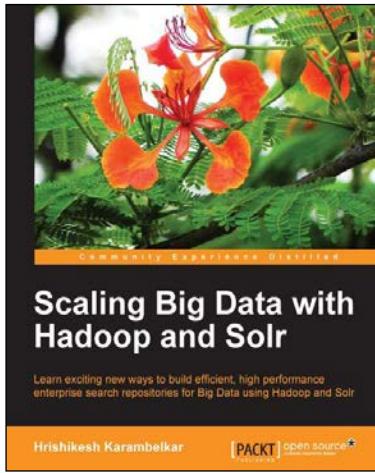
Getting Started with SBT for Scala

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

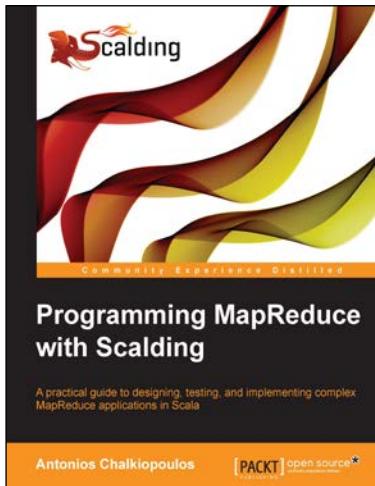


Scaling Big Data with Hadoop and Solr

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