

Patrick R. Nicolas

Scala for Machine Learning

Second Edition

Data processing, ML algorithms, smart analytics,
and more



Packt

Scala for Machine Learning Second Edition

Copyright © 2017 Packt Publishing

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

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

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

First published: December 2015

Second edition: September 2017

Production reference: 1190917

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78712-238-3

www.packtpub.com

Table of Contents

[Scala for Machine Learning Second Edition](#)

[Credits](#)

[About the Author](#)

[About the Reviewers](#)

[www.PacktPub.com](#)

[eBooks, discount offers, and more](#)

[Why subscribe?](#)

[Customer Feedback](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Downloading the color images of this book](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Getting Started](#)

[Mathematical notations for the curious](#)

[Why machine learning?](#)

[Classification](#)

[Prediction](#)

[Optimization](#)

[Regression](#)

[Why Scala?](#)

[Scala as a functional language](#)

[Abstraction](#)

[Higher kinded types](#)

[Functors](#)

- [Monads](#)
- [Scala as an object oriented language](#)
- [Scala as a scalable language](#)
- [Model categorization](#)
- [Taxonomy of machine learning algorithms](#)
 - [Unsupervised learning](#)
 - [Clustering](#)
 - [Dimension reduction](#)
 - [Supervised learning](#)
 - [Generative models](#)
 - [Discriminative models](#)
 - [Semi-supervised learning](#)
 - [Reinforcement learning](#)
 - [Leveraging Java libraries](#)
- [Tools and frameworks](#)
 - [Java](#)
 - [Scala](#)
 - [Eclipse Scala IDE](#)
 - [IntelliJ IDEA Scala plugin](#)
 - [Simple build tool](#)
 - [Apache Commons Math](#)
 - [Description](#)
 - [Licensing](#)
 - [Installation](#)
 - [JFreeChart](#)
 - [Description](#)
 - [Licensing](#)
 - [Installation](#)
 - [Other libraries and frameworks](#)
- [Source code](#)
 - [Convention](#)
 - [Context bounds](#)
 - [Presentation](#)
 - [Primitives and implicits](#)
 - [Immutability](#)
 - [Let's kick the tires](#)
 - [Writing a simple workflow](#)

Step 1 – scoping the problem
Step 2 – loading data
Step 3 – preprocessing data
 Immutable normalization
Step 4 – discovering patterns
 Analyzing data
 Plotting data
 Visualizing model features
 Visualizing label
Step 5 – implementing the classifier
 Selecting an optimizer
 Training the model
 Classifying observations
Step 6 – evaluating the model

Summary

2. Data Pipelines

Modeling

What is a model?

Model versus design

Selecting features

Extracting features

Defining a methodology

Monadic data transformation

Error handling

Monads to the rescue

Implicit models

Explicit models

Workflow computational model

Supporting mathematical abstractions

Step 1 – variable declaration

Step 2 – model definition

Step 3 – instantiation

Composing mixins to build workflow

Understanding the problem

Defining modules

Instantiating the workflow

Modularizing

[Profiling data](#)

[Immutable statistics](#)

[Z-score and Gauss](#)

[Assessing a model](#)

[Validation](#)

[Key quality metrics](#)

[F-score for binomial classification](#)

[F-score for multinomial classification](#)

[Area under the curves](#)

[Area under PRC](#)

[Area under ROC](#)

[Cross-validation](#)

[One-fold cross-validation](#)

[K-fold cross-validation](#)

[Bias-variance decomposition](#)

[Overfitting](#)

[Summary](#)

[3. Data Preprocessing](#)

[Time series in Scala](#)

[Context bounds](#)

[Types and operations](#)

[Transpose operator](#)

[Differential operator](#)

[Lazy views](#)

[Moving averages](#)

[Simple moving average](#)

[Weighted moving average](#)

[Exponential moving average](#)

[Fourier analysis](#)

[Discrete Fourier transform \(DFT\)](#)

[DFT-based filtering](#)

[Detection of market cycles](#)

[The discrete Kalman filter](#)

[The state space estimation](#)

[The transition equation](#)

[The measurement equation](#)

[The recursive algorithm](#)

Prediction
Correction
Kalman smoothing
Fixed lag smoothing
Experimentation
Benefits and drawbacks

Alternative preprocessing techniques

Summary

4. Unsupervised Learning

K-mean clustering

K-means

Measuring similarity
Defining the algorithm
Step 1 – Clusters configuration

Defining clusters

Initializing clusters

Step 2 – Clusters assignment

Step 3 – Reconstruction error minimization

Creating K-means components

Tail recursive implementation

Iterative implementation

Step 4 – Classification

Curse of dimensionality

Evaluation

The results

Tuning the number of clusters

Validation

Expectation-Maximization (EM)

Gaussian mixture model

EM overview

Implementation

Classification

Testing

Online EM

Summary

5. Dimension Reduction

Challenging model complexity

[The divergences](#)

[The Kullback-Leibler divergence](#)

[Overview](#)

[Implementation](#)

[Testing](#)

[The mutual information](#)

[Principal components analysis \(PCA\)](#)

[Algorithm](#)

[Implementation](#)

[Test case](#)

[Evaluation](#)

[Extending PCA](#)

[Validation](#)

[Categorical features](#)

[Performance](#)

[Nonlinear models](#)

[Kernel PCA](#)

[Manifolds](#)

[Summary](#)

[6. Naïve Bayes Classifiers](#)

[Probabilistic graphical models](#)

[Naïve Bayes classifiers](#)

[Introducing the multinomial Naïve Bayes](#)

[Formalism](#)

[The frequentist perspective](#)

[The predictive model](#)

[The zero-frequency problem](#)

[Implementation](#)

[Design](#)

[Training](#)

[Class likelihood](#)

[Binomial model](#)

[Multinomial model](#)

[Classifier components](#)

[Classification](#)

[F1 Validation](#)

[Features extraction](#)

Testing

Multivariate Bernoulli classification

Model

Implementation

Naïve Bayes and text mining

Basics information retrieval

Implementation

Analyzing documents

Extracting relative terms frequency

Generating the features

Testing

Retrieving textual information

Evaluating text mining classifier

Pros and cons

Summary

7. Sequential Data Models

Markov decision processes

The Markov property

The first-order discrete Markov chain

The hidden Markov model (HMM)

Notation

The lambda model

Design

Evaluation (CF-1)

Alpha (forward pass)

Beta (backward pass)

Training (CF-2)

Baum-Welch estimator (EM)

Decoding (CF-3)

The Viterbi algorithm

Putting it all together

Test case 1 – Training

Test case 2 – Evaluation

HMM as filtering technique

Conditional random fields

Introduction to CRF

Linear chain CRF

Regularized CRF and text analytics

The feature functions model

Design

Implementation

Configuring the CRF classifier

Training the CRF model

Applying the CRF model

Tests

The training convergence profile

Impact of the size of the training set

Impact of L2 regularization factor

Comparing CRF and HMM

Performance consideration

Summary

8. Monte Carlo Inference

The purpose of sampling

Gaussian sampling

Box-Muller transform

Monte Carlo approximation

Overview

Implementation

Bootstrapping with replacement

Overview

Resampling

Implementation

Pros and cons of bootstrap

Markov Chain Monte Carlo (MCMC)

Overview

Metropolis-Hastings (MH)

Implementation

Test

Summary

9. Regression and Regularization

Linear regression

Univariate linear regression

Implementation

Test case

Ordinary least squares (OLS) regression

Design

Implementation

Test case 1 – trending

Test case 2 – features selection

Regularization

Ln roughness penalty

Ridge regression

Design

Implementation

Test case

Numerical optimization

Logistic regression

Logistic function

Design

Training workflow

Step 1 – configuring the optimizer

Step 2 – computing the Jacobian matrix

Step 3 – managing the convergence of optimizer

Step 4 – defining the least squares problem

Step 5 – minimizing the sum of square errors

Test

Classification

Summary

10. Multilayer Perceptron

Feed-forward neural networks (FFNN)

The biological background

Mathematical background

The multilayer perceptron (MLP)

Activation function

Network topology

Design

Configuration

Network components

Network topology

Input and hidden layers

Output layer

Synapses
Connections
Weights initialization
Model
Problem types (modes)
Online versus batch training
Training epoch
 Step 1 – input forward propagation
 Computational flow
 Error functions
 Operating modes
 Softmax
 Step 2 – error backpropagation
 Weights adjustment
 Error propagation
 The computational model
 Step 3 – exit condition
 Putting it all together
Training and classification
 Regularization
 Model generation
 Fast Fisher-Yates shuffle
 Prediction
 Model fitness
Evaluation
 Execution profile
 Impact of learning rate
 Impact of the momentum factor
 Impact of the number of hidden layers
 Test case
 Implementation
 Models evaluation
 Impact of hidden layers' architecture
Benefits and limitations
Summary

11. Deep Learning
Sparse autoencoder

[Undercomplete autoencoder](#)
[Deterministic autoencoder](#)
[Categorization](#)
[Feed-forward sparse, undercomplete autoencoder](#)
[Sparsity updating equations](#)
[Implementation](#)
[Restricted Boltzmann Machines \(RBMs\)](#)
[Boltzmann machine](#)
[Binary restricted Boltzmann machines](#)
[Conditional probabilities](#)
[Sampling](#)
[Log-likelihood gradient](#)
[Contrastive divergence](#)
[Configuration parameters](#)
[Unsupervised learning](#)
[Convolution neural networks](#)
[Local receptive fields](#)
[Weight sharing](#)
[Convolution layers](#)
[Sub-sampling layers](#)
[Putting it all together](#)
[Summary](#)

[12. Kernel Models and SVM](#)

[Kernel functions](#)
[Overview](#)
[Common discriminative kernels](#)
[Kernel monadic composition](#)
[The support vector machine \(SVM\)](#)
[The linear SVM](#)
[The separable case \(hard margin\)](#)
[The non-separable case \(soft margin\)](#)
[The nonlinear SVM](#)
[Max-margin classification](#)
[The kernel trick](#)
[Support vector classifier \(SVC\)](#)
[The binary SVC](#)
[LIBSVM](#)

[Design](#)
[Configuration parameters](#)
 [The SVM formulation](#)
 [The SVM kernel function](#)
 [The SVM execution](#)
[Interface to LIBSVM](#)
[Training](#)
[Classification](#)
[C-penalty and margin](#)
[Kernel evaluation](#)
 [Application to risk analysis](#)
[Anomaly detection with one-class SVC](#)
[Support vector regression \(SVR\)](#)
 [Overview](#)
 [SVR versus linear regression](#)
[Performance considerations](#)
[Summary](#)

[13. Evolutionary Computing](#)

[Evolution](#)
 [The origin](#)
 [NP problems](#)
 [Evolutionary computing](#)
[Genetic algorithms and machine learning](#)
[Genetic algorithm components](#)
 [Encodings](#)
 [Value encoding](#)
 [Predicate encoding](#)
 [Solution encoding](#)
 [The encoding scheme](#)
 [Flat encoding](#)
 [Hierarchical encoding](#)
[Genetic operators](#)
 [Selection](#)
 [Crossover](#)
 [Mutation](#)
 [Fitness score](#)
[Implementation](#)

Software design

Key components

Population

Chromosomes

Genes

Selection

Controlling population growth

GA configuration

Crossover

Population

Chromosomes

Genes

Mutation

Population

Chromosomes

Genes

Reproduction

Solver

GA for trading strategies

Definition of trading strategies

Trading operators

The cost function

Market signals

Trading strategies

Signal encoding

Test case – Fall 2008 market crash

Creating trading strategies

Configuring the optimizer

Finding the best trading strategy

Tests

The weighted score

The unweighted score

Advantages and risks of genetic algorithms

Summary

14. Multiarmed Bandits

K-armed bandit

Exploration-exploitation trade-offs

Expected cumulative regret

Bayesian Bernoulli bandits

Epsilon-greedy algorithm

Thompson sampling

Bandit context

Prior/posterior beta distribution

Implementation

Simulated exploration and exploitation

Upper bound confidence

Confidence interval

Implementation

Summary

15. Reinforcement Learning

Reinforcement learning

Understanding the challenge

A solution – Q-learning

Terminology

Concept

Value of policy

Bellman optimality equations

Temporal difference for model-free learning

Action-value iterative update

Implementation

Software design

The states and actions

The search space

The policy and action-value

The Q-learning components

The Q-learning training

Tail recursion to the rescue

Validation

The prediction

Option trading using Q-learning

Option property

Option model

Quantization

Putting it all together

[Evaluation](#)

[Pros and cons of reinforcement learning](#)

[Learning classifier systems](#)

[Introduction to LCS](#)

[Combining learning and evolution](#)

[Terminology](#)

[Extended learning classifier systems](#)

[XCS components](#)

[Application to portfolio management](#)

[XCS core data](#)

[XCS rules](#)

[Covering](#)

[Example of implementation](#)

[Benefits and limitations of learning classifier systems](#)

[Summary](#)

[16. Parallelism in Scala and Akka](#)

[Overview](#)

[Scala](#)

[Object creation](#)

[Streams](#)

[Memory on demand](#)

[Design for reusing Streams memory](#)

[Parallel collections](#)

[Processing a parallel collection](#)

[Benchmark framework](#)

[Performance evaluation](#)

[Scalability with Actors](#)

[The Actor model](#)

[Partitioning](#)

[Beyond Actors – reactive programming](#)

[Akka](#)

[Master-workers](#)

[Messages exchange](#)

[Worker Actors](#)

[The workflow controller](#)

[The master Actor](#)

[Master with routing](#)

[Distributed discrete Fourier transform](#)

[Limitations](#)

[Futures](#)

- [Blocking on futures](#)
- [Future callbacks](#)
- [Putting it all together](#)

[Summary](#)

[17. Apache Spark MLlib](#)

[Overview](#)

[Apache Spark core](#)

[Why Spark?](#)

[Design principles](#)

- [In-memory persistency](#)

- [Laziness](#)

- [Transforms and actions](#)

- [Shared variables](#)

[Experimenting with Spark](#)

- [Deploying Spark](#)

- [Using Spark shell](#)

[MLlib library](#)

[Overview](#)

[Creating RDDs](#)

[K-means using MLlib](#)

[Tests](#)

[Reusable ML pipelines](#)

[Reusable ML transforms](#)

- [Encoding features](#)

- [Training the model](#)

- [Predictive model](#)

- [Training summary statistics](#)

- [Validating the model](#)

- [Grid search](#)

[Apache Spark and ScalaTest](#)

[Extending Spark](#)

[Kullback-Leibler divergence](#)

[Implementation](#)

[Kullback-Leibler evaluator](#)

Streaming engine

Why streaming?

Batch and real-time processing

Architecture overview

Discretized streams

Use case – continuous parsing

Checkpointing

Performance evaluation

Tuning parameters

Performance considerations

Pros and cons

Summary

A. Basic Concepts

Scala programming

List of libraries and tools

Code snippets format

Best practices

Encapsulation

Class constructor template

Companion objects versus case classes

Enumerations versus case classes

Overloading

Design template for immutable classifiers

Utility classes

Data extraction

Financial data sources

Documents extraction

DMatrix class

Counter

Monitor

Mathematics

Linear algebra

QR decomposition

LU factorization

LDL decomposition

Cholesky factorization

Singular Value Decomposition (SVD)

[Eigenvalue decomposition](#)
[Algebraic and numerical libraries](#)
[First order predicate logic](#)
[Jacobian and Hessian matrices](#)
[Summary of optimization techniques](#)
 [Gradient descent methods](#)
 [Steepest descent](#)
 [Conjugate gradient](#)
 [Stochastic gradient descent](#)
 [Quasi-Newton algorithms](#)
 [BFGS](#)
 [L-BFGS](#)
 [Nonlinear least squares minimization](#)
 [Gauss-Newton](#)
 [Levenberg-Marquardt](#)
 [Lagrange multipliers](#)
[Overview dynamic programming](#)
[Finances 101](#)
 [Fundamental analysis](#)
 [Technical analysis](#)
 [Terminology](#)
 [Trading data](#)
 [Trading signal and strategy](#)
 [Price patterns](#)
 [Options trading](#)
 [Financial data sources](#)
[Suggested online courses](#)
[References](#)
[B. References](#)
 [Chapter 1](#)
 [Chapter 2](#)
 [Chapter 3](#)
 [Chapter 4](#)
 [Chapter 5](#)
 [Chapter 6](#)
 [Chapter 7](#)
 [Chapter 8](#)

[Chapter 9](#)

[Chapter 10](#)

[Chapter 11](#)

[Chapter 12](#)

[Chapter 13](#)

[Chapter 14](#)

[Chapter 15](#)

[Chapter 16](#)

[Chapter 17](#)

[Index](#)

Credits

Author

Patrick R. Nicolas

Reviewers

Sumit Pal

Dave Wentzel

Commissioning Editor

Amey Varangaonkar

Acquisition Editor

Tushar Gupta

Content Development Editor

Amrita Noronha

Technical Editor

Nilesh Sawakhande

Copy Editors

Safis Editing

Laxmi Subramanian

Project Coordinator

Shweta H Birwatkar

Proofreader

Safis Editing

Indexer

Mariammal Chettiar

Graphics

Tania Dutta

Production Coordinator

Shantanu Zagade

Cover Work

Deepika Naik

About the Author

Patrick R. Nicolas is the director of engineering at Agile SDE, California. He has more than 25 years of experience in software engineering and building applications in C++, Java, and more recently in Scala/Spark, and has held several managerial positions. His interests include real-time analytics, modeling, and the development of nonlinear models.

About the Reviewers

Sumit Pal has more than 24 years of experience in the software industry, spanning companies from start-ups to enterprises.

He is a big data architect, visualization, and data science consultant, and builds end-to-end data-driven analytic systems.

Sumit has worked for Microsoft (SQLServer), Oracle (OLAP), and Verizon (big data analytics).

Currently, he works for multiple clients, building their data architectures and big data solutions and works with Spark, Scala, Java, and Python.

He has extensive experience in building scalable systems in middle tier, data tier to visualization for analytics applications, using big data and NoSQL databases.

Sumit has expertise in database internals, data warehouses, and dimensional modeling, as an associate director for big data at Verizon. Sumit strategized, managed, architected, and developed analytic platforms for machine learning applications. Sumit was the chief architect at ModelN/LeapfrogRX (2006-2013), where he architected the core analytics platform.

He is the author of *SQL On Big Data - Technology, Architecture and Roadmap* published by Apress in October 2016.

He has spoken on the topic covered in this book at the following conferences:

- May 2016, Big Data Conference—Linux Foundation in Vancouver, Canada
- March 2016, World Data Center Conference in Las Vegas, USA
- November 2015, BigData TechCon in Chicago, USA
- August 2015, Global Big Data Conference in Boston, USA

He is also the author of *SQL On Big Data* by Apress in December 2016.

Dave Wentzel is the **Chief Technology Officer (CTO)** of Capax Global, a premier Microsoft consulting partner. Dave is responsible for setting the strategy and defining service offerings and capabilities for the data platform and Azure practice at Capax. Dave also works directly with clients to help them with their big data journey. Dave is a frequent blogger and speaker on big data and data science topics.

eBooks, discount offers, and more

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 <customercare@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://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1787122387>.

If you'd like to join our team of regular reviewers, you can e-mail us at <customerreviews@packtpub.com>. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Preface

Not a single day passes 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 knowledge. This book covers the what, why, and how of machine learning:

- What are the objectives and the mathematical foundations 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 formulations, 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](#), *Data Pipelines*, 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*, covers key clustering methods such as K-means clustering, Gaussian mixture Expectation-Maximization and function approximation.

[Chapter 5](#), *Dimension Reduction*, describes the Kullback-Leibler divergence, the principal component analysis for linear models followed by an overview of manifold applied to non-linear models.

[Chapter 6](#), *Naive Bayes Classifiers*, focuses on the probabilistic graphical models and more specifically the implementation of Naive Bayes models and its application to text mining.

[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](#), *Monte Carlo Inference*, describes Gaussian sampling using Box-Muller technique, Bootstrap replication with replacement, and the ubiquitous Metropolis-Hastings algorithm for Markov Chain Monte Carlo.

[Chapter 9](#), *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 10](#), *Multilayer Perception*, describes feed-forward neural networks followed by a full implementation of the multilayer perceptron classifier.

[Chapter 11](#), *Deep Learning*, implements a sparse auto encoder and a restricted Boltzmann machines for dimension reduction in Scala followed by an overview of the convolutional neural network.

[Chapter 12](#), *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 13](#), *Evolutionary Computing*, covers describes the basics of evolutionary computing and the implementation of the different components of a multipurpose genetic algorithm.

[Chapter 14](#), *Multiarmed Bandits*, Multiarmed Bandits, introduces the concept of exploration-exploitation trade-off using Epsilon-greedy algorithm, the Upper confidence bound technique and the context-free Thompson sampling.

[Chapter 15](#), *Reinforcement Learning*, covers 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 16](#), *Parallelism in Scala and Akka*, describes some of the artifacts and frameworks to create scalable applications and evaluate the relative performance of Scala parallel collections and Akka-based distributed computation.

[Chapter 17](#), *Apache Spark MLlib*, covers the architecture and key concepts of Apache Spark, machine learning leveraging resilient distributed datasets, reusable ML pipelines, extending MLlib with distributed divergences and an example of Spark streaming library.

[Appendix A](#), *Basic Concepts*, describes the Scala language constructs used throughout the book, elements of linear algebra and optimization techniques.

[Appendix B](#), *References*, provides a chapter-wise list of references [source, entry] for each chapter.

What you need for this book

A decent command of the Scala programming language is a prerequisite. Reading through a mathematical formulation, conveniently defied 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 2.11.8 or higher
- Java 1.8.0_25
- SBT 0.13 or higher
- JFreeChart 1.0.17
- Apache Commons Math library 3.5 ([Chapter 3, Data Pre-processing](#), [Chapter 4, Unsupervised Learning](#), and [Chapter 9, 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 2.3.8 or higher (or Typesafe activator 1.2.10 or higher) ([Chapter 16, Parallelism in Scala and Akka](#))
- Apache Spark 2.1.0 or higher ([Chapter 17, Apache Spark MLlib](#))

Tip

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.

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**."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or 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 for this book 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.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Scala-for-Machine-Learning-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you 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.

Chapter 1. Getting Started

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

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

As we will emphasize throughout this book, each stage of the process is critical for building a model appropriate for the problem.

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

- Scala coding conventions and standards used in the book in the Appendix
- API Scala docs
- Fully documented source code, available online

This first chapter introduces the following elements:

- Basic concept of machine learning
- Taxonomy of machine learning algorithms
- Language, tools, frameworks, and libraries used throughout the book
- A typical workflow of model training and prediction
- A simple concrete application using binomial logistic regression

Mathematical notations 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 , as mentioned in a tip box will be as follows:

Note

Convention and notation

This book uses the 0 zero-based indexing of datasets in mathematical formulas.

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

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

Why machine learning?

The recent explosion in the number of digital devices has generated an ever-increasing amount of data. The best analogy I can find to describe the need, desire, and urgency for extracting 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 body temperature (continuous variable), congestion (discrete variables of *HIGH*, *MEDIUM*, and *LOW*), and the actual diagnosis (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 diagnosis.

Prediction

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

Optimization

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

Regression

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

Why Scala?

Like most functional languages, Scala provides developers and scientists with a toolbox to implement iterative computations that can be easily woven into a coherent dataflow. To some extent, Scala can be regarded as an extension of the popular map-reduce model for distributed computation of large amounts of data.

Note

Disclaimer

This section does not constitute a formal introduction or description of the features of Scala. It merely mentions some of its features that are valuable to machine learning practitioners. Experienced Scala developers may skip to the next section.

Among the capabilities of the language, the following features are deemed essential in machine learning and statistical analysis.

Scala as a functional language

There are many functional features in Scala which may unsettle software engineers with experience in object-oriented programming. This section deals specifically with monadic and functorial representations of data. Functors and monads are concepts defined in the field of mathematics known as category theory. Formerly:

- A functor is a data type that defines how a transformation known as a map applies to it. Scala implements functors as type classes with a `map` method.
- A monad is a wrapper around an existing data type. It applies a transformation to a data of wrapper type and returns a value of the same wrapper type. Scala implements monads as type classes with `unit` and `flatMap` methods. Monads extends functors in Scala.

Abstraction

Functors and *monads* are important concepts in functional programming.

Functors and monads are derived from category and group theory; they allow developers to create a high-level abstraction, as illustrated in the following Scala libraries:

- **Scalaz**: <https://github.com/scalaz>
- **Twitter's Algebird**: <https://github.com/twitter/algebird>
- **Google's Breeze**: <https://github.com/dlwh/breeze>

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

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

Covariant, *contravariant functors*, and *bi-functors* are well-understood

concepts in algebraic topology that are related to manifold and vector bundles. They are commonly used in differential geometry for the generation of non-linear models.

Higher kinded types

Higher kinded types (HKTs) are abstractions of types. They generate a new type from existing types. Let's consider the following parameterized trait:

```
trait M[T] { . }
```

A higher kinded type H over a trait M is defined as follows:

```
trait H[M[_]]; class H[M[_]]
```

Functors and monads are higher kinded types.

How are higher kinded types relevant to data analysis and machine learning?

Scientists define observations as sets or vectors of features.

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

Note

Inner product

M1: Definition of inner product $\langle \cdot, \cdot \rangle$ and co-vector α :

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

Let's define a vector as a constructor from any field, $_ \Rightarrow \text{Vector}[_]$. A co-

vector is then defined as the mapping function of a vector to its field:
`Vector[_]`.

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

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

Note

Tensors and manifolds

Vector and co-vector are classes of tensor (contravariant and covariant). Tensors (fields) are used in manifold learning non-linear models and in the generation of kernel functions. Manifolds are briefly introduced in the *Manifolds* section in. The topic of tensor fields in manifold learning is beyond the scope of this book.

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

- Covariant functor for the right projection
- Contravariant functor for the left projection

Functors

A **covariant functor** is a mapping function, such as $F: C \Rightarrow C$, with the following properties:

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

The definition of the covariant functor is $F[U \Rightarrow V] := F[U] \Rightarrow F[V]$. Its implementation in Scala is:

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

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

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

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

A contravariant functor is a mapping function, $F: C \Rightarrow C$, with the following properties:

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

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

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

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

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

}

Monads

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

Monads provide the ability for those collections to do the following:

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

The following Scala definition of a monad as a trait illustrates the concept of a higher kinded `Monad` trait for type `M`:

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

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

Note

Monadic composition of kernel functions

Monads are used in the composition of kernel functions in the *Kernel functions monadic composition* section in [Chapter 12, Kernel Models and Support Vector Machines](#).

Scala as an object oriented language

Machine learning models are generated through sequences of tasks or dataflows that demand a modular design.

As an object-oriented programming language, Scala allows developers to do the following:

- Define high-level component abstraction
- Allow different developers to work concurrently on different components
- Reuse code
- Isolate functionality for easier debugging and testing (unit tests)

You may wonder how Scala fares as an object-oriented programming against Java.

Tip

Scala versus Java

Scala is the purest form of object oriented language than Java. It does not support static methods (static methods are methods of singletons) and primitive types.

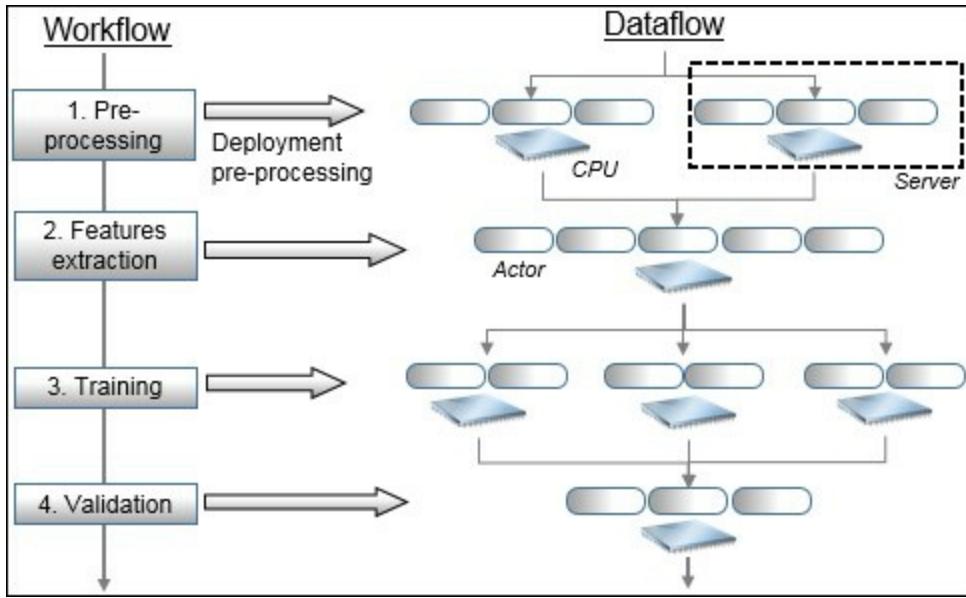
One important facet of object oriented programming is the ability to change modules or implement functionality on the fly, without the need to recompile the client code. This technique is known as dependency injection. Scala supports **dependency injection** using a combination of abstract variables, self-referenced composition, and stackable traits [1:3]. One of the most commonly used dependency injection patterns, the **cake pattern**, is described in the *Building workflows with mixins* section in [Chapter 2, Data Pipelines](#).

Scala as a scalable language

As seen previously, functors and monads enable the parallelization and chaining of data processing functions by leveraging the Scala higher-order methods. In terms of implementation, *actors* are one of the core elements that make Scala scalable. Actors provide Scala developers with a high level of abstraction to build scalable, distributed, and concurrent applications. Actors hide the nitty-gritty implementation of concurrency and the management of the underlying threads pool. Actors communicate through asynchronous immutable messages. A distributed computing Scala framework such as **Akka** or **Apache Spark** extends the capabilities of the Scala standard library to support computation on very large datasets. *Akka* and *Apache Spark* are described in detail in the last chapter of this book [1:4].

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

The following figure visualizes the different elements of the definition and deployment of a workflow (or data pipeline):



Deployment of a workflow for model training as a distributed computation

In the preceding diagram, a controller, that is, the *master node*, manages the sequence of tasks 1 to 4 in a similar way to a scheduler. These tasks are actually executed over multiple worker nodes, and are implemented by actors. The master node or actor exchanges messages with the workers to manage the state of the execution of the workflow, as well as its reliability, as illustrated in the Scalability with actors section of [Chapter 16, Parallelism with Scala and Akka](#). The high availability of these tasks is maintained through a hierarchy of supervising actors.

Tip

Domain-specific languages (DSLs)

Scala embeds *DSLs* 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. A notorious application of DSLs is the definition of the emulation of the syntax use in the MATLAB program, familiar to most data scientists.

Model categorization

A model can be predictive, descriptive, or adaptive.

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

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

A third category of models, known as **adaptive modeling**, is created through *reinforcement learning*. **Reinforcement learning** consists of one or several decision-making agents that recommend, and possibly execute, actions in an attempt to solve 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 which can be analyzed or mined to extract knowledge or 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 [1:5].

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: the discovery of data clusters and the discovery of latent factors. The methodology consists of processing input data to understand patterns similar to the natural learning process in infants or animals.

Unsupervised learning does not require labeled data (or expected values), and therefore, 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 in future classifications.

Clustering

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

- Creating a model making an assumption on the input data
- Selecting the objective function or goal of the clustering
- Evaluation of one or more algorithms to optimize the objective function

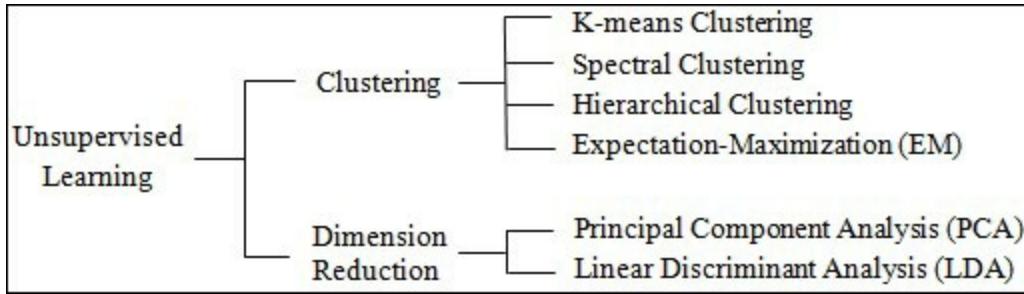
Data clustering is also known as **data segmentation** or **data partitioning**.

Dimension reduction

Dimension reduction techniques aim to find the smallest, yet most relevant, set of features needed to build a reliable model. There are many reasons for reducing the number of features or parameters in a model, from avoiding

overfitting to reducing computation costs.

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



Taxonomy of unsupervised learning algorithms

Supervised learning

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

Supervised machine learning algorithms can be broken down into two categories:

- *Generative* models
- *Discriminative* models

Generative models

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

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

Generative models attempt to fit a joint probability distribution $p(X, Y)$ of two events (or random variables), X and Y , representing two set of observed and hidden variables, x, y . Discriminative models compute the conditional probability $p(Y|X)$ of an event or random variable Y of hidden variables y , given an event or random variable X of observed variables x . Generative models are commonly introduced through 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 [1:6].

Note

Bayes' rule

Joint probability for independent random variables $X=x$ and $Y=y$:

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

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

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

Bayes' formula

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

Bayes' rule is the foundation of the Naïve Bayes classifier, which is described in the *Introducing the multinomial Naïve Bayes* section in [Chapter 6, Naïve Bayes Classifiers](#).

Discriminative models

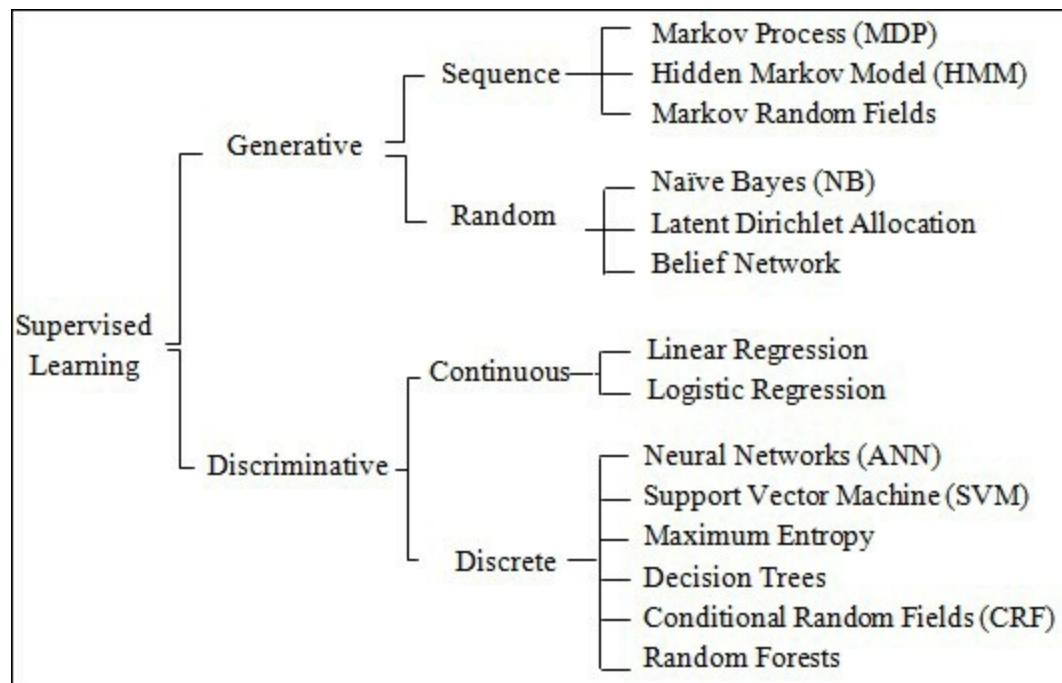
Contrary to generative models, discriminative models compute the conditional probability $p(Y|X)$ directly, using the same algorithm for training and classification.

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 are some brief guidelines describing which type of models make sense according to the objective or criteria of the project:

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

Constraints	degree of independence among the model features.	dependencies between features.
-------------	--	--------------------------------

We can further refine the taxonomy of supervised learning algorithms by segregating arbitrary, between sequential and random variables for generative models and by breaking down discriminative methods as applied to continuous processes (regression) and discrete processes (classification). The following figure illustrates a partial taxonomy of supervised learning algorithms:



Taxonomy of supervised learning algorithms

Semi-supervised learning

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

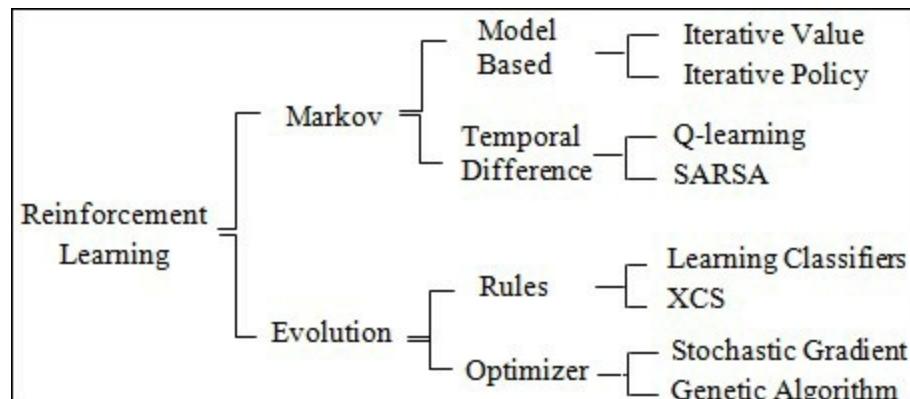
Reinforcement learning

Reinforcement learning is not as well understood as supervised and unsupervised learning outside the realm of robotics or game strategy. However, since the 1990s, genetic-algorithm-based classifiers have become increasingly popular in solving problems that require the collaboration of a system with a domain expert.

For some types of applications, reinforcement learning algorithms output a set of recommended actions for the adaptive system to execute. In its simplest form, these algorithms estimate the best course of action. Most complex systems based on reinforcement learning establish and update policies that can be vetoed by an expert, if necessary. The foremost challenge developers of reinforcement learning systems face is that the recommended action or policy may depend on a partially observable state.

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

As with the two previous learning strategies, reinforcement learning models can be categorized as Markovian or evolutionary. The following figure represents a partial taxonomy of the reinforcement learning algorithms:



Taxonomy of reinforcement learning algorithms

The genetic algorithm is described in [Chapter 13](#), *Evolutionary Computing*, and the Q-learning reinforcement method is introduced in [Chapter 15](#), *Reinforcement Learning*.

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

Leveraging Java libraries

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

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

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

Tools and frameworks

Before getting your hands dirty, you need to download and deploy the minimum set of tools and libraries; there is no need to reinvent the wheel, after all. A few key components have to be installed in order to compile and run the source code described throughout this book. We will focus on open source and commonly available libraries, although you are invited to experiment with the 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 2.11.4** and **2.11.8**. We recommend using Scala version **2.11.4** or higher with **SBT 0.13.1** or higher. Let's assume that the Scala runtime (`REPL`) and libraries have been properly installed and that the environment variables `SCALA_HOME`, and `PATH` have been updated.

The Scala standard library can be downloaded as binaries or as part of the *Typesafe Activator* tool by visiting <http://www.scala-lang.org/download/>.

Eclipse Scala IDE

The description and installation instructions for the *Eclipse Scala IDE* version 4.0 and higher is available at <http://scala-ide.org/docs/user/gettingstarted.html>.

IntelliJ IDEA Scala plugin

You can also download the **IntelliJ IDEA Scala plugin** version 13 or higher from the *JetBrains* website at <http://confluence.jetbrains.com/display/SCA/>.

Simple build tool

The ubiquitous **Simple Build Tool (SBT)** will be our primary building engine. It can be downloaded as part of the Typesafe activator or directly from <http://www.scala-sbt.org/download.html>.

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. To build Scala for machine learning, do the following:

- Set the maximum size for the JVM heap to 2058 Mbytes or higher and the permanent memory to 512 Mbytes or higher (that is, `-Xmx4096m -Xms512m -XX:MaxPermSize=512m`)
- To build the Scala for machine learning library package: `$(ROOT) / sbt clean publish-local`
- To build the package including test and resource files: `$(ROOT) / sbt clean package`
- To generate Scala doc for the library: `$(ROOT) / sbt doc`
- To generate Scala doc for the example: `$(ROOT) / sbt test:doc`
- To generate report for compliance to Scala style guide: `$(ROOT) / sbt scalastyle`
- To compile all examples: `$(ROOT) / sbt test:compile`

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 3.5 or higher.

The math library supports the following:

- Functions, differentiation, integral, and ordinary differential equations
- Statistics distributions
- Linear and non-linear optimization
- Dense and sparse vectors and matrix
- 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 <https://www.apache.org/licenses/LICENSE-2.0>.

Installation

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

1. Go to the download page at

http://commons.apache.org/proper/commons-math/download_math.cgi.

2. Download the latest .jar files in the binary section, commons-math3-3.6-bin.zip (for version 3.6, for instance).
3. Unzip and install the .jar file.
4. Add commons-math3-3.6.jar to the CLASSPATH, as follows:

- For macOS X:

```
export CLASSPATH=$CLASSPATH:/Commons_Math_path  
/commons-math3-3.6.jar
```

- For Windows:

Go to **System property | Advanced system settings | Advanced | Environment variables** and then edit the entry CLASSPATH variable.

5. Add the commons-math3-3.6.jar file to your IDE environment if needed:

- **Eclipse Scala IDE:** Project | Properties | Java Build Path | Libraries | Add External JARs
- **IntelliJ IDEA:** File | Project Structure | Project Settings | Libraries |

the source commons-math3-3.6-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** [1:8].

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 algorithm 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:
<https://sourceforge.net/projects/jfreechart/files/>.
3. Unzip and deploy the .jar file.
4. Add `jfreechart-1.0.17.jar` (for version 1.0.17) to the `CLASSPATH`, as follows:
 - For macOS X:

```
export CLASSPATH=$CLASSPATH:/JFreeChart_path/jfreechart-1
```

- For Windows:

Go to **System property** | **Advanced system settings** | **Advanced** | **Environment variables** and then edit the entry `CLASSPATH` variable.

5. Add the `jfreechart-1.0.17.jar` file to your IDE environment:
 - **Eclipse Scala IDE:** Project | Properties | Java Build Path | Libraries | Add External JARs
 - **IntelliJ IDEA:** File | Project Structure | Project Settings | Libraries | +

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 instructions for downloading them. Libraries related to the conditional random fields and support vector machines are described in their respective chapters.

Note

Why aren't we using Scala algebra and Scala numerical libraries?

Libraries such as **Breeze**, **ScalaNLP**, and **Algebroid** are interesting Scala frameworks for linear algebra, numerical analysis, and machine learning. They provide even the most seasoned Scala programmer with a high-quality layer of abstraction. However, this book is designed as a tutorial that allows developers to write algorithms from the ground up using existing or legacy java libraries [1:9].

Source code

The Scala programming language is used to implement and evaluate the machine learning techniques covered in Scala for machine learning. The source code presented in the book has been reduced to the minimum essential to the understanding of machine learning algorithms. The formal implementation of these algorithms is available on the website of *Packt Publishing*, <http://www.packtpub.com>.

Convention

The source code presented throughout the book follows a simple style guide and set of conventions.

Context bounds

Most of the Scala classes discussed in the book are parameterized with a type associated to the discrete/categorical value (`Int`) or continuous value (`Double`) [1:10]. For this book, **context bounds** are used instead of view bounds, as follows:

```
class A[T:ToInt] (param: Param//implicit conversion to Int  
class C[T:ToDouble] (param: Param)//implicit conversion to Double
```

Note

View bound deprecation

The notation for the view bound, `T <% Double`, is being deprecated in Scala 2.11 and higher. The declaration `class A[T <% Float]` is the short notation for `class A[T] (implicit f: T => Float)`.

Presentation

For the sake of readability of the implementation of algorithms, code non-essential to the understanding of a concept or algorithm, such as error checking, comments, exception, or import, is omitted. The following code elements are shown in the code snippets presented in the book:

- Code documentation:

```
// ....  
/* ... */
```

- Validation of class parameters and method arguments:

```
require( Math.abs(x) < EPS, " ...")
```

- Class qualifiers and scope declaration:

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

- Method qualifiers:

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

- Exceptions:

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

- Logging and debugging code:

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

- Non-essential annotation:

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

- Non-essential methods

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

Primitives and implicits

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

code, the following primitive types will be used:

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

Time series, introduced in the *Time series* section in [Chapter 3, Data Preprocessing](#), are implemented as `XSeries[T]` or `XVSeries[T]` of the parameterized type `T`. Make a note of these six types; they are used across the entire book.

The conversion between the primitive types listed above and types introduced in the particular library (that is, the *Apache Commons Math library*) is described in the relevant chapters.

Immutability

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

For example, 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 the following:

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

Tip

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. For instance, the monadic for expression is to be avoided as a counting iterator. The source code presented in this book uses the higher-order method `foreach` for iterative counting.

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.

Writing a simple workflow

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

Note

Introduction to logistic regression

Logistic regression is treated in depth in the *Logistic regression* section in [Chapter 9, Regression and Regularization](#). The model treated in this example is the simple binomial logistic regression classifier for two-dimension observations.

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

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

Step 1 – scoping the problem

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

Finances using the CSV format with the following fields:

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

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

```
type Features = Array[Double]
type Weights = Array[Double]
type ObsSet = Vector[Features]
type Fields = Array[String]

object YahooFinancials extends Enumeration {
    type YahooFinancials = Value
    val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME, ADJ_CLOSE=Value
    def toDouble(v: Value): Fields => Double =    7/1
        (s: Fields) => s(v.id).toDouble
    def toArray(vs: Array[Value]): Fields => Features = //2
        (s: Fields) => vs.map(v => s(v.id).toDouble)
}
```

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

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 Cisco stock 2012-2013

Step 2 – loading data

The second step is loading the dataset from local or remote data storage. Typically, a large dataset is loaded from a database or distributed filesystem such as **Hadoop Distributed File System (HDFS)**. The `load` method takes an absolute path name, `extract`, and transforms the input data from a file into a time series of type `Vector[DblPair]`:

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

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

Tip

Data extraction

The method invocation pipeline `Source.fromFile.getLines.map` returns an `Iterator`, which can be traversed only once.

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

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

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

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

Tip

Code readability

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

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

Then, break them down into several steps:

```
val lines = Source.fromFile.getLines
```

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

We strongly encourage the reader to consult the excellent guide *Effective Scala* written by *Marius Eriksen* from Twitter. This is definitively a must-read for any Scala developer [1:11].

Step 3 – preprocessing data

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

Immutable normalization

Logistic regression relies on the sigmoid curve or logistic function described in the *Logistic function* section in [Chapter 9, Regression and Regularization](#). The logistic function is used to segregate training data into classes. The output value of the logistic function ranges from 0 for $x = -\text{INFINITY}$ to 1 for $x = +\text{INFINITY}$. Therefore, it makes sense to normalize the input data or observation over $[0, 1]$.

Tip

To normalize or not normalize?

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

Normalization techniques include linear normalization and Z-score.

Normalization is an expensive operation that is not always needed.

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

Note

Linear normalization

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

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

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

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

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

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

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

```
class MinMax[T : ToDouble] (val values: Vector[T])  
{ //8  
  val zero = (Double.MaxValue, -Double.MaxValue)  
  val (min, max) = values./:(zero){ case ((mn, mx), x) => {  
    val z = implicitly[ToDouble[T]].apply(x)  
    if(z < mn) z else mn, if(z > mx) z else mx} //9
```

```

    }
  case class ScaleFactors(
    low:Double, high:Double, ratio: Double
  )

  var scaleFactors: Option[ScaleFactors] = None //10
  def normalize(low: Double, high: Double): Vector[Double]//1
  def normalize(value: Double): Double
}

```

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

```

def normalize(low: Double, high: Double): Vector[Double] =
  setScaleFactors(low, high).map( scale => { //12
    values.map(x =>{
      val z = implicitly[ToDouble[T]].apply(x)
      (z - min)*scale.ratio + scale.low //13
    })
  }).getOrElse(/* ... */)

def setScaleFactors(l: Double, h: Double): Option[ScaleFactors]={
  // .. error handling code
  Some(ScaleFactors(l, h, (h - l)/(max - min)))
}

```

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

```

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

```

The class `MinMax` normalizes single variable observations.

Tip

Statistics class

The class that extracts the basic statistics from a dataset, `stats`, introduced in the *Profiling data* section in [Chapter 2, Data Pipelines](#), inherits the class `MinMax`.

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

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

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

Step 4 – discovering patterns

The price action chart has a very interesting characteristic.

Analyzing data

At a closer look, a sudden change in price and increase in volume occurs about every 3 months or so. Experienced investors will undoubtedly recognize that these price-volume patterns are related to the release of quarterly earnings of Cisco. Such a regular but unpredictable pattern can be a source of concern or opportunity if risk can be properly managed. The strong reaction of the stock price to the release of corporate earnings may scare some long-term investors while enticing day traders.

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



Price-volume correlation for Cisco stock 2012-2013

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

Plotting data

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

Note

Plotting classes

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

Some of the classes used in visualization are described in the Appendix.

The dataset (volatility, volume) is converted into internal JFreeChart data

structures.

The following code snippet defines the key components of a simple scatter plot:

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

The class `ScatterPlot` implements a simple configurable scatter plot with the following arguments:

- `config`: Information, labels, and fonts of the plot
- `theme`: Predefined theme for the plot (black, white background, and so on)

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

Note

Visualization

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

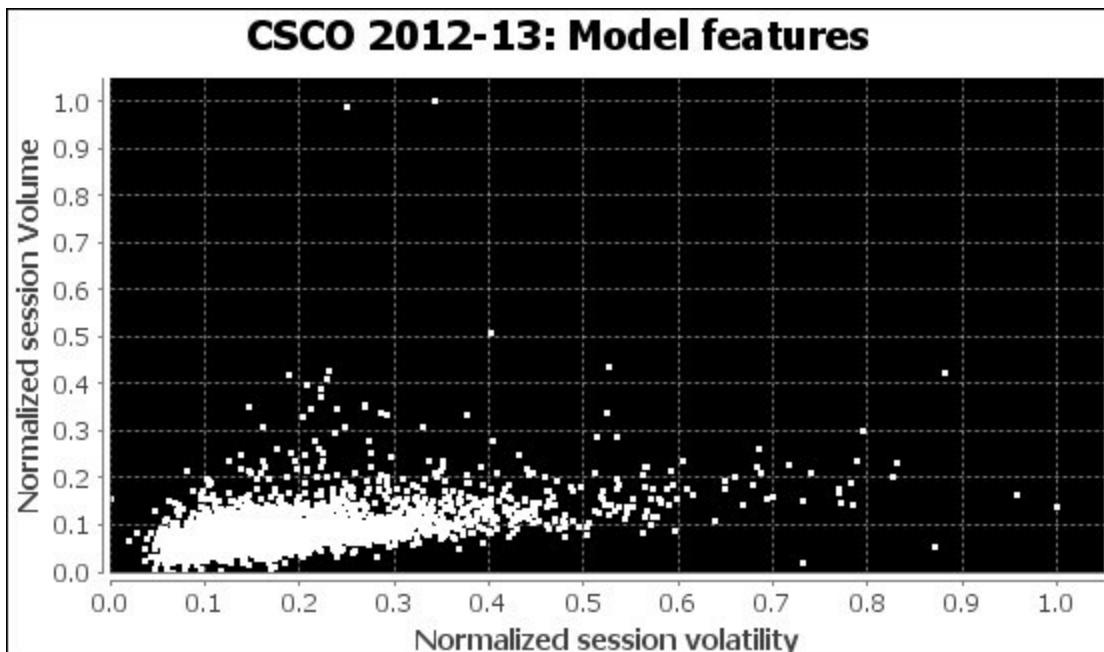
Visualizing model features

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

```
val plot = new ScatterPlot(("CSCO 2012-13 Model features",
  "Normalized session volatility", "Normalized session Volume"),
  new BlackPlotTheme)
```

```
plot.display(volatilityVolume, 250, 340)
```

The invocation of the method `display` generates the following output:

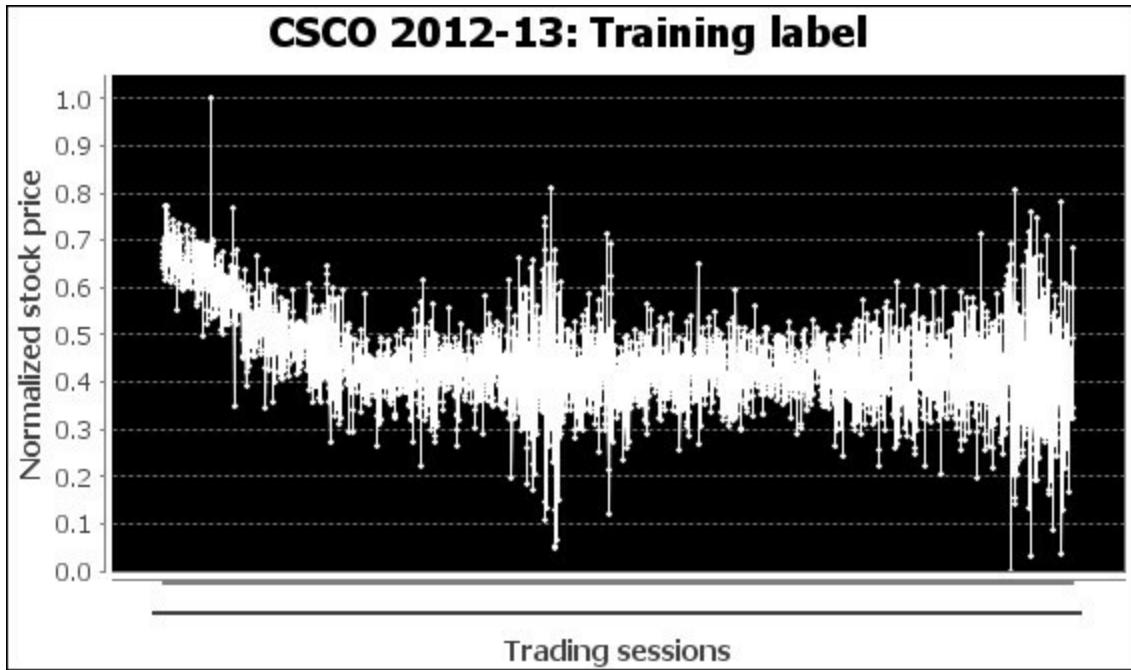


Scatter plot of volatility and volume for Cisco stock 2012-2013

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

Visualizing label

The normalized variation of the stock price between the opening and closing of the trading session is selected as the label for this classifier:



Classifier training label: normalized variation of stock price within a trading session

Step 5 – implementing the classifier

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

Logistic regression is commonly used in statistics inference.

Note

Logistic regression model (M4)

Given a model with weight w_i , the margin f and the logistic function l are defined as:

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

The first weight w_0 is known as the **intercept**. The binomial logistic regression is described in detail in the *Logistic regression* section in [Chapter 9, Regularization and Regression](#).

The following implementation of the binomial 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 class/model `LogBinRegression` is instantiated. As mentioned earlier, the sections of the code non-essential to the understanding of the algorithm are omitted.

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

- `observations`: Vector observations representing volume and volatility
- `expected`: A vector of expected values (relative price movement)
- `maxIters`: The maximum number of iterations allowed for the optimizer to extract the regression weights during training
- `eta`: Learning or training rate
- `eps`: The maximum value of the error (predicted – expected) for which the model is valid:

```
class LogBinRegression(
    observations: Vector[Features],
    expected: Vector[Double],
    maxIters: Int,
    eta: Double,
    eps: Double) { //18
    val model: LogBinRegressionModel = train //19
    def classify(obs: Feature): Try[(Int, Double)] //20
    def train: LogBinRegressionModel
    def intercept(weights: Weights): Double
    ...
}
```

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

```

case class LogBinRegressionModel (
    weights: Weights,
    losses: List[Double]
)

```

The model is fully defined by its `weights` as described in the mathematical formula M4. The intercept `weights(0)` represents the mean value of the prediction for observations whose variables are zero. The list `losses` contain the logistic loss collected at each iteration. It is used for debugging purposes. The intercept does not have a specific meaning in most cases and it is not always computable.

Tip

To intercept or not intercept?

The intercept corresponds to the value of weights when the observations have null values. It is a common practice to estimate, whenever possible, the intercept for binomial linear or logistic regressions independently from the slope of the model in the minimization of the error function. The multinomial regression models treat the intercept or weight w_0 as part of the regression model, as described in the *Ordinary least square regression* section of [Chapter 9, Regression and Regularization](#).

The following code snippet implements the computation of the `intercept` given a model, `Weights`:

```

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

```

The `classify` method takes new observations as input and computes the index of the classes (0 or 1) that the observations belong to, along with the actual likelihood (*line 20*).

Selecting an optimizer

The goal of the training of a model using expected values is to compute the optimal weights that minimize the error or **loss function**.

Note

Least squares or logistic loss

The sum of least squares loss is more often used for regression problems while the logistic loss is more commonly applied to classification.

We select the **Stochastic Gradient Descent (SGD)** algorithm to minimize the cumulative error between the predicted and expected values for all the observations. Although there are quite a few alternative optimizers, the SGD is quite robust and simple enough for this first chapter. The algorithm consists of updating the weights w_i of the regression model by minimizing the cost.

Note

Cost functions

M5: Logistic loss

$$\mathcal{L}(x | w) = \frac{1}{N} \sum_{n=0}^{N-1} \log\left(1 + e^{-y_n w^T x_n}\right)$$

M6: SGD method to update model weights at iteration t , w_t :

$$w_i^{(t+1)} = w_i^{(t)} + \eta \frac{x_i y}{1 + e^{y w^T x}}$$

For those interested in learning about optimization techniques, the *Summary of optimization techniques* section in the *Appendix* presents an

overview of the most commonly used optimizers. The stochastic descent gradient is used for the training of the multilayer perceptron (refer to the *The training epoch* subsection in the *The multilayer perceptron (MLP)* section of [Chapter 10, Multilayer Perceptron](#) for more detail).

The execution of the SGD algorithm follows these steps:

1. Initialize the weights of the regression model.
2. Shuffle the order of observations and expected pair of values.
3. Select the first pair of observations and expected value.
4. Compute the loss for this pair.
5. Update the model weights using the derivatives of the loss over each weight.
6. Repeat from step 3 until either the maximum number of iterations is reached or the incremental update of the loss is close to zero.

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

Tip

Batch and SGD

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

Training the model

The training method, `train`, consists of iterating through the computation of the weight using a simple descent gradient method. The method `train` computes the `weights`, collects the logistic loss, `losses`, at each iteration and returns an instance of the model `LogBinRegressionModel`. The code is

represented here:

```
def train: LogBinRegressionModel = {
    val init = Array.fill(nWeights) (Random.nextDouble) //22
    val (weights, losses) = sgd(
        0, init, List[Double]()
    )
    new LogBinRegressionModel(weights, losses.reverse) //23
}
```

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

Tip

Unwrapping values from Try:

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

- - catch the failure with `match{ case Success(m) => .case Failure(e) => }`
- - extract safely the result `getOrElse(/* ... */)`
- - propagate the results as a `Try` type `map(_.m)`

Let's look at the computation for the `weights` through the minimization of the loss function in the `sgd` method:

```
val shuffled = shuffle(observations.zip(expected)) //24
@tailrec
def sgd(  nIters: Int,
          weights: Weights, //25
          losses: List[Double]): (Weights, List[Double]
) = { //26
    if(nIters >= maxIters)
        (weights, losses) //27
    else {
```

```

    val (x, y) = shuffled(nIter % observations.size)
    val (newLoss, grad) = {
      val yDot = y * margin(x, weights)
      val gradient = derivativeLoss(y, yDot)
      (logisticLoss(yDot), // 28
       Array[Double](gradient) ++ x.map(_ * gradient)) // 29
    }

    if(newLoss < eps) // 30
      (weights, newLoss :: losses) // 31
    else {
      val newWeights = weights.zip(grad).map{
        case (w, df) => w - eta*df // 33
      }
      sgd(
        nIter + 1, // 34
        newWeights,
        newLoss :: losses)
    }
}

```

The `sgd` method recurses on the following arguments:

- The next labeled observation defined as a pair (observation, label) (*line 24*)
- The current number of iterations, `nIter`
- The model `weights` computed in the previous recursion (*line 25*)
- The current list of logistic loss values, `losses`, for debugging purposes (*line 26*)

Note

SGD implementation

This recursive implementation of SGD is simple and understandable but far from optimized. The different incarnation of SGD is a very well researched and documented field [1:12].

The method returns the pair of `weights` and the list of `losses` computed at each iteration if the maximum number of iterations allowed for the optimization is reached (*line 27*). The client code evaluates either the size of

the losses list or extracts its head value to validate whether SGD converged.

Note

SGD exit strategies

There are many different possible behaviors when the SGD reaches the maximum allowed number of iterations:

- Returns the final weights with a warning or a flag
- Throws an exception with a recovery mechanism
- Allows more iterations

The formula, M4, for the computation of the loss (*line 28*) and the gradient of the loss over each weight in formula, M5 (*line 29*), relies on two simple methods: `logisticLoss` and `derivativeLoss`. The code is as follows:

```
def logisticLoss(z: Double): Double =  
    log(1.0 + exp(-z)) / observations.size //30  
def derivedLoss(y: Double, yDot: Double):Double =  
    -y / (1.0 + exp(yDot))
```

The logistic loss is normalized by the number of observations (*line 30*).

The method evaluates new loss against the convergence criterion `eps` (*line 31*) and returns a version of the pair `(weights, losses)` (*line 32*) if the SGD converges. The formula M4 that updates the weights is implemented by zipping the weights and the gradient (*line 33*). The next invocation of SGD selects the next observation in the shuffled sequence of observations using a modulo operator to avoid overflowing (*line 34*).

Finally, here is an example of implementation of the margin formula:

```
def margin(observation: Features, weights: Weights):Double =  
    weights.drop(1).zip(observation.view)  
        .aggregate(weights.head) (dot, _ + _)
```

This implementation of the margin includes the intercept with its weight associated to the bias, a feature of the value 1.0.

Note

Bias value

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

This technique is used in the computation of the activation function of the multilayer perceptron as described in the *Multilayer perceptron* section in [Chapter 9, Artificial.](#)

The sequence of observations is randomly shuffled before the SGD is computed. This implementation of shuffling relies on the Scala standard library method, `scala.util.Random.shuffle` [1:13].

Note

Fisher-Yates shuffling

The *Training and classification* subsection in the *The multilayer perceptron (MLP)* section of [Chapter 10, Multilayer Perceptron](#), describes an alternative and efficient shuffling algorithm.

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

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

In this particular case, the labeling is automated because the relative price movement is extractable from raw data.

Tip

Automated labelling

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

Classifying observations

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

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

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

The computation of the margin as product of weights and observations is as follows:

```
def margin(obs: Features, weights: Weights): Double =  
    weights.drop(1).zip(obs.view)  
        .aggregate(0.0){case (s, (w,x)) => s + w*x, _ + _ }
```

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

Step 6 – evaluating the model

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

```
val NITERS = 4096; val EPS = 0.001; val ETA = 0.0001
val path_training = "supervised/regression/CSCO.csv"
val path_test = "supervised/regression/CSCO2.csv"
```

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

```
for {
    path <- getPath(path_training)
    (volatility, vol) <- load(path)
    minMaxVec <- Try(new MinMaxVector(volatility))
    normVolatilityVol <- Try(minMaxVec.normalize(0.0, 1.0))
    classifier <- logRegr(normVolatilityVol, vol)

    testValues <- load(path_test)
    normTestValue0 <- minMaxVec.normalize(testValues._1(0))
    class0 <- classifier.classify(normTestValue0)
    normTestValue1 <- minMaxVec.normalize(testValues._1(1))
    class1 <- classifier.classify(normTestValue1)
} yield {
    val modelStr = model.toString
}
```

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

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

```

type Labels = (Vector[Features], Vector[Double])

def load(fileName: String): Try[Labels] = {
  val src = Source.fromFile(fileName)
  val data = extract(src.getLines.map(_.split(",")).drop(1)) //4
  src.close; data //47
}

```

The method `logRegr`, implemented in the following code snippet, has two purposes:

- Labeling automatic observations, `obs`, to generate `real` values after normalization (*line 48*)
- Initializing (the instantiation and training of the model) the binomial logistic regression (*line 49*):

```

def logRegr(x: Vector[Features]): Try[LogBinRegression] = Tr
  val (obs, real) = x
  val normReal = normalize(real)
    .getOrElse(Vector.empty[Double]) //48
  new LogBinRegression(obs, normReal, NITERS, ETA, EPS) //49
}

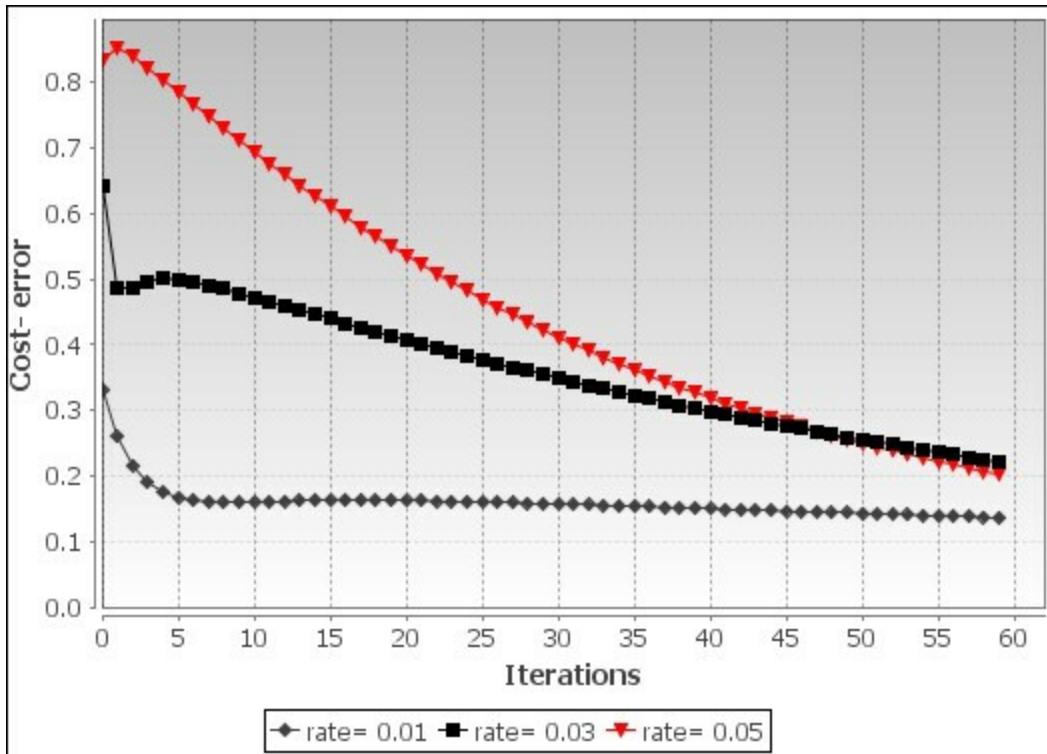
```

Note

Validation

The simple classification in this test case is provided for illustrating the runtime application of the model. It does not constitute a validation of the model by any stretch of imagination. The next chapter digs into validation methodologies (refer to the *Accessing a model* section of [Chapter 2, Data Pipelines](#), for more detail).

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



Impact of learning rate on the SGD on the convergence of the loss

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

The execution of the test produces the following model:

```
iters = 495
weights: 0.859,-3.617,-6.927
input (0.0088, 4.10E7) normalized (0.063,0.061) class 1 predictio
input (0.0694, 3.68E8) normalized (0.517,0.641) class 0 predictio
```

These values may differ between experiments as the initial weights of the model are initialized randomly.

Note

Learning more about regressive models

The binomial logistic regression is merely used to illustrate the concept of training and prediction. It is described in detail in the *Logistic regression*

section in [Chapter 9](#), *Regularization and Regression*.

Summary

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

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

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

Chapter 2. Data Pipelines

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, followed by a description of a configurable workflow computation model. The chapter concludes with an overview of different model validation techniques.

You will learn how to do the following:

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

Modeling

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

What is a model?

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 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 relations between those features. For instance, the model represented by the function $f(x,y) = x \cdot \sin(2y)$ has two features x and y and a relation f . Those two features are assumed to be independent. If the model is subject to a constraint, such as $f(x, y) < 20$, for example, then the conditional independence is no longer valid.

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

Models come in a variety of shapes and forms:

- **Parametric:** This consists of functions and equations (for example, $y = \sin(2t+w)$)
- **Differential:** This consists of ordinary and partial differential equations (for example, $dy = 2x \cdot dx$)

- **Probabilistic:** This consists of probability distributions (for example, $p(x|c) = \exp(k \log x - x)/x!$)
- **Graphical:** This consists of graphs that abstract out the conditional independence between variables (for example, $p(x,y|c) = p(x|c).p(y|c)$)
- **Directed graphs:** This consists of a temporal, spatial relationships (for example, scheduler)
- **Numerical method:** This consists of computational method such as finite difference, finite elements or Newton-Raphson
- **Chemistry:** This consists of formulas and components (for example, $H_2O, Fe + C_{12} = FeC_{13}$)
- **Taxonomy:** This consists of a semantic definition and a 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 rules (for example, *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 assumes, which becomes an assertion if proven correct (for example, the US population p increases by 1.2% a year, $dp/dt = 1.012$).
- **Designing:** This is manipulating representation for things you don't know. Designing can be regarded as the exploration phase of modeling (for example, what are the features that contribute to the growth of US population? Birth rate? Immigration? Economic conditions? Social policies?).

Selecting features

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

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

1. Search for new feature subsets.
2. Evaluate these feature subsets using a scoring mechanism.

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

Extracting features

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

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

Defining a methodology

Let's start by clarifying the role of the data scientist, software engineer, and domain expert.

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 creating software applications, including analysis, design, coding, testing, and deployment.

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

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

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

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

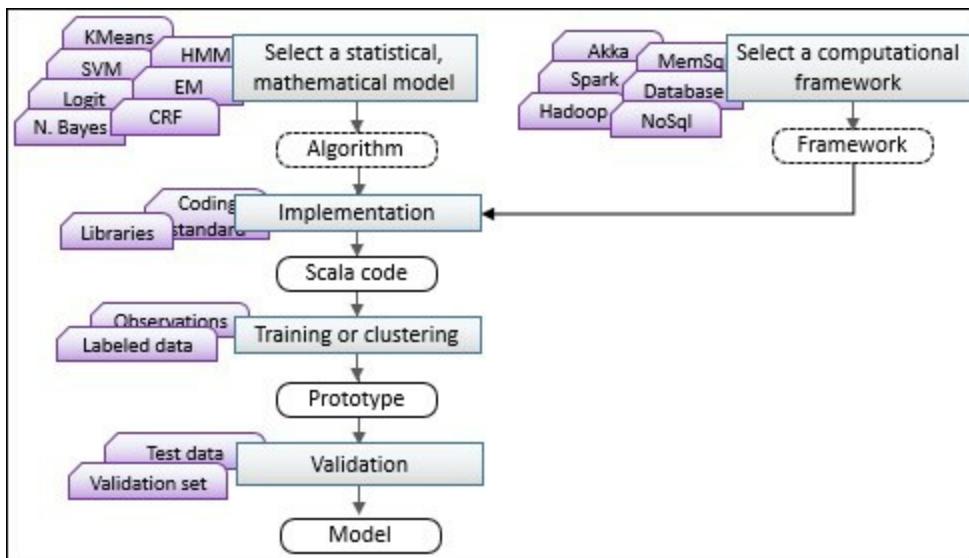
- Available resources, such as machines, CPU, memory, or I/O bandwidth
- Implementation strategy, such as iterative versus recursive computation or caching

- Requirement for responsiveness of the overall process, such as duration of computation or display of intermediate results

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

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

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



Statistical and computation modelling for machine learning applications

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

Monadic data transformation

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

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

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

Error handling

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

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

Tip

Partial function reusability

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

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

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

Note

Runtime validation of a partial function

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

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

This preemptive approach allows the developer to select an alternative method or a full function [2:3]. It is an efficient alternative to catching a `MathErr` exception.

The validation of partial functions is omitted throughout the book for the sake of clarity.

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

```
def |> : PartialFunction[T, Try[A]]
```

F# language reference

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

Monads to the rescue

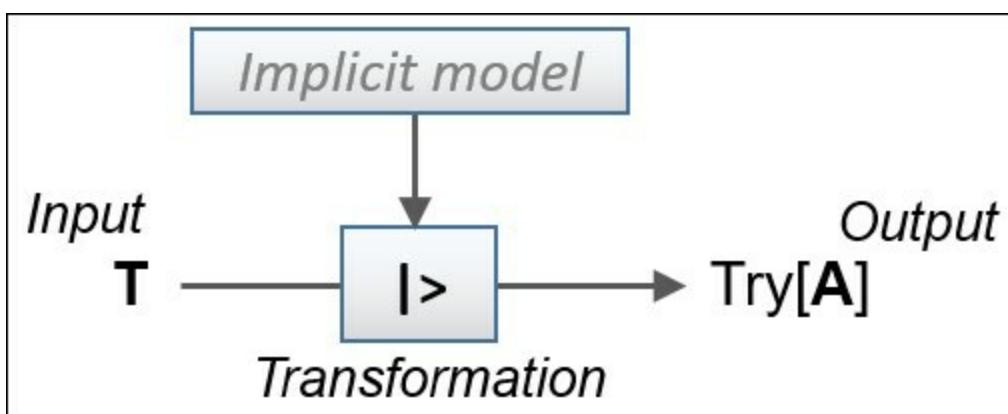
The objective is to define a symbolic representation of the transformation of different types of data without exposing the internal state of the algorithm implementing the data transformation.

Note

This section illustrates the concept of monadic data transformation, which is not essential to the understanding of machine learning algorithms as described throughout the book. You can safely skip to the *Workflow computational models* section.

Implicit models

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



Visualization of implicit models

```
trait ITransform[T, A] { //1
  self =>
    def |> : PartialFunction[T, Try[A]] //2
    def map[B](f: A => B): ITransform[T, B]
```

```

def flatMap[B] (f: A => ITransform[T, B]): ITransform[T, B]
def andThen[B] (tr: ITransform[A, B]): ITransform[T, B]
}

```

An implicit transformation has a type `ITransform` with two parameter types (line 1):

- **T**: Type of feature or element of the input collection
- **A**: Type of element of the output collection

For instance, the moving average on a time series of a single variable is computed by a `ITransform[Double, Double]`. The input collection is the time series and the output is a smoothed time series.

Note

Apache Spark ML transformers

The concept behind `ITransform` is somewhat similar to the Apache Spark MLlib transformers on data frames described in the *ML Reusable Pipelines* section of [Chapter 17, Apache Spark MLlib](#).

The method `|>` declares the transformation that is defined by implementing the trait `ITransform` (line 2). Let's look at the monadic operators.

The `map` method applies a function to each element of the output of the transformation `|>`. It generates a new `ITransform` by overriding the `|>` method (line 3).

A new implementation of the data transformation `|>` returning an instance of `PartialFunction[T, Try[B]]` (line 4) is created by overriding the methods `isDefinedAt` (line 5) and `apply` (line 6):

```

def map[B] (f: A => B): ITransform[T, B] = new ITransform[T, B] {
  override def |>: PartialFunction[T, Try[B]] = //3
    new PartialFunction[T, Try[B]] { //4
      override def isDefinedAt(t: T): Boolean = //5
        self.|>.isDefinedAt(t)
      override def apply(t: T): Try[B] = self.|>(t).map(f) //6
    }
}

```

The overridden methods for the instantiation of `ITransform` in `flatMap` follow the same design pattern as the `map` method. The argument `f` converts each output element into an implicit transformation of type `ITransform[T, B]` (line 7), and outputs a new instance of `ITransform` after flattening (line 8).

As with the `map` method, it overrides the implementation of the data transformation `|>` returning a new partial function (line 9) after overriding the `isDefinedAt` and `apply` methods:

```
def flatMap[B] (
  f: A => ITransform[T, B] //7
): ITransform[T, B] = new ITransform[T, B] { //8

  override def |> : PartialFunction[T, Try[B]] =
    new PartialFunction[T, Try[B]] { //9
  override def isDefinedAt(t: T): Boolean =
    self.|>.isDefinedAt(t)
  override def apply(t: T): Try[B] =
    self.|>(t).flatMap(f(_)).|>(t)
}
```

The method `andThen` is not a proper element of a monad. Its meaning is similar to the Scala method `Function1.andThen` that chains a function with another one. It is indeed useful to create chains of implicit transformations. The method applies the transformation `tr` (line 10) to the output of this transformation. The output type of the first transformation is the input type of the next transformation, `tr`.

The implementation of the method `andThen` follows a pattern similar to the implementation of `map` and `flatMap`:

```
def andThen[B] (
  tr: ITransform[A, B] (line 10)
): ITransform[T, B] = new ITransform[T, B] {

  override def |> : PartialFunction[T, Try[B]] =
    new PartialFunction[T, Try[B]] {
  override def isDefinedAt(t: T): Boolean =
    self.|>.isDefinedAt(t) &&
      tr.|>.isDefinedAt(self.|>(t).get)
  override def apply(t: T): Try[B] = tr.|>(self.|>(t).get)
}
```

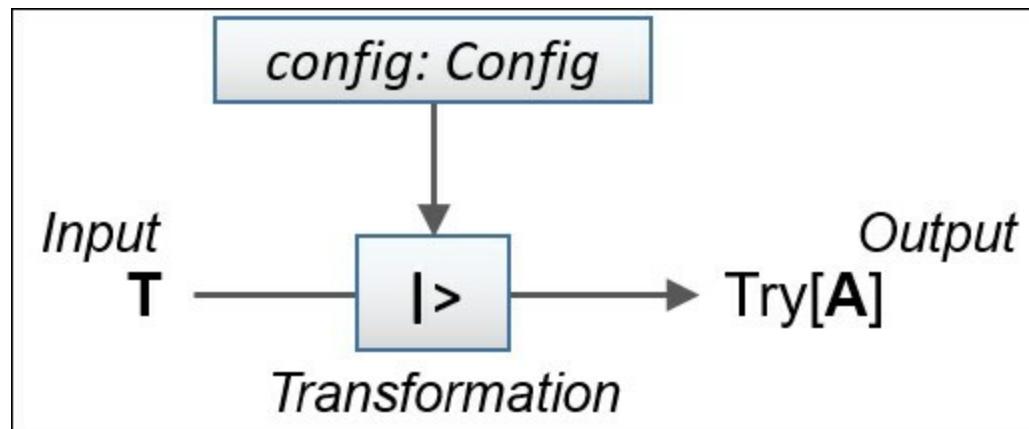
Note

andThen and compose

The reader is invited to implement a compose method which executes the `|>` in a reverse order as `andThen`.

Explicit models

The transformation on a dataset is performed using a model or configuration fully defined by the user, as illustrated in the following diagram:



Visualization of explicit models

The execution of a data transformation may depend on some context or external configuration. Such transformations are defined having the type `ETransform` parameterized by the type `T` of the elements of input collection and the type `A` of element of output collection (line 11). The context or configuration is defined by the trait `config` (line 12).

An explicit transformation is a transformation with the extra capability to use a set of external configuration parameters to generate an output. Therefore, `ETransform` inherits from `ITransform` (line 13):

```
abstract class ETransform[T,A] ( //11
  config: Config//12
) extends ITransform[T,A] { //13
```

```

self =>
def map[B](f: A => B): ETransform[T,B] =
  new ETransform[T,B](config) {
    override def |> : PartialFunction[T, Try[B]] = super.|>
  }

def flatMap[B](f: A => ETransform[T,B]): ETransform[T,B] =
  new ETransform[T,B](config) {
    override def |> : PartialFunction[T, Try[B]] = super.|>
  }

def andThen[B](tr: ETransform[A,B]): ETransform[T,B] =
  new ETransform[T,B](config) {
    override def |> : PartialFunction[T, Try[B]] = super.|>
  }
}

```

The client code is responsible for specifying the type and value of the configuration used by a given explicit transformation. Here are a few examples of configuration classes:

```

Trait Config
case class ConfigInt(iParam: Int) extends Config
case class ConfigDouble(fParam: Double) extends Config
case class ConfigArrayDouble(fParams: Array[Double])
  extends Config

```

Tip

Memory cleaning

Instances of `ITransform` and `ETransform` do not release memory allocated for the input data. The client code is responsible for the memory management of input and output data. However, the method `|>` is to release any memory associated to temporary data structure(s) used for the transformation.

The supervised learning models described in future chapters, such as logistic regression, support vector machines, Naïve Bayes or multilayer perceptron are defined as implicit transformations and implement the `ITransform` trait. Filtering and data processing algorithms such as data extractor, moving averages, or Kalman filters inherit the `ETransform` abstract class.

Note

Immutable transformations

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

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

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

Workflow computational model

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

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

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

Supporting mathematical abstractions

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

1. Declare the variables relevant to the problem.
2. Define a model (equation, algorithm, formulas...) as the solution to the problem.
3. Instantiate the variables and execute the model to solve the problem.

Let's consider the example of the concept of kernel functions (see the *Kernel functions* section of [Chapter 12, Kernel Models and Support Vector Machines](#)), a model that consists of the composition of two mathematical functions, and its potential implementation in Scala.

Step 1 – variable declaration

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

The mathematical formalism is as follows:

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

The Scala implementation is represented here:

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

Step 2 – model definition

The model is defined as the composition of the two functions. The stack of traits `G, F` describes the type of compatible functions that can be composed

using the self-referenced constraint `self: G with F`:

Formalism $h = f \circ g$

The implementation is as follows:

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

Step 3 – instantiation

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

The formalism is as follows:

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

The implementation is as follows:

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

Tip

Lazy value trigger

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

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

Note

Emulation of R

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

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

Composing mixins to build workflow

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

Note

Traits and mixins

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

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

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

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

Understanding the problem

Dependency injection is a fancy name for a reverse look-up and binding to dependencies. Let's consider the simple application that requires data preprocessing, classification and validation.

A simple implementation using traits looks like this:

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

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

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

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

Tip

Mixins linearization [2:6]

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

- Trait B extends A
- Trait C extends A
- Class M extends N with C with B
- The Scala compiler implements the linearization as follows: $A \Rightarrow B \Rightarrow C \Rightarrow N$

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

`PreProcessingWithValidation`:

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

```
}
```

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

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

Tip

Overriding def with val

It is advantageous to override the declaration of a method with a declaration of a value with the same signature. Contrary to a value which is assigned once for all during instantiation, a method may return a different value for each invocation.

A `def` is a proc that can be redefined as a `def`, a `val`, or a `lazy val`. Therefore, you should not override a value declaration with a method with the same signature:

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

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

Defining modules

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

Tip

Use case for describing the cake pattern

It is difficult to build an example of real-world workflow using classes and

algorithms introduced later in the book.

The following simple example is realistic enough to illustrate the different component of the cake pattern.

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

`Etransform`:

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

```
trait Sampling[T,A] { val sampler: ETransform[T, A] }
trait Normalization[T,A] { val normalizer: ETransform[T, A] }
trait Aggregation[T,A] { val aggregator: ETransform[T, A] }
```

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

Tip

Scala traits and Java packages

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

Instantiating the workflow

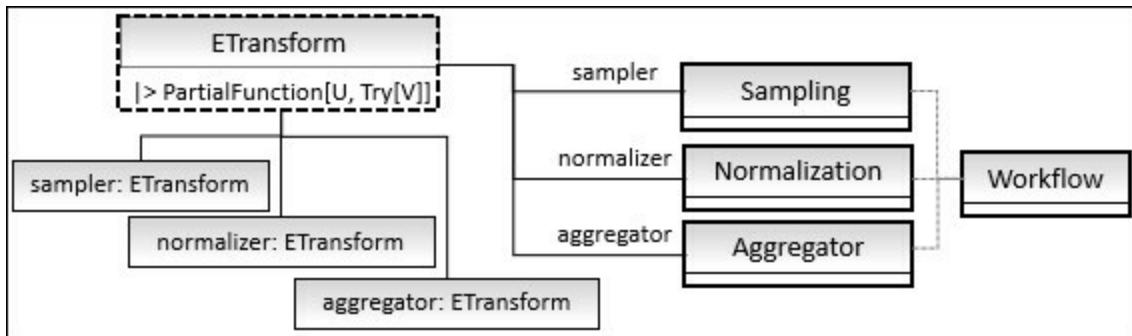
The next step is to write the different modules into a workflow. This is achieved by using the self reference to the stack of the three traits defined in the previous paragraph.

Here's the code:

```
class Workflow[T,U,V,W] {
  self: Sampling[T,U]
    with Normalization[U,V] with Aggregation[V,W] =>

  def |> (t: T): Try[W] = for {
    u <- sampler |> t
    v <- normalizer |> u
    w <- aggregator |> v
  } yield w
}
```

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



UML class diagram of the workflow factory

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

```
Type DblF = Double => Double
type DblVec = Vector[Double]
val samples = 100; val normRatio = 10; val splits = 4

val workflow = new Workflow[DblF, DblVec, DblVec, Int]
  with Sampling[DblF, DblVec]
    with Normalization[DblVec, DblVec]
      with Aggregation[DblVec, Int] {
  val sampler = ??? //1
  val normalizer = ???
  val aggregator = ???
```

```
}
```

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

The first transformation, `sampler`, samples a function `f` with frequency $1/\text{samples}$ over the interval $[0, 1]$. The second transformation, `normalizer`, normalizes the data over the range $[0, 1]$ using the `Stats` class introduced in the next chapter.

The last transformation, `aggregator`, extracts the index of the large sample (value 1.0):

```
val sampler = new ETransform[DblF,DblVec] (ConfigInt(samples)) { //  
  override def |> : PartialFunction[DblF, Try[DblVec]] = {  
    case f: =>  
      Try(Vector.tabulate(samples) (n =>f(1.0*n/samples))) //5  
  }  
}
```

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

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

```
val normalizer = new ETransform[DblVec, DblVec] (  
  ConfigDouble(normRatio)  
) {  
  override def |> : PartialFunction[DblVec,Try[DblVec]] = {  
    case x: DblVec if(x.size > 0) =>  
      Try((Stats[Double](x)).normalize)  
  }  
}  
  
val aggregator = new ETransform[DblVec, Int] (ConfigInt(splits)) {  
  override def |> : PartialFunction[DblVec, Try[Int]] = {  
    case x:DblVec if(x.size> 0) =>  
      Try((0 until x.size).find(x(_)==1.0).getOrElse(-1))  
  }  
}
```

```
}
```

The instantiation of the transformation function follows the template described in the *Monadic data transformation* section of [Chapter 1, Getting Started](#).

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

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

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

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

Note

Mixin composition for `ITransform`

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

Modularizing

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

- Select the appropriate workflow as a sequence of module or tasks according to the objective of the execution (regression, classification, clustering...)
- Select the appropriate algorithm to fulfill a task according to the quality of the data (noisy, incomplete, ...)
- Select the appropriate implementation of the algorithm according to the environment (distributed with high latency network, single host...):

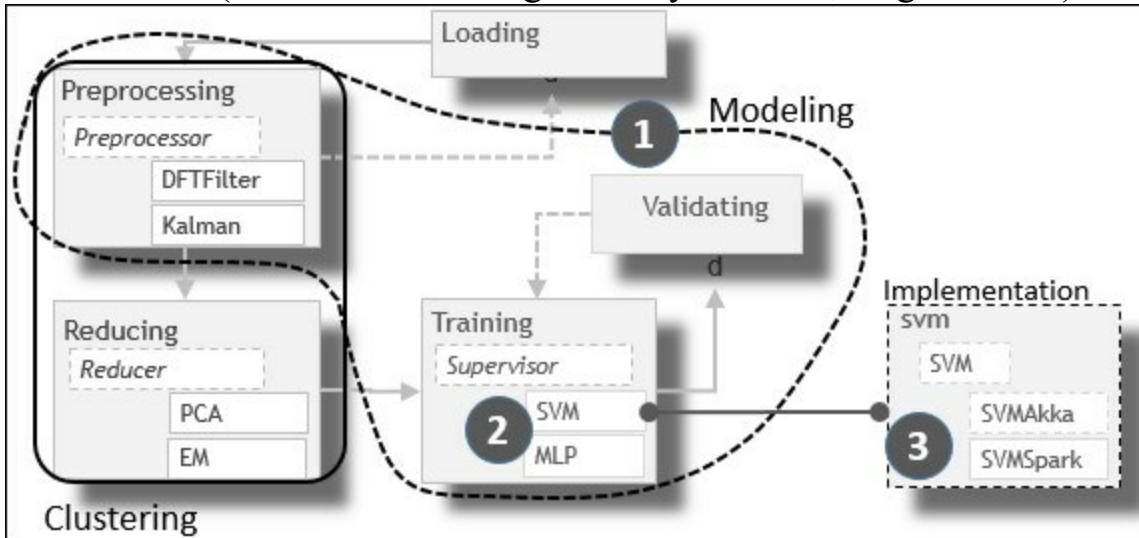


Illustration of the dynamic creation of workflow from modules/traits

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

```
trait PreprocessingModule[T] {
```

```

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

class ExpMovingAverage[T : ToDouble](p: Int) //9
(implicit num: Numeric[T]) extends Preprocessor[T] {

    val expMovingAvg = filtering.ExpMovingAverage[T](p) //10
    val pfn = expMovingAvg |> //11
    override def execute(x: Vector[T]): Try[DblVec] = pfn(x)
}

class DFTFilter[T : ToDouble](
    fc: Double,
    g: (Double, Double) => Double
) extends Preprocessor[T] //12

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

```

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

The first filtering algorithm of type `ExpMovingAverage` implements the `Preprocessor` trait and overrides the `execute` method (line 9). The class declares the algorithm but delegates its implementation to a class with an identical signature, `org.scalaml.filtering.ExpMovingAverage` (line 10). Data of generic type `T` is automatically converted into a vector of `Double` using a context bound with the syntax `T: ToDouble`. The context bound is implemented by the following trait:

```
Trait ToDouble[T] { def apply(t: T): Double }
```

The partial function returned from the `|>` method is instantiated as a value, `pfn`, so it can be applied multiple times (line 11). The same design pattern is used for the discrete Fourier transform filter (line 12).

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

Note

Filtering algorithms

The filtering algorithms used to illustrate the concept of modularization in the context of the cake pattern are described in detail in [Chapter 3, Data Pre-processing](#).

Profiling data

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

Immutable statistics

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

Note

Mean and variance

Arithmetic mean:

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

Variance:

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

Variance adjusted for sampling bias:

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

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

```
class Stats[T: ToDouble] (values: Vector[T])
extends MinMax[T] (values) {

  val zero = (0.0, 0.0)
  val sums = values.:/:(zero)((s, x) => (s._1 + x, s._2 + x*x)) //1
  lazy val mean = sums._1/values.size //2
  lazy val variance =
    (sums._2 - mean*mean*values.size)/(values.size-1)
  lazy val stdDev = sqrt(variance)
```

```
...  
}
```

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

Z-score and Gauss

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

Note

Gaussian distribution

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

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

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

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

Note

Z-score normalization

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

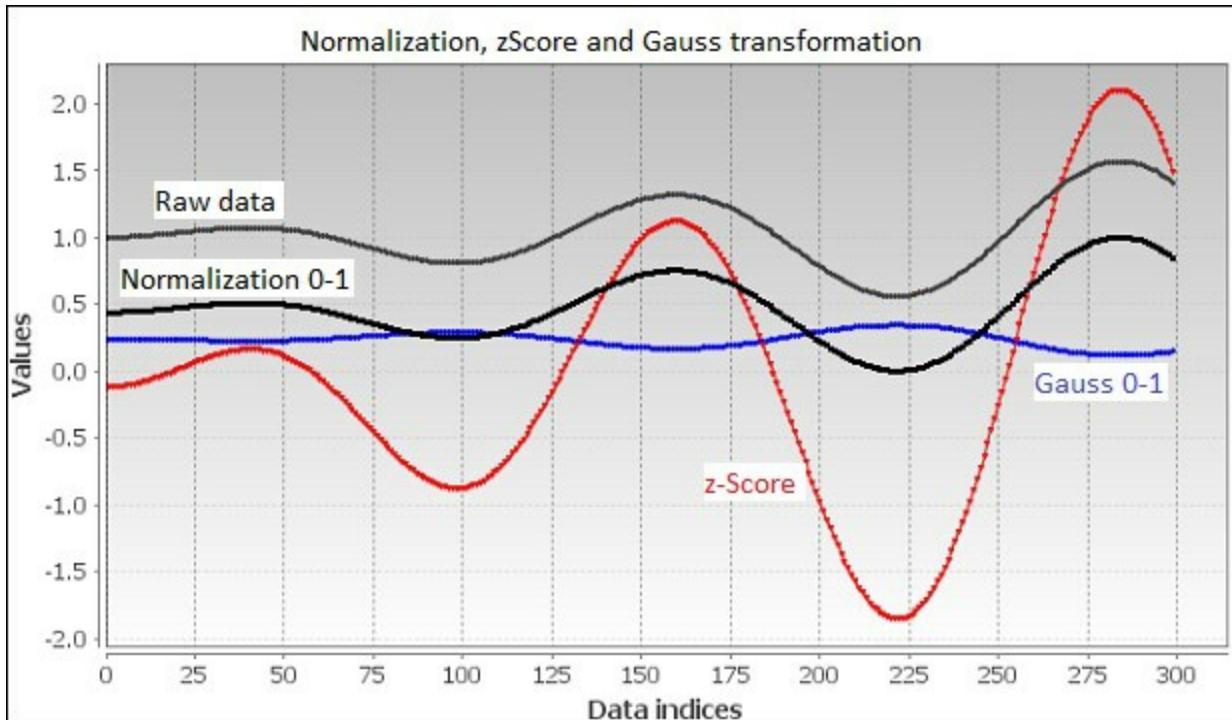
$$z_i = \frac{x_i - \mu}{\sigma}$$

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

Stats:

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

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



Comparative analysis of linear, Gaussian, and Z-score normalization

Assessing a model

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

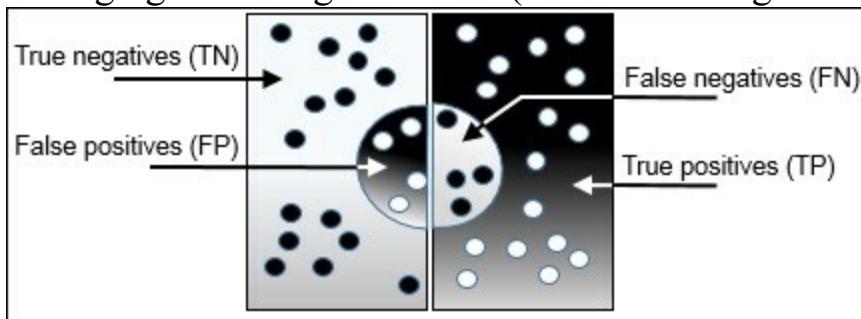
Validation

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

Key quality metrics

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

- **True Positives (TPs):** These are observations that are correctly labeled as belonging to the positive class (white dots on dark background)
- **True Negatives (TNs):** These are observations that are correctly labeled as belonging to the negative class (black dots on light background)
- **False Positives (FPs):** These are observations incorrectly labeled as belonging to the positive class (white dots on dark background)
- **False Negatives (FNs):** These are observations incorrectly labeled as belonging to the negative class (dark dots on light background)



Categorization of validation results

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

and F and G measures as follows:

- **Accuracy:** 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_1 -measure or F_1 -score:** Represented as F_1 , this measure strikes a balance between precision and recall. It is computed as the harmonic mean of the precision and recall with values ranging between 0 (worst score) and 1 (best score).
- **F_n score:** Represented as F_n , this is the generic F-scoring method with an arbitrary degree n .
- **G measure:** Represented as G , this is like the F-measure but is computed as the geometric mean of precision p and recall r .

Note

Precision, recall, and F_1 -score

M3: Accuracy ac , precision, p , recall r , F_1 , F_n , and G scores:

$$ac = \frac{tp + tn}{tp + tn + fp + fn} \quad p = \frac{tp}{tp + fp} \quad r = \frac{tp}{tp + fn}$$

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

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

- F-score validation for binomial (two classes) classification (that is,

- positive and negative outcome)
- F-score validation for multinomial (more than two classes) classification

F-score for binomial classification

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

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

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

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

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

```
class BinaryValidation[T: ToDouble] (
  expected: Vector[Int],
  xt: Vector[Array[T]]) (predict: Array[T] => Int) (
  extends AValidation[T](expected, xt)(predict) //1

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

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

  def compute(n: Label): Double =
    1.0 / (1.0 + counters(n) / counters(TP())))

```

```

def classify(predicted: Int, expected: Int): VLabel = //5
  if(expected == predicted)
    if(expected == POSITIVE) TP() else TN()
  else if(expected == POSITIVE) FP() else FN()
}

```

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

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

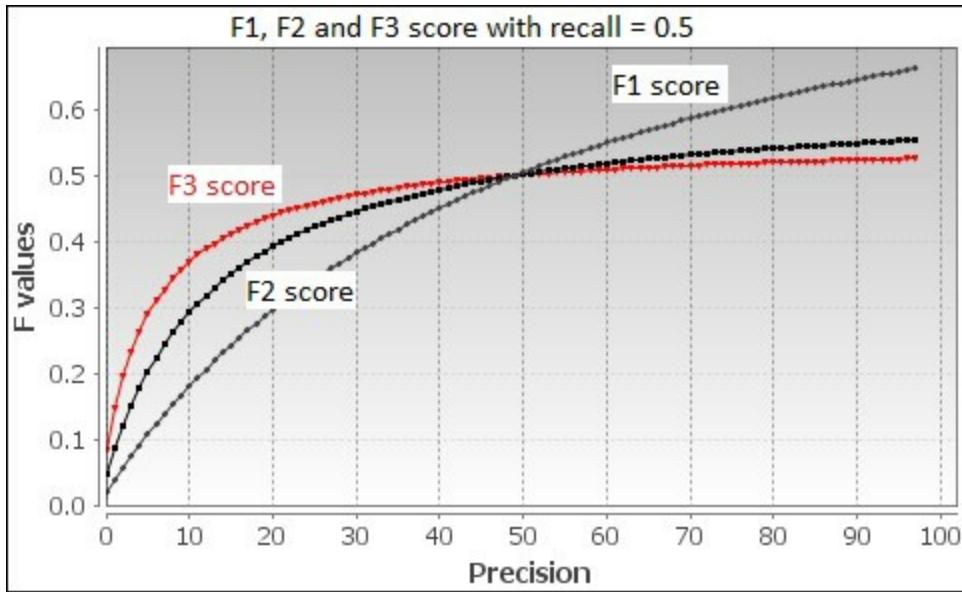
The validation labels of type `VLabel` are defined as sealed case classes (line 6). Although Scala supports enumeration in a fashion similar to Java, Scala programmers prefer case classes and pattern matching as an alternative to extending `Enumeration` type:

```

sealed trait Vlabel  //6
case class TP() extends Vlabel
case class TN() extends Vlabel
case class FP() extends Vlabel
case class FN() extends VLabel

```

The F-score formula with higher cardinality F_n with $n > 1$ favors precision over recall as illustrated in the following chart:



Comparative analysis of impact of precision on F1, F2, and F3 score for a given recall

Tip

Multiclass scoring

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

F-score for multinomial classification

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

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

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

We will use the macro formulas from now on.

Note

Macro formulas for multinomial precision and recall

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

$$p^* = \frac{1}{c} \sum_{i=0}^{c-1} \frac{tp_i}{tp_i + fp_i} \quad r^* = \frac{1}{c} \sum_{i=0}^{c-1} \frac{tp_i}{tp_i + fn_i}$$

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

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

Confusion matrix for six-class classification

The multinomial validation class, `MulticlassValidation`, takes four

parameters:

- The expected class index with value 0 for negative outcome and 1 for positive outcome (line 6)
- The set of observations, `xt` used for validating the model (line 7)
- The number of classes in the model
- The predictive function, `predict` classifies observations (line 7):

```
class MulticlassValidation[T: ToDouble] (
    expected: Vector[Int],
    xt: Vector[Array[T]],
    nClasses: Int) (predict: Array[T] => Int)
extends Validation{ //7

    val confusionMatrix: Matrix[Int] = //8
        labeled./:(Matrix[Int](nClasses)) {
            case (m, (x,n)) => m + (n, predict(x), 1) //9
        }

    lazy val (precision, recall): DblPair = //10
        (0 until classes)./:(0.0,0.0)((s, n) => {
            val tp = confusionMatrix(n, n) //11
            val fn = confusionMatrix.col(n).sum-tp //12
            val fp = confusionMatrix.row(n).sum-tp //13
            (s._1 + tp.toDouble/(tp + fp)/nClasses,
             s._2 +tp.toDouble/(tp + fn)/nClasses)
        })

    def score: Double =
        2.0*precision*recall/(precision+recall)
}
```

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

As stated in the introduction of the section, we use the macro definition of the precision and recall (line 10). The count of true positive, `tp`, for each class corresponds to the diagonal element of the confusion matrix (line 11). The count of false negatives, `fn`, for a class is computed as the sum of the counts

for all the predicted classes (column values), given an expected class except the true positive class (line 12). The count of false positives, f_p , for a class is computed as the sum of the counts for all the expected classes (row values), given a predicted class except the true positive class (line 13).

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

Area under the curves

A more sophisticated measurement of the quality of a model is known as the area under the curves. There are two commonly used measures:

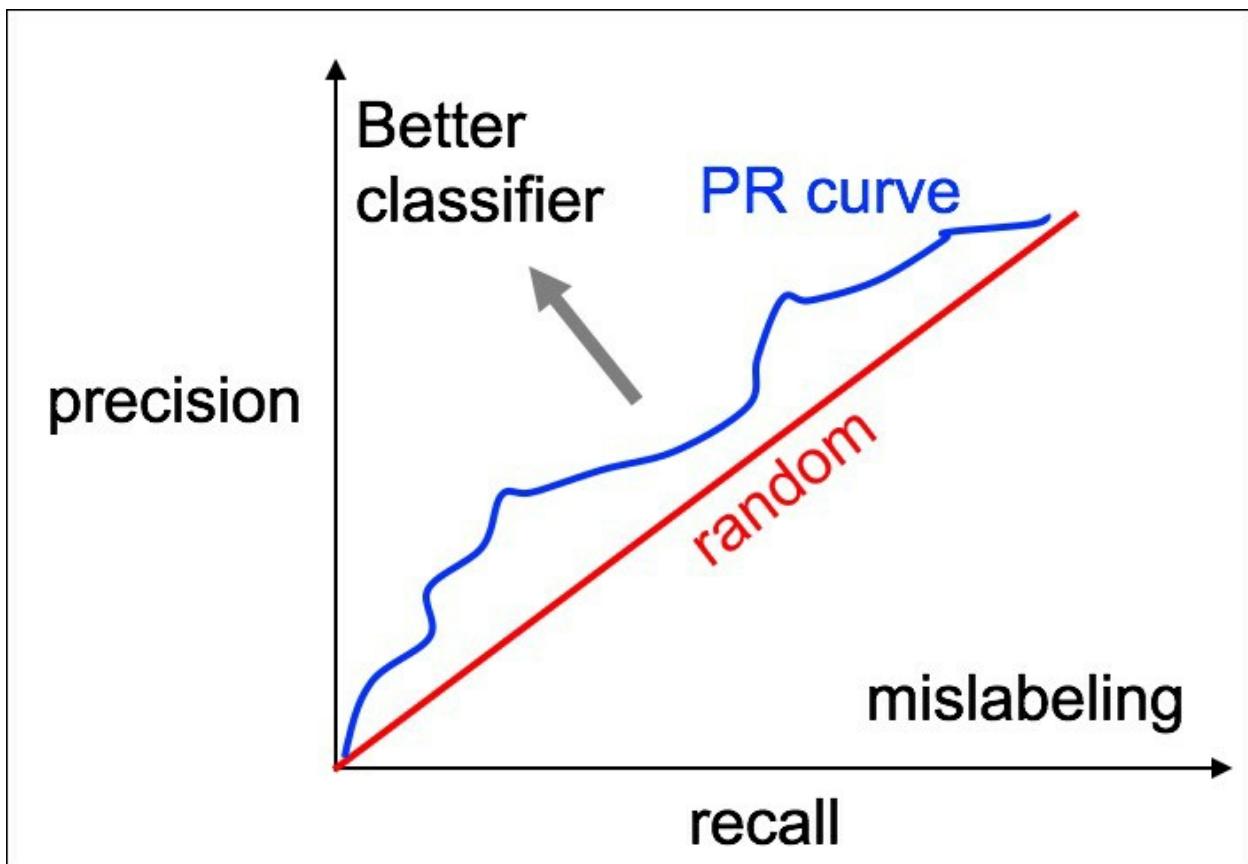
- **Area under the precision-recall curve (AuPRC)**
- **Area under the receiver operating characteristics (AuROC)**

Area under PRC

The previous section introduces the concepts of precision and recall and their application to the computation of the F_1 score. There is a more appealing application of precision and recall that provides scientists with a more accurate evaluation and a convenient visualization of the performance of a binary classifier.

Let's consider two classes C0 and C1. The classification of a new observation is accomplished by setting a threshold on the predicted value (or probability the observation belongs to class C1, for instance). The higher the threshold, that the more selective the algorithm is to assign a new observation to the proper class.

The **precision-recall curve (PRC)** is generated by plotting the pair of (precision, recall) values on an xy graph and varying the threshold between 0 and 1.0:



Visualization of the performance of a binary classifier using precision-recall curve

The performance of the classifier improves as the precision increases and the recall decreases, toward the upper-left corner of the graph. A pure random process such as flipping a coin has similar values for the precision and recall (random line). An unusual case for which the recall is very high and precision close to zero is most often associated with an error in labeling observations prior to training.

Note

M5: AuPRC

$$auPRC = 0.5 + \frac{1}{N} \sum_{i=0}^{N-1} (p_i - r_i)$$

Let's look at a simple implementation of the AuPRC, after a list of validation values, `binaryValidations`, has been generated through classification using a variable threshold:

```
def auPRC[T: ToDouble] (
  binValidations: List[BinaryValidation]
) : Double = binValidations./:(0.5) (
  (s, valid) => s + valid.precision - valid.recall
)/binaryValidations.size
```

The performance of the classifier is quantified by computing the area under the PR curve (integral value). A value of 1.0 signifies a perfect classifier and a value of 0.5 represents a pure random process.

Area under ROC

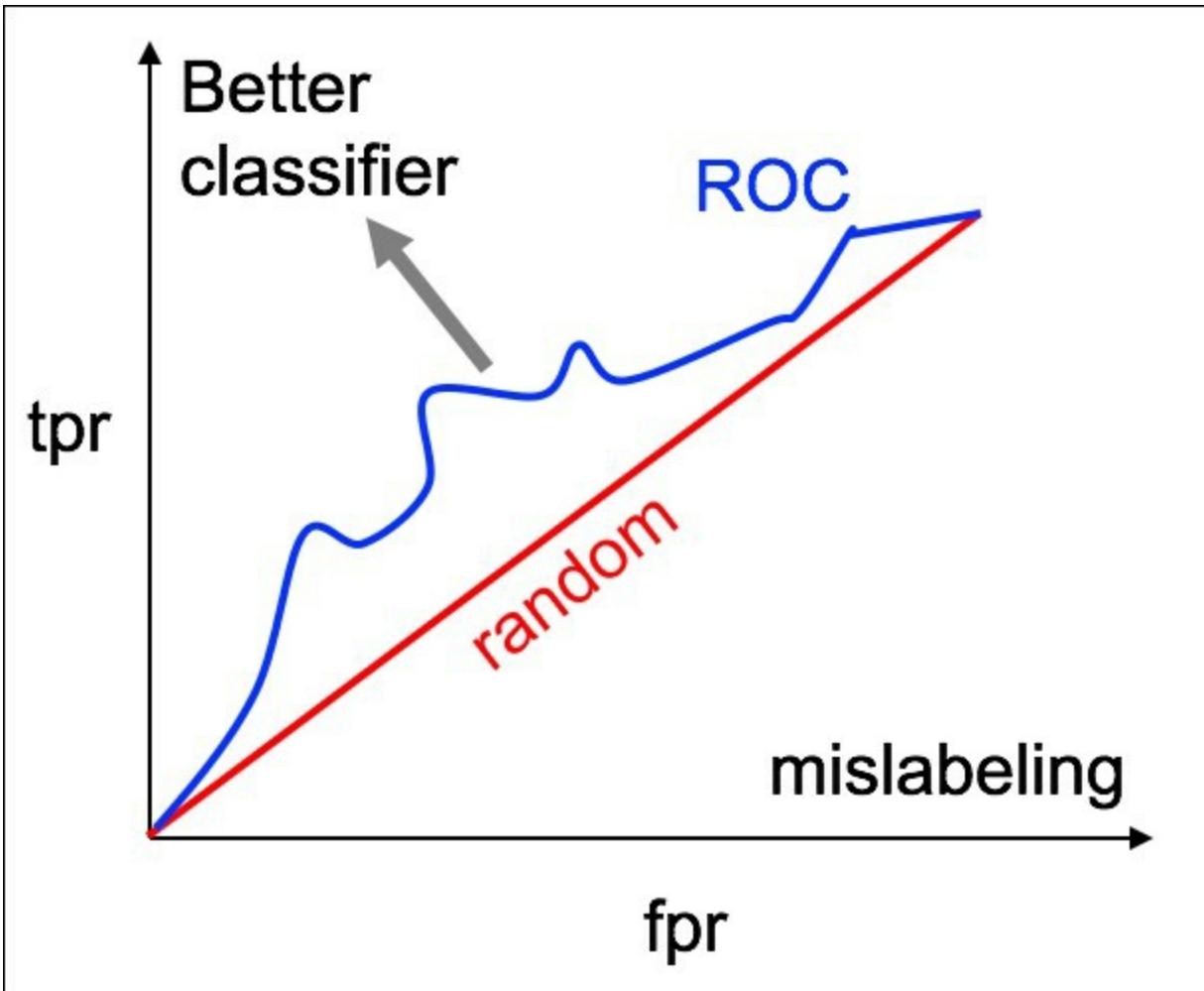
Receiver Operating Characteristics (ROC) is a very convenient tool that visualizes the performance of a binary classifier [2:7]. The curve or plot is generated by computing the **true positive rate (TPR)** and the **false positive rate (FPR)** values for different classification threshold on a validation set.

Note

M5: True positive and false positive rates

$$trp = \frac{tp}{tp + fn} \quad fpr = \frac{fp}{fp + tn}$$

The methodology that generates the ROC is identical to the process of creating the AuPRC, by plotting the pair of TPR and FPR on an *xy* graph and varying the threshold between 0 and 1.0:



Visualization of the performance of a binary classifier using ROC

The performance of the binary classifier is measured by computing the area or integral under the ROC in a similar fashion as the computation of the AuPRC.

Cross-validation

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

One-fold cross-validation

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

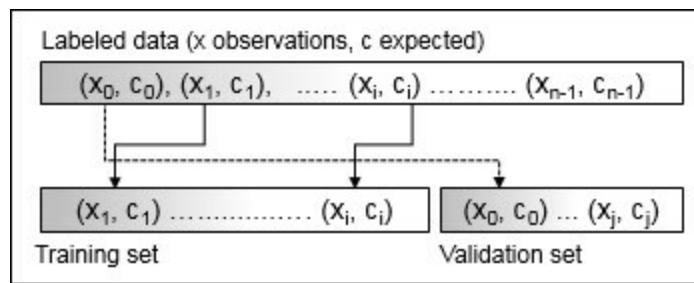


Illustration of the generation of a one-fold validation set

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

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

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

```
type LabeledData[T] = Vector[(Array[T], Int)]
```

```

class OneFoldValidation[T: ToDouble] (
  xt: Vector[Array[T]],
  expected: Vector[Int],
  ratio: Double) { //14

  val (trainSet, validSet): (LabeledData[T], LabeledData[T]) //15
}

```

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

```

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

  val valSz = labeledData.size - trainingSize
  val adjValSz = if(valSz < 2) 1
  else if(valSz >= labeledData.size) labeledData.size - 1
  else valSz //18

  val ordLabeledData = labeledData
    .map( _, nextDouble) //19
    .sortWith( _._2 < _._2).unzip._1//20

  (ordlabeledData.takeRight(adjValSz),
   ordlabeledData.dropRight(adjValSz)) //21
}

```

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

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

K-fold cross-validation

The data scientist creates K training-validation datasets by selecting one of the groups as a validation set then combining all remaining groups into a training set, as illustrated in the next diagram. The process is known as **K-fold cross-validation** [2:8]:



Illustration of the generation of a K-fold cross-validation set

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

Bias-variance decomposition

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

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

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

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

Note

M5: Variance and bias for a true model, θ :

$$var\hat{\theta} = E\left[\left(\hat{\theta} - E\left[\hat{\theta}\right]\right)^2\right] \quad bias\hat{\theta} = \hat{\theta} - \theta$$

M6: Mean square error:

$$MSE = var(\tilde{\theta}) + bias(\tilde{\theta})$$

Let's illustrate the concept of bias, variance, and mean square error with an example. At this stage, you have not been introduced to most of the machines

learning techniques. Therefore, we create a simulator to illustrate the relation between bias and variance of a classifier. The components of the simulation are as follows:

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

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

Note

Empirical estimation of overfitting

Overfitting models are specific to the training set. The prediction over a validation set will have a low bias and a high variance.

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

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

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

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

```
} }
```

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

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

The method accumulate returns a tuple (variance, bias) for each of the models f (line 25). The model candidates are defined by the following family of single variables for values $n = 1, 2, 4$:

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

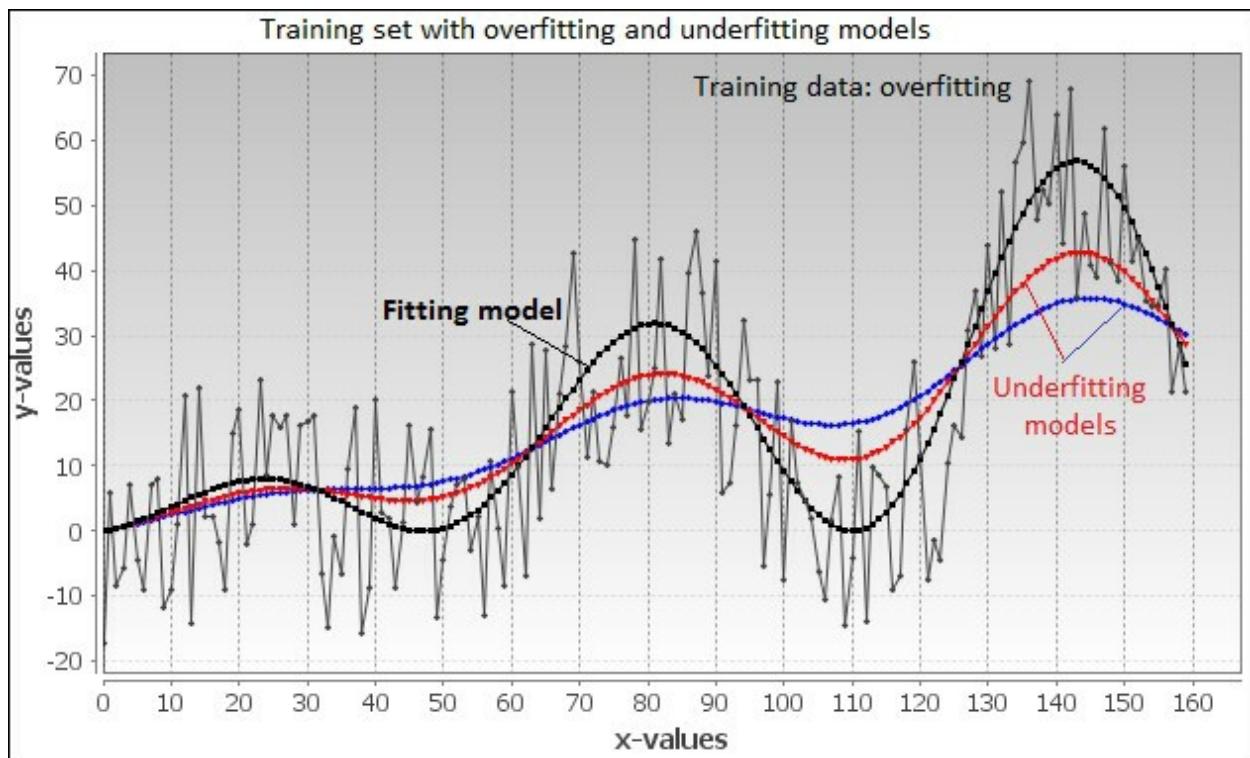
The **target** model (line 26) and the **models** (line 27) belong to the same family of single variable functions:

```
val template = (x: Double, n : Int) =>
  0.2*x*(1.0 + sin(x*0.1)/n)
val training = (x: Double) => {
  val r1 = 0.45*(nextDouble-0.5)
  val r2 = 38.0*(nextDouble - 0.5) + sin(x*0.3)
  0.2*x*(1.0 + sin(x*0.1 + r1)) + r2
}

val target = (x: Double) => template(x, 1) //26

val models = List[(DblF, String)] ( //27
  ((x: Double) => template(x, 4), "Underfit1"),
  ((x: Double) => template(x, 2), "Underfit2"),
  ((x : Double) => training(x), "Overfit"),
  (target, "target"),
)
val evaluator = new BiasVariance[Double](target, 200)
evaluator.fit(models.map(_._1)) match { /* ... */ }
```

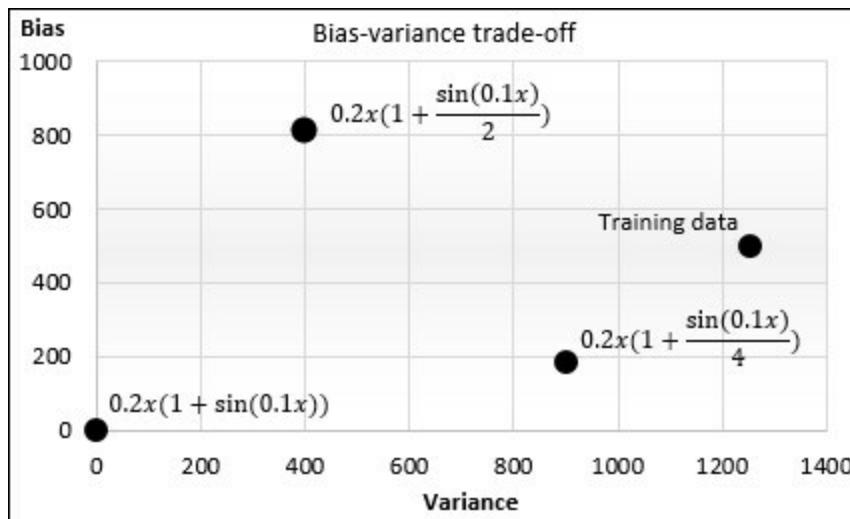
The JFreeChart library is used to display the training dataset and the models:



Fitting models to a dataset

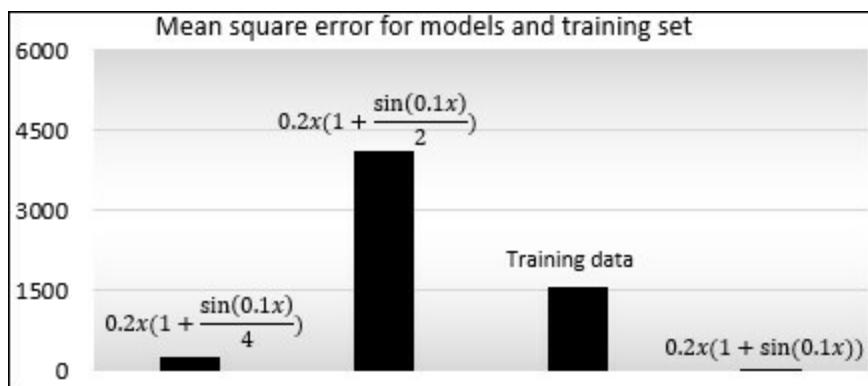
The model that replicates the training data overfits. The models for which the sine component has a lower amplitude underfits.

The **variance-bias trade-off** for the different models and the training data is illustrated with the following scatter chart:



Scatter plot for the bias-variance trade-off for four models, one duplicating the training set.

The variance of each of the smoothing or approximating models is lower than the variance of the training set. As expected, the target model, $0.2x(1 + \sin(0.1x)/2)$, has no bias and no variance. The training set has a very high variance because it overfits any target model. The last chart compares the mean square error between each of the models, the training set and the target model:



Comparative mean square error for four models

Note

Evaluating bias and variance

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

Overfitting

You can apply the methodology presented in the example to any classification and regression models. The list of models with low variance includes constant functions 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 low bias, while linear regression applied to non-linear data has a higher bias [2:9].

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

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

However, there are well-proven solutions to reduce overfitting [2:10]:

- Increasing the size of the training set whenever possible
- Reducing noise in labeled observations using smoothing and filtering techniques
- Decreasing the number of features using techniques such as principal components analysis, as described in the *Principal components analysis* section of [Chapter 5, Dimension Reduction](#)
- Modeling observable and latent noisy data using Kalman or auto-regressive models as described in [Chapter 3, Data Pre-processing](#)
- Reducing inductive bias in training set by applying cross-validation
- Penalizing extreme values for some of the model's features using regularization techniques, as described in the *Regularization* section of [Chapter 9, Regression and Regularization](#)

Summary

In this chapter, we established the framework for the different data processing units that will be introduced in this book. There is a very good reason why the topics of model validation and overfitting are treated 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 following topics:

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

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

Chapter 3. Data Preprocessing

Real-world observations are usually noisy and inconsistent, with missing data. No classification, regression, or clustering model can extract reliable information from data that has not been cleansed, filtered, or analyzed.

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

- Identify flaws in raw input data
- Provide unsupervised or supervised learning with a clean and reliable dataset

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 do the following:

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

Time series in Scala

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

Context bounds

The algorithms presented in this chapter are applied to time series with a single variable of type `Double`. Therefore we need a mechanism to convert implicitly a given type `T` to a `Double`. Scala provides developers with such design: context bounds [3:1]:

```
trait ToDouble[T] { def apply(t: T): Double }
implicit val str2Double = new ToDouble[String] {
    def apply(s: String): Double = s.toDouble
}
```

Types and operations

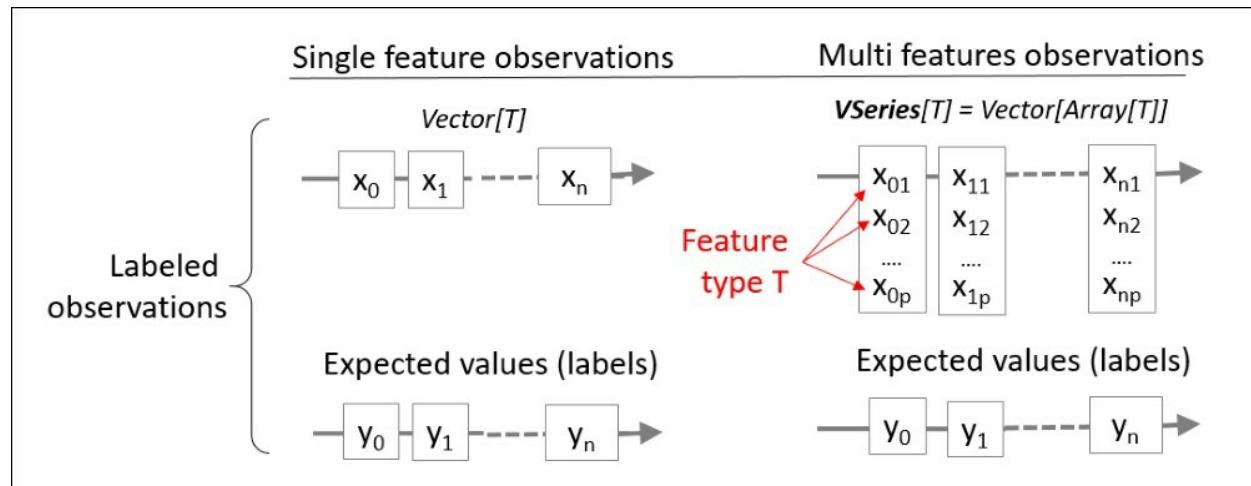
The *Defining primitives types* section under *Source code* in [Chapter 1](#), *Getting Started* introduced the types for time series of single variable, `vector[T]`, and multiple variables, `vector[Array[T]]`.

A time series of observations is a vector (type `vector`) of observation elements:

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

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

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



Visualization of the single feature and multi features observations

The two generic types for time series, `vector[T]` and `vector[Array[T]]`, will be used as the two primary classes for input data, from now on.

Note

Labeled observations structure:

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

The class `Stats` introduced in the *Profiling data* section of [Chapter 2, Data Pipelines](#), implements some basic statistics and normalization for single variable observations.

Let us create a singleton, `TSeries`, to compute statistics and normalize multi-dimensional observations:

```
type DblVec = Vector[Double]

Object TSeries {
    def zipWithShift[T](v: Vector[T], n: Int): Vector[(T, T)] =
        v.drop(n).zip(v.view.dropRight(n)) //1

    def statistics[T: ToDouble](vs: Vector[Array[T]]):
        Vector[Stats[T]] = vs.transpose.map( Stats[T](_)) //2

    def normalize[T: ToDouble]( //3
        vs: Vector[Array[T]], low: Double, high: Double
    )(implicit ordering: Ordering[T]): Try[DblVec] =
        Try(Stats[T](vt).normalize(low, high) )
    ...
}
```

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

Here is a partial list of other commonly used operators:

- Create a time series of type `Vector[(T, T)]` by zipping two vectors, `x` and `y`:

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

- Split a single- or multi-dimensional time series, xv , into two time series at index n :

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

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

```
def zScore[T: ToDouble] (xt: Vector[T]): Try[DblVec]
```

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

```
def delta(x: DblVec): DblVec
```

- Compute the sum of squared error between two arrays x, z :

```
def sse[T: ToDouble] (x: Array[T], z: Array[T]): Double
```

- Compute the statistics for each features of a multi-dimensional time series:

```
def statistics[T: ToDouble] (xt: Vector[Array[[T]]]):  
  Vector[Stats[T]]
```

Tip

Magnet pattern:

Some operations on time series may have a large variety of input and output types. Scala and Java support method overloading, which has the following limitations:

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

The magnet pattern used in the implementation of the `Transpose` and `Differential` operators remedies these limitations.

Transpose operator

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

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

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

```
implicit def vSeries2Matrix[T: ClassTag] (from: Vector[Array[T]]) =
    new Transpose { type Result = Array[Array[T]] //6
        def apply(): Result = from.toArray.transpose
    }
```

The first implicit `vSeries2Matrix` transposes a time series of type `Vector[Array[T]]` into a matrix with elements of type `T` (line 6). The generic `transpose` method is written as follows:

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

Differential operator

The second candidate to the magnet pattern is the computation of the differential in a time series. The purpose is to generate the time series

$\{x_{t+1} - x_t\}$ from a time series $\{x_t\}$:

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

The trait `Difference` allows us to compute the differential of time series with arbitrary element types. For instance, the differential on a one-dimensional

vector of type `Double` is defined by the following implicit conversion:

```
implicit def vector2Double[T] (x: DblVec) = new Difference[T] {  
    type Result = Vector[T]  
    def apply(f: (Double, Double) => T): Result = //7  
        zipWithShift(x, 1).collect{case(next,prev) =>f(prev,next)}  
}
```

The constructor `apply()` takes one argument: the user-defined function `f` that computes the difference between two consecutive elements of the time series (line 7). The generic difference method is as follows:

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

Here are some of the predefined differential operators on time series for which the output of the operator has the type `Double` (line 8), `Int` (line 9), and `Boolean` (line 10):

```
val diffDouble = (x: Double,y: Double) => y -x //8  
val diffInt = (x: Double,y: Double) => if(y > x) 1 else 0 //9  
val diffBoolean = (x: Double,y: Double) => (y > x) //10
```

Lazy views

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

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

Let's look at the data transformation, `aggregator`, introduced in the *Instantiating the workflow* section under *Workflow computational model* in [Chapter 2, Data Pipelines](#). There is no need to allocate the entire set of `x.size` of elements: the higher-order method, `find` may exit after only a few elements have been read (line 11):

```
val aggregator = new ETransform[Int, Int](ConfigInt(splits)) {  
    override def |> : PartialFunction[Int, Try[Int]] = {  
        case x: U if(!x.isEmpty) =>  
            Try(Range(0, x.size).view.find(x(_) == 1.0).get) //11  
    }  
}
```

Tip

Views, iterators, and streams:

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

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

Moving averages

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

Note

Averaging reducing function:

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

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

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

Simple moving average

Simple moving average is the simplest form of the moving averaging algorithms [3:2]. 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:

Note

Simple moving average:

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

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

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

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

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

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

```
trait MovingAverage[T]
```

We use the generic type `Vector[T]` and the data transform with explicit configuration `ETransform` introduced in the *Explicit models* section under

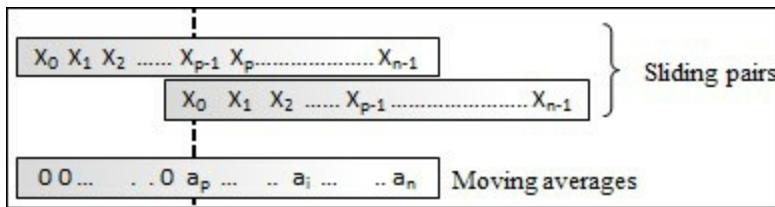
Monadic data transformation in [Chapter 2, Data Pipelines](#) to implement the simple moving average, SimpleMovingAverage:

```
class SimpleMovingAverage[@specialized(Double) T: ToDouble] (
    period: Int)(implicit num: Numeric[T] //1
) extends Etransform[Vector[T], DblVec](ConfigInt(period))
    with MovingAverage[T] {
    val zeros = Vector.fill(0.0)(period-1)

    override def |> : PartialFunction[Vector[T], Try[DblVec]] = {
        case xt: Vector[T] if( xt.size >= period ) => {
            val (first, second) = xt.splitAt(period) //2
            val slider = xt.take(xt.size - period).zip(second) //3

            val c = implicitly[ToDouble[T]]
            val zero = first.sum/period //4
            Try( zeros ++ slider.scanLeft(zero) { //5
                case (s, (x,y)) => s + (c(y) - c(x))/period })
        }
    }
}
```

The class is parameterized for the type T of elements of the input time series: *we cannot make any assumption regarding the type of input data*. The type of the elements of the output time series is `Double`. The implicit instantiation of the class `Numeric[T]` is required by the arithmetic operators `sum` and `/` (line 1). The implementation has a few interesting elements. First, the set of observations is split with the first period observations (line 2) and the index in the resulting clone instance is shifted by p observations before being zipped with the original to the array of pair of values: `slider` (line 3):



Sliding algorithm to compute moving averages

The average value is initialized with the mean value of the first `period` data points (line 4). The first `period` values of the trends are initialized as zero (line 5). The method concatenates the initial null values and the computed average values to implement the formula M2 (line 5).

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 [3:3]. The weights α_j are assigned to each of the last p data point x_j and normalized by the sum of the weights.

Note

Weighted moving average:

M3: Weighted moving average of a series $\{x_t\}$ with p normalized weights distribution $\{\alpha_j\}$:

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

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

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

```
class WeightedMovingAverage[@specialized(Double) T: ToDouble] (
    weights: Features)(implicit num: Numeric[T])
) extends SimpleMovingAverage[T](weights.length) { //7

  override def |> : PartialFunction[Vector[T], Try[DblVec]] = {
    case xt: Vector[T] if(xt.size >= weights.length) => {
      val smoothed = (config to xt.size).map( i =>
        xt.slice(i - config, i).zip(weights).map { // 8
          case(x, w) =>
            implicitly[ToDouble[T]].apply(x) * w}.sum //9
    }
  }
}
```

```

        )
    Try(zeros ++ smoothed) //10
}
}
}

```

The computation of the weighted moving average is a bit more involved than the simple moving average. Therefore, we specify the generation of byte code dedicated to `Double` type using the `@specialized` annotation. The weighted moving average inherits the class `SimpleMovingAverage` (line 7) and therefore implements the explicit transformation `ETransform` for a configuration of `weights`, with input observations of type `Vector[T]` and output of type `DblVec`. The implementation of formula M3 generates a smoothed time series by slicing (line 8) the input time series and then computing the inner product of `weights` and the slice of the time series (line 9).

As with the simple moving average, the output is the concatenation of the initial `weights.size` null values, `zeros`, and the `smoothed` data (line 10).

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 [3:4].

Note

Exponential moving average:

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

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

$$\tilde{x}_0 = x_0$$

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

The implementation of the `ExpMovingAverage` class is rather simple. The constructor has a single argument, α (decay rate) (line 11):

```
class ExpMovingAverage[@specialized(Double) T: ToDouble] (
    alpha: Double //11
) extends ETransform[Vector[T], DblVec](ConfigDouble(alpha))
    with MovingAverage[T] { //12

    override def |> : PartialFunction[Vector[T], Try[DblVec]] = {
        case xt: Vector[T] if( xt.size > 0) => {
            val c = implicitly[.ToDouble[T]]
            val alpha_1 = 1-alpha
            var y: Double = data(0)
            Try( xt.view.map(x => {
                val z = c.apply(x)*alpha + y*alpha_1; y = z; z }))
        } //13
    }
}
```

The exponential moving average implements the `ETransform` with an input of type `Vector[T]` and an output of type `DblVec` (line 12). The method `|>` applies the formula M4 to all observations of the time series within a `map` (line 13).

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

```
def apply[T: ToDouble](p: Int): ExpMovingAverage[T] =
  new ExpMovingAverage[T](2 / (p + 1))
```

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

```
import YahooFinancials._
type DblSeries = Vector[Array[Double]]
val hp = p >> 1
val w = Array.tabulate(p)(n =>
  if(n == hp) 1.0 else 1.0 / (Math.abs(n - hp) + 1)) //14
val sum = w.sum
val weights = w.map { _ / sum } //15

val dataSrc = DataSource(s"$RESOURCE_PATH$symbol.csv", false) //16
val pfnSMvAve = SimpleMovingAverage[Double](p) |> //17
val pfnWMvAve = WeightedMovingAverage[Double](weights) |>
val pfnEMvAve = ExpMovingAverage[Double](p) |>

for {
  price <- dataSrc.get(adjClose) //18
  if(pfnSMvSve.isDefinedAt(price) )
    sMvOut <- pfnSMvAve(price) //19
  if(pfnWMvSve.isDefinedAt(price)
    eMvOut <- pfnWMvAve(price)
  if(pfnEMvSve.isDefinedAt(price)
    wMvOut <- pfnEMvAve(price)
} yield {
  val dataSink = DataSink[Double](s"$OUTPUT_PATH$p.csv")
  val results = List[DblSeries](price, sMvOut, eMvOut, wMvOut)
```

```
    dataSink |> results //20
}
```

Tip

isDefinedAt:

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

The coefficients for the weighted moving average are generated (line 14) and normalized (line 14). The trading data regarding the ticker symbol, BAC, is extracted from the Yahoo finances CSV file (line 16) `YahooFinancials` using the `adjClose` extractor (line 17). The next step is to initialize the partial functions `pfnSMvAve`, `pfnWMvAve`, and `pfnEMvAve` related to each of the moving averages (line 18). The invocation of the partial functions with `price` as argument generated the three smoothed time series (line 19).

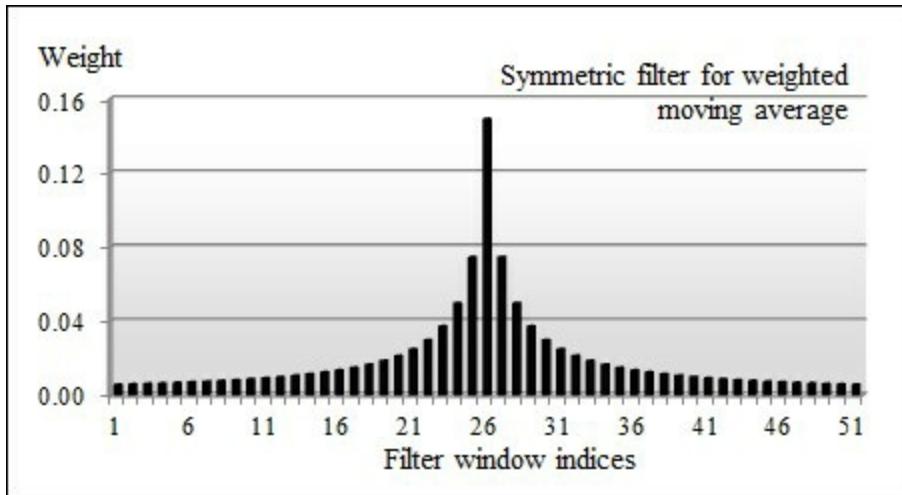
Finally, a `DataSink` instance formats and dumps the results into a file (line 20).

Tip

Implicit postfixOps:

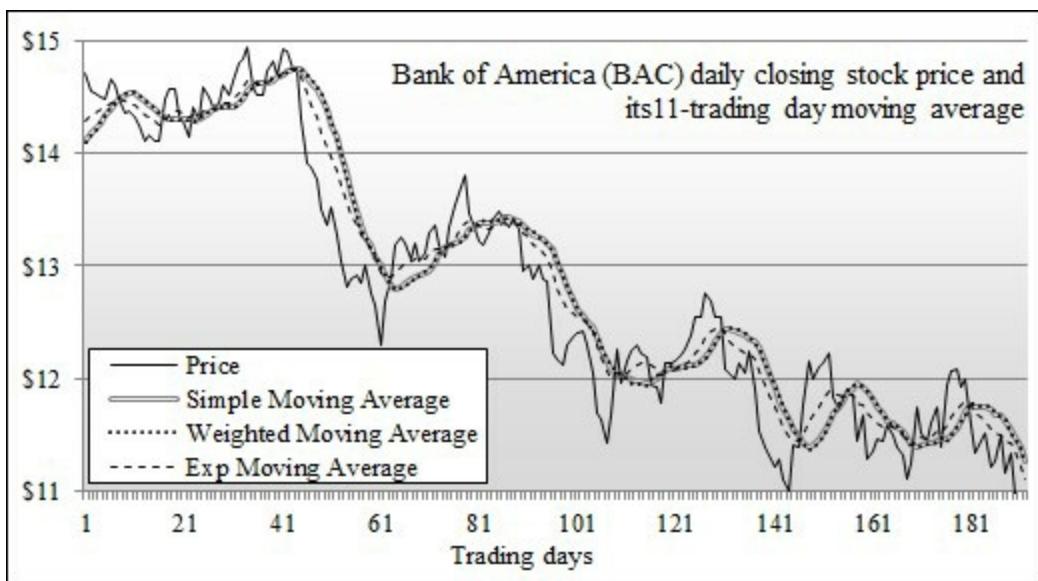
The instantiation of the partial function `filter |>` requires that the postfix operation `postfixOps` be made visible by importing `scala.language.postfixOps`.

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



Example of symmetric filter for weighted moving average

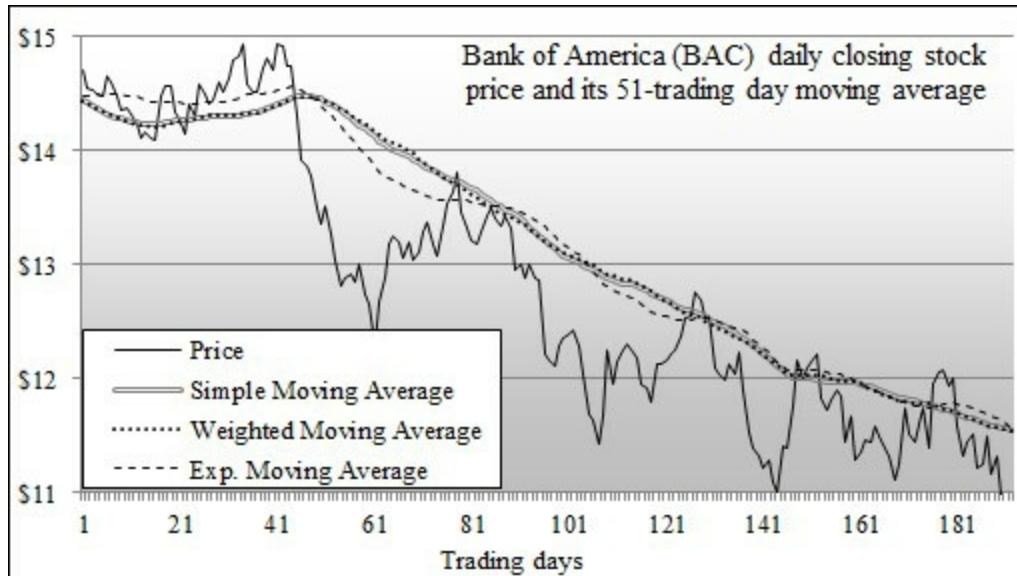
The three moving average techniques are applied to the price of the BAC over 200 trading days. Both the simple and weighted moving average use a period of 11 trading days. The exponential moving average method uses a scaling factor of $2/(11+1) = 0.1667$:



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

The three techniques filter the noise out of the original historical price time series. The exponential moving average reacts to sudden price fluctuation despite the fact that the smoothing factor is low. If you increase the period to

51 trading days (equivalent to 2 calendar months), the simple and weighted moving average produce a time series smoother than the exponential moving average with $\alpha = 2/(p+1) = 0.038$:



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

You are invited to experiment further with different smooth factors and weight distributions. You will be able to confirm the following basic rule: as the period of the moving average increases, noise with decreasing frequencies is eliminated.

In other words, the window of allowed frequencies is shrinking. The moving average acts as a low-pass filter that only preserves lower frequencies.

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

Tip

Moving average on multi-dimensional time series:

The moving average techniques are presented for single features or variable

time series, for the sake of simplicity. Moving average on multi-dimensional time series are computed by executing a single variable moving average to each feature using the `transform` method of `Vector[T]` introduced in the first section. Let's take for example the simple moving average applied to a multi-dimensional time series `xt`. The smoothed values are computed as follows:

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

Fourier analysis

The purpose of **spectral density estimation** is to measure the amplitude of a signal or a time series according to its frequency [3:5]. The objective is to estimate the spectral density by detecting periodicities in the dataset. A scientist can better understand a signal or time series by analyzing its harmonics.

Note

Spectral theory:

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

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

Note

Complex Fourier transform:

This section focuses on the discrete Fourier series for real value. The generic Fourier transform applies to complex values [3:6].

The decomposition in a basic trigonometric functions process is known as the **Fourier transform** [3:7].

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 is formulated as a linear combination of sine and cosine functions. The **DFT** is a linear transformation that converts a times 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 $a + j\cdot\varphi$ ($j^2 = -1$) 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 of real value.

Note

Real DFT:

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

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

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

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

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

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

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

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

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

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

The computation of the Fourier trigonometric series is time-consuming, with an asymptotic time complexity of $O(n^2)$. Scientists and mathematicians Several attempts have been working to make the computation as effective as possible. The most common numerical algorithm used to compute the Fourier series is the **Fast Fourier Transform (FFT)** created by *J.W Cooley* and *J. Tukey* [3:8].

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

Tip

Cooley-Tukey algorithm:

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

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

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

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

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

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

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

```
def pad(vec: DblVec, even: Boolean = true): DblVec = {
    val newSize = padSize(vec.size, even) //1
    if( newSize > 0) arr ++ Array.fill(newSize)(0.0) else arr //2
}

def padSize(xtSz: Int, even: Boolean= true): Int = {
    val sz = if( even ) xtSz else xtSz-1 //3
    if( (sz & (sz-1)) == 0) 0
    else {
        var bitPos = 0
        do { bitPos += 1 } while( (sz >> bitPos) > 0) //4
    }
}
```

```

        (if(even) (1<<bitPos) else (1<<bitPos)+1) - xtSz
    }
}

```

The method `pad` computes the optimal size of the frequency vector as 2^N by invoking the method `padSize` (line 1). It then concatenates the padding with the original time series or vector of observations (line 2). The method `padSize` adjusts the size of the data depending on whether the time series has initially an even or odd number of observations (line 3). It relies on bit operations to find the next radix N (line 4).

Tip

While loop:

Scala developers prefer Scala higher-order methods for collections to implement iterative computation. However, nothing prevents you from using the traditional `while` or `do { ... } while` loop if either readability or performance is an issue.

The fast implementation of the padding method, `pad`, consists of detecting the number of observations, N , as a power of 2 (next highest radix). The method evaluates if $N \& (N-1)$ is zero after it shifts the number of bits in the value N .

The next step is to write the class `DFT` for the real discrete transforms, sine and cosine, by sub-classing `DTransform`. The class relies on the padding mechanism implemented in `DTransform` whenever necessary:

```

class DFT[@specialized(Double) T: ToDouble] (
  eps: Double
) extends ETransform[Vector[T], DblVec](ConfigDouble(eps))
  with DTransform { //5
  protected val c: ToDouble[T] = implicitly[ToDouble[T]]

  override def |> : PartialFunction[Vector[T], Try[DblVec]] = { //6
    case xv: Vector[T] if(xv.size >= 2)
      => fwd(xv).map(_.2.toVector)
  }
}

```

We treat the discrete Fourier transform as a transformation on time series using an explicit configuration, `ETransform` (line 5). The transformation function `|>` delegates the computation to the method `fwrd` (line 6):

```
def fwrd(xv: Vector[T]): Try[(RealTransformer, Array[Double])] = {
    val rdt = if(Math.abs(c.apply(xv.head)) < eps) //7
              new FastSineTransformer(SINE) //8
        else new FastCosineTransformer(COSINE) //9
    val padded = pad(xv.map(c.apply(_)), xv.head == 0.0)
    Try( (rdt, rdt.transform(padded.toArray, FORWARD)) )
}
```

The method `fwrd` selects the discrete sine Fourier series if the first value of the time series is 0.0, the discrete cosine series otherwise. This implementation automates the selection of the appropriate series by evaluating `xv.head` (line 7). The transformation invokes the `FastSineTransformer` (line 8) and `FastCosineTransformer` (line 9) classes of the Apache Commons Math library [3:9] introduced in the first chapter.

This example uses the standard formulation of the cosine and sine transformation, defined by the argument `COSINE`. 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.

Tip

@specialized annotation:

The annotation `@specialized(Double)` 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 that the duplication of byte code as the specialized version coexists with the parameterized classes [3:10].

To illustrate the different concepts behind DFTs, let's consider the case of a time series generated by a sequence `h` of sinusoidal functions:

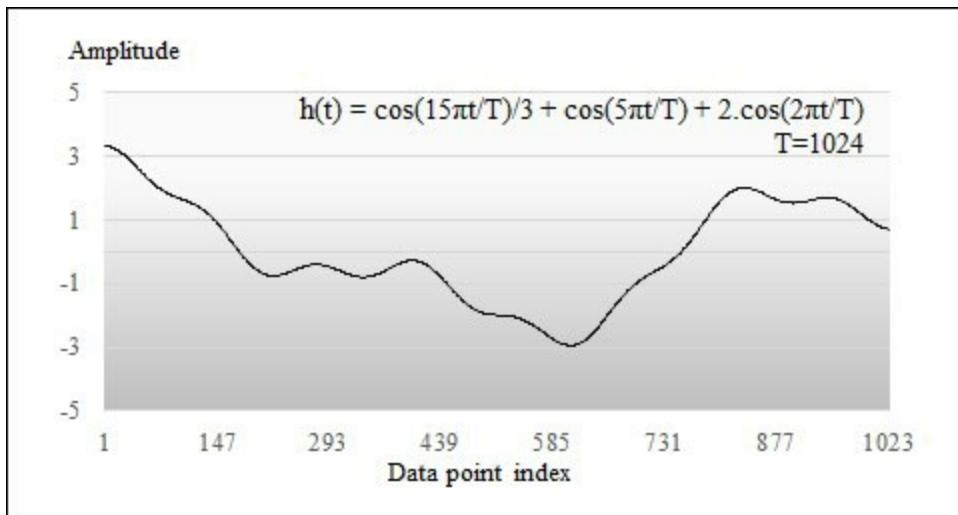
```

val F = Array[Double](2.0, 5.0, 15.0)
val A = Array[Double](2.0, 1.0, 0.33)

def harmonic(x: Double, n: Int): Double =
    A(n)*Math.cos(Math.PI*F(n)*x)
val h = (x: Double) =>
    Range(0, A.size).aggregate(0.0)((s, i) =>
        s + harmonic(x, i), _ + _)

```

As the signal is synthetically created, we can select the size of the time series to avoid padding. The first value in the time series is not null, so the number of observations is 2^n+1 . The data generated by the function `h` is plotted as follows:



Example of 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 explicit data transformation of the `DFT` class `|>`:

```

val OUTPUT1 = "output/filtering/simulated.csv"
val OUTPUT2 = "output/filtering/smoothed.csv"
val FREQ_SIZE = 1025; val INV_FREQ = 1.0/FREQ_SIZE

val pfndft = DFT[Double] |> //10
for {
    values <- Try(Vector.tabulate(FREQ_SIZE)
                  (n => h(n*INV_FREQ))) //11
}

```

```

    output1 <- DataSink[Double](OUTPUT1).write(values)
    spectrum <- pfnDFT(values)
    output2 <- DataSink[Double](OUTPUT2).write(spectrum) //12
} yield {
    val results = format(spectrum.take(DISPLAY_SIZE),
        "x/1025", SHORT)
    show(s"$DISPLAY_SIZE frequencies: ${results}")
}

```

The execution of the data simulator follows these steps:

1. Generate a raw data with the three-harmonic function h (line 11).
2. Instantiate the partial function generated by the transformation (line 10).
3. Store the resulting frequencies into a data sink (filesystem) (line 12).

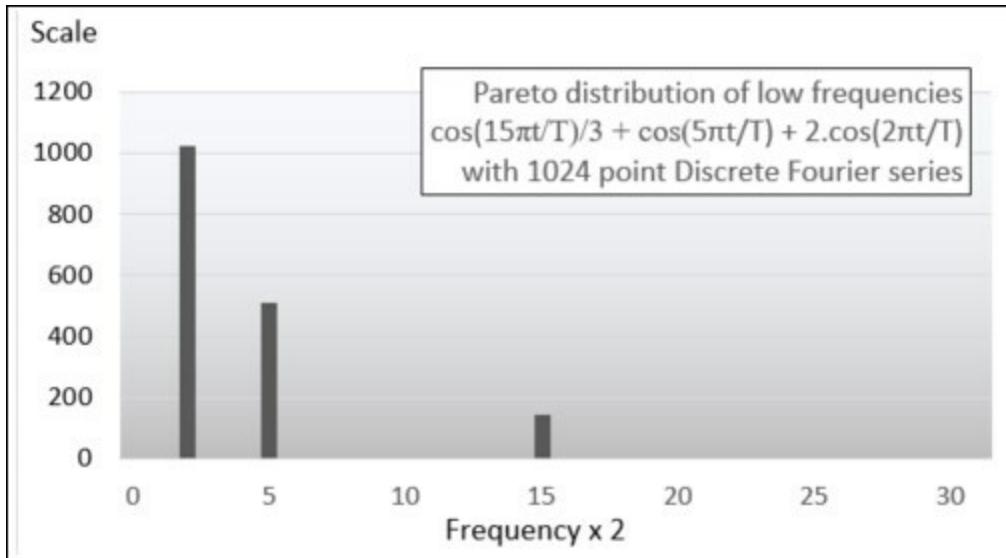
Tip

Data sinks and spreadsheets:

In this particular case, the results of the Discrete Fourier transform are dumped into a CSV file so they can be loaded into a spreadsheet. Some spreadsheets support a set of filtering techniques that can be used to validate the result of the example.

A simpler alternative would be to use JFreeChart.

The spectrum of frequencies for the time series, plotted for the first 32 points, clearly shows three frequencies at $k=2, 5$, and 15 . The result is expected because the original signal is composed of three sinusoidal functions. The amplitude of these frequencies is $1024/1$, $1024/2$, and $1024/6$, respectively. The following plot represents the first 32 harmonics for the time series:



Frequency spectrum for three-frequency sinusoidal

The next step is to use the frequencies spectrum to create a low-pass filter using DFT. There are many algorithms available to implement a low or pass band filter in the time domain, from *autoregressive models* to the *Butterworth* algorithm. However, the discrete Fourier transform is still a very popular technique to smooth signals and identify trends.

Note

Big data:

A DFT for a large time series can be very computationally intensive. One option is to treat the time series as a continuous signal and sample it using the **Nyquist** frequency. The Nyquist frequency is half of the sampling rate of a continuous signal.

DFT-based filtering

The purpose of this section is to introduce, describe, and implement a noise filtering mechanism that leverages the discrete Fourier transform. The idea is quite simple: the forward and inverse Fourier series are used sequentially to convert the raw data from the time domain to the frequency domain and back. The only input you need to supply is a function g that modifies the sequence of frequencies. This operation is known as the convolution of the filter g and the frequencies spectrum.

A convolution is similar to an inner product of two time series in the frequencies domain. Mathematically, the convolution is defined as follows.

Note

Convolution:

M10: The convolution of two functions f and g is defined as follows:

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

M11: The convolution F of a time series, $x = \{x_j\}$ with a frequency spectrum ω^x and a filter f in frequency domain ω^f is defined as follows:

$$F(x * f) = F(x) \cdot F(g) = \sum_{j=0}^{N-1} \omega_j^x \cdot \omega_{k-j}^f$$

Let's apply the convolution to our filtering problem. The filtering algorithm using the discrete Fourier transform consists of five steps:

1. Pad the time series to enable the discrete sine or cosine transform.
2. Generate the ordered sequence of frequencies using the forward

transform, F .

3. Select the filter function G in the frequency domain and a cut-off 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:

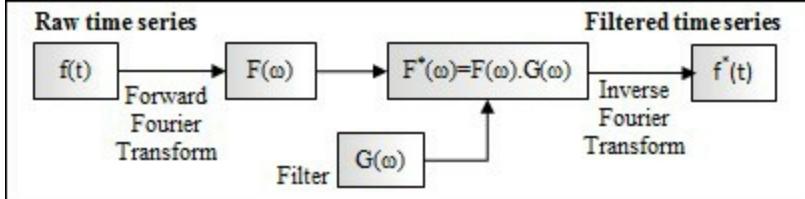


Diagram of the discrete Fourier filter

The most commonly used low-pass filter functions are known as the `sinc` and `sinc2` functions defined as a rectangular function and a triangular function, respectively. These functions are partially applied functions derived from a generic `convol` method. The simplest function `sinc` returns 1 for frequencies below a cut-off frequency f_C , 0 if it is higher:

```

val convol = (n: Int, f: Double, fC: Double) =>
  if( Math.pow(f, n) < fC) 1.0 else 0.0
val sinc = convol(1, _: Double, _:Double)
val sinc2 = convol(2, _: Double, _:Double)
val sinc4 = convol(4, _: Double, _:Double)
  
```

Tip

Partially applied functions versus partial functions:

Partial functions and partially applied functions are not actually related.

A partial function f is a function that is applied to a subset X' of the input space X . It does not execute for all possible input values:

```

def f(x: X): PartialFunction[X, Y] = {
  case x: X if(x belong X') => execute
}
  
```

A partially applied function f_2 is a function value for which the user supplies

the value for one or more argument. The projection reduces the dimension of the input space (x, z):

```
def f(x: X, z: Z): Y
val z0: Z = a
val f2 = (x: X) => f(x, _: Z)
```

The class `DFTFilter` inherits from the `DFT` class in order to reuse the forward transform function `fwd`. The frequency domain function `g` is an attribute of the filter. The `g` function takes the frequency cut-off value `fc` as second argument (line 13). The two filters `sinc` and `sinc2` defined in the previous section are examples of filtering functions:

```
class DFTFilter[@specialized(Double) T: ToDouble] (
  fc: Double,
  eps: Double)
  (g: (Double, Double) => Double) extends DFT[T](eps) { //13

  override def |>: PartialFunction[Vector[T], Try[DblVec]] = {
    case xt: Vector[T] if( xt.size >= 2 ) => {
      fwd(xt).map{ case(trf, freq) => { //14
        val cutOff = fc*freq.size
        val filtered = freq.indices
          .map{n => freq(n)*g(n, cutOff)} //15
        trf.transform(filtered.toArray, INVERSE).toVector } } //16
    }
  }
}
```

The filtering process follows three steps:

1. Computation of the discrete Fourier forward transformation (sine or cosine), `fwd` (line 14).
2. Apply the filter function (formula M11) through a Scala `map` method (line 15).
3. Apply the inverse transform on the frequencies (line 16).

Let us evaluate the impact of the cut-off values on the filtered data. The implementation of the test program consists of loading the data from file (line 17), then invoking the `DFTFilter` partial function `pfnDFTfilter` (line 18):

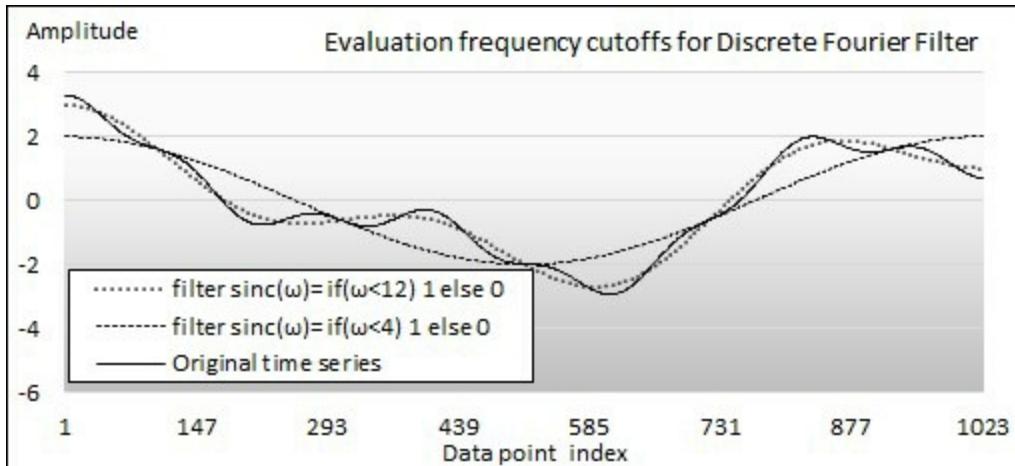
```
import YahooFinancials._
```

```

val inputFile = s"$RESOURCE_PATH$symbol.csv"
val CUTOFF = 0.005
val pfnDFTfilter = DFTFilter[Double](CUTOFF)(sinc) |>
for {
  path <- getPath(inputFile)
  src <- DataSource(path, false, true, 1)
  price <- src.get(adjClose) //17
  filtered <- pfnDFTfilter(price) //18
} yield { /* ... */ }

```

Filtering the noise out 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:



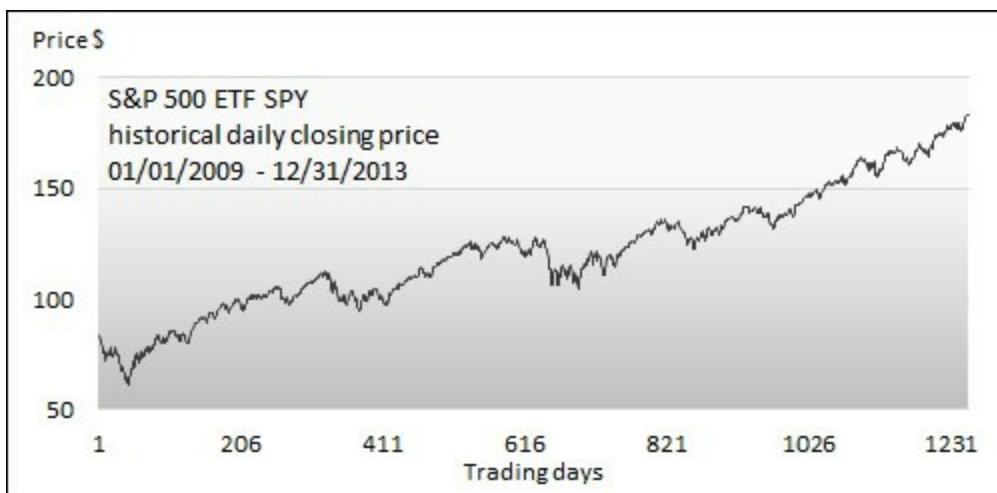
Plotting of discrete Fourier filter based smoothing

As you would expect, the low-pass filter with a cut-off value of 12 eliminates the noise with the highest frequencies. The filter with the cutoff value 4 cancels out the second harmonic (low-frequency noise), leaving out only the main trend cycle.

Detection of market cycles

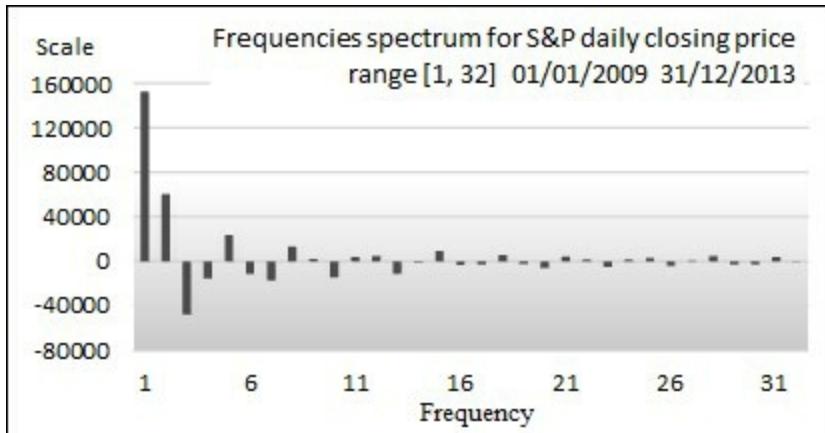
Using the discrete Fourier transform to generate the frequencies spectrum of a periodical time series is easy. However, what about real-world signals such as the time series representing the historical price of a stock?

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 frequencies 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:

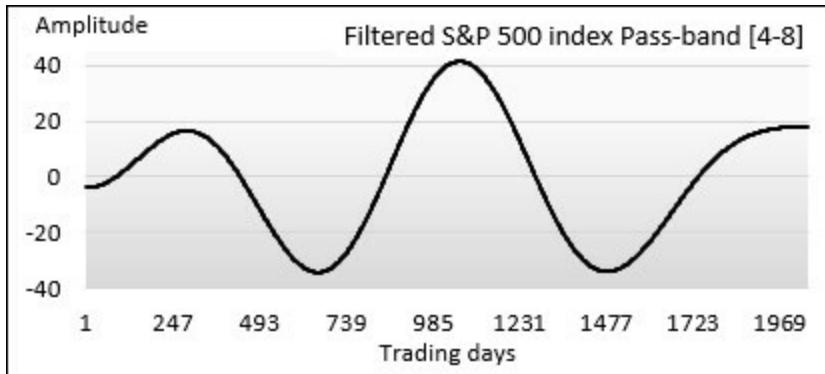
- Both positive and negative amplitudes are present, as you would expect in a time series with complex values. The cosine series contributes to the positive amplitudes while the sine series affects both positive and negative amplitudes ($\cos(x+\pi) = \sin(x)$).
- The decay of the amplitude along the frequencies is steep enough to warrant further analysis beyond the first harmonic: the first harmonic represents the main trend of the historical stock price. The next step is to apply a band-pass filter technique to the S&P 500 historical data in order to identify short-term trends with lower periodicity.

A low-pass filter is limited to reduce or cancel out the noise in the raw data. In this case, a band-pass filter using a range or window of frequencies is appropriate to isolate the frequency or the group of frequencies that characterize a specific cycle. The `sinc` function introduced in the previous section to implement a low-pass filter is modified to enforce the band-pass within a window $[w_1, w_2]$ as follows:

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

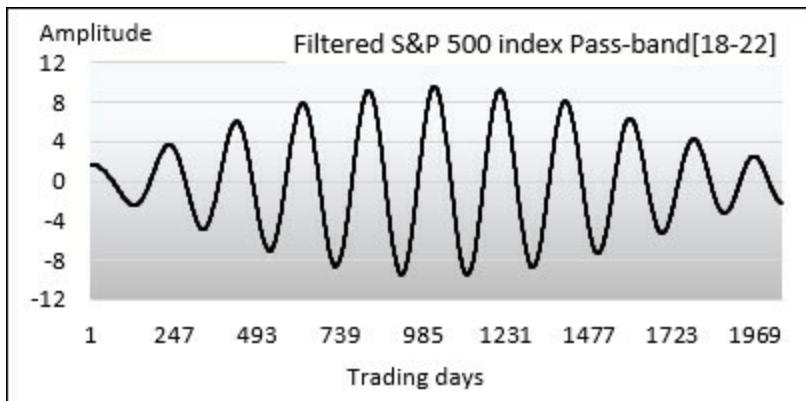
Let us define a DFT-based band-pass filter with a window of width 4, $w=(i, i+4)$ with i ranging between 2 and 20. Applying the window $[4, 8]$ isolates the impact of the second harmonic on the price. As we eliminate the main

upward trend with frequencies less than 4, all filtered data varies within a short range relative to the main trend. The following graph shows the output of the filter:



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

In this next 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:



Output of a band-pass DFT filter range 18-22 on historical S&P index

There is a possible rational explanation for the shape of the S&P 500 data filtered by a band-pass with a frequency 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 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 breaks through its resistance and resumes a strong uptrend characterized by a high amplitude low frequency, as shown in the following graph:

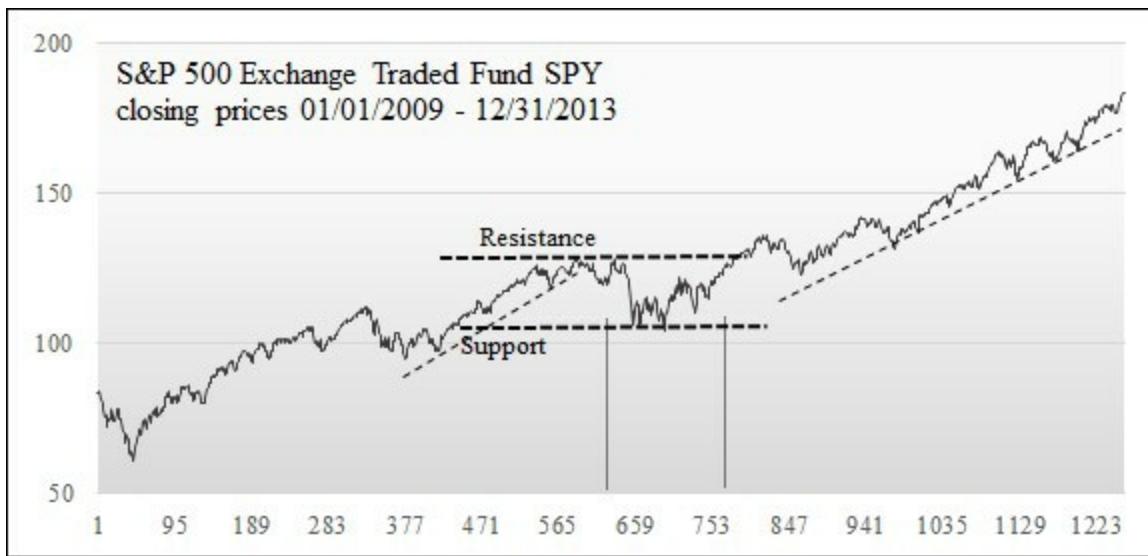


Illustration of support and resistance levels for the historical S&P 500 index prices

One of the limitations of using the discrete Fourier based filters to clean up data is that it requires the data scientist to extract the frequencies spectrum and modify the filter on a regular basis, as they are never sure that the most recent batch of data does not introduce noise with different frequency. The Kalman filter addresses this limitation.

The discrete Kalman filter

The Kalman filter is a mathematical model that provides an accurate and recursive computation approach to estimate the previous states and predict the future states of a process for which some variables may be unknown. *R.E. Kalman* introduced it in the early 1960s to model dynamics systems and predict trajectory in aerospace [3:11]. 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 model (HMM)** described in the *Hidden Markov model* section of [Chapter 7, Sequential Data Models](#) [3:12].

The Kalman filter is:

- A predictor of the next data point from the current observation
- A filter that weeds out noise by processing the last two observations
- A smoothing model that identifies trends from a history of observations

Note

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 [3:13].

Note

Kalman and non-linear systems:

The Kalman filter estimates the internal state of a linear dynamic system. However, it can be extended to model non-linear state space using linear or quadratic approximation functions. These filters are known as, you guessed it, **Extended Kalman Filters (EKFs)**, 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 dynamic 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 (M12):

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

- A_t is the square matrix of dimension n that represents the transition from state x_{t-1} at $t-1$ to state x_t at t . The matrix is intrinsic to the dynamic system under consideration.
- B_t 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_t .
- w_t 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 the chapter, have no external input to the state of the model.

Note

White and Gaussian noise:

A white noise is a Gaussian noise, following a normal distribution with zero mean.

The measurement equation

The measurement of m values z_t of the state of the system is defined by the following equation (M13):

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

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

Note

Time dependency model:

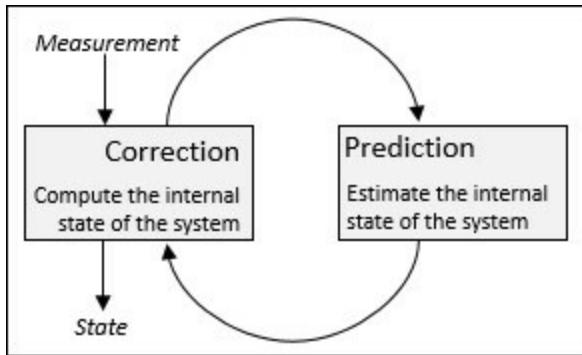
We cannot assume that the parameters of the generalized discrete Kalman filter such as state transition A_t , control input B_t , and observation (or measurement dependency) H_t matrices are independent from time. However, these parameters are constant in most practical applications.

The recursive algorithm

The set of equations for the discrete Kalman filter is implemented as a recursive computation with two distinct steps:

- The algorithm uses the transition equations to estimate the next observation
- The estimation is created with the actual measurement for this observation

The recursion is visualized in the following diagram:



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 to 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 the trend in interest rates a bit more reliable [3:14].

Applying the Kalman filter to clean raw data requires you to define a model that encompasses both observed and non-observed states. In the case of the trend analysis, we can safely create our model with a two-variable state: the current yield x_t and the previous yield x_{t-1} .

Note

State of dynamic systems:

The term "state" refers to the state of the dynamic system under consideration, not the state of the execution of the algorithm.

This implementation of the Kalman filter uses the Apache Commons Math library. Therefore, we need to specify the implicit conversion from our primitives introduced in the *Primitives and implicits* section of [Chapter 1, Getting Started](#) to Apache Commons Math types `RealMatrix`, `RealVector`, `Array2DRowRealMatrix`, and `ArrayRealVector`:

```
type DblMatrix = Array[Array[Double]]  
  
implicit def double2RealMatrix(x: DblMatrix): RealMatrix =  
  new Array2DRowRealMatrix(x)  
implicit def double2RealRow(x: DblVec): RealMatrix =  
  new Array2DRowRealMatrix(x)  
implicit def double2RealVector(x: DblVec): RealVector =  
  new ArrayRealVector(x)
```

The client code has to import the implicit conversion functions within its scope.

The Kalman model assumes that process and measurement noise follows a Gaussian distribution, also known as white noise. For the sake of maintainability, the generation of the white noise is encapsulated in the class `QRNoise` with the following arguments (line 1):

- `qr`: Tuple of scale factors for the process noise matrix \mathbf{Q} and the measurement noise \mathbf{R}
- `profile`: Noise profile with the normal distribution as default

The two methods, `noiseQ` and `noiseR`, generate an array of two independent white noise elements (line 2):

```
val normal = Stats.normal(_)

class QRNoise(qr: DblPair, profile: Double=>Double = normal) { //1
    def q = profile(qr._1)
    def r = profile(qr._2)
    lazy val noiseQ = Array[Double](q, q)      //2
    lazy val noiseR = Array[Double](r, r)
}
```

Tip

Experimenting with noise profile:

Although the discrete Kalman filter assumes the noise `profile` follows a normal distribution, the class `QRNoise` allows the user to experiment with different noise profiles.

The easiest approach to manage the matrices and vectors used in the recursion is to define them as arguments of a configuration class `kalmanConfig`. The arguments of the configuration follow the naming convention defined in the mathematical formulas: `A` is the state transition matrix, `B` is the control matrix, `H` is the matrix of observations defining the dependencies between measurement and system state, and `P` is the covariance error matrix:

```
case class KalmanConfig(A: DblMatrix, B: DblMatrix,
                        H: DblMatrix, P: DblMatrix)
type U = Vector[DblPair] //3
type V = Vector[DblPair] //4
```

Let us implement the Kalman filter as a transformation `DKalman` of type `ETransform` on a time series with a predefined configuration `KalmanConfig`:

```
class DKalman(config: KalmanConfig)(implicit qrNoise: QRNoise)
    extends ETransform[U, V](config) {
    type KRState = (KalmanFilter, RealVector) //5
    override def |> : PartialFunction[U, Try[V]] =
    ...
```

}

As with any explicit data transformation, we need to specify the type `U` and `V` (lines 3 and 4) that are identical: the Kalman filter does not alter the structure of the data, only the values. We define an internal state for the Kalman computation, `KRState`, by creating a tuple of two Apache Commons Math types, `KalmanFilter` and `RealVector` (line 5).

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 state x (yield of the Treasury 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:

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

Visualization of the transition equation of the Kalman filter

The purpose of this exercise is to evaluate the impact of the different parameters of the transition matrix A in terms of smoothing.

Note

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, for example.

The mathematics behind the Kalman filter presented as reference to the implementation in Scala use the same notation for matrices and vectors. It is 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 next paragraph describes the two equations for the prediction step.

The **prediction step** is defined as follows:

The prediction of the state at time t is computed by extrapolating the state estimate:

$$\hat{x}_t' = A_t \cdot \hat{x}_{t-1} + B_t \cdot u_t$$

A is the square matrix of dimension n that represents the transition from state x at $t-1$ to state x at time t .

$x't$ 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:

$$P_t' = A_t \cdot P_{t-1} + A_t^T + Q_t$$

Where:

AT 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 Common Math library. The types of matrices and vectors are automatically converted into `RealMatrix` and `RealVector` classes.

The implementation of the equation M14 is as follows:

```
x = A.operate(x).add(qrNoise.create(0.03, 0.1))
```

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

Correction

The second step of the recursive Kalman algorithm is the correction of the estimated yield of the 10-year Treasury bond with the actual yield. In this example, the white noise of the measurement is negligible. The measurement equation is simple because the state is represented by the current and previous yield, and their measurement z .

$$\begin{bmatrix} z_t \\ z_{t-1} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \cdot \begin{bmatrix} \hat{x}_t \\ \hat{x}_{t-1} \end{bmatrix} + \begin{bmatrix} v_t \\ v_{t-1} \end{bmatrix}$$

Visualization of the measurement equation of the Kalman filter

The sequence of mathematical equations of the correction phase consists of updating the estimation of the state x using the actual values z and computing the Kalman gain K .

The **correction step** is defined as follows:

M16: The state of the system x is estimated from the actual measurement z through the formula:

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

Where:

r_t is the residual between the predicted measurements and the actual measured values

K_t is the Kalman gain for the correction factor

M17: The Kalman gain is computed as follows:

$$K_t = P_t' \cdot H_t^T \left(H_t \cdot P_t' \cdot H_t^T + R_t \right)^{-1}$$

Here, H_T is the matrix transpose of H and P_t' is the estimate of the error covariance.

Kalman smoothing

It is time to put our knowledge of the transition and measurement equations to the test. The Apache common 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 are loaded through the `DataSource` method and mapped to a smoothed series that is the output of the filter:

```
override def |> : PartialFunction[U, Try[V]] = {
  case xt: U if( !xt.isEmpty) => Try(
    xt.map { case(current, prev) => {
      val models = initialize(current, prev) //6
      val nState = newState(models) //7
      (nState(0), nState(1)) //8
    } }
  )
}
```

The data transformation for the Kalman filter initializes the process and measurement model for each data point in the private method `initialize` (line 6), update the state using the transition and correction equations iteratively in the method `newState` (line 7), and return the filtered series of pair values (line 8).

Tip

Exception handling:

The code to catch and process exceptions thrown by the Apache Commons Math library is omitted as 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 method `initialize` encapsulates the initialization of the process model, `pModel` (line 9) and measurement (observations dependencies) model, `mModel` (line 10) as defined in the Apache Common Math library:

```
def initialize(current: Double, prev: Double): KRState = {
    val pModel = new DefaultProcessModel(
        config.A, config.B, Q, input, config.P
    ) //9
    val mModel = new DefaultMeasurementModel(config.H, R) //10
    val in = Array[Double](current, prev)
    (new KalmanFilter(pModel, mModel), new ArrayRealVector(in))
}
```

The exceptions thrown by the Apache Commons Math API are caught and processed through a `Try` instance. The iterative prediction and correction of the smoothed yields of 10-year Treasury bond is implemented by the `newState` method. The method iterates through the following steps:

1. Estimate the new values of the state by invoking an Apache Commons Math `KalmanFilter.predict` method that implements the formula **M14** (line 11).
2. Apply the formula **M12** to the new state x at time t (line 12).
3. Compute the measured value z at time t using the **M13** formula (line 13).
4. Invoke the Apache Commons Math `KalmanFilter.correct` method to implement the formula **M16** (line 14).

5. Return the estimate value for the state x by invoking the Apache Commons Math `KalmanFilter.getStateEstimation` method (line 15):

```
def newState(state: KRState): Array[Double] = {
    state._1.predict //11
    val x = config.A.operate(state._2).add(qrNoise.noisyQ) //12
    val z = config.H.operate(x).add(qrNoise.noisyR) //13
    state._1.correct(z) //14
    state._1.getStateEstimation //15
}
```

Tip

Exit condition:

In the code snippet for the method `newState`, the iteration for specific data points exits when the maximum number of iterations is reached. A more elaborate implementation consists of either evaluating the matrix P at each iteration or estimation converged within a predefined range.

Fixed lag smoothing

So far, we have studied the Kalman filtering algorithm. We need to adapt it to the smoothing of time series. The **fixed lag smoothing** technique consists of backward correcting the previous data point taking into account the latest actual value.

An N -lag smoother defines the input as a vector $X = \{x_{t-N-1}, x_{t-N-2}, \dots, x_t\}$ for which the value x_{t-N-j} is corrected taking into account the current value of x_t .

The strategy is quite similar to the hidden Markov model forward and backward passes (refer to the *Evaluation* section under *Hidden Markov model (HMM)* in [Chapter 7, Sequential Data Models](#)).

Note

Complex strategies for lag smoothing:

There are numerous formulas or methodologies to implement an accurate fixed lag smoothing strategy and correct the predicted observations. Such strategies are beyond the scope of this book.

Experimentation

The objective is to smooth the yield of the 10-year Treasury bond using a **two-step lag smoothing** algorithm.

Note

Two-step lag smoothing:

M18: 2-step lag smoothing algorithm for state S_t using a single smoothing factor α :

$$S_t = [x_{t+1}, x_t]^T \text{ with } \begin{vmatrix} x_{t+1} \\ x_t \end{vmatrix} = \begin{vmatrix} \alpha & 1 - \alpha \\ 1 & 0 \end{vmatrix} \cdot \begin{vmatrix} x_t \\ x_{t-1} \end{vmatrix}$$

The state equation updates the values of the state $[x_t, x_{t-1}]$ using the previous state $[x_{t-1}, x_{t-2}]$ where x_t represents the yield of the 10-year Treasury bond at time t . This is accomplished by shifting the values of the original time series $\{x_0 \dots x_{n-1}\}$ by 1 using the drop method: $X1 = \{x_1 \dots x_{n-1}\}$, creating a copy of the original time series without the last element $X2 = \{x_0 \dots x_{n-2}\}$, and zipping $X1$ and $X2$. This process is implemented by the `zipWithShift` method introduced in the first section of the chapter.

The resulting sequence of state vector $S_k = [x_k, x_{k-1}]^T$ is processed by the Kalman algorithm:

```
Import YahooFinancials._
val RESOURCE_DIR = «resources/data/filtering/»
implicit val qrNoise = new QRNoise((0.7, 0.3)) //16

val H: DblMatrix = ((0.9, 0.0), (0.0, 0.1)) //17
```

```

val P0: DblMatrix = ((0.4, 0.3), (0.5, 0.4))    //18
val ALPHA1 = 0.5; val ALPHA2 = 0.8

(src.get(adjClose)).map(zt => {    //19
  twoStepLagSmoothen(zt, ALPHA1)      //20
  twoStepLagSmoothen(zt, ALPHA2)
})

```

Tip

Implicit noise instance:

The noise for the process and measurement is defined as an implicit argument to the Kalman filter, `DKalman`, for two reasons:

- The profile of the noise is specific to the process or system under evaluation and its measurement. It is independent from the Kalman configuration parameters A , B , and H . Therefore, it cannot be a member of the `KalmanConfig` class.
- The same noise characteristics should be shared with other alternative filtering techniques, if needed.

The white noise for the process and the measurement are initialized implicitly with the value `qrNoise` (line 16). The code initializes the matrices `H` of the measurement dependencies on the state (line 17) and `P0` containing the initial covariance errors (line 18). The input data is extracted from a CSV file containing the daily Yahoo financial data (line 19). Finally, the method executes the two-step lag smoothing algorithm, `twoStepLagSmoothen`, with two different `alpha` parameter values, `ALPHA1` and `ALPHA2` (line 20).

Let's consider the `twoStepLagSmoothen` method:

```

def twoStepLagSmoothen(zSeries: DblVec, alpha: Double): Int = {
  val A: DblMatrix = ((alpha, 1.0-alpha), (1.0, 0.0)) //21
  val xt = zipWithShift(1) //22
  val pfnKalman = DKalman(A, H, P0) |> //23
  pfnKalman(xt).map(filtered => //24
    display(zSeries, filtered.map(_.1), alpha) )
}

```

The method `twoStepLagSmoothen` takes two arguments:

- A single variable time series `zSeries`
- A state transition parameter `alpha`

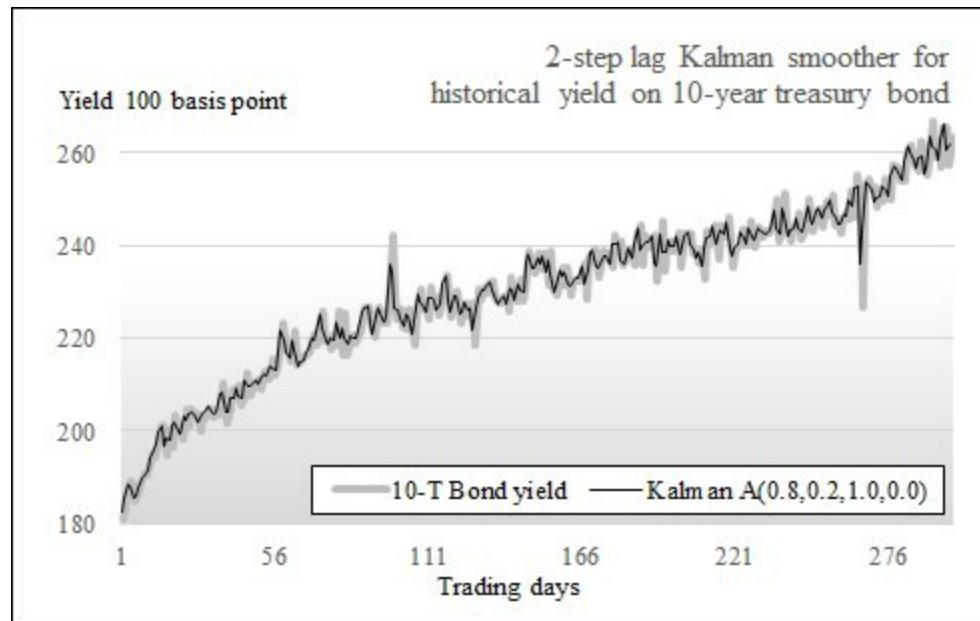
It initializes the state transition matrix `A` using the exponential moving average decay parameter `alpha` (line 21). It creates the two-step lag time series `xt` using the `zipWithShift` method (line 22). It extracts the partial function, `pfnKalman` (line 23), processes, and finally displays the two-step lag time series (line 24).

Tip

Modeling state transition and noise:

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

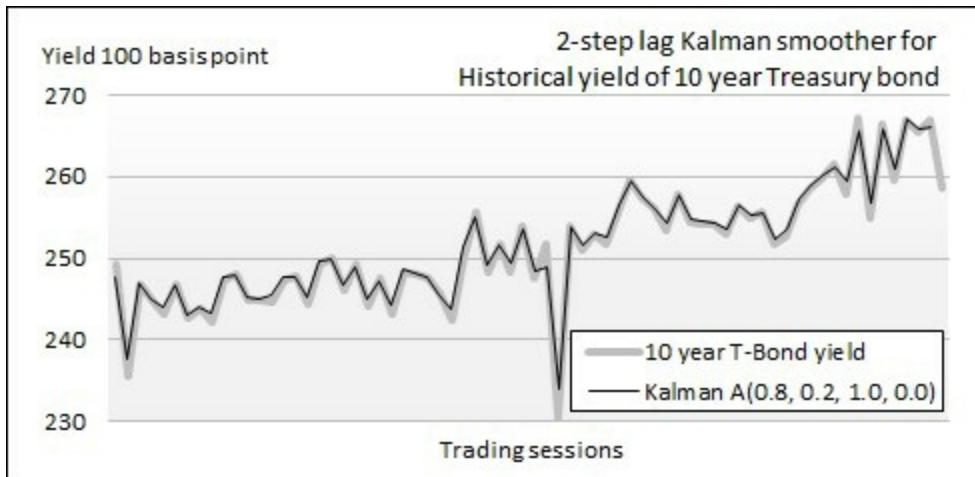
The smoothed yield is plotted along the raw data as follows:



Output of the Kalman filter for 10-year Treasury-Bond historical prices

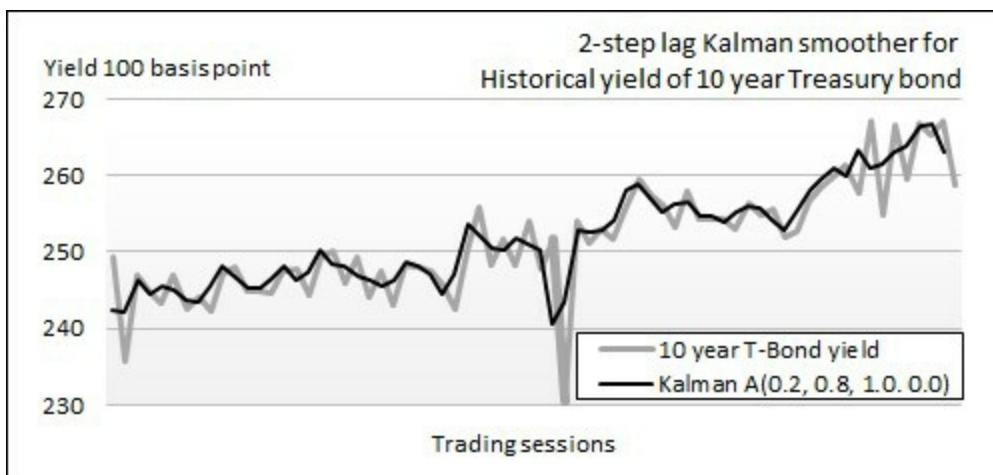
The Kalman filter can smooth the historical yield of the 10-year Treasury bond while preserving the spikes and lower frequency noise.

Let's analyze the data for a shorter period during which the noise is the strongest, between the 190th and the 275th trading days:



Output of Kalman filter for the 10-year Treasury bond prices 0.8-.02

The high-frequency noise has been significantly reduced without canceling 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:



Output of Kalman filter for the 10-year Treasury bond price 0.2-0.8

Benefits and drawbacks

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

However, the linear discrete 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 Gaussian sum filter or adaptive Gaussian mixture [3:15].
- It requires that the underlying process is linear. However, researchers have been able to formulate extensions to the Kalman filter, known as the **extended Kalman filter (EKF)** to filter signals from non-linear dynamic systems, at the cost of significant computational complexity.

Note

Continuous-time Kalman filter:

The Kalman filter is not restricted to dynamic systems with discrete states, x . The case of continuous state-time is handled by modifying the state transition equation so the estimated state is computed as derivative dx/dt .

Alternative preprocessing techniques

For the sake of space and your time, this chapter introduced and applied three filtering and smoothing classes of algorithms. Moving averages, Fourier series, and Kalman filter are far from being the only techniques used in cleaning raw data. The alternative techniques can be classified into the following categories:

- Autoregressive models that encompass **Auto-Regressive Moving Average (ARMA)**, **Auto-Regressive Integrated Moving Average (ARIMA)**, **generalized autoregressive conditional heteroscedasticity (GARCH)**, and Box-Jenkins rely on some form of autocorrelation function
- **Curve-fitting** algorithms that include the polynomial and geometric fit with ordinary least squares method, non-linear least squares using the **Levenberg-Marquardt** optimizer and probability distribution fitting
- Non-linear dynamic systems with Gaussian noise such as **particle filter**
- Hidden Markov models as described in *Hidden Markov models* section of [Chapter 7, Sequential data models](#)

Summary

This completes the overview of the most commonly used data filtering or smoothing techniques. There are other types of data preprocessing algorithms such as normalization, analysis and reduction of variance, and identification of missing values that are 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 step introduced in the next chapter: unsupervised learning and, more specifically, clustering.

Chapter 4. Unsupervised Learning

Labeling a set of observations for classification or regression can be a daunting task, especially in the case of a large features set. In some cases, labeled observations are either unavailable or not possible to create. In an attempt to extract some hidden associations or structures from observations, the data scientist relies on unsupervised learning techniques to detect patterns or similarity in data.

The goal of unsupervised learning is to discover patterns of regularities and irregularities in a set of observations. These techniques are also applied in reducing the solution or features space.

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

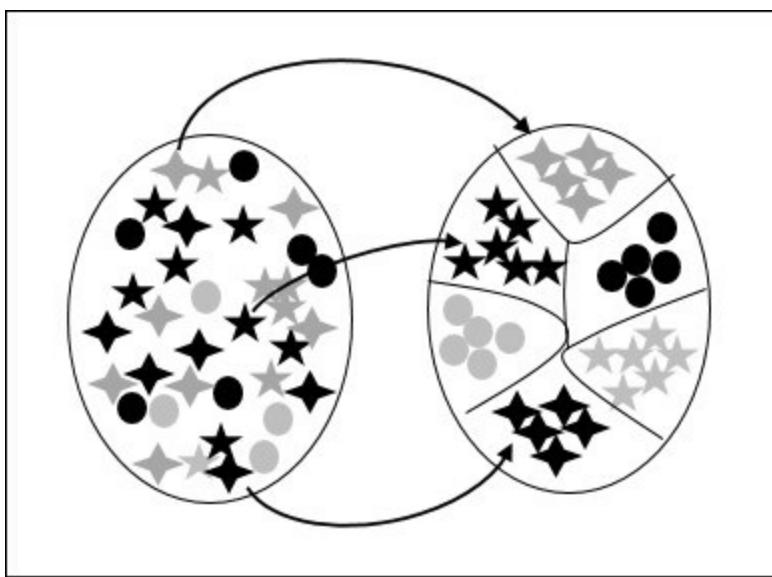
- **K-means:** Clustering observed features
- **Expectation-Maximization (EM):** Clustering observed and latent features
- Function approximation

Any of these algorithms can be applied to technical analysis or fundamental analysis. Fundamental analyses of financial ratios and technical analyses of price movements are described in the *Technical analysis* section under *Finances 101* in the *Appendix*. The K-means algorithm is fully implemented in Scala, while the EM and principal components analyses leverage the Apache commons math library.

The chapter concludes with a brief overview of dimension reduction techniques for non-linear models.

K-mean clustering

Problems involving many features for large datasets become quickly intractable, and it is quite difficult to evaluate the independence between features. Any computation that requires some level of optimization and, at a minimum, the computation of first order derivatives, demands 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 appropriate. The objective is to reduce very large sets of observations into a small group of observations that share some common attributes:



Visualization of data clustering

This approach is known as vector quantization. Vector quantization is a method that divides a set of observations into groups of similar sizes. The main benefit of vector quantization is that 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.

Note

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 for handling missing data.

K-means

K-means is a popular clustering algorithm that can be implemented either iteratively or recursively. The representative of each cluster is computed as the center of the cluster, known as the **centroid**. The similarity between observations within a single cluster relies on the concept of distance (or similarity) between observations.

Measuring similarity

There are many ways to measure the similarity between observations. The most appropriate measure has to be intuitive and avoid computational complexity. This section reviews three similarity measures:

- Manhattan distance
- Euclidean distance
- Normalized inner or dot product

The Manhattan distance is defined by the absolute distance between two variables or vectors $\{x_i\}$ and $\{y_i\}$ of the same size (M1):

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

The implementation is generic enough to compute the distance between two vectors of elements of different types if an implicit conversion between each of these types to `Double` values is already defined:

```
def manhattan[@specialized(Double) T: ToDouble,
              @specialized(Double) U: ToDouble] (
    x: Array[T], y: Array[U]): Double = {
  val cu: ToDouble[T] = implicitly[ToDouble[T]]
  val cv: ToDouble[U] = implicitly[ToDouble[U]]
  (x, y).zipped.map{case (u, v)
    .Math.abs(cu.apply(u)-cv.apply(v))}.sum
}
```

The ubiquitous Euclidean distance between two vectors $\{x_i\}$ and $\{y_i\}$ of the same size is defined by the following formula (M2):

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

```
def euclidean[@specialized(Double) T: ToDouble,
              @specialized(Double) U: ToDouble] (
    x: Array[T], y: Array[U]): Double = {
  val cu: ToDouble[T] = implicitly[ToDouble[T]]
  val cv: ToDouble[U] = implicitly[ToDouble[U]]
  Math.sqrt((x,y).zipped.map{case (u,v) =>
    sqr(cu.apply(u)- cv.apply(v)).sum)
}
```

The normalized inner product or cosine distance between two vectors $\{x_i\}$ and $\{y_i\}$ is defined by the following formula (M3):

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

In this implementation, the computation of the dot product and the norms for each dataset is done simultaneously using the tuple within the `fold` method:

```
def cosine[@specialized(Double) T: ToDouble,
           @specialized(Double) U: ToDouble] (
  x: Array[T],
  y: Array[U]): Double = {

  val cu: ToDouble[T] = implicitly[ToDouble[T]]
  val cv: ToDouble[U] = implicitly[ToDouble[U]]
  val norms = (x,y).zipped.map{ case (u,v) => {
    val wu = cu.apply(u)
    val wv = cv.apply(v)
    Array[Double](wu*wv, wu*wu, wv*wv)
  }}./:(Array.fill(3)(0.0))((s, t) => s ++ t)

  norms(0)/sqrt(norms(1)*norms(2))
}
```

Tip

Performance of zip and zipped

The scalar product of two vectors is one of the most common operations. It is tempting to implement the dot product using the generic `zip` method:

```
def dot(x:Array[Double], y:Array[Double]): Array[Double] = x.zip
```

A functional alternative is to use the `Tuple2.zipped` method:

```
def dot(x:Array[Double], y:Array[Double]): Array[Double] = (x, y
```

If readability is not a primary concern, you can always implement the dot method with a `while` loop which prevents you from using the ubiquitous `while` loop.

Defining the algorithm

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

Note

K-means objective

M4: Let's consider K cluster $\{C_k\}$ with means or centroids $\{m_k\}$. The K-means algorithm is indeed an optimization problem, the objective of which is to minimize the reconstruction or total error defined in the total sum of the distance:

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

The four steps of the K-means algorithm are:

1. Clusters the configuration that is initializing the centroids or means m_k of the K clusters.
2. Clusters the assignment that is assigning observations to the nearest cluster given the centroids m_k .
3. Error minimization, that is, computing the total reconstruction error.
4. Computes the centroids m_k that minimize the total reconstruction error for the current assignment:
 1. Reassigns the observations, given the new centroids m_k .
 2. Repeats the computation of the total reconstruction error until no observations are reassigned.
5. Classification of a new observation by assigning the observation to the closest cluster.

We need to define the components of the K-means in Scala before implementing the algorithm.

Step 1 – Clusters configuration

Let us create the two main components of the K-means algorithms:

- Clusters of observations
- The implementation of the K-means algorithm

Defining clusters

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

- Centroid, `center` (line 1)
- The indices of the observations that belong to this cluster, `members` (line 2):

```
class Cluster[T: ToDouble] (val center: Array[Double]) { //1
  type DistanceFunc[T] = (Array[Double], Array[T]) => Double

  val members = new ListBuffer[Int] //2
  def moveCenter(xt: Vector[Array[T]]): Cluster[T]
  ...
}
```

```
}
```

The cluster is responsible for managing its members (data points) at any point of the iterative computation of the K-means. It is assumed that a cluster will never contain the same data points twice. The two key methods in the class Cluster are:

- `moveCenter`: Re-computes the centroid of a cluster
- `stdDev`: Computes the standard deviation of the distance between all the observation members and the centroid

The constructor of the Cluster class is implemented by the `apply` method in the companion object (refer to the *Class constructors template* section under *Source code consideration* in the *Appendix*):

```
def apply[T: ToDouble] (
    center: Array[Double]
  ): Cluster[T] = new Cluster[T](center)
}
```

Let us look at the `moveCenter` method. It creates a new cluster from the existing members with a new centroid. The computation of the values of the centroid requires the transposition of the matrix of observations by features into a matrix of features by observations (line 3). The new centroid is computed by normalizing the sum of each feature across all the observations by the number of data points (line 4):

```
def moveCenter(xt: Vector[Array[T]])(
  implicit m: Manifest[T], num: Numeric[T]
): Cluster[T] = {
  val s = transpose(members.map(xt(_)).toList).map(_.sum) //3
  Cluster[T](
    s.map(implicitly[.ToDouble[T]].apply(_y(_) / members.size)) //4
  )
}
```

The `stdDev` method computes the standard deviation 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 invocation (line 5). It is then loaded into a statistics instance to compute the standard deviation (line 6). The function to compute the distance between the center and an observation is an argument of the method. The default distance is Euclidean:

```

def stdDev(xt: Vector[Array[T]],
           distance: DistanceFunc
): Double = {
  val ts = members.map(xt(_)).map(distance(center,_))//5
  Stats[Double](ts).stdDev //6
}

```

Note

Cluster selection

There are different ways to select the most appropriate cluster when reassigning an observation (updating its membership). In this implementation, we select the cluster with the larger spread or lowest density. An alternative is to select the cluster with the largest membership.

Initializing clusters

The initialization of the cluster centroids is important to ensure fast convergence of 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. Compute the index of the feature $\{x_{k,0}, x_{k,1} \dots x_{k,n}\}$ with the maximum standard deviation.
3. Rank the observations by their increasing value of standard deviation for the dimension k .
4. Divide the ranked observations set equally into K sets $\{S_m\}$.
5. Find the median values $\text{size}(S_m)/2$.
6. Use the resulting observations as initial centroids.

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

```

type U = Vector[Array[T]]
type V = KmeansModel[T]

def initialize(xt: U): V = {
    val stats = statistics(xt) //7
    val mxSDevDim = stats.indices.maxBy(stats(_).stdDev) //8
    val rankedObs = xt.zipWithIndex
        .map{ case (x, n) =>
            (implicitly[ToDouble[T]].apply(x(mxSDevDim)), n)
        }.sortBy(_.1) //9

    val halfSegSize = ((rankedObs.size>>1)/config.K)
                    .floor.toInt //10
    val centroids = rankedObs
        .filter( isContained( _, halfSegSize, rankedObs.size) )
        .map{case(x, n) => xt(n)} //11

    centroids.aggregate(List[Cluster[T]]())((xs, s) => {
        val c = s.map(implicitly[ToDouble[T]].apply(_))
        Cluster[T](c) :: s, _ :::: _) //12
    })
}
}

```

The **statistics** method on time series of type `Vector[Array[T]]` is defined in the *Time Series* section of [Chapter 3, Data Pre-processing](#) (line 7). The dimension (or feature) with the maximum variance or standard deviation `mxSDevDim` is computed by using the method `maxBy` on a `stats` instance (line 8). Then the observations are ranked by the increasing value of the standard deviation, `rankedObs` (line 9).

The ordered sequence of observations is then broken into `xt.size/config.K` segments (line 10) and the indices of the `centroids` are selected as the mid-point (or median) observations of those segments using the filtering condition `isContained` (line 11):

```

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

```

Finally, the list of clusters is generated using an aggregate call on the set of `centroids` (line 12).

Step 2 – Clusters 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 method, `assignToClusters`:

```
def assignToClusters(
    xt: U,
    clusters: V,
    members: Array[Int]): Int = {
    xt.zipWithIndex.filter{ case(x, n) => { //13
        val nearestCluster = getNearestCluster(clusters, x) //14
        val reassigned = nearestCluster != members(n)
        clusters(nearestCluster) += n //15
        members(n) = nearestC/16
        reassigned
    }}.size
}
```

The core of the assignment of observations to each cluster is the filter on the time series (line 13). The filter computes the index of the closest cluster and checks if the observation is to be reassigned (line 14). The observation at index `n` is added to the nearest cluster, `clusters(nearestCluster)` (line 15). The current membership of the observations is then updated (line 16).

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

```
def getNearestCluster(model: V, x: Array[T]): Int =
    model.zipWithIndex./:((Double.MaxValue, 0)) {
    case (p, (c, n)) => {
        val measure = distance(c.center, x) //17
        if(measure < p._1) (measure, n) else p
    }}._2
```

A fold is used to extract from the list of clusters the cluster that is closest to the observation `x` using the `distance` metric defined in the K-means constructor (line 17).

As with other data processing units, the extraction of K-means clusters is encapsulated in a data transformation so clustering can be integrated into a workflow using the composition of mixins described in the *Composing mixins to build workflow* section in [Chapter 2, Data Pipelines](#).

Tip

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 re-initialized and the iterative (or recursive) execution needs to be re-started.

The transformation $|>$ requires the computation of the standard deviation of the distance of the observations related to the centroid c and is computed in the method `stdDev`:

```
def stdDev[T: ToDouble] (
    model: KMeansModel[T],
    xt: Vector[Array[T]],
    distance: DistanceFunc[T]): DblVector =
  model.map(_.stdDev(xt, distance)).toVector
```

Note

Centroid versus mean

The terms centroid and mean refer to the same entity: the center of a cluster. This chapter uses these two terms interchangeably.

Step 3 – Reconstruction error minimization

The clusters are initialized with a predefined set of observations as their members. The algorithm updates the membership of each cluster by minimizing the total reconstruction error. There are two effective strategies to execute the K-means algorithm:

- Tail recursive execution
- Iterative execution

Creating K-means components

Let us declare the K-means algorithm class, `KMeans`, with its public methods. `KMeans` implements a data transformation, `ITransform`, using an implicit model extracted from a training set and described in the *Monadic data transformation* section of [Chapter 2, Data Pipeline](#) (line 18). The configuration of type `KMeansConfig` consists of the tuple `(K, maxIters)`, with `K` being the number of clusters and `maxIters` the maximum number of iterations allowed for the convergence of the algorithm:

```
case class KMeansConf[K: Int, maxIters: Int)
```

The `KMeans` class takes three arguments:

- `Config`: The configuration for execution of the algorithm
- `Distance`: The function to compute the distance between any observation and a cluster centroid
- `xt`: The training set

The implicit conversion of type `T` to a `Double` is implemented as a view bound. The instantiation of the `KMeans` class initializes type `V` of the output from K-means as `Cluster[T]` (line 20). The instance `num` of the class `Numeric` has to be passed implicitly as a class parameter because it is required by the `sortWith` invocation in `initialize`, the `maxBy` method, and the `Cluster.moveCenter` method (line 19). The `Manifest` is required to preserve the type erasure for `Array[T]` in the JVM:

```
class KMeans[@specialized(Double) T: ToDouble] (
    config: KMeansConfig, //18
    distance: DistanceFunc[T],
    xt: Vector[Array[T]]
)(implicit m: Manifest[T], num: Numeric[T]) //19
extends ITransform[Array[T], Cluster[T]] with Monitor[T] {

    type V = Cluster[T] //20
    val model: Option[KMeansModel[T]] = train
    def train: Option[KMeansModel[T]] = ???
    override def |> : PartialFunction[Array[T], Try[V]]
    ...
}
```

The `model` of type `KMeansModel` is defined as the list of clusters extracted through training.

Tail recursive implementation

The transformation or clustering function is implemented by the training method `train` that creates a partial function with `Vector[Array[T]]` as input and `KMeansModel[T]` as output:

```
def train: Option[KMeansModel[T]] = Try {
    // STEP 1
    val clusters = initialize(xt) //21
    if( clusters.isEmpty) /* ... */
    else {
        // STEP 2
        val members = Array.fill(xt.size)(0)
        assignToClusters(xt, clusters, members) //22
        var iters = 0
        if(iters >= config.maxIters) throw new IllegalStateException(
            // STEP 3
            update(clusters, xt, members) //23
        )
    } match { /* ...*/
}
```

The K-means training algorithm is implemented through the following three steps:

1. Initialize the clusters centroid using the method `initialize` (line 11)
2. Assign observations to each of the clusters through the method `assignToClusters` (line 22)
3. Re-compute the total error reconstruction using the recursive method `update` (line 23)

The computation of the total error reconstruction is implemented as a tail recursive method, `update`:

```
@tailrec
def update(clusters: KMeansModel[T],
           xt: U,
           members: Array[Int]): KMeansModel[T] = { //24
```

```

val newClusters = clusters.map(c => {
    if( c.size > 0) c.moveCenter(xt) //25
    else clusters.filter( _.size >0)
        .maxBy(_.stdDev(xt, distance)) //26
})
iters += 1
if( iters >= config.maxIters ||           //27
    assignToClusters(xt, newClusters, members) ==0)
    newClusters
else update(newClusters, xt, membership) //28
}

```

The recursion takes three arguments (line 24):

- The current list of `clusters` that updated during the recursion
- The input time series, `xt`
- The indices of membership to the clusters, `members`

A new list of clusters, `newClusters`, is computed by either recalculating each centroid if the cluster is not empty (line 25) or evaluating the standard deviation of the distance of each observation relative to each centroid otherwise (line 26). The execution exits when either the maximum number of recursive calls, `maxIters`, is reached or there are no more observations reassigned to a different cluster (line 27). Otherwise, the method invokes itself with an updated list of clusters (line 28).

Iterative implementation

The implementation of iterative execution is presented for informational purposes. It follows the same sequence of calls as with the recursive implementation. The new clusters are computed (line 29) and the execution exits when either the maximum number of allowed iterations is reached (line 30) or there are no more observations reassigned to a different cluster (line 31):

```

val members = Array.fill(xt.size) (0)
assignToClusters(xt, clusters, members)
var newClusters = List.empty[Cluster[T]]

(0 until maxIters).find( _ => { //29
    newClusters = clusters.map( c => { //30

```

```

        if( c.size > 0)  c.moveCenter(xt)
        else clusters.filter( _.size > 0)
                      .maxBy( _.stdDev(xt, distance) )
    })
    assignToClusters(xt, newClusters, members) > 0 //31
}).map(_ => newClusters)

```

The density of the clusters is computed in the KMeans class as follows:

```

def density: Option[DblVec] = model.map( _.map(c =>
  c.getMembers.map(xt(_)).map( distance(c.center, _) )).sum)

```

Step 4 – Classification

The objective of the classification is to assign an observation to a cluster with the closest centroid:

```

override def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T] if( x.length == dimension(xt) &&
                        model.isDefined) =>
    Try(model.map(_.minBy(c => distance(c.center, x))).get)
}

```

The most appropriate cluster is computed by selecting the cluster c whose center is the closest to the observation x using the `minBy` higher order method.

Curse of dimensionality

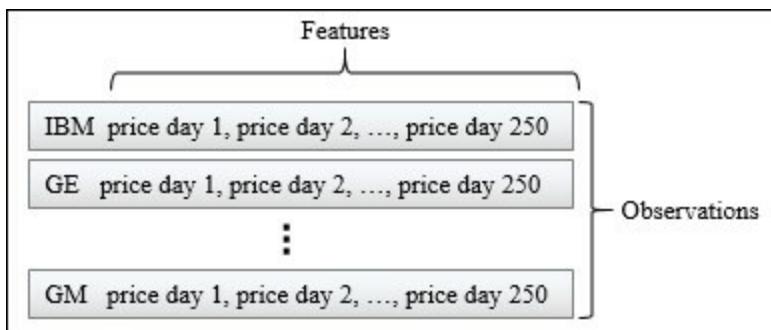
A model with a significant number of features (high dimension) requires a larger number of observations in order to extract relevant and reliable clusters. K-means clustering for very small datasets < 50 produces models with high bias and a limited number of clusters, which are affected by the order of observations [4:5]. I have been using the following simple empirical rule of thumb for a training set of size n , expected K clusters, and N features: $n < K.N$:

Note

Dimensionality and size of the training set

The issue of sizing the training set given the dimensionality of a model is not specific to unsupervised learning algorithms. All supervised learning techniques face the same challenge to set up a viable training plan.

Whichever empirical rule you follow, such a restriction is an issue particularly for analyzing stocks using historical quotes. Let us consider our examples of using technical analysis to categorize stocks according to their price behavior over a period of one year (or approximatively 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 numbers of observations; among them are:

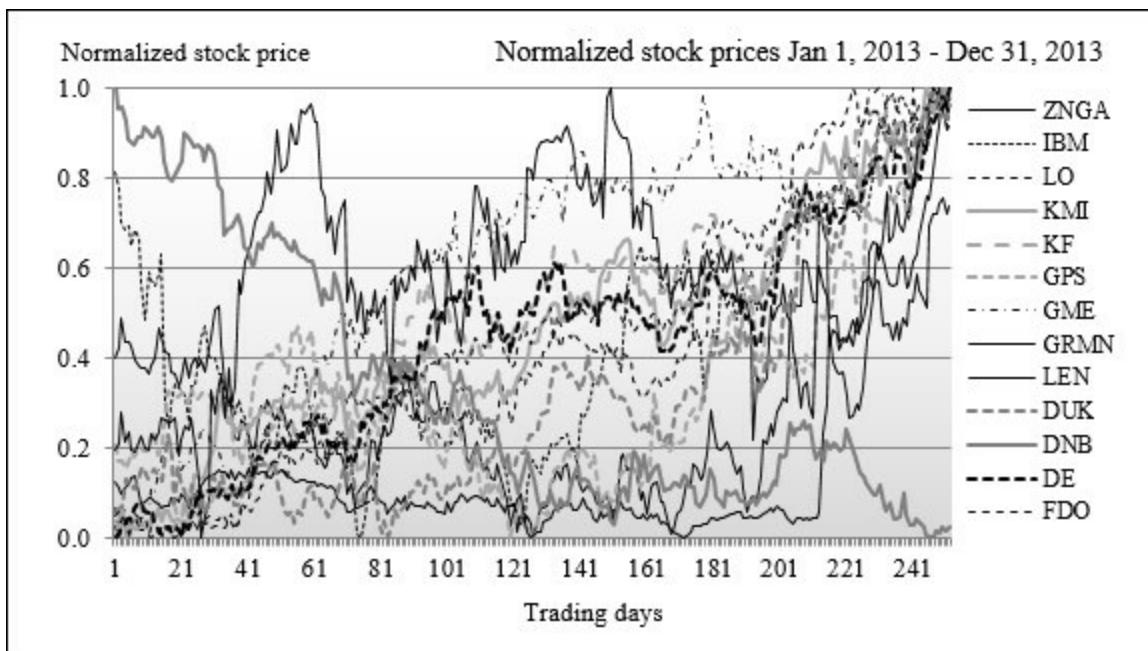
- Sampling the trading data without losing a significant amount of information from the raw data, assuming the distribution of observations follows a known probability density function.
- Smoothing the data to remove the noise, as seen in [Chapter 3, Data Pre-processing](#), assuming the noise is Gaussian. In our test, a smoothing technique will remove the price outliers for each stock and therefore reduce the number of features (trading sessions). This approach differs from the first (sampling) technique because it does not require that the dataset follow a known density function. On the other hand, the reduction of features could be less significant.

These approaches are workaround solutions at best, used for the sake of this tutorial. You need to consider the quality of your data before applying these techniques to actual commercial applications. The principal component

analysis introduced in the last paragraph is one of the reliable dimensions reduction techniques.

Evaluation

The objective is to extract clusters from a set of stocks' price actions during a period between January 1 and December 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:



Price action of a basket of stocks used in K-means clustering

The key is to select the appropriate features prior to clustering and the time window to operate on. It would make sense to consider the entire historical price over the 252 trading days as a feature. However, the number of observations (stocks) is too limited to use the entire price range. The observations are the stock closing price for each trading session between the 80th to the 130th trading day. The adjusted daily closing prices are normalized using their respective minimum and maximum values.

First let us create a simple method to compute the density of the clusters:

```

val MAX_ITERS = 150

def getDensity(  

    K: Int,  

    obs: Vector[Array[Double]]): DblVec =  

    KMeans[Double](KMeansConfig(K, MAX_ITERS))  

        .density.getOrElse(Vector.empty) //method density g  

type INPUT = Array[String] => Double

val START_INDEX = 80; val NUM_SAMPLES = 50 //33
val extractor = adjClose :: List[INPUT]() //34
val symbolFiles = DataSource.listSymbolFiles(path) //35

for {
    prices <- getPrices //36
    values <- Try( getPricesRange(prices) ) //37
    stdDev <- Try( ks.map(getDensity(_, values.toVector))) //38
    pfnKmeans <- Try { //39
        KMeans[Double](KMeansConfig(5, MAX_ITERS), values) |>
    }
    if(pfnKmeans.isDefined)
        predict <- pfnKmeans(values.head) //40
} yield {
    val results = s"""Daily prices ${prices.size} stocks"  

        | \nClusters density ${stdDev.mkString(", ")}"""
    .stripMargin
    show(results)
}

```

As mentioned earlier, the cluster analysis applies to the closing price in the range between the 80th and 130th trading day (line 33). The function `extractor` retrieves the adjusted closing price for a stock from the Yahoo financials pages `YahooFinancials` (line 34). The list of stock tickers (or symbols) is extracted as the list of CSV filenames located in the `path` (line 35). For instance, the ticker symbol for General Electric Corp. is `GE` and the data is located in `GE.csv`.

The execution extracts 50 daily prices using `DataSource` and then filters out the incorrectly formatted data using a filter (line 36):

```

type XVSeriesSet = Array[Vector[Array[Double]]]

def getPrices: Try[XVSeriesSet] = Try {
    symbolFiles.map( DataSource(_, path) |> extractor )

```

```

        .filter( _.isSuccess ).map( _.get)
    }
}

```

The historical stock prices for the trading session between the 80th and 130th days are generated by the `getPricesRange` closure (line 37):

```

def getPricesRange(prices: XVSeriesSet) =
    prices.view.map(_.head.toArray)
        .map( _.drop(START_INDEX).take(NUM_SAMPLES) )
}

```

It computes the density of the clusters by invoking the method `density` for each values `ks` of the number of clusters (line 38).

The partial classification function, `pfnKmeans`, is created for a five-cluster `KMeans` (line 39) and is then used to classify one of the observation (line 40).

The results

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

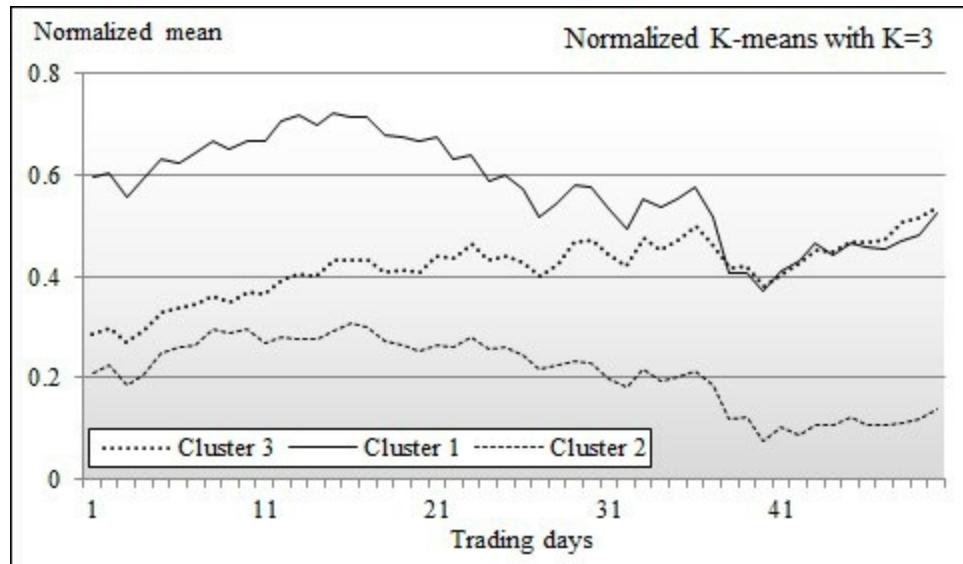


Chart of means of clusters using K-means K=3

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 clusters' mean vectors is almost constant: 0.37. The cluster with 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 easily be explained by the fact that the time window or trading period, the 80th to the 130th trading day, corresponds to the shift in the monetary policy of the federal reserve regarding the quantitative easing program. Here is the partial list of stocks for each of the clusters whose centroid values are displayed on the chart:

Cluster 1	AET, AHS, BBBY, BRCM, C, CB, CL, CLX, COH, CVX, CYH, DE, ...
Cluster 2	AA, AAPL, ADBE, ADSK, AFAM, AMZN, AU, BHI, BTU, CAT, CCL, ...
Cluster 3	ADM, ADP, AXP, BA, BBT, BEN, BK, BSX, CA, CBS, CCE, CELG, CHK, ...

Let's evaluate the impact of the number of clusters K on the characteristics of each cluster.

Tuning the number of clusters

We will repeat the previous test on the 127 stocks in 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:

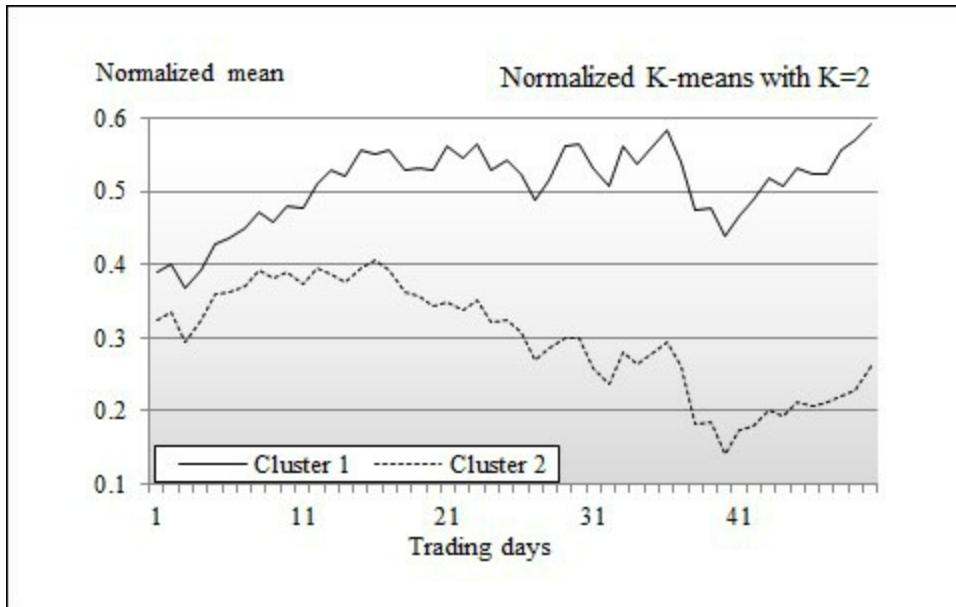


Chart of means of clusters using K-means K=2

The chart of the results of the K-means algorithms with two clusters shows that the mean vector for the cluster labeled 2 is similar to the mean vector labeled 3 on the chart with $K = 5$ clusters. However, the cluster with the mean vector 1 somewhat reflects the aggregation or summation of the mean vectors for the clusters 1 and 3 in the chart $K=5$. The **aggregation effect** explains why the standard deviation for the cluster 1, 0.55, is twice as much as the standard deviation for the cluster 2, 0.28.

The mean (or centroid) vector for each cluster is plotted as follows for $K = 5$:

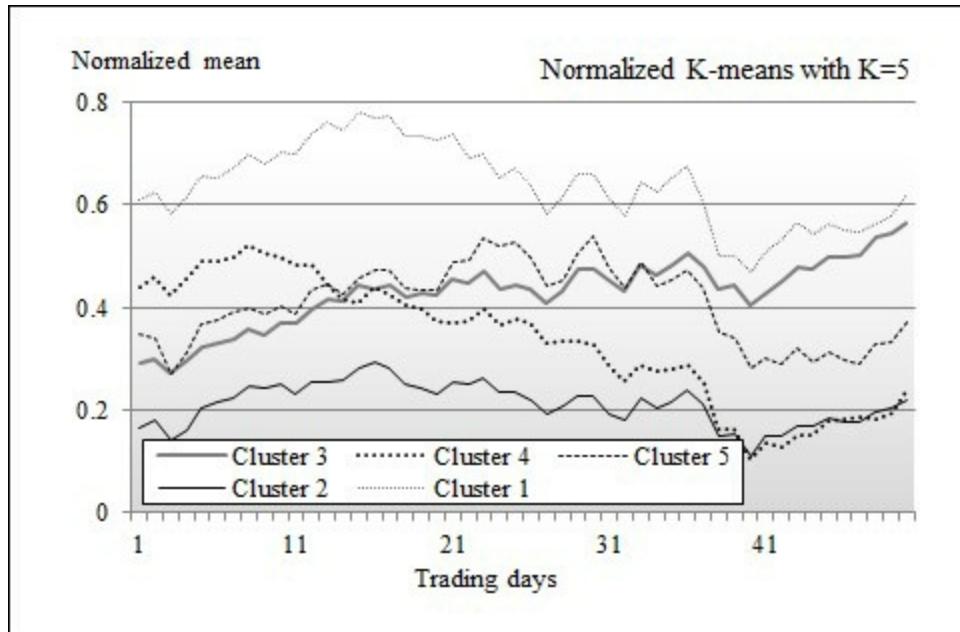


Chart of means of clusters using K-means K=5

In this chart, we can assess that 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 behavior 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 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:

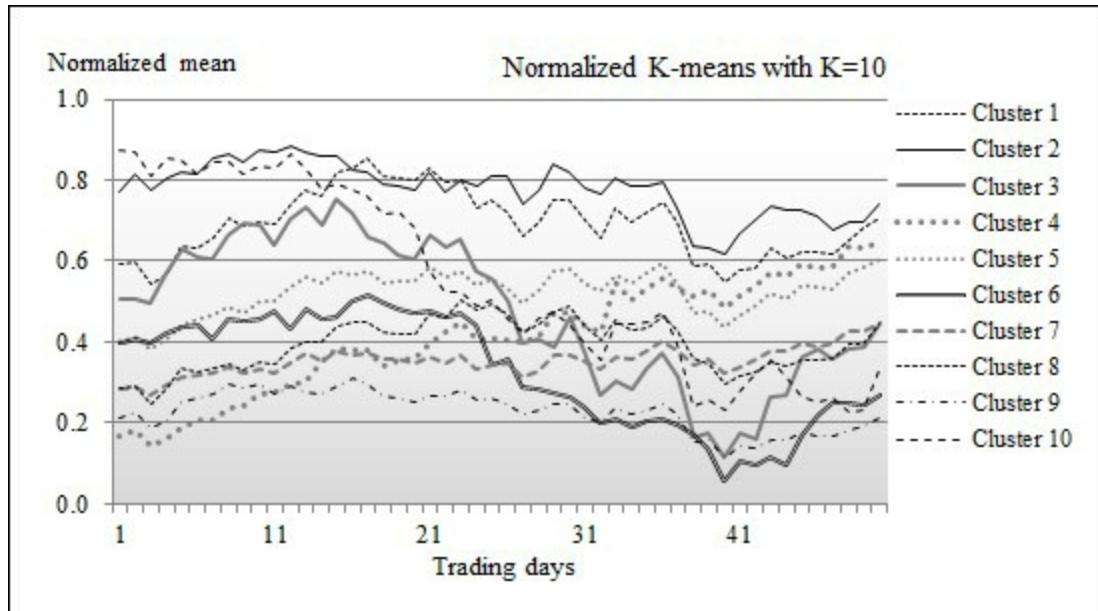


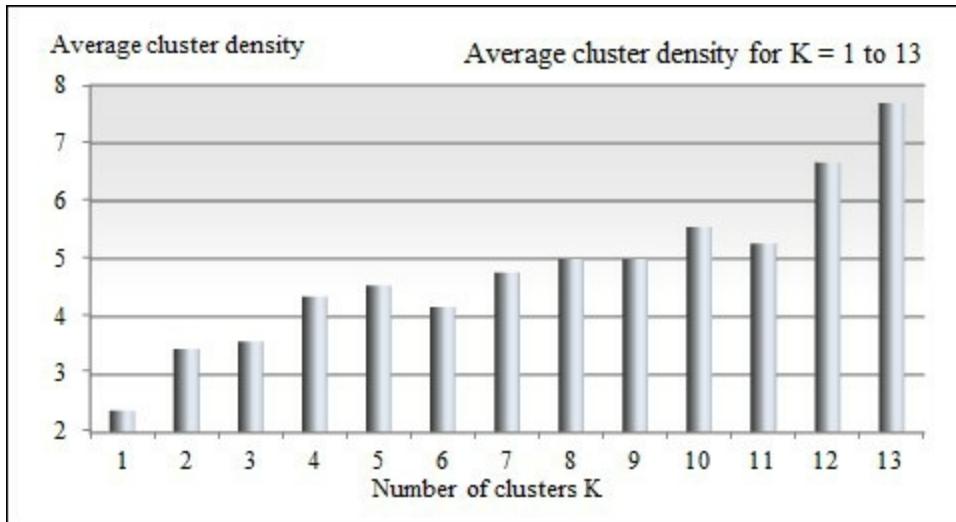
Chart of means of clusters using K-means K=10

The means for clusters 1, 2, and 3 seen in the first chart for the case $K = 2$ are still visible. It is fair to assume that these are very likely the most reliable clusters. These clusters happen to have a low standard deviation or high density.

Let us 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) (M6):

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

The density of the cluster is plotted against the number of clusters with $K = 1$ to 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 ($K=6, K=11$). 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 a test you find that one of the clusters CC contains most of the commodity-related stocks, then you can select another commodity related stock SC which is not part of the first batch and run the entire clustering algorithm again. If SC is a subset of CC, then K-means has performed as expected. If this is the case, you should run a new set of stocks, some of which are commodity related, and measure the number of true positives, true

negatives, false positives, and false negatives.

The values for the precision, recall, and F_1 measures introduced in the *F-score for multinomial classification* section under *Assessing a model* in [Chapter 2, Data Pipelines](#), can confirm whether the tuning parameters and labels you selected for your cluster are indeed correct.

Tip

F 1 validation for K-means

The quality of the clusters as measured by the F_1 score depends on the rule, policy, or formula used to label observations (that is, label a cluster with the industry with the highest relative percentage of stocks in the cluster). This process is quite subjective. The only sure way to validate a methodology is to evaluate several labeling schemes and select the one that generates the highest F_1 score.

An alternative to the measure of the homogeneity of the distribution of observations across the clusters is to compute the statistical entropy as follows:

$$H(p) = - \sum_c p_c \cdot \log_2 p_c$$

The low entropy value indicates that clusters have a low level of impurity. Entropy can be used to find the optimal number of clusters K.

We reviewed some of the tuning parameters that affect the quality of the results of the K-means clustering:

- Initial selection of centroid
- Number of clusters K

In some cases, the similarity criterion (that is, Euclidean distance or cosine distance) can have an impact on the cleanliness or density of the clusters.

A final and important consideration is the computational complexity of the K-means algorithm; the last section of this chapter describes some of the performance issues with K-means and possible remedies.

Despite its many benefits, the K-means algorithm does not handle missing data or unobserved features very well. Features that depend on each other indirectly may in fact depend on a common hidden (also known as latent) feature. The expectation-maximization algorithm described in the next section addresses some of these limitations.

Expectation-Maximization (EM)

The EM was originally introduced to estimate the maximum likelihood in the case of incomplete data [4:7]. The EM algorithm is an iterative method to compute the model features that maximize the likely estimate for observed values, considering unobserved values.

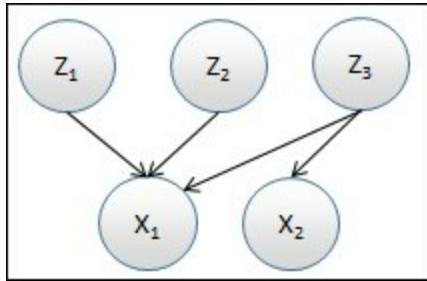
The iterative algorithm consists of computing:

- 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 EM algorithm is applied to solve clustering problems by if 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_i can be visualized as the behavior (or symptoms) of a model (observed), X , for which Z are the root-cause 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 are described here:

Note

Maximization of log likelihood

M7: If $x = \{x_i\}$ is a set of observed features associated with latent features $z = \{z_i\}$, the probability for the feature x_i of observation x given a model parameter θ is:

$$p(x_i | \theta) = - \sum_z p(x_i, z | \theta)$$

M8: The objective is to maximize the likelihood $L(\theta)$:

$$\mathcal{L}(\theta) = \sum_{i=0}^{N-1} \log \left\{ \sum_z p(x_i, z | \theta) \right\} \quad \tilde{\theta} = \operatorname{argmax} \mathcal{L}(\theta)$$

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 (Initial-step).
2. The computation of the expectation of the log likelihood at iteration t (E-step).
3. The maximization of the expectation at iteration t (M-step).

Note

E-step

M9: The expectation Q of the complete data log likelihood for the model parameters θ^n at iteration n is computed using the posterior distribution of latent variable z , $p(z|x, \theta)$ and the joint probability of the observation and the latent variable:

$$Q(\theta, \theta^n) = \sum_z p(z|x_i, \theta^n) \cdot \log p(x_i, z|\theta)$$

M-step

M10: The expectation function Q is maximized for the model features θ to compute the model parameters θ^{n+1} for the next iteration:

$$\theta^{n+1} = \arg \max_{\theta} Q(\theta, \theta^n) \quad |\theta^{n+1} - \theta^n| < \varepsilon$$

A formal, detailed but short mathematical formulation of the EM algorithm can be found in the S. Borman's tutorial [4:9].

Implementation

Let's implement the three steps (Initial-step, E-step, and M-step) in Scala. The internal calculations of the EM algorithm are a bit complex and overwhelming. You may not benefit much from the details of a specific implementation such as computation of the eigenvalues of the covariance matrix of the expectation of the log likelihood. This implementation hides some complexity by using the Apache Common Math library package [4:10].

Tip

Inner workings of EM

You may want to download the source code for the implementation of the EM algorithm in the Apache Commons Math library, if you need to understand the condition for which an exception is thrown.

The expectation-maximization algorithm of type `MultivariateEM` is implemented as a data transformation of type `ITransform` as described in the *Monadic data transformation* section of [Chapter 2, Hello World!](#). The two arguments of the constructors are the number of clusters (or gauss distribution) `K` and the training set `xt` (line 12). The constructor initializes the type `V` of output as `MCluster` (line 21):

```
type V = EMCluster //1

class MultivariateEM[@specialized(Double) T: ToDouble] (
  K: Int,
  xt: Vector[Array[T]]
) extends ITransform[Array[T], V] with Monitor[T] { //2

  val model: Option[EMModel] = train //3
  override def |> : PartialFunction[Arraon[U, Try[V]]
}
```

The multivariate expectation-maximization class has a model that consists of a list of EM clusters of type `EMCluster`. The `Monitor` trait is used to collect

profiling information during training (refer to the *Monitor* section under *Helper classes* in the *Appendix*).

The information about an EM cluster, `EMCluster`, is defined by a key, the centroid, or `means` value and the `density` of the cluster that is the standard deviation of the distance of all the data points to the mean (line 4):

```
case class EMCluster(  
    key: Double,  
    means: Array[Double],  
    density: Array[Double]) //4  
type EMMModel = List[EMCluster]
```

The implementation of the EM algorithm in the method `train` uses the Apache Commons Math `MultivariateNormalMixture` for the Gaussian mixture model and `MultivariateNormalMixtureExpectationMaximization` for the EM algorithm:

```
def train: Option[EMMModel] = Try {  
    val data: DblMatrix = xt.map(  
        _.map(implicitly[ToDouble[T]].apply(_))  
    ).toArray //5  
  
    val multivariateEM = new EM(data)  
    multivariateEM.fit( estimate(data, K) ) //6  
  
    val newMixture = multivariateEM.getFittedModel //7  
    val components = newMixture.getComponents.toList //8  
    components.map(p => EMCluster(  
        p.getKey,  
        p.getValue.getMeans,  
        p.getValue.getStandardDeviations)  
    ) //9  
} match {/* ... */}
```

Let's look at the main method `train` of the wrapper class, `MultivariateEM`. The first step is to convert the time series into a primitive Matrix of Double with observations, with historical quotes as rows and the stock symbols as columns.

The time series `xt` of type `Vector[Array[T]]` is converted to a `DblMatrix` through an induced implicit conversion (line 5).

The initial mixture of Gaussian distributions can be provided by the user or can be extracted from the dataset, `estimate` (line 6). The `getFittedModel` triggers the M-step (line 7).

Note

Conversion from Java and Scala collections

Java primitives need to be converted to Scala types using the package `import scala.collection.JavaConversions`. For example, `java.util.List` is converted to `scala.collection.immutable.List` by invoking the method `asScalaIterator` of the class `WrapAssScala`, one of the base traits of `JavaConversions`.

The Apache Commons Math method, `getComponents`, returns a `java.util.List` that is converted to a `scala.collection.immutable.List` by invoking the method `toList` (line 8). Finally, the data transform returns a list of cluster information of type `EMCluster` (line 9).

Tip

Third-party library exceptions

Scala does not enforce declaration of exception as part of the signature of a method. Therefore, there is no guarantee that all types of exceptions will be caught, locally. This problem occurs when exceptions are thrown from a third-party library in two scenarios:

- The documentation of the API does not list all the types of exceptions
- The library is updated and a new type of exception is added to a method
- One easy workaround is to leverage the Scala exception handling mechanism:

```
Try { ... }
  match {
    case Success(results) => ...
    case Failure(exception) => ...
  }
```

Classification

The classification of new observations or data points is implemented by the method `|>:`

```
override def |> : PartialFunction[Array[T], Try[V]] = {  
  case x:Array[T] if(isModel && x.length == dimension(xt)) =>  
    Try( model.map(_.minBy(c =>  
      euclidean[Double, Double](  
        c.means,  
        x.map(z => implicitly[ToDouble[T]].apply(z))))  
    )  
    .getOrElse(.means,x)).get  
}
```

The method `|>` is similar to the `KMeans.|>` classifier.

Testing

Let's apply the `MultivariateEM` class to the clustering of the same 127 stocks used in evaluating the K-means algorithm.

As discussed in the 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' prices and apply a simple sampling method. The maximum sampling rate is restricted by the frequencies in the spectrum of noises of different types on the historical price of every stock.

Tip

Filtering and sampling

The pre-processing 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 prices of all the stocks share the same noise characteristics. The noise pattern in the price of momentum and heavily traded stocks is certainly different from blue chips securities with a strong ownership, held by large mutual funds.

The sampling rate should take into account the frequencies spectrum of the noise. It should be set as at least twice the frequency of the noise with the lowest frequency.

The object of the test is to evaluate the impact of the sampling rate, `samplingRate`, and the number of clusters `K` used in the EM algorithm:

```
val K = 4; val period = 8
val smAve = SimpleMovingAverage[Double](period) //10
val pfnSmAve = smAve |>      //11

val obs = symbolFiles.map(sym => (
  val pfnSrc = DataSource(sym, path, true, 1) |>
    for {
```

```

xs <- pfnsSrc(extractor) //12
if(pfnsSmAve.isDefined)
  values <- pfnsSmAve(xs.head) //13
y <- Try {
  values.view.indices.drop(period+1).toVector
    .filter(_ % samplingRate == 0)
    .map(values(_)).toArray //14
}
} yield y).getOrElse(Array.empty[Double])) //15

```

The first step is to create a simple moving average with a predefined `period` (line 10) as described in the *Simple moving average* section of [Chapter 3, Data Pre-Processing](#). The test code instantiates the partial function, `pfnsSmAve` that implements the moving average computation (line 11). The symbols of the stocks under consideration are extracted from the names of the files in the path directory (line 12).

The execution of the moving average (line 13) generates a set of smoothed values that are sampled given a sampling rate, `samplingRate` (line 14). Finally, the expectation maximization is instantiated to cluster the sampled data in the `em` method (line 15):

```

def em(K: Int, obs: DblMatrix): Int = {
  val em = MultivariateEM[Double](K, obs.toVector) //16
  show(s"${em.toString}") //17
}

```

The method `em` instantiates the EM algorithm for a specific number of clusters `K` (line 16). The content of the model is displayed by invoking `MultivariateEM.toString`. The results are aggregated and then displayed in a textual format on the standard output (line 17).

The first test is to execute the EM algorithm with `K=3` clusters and a sampling period of 10 on data smoothed by a simple moving average with a period 8. The sampling of historical prices of the 127 stocks between January 1, 2013 and December 31, 2013 with a frequency of 0.1 Hertz produces 24 data points. The following chart displays the mean of each of the three clusters:

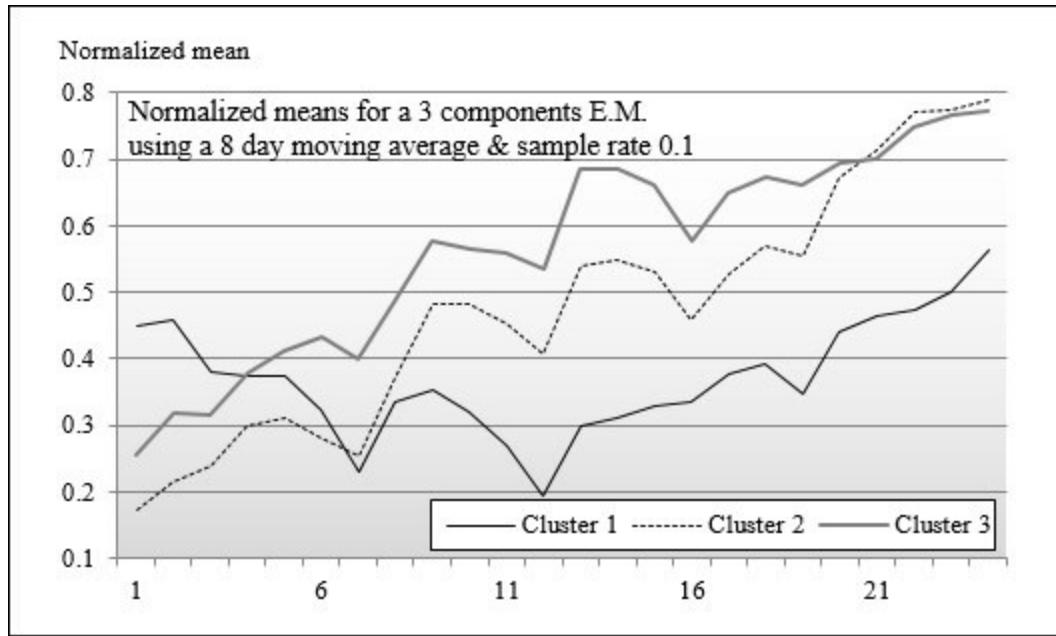


Chart of the normalized means per cluster using EM K=3

The mean vectors of clusters 2 and 3 have similar patterns, which may suggest that a set of three clusters is accurate enough to provide us with 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:

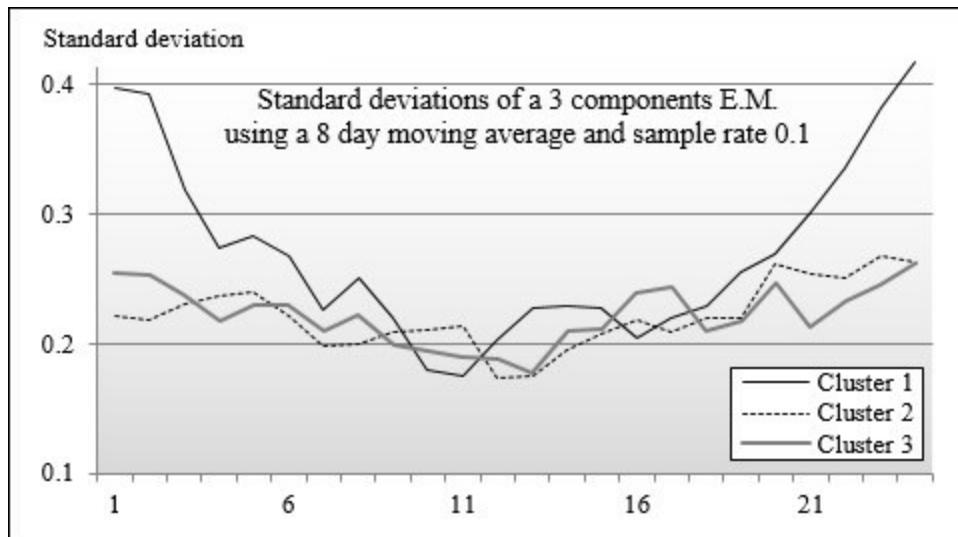


Chart of the normalized standard deviation per cluster using EM K=3

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 semi-homogenous group following the announcement from the Federal Reserve that the monthly quantity of bonds purchased as part of the quantitative easing program will be reduced soon.

Note

Relation to K-Means

You may wonder what the relation is 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 to online EM available if you ever have to deal with large datasets: **batch EM**, **stepwise EM**, **incremental EM**, and **Monte Carlo EM** [4:12].

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

Manifold learning for non-linear models is a technically challenging field with great potential in terms of dynamic object recognition [4:18].

The key point to remember is that unsupervised learning techniques are used:

- By themselves to extract structures and associations from unlabeled observations
- As a pre-processing stage to supervised learning by reducing the number of features prior to the training phase

The distinction between unsupervised and supervised learning is not as strict as you may think. For instance, the K-means algorithm can be enhanced to support classification.

In the next chapter, we will address the second use case and cover supervised learning techniques, starting with generative models.

Chapter 5. Dimension Reduction

As described in the *Assessing a model/overfitting* section of [Chapter 2, Data Pipelines](#), the indiscriminative reliance of a large number of features may cause overfitting; the model may become so tightly coupled with the training set that different validation sets will generate a vastly different outcome and quality metrics such as AuROC.

Dimension reduction techniques alleviate these problems by detecting features that have little influence on the overall model behavior.

This chapter introduces three categories of dimension reduction techniques with two implementations in Scala:

- Divergence with an implementation of the Kullback-Leibler distance
- Principal components analysis
- Estimation of low dimension feature space for nonlinear models

Other types of methodologies used to reduce the number of features such as regularization or singular value decomposition are discussed in future chapters.

But first, let's start our investigation by defining the problem.

Challenging model complexity

Without prior knowledge of the problem domain, data scientists include all possible features in their first attempt to create a classification, prediction, or regression model. After all, making assumptions is a poor and dangerous approach to reducing the search space. Models may require hundreds or thousands of features, adding complexity and significant computation costs to build and validate these models.

Noise-filtering techniques reduce the sensitivity of a model to the features that are associated with the sporadic behavior. However, these noise-related features are unknown prior to the training phase, and therefore cannot be completely discarded. Consequently, the training of a 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 sizes does not allow you to create an accurate model with any features.

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

- Statistical analysis solutions such as **Analysis of Variance (ANOVA)** for a small features set or estimation of divergences
- Regularization and shrinking techniques such as **L1 regularization** that are introduced in the *Regularization* section of [Chapter 9, Regression and Regularization](#)
- Algorithms that maximize the variance of a dataset by transforming the covariance matrix

The divergences

Fundamentally, ces are algorithms that compute the similarity between two probability distributions. In the field of information theory, divergences are used to estimate the minimum discrimination information.

Although divergences are not usually defined as dimension-reduction techniques, they are a vital tool for measuring the redundancy of information between features.

Let's consider a set of observations: X with a feature set $\{f_i\}$. Two features that are highly correlated generate redundant information (or information gains). Therefore, it is conceivable to remove one of these two features from the training set without incurring a loss of information.

The list of divergences is quite extensive and includes the following methods:

- **Kullback-Leibler (KL)** divergence estimates the similarity between two probability distributions [5:1]
- **Jensen-Shannon** metric extends the KL formula with symmetrization and boundary values [5:2]
- **Mutual information**, based on KL, measures the mutual dependence between two random variables as KL of their joint probability over the product of their marginal probability [5:3]
- **Bregman** distance is used for a continuous differentiable convex probability distribution. The Bregman divergence is used for reliability as an objective function for clustering [5:4]

Note

Divergence such as **Mutual information (MI)** is used for feature extraction.

The Kullback-Leibler divergence

The *Kullback-Leibler* divergence is the most commonly used and known divergence measure. It computes the relative entropy between two distributions. This divergence is also known as relative entropy (KL).

Overview

The reader can find a Spark implementation of the Kullback-Leibler divergence in the *Apache Spark extensibility* section of [Chapter 17, Apache Spark MLib](#).

In the context of measuring the similarity between two features, p and q, given the frequency of the distribution over observations Pi and Qi, the Kullback-Leibler is computed as follows:

$$D_{KL}(P\|Q) = \sum_{i=0}^{n-1} p(x_i) \cdot \log \frac{p(x_i)}{q(x_i)}$$

Let's implement the Kullback-Leibler algorithm to compute the similarity of the two datasets:

Implementation

Let's start by defining a generic `Divergence` trait:

```
trait Divergence {
    def divergence(nSteps: Int): Double
}
```

The KL algorithm is implemented by the `KullbackLeibler` class that takes three parameters, the two observed features p and q, and an optional normalization factor (line 1):

```

class KullbackLeibler[@specialized(Double) T: ToDouble] (
  p: Seq[T],
  q: Seq[T],
  normalized: Boolean = true //1
)(implicit ordering: Ordering[T]) extends Divergence { //2

  val (min, max): (Double, Double) = ( //3
    Math.min(p.min, q.min),
    Math.max(p.max, q.max)
  )
  override def divergence(nSteps: Int): Double = {...}
}

```

The class cannot assume that the features have `Scala.Double` as a data type. Therefore, the constructor defines the following items:

- Context-bound parameterized type `T`
- An implicit ordering for the type `T` (line 2)

The frequency distribution for each dataset requires the computation of the minimum and maximum values across the two observations `p` and `q` (line 3).

The KL computation is implemented by the overridden `divergence` method:

```

override def divergence(nSteps: Int): Double = {
  val freq1 = frequencies(p, nSteps) //4
  val freq2 = frequencies(q, nSteps) //5

  val kl = freq1.zip(freq2).map((f1, f2) => { //6
    val num = if (f2 == 0) 1 else f2 //8
    val den = if(f1 == 0) 1 else f1
    f1 - num * Math.log(num) / den
  })
  if(normalized) kl/Math.max(p.size , q.size) else kl //7
}

```

The method `divergence` computes the histograms for the two feature observations (lines 4 and 5). The KL formula is applied cumulatively to each pair of feature instances (`f1` and `f2`) (line 6) and summed by a fold method. The result value is optionally normalized (line 7).

The generation of the histogram is implemented by the `frequencies` method:

```
def frequencies(input: Seq[T], nSteps: Int): Array[Int] = {
    val histogram = new Histogram(min, max)
    histogram.frequencies(nSteps, input.map(toDouble(_)))
}
```

Tip

Handling null frequencies

There is no guarantee that the histograms have no null empty for which the computation would generate a `Double.NaN` value for (case of $\log(0)$ or $/0$). There are three options to handle this condition:

- Throwing an exception
- Discarding the particular observation
- Using Laplace smoothing and adding 1 to the frequency count (line 8)

The implementation of the ancillary class `Histogram` and related methods are available in the companion source code.

Testing

The purpose of the test is to evaluate the impact of the number of histogram buckets, `nSteps`, on the accuracy of the computation of the Kullback-Leibler distance as a reliable estimator of the similarity between two features. The algorithm is tested with two identical random gamma distributions (line 8):

```
val numPoints = 1000000
val nSteps = 100

def gammaDistribution(    //8
    shape: Double,
    scale: Double): Seq[Double] = {
    val gamma = new GammaDistribution(shape, scale)
    Seq.fill(numPoints)(gamma.density(Random.nextDouble))
}

val divergence = new KullbackLeibler[Double](
```

```

    gammaDistribution(2.0, 1.0),
    gammaDistribution(2.0, 1.0)
).divergence(nSteps)

```

The test produces the following output for various values of `nSteps`:

nSteps	KL-divergence
50	2.691
100	0.477
250	6.138e-4
500	7.902e-8

As expected, the divergence between the two datasets decreased significantly as the number of buckets used in the histogram increased.

The mutual information

The mutual information is a symmetric divergence that extends the Kullback-Leibler divergence. It measures the similarity between two random distributions, X and Y , by computing the KL divergence of the joint probability of these distributions over the product of their marginal probability, $p(X)$ and $p(Y)$. Contrary to the KL divergence, the Mutual information is symmetric:

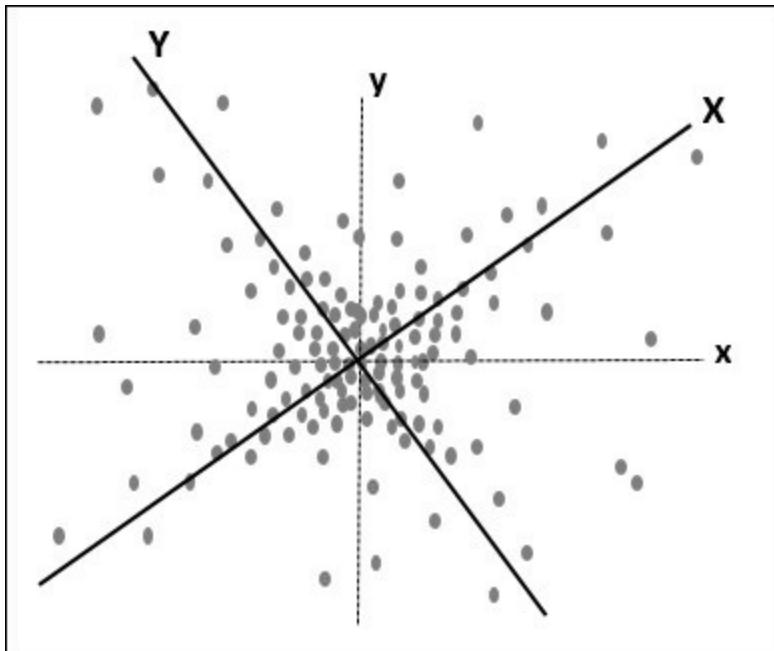
$$I(X;Y) = D_{kl} [p(X,Y), p(X)p(Y)] = \sum_{i=0}^{n-1} p(x_i, y_i) \log \frac{p(x_i, y_i)}{p(x_i)p(y_i)}$$

Principal components analysis (PCA)

The principal components analysis transforms an original set of features into a new set of features ordered by decreasing value of their variance. PCA enables the data scientist to select the features that have the most impact on the classification or prediction (features with the higher variance).

The original observations (vectors of feature instance) 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 in the following chart:



Visualization of the principal components for a two-dimensional model

The features x and y are converted into two variables, X and Y (that is rotation), to appropriately match the distribution of observations. The variable with the highest variance is known as the first principal component.

In the generic case of multiple features, the variable with the n^{th} highest variance is known as the n^{th} principal component. The dimension of the model (number of features selected) m for an observation space of dimension $n >> m$ is then defined as follows:

$$M = \left\{ f_{i:1,m} \left| \sum_{i=1}^m \sigma^2(f_i) < \mu \right. \right\}$$

Algorithm

I highly recommend the tutorial from **Lindsay Smith** [5:6] that describes the PCA algorithm in a very concrete and simple way using a two-dimensional model.

Note

PCA and covariance matrix:

M1: The covariance of two features X and Y with the observations set $\{x_i, y_i\}$ and their respective mean values is defined as:

$$\text{cov}(X, Y) = \frac{1}{n-1} \sum_{i=0}^{n-1} (x_i - \bar{x})(y_i - \bar{y})$$

Here, \bar{x} and \bar{y} are the respective mean values for the observations x and y .

M2: The covariance is computed from the *Z-score* of each observation:

$$x_i \leftarrow \frac{x_i - \bar{x}}{\sigma}$$

M3: For a model with n features x_i , the covariance matrix is defined as follows:

$$\Sigma_{\text{cov}} = \begin{vmatrix} \text{cov}(x_0, x_0) & \dots & \text{cov}(x_0, x_{n-1}) \\ \dots & \text{var}(x_i) & \dots \\ \text{cov}(x_{n-1}, x_0) & \dots & \text{cov}(x_{n-1}, x_{n-1}) \end{vmatrix}$$

M14: The transformation of x from x to X consists of computing the

eigenvalues of the covariance matrix:

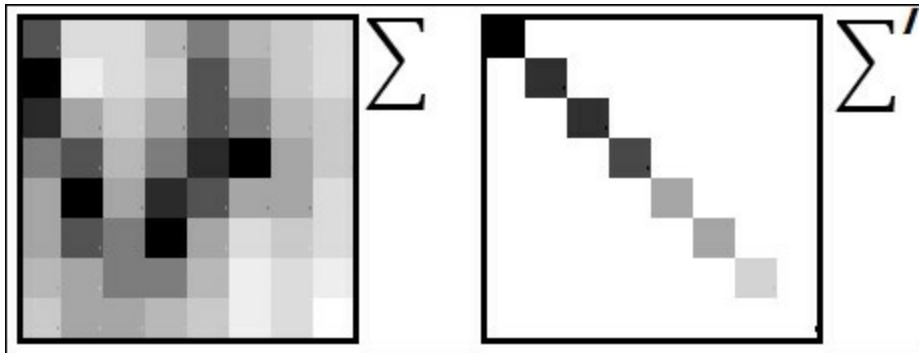
$$\Sigma' = W^T \cdot \Sigma_{cov} \cdot W$$

The eigenvalues are ranked by their decreasing order of variance. Finally, the m top eigenvalues for which the cumulative of variance exceeds a predefined threshold (percentage of the trace of the matrix) are the principal components.

The algorithm is implemented in five steps:

1. Compute the Z-score for the observations by standardizing the mean and standard deviation.
2. Compute the covariance matrix Σ for the original set of observations.
3. Compute the new covariance matrix Σ' for the observations with the transformed features by extracting the eigenvalues and eigenvectors.
4. Convert the matrix to rank eigenvalues by decreasing 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 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 as follows. The shades of gray used to represent the covariance value vary 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 (right-bottom section of the diagonal matrix) becomes insignificant.

Implementation

The principal components analysis can be easily implemented using the Apache common math library methods that compute the eigenvalues and eigenvectors. The `PCA` class is defined as a data transformation of type `ITransform`, (line 2) described in the *Monadic Data transformation* section of [Chapter 2, Data Pipelines!](#).

The `PCA` class has a single argument: the training set `xt` (line 1). The constructor defines the Z-Score function `norm` (line 3):

```
class PCA[@specialized(Double) T: ToDouble] (
  xt: Vector[Array[T]]      //1
) extends ITransform[Array[T], Double] with Monitor[T] { //2

  val model: Option[PCAModel] = train //3
  override def |> : PartialFunction[Array[T], Try[Double]]
}
```

The model for the PCA algorithm is defined by the case class `PCAModel` (line 3). The model `PCAModel` consists of the covariance matrix, `covariance` defined in the formula M1, and the array of `eigenvalues` computed in the formula M16:

```
case class PCAModel(covariance: DblMatrix,
                     eigenvalues: Array[Double])
```

The transformative method `|>` implements computation of the principal components (that is, the eigenvector and eigenvalues):

```
def train: Option[PCAModel] = zScores(xt).map(x => { //4
  val obs: DblMatrix = x.toArray
  val cov = new Covariance(obs).getCovarianceMatrix //5

  val transform = new EigenDecomposition(cov) //6
  val eigenVectors = transform.getV //7
  val eigenValues =
    new ArrayRealVector(transform.getRealEigenvalues)

  val covariance = obs.multiply(eigenVectors).getData //8
}
```

```
PCAModel(covariance, eigenValues.toArray)    //9
}) match {/* ... */}
```

The `train` method normalizes the input data using the Z-score transformation, `zScores` (M2) (line 4). Next, the method computes the `covariance` matrix from the normalized data (line 5). The `eigenVectors` are computed (line 6) and then retrieved using the `getV` method in Apache Commons Math class `EigenDecomposition` (line 7). The method computes the diagonal, transformed covariance matrix from the eigenvector (line 8). Finally, the data transformation returns an instance of the model of type `PCAModel` (line 9).

The predictive method `|>` consists of projecting an observation onto the principal components:

```
override def |> : PartialFunction[Array[T], Try[Double]] = {
  case x: Array[T]
    if(model.isDefined && x.length == dimension(xt)) =>
      Try( inner(x, model.get.eigenvalues) )
}
```

The `inner` method of the `Vector[Array[T]]` object computes the dot product of the values `x` and the model eigenvalues.

Test case

Let's apply the PCA algorithm to extract a subset of the features that represent some of the financial metric ratios of 34 S&P 500 companies. The metrics under consideration are:

- **Trailing price-to earnings ratio (PE)**
- **Price-to-sales ratio (PS)**
- **Price-to-book ratio (PB)**
- **Return on equity (ROE)**
- **Operation margin (OM)**

The financial metrics are described in the *Terminology* section under Finance 101 in *Appendix*.

The input data is specified with the following format as a tuple: Ticker symbol and an array of five financial ratios, *PE*, *PS*, *PB*, *ROE*, and *OM*:

```
val data = Vector[(String, Array[Double])] (  
    // 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 simply defined as follows:

```
val dim = data.head._2.size  
val input = data.map(_.2.take(dim))  
val pca = new PCA[Double](input) //11  
show(s"PCA model: ${pca.toString}") //12
```

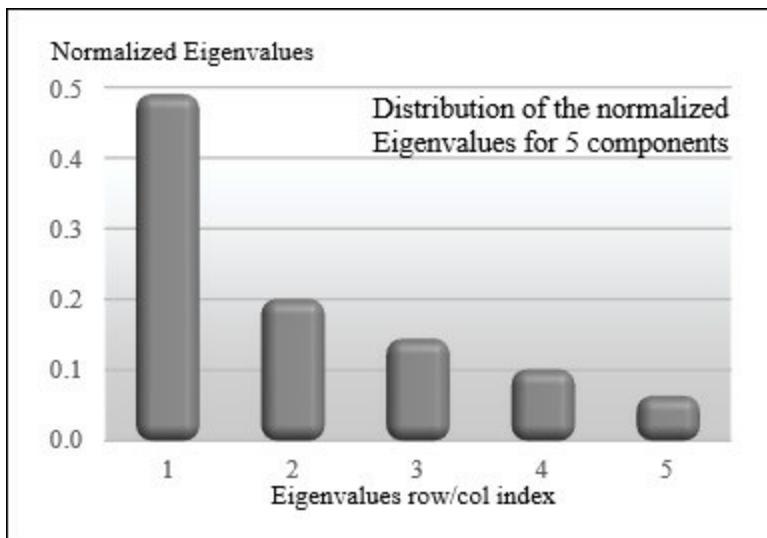
The PCA is instantiated with the input data (line 11) and then displayed in textual format (line 12).

Evaluation

The first test on the 34 financial ratios uses a model that has five dimensions. As expected, the algorithm produces a list of five ordered eigenvalues:

$2.5321, 1.0350, 0.7438, 0.5218, 0.3284$

Let's plot the relative value of the eigenvalues (that is the relative importance of each feature) on the following bar chart:



Distribution of eigenvalues in PCA for five dimensions

The chart shows that three out of five features account for 85 percent of the total variance (trace of the transformed covariance matrix). I invite you to experiment with different combinations of these features. The selection of a subset of the existing features is as simple as applying the Scala's `take` or `drop` methods:

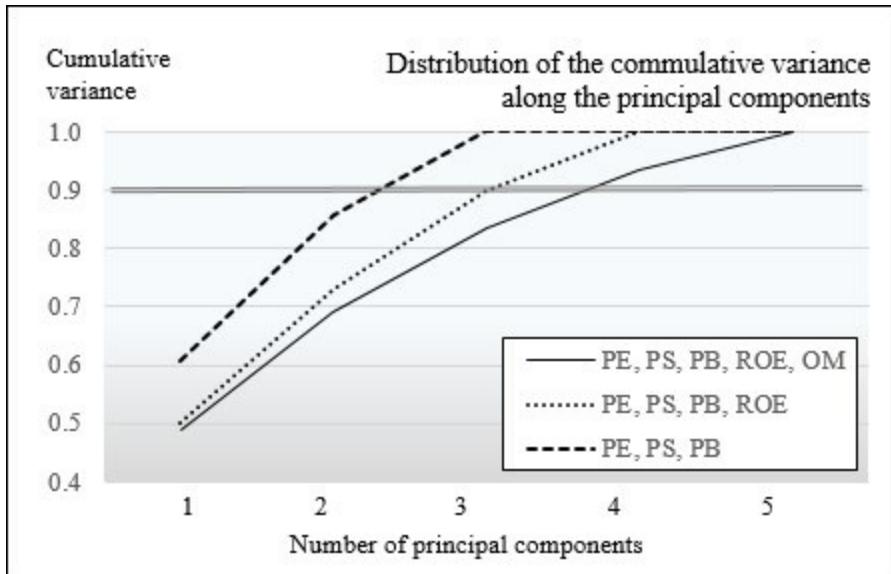
```
data.map( _._2.take(dim))
```

Let's consider the three different models:

- **Five features:** PE, PS, PB, ROE , and OM
- **Four features:** PE, PS, PB , and ROE

- **Three features:** PE , PS , and PB

The following plot displays the cumulative value of eigenvalues for each of the three models.



Distribution of eigenvalues in PCA for three, four and five features

The cumulative value of eigenvalues that are the variance of the transformed features X_i . If we apply a threshold of 90 percent to the cumulative variance, then the number of principal components for each test model is as follows:

- $\{PE, PS, PB\}$: 2
- $\{PE, PS, PB, ROE\}$: 3
- $\{PE, PS, PB, ROE, OM\}$: 3:

In conclusion, the PCA algorithm reduced the dimension of the model by 33 percent for the three-feature model, 25 percent for the four-feature model, and 40 percent for the five-feature model for a threshold of 90 percent.

Extending PCA

PCA is not without challenges and limitations. However, PCA has been extended to support categorical features and model cross-validation.

Validation

Like any other unsupervised learning technique, the resulting principal components should be validated through a one or K-fold cross-validation methodology using a regression estimator such as **Partial Least Square Regression (PLSR)** or **Predicted Residual Error Sum of Squares (PRESS)**. For those who are not afraid of statistics, I recommend *Fast Cross-validation in Robust PCA* by S. Engelen and M. Hubert [5:7]. You need to be aware, however, that the implementation of these regression estimators is not trivial. The validation of PCA is beyond the scope of this book.

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

Categorical features

PCA is a reliable technique for linear models with continuous features or features for which variance can be computed. This is not the case for categorical features. High-density categorical values can be converted into numerical values prior to PCA. However, most of the models with nominal categorical or binary values require an alternative approach to PCA such as the multiple correspondence analysis [5:8]. These techniques attempt to represent the original observations in a low-dimensional space and share a similar approach to manifold learning, briefly described in the next section.

Performance

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 second term reflects the computational complexity of the eigenvalue decomposition.

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

- Assuming that the variance is Gaussian
- Using a sparse matrix to compute eigenvalues for a problem with large features set and missing data
- Investigating alternatives to PCA to reduce the dimension of a model such as the **discrete Fourier transform (DFT)** or **singular value decomposition (SVD)** [4:17]
- Using the PCA in conjunction with EM (at the research stage)
- Deploying a dataset on a parallel data processing framework such as Apache Hadoop or Spark described in [Chapter 16, Scalability](#) and [Chapter 17, Apache Spark Framework](#)

Nonlinear models

The principal components analysis technique requires the model to be linear. Although the study of such algorithms is beyond the scope of the book, it is worth mentioning two approaches that extend PCA for nonlinear models:

- Kernel PCA
- Manifold learning

Kernel PCA

PCA extracts a set of orthogonal linear projections of an array of correlated values $X = \{x_i\}$. The kernel PCA algorithm consists of extracting a similar set of orthogonal projections of the inner product matrix XTX .

Non-linearity is supported by applying a kernel function to the inner product. Kernel functions are described in the *Kernel functions* section of [Chapter 12](#), *Kernel Models and Support Vector Machines*. The kernel PCA is an attempt to extract a low dimension features set (or manifold) from the original observation space. The linear PCA is the projection on the tangent space of the manifold.

Manifolds

The concept of manifolds is borrowed from differential geometry. **Manifolds** generalize the notions of curves in a two-dimensional space or surfaces in a three-dimensional space to higher dimensions. Nonlinear models are associated to Riemannian manifolds, which metric is the inner product XTX on a tangent space. The manifold represents a low dimension features space embedded into the original observation space. The idea is to project the principal components from the linear tangent space to a manifold using exponential map. This feat is accomplished through a variety of techniques, from **Local Linear Embedding**, density preserving map to **Laplacian Eigenmaps** [5:9].

The vector of observations cannot be directly used on a manifold because the metric such as norm or inner product depends on the location on the manifold the vector is applied to. Computation on manifolds relies on **tensors** such as **contravariant** and **covariant** vectors. Tensors algebra is supported by covariant and contravariant functors introduced in the *Abstraction* section of [Chapter 1, Getting Started](#).

Techniques that use differentiable manifolds are known as spectral dimensionality reduction.

Note

Alternative dimension reduction techniques

Here are some more alternative techniques, listed as reference: factor analysis, principal factor analysis, maximum likelihood factor analysis, independent component analysis (ICA), random projection, nonlinear ICA, **Kohonen's self-organizing maps**, neural networks, and multidimensional scaling, just to name a few [5:10].

Manifold learning algorithms are both classifiers and dimension reduction techniques that belong to the category of semi-supervised learning

techniques.

Summary

This completes the overview of some of the techniques to reduce the complexity and dimension of a problem.

We learned that the simplicity of divergences makes these techniques attractive for feature extraction and dimension reduction on a smaller set of features. Principal component analysis is a robust dimension reduction technique for linear or pseudo-linear models. Finally, manifold learning for nonlinear models is a technically challenging field with great potential in terms of dynamic object recognition [5:11].

In the next chapter, we will address supervised learning techniques, starting with generative models.

Chapter 6. Naïve Bayes Classifiers

So far, we have dealt with processing, filtering of data, and discovery of features through unsupervised learning. Although these techniques are critical to understand the problems, trends, and outliers, they do not provide data scientists with the ability to train a model with known, expected outcome, or labelled observations. These techniques are collectively known as supervised learning as described in the *Taxonomy of machine learning algorithms* section of [Chapter 1, Getting Started](#). Supervised learning is further categorized as generative and discriminative techniques.

This chapter describes 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, y .

In this chapter, you will 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 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

Naïve Bayes qualifies as a very simple probabilistic graphical model, which is commonly visualized as a directed graph for which a vertex is a prior or posterior probability and the edge is a conditional probability.

Given two events or observations X, 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 $p(Y|X) = p(X, Y)/p(X)$.

It is obvious that conditional or joint probabilities involving a large number of variables (that is, $p(X, Y, U, V, W | A, B)$), can be difficult to interpret. As a picture worth a thousand words, researchers introduced **graphical models** to describe probabilistic relation between random variables using graphs [5:1].

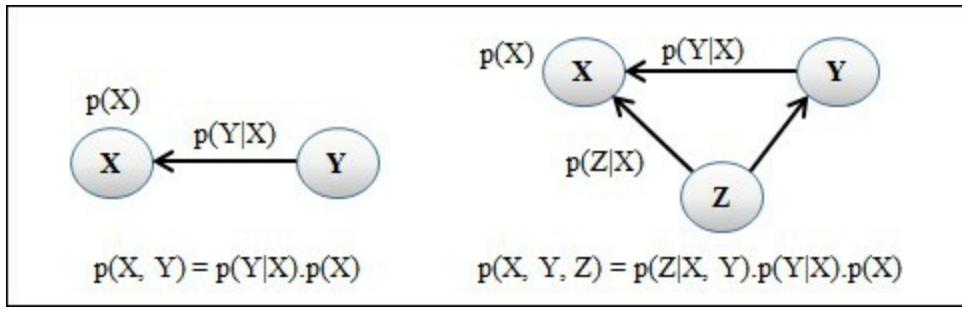
There are two categories of graphs and therefore graphical models:

- 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 between variables
- Represent statistical inference in terms of connectivity between graphical objects

Bayesian networks are graphical models visualizing a joint probability over a set of variables [5:2]. For instance, the two joint probabilities $p(X, Y)$ and $p(X, Y, Z)$ can be graphically modelled 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.

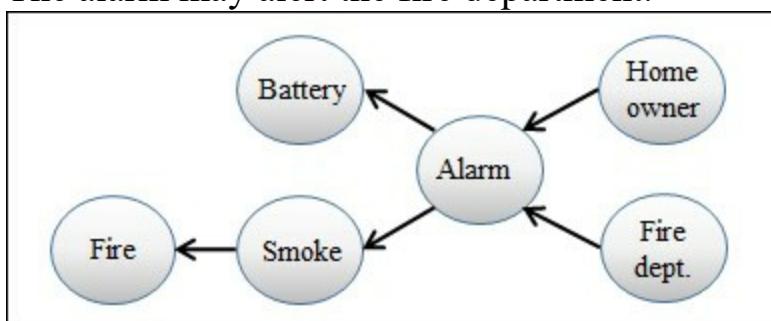
Tip

Metaphor for graphical models

From a software engineering perspective, graphical models visualize probabilistic equations the same way UML class diagrams visualize object-oriented source code.

Here is an example of a real-world Bayesian network—the functioning of a smoke detector, wherein:

1. A fire may generate smoke.
2. The 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.



Bayesian network for smoke detectors

This representation may be a bit counter-intuitive as the vertices are directed from the symptoms (or output) to the cause (or input). Directed graphical models are used in many different models beside Bayesian networks [5:3].

Note

Plate models

There are several alternative representations of probabilistic models beside 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' theorem under the assumption of features independence as mentioned in the *Supervised learning generative models* section of [Chapter 1, Getting Started](#).

Naïve Bayes classifiers

The Naïve Bayes classifier has a strict requirement: the features must be independent (that is, conditional dependence between features is null). It also restricts its applicability. The Naïve Bayes classification is better understood through simple, concrete examples [5:5].

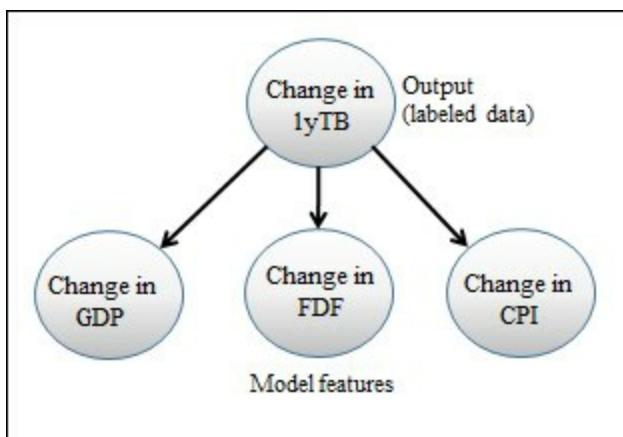
Introducing the multinomial Naïve Bayes

We illustrate the Naïve Bayes classification in the context of predicting the fluctuation of the interest rate of treasury bills.

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 select the **consumer price index (CPI)**, change in the **federal fund rate (FDR)**, and the **growth domestic product (GDP)** as a first set of features. The terminology is described in the *Terminology* section under *Finances 101* in the *Appendix*.

The use case is to predict the direction of the change of the yield of the **1-year Treasury bill (1yTB)**, considering the change in the current CPI, FDR, 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 that 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 1yTB increases, then we can conclude that there is a strong correlation of causal relationships between the features and the output variation in interest rates:



Naive Bayes model for predicting the change in the yield of 1yTB

The correlation (or cause-effect relationship) is to be 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 expected outcome, as illustrated in the following table:

ID	GDP	FDR	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 occurrences of each change {UP, DOWN} for the three features and the output value (direction of change in the yield of the 1yTB):

Number	GDP	FDR	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 1yTB increases (159 occurrences):

Number	GDP	Fed Funds	CPI
UP	110	136	127
DOWN	49	23	32
Total	159	159	159
UP/Total	0.69	0.85	0.80

From this last table, we conclude that the yield of the 1yTB 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:** This is the probability of a class
- **Likelihood:** This is the probability of a class given an observation also known as the probability of the predictor given a class
- **Evidence:** This is the probability of observations occurring also known as the prior probability of the predictor
- **Posterior probability:** This is the probability of an observation x being in a given class

No model can be simpler! The log likelihood, $\log p(x_i | C_j)$ is commonly used instead of the likelihood in order to reduce the impact features x_i with low likelihood.

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

Note

Naïve Bayes classification

M1: The posterior probability $p(C_j | x)$ is defined as follows:

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

Now, we will explain the terms used in this formula:

$x = \{x^i\} (0, n-1)$: This is a set of n features

$\{C_j\}$: This is a set of classes with their class prior $p(C_j)$

$p(x | C^j)$: This is the likelihood for each feature

M2: The computation of the posterior probability $p(C_j | x)$ is simplified with the assumption of conditional independence of features, which is defined as:

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

Here, x_i is independent and the probabilities are normalized for evidence $p(x) = 1$.

M3: **Maximum likelihood estimate (MLE)** is defined as:

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

M4: Naïve Bayes classification of an observation x into a class C_m is defined as:

$$C_m = \arg \max_j \mathcal{L}(C_j | x)$$

This use case has a major drawback: the GDP statistics are provided quarterly, while the CPI data is made available once a month and change in FDR is rather infrequent.

The frequentist perspective

The ability to compute the posteriori 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 1yTB given changes in the GDP, FDR, and CPI.

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

$$P(GDP=UP|1yTB=UP) = 110/159$$

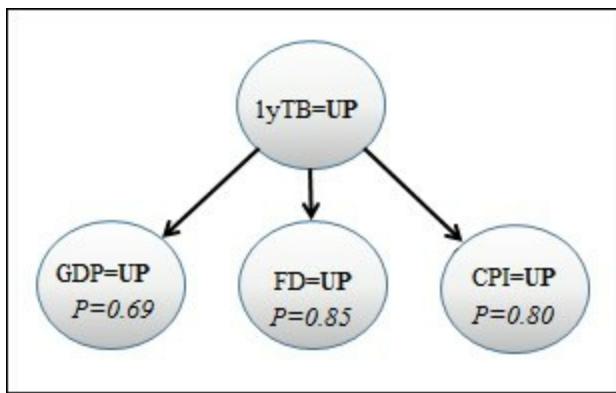
$$P(1yTB=UP) = \text{num occurrences } (1yTB=UP)/\text{total num of occurrences} = 159/256$$

$$p(1yTB=UP|GDP=UP, FDF=UP, CPI=UP) = p(GDP=UP|1yTB=UP) \times$$

$$p(FDF=UP|1yTB=UP) \times$$

$$p(CPI=UP|1yTB=UP) \times$$

$$p(1yTB=UP) = 0.69 \times 0.85 \times 0.80 \times 0.625$$



Bayesian network for the prediction of the change of the yield of the 1yTB

Tip

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.

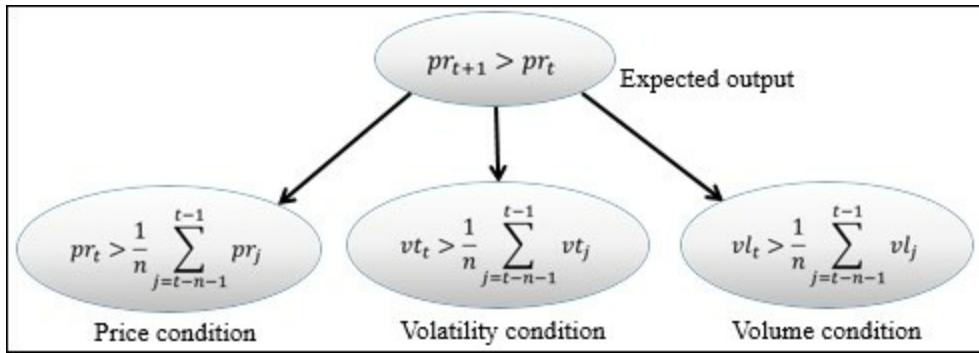
The predictive model

The predictive model is the second use case that consists of predicting the direction of the change of the closing price of a stock price, $pr_{t+1} = \{UP, DOWN\}$ on trading day $t+1$ given the history of its direction of the price, volume, and volatility for the previous t days pri for $i = 0$ to t . The features volume and volatility have been already used in the *Writing an application* section of [Chapter 1, Getting Started](#).

Therefore, the three features under consideration are:

1. 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, $[t-n, t]$.
2. The volume of the last trading day, vl_t , is above or below the average volume of the n previous trading days.
3. 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):



Bayesian model for predicting future direction of stock prices

This model works under the assumption that there is at least one observation or ideally few observations for each feature and for each expected value.

The zero-frequency problem

It is possible that the training set does not contain any observation for a feature associated to a specific label or class. In this case, the mean is computed as $0/N = 0$, 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 with a low number of occurrences that address this issue such as the **Laplace** and **Lidstone** smoothing formulas.

Note

Smoothing factor for counters

M5: Laplace smoothing of the mean k/N out of N observations of features of dimension n is defined as:

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

M6: Lidstone smoothing with a factor α is defined as:

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

These 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

Now it's time to write some Scala code and toy around with Naïve Bayes. Let's start with an overview of the software components.

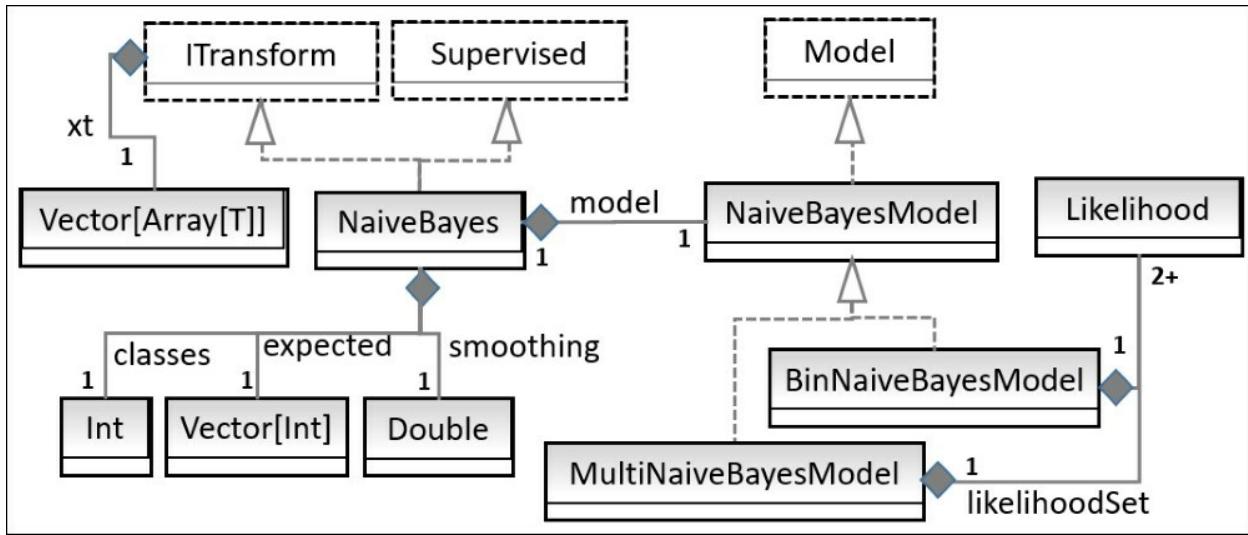
Design

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

- A generic model `NaiveBayesModel` of type `Model` that 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 pair of positive and negative class `Likelihood` instances.
- A model for the multinomial classification `MultiNaiveBayesModel`.
- The classifier class, `NaiveBayes`, has four parameters: a smoothing function such as `Laplace`, a set of observations of type `Vector[Array[Double]]`, a set of labels of type `Vector[Int]`, and the number of classes.

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

The key software components of the Naïve Bayes classifier are described in the following UML class diagram:



The UML diagram omits the helper traits or classes such as `Monitor` or Apache Commons Math components.

Training

The objective of the training phase is to build a model consisting of the likelihood for each feature and the class prior. The likelihood for a feature is identified as:

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

It is assumed for the sake of this test case that the features, technical analysis indicators, price, volume, and volatility are conditionally independent. This assumption is not totally correct.

Note

Conditional dependency

Recent models, known as **Hidden Naïve Bayes (HNB)**, relax the restriction on the independence between features. The HNB algorithm uses conditional

mutual information to describe the interdependency between some of the features [5:8].

Let's write the code to train the binomial and multinomial Naïve Bayes.

Class likelihood

The first step is to define the class likelihood for each feature using historical data.

The `Likelihood` class has the following attributes (line 1):

- The label used for the observation is named `label`
- An array of tuple Laplace or Lidstone smoothed mean and standard deviation, `muSigma`
- The prior probability of a class is named `prior`

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:

```
type DblPair = (Double, Double)
class Likelihood[@specialized(Double) T: ToDouble] (
    val label: Int,
    val muSigma: Vector[DblPair],
    val prior: Double) { //1

    def score(obs: Array[T]): Double = //2
        (obs, muSigma).zipped
            .map{ case(x, (mu,sig)) => (x, mu, sig) }
            ./:(0.0){ case (prob, (x, mean, stDev)) => {
                val z = implicitly[ToDouble[T]].apply(x)
                val likelihood = Stats.logGauss(mu, sig, z) //3
                prob + log(
                    if(likelihood <MINLOGARG) MINLOGVALUE else likelihood
                ) //4
            } } + Math.log(prior)
}
```

The parameterized class `Likelihood` has the following two purposes:

- It defines the statistics regarding a class C_k : its label, its mean and standard deviation, and the prior probability $p(C_k)$.
- It computes the `score` of a new observation for its run-time classification (line 2). The computation of the log of the likelihood uses a `logGauss` method of object `Stats` (line 3). As we will see in the next section, the log density can be either a log Gaussian or a Bernoulli distribution. The `score` method uses the Scala `zipped` method to merge the observation values with the labeled values and implement the formula M3, (line 4).

The **Gaussian mixture** is particularly suited for modeling datasets for which the features have large sets of discrete values or are continuous variables. The conditional probabilities for the feature x is described by the normal probability density function [5:9].

Note

Log likelihood using the Gaussian density

M7: For a Lidstone or Laplace smoothed mean μ' and a standard deviation σ the log likelihood of a posterior probability for a Gaussian distribution is defined as follows:

$$\mathcal{L}(C_j | x) = \sum_{i=0}^{n-1} \left[-\frac{1}{2} \log(2\pi) - \log \sigma - \frac{(x_i - \mu')^2}{2\sigma^2} + \log p(C_j) \right]$$

The log of the Gauss, `logGauss`, and the log of the Normal distribution, `logNormal`, are defined in the `Stats` singleton, introduced in the *Profiling data* section in [Chapter 2, Data Pipelines](#):

```
def logGauss(mean: Double, stdDev: Double, x: Double):Double = {
  val y = (x - mean)/stdDev
  -LOG_2PI - Math.log(stdDev) - 0.5*y*y
}
val logNormal = logGauss(0.0, 1.0, _: Double)
```

The Lognormal computation is implemented as a partially applied function.

Binomial model

The next step is to define the `BinNaiveBayesModel` model for a two-class classification scheme. The two-class model consists of two `Likelihood` instances: positives for the label UP (value 1) and negatives for the label DOWN (value 0).

In order to make the model generic, we created a `NaiveBayesModel` trait that can be extended as needed to support both the binomial and multinomial Naïve Bayes:

```
trait NaiveBayesModel[T] {  
    def classify(x: Array[T]): Int //5  
}
```

The `classify` method uses the trained model to classify a multivariate observation `x` of type `Array[T]` given a probability density function, `density` (line 5). This method returns the class the observation belongs to.

Let's start with the definition of the `BinNaiveBayesModel` class that implements the binomial Naïve Bayes:

```
class BinNaiveBayesModel[T: ToDouble[T]] (  
    pos: Likelihood[T],  
    neg: Likelihood[T]) extends NaiveBayesModel[T] { //6  
    override def classify(x: Array[T]): Int = //7  
        if(pos.score(x) > neg.score(x)) 1 else 0  
    ...  
}
```

The constructor for the `BinNaiveBayesModel` class takes two arguments:

- `pos`: This is the class with the likelihood for observations with a positive outcome
- `neg`: This is the class with the likelihood for observations with a negative outcome (line 6)

The `classify` method called by the `|>` operator in the Naïve Bayes classifier.

It returns `1` if the observation `x` matches the class `Likelihood` for positive cases, and `0` otherwise (line 7).

Tip

Model validation

The parameters of the Naïve Bayes model (`likelihood`) are computed through training and the value `model` is instantiated regardless 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 F_1 measure as described in the *Design template for classifiers* section in the *Appendix*.

Multinomial model

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

```
class MultiNaiveBayesModel[@specialized(Double) T: ToDoubleI](//8
  likelihoodSet: Seq[Likelihood[T]]) extends NaiveBayesModel[T] {

  override def classify(x: Array[T]): Int = {
    val <<< = (p1: Likelihood[T], p2: Likelihood[T]) =>
      p1.score(x) > p2.score(x) //9
    likelihoodSet.sortWith(<<<).head.label //10
  }
  ...
}
```

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

- Its constructor requires a sequence of class likelihood, `likelihoodSet` (line 8).
- The run-time classification method, `classify`, sorts the class likelihoods by their score (posterior probability) using the `<<<` function (line 9). The method returns the ID of the class with the highest log likelihood (line 10).

Classifier components

The Naïve Bayes algorithm is implemented as a data transformation using a model implicitly extracted from a training set of type `ITransform` described in the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#).

The attributes of the multinomial Naïve Bayes are:

- The smoothing formula (Laplace, Lidstone, and so on): `smoothing`
- The set of multivariable observations defined as `xt`
- The expected values (or labels) associated with the set of observations `expected`
- The log of the probability density function: `logDensity`
- The binomial Naive Bayes (type `BinNaiveBayesModel`) has 2 classes, the multinomial Naive Bayes (type `MultiNaiveBayesModel`) has 2 or more classes (line 11)

```
class NaiveBayes[@specialized(Double) T: ToDouble] (
    smoothing: Double,
    xt: Vector[Array[T]],
    expected: Vector[Int],
    classes: Int) //11
extends ITransform[Array[T], Int] //12
with Supervised[Array[T], Int] with Monitor[Double] {

    val model: Option[NaiveBayesModel[T]] //13
    def train(expected: Int): Likelihood[T]
    ...
}
```

The `Monitor` trait defines miscellaneous logging and display functions.

Data transformations of type `ITransform` require an input of type `Array[T]` and an output of type `Int t` (line 12). The output of the Naïve Bayes, `Int`, represents the index of the class an observation belongs to. The type of the model, `model`, can be either `BinNaiveBayesModel` for two classes or `MultiNaiveBayesModel` for a multinomial model (line 13):

```
val model: Option[NaiveBayesModel[T]] = classes match {
    case 0 | 1 => None
    case 2 => BinNaiveBayesModel[T](train(1), train(0))
```

```

    case 3 =>
      MultiNaiveBayesModel[T] (List.tabulate(classes) ( train(_)))
  }

```

Tip

Training and class instantiation

There are several benefits to 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 reduces the number of states of the model (such as Untrained, Partially trained, Trained, and Validated). It is an elegant way to hide the details of the training of the model from the user.

The `train` method is applied to each class. It takes the index or label of the class and generates its log likelihood data (line 14):

```

def train(index: Int): Likelihood[T] = {    //14
  val xv = xt.map(_.map(implicitly[ToDouble[T]].apply(_)))
  val values = xv.zip(expected) //15
    .filter(_._2 == index).map(_.._1) //16
  if(values.isEmpty) throw new IllegalStateException(...)

  val dim = dimension(xt)
  val meanStd = statistics(values).map(stat =>
    (stat.lidstoneMean(smoothing, dim), stat.stdDev)) //17
  Likelihood(index, meanStd, values.size.toDouble/xv.size) //18
}

```

The training set is generated by zipping the vector of observations `xt` with `expected` (line 15). The method filters out the observation for which the label does not correspond to this class (line 16). The pair (mean and standard deviation) `meanStd` is computed using the Lidstone smoothing factor (line 17). Finally, the training method returns the class likelihood corresponding to the index label (line 18).

The `NaiveBayes` class also defined the run-time classification method, `|>`, and the F_1 validation methods. Both the methods are described in the next section.

Note

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 attribute for these observations is not computed. This workaround is obviously made possible because of the conditional independence between features.

The constructor `apply` for the `NaiveBayes` returns the type `NaiveBayes`:

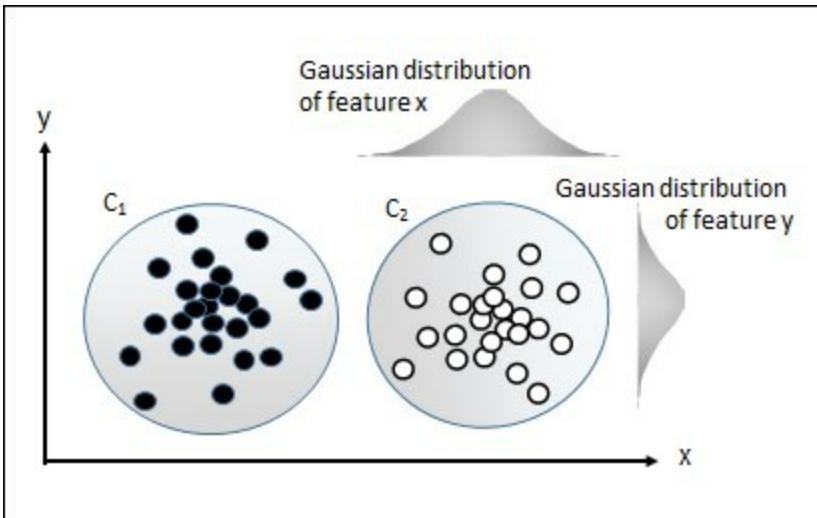
```
object NaiveBayes {
  def apply[@specialized(Double) T: ToDouble] (
    smoothing: Double,
    xt: Vector[Array[T]],
    expected: Vector[Int],
    classes: Int): NaiveBayes[T] =
    new NaiveBayes[T](smoothing, xt, y, classes)
  ...
}
```

Classification

Let's dive into the classification and validation of our model. The execution of the training computes the likelihood and class priors, which are used to validate the model and classify new observations.

The score represents the log of likelihood estimate (or the posterior probability), which is computed as the summation of the log of the Gaussian distribution using the mean and standard deviation extracted from the training phase and the log of the class likelihood.

The Naïve Bayes classification using Gaussian distribution is illustrated using two classes C_1 and C_2 , and a model with two features (x,y) :



The Gaussian Naïve Bayes using a 2-dimension model

The `|>` method returns the partial function that implements the run-time classification of new observation x using one of the two Naïve Bayes models. The `model` and the `logDensity` function are used to assign the observation x to the appropriate class (line 19):

```
override def |> : PartialFunction[Array[T], Try[Int]] = {
  case x: Array[T] if(x.nonEmpty && model.isDefined) =>
    Try( model.map(_.classify(x)).getOrElse(-1)) //19
}
```

F1 Validation

Finally, the Naïve Bayes classifier is implemented by the `NaiveBayes` class. It implements the training and run-time classification using the Naïve Bayes formula. In order to force the developer to define a validation for any new supervised learning technique, the class inherits from the `Supervised` trait that declares the validation method:

```
trait Supervised[T, V] {
  self: ITransform[T, V] => //20
  def validate(xt: Vector[T], expected: Vector[V]): Try[Double]
} //21
```

The validation of a model applies only to a data transformation of type

`ITransform` (line 20).

The `validate` method takes the following arguments (line 21):

- A time series `xt` of multidimensional observations
- A vector of expected class values, `expected`

By default, the `validate` method returns the F_1 score for the model as described in the *Accessing a model* section in [Chapter 2, Data Pipelines](#).

Let's implement the key functionality of the `Supervised` trait for the Naïve Bayes classifier:

```
override def validate(
    xt: Vector[Array[T]],
    expected: Vector[Int]): Try[Double] = Try { //22
  val predict = model.get.classify(_:Array[Int]) //23
  MultiFValidation(expected, xt, classes).score //24
}
```

The predictive, partially applied function, `predict` is created by assigning a predicted class to a new set of observations `xt` (line 23), then the prediction, index of classes, is loaded into the `MultiFValidation` class to compute the F_1 score (line 24).

Features extraction

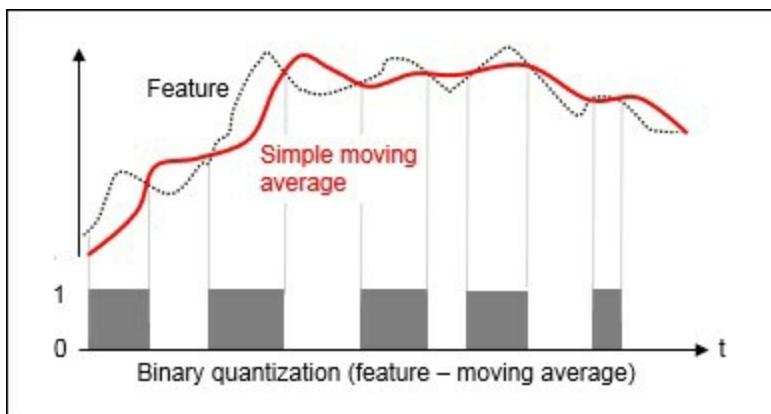
The most critical element in the training of a supervised learning algorithm is the creation of labeled data. Fortunately, in this case, the labels (or expected classes) can be automatically generated. The objective is to predict the direction of the price of a stock for the next trading day, taking into account the moving average price, volume, and volatility over the previous n days.

The extraction of features follows these six steps:

1. Extract historical trading data for each feature (that is, price, volume, and volatility).
2. Compute the simple moving average for each feature.

3. Compute the difference between value and moving average for each feature.
4. Normalize the difference by assigning 1 for positive values and 0 for negative values.
5. Generate a time series of the difference between the closing price of the stock and the closing price of the previous trading sessions.
6. Normalize the difference by assigning 1 for positive values and 0 for negative values.

The following diagram illustrates the feature extraction for steps 1 to 4:



Binary quantization of the difference in value – moving average

The first step is to extract the average price, volume, and volatility (that is, $I - low/high$) for each stock during the period of January 1, 2000 and December 31, 2014 with daily and weekly closing prices. Let's use the simple moving average to compute these averages for the window $[t-n, t]$.

The extractor defines the list of features to extract from the financial data source as described in the *Data extraction and Data sources* section under *Source consideration* in the *Appendix*.

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

The naming convention for the trading data and metrics is described in the *Trading data* section under *Technical analysis in Appendix*.

The training and validation of the binomial Naïve Bayes is implemented using a monadic composition as follows:

```

val trainRatio = 0.8 //25
val period = 4
val symbol ="IBM"
val path = "resources/data/bayes"
val pfnMv = SimpleMovingAverage[Double](period, false) |> //26

for {
    obs <- pfnSrc(extractor) //28
    (x, deltas) <- computeDeltas(obs) //29
    expected <- Try{difference(x.head.toVector, diffInt)}//30
    features <- Try {transpose(deltas)} //31
    labeled <- OneFoldValidation[Int](
        features.drop(1), expected, trainRatio //32
    )
    f1Score <- {
        val nb =NaiveBayes[Int](1.0, labeled.trainingSet) //33
        nb.validate(labeled.validationSet) //34
    }
}
yield {
    show(s"f1 score $f1Score")
}

```

The first decision to make is to distribute the observations between the training set and the validation set. The `trainRatio` value (line 25) defines the ratio of the original observations set to be included in the training set. The simple moving average values are generated by the partial function `pfnMv` (line 26). The extracting partial function, `pfnSrc` (line 27) is used to generate the three trading time series, price, volatility, and volume (line 28).

The next step consists of applying the simple moving average, `pfnMv`, to the multidimensional time series `obs` (line 29) using the `computeDeltas` method.

```

type Obs = Vector[Array[Double]],
type Labeled = (Obs, Vector[Array[Int]])

def computeDeltas(obs: Obs): Try[Labeled] = Try{
    val sm = obs.map(_.toVector).map( pfnMv(_)
        .getOrElse(Vector.empty[Double]).toArray) //35
    val x = obs.map(_.drop(period-1) )
    (x, x.zip(sm).map{ case(z,y) => z.zip(y).map(delta(_)) }) //36
}

```

```
}
```

The `computeDeltas` method computes the time series of observations, `sm`, smoothed with a simple moving average (line 35). The method generates a time series of `0` and `1` for each of the three features in the observations set `x` and smoothed dataset `sm` (line 36).

Next, the call to the differential computation `difference` generates the labels $\{0, 1\}$ representing the change in direction of the price of a security between two consecutive trading sessions: `0` if the price declined and `1` if the price increased (line 30) (refer to *Differential operator* section under *Time series* in [Chapter 3, Data Preprocessing](#)).

The features for the training of the Naïve Bayes model are extracted from these ratios by transposing the ratios-time series matrix in the `transpose` method of the singleton `TSeries` (line 31).

Next, the training set and validation set are extracted from the `features` set using the `OneFoldValidation` class introduced in the *One-fold crossover validation* section under *Crossover validation* in [Chapter 2, Data Pipelines](#) (line 32).

Tip

Selecting the training data

In our example, the training set is simplistically the first `trainRatio` multiplied by the size of dataset observations. Practical applications use a K-fold cross-validation technique to validate models as described in the *K-fold cross validation* section under *Assessing a model* in [Chapter 2, Hello World!](#). A simpler alternative is to create the training set by picking observations randomly and use the remaining data for validation.

The last two stages in the workflow consist of training the Naïve Bayes model by instantiating the `NaiveBayes` class (line 33) and computing the F_1 score for different values of the smoothing coefficient of the simple moving average applied to the stock price, volatility, and volume (line 34).

Tip

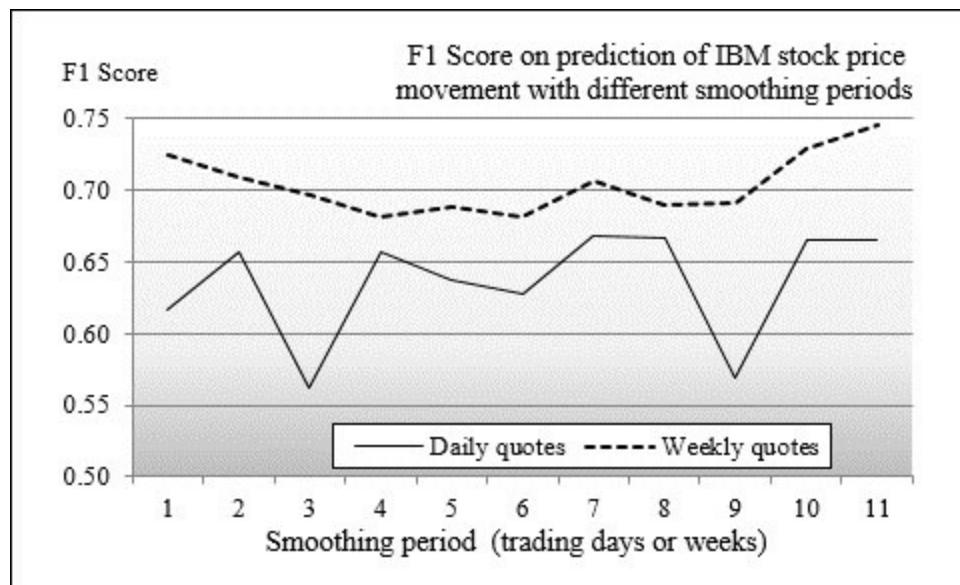
Implicit conversion

The `NaiveBayes` class operates on the element of type `Int` and `Double`; therefore, it assumes that there is a conversion between `Int` and `Double` (view bounded). The Scala compiler may generate a warning because the conversion from `Int` to `Double` has not been defined. Although Scala relies on its own conversion functions, I would recommend explicitly defining and controlling your conversion function:

```
implicit def inttoDouble(n: Int): Double = n.toDouble
```

Testing

The next chart plots the value of the F_1 measure of the predictor of the direction of the IBM stock using price, volume, and volatility over the previous n trading days with n varying from 1 to 12 trading days:



Graph of the F1 measure for the validation of the Naive Bayes model

The chart illustrates the impact of the value of the averaging period (number of trading days) on the quality of the multinomial Naïve Bayes prediction,

using the value of 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 F_1 score for the models using weekly (with respect to daily) closing prices varies between 0.68 and 0.74 (with respect to 0.56 and 0.66).
- The prediction using weekly closing prices is more accurate than the prediction using the daily closing prices. In this particular example, the distribution of the weekly closing prices is more reflective of an intermediate term trend than the distribution of daily prices.
- The prediction is somewhat independent of the period used to average the features.

Multivariate Bernoulli classification

So far, our investigation of the Naïve Bayes has focused on features that are essentially binary $\{UP=1, DOWN=0\}$. 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 in the case of very large labeled datasets. The example is the perfect candidate for the **Bernoulli** model.

Model

The Bernoulli model differs from the Naïve Bayes classifier in that it penalizes the features x , which does not have any observation; the Naïve Bayes classifier ignores them [5:10].

Note

The Bernoulli mixture model

M8: For a feature function f_k with $f_k = 1$ if the feature is observed, 0 otherwise, and the probability p of the observed feature x_k belongs to the class C_j , the posterior probability is computed as follows:

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

Implementation

The implementation of the Bernoulli model consists of modifying the `score` function in the `Likelihood` class using the Bernoulli density method, `bernoulli`, defined in the `Stats` object:

```
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, M8. The second version uses the signature of the *Density (Double*) => Double* type.

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

Naïve Bayes and text mining

The extraction of the most relevant features to build a model relies on discovery and data mining. For many applications, the data available to the scientist is unstructured text. The multinomial Naïve Bayes classifier is particularly suited for **text mining**.

The Naïve Bayes formula is quite effective to classify the following entities:

- E-mail spams
- Business news stories
- Movie reviews
- Technical papers per field of expertise

This third use case consists of predicting the direction of a stock given the financial news. There are two types of news that affects the stock of a particular company:

- **Macro trends:** This consists of the economic or social news such as conflicts, economic trends, or labor market statistics
- **Micro updates:** This consists of the financial or market news related to this specific company such as earnings, change in ownership, or press releases

Micro-economic news related to a specific company has the potential to affect the sentiment of investors toward the company and may lead to a sudden shift in the price of its stock. Another important feature to consider is the average time it takes for investors to react to the news and affect the price of the stock:

- Long-term investors may react within days or even weeks
- Short-term traders adjust their positions within hours, sometimes within the same trading session

The average time the market reacts to significant financial news of a company is illustrated in the following chart:

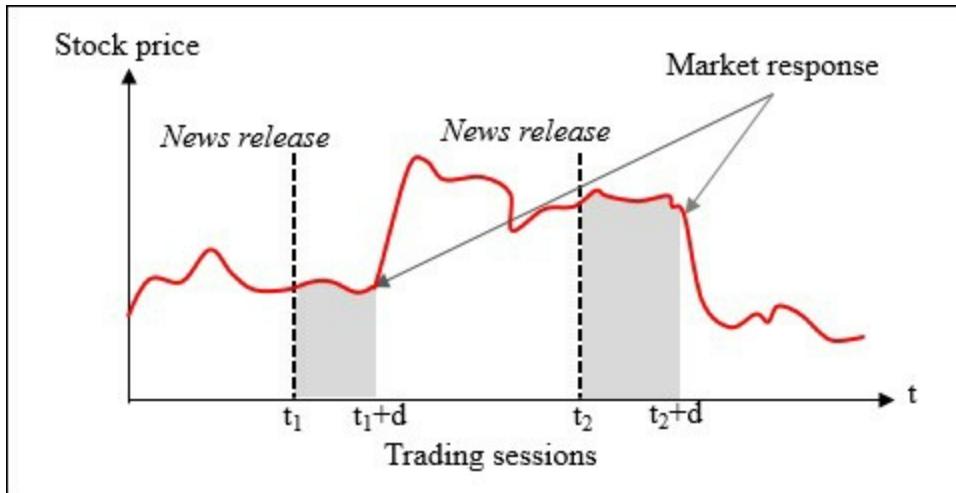
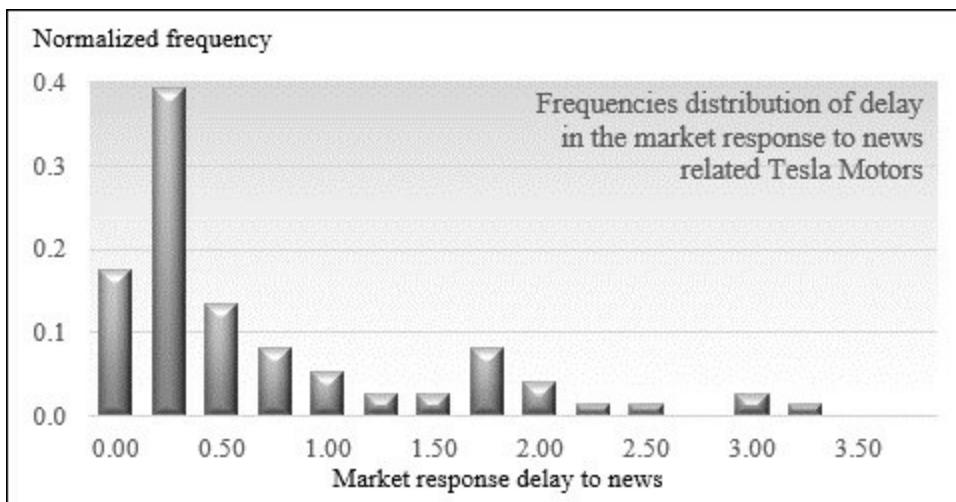


Illustration of the reaction of investors on the price of a stock following a news release

The delay in the market reaction is a relevant feature only if the variance of the response time is significant. The distribution of the frequencies of the delay in the market reaction to any newsworthy articles regarding TSLA is fairly constant. It shows that the stock price reacts within the same day in 82 percent of the cases, as shown in the following bar chart:



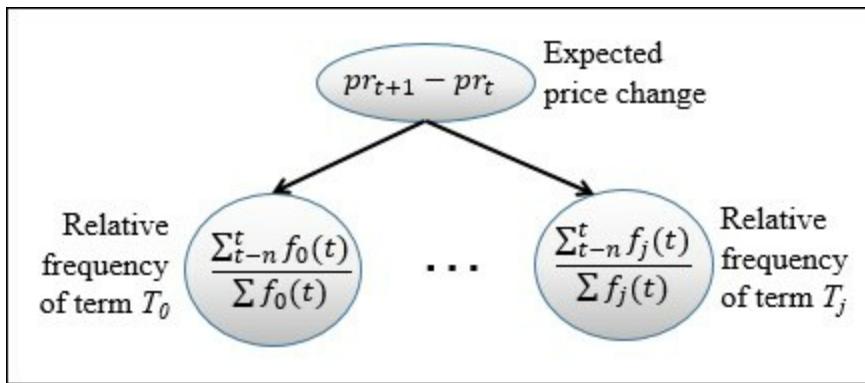
Distribution of the frequencies of the reaction of investors on the price of a stock following a news release

The frequency peak for a market response delay of 1.75 days can be explained by the fact that some news items are released over the weekend and

the investors have to wait for the following Monday to drive the stock price higher or lower. Another challenge is to assign any shift of stock price to a specific news release, taking into account that some news can be redundant, confusing, or simultaneous.

Therefore, the model features for predicting the stock price pr_{t+1} are the relative frequency f_i of occurrence of a term T_i within a time window $[t-n, t]$, where t and n are trading days.

The following graphical model formally describes the causal relation or conditional dependency of the relative change of the stock price between two consecutive trading sessions t and $t+1$, given the relative frequency of appearance of some key terms in the media:



Bayesian model for the prediction of stock movement given the 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 information retrieval

The previous exercise touches the broader problem of extracting information through unstructured text. A full discussion of information retrieval and text mining is beyond the scope of this book [5:11]. For the sake of simplicity, our model relies on a very simple scheme for extracting relevant terms and computing their relative frequency. The following 10-step sequence of action describes one of the numerous methodologies to extract the most relevant terms from a corpus:

1. Create or extract the time stamp 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 **Levenstein** 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 (that is, *NN* and *NNP*).
9. Match the terms against a dictionary that supports senses, hyponyms, and synonyms such as **WordNet**.
10. **Disambiguate** word sense using Wikipedia's repository **DBpedia** [5:12].

Note

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, which requires additional steps such as search, crawling, and scraping [5:13].

Implementation

Let's apply the text mining methodology template to predict the direction of a stock given the financial news. The algorithm relies on a sequence of seven simple steps:

1. Searching and loading the news articles related to a given company and its stock as a document D_t of type Document.
2. Extracting the time stamp, $\text{date} : \tau$ the article using a regular expression.
3. Ordering the documents D_t as per the time stamp.
4. Extracting the terms $\{T_i, D\}$ from the content of each document D_t .
5. Aggregating the terms $\{T_p, D\}$ for the documents D_t sharing the same publication date t .
6. Computing the relative frequency rtf of each term $\{T_i, D\}$ for the date t , as the ratio of number of its occurrences in all the articles released at t over the total number of its occurrences of the term in the entire corpus.
7. Normalizing the relative frequency for the average number of articles per date $nrtf$.

Note

Text analysis metrics

M9: Relative frequency of occurrences for term (or keyword) t_i with n_i^a occurrences in article a is defined as:

$$rtf\{t_i\} = \frac{\sum_{a \in D_t} n_i^a}{\sum_{a \in Corpus} n_i^a}$$

M10: Relative frequency of occurrences of a term t_i normalized by daily average of the number of articles for which N_a is the total number of articles

and N_d is the number of days in the survey is defined as:

$$nrt\{t_i\} = \frac{rtf\{t_i\} N_d}{N_a}$$

The news articles are minimalist documents with a time stamp, a title, and a content as implemented by the `Document` class:

```
case class Document[T <: AnyVal] ( //1
  date: T,
  title: String,
  content: String) (implicit f: T => Long)
```

The time stamp, `date`, has a type bounded to the type `Long`, so it can be converted to current time in milliseconds of the JVM (line 1).

Analyzing documents

This section is dedicated to the implementation of the simple text analyzer. Its purpose is to convert a set of documents of type `Document`, in our case, news articles, into a distribution of relative frequencies of keywords.

The `TextAnalyzer` class implements a data transformation of type `ETransform` described in the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#). It transforms a sequence of documents into a sequence of relative term frequency distributions.

The `TextAnalyzer` class has two arguments (line 4):

- A simple text parser, `parser`, that extracts an array of keywords from the title and content of each news article (line 2).
- A `lexicon` that lists keywords used in monitoring news related to a company and their synonyms. The synonyms or terms that are semantically similar to each keyword are defined in an immutable map

The functionality of the `TextAnalyzer` class is implemented as follows:

```

type TermsRF = Map[String, Double]
type TextParser = String => Array[String] //2
type Lexicon = Map[String, String] //3
type Corpus[T] = Seq[Document[T]]

class TextAnalyzer[T <: AnyVal]( //4
    parser: TextParser,
    lexicon: Lexicon)
(implicit ordering: Ordering, f: T => Long)
extends ETransform[Corpus[T], Seq[TermsRF]](lexicon) {

    type U = Corpus[T] //5
    type V = Seq[TermsRF] //6

    override def |> : PartialFunction[U, Try[V]] = {
        case docs: U => Try(score(docs))
    }

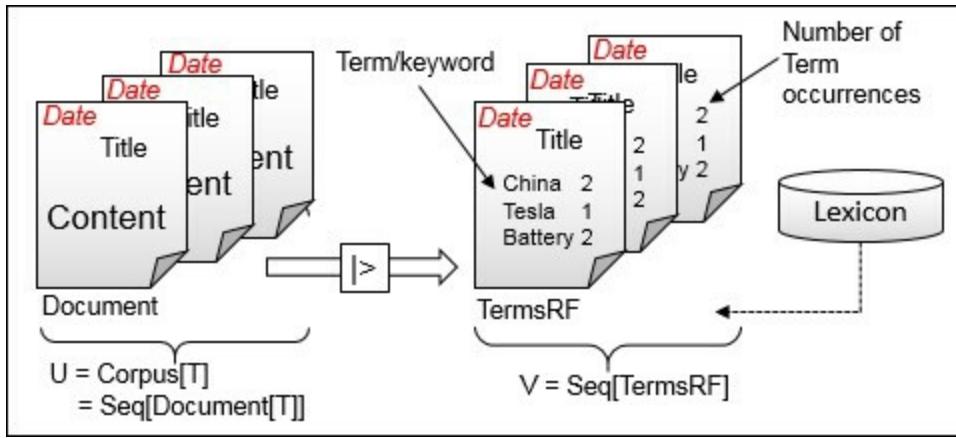
    def score(corpus: Corpus[T]): Seq[TermsRF] //7
    def quantize(termsRFSeq: Seq[TermsRF]): //8
        Try[(Array[String], Vector[Array[Double]])]
    def count(term: String): Counter[String] //9
}

```

The type **U** of input into the data transformation `|>` is the corpus or sequence of news articles (line 5). The type **V** of output from the data transformation is the sequence of relative term frequency distribution of type `TermsRF` (line 6).

The private method `score` does the heavy lifting for the class (line 7). The `quantize` method creates a homogenous set of observed features (line 8) and the `count` method counts the number of occurrences of terms or keywords across the documents or news articles sharing the same publication date (line 9).

The following diagram describes the different components of the text mining process:



Extracting relative terms frequency

Let's dive into the `score` method:

```
def score(corpus: Corpus[T]): Seq[TermCount] = {
    val termsCount = corpus.map(doc => //10
        (doc.date, count(doc.content)))
    )
    val termsCountMap = termsCount.groupBy(_._1).map{
        case (t, seq) => (t, seq.aggregate(new Counter[String]))
            ((s, cnt) => s ++ cnt._2, _ ++ _)) //11
    }
    val termsCountPerDate = termsCountMap.toSeq
        .sortWith(_._1 < _._1).unzip._2 //12
    val allTermsCounts = termsCountPerDate
        .aggregate(new Counter[String])((s, cnt) =>
            s ++ cnt, _ ++ _)) //13

    termsCountPerDate.map(_ /allTermsCounts).map(_.toMap) //14
}
```

The first step in the execution of the `score` method is the computation of the number of occurrences of keywords of the lexicon on each of the documents/news articles (line 10). The computation of the number of occurrences is implemented by the `count` method:

```
def count(term: String): Counter[String] =
    parser(term)./:(new Counter[String])( (cnt, w) => //16
        if(lexicon.contains(w)) cnt + lexicon(w) else cnt
    )
```

The method relies on the term counting class `Counter`, which subclasses `mutable.Map[String, Int]` as described in the *Terms counter* section under *Scala programming in Appendix*. It uses a fold to update the count for each of the terms associated to a keyword (line 16). The terms count for the entire corpus is computed by aggregating the terms count for all documents (line 11).

The next step consists of aggregating the keywords count across the document for each time stamp. A map, `termsCountMap`, with date as key and keywords counter as values is generated by invoking the higher order method `groupBy` (line 11). Next the method `score` extracts a sorted sequence of keywords counts, `termsCountPerDate` (line 12). The total counts for each keyword over the entire corpus, `allTermsCounts` (line 13) is used to compute the relative or normalized keywords frequencies: formulas M9 and M10 (line 14).

Generating the features

There is no guarantee that all the news articles associated to a specific publication date all the keywords used in the model. The `quantize` method assigns relative frequencies of *0.0* for keywords that are missing from the news articles, as illustrated in the following table:

	Keyword 1	Keyword 2	Keyword 3	...	Keyword N
Date 1	0.42	0.00	0.07		0.23
Date 2	0.00	0.11	0.18		0.04
...					
Date J	0.13	0.29	0.00		0.00

Table on relative frequencies of keywords per publishing date

The `quantize` method transforms a sequence of term-relative frequencies into a pair of keyword observations:

```
def quantize(termsRFSeq: Seq[TermsRF]): Try[(Array[String], Vector[Array[Double]])] = Try {
```

```

val keywords = lexicon.values.toArray.distinct //15
val relFrequencies = termsRFSeq.map( tf => //16
    keywords.map(key =>
        if(tf.contains(key)) tf.get(key).get else 0.0
    )
)
(keywords, relFrequencies.toVector) //17
}

```

The `quantize` method extracts an array of `keywords` from the `lexicon` (line 15). The vector of features `relFrequencies` is generated by assigning the relative keyword frequency *0.0* for keywords that are not detected across the news articles published at a specific date (line 16). Finally, the method returns the key-value pair, (`keywords`, relative keyword frequency) (line 17).

Tip

Sparse relative frequencies vector

Text analysis and natural language processing deals with very large feature sets with potentially hundreds of thousands of features or keywords. Such computation would be almost intractable if it was not for the fact that the vast majority of keywords are not present in each document. It is a common practice to use sparse vectors and sparse matrices to reduce memory consumption during training.

Testing

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

Retrieving textual information

Let's start implementing and defining the two components of `TextAnalyzer`; the parsing function and the lexicon:

```
val LEXICON = loadLexicon //18

def parse(content: String): Array[String] = {
    val regExpr = "[',.|.|?|!|:|\\"]"
    content.trim.toLowerCase.replace(regExpr, " ") //19
        .split(" ") //20
        .filter( _.length > 2) //21
}
```

The lexicon is loaded from a file (line 18). The `parse` method uses a simple regular expression, `regExpr`, to replace any punctuation into a Space character (line 19) used as a word delimiter (line 20). All words shorter than three characters are discounted (line 21).

Let's describe the workflow to load, parse, and analyze news articles related to the company Tesla, Inc. and its stock, ticker symbol `TSLA`.

The first step is to load and clean all the articles (`corpus`) defined in the directory `pathCorpus` (line 22). This task is performed by the `DocumentsSource` class described in the *Documents extraction* section under *Scala Programming in Appendix*.

```
val pathCorpus = "resources/text/bayes/" //22
val dateFormat = new SimpleDateFormat("MM.dd.yyyy")
val pfnDocs = DocumentsSource(dateFormat, pathCorpus) |> //23

val textAnalyzer = TextAnalyzer[Long](parse, LEXICON)
val pfnText = textAnalyzer |> //24
```

```

for {
    corpus <- pfndocs(None) //25
    termsFreq <- pfntext(corpus) //26
    featuresSet <- textAnalyzer.quantize(termsFreq) //27
    expected <- Try(difference(TSLA_QUOTES, diffInt)) //28
    nb <- NaiveBayes[Double](1.0, featuresSet._2.zip(expected)) //29
} yield {
    show(s"Naive Bayes model${nb.toString(quantized._1)}")
    ...
}

```

A document source is fully defined by the path of the data input files and the format used in the time stamp (line 23). The text analyzer and its explicit data transformation, `pfntext`, is instantiated (line 24). The text processing pipeline is defined using the following steps:

1. Transformation of input source file into a `corpus` (sequence of news articles) using the partial function, `pfndocs` (line 25).
2. Transformation of `corpus` into a sequence of relative keyword frequency vectors, `termsFreq`, using the partial function, `pfntext` (line 26).
3. Transformation of sequence of relative keywords frequency vector into `featuresSet` using `quantize` (line 27); (refer to *Differential operator* section under *Time series* in [Chapter 3, Data Pre-processing](#)).
4. Creation of the binomial `NaiveBayes` model using the pair (`featuresSet._2`, `expected`) as training data (line 29).

The expected class values $\{0, 1\}$ are extracted from the daily stock price for Tesla Motors, `TSLA_QUOTES`:

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

Note

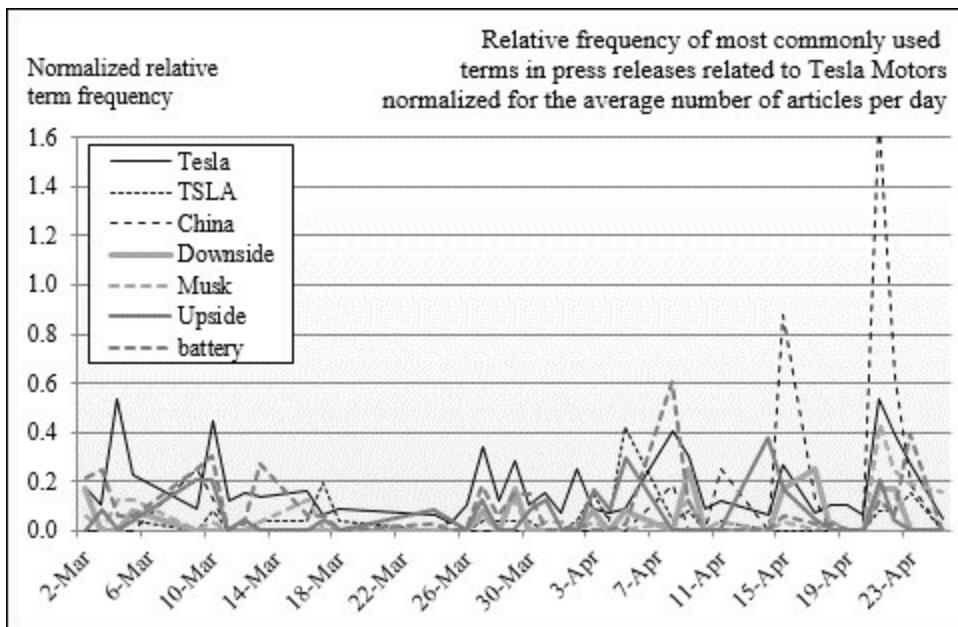
Semantic analysis

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 **Dirichlet Latent Allocation**.

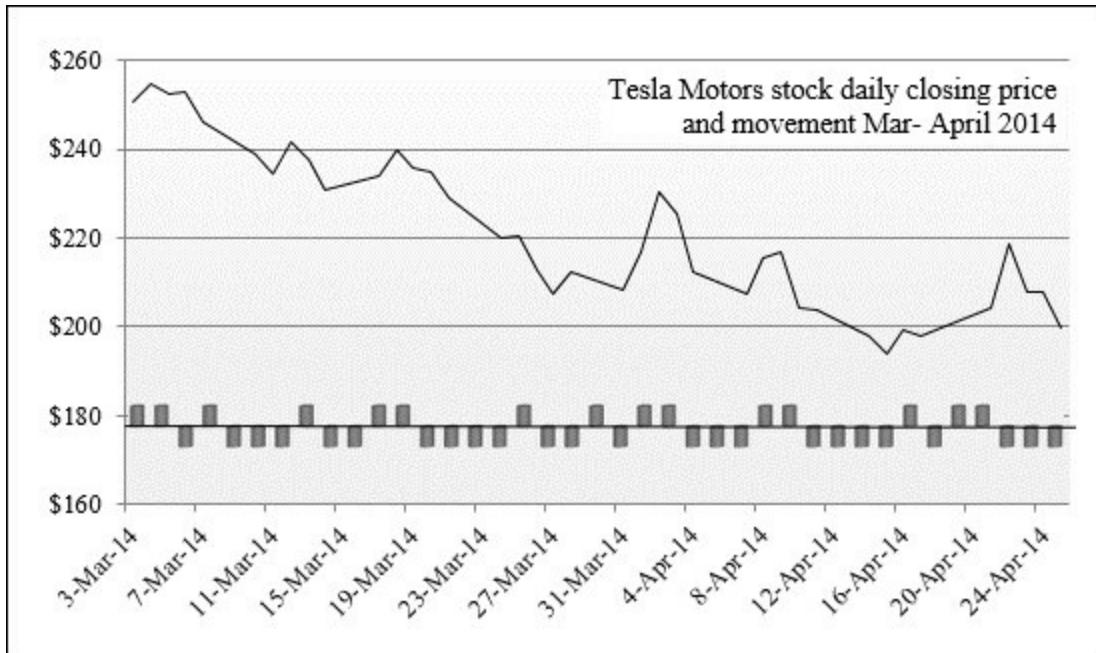
Evaluating text mining classifier

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



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

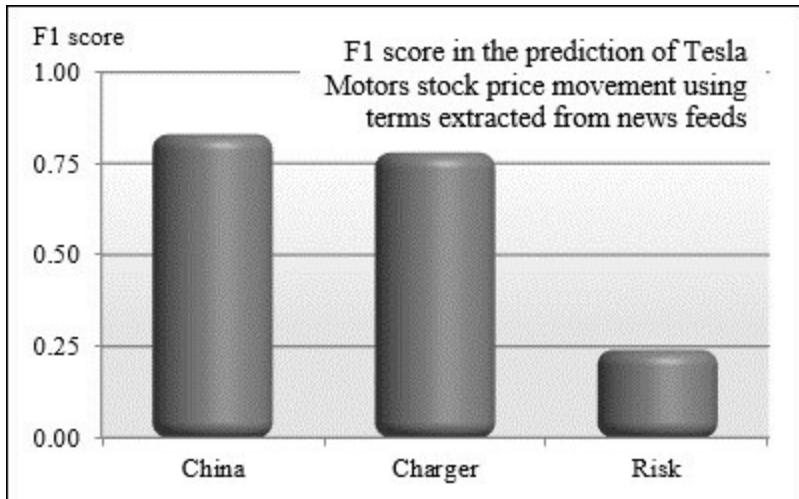
The next chart plots the expected change in the direction of the stock price on the trading day following the press release(s) or news article(s):



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*, *DOWN*). The classification on 15 percent of the labeled data selected for the validation of the classifier has an F_1 score of 0.71. You need to keep in mind that no preprocessing or clustering was performed to isolate the most relevant features/keywords. We initially selected the keywords according to the frequency of their 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 F_1 score for a validation for which only the observations that have a high number of occurrences of a specific keyword are used:



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 classifiers that are more elaborate. However, you should not regard this procedure as the primary methodology for selecting features. It is a by-product of the Naïve Bayes formula applied to models with a very small number of relevant features. Techniques such as the principal components analysis described in *Principal components analysis* in [Chapter 5, Dimension Reduction](#) are available to reduce the dimension of the problem and make Naïve Bayes a viable classifier.

Pros and cons

There is so much information that can be crammed into one chapter. The examples selected in this chapter do not do justice to the versatility and accuracy of the Naïve Bayes family of classifiers.

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

- It is easy to implement and parallelize
- It has a very low computational complexity: $O((n+c)*m)$, where m is the number of features, c is the number of classes, and n is the number of observations
- It handles missing data
- It supports incremental updates, insertions, and deletions

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

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

Summary

Naïve Bayes should come to mind when you are considering creating a model from a labeled dataset, for problems or application for which the features are conditionally independent. Its simplicity and robustness make Naïve Bayes one of the most widely applied supervised learning techniques.

This chapter illustrates the versatility of Naïve Bayes for text mining applications.

However, it should be noted that the requirement of feature independence cannot always be met. In the case of the classification of documents or news releases, Naïve Bayes incorrectly assumes that terms are semantically independent: the two entities *age* and *date of birth* are highly correlated. The discriminative classifiers described in the next few chapters address some of Naïve Bayes' limitations [5:14].

This chapter does not treat temporal dependencies, sequence of events, or conditional dependencies between observed and hidden features. These types of dependency necessitate a different approach to modeling, which is the topic discussed in [Chapter 7, Sequential Data Models](#).

Chapter 7. Sequential Data Models

As seen in the previous chapter, the Naive Bayes model does not make any assumptions regarding the order of events. The likelihood and prior probabilities are computed by aggregating counts. However, some applications such as text or voice recognition, language translation, or decoding, rely on an ordered sequence of states or events [7:1].

[Chapter 3](#), *Data Pre-processing*, introduced some deterministic solutions to modeling sequences of data, collectively known as time series analysis. An alternative probabilistic solution relies on Markov process and models.

The broad universe of Markov models encompasses computational concepts such as the **Markov decision process**, **discrete Markov**, **Markov chain Monte Carlo for Bayesian networks**, and **hidden Markov models**.

The first section of this chapter introduces and describes the hidden Markov model along with the dynamic programming techniques used in the evaluation, decoding, and training of models for sequential events.

The second and last section of the chapter is dedicated to a discriminative model alternative to the hidden Markov model: conditional random fields. Our example leverages the open source CRF Java library authored by *Sunita Sarawagi* from the Indian Institute of Technology, Bombay [7:2].

Markov decision processes

This first section also describes the basic concepts you need to know to understand, develop, and apply the hidden Markov model, starting with 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 [7:3].

Let's consider the following use case. N balls of different colors are hidden in N boxes (one each). The balls can have only three colors {Blue, Red, and Green}. The experimenter draws the balls one by one. The state of the discovery process is defined by the color of the latest ball drawn from one of the boxes: $S_0 = \text{Blue}$, $S_1 = \text{Red}$, and $S_2 = \text{Green}$.

Let $\{\pi_0, \pi_1, \pi_2\}$ be the initial probabilities for having an initial set of colors 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 the following:

$$p(q_t = S_k | q_{t-1} = S_j) = a_{jk}.$$

The probability to draw a red ball at the first attempt is $p(q_{t0} = S_1) = \pi_1$.

The probability to draw a blue ball in the second attempt is as follows:

$$p(q_0 = S_1) p(q_1 = S_0 | q_0 = S_1) = \pi_1 a_{10}.$$

The process is repeated to create a sequence of the state $\{S_t\} = \{\text{Red}, \text{Blue}, \text{Blue}, \text{Green}, \dots\}$ with the following probability:

$$p(q_0 = S_1) \cdot p(q_1 = S_0 | q_0 = S_1) \cdot p(q_2 = S_0 | q_1 = S_0) \cdot p(q_3 = S_2 | q_2 = S_0) \dots = \pi_1 \cdot a_{10} \cdot a_{00} \cdot a_{02} \dots$$

The sequence of states/colors can be represented as follows:

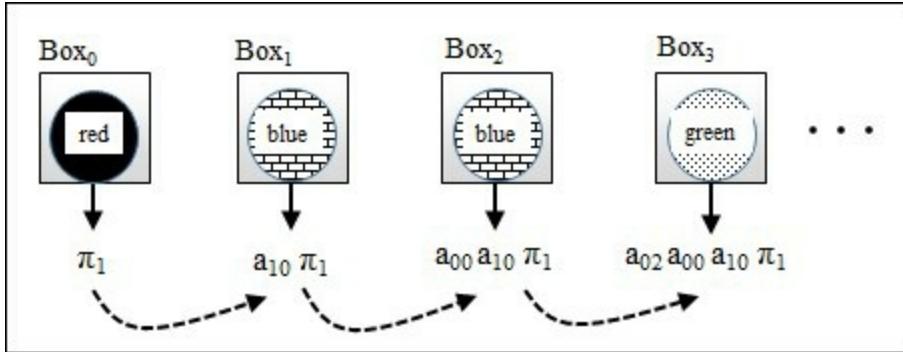


Illustration of the ball and boxes example

Let's estimate the probabilities p using historical data (learning phase):

- 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.
- 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.

Note

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.

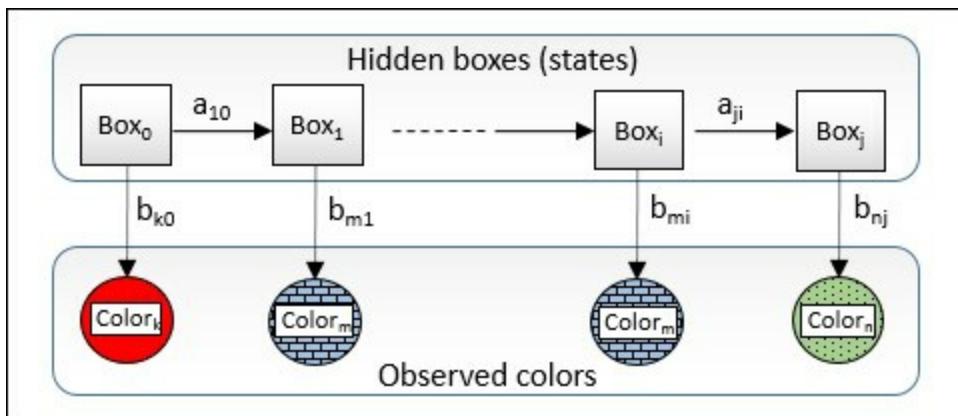
The hidden Markov model (HMM)

A HMM is indeed 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 HMM 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).

Note

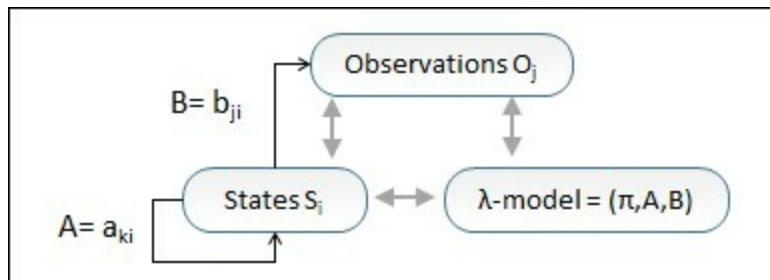
Time invariance:

Contrary to the Kalman filter, for example, the hidden Markov model requires that the transition elements, a_{ji} , 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 or problems**:

- **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	$O = \{o_0, o_1, \dots, o_{T-1}\} = O_{0:T-1}$

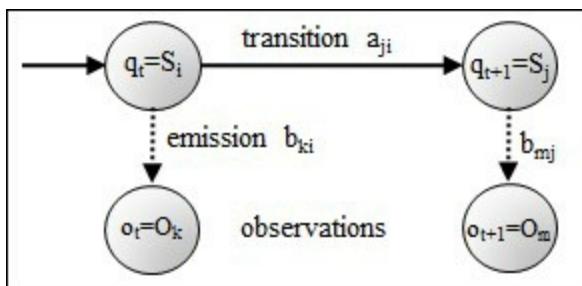
A	The state transition probability matrix	$a_{ji} = p(q_{t+1} = S_i q_t = S_j)$
B	The emission probability matrix	$b_{jk} = p(o_t = O_k q_t = S_j)$
π	The initial state probability vector	$\pi_i = p(q_0 = S_j)$
λ	The hidden Markov model	$\lambda = (\pi, A, B)$

Note

Variance in notation:

Some authors use the symbol z to represent the hidden states instead of q and x to represent the observations O (K. Murphy Machine Learning: A Probabilistic Approach)

For convenience, let's simplify the notation of the sequence of observations and states using the following condensed form: $p(O_{0:T}, q_t | \lambda) = p(O_0, O_1, \dots, O_T, q_t | \lambda)$. It is quite common to visualize a hidden Markov model with a lattice of states and observations 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:

```
case class HMMConfig(  
    numObs: Int,  
    numStates: Int,  
    numSymbols: Int,  
    maxIters: Int,  
    eps: Double) extends Config
```

The input parameters for the class are as follows:

- `numObs`: The number of observations used in the model
- `numStates`: The number of hidden states
- `numSymbols`: The number of observation symbols or features
- `maxIters`: The maximum number of iterations for the HMM training
- `eps`: The convergence criteria for the HMM training.

Note

Consistency with mathematical notation:

The implementation uses `numObs` (with respect to `numStates`, `numSymbols`) 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.

The `HMMConfig` companion object defines the operations on ranges of index of matrix rows and columns. The `foreach` (line 1), `foldLeft(/:)` (line 2), and `maxBy` (line 3) methods are regularly used in each of the three canonical forms:

```
def foreach(i: Int, f: Int => Unit): Unit =  
  (0 until i).foreach(f) //1
```

```

def /:(i: Int, f: (Double, Int) => Double, zero: Double) =
  (0 until i).:/((zero)(f)) //2
  def maxBy(i: Int, f: Int => Double): Int =
    (0 until i).maxBy(f) //3
    ...
}

```

Note

λ notation:

The λ model in HMM should not be confused with the regularization factor discussed in the *Ln roughness penalty* section in [Chapter 9, 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 class, `HMMModel`, using the `DMatrix` class defined in the *Utility classes* section in *Appendix*. The simplest constructor for the `HMMModel` class is invoked in the case where the state-transition probability matrix, the emission probability matrix, and the initial states are known, as shown here:

```

class HMMModel(val A: DMatrix,
                    val B: DMatrix,
                    var pi: Array[Double],
                    val numObs: Int) { //4
  val numStates = A.nRows
  val numSymbols = B.nCols

  def setAlpha(obsSeqNum: Vector[Int]): DMatrix
  def getAlphaVal(a: Double, i: Int, obsId: Int): Double
  def getBetaVal(b: Double, i: Int, obsId: Int): Double
  def update(gamma: Gamma, diGamma: DiGamma, obsSq: Vector[Int])
  def normalize: Unit
}

```

The constructor of the class `HMMModel` has four arguments (line 4):

- **A**: State transition probabilities matrix
- **B**: Omission probabilities matrix

- pi : Initial probability for the states
- numObs : Number of observations

The number of states and symbols are extracted from the dimension of the matrices A and B .

The `HMMModel` class has several methods that will be described in detail whenever required by the execution of the model. The probabilities for the initial states pi are unknown, and therefore, initialized with a random generator of values $[0, 1]$.

Tip

Normalization:

Input states and observation 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.

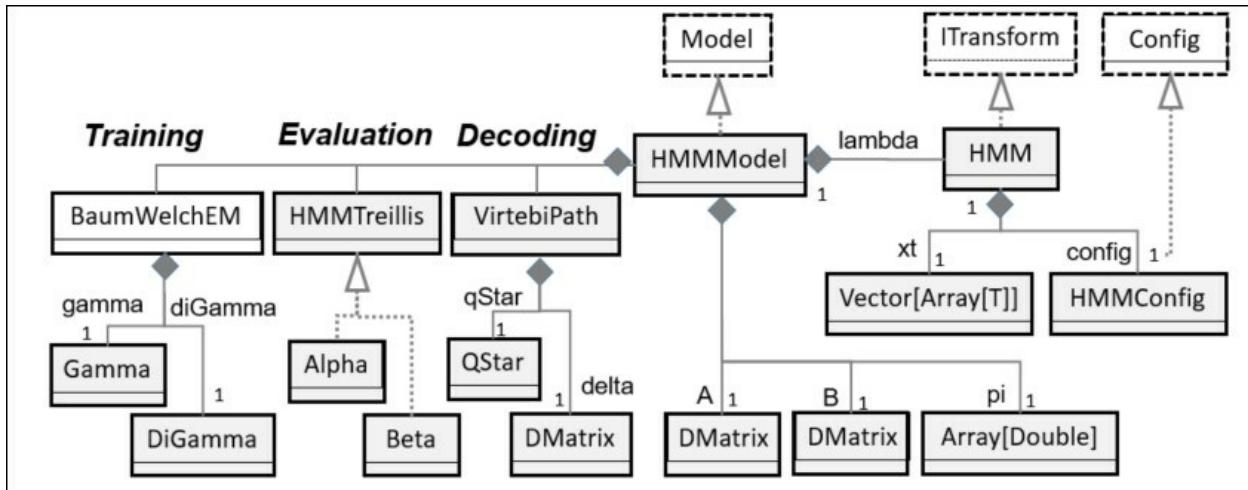
Design

The canonical forms of the HMM are implemented through dynamic programming techniques. These techniques rely on variables that define the state of the execution of the HMM for any of the canonical forms:

- **Alpha** (forward pass): 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** (backward pass): The probability of observing the remainder of the sequence q_t for a specific state $\beta_t(i) = p(O_{t+1:T-1} | q_t = S_i, \lambda)$
- **Gamma**: The probability of being in a specific state given a sequence of observations and a model $\gamma_t(i) = p(q_t = S_i | O_{0:T-1}, \lambda)$
- **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 $Q_{0:T-1}$
- **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 $\gamma_t(i,j) = p(q_t = S_i, q_{t+1} = S_j | O_{0:T-1}, \lambda)$

Each of the parameters is described mathematically and programmatically in the section related to each specific canonical form. The `Gamma` and `DiGamma` classes are used and described in the evaluation canonical form. The `DiGamma` singleton is described as part of the Viterbi algorithm to extract the sequence of states with the highest probability, given a λ model and a set of observations.

The list of dynamic-programming-related algorithms used in any of the three canonical forms is visualized through the class hierarchy of our implementation of the HMM:



Scala classes' hierarchy for HMM (UML class diagram)

The UML diagram omits the utility traits and classes such as `Monitor` or Apache commons math components.

The λ model, the HMM state, and the sequence of observations are all the elements needed to implement the three canonical cases. Each class is described as needed in the description of the three canonical forms of HMM. It is time to dive into the implementation details of each of the canonical forms, starting with the evaluation.

The execution of any of the three canonical forms relies on dynamic programming techniques (refer to the *Overview of dynamic programming* section in *Appendix*) [7:5]. The simplest of the dynamic programming techniques is a single traversal of the observations/state chain.

Evaluation (CF-1)

The objective is to compute the probability (or likelihood) of the observed sequence O_t given a λ model. A dynamic programming technique is used to break down the probability of the sequence of observations into two probabilities (M1):

$$p(O_{0:T-1} | \lambda) = \alpha p(O_{0:t} | \lambda) \cdot p(O_{t+1:T-1} | \lambda)$$

The likelihood is computed by marginalizing over all the hidden states [7:6] $\{S_i\}$ (M2):

$$p(O_{0:T-1} | \lambda) = \sum_{i=0}^{N-1} p(O_{0:T-1}, q_t = S_i | \lambda)$$

If we use the notation introduced in the previous chapter for alpha and beta variables, the probability for the observed sequence O_t given a λ model can be expressed as (M3):

$$p(O_{0:T-1} | \lambda) = \sum_i \alpha_t(i) \cdot \beta_t(i)$$

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 (forward pass)

The computation of the probability of observing a specific sequence, given a sequence of hidden states and a λ model relies on a two-pass algorithm. The alpha algorithm consists of the following steps:

- Compute the initial alpha value [M4]. The value is then normalized by

- the sum of alpha values across all the hidden states [M5].
- Compute the alpha value iteratively for the time θ to time t , then normalize by the sum of alpha values for all states [M6].
- The final step is the computation of the log of the probability of observing the sequence [M7].

Note

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 as follows:

Note

M4: Initialization:

$$\alpha_0(i) = \pi_i \cdot b_i(O_0)$$

M5: Normalization of initial values: $N-1$

$$\hat{a}_0(i) = \alpha_0(i) \Big/ \sum_{j=0}^{N-1} \alpha_0(j)$$

M6: Normalized summation:

$$\alpha_t(i) = \sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji} b_i(O_t) \quad c_t = 1 \Big/ \sum_{i=0}^{N-1} \alpha_t(i) \quad \hat{a}_t(i) = \alpha_t(i) \cdot c_t$$

M7: 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 start with the implementation of the Alpha phase, using the mathematical expressions referenced in the previous section. The alpha and beta values have to be normalized [M3], and therefore, we define a base class, `HMMTreillis`, for the alpha and beta algorithms that implements the normalization:

```
class HMMTreillis(lambda: HMMModel) { //5
    var treillis: DMatrix = _           //6
    val ct = Array.fill(numObs)(0.0)

    def normalize(t: Int): Unit = { //7
        ct.update(t, /:(numStates, (s,n) => s+treillis(t,n)))
        treillis /= (t, ct(t))
    }
}
```

The class `HMMTreillis` has two configuration parameters: The number of observations `numObs`, and the number of states `numStates` (line 5). The variable `treillis` represents the scaling matrix used in the alpha (or forward) and beta (or backward) passes (line 6).

The normalization method, `normalize`, implements the formula M6 by re-computing the scaling factor, `ct` (line 7).

Note

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 composition 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 expressions, M4, M5, and

M6:

```
class Alpha(lambda: HMMModel, obsSeq: Vector[Int]) //8
extends HMMTreillis(lambda) {

    val alpha: Option[Double] = Try {
        treillis = lambda.setAlpha(obsSeq) //9
        normalize(0) //10
        sumUp //11
    }.toOption

    override def isInitialized: Boolean = alpha.isDefined

    def sumUp: Double = {
        foreach(1, lambda.numObs, t => {
            updateAlpha(t) //12
            normalize(t) //13
        })
        val last = lambda.numObs-1
        /:(lambda.numStates, (s,k) => s + treillis(last,k))
    }

    def updateAlpha(t: Int): Unit =
        foreach(lambda.numStates, i => { //14
            val newAlpha = lambda.getAlphaVal(treillis(t-1, i)
            treillis += (t, i, newAlpha, i, obsSeq(t)))
        })

    def logProb: Option[Double] = alpha.map(m =>
        /:(lambda.numObs), (s,t) => log(ct(t)), log(m) //15
    )
}
```

The `Alpha` class has two arguments: the `lambda` model and the sequence of observations `obsSeq` (line 8). The definition of the scaling factor, `alpha` initializes the scaling matrix, `treillis`, using the `HMMModel.setAlpha` method (line 9), normalizes the initial value of the matrix by invoking `HMMTreillis.normalize` method for the first observation, (line 10) and sums the matrix element to return the scaling factor by invoking `sumUp` (line 11).

The method `setAlpha` is implemented the mathematical expression M4 as follows:

```
def setAlpha(obsSeq: Array[Int]): DMatrix =
    (0 until numStates)./:(DMatrix(numObs, numStates)) (
```

```

        (m,j) => m += (0, j, pi(j)*B(j, obsSq.head)))
    }
}

```

The fold function (`/:`) generates an instance of the class `DMatrix` described in *Utility classes* section in *Appendix*.

The `sumUp` method implements the mathematical expression M6 as follows:

- Update the matrix `treillis` of scaling factor in method `updateAlpha` (line 12)
- Normalize all scaling factors for all the remaining observations (line 13)

The method `updateAlpha` updates the scaling matrix `treillis` by computing all the factor alpha for all states (line 14). The method `logProb` implements the mathematical expression M7. It computes the logarithm of the probability to observe a specific sequence, given the sequence of states and a predefined λ model (line 15).

Note

Log probability:

The method `logProb` 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 (backward pass)

The computation of beta values is similar to the `Alpha` class except that the iteration executes backward on the sequence of states.

The implementation of `Beta` is like the alpha class:

- Compute [M5] and normalize [M6] the value of beta at $t=0$ across states
- Compute and normalize iteratively the beta at the time $T-1$ to t updated from its value at $t+1$ [M7]

M8: Initialization of beta:

$$\beta_{T-1}(t) = 1$$

M9: Normalization of initial beta values:

$$\hat{\beta}_{T-1}(i) = \beta_{T-1}(i) \sqrt{\sum_{j=0}^{N-1} \beta_{T-1}(j)}$$

M10: Normalized summation of beta:

$$\beta_t(i) = \sum_{j=0}^{N-1} \beta_{t-1}(j) \cdot a_{ij} \cdot b_j(O_{t+1}) \quad c_t = \sqrt{\sum_{j=0}^{N-1} \beta_t(j)} \quad \hat{\beta}_t(i) = \beta_t(i) \cdot c_t$$

The definition of the class for the `Beta` class is similar to the `Alpha` class:

```
class Beta(lambda: HMMModel, obsSeq: Vector[Int])
extends HMMTreillis(lambda) {

    val initialized: Boolean //16

    override def isInitialized: Boolean = initialized
    def sumUp: Unit = //17
        (lambda.numObs-2 to 0 by -1).foreach(t => { //18
            updateBeta(t) //19
            normalize(t)
        })
    }

    def updateBeta(t: Int): Unit =
        foreach(lambda.numStates, i => {
            val newBeta = lambda.getBetaVal(treillis(t+1, i)
                treillis += (t, i, newBeta, i, obsSeq(t+1))) //20
        })
}
```

Contrary to the `Alpha` class, the `Beta` class does not generate an output value. The class `Beta` has a Boolean attribute, `initialized`, to indicate whether the constructor has executed successfully (line 16). The constructor updates and normalizes the beta matrix by traversing the sequence of observations backward from before the last observation to the first.

The `sumUp` method is similar to `Alpha.sumUp` (line 17). It traverses the sequence of observations backward (line 18), and updates the beta scaling matrix as defined in the mathematical expression M9 (line 19). The implementation of mathematical expression M10 in the method `updateBeta` is similar to the alpha pass: it updates the scaling matrix `treillis` with the `newBeta` values computed in the lambda model (line 20).

```
val initialized: Boolean = Try {  
    treillis = DMatrix(lambda.numObs, lambda.numStates)  
    treillis += (lambda.numObs-1, 1.0) //21  
    normalize(lambda.numObs-1) //22  
    sumUp //23  
}.isSuccess
```

The initialization of the beta scaling matrix, `treillis`, of type `DMatrix`, assigns the value `1.0` to the last observation (line 21), and normalizes the beta values for the last observation as defined in M8 (line 21). It implements the mathematical expression M9 and M10 by invoking `sumUp` method (line 23).

Note

Constructor and initialization:

The alpha and beta values are computed within the constructors of their respective class. The client code has to validate if these instances are valid by invoking `isInitialized`.

What is the value of a model if it cannot be created? The next canonical form, CF2, leverages dynamic programming and recursive functions to extract the λ model.

Training (CF-2)

The objective of this canonical form is to extract the λ model given a set of observations and a sequence of states. It is similar to the training of a classifier. The simple dependency of a current state on the previous state enables an implementation using an iterative procedure, known as the **Baum-Welch estimator or expectation-maximization (EM)**.

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 $t=0$) [M11].
2. Compute and normalize the state's transition probabilities matrix A [M12].
3. Compute and normalize the matrix of emission probabilities B [M13].
4. Repeat steps 2 and 3 until the change of likelihood is insignificant.

The algorithm uses the digamma and summation gamma classes.

Note

The Baum-Welch algorithm

M11: Joint probability of the state q_i at t and q_j at $t+1$ (digamma):

$$\gamma_t(i, j) = p(q_t = S_i, q_{t+1} = S_j | O, \lambda)$$
$$\gamma_t(i, j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{j=0}^{N-1} \alpha_t(i) \beta_t(j)}$$

M12: The initial probabilities vector: $N-1$ and sum of joint probabilities for all the states (gamma):

$$\hat{\pi}_i = \gamma_0(i) \quad \gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i, j)$$

M13: Update of the transition probabilities matrix:

$$\hat{a}_{ij} = \frac{\sum_{t=0}^{T-1} [\gamma_t(i, j)]}{\sum_{t=0}^{T-1} \gamma_t(i)}$$

M14: Update of the emission probabilities matrix:

$$\hat{b}_{ij} = \frac{\sum_{t=0}^{O_j} \gamma_t(i)}{\sum_{t=0}^{T-1} \gamma_t(i)}$$

The Baum-Welch algorithm is implemented in the class `BaumWelchEM` and requires the following two inputs (line 24):

- The λ model, `lambda` computed from the configuration, `config`
- The sequence (vector) of observations, `obsSq`:

```
class BaumWelchEM(config: HMMConfig, obsSq: Vector[Int]) { //24
    val lambda = HMMModel(config)
    val diGamma = new DiGamma(lambda) //25
    val gamma = new Gamma(lambda) //26
    val maxLikelihood: Option[Double] //27
}
```

The class `DiGamma` defines the joint probabilities for any consecutive states (line 25):

```
Final class DiGamma(lambda: HMMModel) {
    val diGamma = Array.fill(lambda.numObs-1)
        (DMatrix(lambda.numStates))
```

```

def update(alpha: DMatrix, beta: DMatrix, A: DMatrix,
           B: DMatrix, obsSeq: Array[Int]): Try[Int]
}

```

The variable `diGamma` is an array of matrices representing the joint probabilities of two consecutive states. It is initialized through an invocation of the method `update` which implements the mathematical expressions M11.

The class `Gamma` computes the sum of the joint probabilities across all the states (line 26):

```

Final class Gamma(lambda: Model) {
  val gamma = DMatrix(lambda.numObs, lambda.numStates)
  def update(alpha: DMatrix, beta: DMatrix): Unit
}

```

The method `update` of the class `Gamma` implements the mathematical expression M12.

Note

Source code for Gamma and DiGamma:

The classes `Gamma` and `DiGamma` implement the mathematical expressions for the Baum-Welch algorithm. The `update` method uses simple linear algebra and is not described: refer to the documented source code for details.

The maximum likelihood for the sequence of states given an existing lambda model and a sequence of observations (line 27) is implemented by the value, `maxLikelihood`. It is computed using the tail recursive method `getLikelihood` as follows:

```

val maxLikelihood: Option[Double] = Try {
  @tailrec
  def getLikelihood(likelihood: Double, index: Int): Double = {
    lambda.update(gamma, diGamma, obsSq) //28
    frwrdBckwrdLattice match { //29
      case None => throw new IllegalStateException("...")
    }
  }
}

```

```

case Some(estimate) =>
    val diff = likelihood - estimate

    if( diff < config.eps )
        estimate //30
    else if (index >= config.maxIters) //31
        throw new IllegalStateException(" ... ")
    else getLikelihood(estimate, index+1)
}

val likelihood = frwrBckwrdLattice.map(getLikelihood(_, 0))
lambda.normalize //32
likelihood
}

```

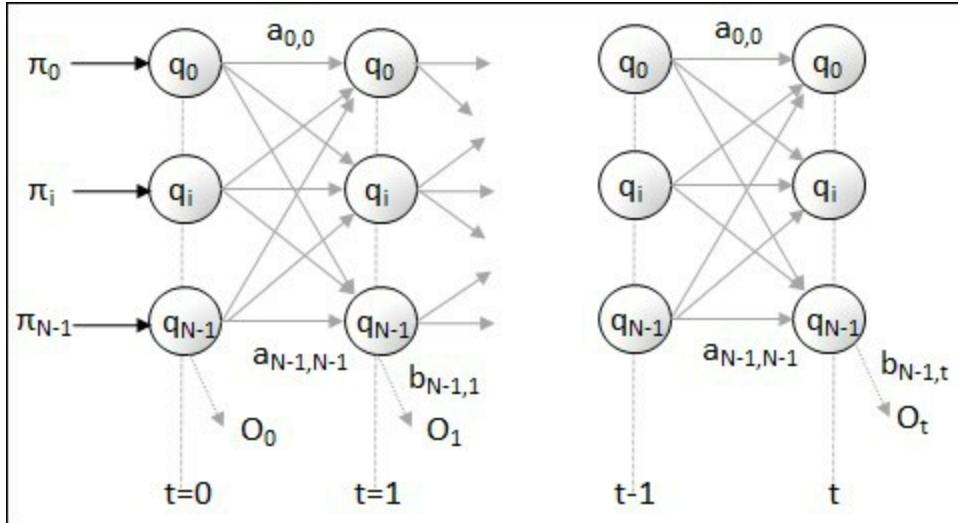
The value `maxLikelihood` is implementing the mathematical expressions M13 and M14. The recursive method, `getLikelihood`, updates the `lambda` model matrices, A , B , and initial state probabilities, pi (line 28). The likelihood for the sequence of states is recomputed using the forward-backward lattice algorithm implemented in the `frwrBckwrdLattice` method (line 29).

Note

Update of lambda model

The `update` method of object `HMMModel` uses simple linear algebra and is not described: refer to the documented source code for details.

The core of the Baum-Welch expectation maximization is the iterative forward and backward update of the lattice of states and observations between time t and $t+1$. The lattice-based iterative computation is illustrated in the following diagram:



Visualization of HMM graph lattice for the Baum-Welch algorithm

```
def frwrdBckwrdLattice: Double = {
    val pAlpha = Alpha(lambda, obsSq) //33
    val beta = Beta(lambda, obsSq).getTreillis //34
    val alphas = pAlpha.getTreillis

    gamma.update(alphas, beta) //35
    diGamma.update(alphas, beta, lambda.A, lambda.B, obsSq)
    pAlpha.alpha
}
```

The forward-backward algorithm uses the `Alpha` class for the computation/update of the `lambda` model in the forward pass (line 33) and the `Beta` class for the update of `lambda` in the backward pass (line 34). The joint probabilities related matrices, `gamma` and `diGamma` are updated at each recursion (line 35) reflecting the iteration of mathematical expressions M11 to M14.

The recursive computation of `maxLikelihood` exists if the algorithm converges (line 30). It throws an exception if the maximum number of recursions is exceeded (line 31).

Decoding (CF-3)

This last canonical form consists of extracting the most likely sequence of states $\{q_t\}$ given a set of observations O_t and a λ model. Solving this problem requires, once again, a recursive algorithm.

The Viterbi algorithm

The extraction of the best state sequence (the sequence of 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** [7:7]. 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 .

Note

M12: Definition of delta function:

$$\delta_t(i) = \max_{q_j: \{o, T-1\}} p(q_{0:T-1} = S_i, O_{o:T-1} | \lambda)$$

M13: Initialization of delta:

$$\delta_0(i) = \pi_i b_i(O_0) \psi_0(i) = 0 \forall i$$

M14: Recursive computation of delta:

$$\delta_t(j) = \max_i (\delta_{t-1}(i) \cdot a_{ij} \cdot b_j(O_t)) \quad \psi_t(j) = \arg \max_i (\delta_{t-1}(i) \cdot a_{ij})$$

M15: Computation of the optimum state sequence $\{q\}$:

$$q^*_{t+1} = \psi_{t+1}(q^*_{t+1}) \quad q^*_T = \arg \max_i \delta_T(i)$$

The class, `ViterbiPath`, implements the Viterbi algorithm. Its purpose is to compute the optimum sequence (or path) of states given a set of observations and a λ model. The optimum sequences or path of states is computed by maximizing the delta function.

The constructors for the class, `ViterbiPath`, have the same arguments as the forward, backward, and Baum-Welch algorithm: the lambda model and the set of observations.

```
class ViterbiPath(lambda: HMMModel, obsSq: Vector[Int]) {
    val numObs = lambda.numObs
    val numStates = lambda.numStates
    val psi = Array.fill(numObs)(Array.fill(numStates)(0)) //35
    val qStar = new QStar(numObs, numStates) //36

    val delta = //37
        (0 until numStates)./:(DMatrix(numObs, numStates))((m, n) =>{
            psi(0)(n) = 0
            m += (0, n, lambda.pi(n) * lambda.B(n, obsSq.head))
        })

    val path = HMMPrediction(viterbi(1), qStar()) //38
}
```

In accordance with the mathematical expressions for the Viterbi algorithm seen previously, the following matrices have to be defined:

- `psi`: The matrix of indices of `numObs` observations by indices of `numStates` states (line 35).
- `qStar`: The optimum sequence of states at each recursion of the Viterbi algorithm (line 36).
- `delta`: The sequence to have the highest probability path for the first n observations. It also sets the `psi` values for the first observation to 0. (line 37).

All members of the `ViterbiPath` class are private except `path`, which defines

the optimum sequence or path of states given the observations, `obsSq` (line 38).

The matrix that defines the maximum of probability, `delta`, of any sequence of states, given the `lambda model` and the observation `obsSq`, is initialized using the mathematical expression M13 (line 37). The predictive model returns the path or optimum sequence of states as an instance of `HMPrediction`:

```
case class HMPrediction(likelihood: Double, states: Array[Int])
```

The first argument of `likelihood` is computed by the recursive method, `viterbi`. The indices of the states in the optimum sequence, `states` are computed by the class `QStar` (line 38).

Let's look under the hood of the `viterbi` recursive method:

```
@tailrec
def viterbi(t: Int): Double = {
  (0 until numStates).foreach(updateMaxDelta(t, _)) //39

  if( t == obsSq.size-1) { //40
    val (index, prob) = (0 until numStates)
      .map(i => (i, delta(t, i))).maxBy(_.value) //41
    qStar.update(t+1, index, psi) //42
    prob
  }
  else viterbi(t+1) //43
}
```

The recursion started on the second observation as the parameters `qStar`, `psi` and `delta` have already been initialized in the constructor. The recursive implementation invokes the method, `updateMaxDelta`, to update the indexing matrix `psi` and the highest probability for any state as follows:

```
def updateMaxDelta(t: Int, j: Int): Unit = {
  val (psiVal, index) = (0 until numStates)
    .map(i => (i, delta(t-1, i)*lambda.A(i, j)))
    .maxBy(_.value) //44
  psi(t)(j) = psiVal
  delta += (t, j, index) //45
}
```

The method, `updateMaxDelta`, implements the mathematical expression M14, that extracts the index of the state that maximizes `psi` (line 44). The `delta` probability matrix and the indexing matrix `psi` are updated accordingly (line 45).

The `viterbi` method is called recursively for the remaining observations but the last one (line 43). At the last observation of index, `obsSeq.size-1`, the algorithm executes the mathematical expression M15 which is implemented in the class `QStar` (line 42).

Note

QStar:

The `QStar` class and its method `update` use linear algebra and are not described: refer to the documented source code and Scaladocs files for details.

This implementation of the decoding form of the hidden Markov model completes the description of the hidden Markov model and its implementation in Scala. Now, let's put this knowledge into practice.

Putting it all together

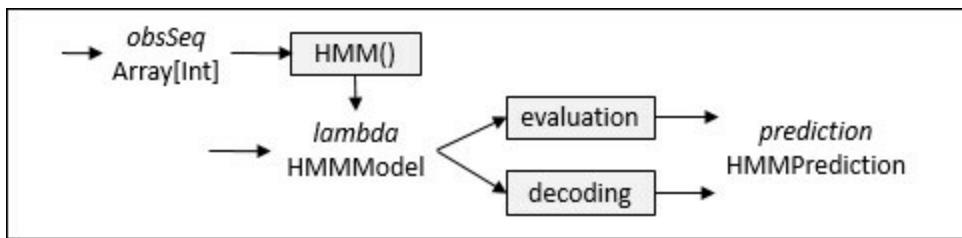
The main class `HMM` implements the three canonical forms. A view bound to an array of integers is used to parameterize the `HMM` class. We assume that a time series of continuous or pseudo-continuous values is quantized into discrete symbol values.

The `@specialized` annotation ensures that the byte code is generated for the `Array[Int]` primitive without executing the conversion implicitly declared by the bound view.

There are two modes that execute any of the three canonical forms of the hidden Markov model:

- `ViterbiPath` class: The constructor initializes/trains a model similarly to any other learning algorithm described in the book as described in *Design template for classifiers* section of *Appendix*. The constructor generates the model by executing the Baum-Welch algorithm. Once the model is successfully created, it can be used for decoding or evaluation.
- `ViterbiPath` singleton: The companion provided two methods, `decode`, and `evaluate`, for the decoding and evaluation of the sequence of observations using HMM.

The two modes of operations are described in the following diagram:



Computational flow for hidden Markov model

Let's complete our implementation of the HMM with the definition of its class. The class `HMM` is defined as a data transformation using a `model` implicitly generated from a training set, `xt`, as described in the *Monadic data*

transformation section in [Chapter 2, Data Pipelines](#) (Line 46):

```
type V = HMMPrediction //47

class HMM[@specialized(Double) T: ToDouble] (
    config: HMMConfig,
    xt: Vector[Array[T]],
    form: HMMForm)
    (implicit quantize: Array[T] => Int)
extends ITransform[Array[T], V] with Monitor[Double] //46

val obsSq: Vector[Int] = xt.map(quantize(_)) //48

val model: Option[HMMModel] = train //49
override def |>: PartialFunction[Array[T], Try[V]] //50
}
```

The `HMM` constructor takes four arguments (line 46):

- `config`: Configuration of the HMM that is the dimension of lambda model and execution parameters
- `xt`: Multi-dimensional time series of observations which features have the type `T`
- `form`: Canonical form to be used once the model is generated (Evaluation or Decoding)
- `quantize`: Quantization function that converts an observation of type `Array[T]` to an `Int`.

The constructor has to override the type `V` (`HMMPrediction`) of its output data (line 47) declared in the abstract class, `ITransform`. The structure of the class, `HMMPrediction`, has been defined in the previous section. The `Monitor` trait is used to collect profiling information during training (refer to the *Monitor* section under *Utility classes* in the *Appendix*).

The time series of observations, `xt`, is converted into a vector of observed stated, `obsSq`, by applying the quantization function, `quantize`, to each observation (line 48). As with any supervised learning technique, the model is created through training (line 49). Finally, the polymorphic predictor `|>` invokes either the `decode` method or the `evaluate` method (line 50).

The `train` method consists of the execution of the Baum-Welch algorithm

and returns the lambda model:

```
def train: Option[HMMModel] = Try {
  BaumWelchEM(config, obsSq).lambda
}.toOption
```

Finally, the predictor `|>` is a simple wrapper to the evaluation form (`evaluate`) and the decoding form (`decode`):

```
override def |>: PartialFunction[Array[T], Try[V]] = {
  case ySeq: Array[T] if(isModel && x.length > 1) =>
    form match {
      case _: EVALUATION =>
        evaluation(model.get, Vector[Int](quantize(ySeq)))
      case _: DECODING =>
        decoding(model.get, Vector[Int](quantize(ySeq)))
    }
}
```

The protected `evaluation` method of the companion object, `HMM`, is a wrapper around the `Alpha` computation:

```
def evaluation(
  model: HMMModel,
  obsSq: Vector[Int]): Try[HMPrediction] = Try {
  HMPrediction(-Alpha(model, obsSq).logProb.get, obsSq.toArray)
}
```

The `evaluate` method of `HMM` object exposes the evaluation canonical form:

```
def evaluate[T: ToDouble] (
  model: HMMModel,
  xt: Vector[Array[T]])
)(implicit quantize: Array[T] => Int): Option[HMPrediction] =
  evaluation(model, xt.map(quantize(_))).toOption
```

The `decoding` method wraps the Viterbi algorithm to extract the optimum sequences of states:

```
def decoding(model: HMMModel, obsSq: Vector[Int]): Try[HMPrediction] = Try(ViterbiPath(model, obsSq).path)
```

The `decode` method of `HMM` object exposes the decoding canonical form:

```
def decode[T: ToDouble] (
  model: HMMModel,
  xt: Vector[Array[T]])
(implicit quantize: Array[T]=>Int): Option[HMPrediction]=
  decoding(model, xt.map(quantize(_))).toOption
```

Note

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 1 – Training

Our first test case is to train an HMM to predict the sentiment of investors as measured by the weekly sentiment survey of the members of the **American Association of Individual Investors (AAII)** [7:8]. The goal is to compute the transition probabilities matrix, A , the emission probabilities matrix, B , and the steady state probability distribution, π , given the observations and hidden states (training canonical 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*.

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

t0	0.38	0.15	0.47	2.53	1.0
t1	0.41	0.25	0.34	1.68	0.53
t2	0.25	0.35	0.40	0.71	0.0
...

The sequence of non-normalized observations (ratio of bullish sentiment over bearish sentiment) is defined in a CSV file as follows:

```

val OBS_PATH = "resources/data/supervised/hmm/obsprob.csv"
val NUM_SYMBOLS = 6
val NUM_STATES = 5
val EPS = 1e-4
val MAX_ITERS = 150
val observations = Vector[Double](
  0.01, 0.72, 0.78, 0.56, 0.61, 0.56, 0.45
)

val quantize = (x: Array[Double]) =>
  (x.head*(NUM_STATES+1)).floor.toInt //51
val xt = observations.map(Array[Double](_))

val config = HMMConfig(xt.size, NUM_STATES, NUM_SYMBOLS,
  MAX_ITERS, EPS)
val hmm = HMM[Array[Int]](config, xt) //52

```

The constructor for HMM class requires an implicit conversion, $T \Rightarrow$ $\text{Array}[\text{Int}]$ which is implemented by the function `quantize` (line 51). The model, `hmm.model`, is created by instantiating a `HMM` class with a predefined configuration and a sequence, `obsSeq`, of observed states (line 52).

The training of HMM generates the following state-transition probabilities matrix:

--	--	--	--	--	--

A	1	2	3	4	5
1	0.090	0.026	0.056	0.046	0.150
2	0.094	0.123	0.074	0.058	0.0
3	0.093	0.169	0.087	0.061	0.056
4	0.033	0.342	0.017	0.031	0.147
5	0.386	0.47	0.314	0.541	0.271

The emission matrix is as follows:

B	1	2	3	4	5	6
1	0.203	0.313	0.511	0.722	0.264	0.307
2	0.149	0.729	0.258	0.389	0.324	0.471
3	0.305	0.617	0.427	0.596	0.189	0.186
4	0.207	0.312	0.351	0.653	0.358	0.442
5	0.674	0.520	0.248	0.294	0.259	0.03



Test case 2 – Evaluation

The objective of the evaluation is to compute the probability of the observed data, xt , given a λ model (A_0 , B_0 , PI_0).

```
val A0 = Array[Array[Double]](
  Array[Double](0.21, 0.13, 0.25, 0.06, 0.11, 0.24),
  Array[Double](0.31, 0.17, 0.18, 0.04, 0.19, 0.11),
  ...
)
val B0 = Array[Array[Double]](
  Array[Double](0.61, 0.39),
  Array[Double](0.54, 0.46),
  ...
)
val PI0 = Array[Double](0.26, 0.04, 0.11, 0.26, 0.19, 0.14)
val data = Vector[Double](
  0.0, 1.0, 2.0, 1.0, 3.0, 0.0, 1.0, 0.0, 1.0, 2.0,
  3.0, 1.0, 0.0
)

val xt = data.map(Array[Double](_))
val max = data.max
val min = data.min
implicit val quantize = (x: Array[Double]) =>
  ((x.head / (max - min) + min) * (B0.head.length - 1)).toInt //55

val lambda =
  HMMModel(DMatrix(A0), DMatrix(B0), PI0, xt.length) //53
evaluation(lambda, xt).map(_.toString).map(show(_)) //54
```

The model is created directly by converting the state-transition probabilities, A_0 , and emission probabilities, B_0 , as matrices of type `DMatrix` (line 53). The evaluation method generated a `HMPrediction` object which is stringized, then displayed in the standard output (line 54).

The quantization method consists of normalizing the input data over the number (or range) of symbols associated with the lambda model. The number of symbols is the size of the rows of the emission probabilities matrix B . In this case the range of the input data is $[0.0, 3.0]$. The range is normalized using the linear transform, $f(x) = x/(max - min) + min$, then adjusted for the

number of symbols (or values for states) (line 55).

The quantization function, `quantize`, has to be explicitly defined before invoking the `evaluation` method.

Note

Test case for decoding:

Refer to the source code and the API documents for the test case related to the decoding form.

HMM as filtering technique

The evaluation form of the hidden Markov model is very suitable for filtering data for discrete states. Contrary to time series filters such as the Kalman filter introduced in the *Kalman filter* section in [Chapter 3, Data Pre-processing](#), HMM requires data to be stationary in order to create a reliable model. However, the hidden Markov model overcomes some of the limitations of analytical time series analysis. Filters and smoothing techniques assume that the noise (frequency mean, variance, and covariance) is known, and usually follows a Gaussian distribution.

The hidden Markov model does not have such a restriction. Filtering techniques such as moving averaging techniques, discrete Fourier transforms, and Kalman filter applies to both discrete and continuous states while HMM does not. Moreover, the extended Kalman filter can estimate nonlinear states.

Conditional random fields

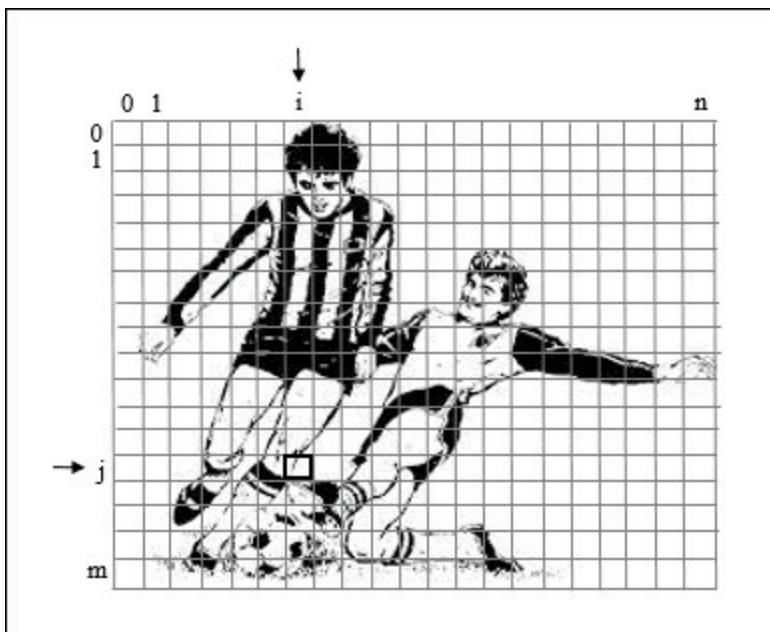
Discriminative models such as linear, logistic regression, multilayer perceptron, and support vector machines are described in part 3 – Gradient-based Learning. However, it would make sense to introduce a discriminative alternative to HMM in this chapter dedicated to sequential data models.

The **conditional random field (CRF)** is a discriminative machine learning algorithm introduced by *John Lafferty, Andrew McCallum, and Fernando Pereira* [7:9]. 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.

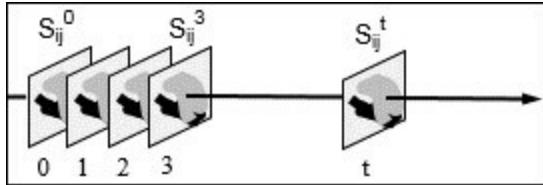
Introduction to CRF

Let's consider the problem of detecting a fault 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:



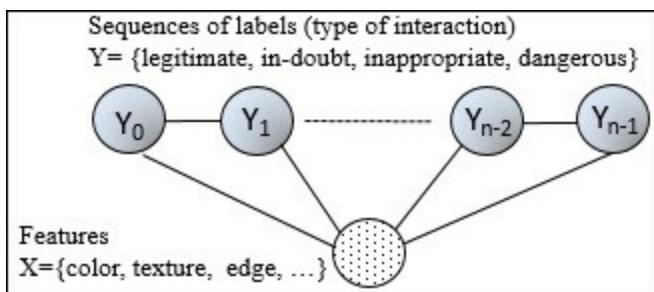
Example of image processing problem requiring machine learning

The analysis of the video consists of segmenting each video frame and extracting image features such as colors or edges [7:10]. A simple segmentation scheme consists of breaking down each video frame into tiles or groups of pixels indexed by their coordinates on the screen. A time sequence is then created for each tile, S_{ij} , as represented in the following image:



Learning strategy for pixel in a sequence of video frames

The image segment, S_{ij} , is one of the labels that are associated with multiple observations. The same feature's 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 is 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:

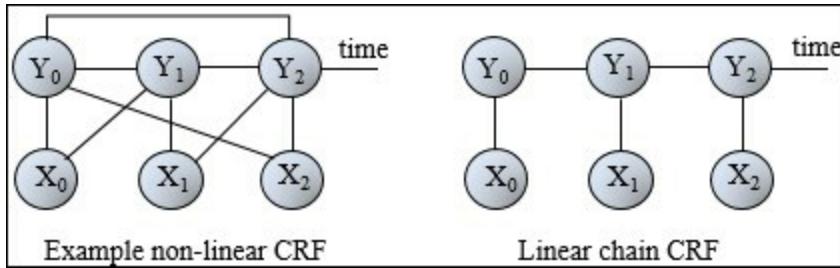


Illustration of non-linear and linear chain CRF

One main advantage of the linear chain CRF is that the maximum likelihood, $p(Y|X, w)$, can be estimated using dynamic programming techniques such as the Viterbi algorithm used in the HMM. From now on, the section focuses exclusively on the linear chain, CRF, 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.

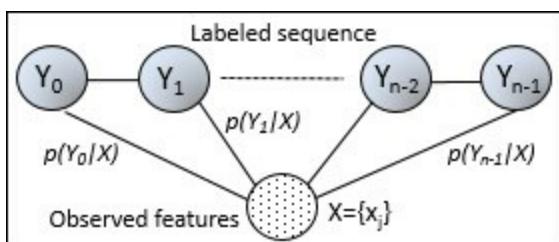
Note

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 $p(Y_1 | X, Y_{j \neq 1}) = p(Y_1 | X, Y_0, Y_2)$ and $p(Y_i | X, Y_{j \neq i}) = p(Y_i | X, Y_{i-1}, Y_{i+1})$.

The conditional random fields introduce a new set of entities and a new terminology:

- **Potential functions (fi):** These strictly positive, real value functions represent a set of constraints on the configurations of random variables. They do not have any obvious probabilistic interpretation.
- **Identity potential functions:** These are potential functions, $I(x, t)$, that take 1 if the condition on the feature x at time t is true, and 0 otherwise.
- **Transition feature functions:** Simply known as **feature functions**, t_i , are potential functions that take a sequence of features $\{X_i\}$, 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_j , are potential functions that take the sequence of features $\{X_i\}$, the current label, Y_i , and the index, i . They play a similar role as the emission factors in the HMM.

A CRF defines the log probability of a 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 as follows:

Note

CRF conditional distribution:

M1: The log probability of a label's sequence y , given an observation x :

$$\log f_i(y_{i-1}, y_i, x, i) = w_c + \sum_{i=0}^{K-1} w_i t_i(y_{i-1}, y_i, x, i) + \sum_{j=0}^{K-1} \mu_j s_j(y_i, x, i)$$

M2: Transition feature functions:

$$t_i(y_{i-1}, y_i, x, i) = I(y_{i-1} = l_1) \cdot I(y_i = l_2) \cdot I(x = 0)$$

M3: Using the notation:

$$F_i(y, x) = \sum_{j=0}^{K-1} f_i(y_{j-1}, y_j, x, i) \quad \log p(x, \lambda) \propto \sum_{j=0}^{K-1} w_j F_j(x, y)$$

M4: 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*.

Regularized 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 underperformance. 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 over perform, 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.

Note

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 can 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 // $Y(1)$

Macy's // $Y(2)$

from // $Y(3)$

Buy // $Y(4)$

to // $Y(5)$

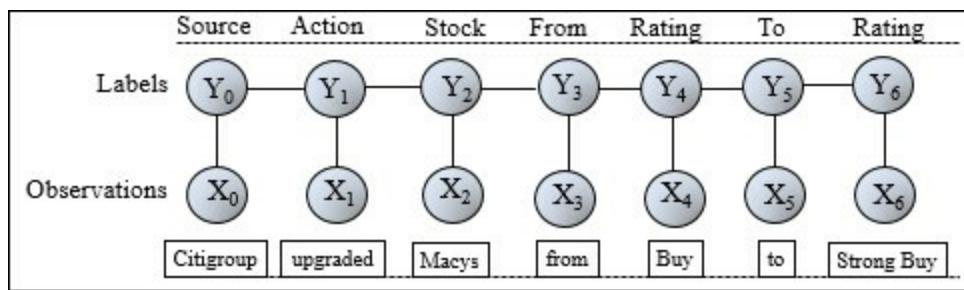
Strong Buy // $Y(6) = 7$

Note

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.

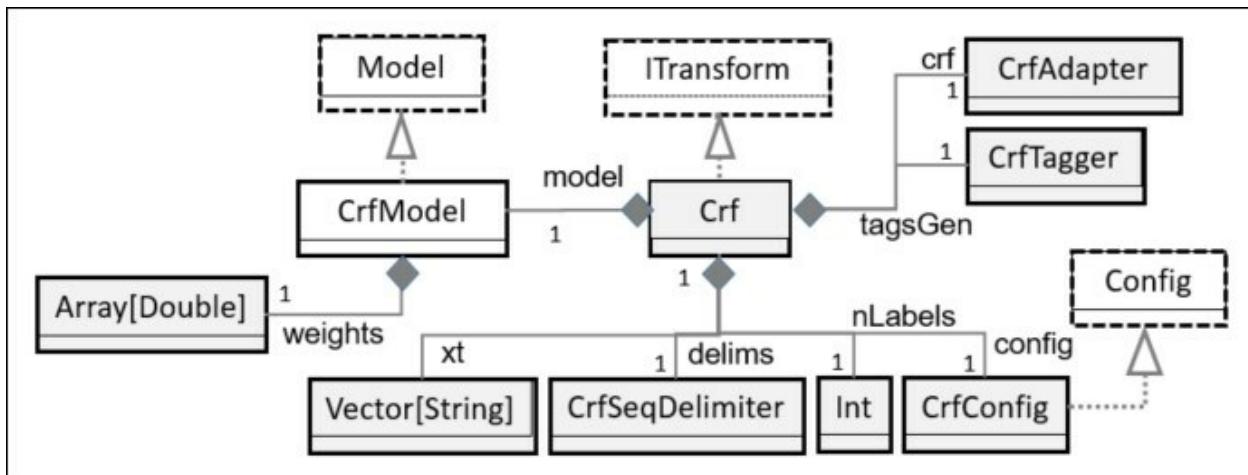
Design

The implementation of the conditional random fields follows the design template for classifier, which is described in the *Design template for classifiers* section under *Source code considerations* in *Appendix*.

Its key components are as follows:

- A `CrfModel` model of the type `Model`, is initialized through training during the instantiation of the classifier. A model is an array of `weights`.
- The predictive or classification routine is implemented as an implicit data transformation of type `ITransform`, for which the .
- 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 a vector of name of files `xt`, that contains the tagged observations.
- The `CrfAdapter` class that interfaces with the IITB CRF library.
- The `CrfTagger` class that extracts features from the tagged files.

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 UML diagram omits the utility traits and classes such as `Monitor` or Apache commons math components.

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 SourceForge.net (<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

Configuring the CRF classifier

Let's look at the class `Crf` that implement the conditional random fields classifier. The class `Crf` is defined as a data transformation of type `ITransform`, described in the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#) (line 2):

```
class Crf(  
    nLabels: Int,  
    config: CrfConfig,  
    delims: CrfSeqDelimiter,  
    xt: Vector[String])//1  
extends ITransform[String, Double] with Monitor[Int] //2  
  
type V = Double //3  
val tagsGen = new CrfTagger(nLabels) //4  
val crf = CrfAdapter(nLabels, tagsGen, config.params) //5  
val model: Option[CrfModel] = train //6  
  
def weights: Option[Array[Double]] = model.map(_.weights)  
override def >: PartialFunction[String, Try[V]] //7  
}
```

The constructor for `Crf` has 4 arguments (line 1):

- `nLabels`: Number of labels used for the classification
- `config`: Configuration parameters used to train the `Crf`
- `delims`: Delimiters used in raw and tagged files
- `xt`: Vector of name of files that contains raw and tagged data

Note

File names for raw and tagged data:

For the sake of simplicity, the files for the raw observations and the tagged observations have the same name with different extensions: `filename.raw` and `filename.tagged`

The `Monitor` trait is used to collect profiling information during training (refer to the *Monitor* section under *Utility classes* in the *Appendix*).

The type `v` of the output of the predictor `|>` is defined as `Double` (line 3).

The execution of the CRF algorithm is controlled by a wide variety of configuration parameters encapsulated in the configuration class, `CrfConfig`:

```
class CrfConfig(w0: Double,
                 maxIters: Int,
                 lambda: Double,
                 eps: Double) extends Config { //8
  val params = s"""initValue $w0 maxIters $maxIters
    | lambda $lambda scale true eps $eps""".stripMargin
}
```

For the sake of simplicity, we use the default configuration parameters, `CrfConfig`, to control the execution of the learning algorithm, with the exception of the following four variables (line 8):

- Initialization of the weight, `w0`, 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 `lambdab` for the L2 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)

Note

L₂ regularization:

This implementation of the conditional random fields support the L2 regularization as described in the section *Regularization* of [Chapter 6, Regression and Regularization](#). The regularization is turned off by setting $\lambda=0$.

The case class `CrfSeqDelimiter` specifies the following regular expressions:

- `obsDelim` to parse each observation in the raw files
- `labelsDelim` to parse each labeled record in the tagged files
- `seqDelim` to extract records from raw and tagged files

```
case class CrfSeqDelimiter(obsDelim: String,
                           labelsDelim: String,
                           seqDelim: String)
```

The instance `DEFAULT_SEQ_DELIMITER`, is the default sequence delimiter used in this implementation:

```
val DEFAULT_SEQ_DELIMITER =
  new CrfSeqDelimiter("\t/ -() :.;'?#`&_", "//", "\n")
```

The tag or label generator, `CrfTagger`, iterates through the tagged file and applies the relevant regular expressions of `CrfSeqDelimiter` to extract the symbols used in training (line 4).

The object `CrfAdapter`, defines the different interfaces to the IITB CRF library (line 5). The factory for CRF instances is implemented by the constructor, `apply`, as follows:

```
def apply(nLabels: Int,
```

```
    tagger: CrfTagger,  
    config: String): CRF = new CRF(nLabels, tagger, config)
```

Note

Adapter classes to HTB CRF library

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]. Each of these interfaces have to be implemented as adapter classes:

```
class CrfTagger(nLabels: Int) extends FeatureGenerator  
class CrfDataSeq(nLabels: Int, tags: Vector[String],  
                  delim: String) extends DataSequence  
class CrfSeqIter(nLabels: Int, input: String,  
                  delim: CrfSeqDelimiter) extends DataIter
```

Refer to the documented source code and Scaladocs files for the description and implementation of these adapter classes.

Training the CRF model

The objective of the training is to compute the weights, w_j , that maximize the conditional log-likelihood without the L_2 penalty function. Maximizing the log-likelihood function 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.

M5: Conditional log-likelihood for a linear chain CRF training set $D = \{x_i, y_i\}$ is given as follows:

$$\mathcal{L}(w, D) = -\sum_{i=0}^{n-1} \log p(y_i | x_i, w)$$

M6: Maximization of loss function and L_2 penalty is given as follows:

$$w^* = \arg \max_{\lambda} \left[\mathcal{L}(w, D) + \lambda \|w\|^2 \right] \quad \lambda = \frac{1}{2\sigma^2}$$

The training file consists of a pair of files:

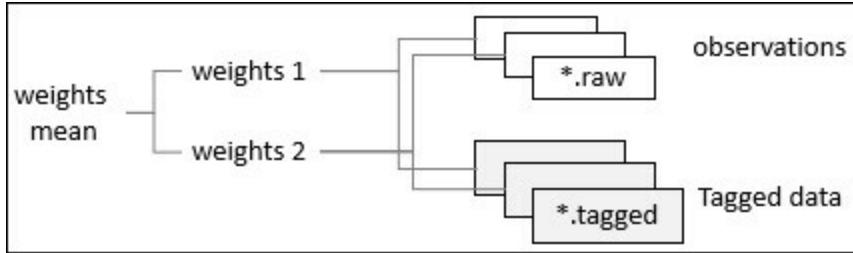
- **Raw dataset:** Recommendations (such as *Raymond James* upgrades Gentiva Health Services from underperform to market perform)
- **Tagged dataset:** tagged recommendations (such as *Raymond James* [1] upgrades [2] Gentiva Health Services [3], from [4] underperform [5] to [6] market perform [7])

Tip

Tags type:

In this implementation, the tags have the type `Int`. However, alternative types such as enumerator or even continuous values (that is, `Double`) can be used.

The training or computation of weights can be quite expensive. It is highly recommended to distribute the observations and tagged observations' dataset across multiple files, so they can be processed concurrently:



Distribution of the computation of the weights of the CRF

The `train` method creates the model by computing the `weights` of the CRF. It is invoked by the constructor of `Crf`:

```

def train: Option[CrfModel] = Try {
    val weights = if(xt.size == 1) //9
        computeWeights(xt.head)
    else {
        val weightsSeries = xt.map( computeWeights(_) )
        statisticsDouble(weightsSeries).map(_.mean).toArray //10
    }
    new CrfModel(weights) //11
}.toOption

```

We cannot assume that there is only one tagged dataset (that is, a single pair of `*.raw`, `*.tagged` files) (line 9). The method, `computeWeights`, to computation of weights for the CRF is applied to the first dataset if there is only one pair of raw, tagged files. In the case of multiple datasets, the `train` method computes the `mean` of all the weights extracted from each tagged dataset (line 10). The `mean` of the `weights` is computed using the `statistics` method of the `XTSerie`s object introduced in the *Time Series* section of [Chapter 3, Data Pre-processing](#). The `train` method returns a `CrfModel` if successful, but none otherwise (line 11).

For efficiency purposes, the map should be parallelized using the `ParVector` class as follows:

```

val parXt = xt.par
val pool = new ForkJoinPool(nTasks)
v.tasksupport = new ForkJoinTaskSupport(pool)
parXt.map(computeWeights(_) )

```

The parallel collections are described in detail in the *Parallel collections*

section under *Scala* in [Chapter 16, Parallelism in Scala and Akka](#).

Tip

CRF weights computation:

It is assumed that input tagged files share the same list of tags or symbols, so each dataset produces the same array of weights.

The method, `computeWeights`, extracts the weights from each pair of observations and tagged observations' files. It invokes the train method of the tag generator, `CrfTagger` (line 12), to prepare, normalize, and set up the training set, and then invokes the training procedure on the IITB `CRF` class (line 13):

```
def computeWeights(tagsFile: String): DblArray = {  
    val seqIter = CrfSeqIter(nLabels, tagsFile, delims)  
    tagsGen.train(seqIter) //12  
    crf.train(seqIter) //13  
}
```

Note

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.

Applying the CRF model

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:

```
override def |> : PartialFunction[String, Try[Double]] = {
```

```

case obs: String if(obs.nonEmpty && model.isDefined) => {
  val dataSeq = new CrfDataSeq(nLabels, obs, delims.obsDelim)
  Try (crf.apply(dataSeq)) //14
}
}

```

The method `|>` merely creates a data sequence, `dataSeq`, and invokes the constructor of the IITB `CRF` class (line 14). The condition on the input argument, `obs`, to the partial function is rather rudimentary. A more elaborate condition of the observation should be implemented using a regular expression. The code to validate the arguments/parameters of the class and methods is omitted along with the exception handler for the sake of readability.

Note

Advanced CRF configuration

The CRF model of the iitb library is highly configurable. It allows developers to specify a state-label undirected graph with any combination of flat and nested dependencies between states. The source code includes several training algorithms such as the exponential gradient.

Tests

The client code to execute the test consists of defining the number of labels, `NLABELS` (that is the number of tags for recommendation), the L_2 penalty factor, `LAMBDA`, the maximum of iterations, `MAX_ITERS`, allowed in the minimization of the loss function, and the convergence criteria, `EPS`:

```
val LAMBDA = 0.5
val NLABELS = 9
val MAX_ITERS = 100
val W0 = 0.7
val EPS = 1e-3
val PATH = "resources/data/supervised/crf/rating"
val OBS_DELIM = ",\t/ -():.;'?#`&_"

val config = CrfConfig(W0, MAX_ITERS, LAMBDA, EPS) //15
val delims = CrfSeqDelimiter(DELIM,"//","\n") //16
val crf = Crf(NLABELS, config, delims, PATH) //17
crf.weights.map( display(_) )
```

The following three simple steps are:

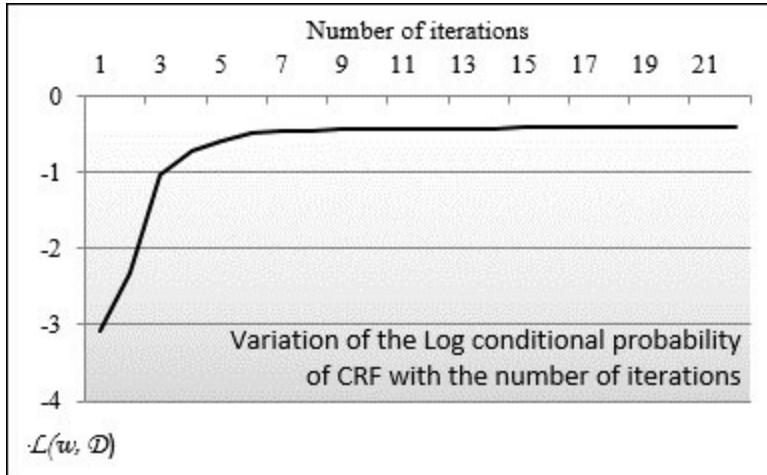
- Instantiate the configuration `config` for the CRF (line 15)
- Define the three delimiters, `delims`, to extract tagged data (line 16)
- Instantiated and train the CRF classifier, `crf` (line 17)

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 analysts' stock recommendations. The algorithm converged after 21 iterations. The value of

the log of the likelihood for each of those iterations is plotted to illustrate the convergence toward a solution of optimum w :



Visualization of the log conditional probability of CRF during training

The training phase converges quickly toward a solution. It can be explained by the fact that there is little variation in the six-field format of the analysts' 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

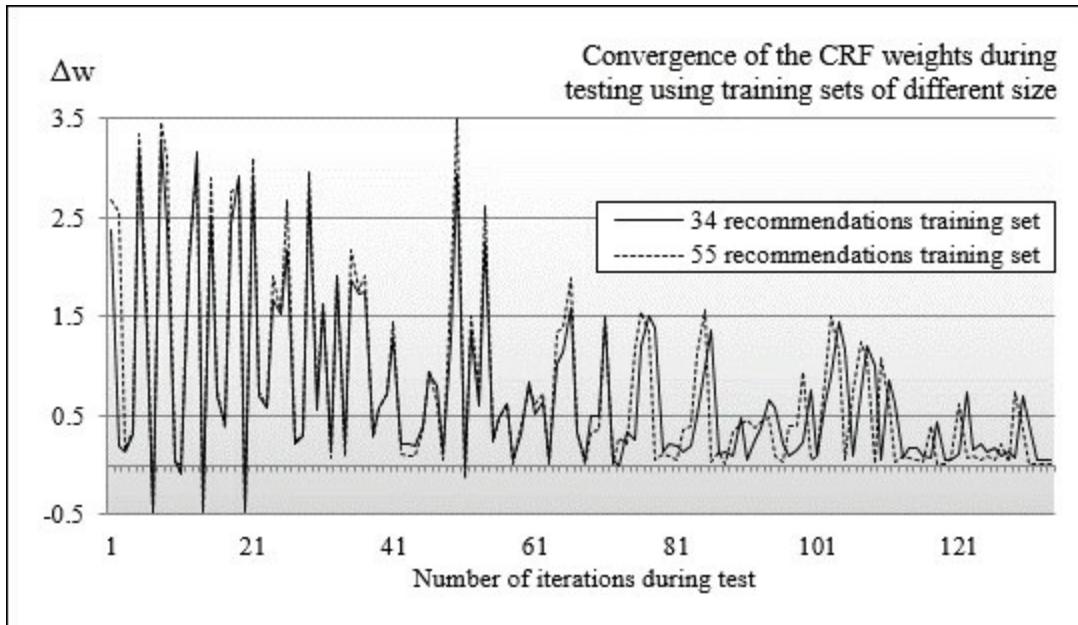
The second test evaluates the impact of the size of the training set on the convergence of the training algorithm. It consists of computing the difference, Δw , of the model parameters (weights) between two consecutive iterations, $\{w_i\}_{t+1}$ and $\{w_i\}_t$:

$$\Delta w = \sum_{i=0}^{D-1} (w_i^{t+1} - w_i^t)$$

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:

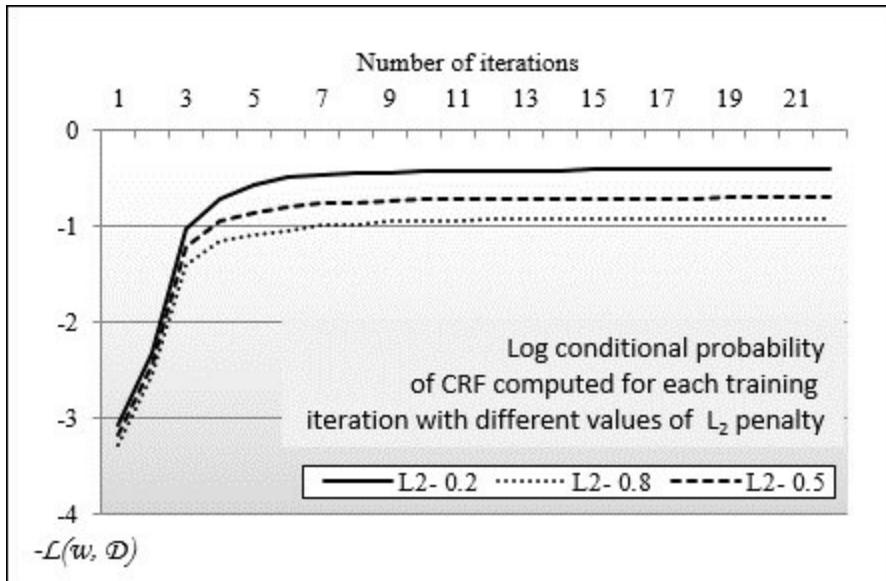


Convergence of CRF weight using training sets of difference size

The disparity between the test runs using two different sizes 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 L₂ regularization factor

The third test evaluates the impact of the L₂ 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 L_2 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 L_2 regularization factor, the higher the conditional probability.

The variation of the analysts' recommendations within the training set is small, which limits the risk of overfitting. A free-style recommendation format would have been more sensitive to overfitting.

Comparing CRF and HMM

A complete comparison of CRF and HMM models is beyond the scope of this book. However, there are some obvious differences due to the simple fact that HMM is a generative model and CRF is a discriminative model.

Contrary to the hidden Markov model, the conditional random field does not require the observations to be independent beside the time and order dependency. The conditional random field can be regarded as a generalization of the HMM: It extends the transition probabilities to arbitrary feature functions that can depend on the input sequence. You need to remember that HMM assumes the transition probabilities matrix to be constant.

HMM learns the transition probabilities, a_{ij} , on its own by training on an increasing amount of input data. The HMM is a special case of CRF where the probabilities used in the state transition are constant.

Performance consideration

Data scientists should consider the computation cost during the evaluation of a sequential data model. The time complexity for decoding and evaluating canonical forms of the hidden Markov model for N states and T observations is $O(N^2 T)$. The training of HMM using the Baum-Welch algorithm is $O(N^2 TM)$, where M is the number of iterations.

There are several options to improve the performance of HMM:

- Avoid unnecessary multiplication by 0 in the emission probabilities matrix by either using sparse matrices or tracking the null entries.
- Train HMM on the most *relevant* subset of the training data. This technique can be particularly effective in the case of tagging of words and bag of words in natural language processing.

The training of the linear chain conditional random fields is implemented using the same dynamic programming techniques as HMM implementation (Viterbi, forward-backward passes). Its time complexity for training T data sequence, N labels (or expected outcome), and M weights/features λ is $O(MTN^2)$.

The time complexity of the training of a CRF can be reduced by distributing the computation of the log likelihood and gradient over multiple nodes using a framework such as Akka or Apache Spark described [Chapter 17, Apache Spark MLlib \[7:13\]](#).

Summary

The study of the combination of two concepts: Markov processes and latent variables or states can be overwhelming at times. The implementation of the hidden Markov model, for instance, is particularly challenging for engineers with limited exposure to dynamic programming techniques.

In this chapter, you learned about the Markov processes, the generative HMM to maximize the disjoint probability, $p(X, Y)$, and the discriminative CRF to maximize log of the condition probability, $p(Y|X)$.

Markov decision processes are conceptually also used in reinforcement learning; see: [Chapter 15, Reinforcement Learning](#).

HMM is a special form of Bayes Network: It requires the observations to be independent. Although restrictive, the conditional independence pre-requisite makes the HMM easy to understand and validate, which is not the case for CRF. As a side note, recurrent neural networks are an alternative to HMM for predicting state given a sequence of observations.

The conditional random fields estimate the optimal weights of the model. Such techniques are also used in the multiple layer perceptron, introduced in [Chapter 10, Multilayer Perceptron](#).

The next chapter describes parametric and non-parametric sampling methods that leverage bootstrapping and Monte Carlo simulators.

Chapter 8. Monte Carlo Inference

One of the key challenges in supervised learning is the generation or extraction of an appropriate training set. Despite the effort and best intentions of the data scientist, the labeled data is not directly usable.

Let's take, for example, the problem of predicting the click through rate for an online display. 95-99% of data is labeled with a no-click event (negative classification class) while 1-5% of events are labeled as clicked (positive class). The unbalanced training set may produce an erroneous model unless the negatively-labeled events are reduced through sampling.

This chapter deals with the need, role, and some common methods of sampling a dataset. It covers the following topics:

- Generation of random samples from a given distribution
- Application of Monte Carlo numerical sampling to approximation
- Bootstrapping
- Markov Chain Monte Carlo for estimating parametric distribution

Although random generators are of critical importance in statistics and machine learning, they are not covered in this book. There is a wealth of references regarding the benefits and pitfalls of various schemes for generating uniform random values [8:1].

The purpose of sampling

Sampling is the process to extract a subset of a dataset that is chosen to draw inferences about the properties of this dataset. It is not always practical to use an entire dataset for the following reasons:

- Dataset is too large
- Dataset is not available in a timely fashion
- Extraction of complex features is very computationally intensive
- A very large percentage of the training data is labeled to one of the classes which require down-sampling
- Data is a continuous signal

The most commonly-cited benefits of sampling are reduction of computation cost and latency of execution.

Note

Independent and identical distribution

It is generally assumed that the original dataset reflects an **independent and identically distributed population (i.i.d.)**.

The challenge is to devise a procedure to generate a sample that represents accurately the original dataset so that any inference derived from the sample applies equally to the original dataset.

Gaussian sampling

Gaussian sampling consists of extracting a sample from a population with a distribution that follows a Gaussian or normal distribution. This section describes a commonly used algorithm known as the Box-Muller transform to generate accurate Gaussian sampling from a uniform random generator [8:2].

Box-Muller transform

The purpose of the Box-Muller scheme is to generate a sample of normal distribution (Gaussian distribution of mean 0 and variance 1) from two independent samples following uniform random distributions. Let's consider u_1, u_2 two uniformly distributed random distribution over the interval $[0, 1]$, then the following random variables:

$$\sqrt{-2 \log(u_1)} \cdot \sin(2\pi u_2) \quad \sqrt{-2 \log(u_1)} \cdot \cos(2\pi u_2)$$

are two independent standard normal distribution variables.

The class `BoxMuller` implements the Box-Muller transform. The class takes two arguments; a function `r` that generates uniformly distributed random values over $[0, 1]$ (line 1) and a flag, `cosine`, that selects either the cosine or sine function for the normal sample values (line 2):

```
class BoxMuller(
    r: () => Double = () => Random.nextDouble,    //1
    cosine: Boolean = true           //2
) {
    private def uniform2PI: Double = 2.0*PI*r()

    def nextDouble
    : Double = {
        val theta = uniform2PI
        val x = -2.0*log(r())
        sqrt(x) * (if(cosine) cos(theta) else sin(theta))
    }
}
```

The algorithm is easily evaluated with the following test code:

```
val bm = new BoxMuller()
val input = Array.fill(100000)(bm.nextDouble)
val mean = input.reduce(_ + _)/input.length
val stdDev = input./:(0.0) (
    (s, x) => s + (x-mean)*(x-mean))/input.length
```

Note

Alternative sampling methods

There are many basic sampling methods besides Box-Muller, such as rejection sampling, adaptive rejection sampling, importance sampling, and resampling. Each technique addresses a specific set of constraints or complexity of the multi-variate distribution to sample from.

Monte Carlo approximation

Monte Carlo experiments or sampling leverages randomness to solve mathematical or even deterministic problems [8:3]. There are three categories of problems:

- Sampling from a given or empirical probability distribution
- Optimization
- Numerical approximation

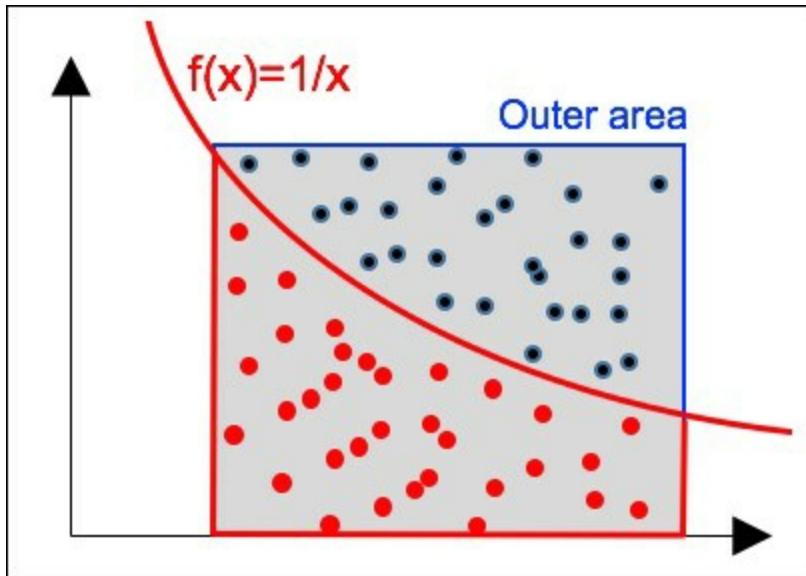
This section focuses on the numerical integration.

Overview

Let's apply the Monte Carlo simulation to numerical integration. the goal is to compute the area under a given single variable function [8:4]. The method consists of the following three-step process:

1. Define the outer area that is defined by the x axis and the maximum value of the function over the integration interval.
2. Generate a uniformly random distributed data point $\{x, y\}$ over the outer area.
3. Count, then compute, the ratio of the number of data points under the function over the total number of random points.

The following diagram illustrates the three-step numerical integration for the function $1/x$:



Monte Carlo numerical integration

Implementation

The `MonteCarloApproximation` class implements the numerical probabilistic integration of a function `f` with `numPoints` number of data points (line 1):

```
class MonteCarloApproximation(
    f: Double => Double,
    numPoints: Int //1
) {
    def sum(from: Double, to: Double): Double = { //2
        val (min, max) = getBounds(from, to) //3
        val width = to - from
        val height = if (min >= 0.0) max else max - min
        val outerArea = width * height //4
        val rx = new Random(System.currentTimeMillis)
        val ry = new Random(System.currentTimeMillis + 42L)

        def randomSquare: Double = { //5
            val numInsideArea = Range(0, numPoints)./:(0) (
                (s, n) => {
                    val ptx = rx.nextDouble * width + from
                    val pty = ry.nextDouble * height

                    rx.setSeed(ry.nextLong)
                    ry.setSeed(rx.nextLong)

                    s + (if (pty > 0.0 && pty < f(ptx)) 1
                        else if (pty < 0.0 && pty > f(ptx)) -1
                        else 0)
                }
            )
            numInsideArea.toDouble * outerArea / numPoints //6
        }
        randomSquare
    }

    def getBounds(from: Double, to: Double): (Double, Double)
}
```

The `sum` method implements numerical integration of the function `f` using Monte Carlo sampling (line 2). The first step consists of extracting the minimum and maximum values for the function with the `getBounds` method

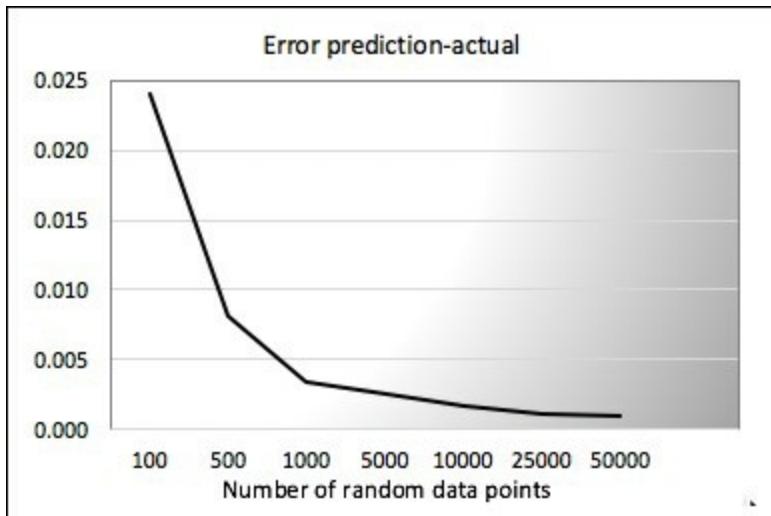
(line 3). The `getBounds` method is available in the accompanying source code. The number of random points in the overall outer area is computed with the bounds of the function `f` (line 4).

The `randomSquare` method (line 5) computes the ratio of the counts of random points within the function `f` over the total number of random points in the outer area (line 6).

The implementation of the Monte Carlo approximation is evaluated in the following test:

```
val approximator = new MonteCarloApproximation(  
    (x: Double) => 1.0/x, NumPoints  
)  
val area = approximator.sum(1.0, 2.0)
```

The test is repeated with a different number of data points to evaluate the impact of the size of the Monte Carlo simulation on the accuracy of the numerical integration, as illustrated by the following plot:



Impact of the number of data points on accuracy of Monte Carlo approximation

As expected, the error on the Monte Carlo simulation decreases as the number of random data points increases.

Bootstrapping with replacement

Bootstrapping is a Monte Carlo sampling technique to evaluate the sampling of a given distribution. This technique extracts samples from an independent and identically distributed dataset with a distribution that may not be known and represented as an empirical distribution function. The sample is created by selecting and replacing a data point, randomly chosen from the original population [8:5].

Overview

The purpose of bootstrapping is to estimate the accuracy of the resulting sampling by computing statistics characteristics such as mean, bias, standard deviation, average prediction errors, or confidence factors. As with any other sampling techniques, the resulting sample should be precise enough so that any statistical inference derived from the sample also applies to the original dataset.

One common application of bootstrapping is to estimate the empirical distribution of a statistic such as mean, standard deviation, or kurtosis. This approach is known as **resampling**.

Resampling

The methodology applies the Monte Carlo method to repeatedly resample the dataset with a replacement while making sure that the size of the resample is identical to the original dataset. The algorithm computes the target statistics (mean, variance,) and then estimates errors at each resampling episode. The iterative process exits when the target statistic reaches a given precision.

The statistics for each sample is known as a **replicate**.

Let's B be the number of bootstrap samples $x^{(i)}$ with $i = 0, B-1$.

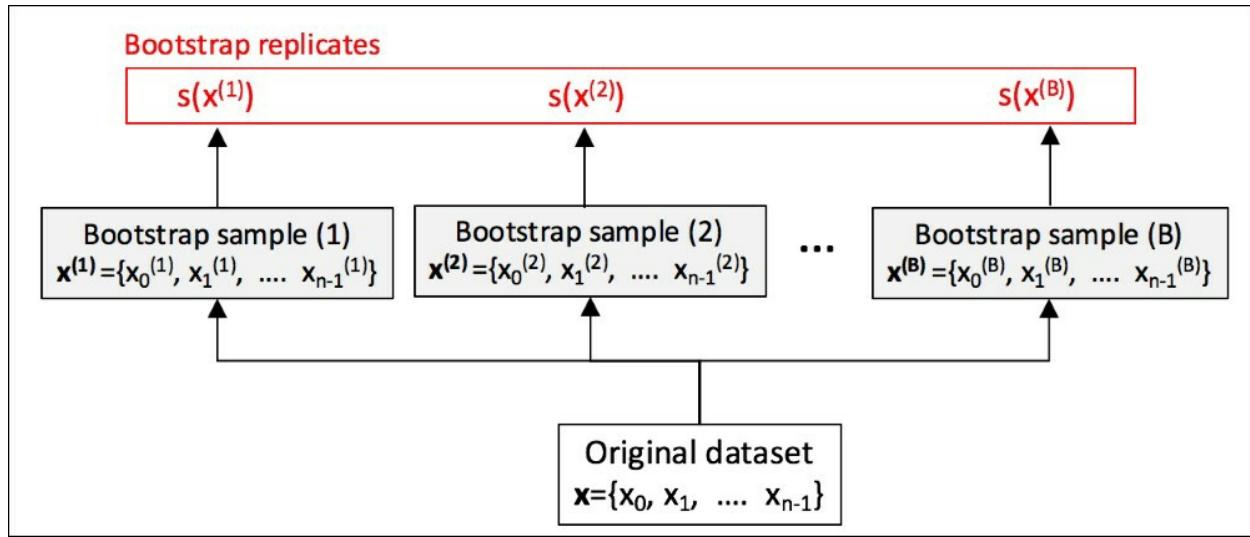
Bootstrap replication for statistics s is as follows:

$$x^{(i)} = \left\{ x_j^{(i)} \right\} \hat{\theta}^{(i)} = s(x^{(i)})$$

M1: The standard error is computed as follows:

$$\hat{s} = \sqrt{\frac{1}{B-1} \sum_{j=0}^{B-1} (\hat{\theta}_j^{(i)} - \bar{\theta})^2} \quad \bar{\theta} = \frac{1}{B} \sum_{j=0}^{B-1} \hat{\theta}_j^{(i)}$$

The resampling process is illustrated in the following diagram:



Extraction of bootstrap replicates

B samples are extracted from the original dataset through sampling. Each sample is reinserted into the original dataset before the next sampling. The size of the samples is identical to the size of the original set. The statistics s is computed on each sample.

Implementation

The `Bootstrap` class implements the bootstrap with a replacement algorithm. The class has the following four arguments:

- Number of replicates or samples, `numSamples` (line 1)
- Target statistics s (line 2)
- Original input dataset, `input` (line 3)
- Random integer generator over interval 0, size dataset, `randomizer` (line 4)

Let's review at the implementation of the key components of the `Bootstrap` class:

```
class Bootstrap (
    numSamples: Int,      //1
    s: Vector[Double] => Double,   //2
    input: Vector[Double],   //3
    randomizer: Int => Int     //4
) {

    lazy val replicates: Array[Double] =    //6
        (0 until numSamples)./:(ArrayBuffer[Double]())(
            (buf, _) => buf += createBootstrapSample
        ).toArray

    def createBootstrapSample: Double = s(      //5
        (0 until input.size)./:(ArrayBuffer[Double]())(
            (buf, _) => {
                val randomValueIndex = randomizer(input.size)
                buf += input(randomValueIndex)
            }
        ).toVector
    )

    lazy val mean = replicates.reduce(_ + _) / numSamples
    def error: Double = { //7
        val sumOfSquaredDiff = replicates.reduce(
            (s1: Double, s2: Double) =>
                (s1 - mean)*(s1 - mean) + (s2 - mean)*(s2 - mean)
        )
    }
}
```

```

        sqrt(sumOfSquaredDiff / (numSamples-1))
    }
}

```

The replicates are computed by the method `createBootstrapSample` (line 5). The method selects random samples with a number that equals the size of the original dataset, then applies the target statistics `s`. The replicates are stored in the lazy value `replicates` (line 6). The `error` method (line 7) implements the formula M1.

Note

Formula for the standard error

The value of the standard error in our implementation is scaled by a factor of $1/(B-1)$ with B being the number of bootstrap samples. Some authors use $1/B$ as an alternative scaling factor.

The `bootstrapEvaluation` method evaluates the `Bootstrap` class for the mean. The original dataset is generated through a continuous probability density function, `dist`, implemented in Apache Commons Math (line 8):

```

def bootstrapEvaluation(
    dist: RealDistribution,    //8
    random: Random,
    linear: (Double, Double)  //9
): (Double, Double) = {

    val input = (0 until m)./:(ArrayBuffer[(Double, Double)]())(
        (buf, _) => {
            val (a, b) = linear
            val x = a * nextDouble - b //10
            buf += ( (x, dist.density(x)) )
        }
    ).toVector

    val bootstrap = new Bootstrap(
        numReplicates,
        (x: Vector[Double]) => x.sum/x.length,      //11
        input.map(_._2),     //12
        (n: Int) => new Random(42L).nextInt(n)       //13
    )
}

```

```
    (bootstrap.mean, bootstrap.error)
}
```

The input is generated through a linear probability distribution (line 9). $y = a.random + b$ (line 10). The bootstrap is instantiated with the following arguments:

- Number of replicates, `numReplicates`
- The target statistics to evaluate; in this case, `mean` (line 11)
- The probability density function (line 12)
- A uniform pseudo-random number generator over a given interval (line 13).

Let's evaluate the impact of the number of replicates on the standard error as illustrated in the following plot:



Distribution of the standard error on bootstrap replicates

As expected, the standard error decreases as the number of replicates increases. However, the graph shows that there is little benefit to increasing the number of replicates beyond 1028.

Tip

Optimized number of bootstrap samples

The optimum number of bootstrap samples to generate for accurate results depends partially on the type of noise in the original dataset. As a rule of thumb, a range of 500 to 2,000 for the number of samples produces the most accurate replicates.

Pros and cons of bootstrap

Clearly, the main advantage of bootstrap sampling is its simplicity and the ability for the analyst to improve accuracy by tuning the number of samples.

The main limitation of this technique is the stringent requirement that the samples are independent.

Markov Chain Monte Carlo (MCMC)

As we have seen in *The Markov property* section of [Chapter 7, Sequential Data Models](#), the state or prediction in a sequence is a function of the previous state(s). In the first order, Markov processes the probability of a state at time t depending on the probability of the state at time $t-1$.

The concept of a Markov chain can be extended with the traditional Monte Carlo sampling to model distributions with a large number of variables (high dimension) or parametric distributions.

Overview

The idea behind the **Markov Chain Monte Carlo** inference or sampling is to randomly walk along the chain from a given state and successively select (randomly) the next state from the state-transition probability matrix (*The Hidden Markov Model/Notation* in [Chapter 7, Sequential Data Models](#)) [8:6].

This iterative process explores the distribution from the transition probability matrix if it matches the target distribution also known as the **proposal distribution**. At each iteration, MCMC selects a sample from the proposal distribution. The sample is accepted or rejected to a specific criterion.

The Metropolis-Hastings algorithm, described in the next section, is the most versatile and commonly applied MCMC method.

Metropolis-Hastings (MH)

As discussed earlier, MCMC transition from one state, s , to another state, s' , through a proposal distribution q , selected by the data scientist with a transition probability of $q(s, s')$. One simple approach is to model the probability q as a Gaussian distribution of mean s/s' [8:7]

The Metropolis-Hastings algorithm accepts or rejects the proposal distribution q according to the condition that the time spent on new state, s' , is proportional to the probability density $p^*(s')$ of the target distribution p^* .

Following are the Metropolis-Hastings steps:

1. From state s_t sample s' with $q(s'|s_t)$.
2. Compute the acceptance criterion as (M3):

$$\alpha = \frac{\hat{p}(s') q(s^t | s')}{\hat{p}(s^t) q(s' | s^t)}$$

3. Generate a uniformly random value u over $[0, 1]$.
4. Apply accept/reject the criteria (M4):

$$s^{t+1} = \begin{cases} s' & \text{if } u < \min(1, \alpha) \\ s^t & \text{if } u \geq \min(1, \alpha) \end{cases}$$

The Metropolis-Hastings algorithm can be simplified by assuming the proposal distribution q is symmetric. In this case, the acceptance ratio is computed simply as the ratio of $p(s')/p(s_t)$. The method is then referred to as the Metropolis algorithm.

Implementation

The computation of acceptance/rejection probability α relies on the product and quotient of probability and may generate overflow or underflow values. Therefore, it is more convenient to use the logarithm expressions:

Note

Log MH

$$\log \alpha = \log(\hat{p}(s')) - \log(\hat{p}(s^t)) + \log(q(s^t | s')) - \log(q(s' | s^t))$$
$$s^{t+1} = \begin{cases} s' & \text{if } u < e^\alpha \\ s^t & \text{if } u \geq e^\alpha \end{cases}$$

The first step is to keep track of the path or history of the execution of the random walk of the MH algorithm. The `Trace` class maintains the state of the execution with a sequence of states, `profile` (line 1), and the number of accepted tests, `accepted` (line 2):

```
type State = Vector[Double]

class Trace {
    val profile = ArrayBuffer[State]() //1
    var accepted: Double = -1.0 //2

    def +=(s: State): Unit = profile.append(s) //3
    def +=(): Unit = accepted += 1 //4

    def rate(iters: Int): Double = accepted.toDouble/iters
}
```

The two methods `+=` update the profile or history of the execution of MCMC (line 3) and the number of accepted tests (line 4).

The `MetropolisHastings` class implements the Metropolis-Hastings

algorithm for multi-variate distribution for which observations have the type `vector[Double]`. The class is instantiated with the following arguments:

- The target distribution `p` (line 5)
- The proposal (transition) probability `q` (line 6)
- The state transition `proposer` function (line 7)
- The pseudo-random generator, `rand`

The implementation of the class `MetropolisHastings` and its random walk method, `mcmc` follow:

```
class MetropolisHastings(
    p: State => Double,           //5
    q: (State, State) => Double,   //6
    proposer: State => State,     //7
    rand: () => Double
) {

    def mcmc(initial: State, numIters: Int): Trace = { //8
        var s: State = initial
        var ps = p(initial) //9
        val trace = new Trace //10

        (0 until numIters).foreach(iter => {
            val sPrime = proposer(s) //11

            val psPrime = p(sPrime) //12
            val logAlpha = psPrime - ps + q(sPrime, s) - q(s, sPrime) //13

            if (logAlpha >= eps || rand() < exp(logAlpha)) { //14
                trace.+=(())
                s = sPrime
                ps = psPrime
            }
            trace.+=(s) //15
        })
        trace
    }
}
```

The algorithm is executed by invoking the `mcmc` method with the initial state (line 8). The probability `ps` on the state, `s` (features vector) is computed once the state is initialized (line 9). A `trace` is created to collect the random walk (line 10).

At each iteration, the algorithm

1. Computes the new state, `sPrime` using the proposer transition provided by the user (line 11)
2. Updates the state probability (line 12)
3. Computes the logarithm value of the acceptance/rejection probability `logAlpha` by applying the MH formula (line 13)
4. Applies the selection criteria to proceed with the random walk and collect the new state `sPrime` and its probability `psPrime` (line 14)

The original state is collected if the selection criteria is not met (line 15).

Note

Modified Acceptance criterion

The condition `logAlpha >= eps` is not actually part of the original MH algorithm. It is added during the conversion to the logarithm form to avoid numerical exceptions from being thrown.

Test

This simple test uses a univariate square distribution. Let's create a univariate MH class, `OneMetropolisHastings`, which takes the same argument as the generic class:

```
final class OneMetropolisHastings(  
    p: Double => Double,  
    q: (Double, Double) => Double,  
    proposer: Double => Double,  
    rand: () => Double  
) extends MetropolisHastings(  
    (s: State) => p(s.head),  
    (s: State, sPrime: State) => q(s.head, sPrime.head),  
    (s: State) => Vector[Double](proposer(s.head)),  
    rand  
)
```

We choose the linear probability density function for the target distribution p , over the interval $[0,1]$ (line 17), the mean value of the current and next candidate state for the proposal function q (line 18), and linear random function as a proposer (line 19).

```
val p = (x: Double) => if(x < 0.0 && x >= 1.0) 0.0 else x //17  
val rand = new Random //20  
val q = (s: Double, sPrime: Double) => 0.5*(s+sPrime) //18  
val proposer = (s: Double) => {  
    val r = rand.nextDouble  
    (if(r < 0.2 || r > 0.8) s*r else 1.0) //19  
}  
  
val trace = new OneMetropolisHastings(  
    p, q, proposer, () => rand.nextDouble  
) .mcmc(1.0, 100)
```

The randomizer, `rand` (line 20) is simply the default Scala random generator. The execution of the test produces the following results:

```
trace(97): 0.02042  
trace(98): 0.01740  
trace(99): 0.01624
```

Accepted: 85.0

Note

Gibbs sampling

The Gibbs sampling technique is an MCMC method, commonly used in Bayesian inference. It can be considered as a special case of the Metropolis-Hastings algorithm. The idea is to replace sampling from the joint probability $p(x_0, \dots, x_{n-1})$ with sampling from a conditional probability multivariate distribution $p(x_i | x_0, \dots, x_{i-1}, x_{i+1}, \dots, x_{n-1})$ for sample i . Once the n samples have been generated from each conditional probability, the process is repeated for the next state of the Markov chain [8:8].

Summary

Monte Carlo methods are important components of the data scientist's toolbox. Such techniques are used in risk analysis, decision making, and obviously pre-conditioning data for training a machine learning model.

In this chapter, you learned about generating normally distributed random values using the Box-Muller technique, applying Monte Carlo sampling to numerical integration, leveraging bootstrapped samples with replacements to construct statistics and finally extending sampling to a large dimension dataset using Markov Chain Monte Carlo.

This chapter completes the study of generative machine learning algorithms. The next part of the book introduces discriminative models, starting with the ubiquitous linear and logistic regression classifier.

Chapter 9. Regression and Regularization

We selected binary logistic regression to introduce the basics of machine learning in the *Kicking the tires* section of [Chapter 1, Getting Started](#). The purpose was to illustrate the concept of discriminative classification. It is important to keep in mind that some regression algorithms, such as logistic regression, are classification models.

The variety and the number of regression models go well beyond the ubiquitous ordinary least square linear regression and logistic regression [9:1]. Have you heard of isotonic regression?

The purpose of regression is to minimize a loss function, the **residual sum of squares (RSS)** being one that is commonly used. The *Accessing a model* section in [Chapter 2, Data Pipelines](#), introduced the thorny challenge of overfitting, which will be partially addressed in this chapter by adding a **penalty term** to the loss function. The penalty term is an element of the larger concept of **regularization**.

The chapter starts with a description of the linear **least squares regression**. The second section introduces the concept of regularization with an implementation of **ridge regression**. Finally, logistic regression will be revisited in detail, then applied as a classification model.

Linear regression

Although simplistic, linear regression should have a prominent place in your machine learning toolbox. The term regression is usually associated with the concept of fitting a model to data and minimizing the error between the expected and predicted values by computing the sum of square errors, residual sum of square errors, or least square errors.

Least square problems fall into two broad categories:

- Ordinary least squares
- Non-linear least squares

Univariate linear regression

Let's start with the simplest form of linear regression, which is single variable regression, in order to introduce the terms and concepts behind linear regression. In its simplest interpretation, one variate linear regression consists of fitting a line to a set of data points $\{x, y\}$.

Note

M1: This is a single variable linear regression for a model f , with weights w_j for features x_j , and labels (or expected values) y_j :

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

Here, w_1 is the slope, w_0 is the intercept, f is the linear function that minimizes the RSS, and (x_j, y_j) is the set of n observations.

The RSS is also known as the **sum of square errors (SSE)**. The **mean squared error (MSE)** for n observations is derived from the SSE and computed as the ratio of RSS/n .

Note

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 start with the implementation of formula M1 and create the class `SingleLinearRegression`. The linear regression is a data transformation that uses a model implicitly built from input data. Therefore, the simple linear regression.

The `SingleLinearRegression` class takes two arguments: implements the `ITransform` trait (*line 1*) as described in the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#).

- An `xt` vector of single variable observations of type `T`
- A vector of expected values or labels (*line 1*).

Let's implement our first and simplest regression model:

```
class SingleLinearRegression[T: ToDouble] (
    xt: Vector[T],
    expected: Vector[T])
  extends ITransform[T, Double] with Monitor[Double] { //1
  type DblPair = (Double, Double)

  val model: Option[DblPair] = train //3
  def train: Option[DblPair] = { ... }
  override def |> : PartialFunction[T, Try[Double]] //2
}
```

The `Monitor` trait is used to collect profiling information during training (refer to the *Monitor* section under *Utility classes* in the *Appendix*).

The class has to define the type of the output of the prediction method, `|>`, that is a `Double` (*line 2*).

Tip

Model instantiation

The model parameters are computed through training and the model is instantiated regardless whether the model is actually validated. A commercial

application requires the model to be validated using a methodology such as K-fold validation as described in the *Design template for classifiers* section of *Appendix*.

The training generates the model defined as the regression weights, in this case the slope and intercept (*line 3*). The model is set as `None` if an exception is thrown during training:

```
def train: Option[DblPair] = {
    val regr = new SimpleRegression(true) //4
    regr.addData(zipToSeries(xt, expected).toArray) //5
    Some((regr.getSlope, regr.getIntercept)) //6
}
```

The regression weights or coefficients pairs, `model`, are computed using the `SimpleRegression` class from the `stats.regression` package of Apache Commons Math library, along with the `true` argument to trigger the computation of the intercept (*line 4*). The input time series and the labels (or expected values) are zipped to generate an array of two values (input, expected) (*line 5*). The `model` is initialized with the slope and intercept computed during the training (*line 6*).

The `zipToSeries` method converts a pair of vectors of type `Vector[T]` into a vector of type `Vector[Array[Double]]`. The method is defined in the `TSeries` object described in the *Time series in Scala* section of [Chapter 3, Data Pre-processing](#).

Tip

Private versus `private[this]`

A private value or variable can be accessed by all the instances of a class. A value declared `private[this]` can be manipulated only by `this` instance. For instance, the value `model` can be accessed only by this instance of the `SingleLinearRegression`.

Test case

For our first test case, we compute the single variate linear regression of the price of the copper ETF (ticker symbol: CU) over a period of six months (January 1, 2013 to June 30, 2013):

```

for {
  path <- getPath(s"supervised/regression/CU.csv")
  src <- DataSource(path, false, true, 1)
  price <- src get adjClose //7
  days <- Try(Vector.tabulate(price.size) (_.toDouble)) //8
  linRegr <- SingleLinearRegression[Double](days, price) //9
} yield {
  if( linRegr.isModel )
    linRegr.model match {
      case Some(m) =>
        val (slope, intercept) = m
        val error = mse(days, price, slope, intercept)//10
      case None =>
        ...
    }
}

```

The daily closing `price` for the ETF, CU is extracted from a CSV file (*line 7*) as the expected values using a `DataSource` instance described in the *Data extraction and Data sources* section under *Source consideration* in the *Appendix*. The x-values `days` are automatically generated as a linear function (*line 8*). The expected values (`price`) and the sessions (`days`) are the input to the instantiation of the simple linear regression (*line 9*).

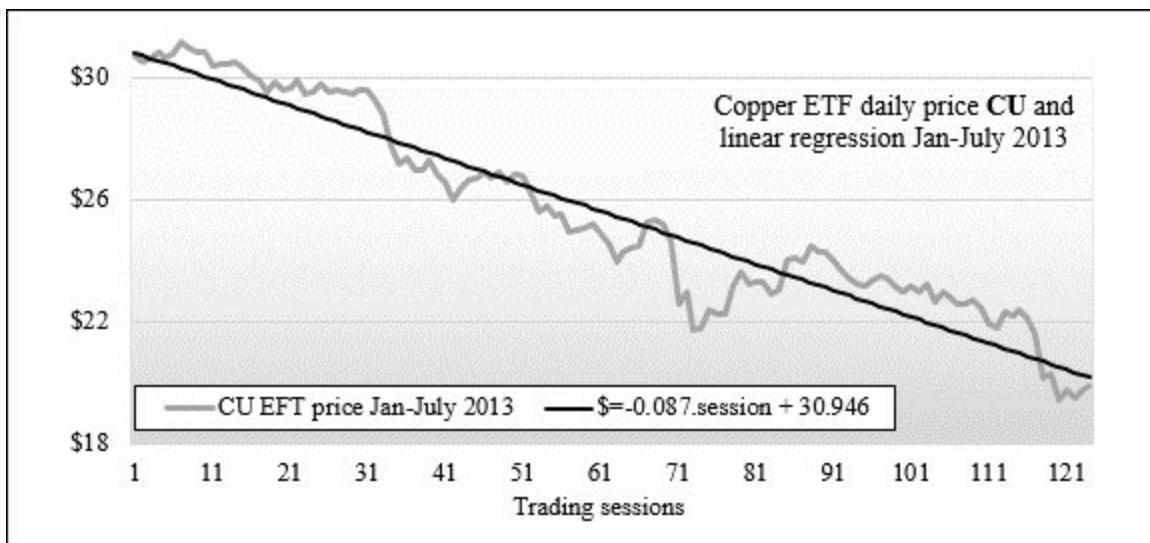
Once the model is created successfully, the test code computes the mean squared error, `mse`, between the predicted and expected values (*line 10*):

```

def mse(
  predicted: DblVec,
  expected: DblVec,
  slope: Double,
  intercept: Double): Double = {
  val predicted = xt.map( slope*_ + intercept)
  Loss.mse(predicted, expected) //11
}

```

The mean least squared error is computed using the `mse` method of `Loss` singleton (*line 11*). The original stock price and the linear regression equation are plotted in the following chart:



Simple linear regression of CU ETF daily price

The linear model, $y = -0.087x + 30.947$ is accurate with a mean least square error (mse) 0.0832.

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

OLS is a generalization of the simple linear regression described in the previous section. It applies to problems with more than one feature or variable. It computes the parameters, w , of a linear *function* $y = f(x_0, x_1, \dots, x_d)$ by minimizing the residual sum of squares. The optimization problem is solved by performing vector and matrix operations (transposition, inversion, and substitution).

M2: Given the weights w_j , the n observations $(x_i, y_i)_{i:0,n-1}$ from the vector x and the expected output values y and the linear multivariate *function* $y = f(x_0, x_1, \dots, x_d, \dots)$, the minimization of loss function is computed by the following formula:

$$\tilde{w} = \arg \min_{w,r} \left\{ \sum_{j=0}^{N-1} (y_j - f(x_j | w))^2 \right\} \quad f(x | w) = w_0 + \sum_1^{D-1} w_d x_d$$

They are several methodologies to minimize the RSS for a linear regression:

- Resolution of the set of n equations with d variables (weights) using **QR decomposition** of the n by d matrix representing the time series of n observations of a vector of d dimensions with $n \geq d$ [9:2]
- **Singular value decomposition** on the observations-features matrix in the case the dimension d exceeds the number of observations n [9:3]
- Minimization of loss function using the batch gradient descent [9:4]
- Minimization of loss function using the stochastic gradient descent [9:5]

An overview of these matrix decompositions and optimization techniques can be found in the *Elements of linear algebra* and *Summary of optimization techniques* sections of the *Appendix*.

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

We select the linear algebra classes and methods defined in the Apache Commons Math library to implement our ordinary least squares regression [9:6].

This chapter describes several types of regression algorithm. It makes sense to define a generic trait regression that defines the key element component of a regression algorithm:

- A model of type `RegressionModel` (*line 1*)
- Two methods to access the components of the regression model: `weights` and `rss` (*line 2 & 3*)
- A polymorphic method, `train`, that implements the training of this specific regression algorithm (*line 4*)
- A private method, `training`, that wraps `train` into a `Try` monad (*line 5*):

```
trait Regression {  
    type Obs = Array[Double]  
    val model: Option[RegressionModel] = train.toOption //1  
  
    def weights: Option[Obs] = model.map(_.weights) //2  
    def rss: Option[Double] = model.map(_.rss) //3  
    def isModel: Boolean = model != None  
    protected def train: RegressionModel //4  
}
```

The model is simply defined by its `weights` and its residual sum of squares, `rss`:

```
case class RegressionModel( //5  
    weights: Obs,  
    rss: Double) extends Model[RegressionModel]
```

Let's create a class, `MultiLinearRegression`, as a data transformation, the model of which is implicitly derived from the input data (training set) as described in the *Monadic data transformation* section of [Chapter 2, Data Pipelines](#):

```
class MultiLinearRegression[@specialized(Double) T:ToDouble] (
```

```

    xt: Vector[Array[T]],
    expected: DblVec
) extends ITransform[Array[T], Double] with Regression { //7

  override def train: Option[RegressionModel] //8
  override def |>: PartialFunction[Array[T], Try[Double]] //9
}

```

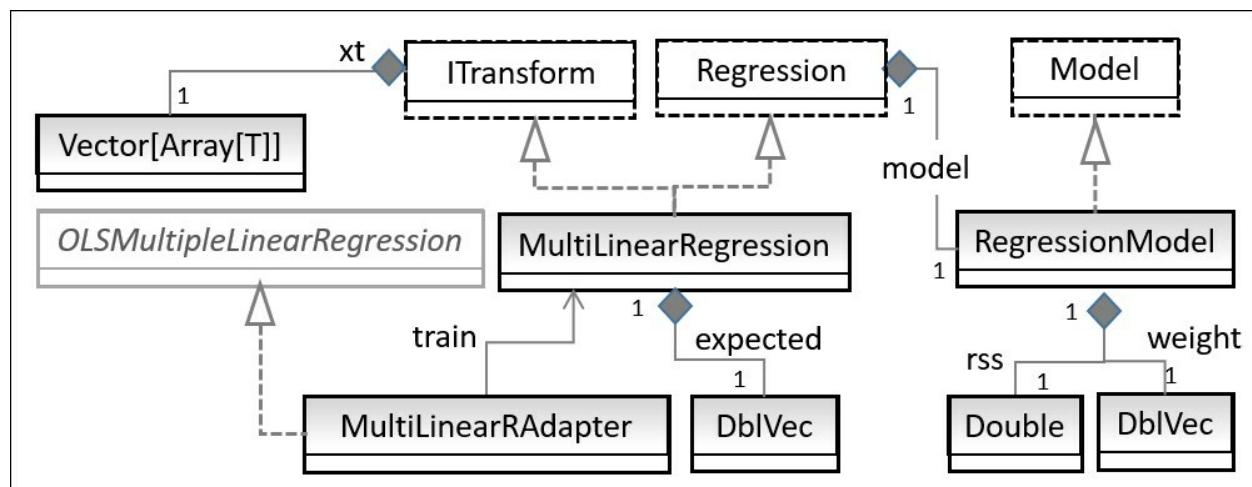
The `MultiLinearRegression` class takes two arguments; the multi-dimensional time series of observations, `xt`, and the vector of `expected` values (*line 7*). The class implements the `ITransform` trait and defines the type of the output value for the prediction or regression as a `Double`. The constructor for `MultiLinearRegression` creates the model through training (*line 8*). The `ITransform` method `|>` implements the run-time prediction for the multi-linear regression (*line 9*).

Tip

Regression model

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.

The relationship between the different components of the multi-linear regression is described in the following UML class diagram:



UML class diagram for multi-linear (OLS) regression

The UML diagram omits the helper traits and classes such as Monitor or Apache Commons Math components.

Implementation

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

```
def train: RegressionModel = {
    val mLr = new MultiLinearRAdapter      //10
    mLr.createModel(
        xt.map(_.map(implicitly[ToDouble[T]].apply(_))),
        expected
    ) //11
    RegressionModel(mLr.weights, mLr.rss) //12
}
```

The functionality of the ordinary least squares regression in the Apache Commons Math library is accessed through a reference `mLr` to the adapter class `MultiLinearRAdapter` (*line 10*).

The `train` method creates the model by invoking the `OLSMultipleLinearRegression` Apache Commons Math class (*line 11*) and returns the regression model (*line 12*). The various methods of the class are accessed through the `MultiLinearRAdapter` adapter class:

```
class MultiLinearRAdapter extends OLSMultipleLinearRegression {

    def createModel(y: DblVec, x: Vector[Obs]): Unit =
        super.newSampleData(y.toArray, x.toArray)
    def weights: Obs = estimateRegressionParameters
    def rss: Double = calculateResidualSumOfSquares
}
```

The `createModel`, `weights`, and `rss` methods route the request to the corresponding methods in `OLSMultipleLinearRegression`.

The Scala exception handling monad `Try{}` is used as the return type for the

`train` method in order to catch the different types of exception thrown by the Apache Commons Math library, such as `MathIllegalArgumentException`, `MathRuntimeException`, and `OutOfRangeException`.

Note

Exception handling

Wrapping up invocation of methods in a separate library or framework with a Scala exception handler `Try { }` matters for a couple of reasons:

It makes debugging easier by segregating your code from the third party

It allows your code to recover from the exception by re-executing the same function with an alternative third-party library methods, whenever possible

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

```
def |> : PartialFunction[Array[T], Try[Double]] = {
  case x: Array[T] if(isModel &&
    x.length == model.get.size -1 =>
    Try(margin(x, model.get) ) //13
}
```

The predictive value is computed using the `margin` method defined in the `RegressionModel` singleton introduced earlier in this section (*line 13*).

Test case 1 – trending

The objective is to identify a trend in a time series. In this context, **trending** consists of extracting the long-term movement in a time series. Trend lines are detected using a multivariate least squares regression. The objective of this first test is to evaluate the filtering capability of the ordinary least squares regression.

The regression is performed on the relative price variation of the Copper ETF

(ticker symbol: CU). The selected features are `volatility` and `volume`, and the label or target variable is the price change between two consecutive trading sessions, y .

The naming convention for the trading data and metrics is described in the *Trading data* section under *Technical analysis in the Appendix*.

The volume, volatility, and price variation for CU between January 1, 2013 and June 30, 2013 are plotted in the following chart:

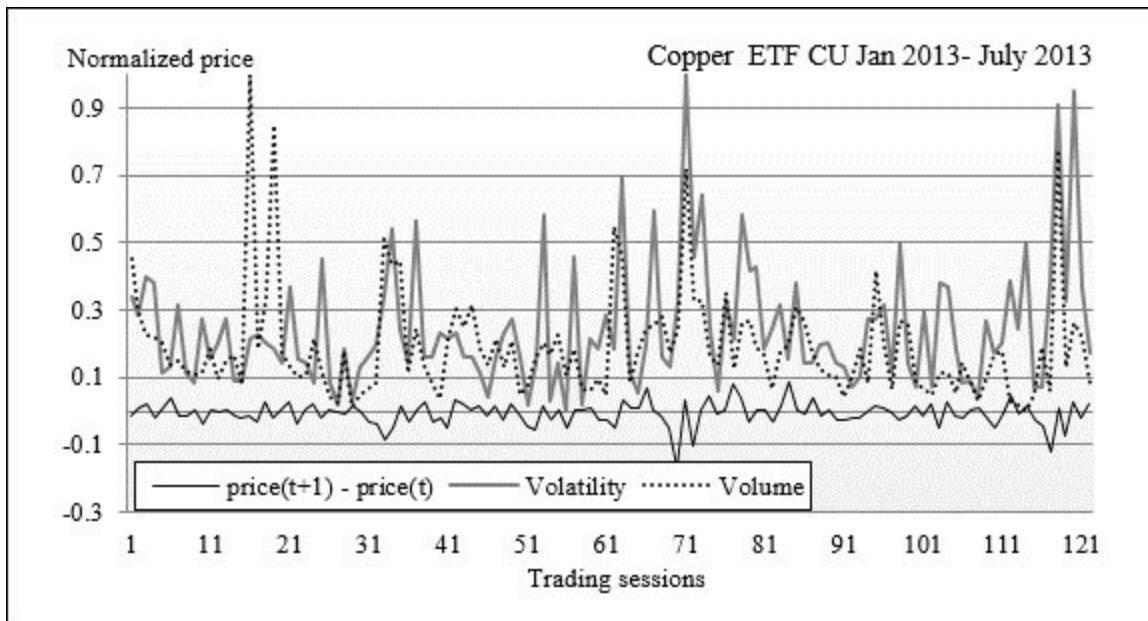


Chart for price variation, volatility, and trading volume for Copper ETF

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

```

for {
    path <- getPath(s"supervised/regression/CU.csv") //14
    src <- DataSource(path, true, true, 1) //15
    price <- src.get(adjClose) //16
    volatility <- src.get(volatility) //17
    volume <- src.get(volume) //18
    (features, expected) <- differentialData(
        volatility, volume, price, diffDouble //19
    )
    regression <- MultiLinearRegression[Double](features, expected)
} yield {}//20
  
```

```

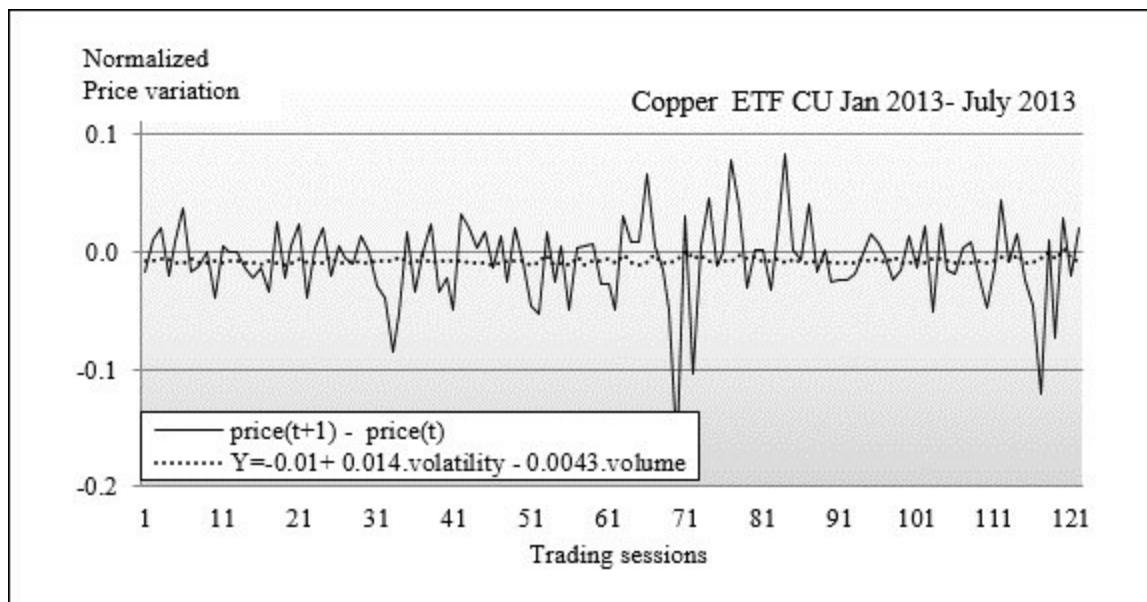
if( regression.isModel ) {
    val trend = features.map(margin(_, regression.weights.get))
    display(expected, trend) //21
}
}

```

Let's look at the steps in the execution of the test. It consists of collecting data, extracting the features and expected values, and training the multi-linear regression model:

1. Locate the CSV-formatted data source file (*line 14*).
2. Create a data source extractor, `DataSource`, for the trading session `closing price`, the session `volatility`, and session `volume` for the ETF, CU (*line 15*).
3. Extract the price of the ETF (*line 16*), its volatility within a trading session (*line 17*), and trading volume during the session (*line 18*), using the `DataSource` transform.
4. Generate the labeled data as a pair of features (relative volatility and relative volume for the ETF) and `expected` outcome $\{0, 1\}$ for training the model for which 1 represents price increase and 0 represents price decrease (*line 19*). The generic `differentialData` method of the `Stats` singleton is described in the *Times series* section of [Chapter 3, Data Pre-processing](#).
5. The multi-linear `regression` is instantiated using the `features` set and the `expected` change in daily ETF price (*line 20*).
6. Display the expected and trending values using `JfreeChart` (*line 21*).

The time series of expected value and the data predicted by the regression are plotted in the following chart:



Price variation and 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:

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

The estimated price change (dotted line in the preceding chart) represents the long-term trend for which the noise is filtering out. In other words, the least squares regression operates as a simple, low-pass filter as an alternative to some of the filtering techniques such as the discrete Fourier transform or the Kalman filter described in [Chapter 3, Data Pre-processing \[9:7\]](#).

Although trend detection is an interesting application of the least squares regression, the method has limited filtering capabilities for time series [9:8]:

- It is quite sensitive to outliers
- As a deterministic method, it does not support noise analysis (distribution, frequencies, and so on)

Test case 2 – features selection

Our objective is to discover which subset of initial features generates the most accurate regression model; that is, the model with the smallest 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 $\{x_{id}\}$ that are the most relevant to the set of observations using a least squares regression. Each subset of features is associated to the model $f_j(x|w_j)$:

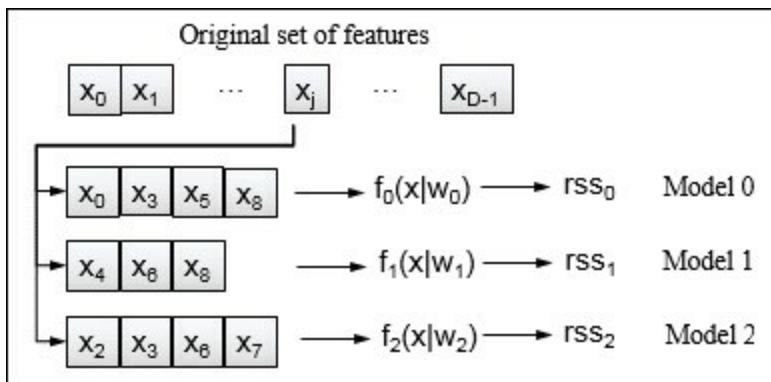


Diagram for the features set selection

The selection of the model parameters, w , uses ordinary least square regression in the case that the features set if small. Performing a regression of each of subset of a large original features set is not practical.

M3: Given the weights w_{jk} of the function/model f_j and the n observations $(x_i, y_i)_{i:0,n-1}$ and the linear multivariate function $y = f(x_0, x_1, \dots, x_d, \dots)$, the features selection can be expressed mathematically as follows:

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

Let's consider the following four financial time series over the period from January 1, 2009 to December 31, 2013:

- Exchange rate of Chinese Yuan to US Dollar
- S&P 500 index

- Spot price of gold
- 10-year treasury bond price

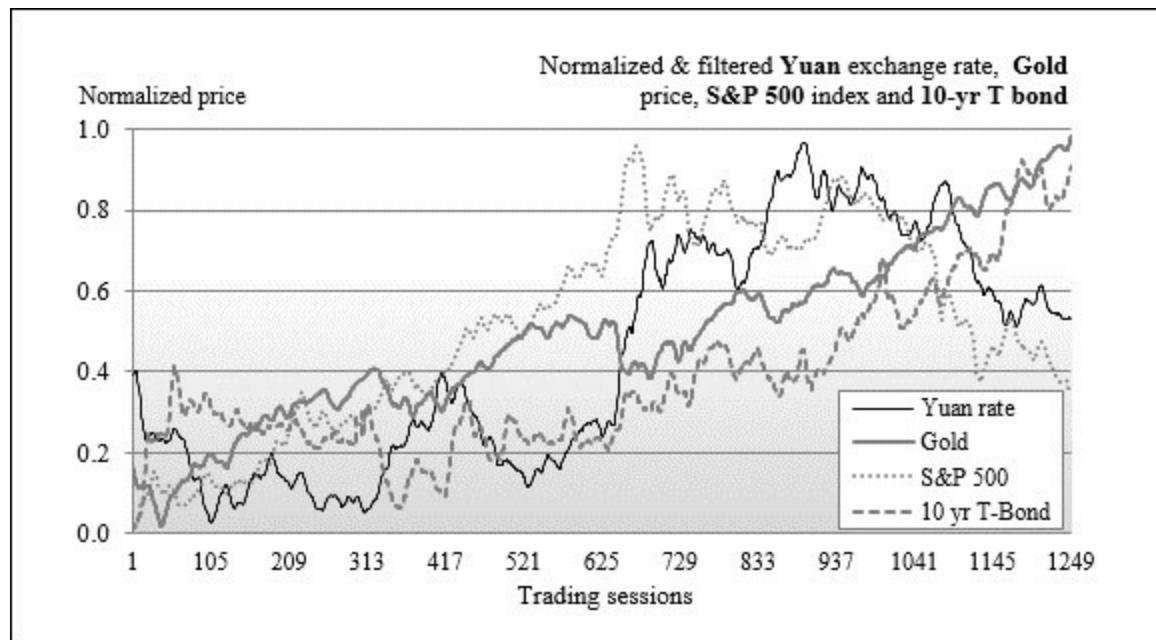
The problem is to estimate which combination of the three variables, S&P 500 index, gold price, and 10-year treasury bond price, is the most correlated to the exchange rate of Yuan. For practical reasons, we use the Exchange Trade Funds CYN as the proxy for the Yuan/US dollar exchange rate (similarly, we use SPY, GLD, and TLT for the S&P 500 index, spot price of gold, and 10-year treasury bond price, respectively).

Tip

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 (such as gradient descent or a genetic algorithm) using $1/\text{RSS}$ as the objective function.

The number of models to evaluate is relatively small, so an ad hoc approach to compute the RSS for each combination is acceptable. 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`, expected (smoothed) values, `y`, and labels for features, `featureLabels`, and returns a textual result:

```
def getRSS(
    xt: Vector[Array[Double]],
    expected: DblVec,
    labels: Array[String]): String = {

  val regression = new MultiLinearRAdapter[Double] (
    xt, expected
  ) //22
  val descriptor = regression.weights.map(
    _.zipWithIndex.map( case(w, n) => {
      ... // Display regression weights
    } )
  )
  s"rss= ${regression.rss}" //23
}
```

The `getRSS` method merely trains the model by instantiating the multi-linear regression class (*line 22*). Once the regression model is trained during the instantiation of the `MultiLinearRegression` class, the coefficients of the regression weights and the RSS values are stringized (*line 23*). The `getRSS` method is invoked for any combination of the ETFs, GLD, SPY, and TLT, against the CNY label.

Let's look at the test code:

```
val SMOOTHING_PERIOD: Int = 16 //24
val symbols = Array[String] ("CNY", "GLD", "SPY", "TLT") //25
val movAvg = SimpleMovingAverage[Double] (SMOOTHING_PERIOD) //26

for {
  pfnMovAve <- Try(movAvg |>) //27
  smoothed <- filter(pfnMovAve) //28
  models <- createModels(smoothed) //29
  rsses <- Try(getModelsRSS(models, smoothed)) //30
  (mses, tss) <- totalSquaresError(models, smoothed.head) //31
} yield {
  """${rsses.mkString("\n")}\n${mses.mkString("\n")}"""
```

```

    | \nResidual error= $tss".stripMargin
}

```

The dataset is large (1,260 trading sessions) and noisy enough to warrant filtering using a simple moving average with a period of 16 trading sessions (*line 24*). The purpose of the test is to evaluate the possible correlation between the four ETFs: CNY, GLD, SPY, and TLT (*line 25*). The execution test instantiates the simple moving average (*line 26*) as described in the *Simple moving average* section of [Chapter 3, Data Pre-processing](#).

The workflow executes the following steps:

1. Instantiate the simple moving average partial function `pfnMovAve` (*line 27*).
2. Generate a smoothed historical prices series for the CNY, GLD, SPY, and TLT ETFs using the function `filter` as follows (*line 28*):

```

type PFNMOVAVE = PartialFunction[DblVec, Try[DblVec]]

def filter(pfnMovAve: PFNMOVEAVE): Try[Array[DblVec]] = Try {
  symbols.map(s =>
    DataSource(s"$getPath(s"$dataPath/$s.csv") }",
              true, true, 1))
  .map(_.get(adjClose))
  .map(pfnMovAve(_))
  .map(_.getOrElse(-1.0))
}

```

3. Generate the list of features for each model using the `createModels` method (*line 30*):

```

type Models = List[(Array[String], DblMatrix)]
type DblMatrix = Array[Array[Double]]

def createModels(smoothed: Array[DblVec]): Try[Models] =
  Try {
    val features = smoothed.drop(1).map(_.toArray) //32
    List[(Array[String], DblMatrix)]( //33
      (Array[String]("CNY", "SPY", "GLD", "TLT"),
       features.transpose),
      (Array[String]("CNY", "GLD", "TLT"),
       features.drop(1).transpose),
      (Array[String]("CNY", "SPY", "GLD"),
       features.take(2).transpose),
      (Array[String]("CNY", "SPY", "TLT"),
       features)
    )
  }
}

```

```

        features.zipWithIndex
            .filter( _._2 != 1)
            .map( _._1)
            .transpose),
        (Array[String] ("CNY", "GLD"),
         features.slice(1,2).transpose)
    )
}

```

The smoothed values for *CNY* are used as the expected values. Therefore, it is removed from the features list (*line 32*). The five models are evaluated by adding or removing elements from the features list (*line 33*).

4. Next, the workflow computes the residual sum of squares for all the models using `getModelsRSS` (*line 30*). The method invokes `getRSS`, introduced earlier in this section, for each model (*line 34*):

```

def getModelsRSS(models: Models, y: Array[DblVec]): List[String] =
  models.map{ case (labels, m) =>
    s"${getRSS(m.toVector, y.head, labels)}" } //34

```

5. Finally, the last step of the workflow consists of computing the mean squared errors, `mse`, for each model and the total squared error, `tss` (*line 31*):

```

def totalSquaresError(
  models: Models,
  expected: DblVec): Try[(List[String], Double)] = Try {

  val errors = models.map{
    case (labels, m) => rssSum(m, expected)._1 //35
  }
  val mse = models.zip(errors).map{
    case(f, e) => s"MSE: ${f._1.mkString(" ")} = $e"
  }
  (mse, Math.sqrt(errors.sum)/models.size) //36
}

```

The `totalSquaresError` method computes the error for each model by summing the *RSS* value, `rssSum`, for each model (*line 35*). The method returns a pair of array of the mean squared error for each model and the total squares error (*line 36*).

The RSS does not always provide an accurate visualization of the fitness of the regression model. The fitness of the regression model is commonly assessed using the **r^2 statistics**. The r^2 value is a number that indicates how well data fits a statistical model.

M4: RSS and r^2 statistics are defined in the following formula:

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

The implementation of the computation of the r^2 statistics is simple: for each model, f_j , the `rssSum` method computes the tuple {RSS, least squares errors} as defined in formula M4:

```
def rssSum(xt: DblMatrix, expected: DblVec): DblPair = {
    val regression = MultiLinearRegression[Double](xt, expected) //37
    val pfnRegr = regression |> //38
    val results = sse(expected.toArray, xt.map(pfnRegr(_).get))
    (regression.rss, results) //39
}
```

The `rssSum` method instantiates the `MultiLinearRegression` class (*line 37*), retrieves the RSS value, then validates the regressive model `pfnRegr` (*line 38*) against the expected values (*line 39*). The output of the test is presented in the following screenshot:

```

CNY = f(SPY, GLD, TLT)
0.16089780923264457 + -0.21189413823325406.x1 + 0.26299169969099795.x2 + 0.3556562652009136.x3
RSS: 3.681353535940423

CNY = f(SPY, TLT)
0.2039015515038045 + -0.03796334296279046.x1 + 0.26219728078589966.x2
RSS: 3.8589613138639227

CNY = f(GLD, TLT)
0.19290917330324198 + 0.015507174710195552.x1 + 0.2204800144601237.x2
RSS: 3.8849688539396317

CNY = f(SPY, GLD)
0.22242202699107552 + 0.17842973203100937.x1 + -0.12099602178260839.x2
RSS: 4.6681933948464645

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

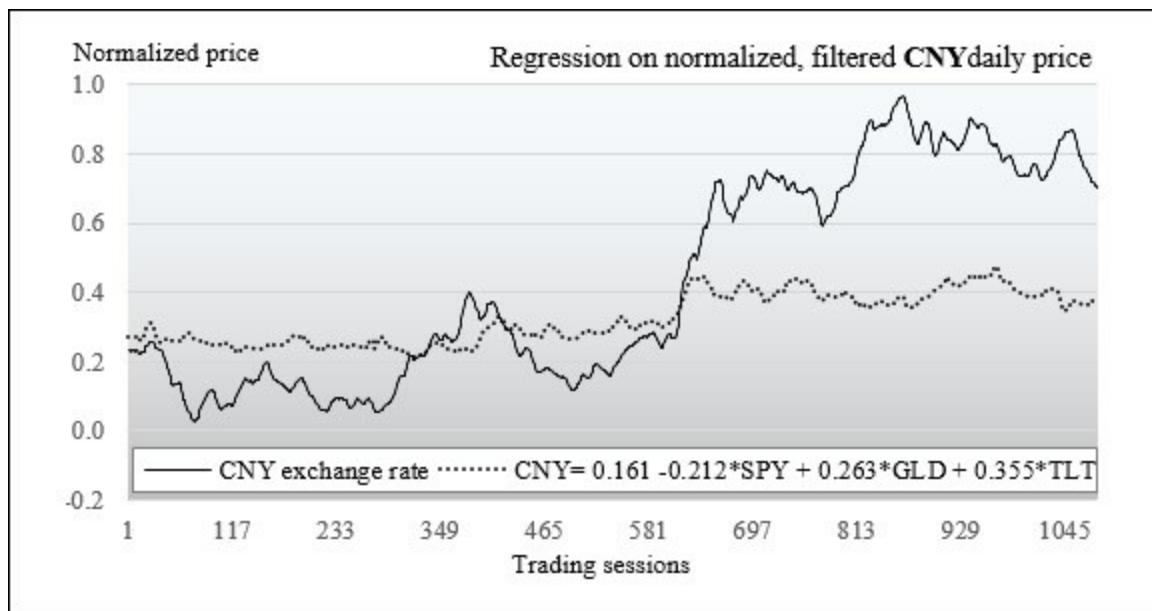
CNY = f(TLT)
0.19724352413716711 + 0.22501420632545652.x1
RSS: 3.8876824376753705

CNY = f(GLD)
0.198195293931846 + 0.16413676262473123.x1
RSS: 5.149661975952835

```

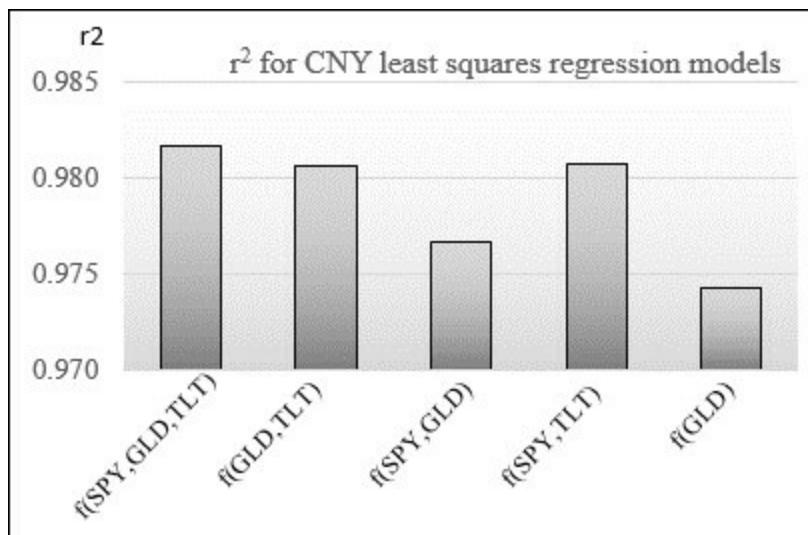
The output results clearly show that the three-variable regression $CNY=f(SPY, GLD, TLT)$ is the most accurate or fittest model for the time series CNY , followed by $CNY=f(SPY, TLT)$. Therefore, the feature selection process generates the features set $\{SPY, GLD, TLT\}$.

Let's plot the model against the raw data:



Ordinary least regression on the Chinese Yuan ETF (CNY)

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



Bar chart of the r^2 value for different regression models for CNY ETF

The graph plotting the r^2 value for each of the models confirms that of the three features models, $CNY=f(SPY, GLD, TLT)$ is the most accurate.

Note

General linear regression

The concept of linear regression is not restricted to polynomial fitting models such as $y = w_0 + w_1 \cdot x + w_2 \cdot x^2 + \dots + w_n \cdot x^n$. Regression models can be also defined as a linear combination of **basis functions** ϕ_j : $y = w_0 + w_1 \cdot \phi_1(x) + w_2 \phi_2(x) + \dots + w_n \phi_n(x)$ [9:9].

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 *Overfitting* section of [Chapter 2, Data Pipelines](#).

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) to prevent the model parameters (also known as weights) from reaching high values. A model that fits a training set very well tends to have many features 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 (M5):

$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w}_d} \left\{ \sum_{i=0}^{n-1} (y_i - f(\mathbf{x}_i | \mathbf{w}))^2 + \lambda J(\mathbf{w}) \right\}$$

The penalty function is completely independent from the training set $\{x, y\}$. The penalty term is usually expressed as a power to function of the norm of the model parameters (or weights) w_d . For a model of D dimensions the generic **L_p-norm** is defined as follows (M6):

$$J_{pq}(\mathbf{w}) = \|\mathbf{w}\|_p^q = \left[\sum_{d=1}^{D-1} |w_d|^p \right]^{q/p}$$

Tip

Notation

Regularization applies to parameters or weights associated to the observations. In order to be consistent with our notation, w_0 being the intercept value, the regularization applies to the parameters $w_1 \dots w_d$.

The two most commonly used penalty functions for regularization are L_1 and L_2 .

Tip

Regularization in machine learning

The regularization technique is not specific to linear or logistic regression. Any algorithm that minimizes the residual sum of squares, such as 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 **Lasso regularization**. **Ridge regression** is 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 feature selection [9:10] [9:11]:

- **Model estimation:** L_1 generates a sparser estimation of the regression parameters than L_2 . For a large non-sparse dataset, L_2 has a smaller estimation error than L_1 .
- **Feature selection:** L_1 is more effective in reducing the regression weights for features with a higher value than L_2 . Therefore, L_1 a reliable features selection tool.
- **Overfitting:** Both L_1 and L_2 reduce the impact of overfitting. However, L_1 has a significant advantage in overcoming overfitting (or excessive complexity of a model); for the same reason, L_1 is more appropriate for selecting features.
- **Computation:** L_2 is conducive to a more efficient computation model. The summation of the loss function and L_2 penalty, w_2 , is a continuous and differentiable function for which the first and second derivative can be computed (**convex minimization**). The L_1 component $|w_i|$, which is a non-continuous function and therefore, not differentiable.

Note

Terminology

Ridge regression is sometimes called **penalized least squares** regression. L2 regularization is also known as the **weight decay**.

Let's implement ridge regression, and then evaluate the impact of the L2-norm penalty factor.

Ridge regression

Ridge regression is a multivariate linear regression with a L2-norm penalty term (M7):

$$\tilde{w}_{ridge} = \arg \min_w \left\{ \sum_{j=0}^{N-1} (y - w_0 - w^T x_j)^2 + \lambda |w|_2^2 \right\} \quad |w|_2^2 = \sum_1^{D-1} w_d^2$$

The computation of the ridge regression parameters requires the resolution of the system of linear equations similar to the linear regression.

M8: A matrix representation of ridge regression closed form for an input dataset X , a regularization factor λ , and an expected values vector y , is as follows (I is the identity matrix):

$$\{X^T X - \lambda I\} \cdot \hat{w}_{Ridge} = X^T y$$

M9: The matrices equation is resolved using QR decomposition as follows:

$$\{X^T X - \lambda I\} = Q \begin{bmatrix} R \\ 0 \end{bmatrix} \quad w_{Ridge} = Q^T y \begin{bmatrix} R \\ 0 \end{bmatrix}^{-1}$$

Design

The implementation of ridge regression adds an L_2 regularization term to the multiple linear regression computation of the Apache Commons Math library. The methods of `RidgeRegression` have the same signature as its ordinary least squares counterpart, except for the L_2 penalty term `lambda` (*line 1*):

```
class RidgeRegression[T: ToDouble] ( //1
```

```

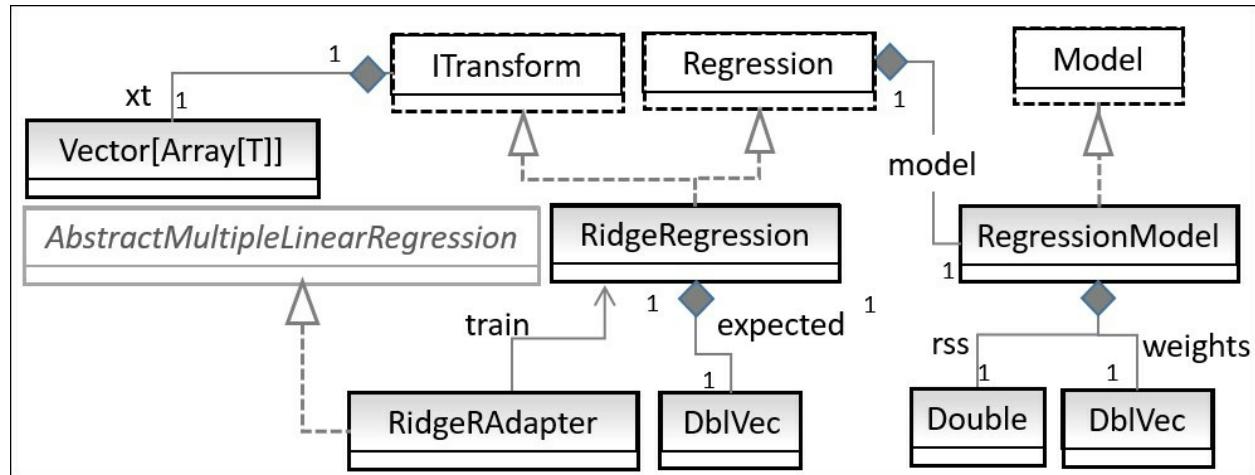
xt: Vector[Array[T]],
expected: DblVec,
lambda: Double)
extends ITransform[Array[T], Double] with Regression {/2

override def train: Option[RegressionModel] //4
override def |> : PartialFunction[Array[T], Try[Double]]//3
}

```

The `RidgeRegression` class is implemented as a data transformation, `ITransform`, which is implicitly derived from the input data (training set) as described in the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#) (line 2). The type of the output of the predictive function `|>` is a `Double` (line 3). The `model` is created through training during the instantiation of the class.

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



UML class diagram for ridge regression

The UML diagram omits the helper traits or classes such as `Monitor` or Apache Commons Math components.

Implementation

Let's look at the training method, `train`:

```

def train: RegressionModel = {
  val mlr = new RidgeRAdapter(lambda, xt.head.size) //4

  mlr.createModel(
    xt.map(_.map(implicitly[ToDouble[T]].apply(_))),
    expected
  ) // 5
  RegressionModel(mlr.getWeights, mlr.getRss) //6
}

```

It is rather simple; it initialized and executed the regression algorithm implemented in the `RidgeRAdapter` class (*line 4*), which acts as an adapter to the internal `AbstractMultipleLinearRegression` Apache Commons Math library class in the `org.apache.commons.math3.stat.regression` package (*line 5*). The method returns a fully initialized regression model, like the ordinary least squared regression (*line 6*).

Let's look at the adapter class, `RidgeRAdapter`:

```

class RidgeRAdapter(
  lambda: Double,
  dim: Int) extends AbstractMultipleLinearRegression {
  var qr: QRDecomposition = _ //7

  def createModel(x: DblMatrix, y: DblVec): Unit = { //8
    this.newXSampleData(x) //9
    super.newYSampleData(y.toArray)
  }
  def getWeights: Array[Double] = calculateBeta.toArray //10
  def getRss: Double = rss
}

```

The constructor for the `RidgeRAdapter` class takes two parameters; the L_2 penalty parameter, `lambda`, and the number of features, `dim`, in an observation. The QR decomposition in the `AbstractMultipleLinearRegression` base class does not process the penalty term (*line 7*). Therefore, the creation of the model has to be redefined in the `createModel` method (*line 8*), which needs us to override the `newXSampleData` method (*line 9*):

```

override protected def newXSampleData(x: DblMatrix): Unit = {
  super.newXSampleData(x) //11
  val r: RealMatrix = getX
}

```

```

        (0 until dim).foreach(i =>
            r.setEntry(i, i, r.getEntry(i,i) + lambda) ) //12
        qr = new QRDecomposition(r) //13
    }
}

```

The `newXSampleData` method overrides the default observations-features matrix `r` (*line 11*) by adding the `lambda` coefficient on its diagonal elements (*line 12*), then updating the QR decomposition components (*line 13*).

The weights for the ridge regression model is computed by implemented formula M6 (*line 11*) in the overridden `calculateBeta` method (line 10):

```

override protected def calculateBeta: RealVector =
    qr.getSolver().solve(getY())
}

```

The predictive algorithm for the ordinary least squares regression is implemented by the data transformation `|>`. The method predicts the output value given a `model` and an input value `x` (*line 14*):

```

def |> : PartialFunction[Array[T], Try[Double]] = {
    case x: Array[T] if(isModel &&
        x.length == model.map(_.size-1).getOrElse(0) =>
        Try(margin(x, model.get) ) //14
    )
}

```

Test case

The objective of the test case is to identify the impact of the L_2 penalization on the RSS value then compare the predicted values with 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 feature. The implementation of the extraction of observations is identical to that of the least squares regression, described in the previous section:

```

val LAMBDA: Double = 0.5

for {
    path <- gePath(s"supervised/regression/CU.csv")
    src <- DataSource(path, true, true, 1) //15
}

```

```
price <- src.get(adjClose) //16
volatility <- src.get(volatility) //17
volume <- src.get(volume) //18
(features, expected) <- differentialData(
    volatility, volume, price, diffDouble //19
)
regression <- RidgeRegression[Double] (
    features, expected, LAMBDA //20
)
} yield {
if( regression.isModel ) {
    val trend = features.map(
        margin(_, regression.weights.get) //21
    )
    val y1 = predict(0.2, expected, volatility, volume) //22
    val y2 = predict(5.0, expected, volatility, volume)
    val output = (2 until 10 by 2).map( n =>
        predict(n*0.1, expected, volatility, volume)
    )
}
```

Let's look at the steps in the execution of the test. It consists of collecting data, extracting the features and expected values, and training the ridge regression model:

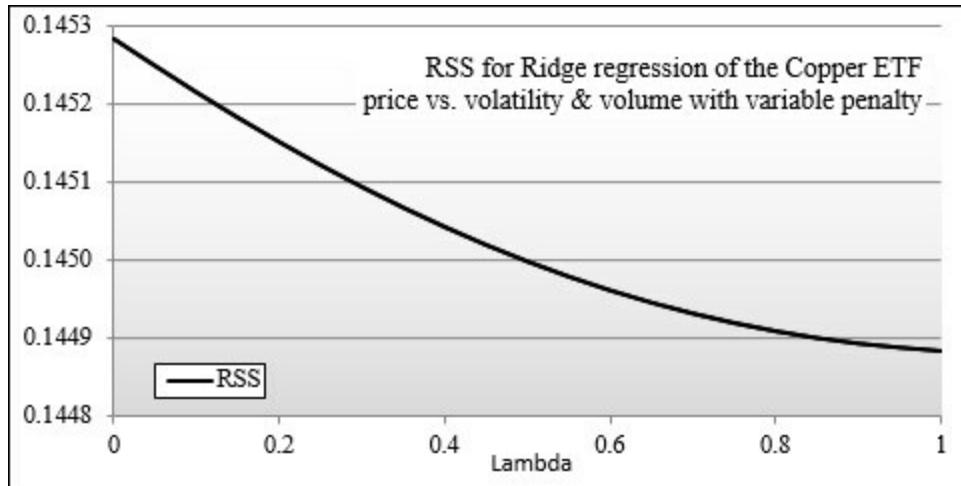
1. Create a data source extractor for the trading session closing price, the session volatility, and session volume for the ETF CU using the DataSource transformation (*line 15*).
 2. Extract the closing price of the ETF (*line 16*), its volatility within a trading session (*line 17*), and trading volume during the same session (*line 18*).
 3. Generate the labeled data as a pair of features (relative volatility and relative volume for the ETF) and expected outcome $\{0, 1\}$ for training the model for which 1 represents price increase and 0 represents price decrease (*line 19*). The generic differentialData method of the XTSeries singleton is described in the *Times series* section of [Chapter 3, Data Pre-processing](#).
 4. Instantiate the RidgeRegression using the features set and the expected change in daily stock price (*line 20*).
 5. Compute the trend values using the margin function of the RegressionModel singleton (*line 21*).

6. Execute a using the ridge regression is implemented by the predict method (*line 22*):

```
def predict(
    lambda: Double,
    deltaPrice: DblVec,
    volatility: DblVec,
    volume: DblVec): DblVec = {

    val observations = zipToSeries(volatility, volume) //25
    val regression = new RidgeRegression[Double](
        observations, deltaPrice, lambda
    )
    val fnRegr = regression |>      //26
    observations.map( fnRegr(_).get) //27
}
```

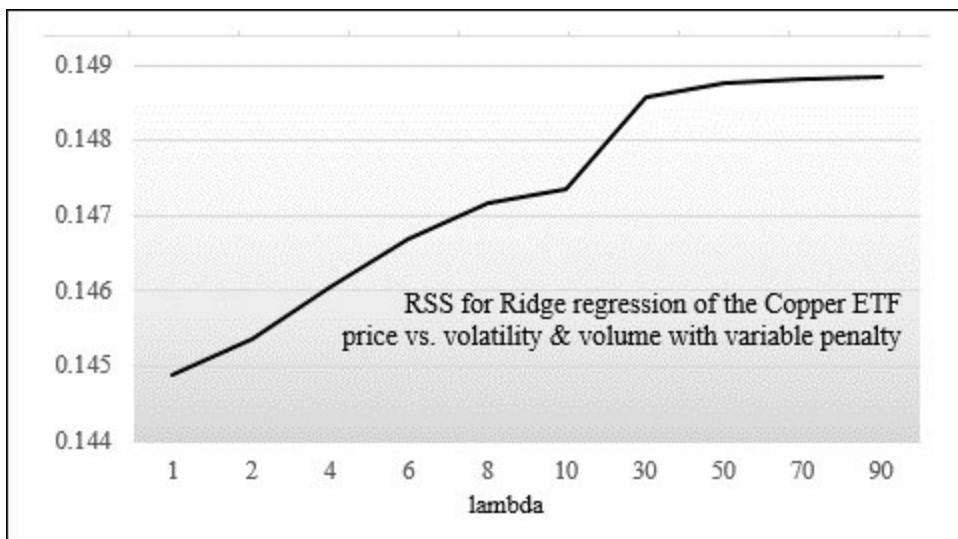
The `observations` are extracted from the `volatility` and `volume` time series (*line 25*). The predictive method for the ridge regression, `fnRegr` (*line 26*), is applied to each observation (*line 27*). The RSS value, `rss`, is plotted for different values of λ , as shown in the following chart:



Graph of RSS versus lambda for copper ETF

The residual sum of squares decreased as λ increases. The curve seems reaching for a minimum around $\lambda=1$. The case of $\lambda = 0$ corresponds to the least squares regression.

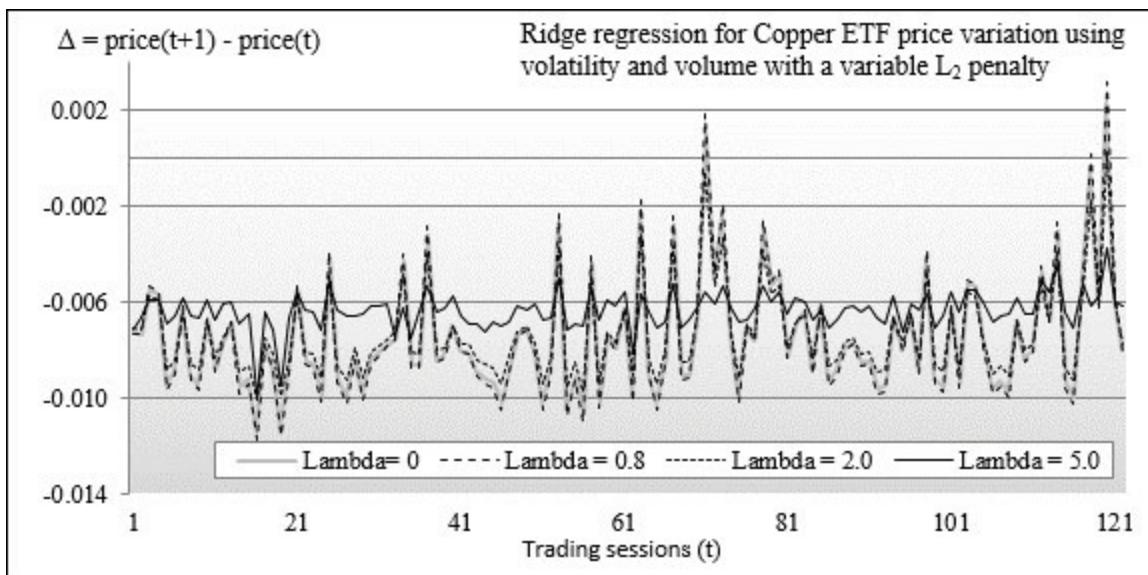
Next, let's plot the RSS value for λ varying between 1 and 100:



Graph RSS versus large value lambda for Copper ETF

This time around, the value of RSS increases with λ , before reaching a maximum for $\lambda > 60$. This behavior is consistent with other findings [9:12]. As λ increases, the overfitting gets more expensive and therefore the RSS value increases.

Let's plot the predicted price variation of the copper ETF using ridge regression with a different value of lambda (λ):



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 $\text{lambda} = 0$. Let's analyze the behavior of predictive model for several values of lambda:

- The predicted values for $\lambda = 0.8$ are very similar to the original values
- The predicted values for $\lambda = 2$ follow the pattern of the original values with reduction of large variations (peaks and troughs).
- The predicted values for $\lambda = 5$ correspond to a smoothed dataset. The pattern of the original values is preserved, but the magnitude of the price variation is significantly reduced.

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. Logistic regression relies on optimization methods. Let's get through a short refresher course in optimization before diving into the logistic regression.

Numerical optimization

This section briefly introduces the different optimization algorithms that can be applied to minimize the loss function, with or without a penalty term. These algorithms are described in greater detail in the *Summary of optimization technique* section of the *Appendix*.

First, let's define the **least squares problem**. The minimization of the loss function consists of nullifying the first order derivatives, which in turn generates a system of D equations (also known as the gradient equations), D being the number of regression weights (parameters). The weights are iteratively computed by solving the system of equation using a numerical optimization algorithm.

M10: The definition of the least squares-based loss function for residual r_i , weights w , a model f , input data x_i and expected values y_i , is as follows:

$$L(w) = \sum_{i=0}^{n-1} r_i(w)^2 \quad r_i(w) = y_i - f(x_i | w)$$

M10: Generation of gradient equations with **Jacobian J** matrix (refer to *Basics of differential calculus* section of the *Appendix*) after minimization loss function, L is described as follows:

$$\sum_{i=0}^{n-1} r_i(w) J_{id}(w) = 0 \quad J_{id}(w) = -\frac{\partial r_i(w)}{\partial w_d}$$

M11: Iterative approximation using the Taylor series on model f for k iterations on the computation of weights w , is defined as follows:

$$f(x_i | w) - f(x_i | w^{(k)}) \sim \sum_{jd=0}^{D-1} \frac{\partial f(x_i | w^{(k)})}{\partial w_d} (w - w^{(k)})$$

Logistic regression is a non-linear function. Therefore, it requires the minimization of non-linear of the sum of least squares. The optimization algorithms for the non-linear least squares problems can be divided into two categories:

- **Newton** (or second-order techniques): These algorithms calculate the second-order derivatives (Hessian matrix) to compute the regression weights that nullify the gradient. The two most common algorithms in this category are the **Gauss-Newton** and **Levenberg-Marquardt** methods (refer to the *Non-linear least squares minimization* section of the *Appendix*). Both algorithms are included in the Apache Commons Math library.
- **Quasi-newton** (or first-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-values functions, not just 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 of the *Appendix*).

Logistic regression

As described in the introduction, *the logistic regression is indeed a classifier*. It is possibly the most commonly used discriminative learning techniques for at least two reasons: its simplicity and the large variety of optimization algorithms used in the training of the model. Conceptually, logistic regression is the quantification of the relationship between an observed, target (or expected) variable, y , and a set of variables, x , that it depends on. Once the model is created (trained), it is available to classify real-time data.

Note

Generalized linear models

Logistic regression belongs to the broad category of **generalized linear models (GLM)**. It relies on a linear combination of inputs or observations passed through a non-linear function, known as the logistic regression [9:13].

A logistic regression can be either binomial (two classes) or multinomial (three or more classes). In a binomial classification, the observed outcome is defined as $\{true, false\}$, $\{0, 1\}$ or $\{-1, +1\}$.

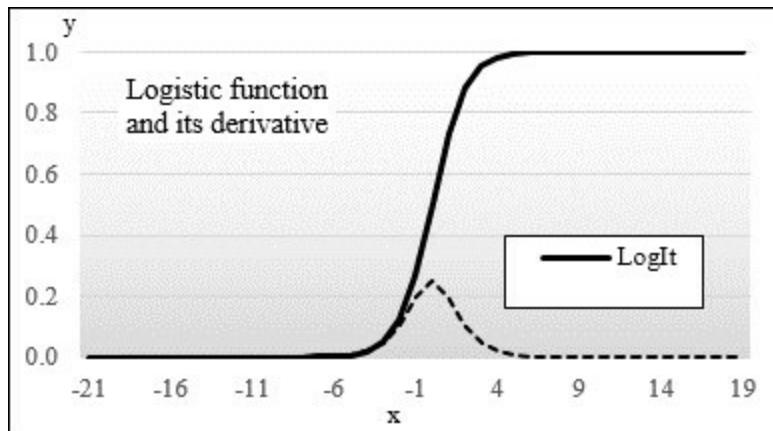
Logistic function

The conditional probability in a linear regression model is a linear function of its weights [9:14]. The logistic regression model addresses the non-linear regression problem by defining the logarithm of the conditional probability as a linear function of its parameters.

First, let's introduce the logistic function and its derivative, which are defined as follows (M12):

$$f(x) = \frac{1}{(1-e^{-x})} \quad \frac{df}{dx} = f(x)(1-f(x))$$

The logistic function and its derivative are illustrated in the following graph:



Graph of the logistic function and its derivative

The remainder of this section is dedicated to the application of the multivariate logistic regression to the binomial classification.

The logistic regression is the most commonly used discriminative model for the following reasons:

- It is available in most statistical software packages and open source

libraries

- Its S-shape describes the combined effect of several explanatory variable
- Its range of values [0, 1] is intuitive from a probabilistic perspective
- It is a generalized linear model

Let's consider the classification problem using two classes. As discussed in the *Validation* section of [Chapter 2, Data Pipelines](#), even the best classifier produces false positives and false negatives. Whatever metric(s) is used to evaluate the quality of a model, the objective of the training is to identify a geometrical shape (or hyperplane) that segregates the observations into two distinct classes.

The following diagram illustrates the computation of the hyperplane for a generic binomial classification model:

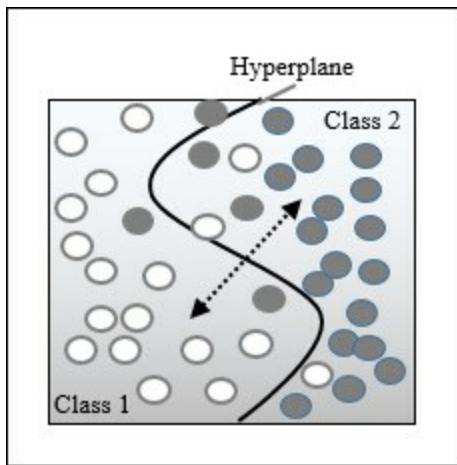


Illustration of the binomial classification for two-dimensional dataset

Mathematically speaking, a hyperplane in an n-dimensional space (number of features) is a sub-space of n-1 dimension as described in the *Manifolds* section under *Non-linear models* in [Chapter 5, Dimension Reduction](#).

Note

Max-margin classifier

The concept of hyperplane is revisited in [Chapter 12, Kernel Models and](#)

Support Vector Machines. The support vector machine algorithm attempts to increase and maximize the distance between class of observations during training by defining two vectors on each side of the segregating hyperplane.

The separating hyperplane of a three-dimensional space is a curved surface. The separating hyperplane of a two-dimension problem (plane) is a line. In the 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).

Note

Hyperplane for a generalized linear model

The hyperplane for generalized linear model is defined a multi-dimensional linear function, margin = 0. In the case of the binomial logistic regression, the hyperplane is simply the scalar product of the regression parameters (weights) and inputs.

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 class.

In the case of two classes, $c1$ and $c2$ the probability for an observation to belong to the class $c1$ is $p(C=c1 | X=x_i|w) = p(x_i|w)$ and the probability it belongs to the class $c2$ is $p(C=c2 | X=x_i|w) = 1 - p(x_i|w)$, where w is the model weights.

Note

Logistic regression:

M13: Here is the log-likelihood for N observations x_i given regression weights w :

$$L(w) = \sum_{i=0}^{N-1} \log p(x_i | w)$$

M14: Here are the conditional probabilities $p(x|w)$ with regression weights w , using the logistic function for N observations with d features $\{x_{ij}\}_{j=0:d-1}$:

$$x_i = \{1, x_{i0}, \dots, x_{id-1}\} \quad p(x_i | w) = \frac{1}{1 + e^{-w^T x_i}} \quad w^T x_i = \sum_{j=0}^d w_j x_{ij}$$

M15: Here is the sum of square errors, sse , for the binomial logistic regression with weights w , input values x_i , and expected binary outcome y :

$$sse(w) = \frac{1}{2} \sum_{i=0}^{N-1} \left\{ y_i - \log \left(1 + e^{-w^T x_i} \right) \right\}^2 \quad y \in \{0, 1\}$$

M16: Computation of the weights w of the logistic regression by maximizing the log-likelihood given the input data x_i and the expected outcome (labels) y_i :

$$\frac{\partial L(\tilde{w})}{\partial w_j} = \sum_{i=0}^{N-1} x_{ij} \left(y_i - \frac{1}{1 + e^{-\tilde{w}^T x_i}} \right) = 0$$

Let's implement the logistic regression without regularization using the Apache Commons Math library. The library contains several least squares optimizers, allowing you to specify the minimizing algorithm, `optimizer`, for the loss function in the `LogisticRegression` class.

The constructor for the `LogisticRegression` class follows a very familiar pattern. It defines a data transformation, `ITransform`, which is implicitly derived from the input data (training set) as described in the *Monadic data transformation* section in [Chapter 2, Data Pipelines \(line 2\)](#). The output of the predictor, `|>`, is a class id. Therefore, the type of the output is an `Int` (*line 3*):

```

class LogisticRegression[T: ToDouble] (
  xt: Vector[Array[T]],
  expected: Vector[Int],
  optimizer: LogisticRegressionOptimizer) //1
extends ITransform[Array[T], Int] with Regression { //2
  override def train: RegressionModel //4
  def |> : PartialFunction[Array[T], Try[Int]] //3
}

```

The parameters of the logistic regression class are the multivariate time series (features), `xt`, the target or expected classes, `expected`, and the `optimizer` used to minimize the loss function or residual sum of squares (*line 1*). In the case of the binomial logistic regression, `expected` is assigned the values of 1 for one class and 0 for the other.

The purpose of the training is to determine the regression weights that minimize the loss function as defined in formula M14 as well as the RSS (*line 4*).

Tip

Target values

There is no specific rule to assign the two values to observed data for the binomial logistic regression: $\{-1, +1\}$, $\{0, 1\}$ or $\{\text{false}, \text{true}\}$. The values pair $\{0, 1\}$ is convenient because it allows the developer to reuse the code for multinomial logistic regression using normalized class values.

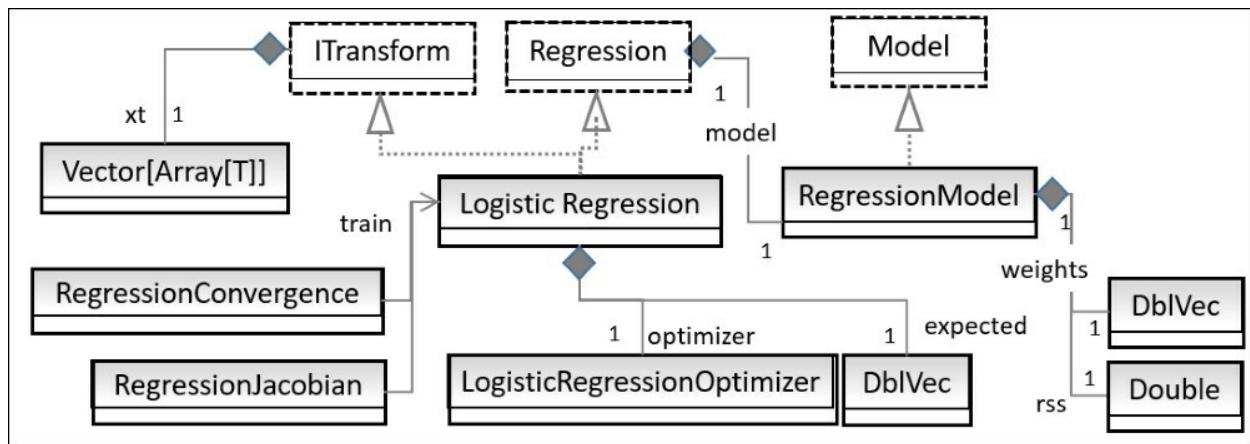
For convenience, the definition and the configuration of the optimizer are encapsulated in the `LogisticRegressionOptimizer` class.

Design

The implementation of the logistic regression uses the following components:

- A `RegressionModel` model of type `Model` that is initialized through training during the instantiation of the classifier. We reuse the `RegressionModel` type introduced in the *linear regression* section.
- The `LogisticRegression` class, which implements an `ITransform` for the prediction of future observations.
- An adapter for the computation of the Jacobian; `RegressionJacobian`.
- An adapter to manage the convergence criteria and exit conditions of the minimization of the sum of square errors; `RegressionConvergence`.

The key software components of logistic regression are described in the following UML class diagram:



UML class diagram for the logistic regression

The UML diagram omits the helper traits or classes such as `Monitor` or `Apache Commons Math` components.

Training workflow

Our implementation of the training of the logistic regression model leverages either **Gauss-Newton** or the **Levenberg-Marquardt** non-linear least squares optimizers (refer to the *Non-linear least squares minimization* section of *Appendix*) packaged with the Apache Commons Math library.

The training of the logistic regression is performed by the `train` method.

Tip

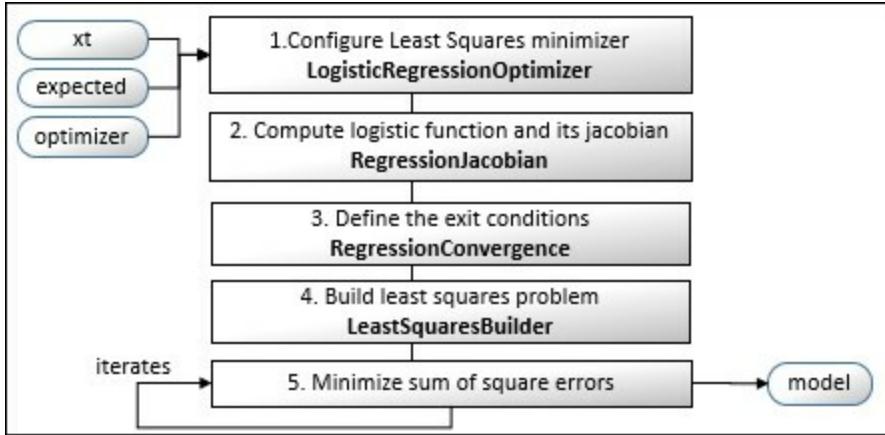
Handling exceptions from Apache Commons Math library

The training of the logistic regression using Apache Commons Math library requires handling the `ConvergenceException`, `DimensionMismatchException`, `TooManyEvaluationsException`, `TooManyIterationsException` and `MathRuntimeException` exceptions. Debugging is greatly facilitated by understanding the context of these exceptions in the Apache library source code.

The implementation of the training method, `train`, relies on five steps:

1. Select and configure the least squares optimizer.
2. Define the logistic function and its Jacobian.
3. Specify the convergence and exit criteria.
4. Compute the residuals using the least squares problem builder.
5. Run the optimizer.

The workflow and the Apache Commons Math classes used in the training of the logistic regression is visualized by the following flow diagram:



Workflow for training 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 the minimization of the loss function. Let's start with the configuration of the least squares optimizer:

```

def train: RegressionModel = {
    val weights0 = Array.fill(data.head.length +1) (INITIAL_WEIGHT)
    val lrJacobian = new RegressionJacobian(data, weights0) //5
    val exitCheck = new RegressionConvergence(optimizer) //6

    def createBuilder: LeastSquaresProblem //7
    val optimum = optimizer.optimize(createBuilder) //8
    RegressionModel(optimum.getPoint.toArray, optimum.getRMS)
}

```

The `train` method implements the last four steps of the computation of the regression model:

1. Computation of logistic values and the Jacobian matrix (*line 5*).
2. Initialization of the convergence criteria (*line 6*).
3. Definition of the least square problem (*line 7*).
4. Minimization of the sum of square errors (*line 8*): this is performed by the optimizer as part of the constructor of `LogisticRegression`.

Step 1 – configuring the optimizer

In this step, you have to specify the algorithm to minimize the residual of the

sum of squares. The `LogisticRegressionOptimizer` class is responsible for configuring the optimizer. The class has two purposes:

- Encapsulate the configuration parameters for the optimizer
- Invoke the `LeastSquaresOptimizer` interface defined in the *Apache Commons Math library*.

The implementation of the class `LogisticRegressionOptimizer` follows:

```
class LogisticRegressionOptimizer(  
    maxIters: Int,  
    maxEvals: Int,  
    eps: Double,  
    lsOptimizer: LeastSquaresOptimizer) { //9  
def optimize(lsProblem: LeastSquaresProblem): Optimum =  
    lsOptimizer.optimize(lsProblem)  
}
```

The configuration of the logistic regression optimizer is defined as the maximum number of iterations `maxIters`, the maximum number of evaluations, `maxEval` for the logistic function and its derivatives, the convergence criteria `eps` on the residual sum of squares, and the instance of the least squares problem (*line 9*).

Step 2 – computing the Jacobian matrix

The next step consists of computing the value of the logistic function and its first order partial derivatives with respect the weights by overriding the `value` method of the `fitting.leastsquares.MultivariateJacobianFunction` Apache Commons Math interface:

```
class RegressionJacobian[T: ToDouble]( //10  
    xv: Vector[Array[T]],  
    weights0: Array[Double])  
extends MultivariateJacobianFunction {  
  
    type GradientJacobian = Pair[RealVector, RealMatrix]  
    override def value(w: RealVector): GradientJacobian = { //11  
  
        val gradient = xv.map( g => { //12  
            val f = logistic(margin(g, w)) //13  
            ...  
        })  
        ...  
    }  
}
```

```

        (f, f*(1.0-f)) //14
    })
xv.indices //15
./Array.ofDim[Double](xv.size, weights0.size)) {
case (j, i) => {
    val df = gradient(i)._2
    val z = implicitly[ToDouble[T]].apply(xv(i)(n))
    x.indices.foreach(n => j(i)(n+1) = z*df)
    j(i)(0) = 1.0; j //16
}
}
(new ArrayRealVector(gradient.map(_._1).toArray),
 new Array2DRowRealMatrix(jacobian)) //17
}
}

```

The constructor for the `RegressionJacobian` class requires two arguments (*line 10*):

- The time series of observations, `xv`
- The initial regression weights, `weights0`

The `value` method uses the primitive types `RealVector`, `RealMatrix`, `ArrayRealVector`, and `Array2DRowRealMatrix` defined in the `org.apache.commons.math3.linear` Apache Commons Math package (*line 11*). It takes the regression weights, `w`, as an argument and computes the gradient (*line 12*) of the logistic function (*line 13*) for each data point that is returned, along with the values of its derivative (*line 14*).

The Jacobian matrix is populated with the values of the derivative of the logistic function (*line 15*). The first element of each column of the Jacobian matrix is set to 1.0 to take into account the intercept (*line 16*). Finally, the `value` function returns the pair of gradient values and the Jacobian matrix using types that comply with the signature of the method `value` in the Apache Commons Math library (*line 17*).

Step 3 – managing the convergence of optimizer

The third step defines the exit condition for the optimizer. It is accomplished by overriding the `converged` method of the parameterized

ConvergenceChecker interface in the Java package,
org.apache.commons.math3.optim:

```
val exitCheck = new ConvergenceChecker[PointVectorValuePair] {
    override def converged(
        iters: Int,
        prev: PointVectorValuePair,
        current: PointVectorValuePair): Boolean =
        sse(prev.getValue, current.getValue) < optimizer.eps
        && iters >= optimizer.maxIterations //18
}
```

This implementation computes the convergence or exit condition as follows:

- Either the sum square errors, `sse`, between weights of two consecutive iterations is smaller than the convergence criteria `eps`.
- Or the `iters` exceeds the maximum number of iterations `maxIterations` allowed (*line 18*).

Step 4 – defining the least squares problem

The Apache Commons Math least squares optimizer package requires all the input to the non-linear least squares minimizer to be defined as an instance of the `LeastSquareProblem` generated by the factory `LeastSquareBuilder` class:

```
def createBuilder: LeastSquaresProblem =
  (new LeastSquaresBuilder).model(lrJacobian)           //19
    .weight(MatrixUtils.createRealDiagonalMatrix(
      Array.fill(xt.size)(1.0))) //20
    .target(expected.toArray) //21
    .checkerPair(exitCheck) //22
    .maxEvaluations(optimizer.maxEvals) //23
    .start(weights0) //24
    .maxIterations(optimizer.maxIterations) //25
    .build
```

The diagonal elements of the weights matrix are initialized to 1.0 (*line 20*). Besides the initialization of the model with the Jacobian matrix `lrJacobian` (*line 19*), the sequence of method invocations sets the maximum number of evaluations (*line 23*), maximum number of iterations (*line 25*), and the exit

condition (*line 22*).

The regression weights are initialized with the `weights0` arguments of the constructor for `LogisticRegression` (*line 24*). Finally, the expected or target values it initialized (*line 21*).

Step 5 – minimizing the sum of square errors

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 RMS are returned by invoking the `getPoint` method on the `Optimum` class of the Apache Commons Math library.

Test

Let us test our implementation of the binomial multivariate logistic regression using the example of the price variation versus volatility and volume for the Copper ETF, used in the two previous sections. The only difference is that we need to define the target values as 0 if the ETF price decreases between two consecutive trading sessions, and 1 otherwise:

```
import YahooFinancials._
val maxIters = 250
val maxEvals = 4500
val eps = 1e-7
val optimizer = new LevenbergMarquardtOptimizer //27

for {
    path <- getPath(s"supervised/regression/CU.csv")
    src <- DataSource(path, true, true, 1) //26
    price <- src.get(adjClose) //28
    volatility <- src.get(volatility) //29
    volume <- src.get(volume) //30
    (features, expected) <- differentialData(
        volatility, volume, price, diffInt //31
    )
}
```

```

    )
lsOpt <- LogisticRegressionOptimizer(
    maxIters, maxEvals, eps, optimizer //32
)
regr <- LogisticRegression[Double](features, expected, lsOpt)
pfnRegr <- Try(regr |>) //33
}
yield {
    val accuracy = features.map(prfRegr(_)).zip(expected).map(
        case (tp, e) => tp.map(p =>if (p==e) 1 else 0).getOrElse(-1)
    ).sum/expected.size.toDouble
    // ..
}

```

Let's look at the steps in the execution of the test that consists of collecting data, initializing the parameters for the minimization of the sum of square errors, training a logistic regression model, and run prediction:

1. Create a data source, `src`, to extract market and trading data (*line 26*).
2. Select the Levenberg-Marquardt algorithm, `LevenbergMarquardtOptimizer` as the `optimizer` (*line 27*).
3. Load the daily closing `price` (*line 28*), volatility within a trading session (*line 29*), and daily trading `volume` (*line 30*) for the ETF, CU.
4. Generate the labeled data as a pair of features (relative volatility and relative volume for the ETF) and `expected` outcome $\{0, 1\}$ for training the model for which 1 represents price increase and 0 represents price decrease (*line 31*). The generic `differentialData` method of the `TSeries` singleton is described in the *Times series* section of [Chapter 3, Data Pre-processing](#).
5. Instantiate the optimizer, `lsOpt`, to minimize the sum of square errors during training (*line 32*).
6. Train the model `regr` and return the predictor partial function `pfnRegr` (*line 33*).

There are many alternative optimizers available to minimize the sum of square errors optimizers (refer to the *Non-linear least squares minimization* section of the *Appendix*).

Tip

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-Marquard* section under *Non-linear least squares minimization* in *Appendix*).

The execution of the test produces the following results:

- **Residual mean square:** 0.495
- **Weights:** 0.425 for intercept, -2.354 for ETF volatility, 0.239 for ETF volume

The last step is the classification of real-time data.

Classification

As mentioned earlier, and despite its name, binomial logistic regression is actually a binary classifier. The classification method is implemented as an implicit data transformation, `|>:`

```
val HYPERPLANE = -log(1.0/INITIAL_WEIGHT -1)

def |> : PartialFunction[Array[T], Try[Int]] = {
  case x: Array[T] if(isModel &&
    model.get.size-1 == x.length) =>
    Try(if(margin(x, model.get) > HYPERPLANE) 1 else 0) //34
}
```

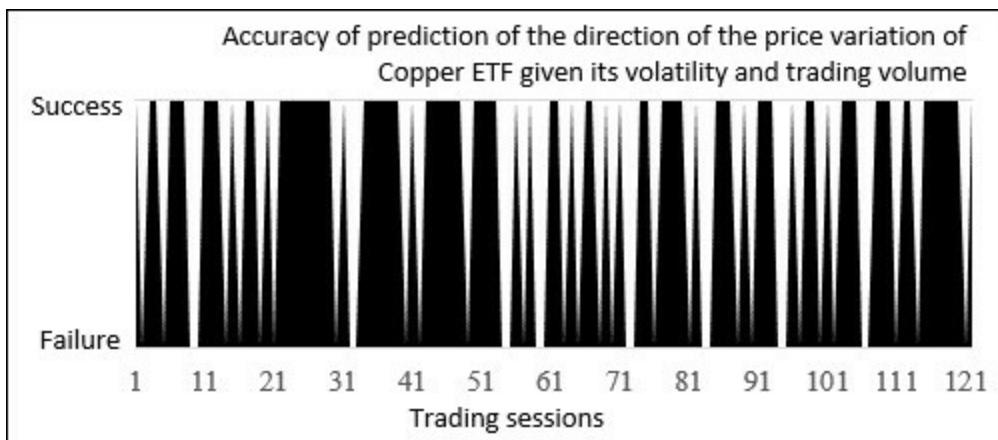
The `margin` (or scalar product) of the observation, `x`, with the `weights` `model`, is evaluated against the hyperplane. The predicted class is 1 if the produce exceeds the `HYPERPLANE`; otherwise, it's 0 (*line 34*).

Note

Class identification

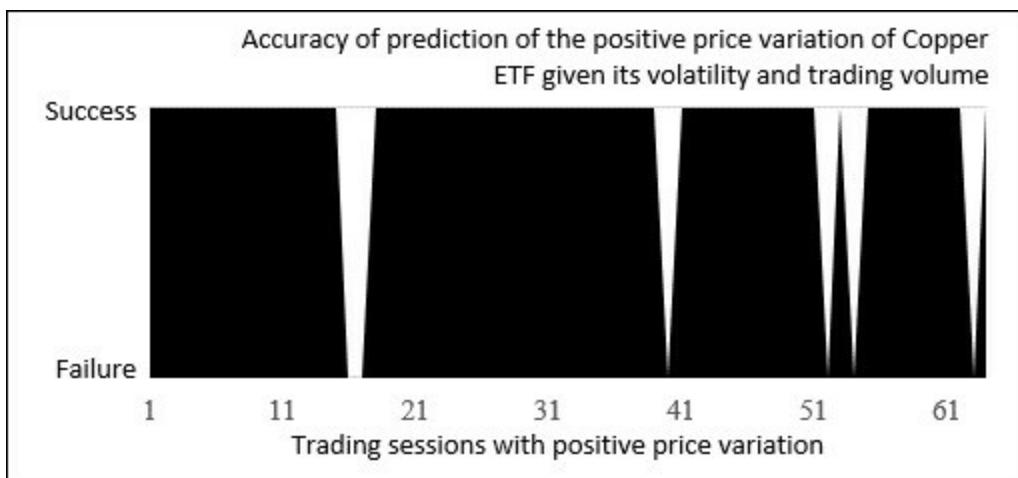
The class the new data, x , belongs to is determined by the test $\langle x, w \rangle > 0.5$ with $\langle x, w \rangle = w_0 + w_1.volatility + w_2.volume$. You may find different classification scheme in the scientific literature.

The direction of the price variation of the copper ETF, CU $price(t+1) - price(t)$ is compared to the direction predicted by the logistic regression. The result is plotted with a *success* value if the positive or negative direction is correctly predicted. It's plotted with a *failure* value otherwise:



Prediction of the direction of the variation of price of copper using logistic regression

The logistic regression was able to classify 78 out of 121 trading sessions (65% accuracy) Now, let's use the logistic regression to predict the positive price variation for the copper ETF, given its volatility and trading volume. This trading or investment strategy is known as being long on the market. This particular use case ignores the trading sessions for which the price was either flat or in decline:



Prediction of the direction of the variation of price of copper using logistic regression

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, $w_0 + w_1 \cdot volatility + w_2 \cdot volume$ is used to segregate the features generating either positive or negative price variation. Therefore, the overall accuracy of the classification is negatively impacted by the overlap of the features from the two classes.

In the second case, the classifier has to consider only the observations *located on the positive side* of the plane equation, without taking into account the false negatives.

Tip

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: $w_0 + w_1 \cdot volatility + w_2 \cdot volume$. It reduces the accuracy of the prediction of both the positive and negative price variation.

The accuracy of the binary classifier can be further improved by considering the positive variation of price using a margin error EPS as $price(t+1) - price(t) > EPS$.

Tip

Validation methodology

The validation set is generated by randomly selecting observations from the original labeled dataset. A formal validation requires us to use a K-fold validation methodology to compute the recall, precision, and F1 measure for the logistic regression model.

The logistic regression is the foundation of the conditional random fields as described in the *Conditional random fields* section of [Chapter 7, Sequential Data Models](#) and multilayer perceptron is introduced in the *M ultilayer perceptron* section of [Chapter 10, Multilayer Preceptron](#).

Summary

This chapter barely scratches the surface of the topic of generalized linear models with the description of linear and logistic regression algorithms. Regression models, along with Naïve Bayes classification, are the most well-understood techniques by those without a deep knowledge of statistics or machine learning.

At the end of this chapter, you hopefully have a grasp of the following:

- Linear and non-linear least squares-based optimization
- The implementation of ordinary least square regression, as well as logistic regression as classifiers and predictive models
- The purpose of regularization as illustrated with ridge regression

The regression models do not impose the condition that the features have to be independent, contrary to the Naïve Bayes models (refer to [Chapter 6](#), *Naïve Bayes Classifiers*). However, these models do not take into account the sequential nature of time series commonly used in dynamic asset pricing. The next chapter introduces models for sequential data, with two classifiers that take into account the time dependency and order of observations: hidden Markov model and conditional random fields.

Chapter 10. Multilayer Perceptron

The concept of artificial neural networks is rooted in biology with the idea of mimicking some of the brain's functions. Computer scientists thought that such a concept could be applied to the broader problem of parallel processing [10:1]. The key question in the 1970s was: how can we distribute the computation of tasks across a network or cluster of machines without having to program each machine? One simple solution consists of training each machine to execute the given tasks. The popularity of neural networks surged in the 1990s.

At its core, a neural network is a nonlinear statistical model that leverages the logistic regression to create a nonlinear distributed model.

Note

Deep learning:

Deep learning techniques (introduced in the next chapter) extend the concept of artificial neural networks. This chapter should be regarded as the first part of the presentation of an algorithm generally associated with deep learning.

In this chapter, you will move beyond the hype and learn the following:

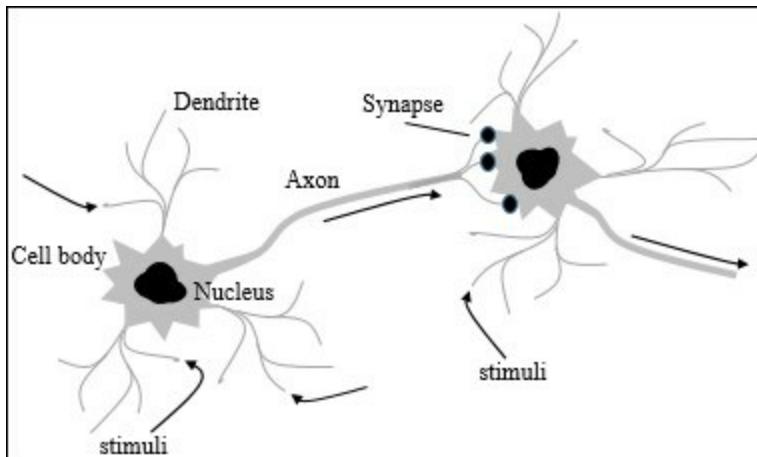
- The concept and elements of the **multilayer perceptron (MLP)**
- How to train a neural network using error backpropagation
- The evaluation and tuning of MLP configuration parameters
- A Scala implementation of the MLP classifier
- A simple application of MLP for modeling currency exchange rates

Feed-forward neural networks (FFNN)

The brain is a very powerful information processing engine that surpasses the reasoning ability of computers in domains such as learning, inductive reasoning, prediction, vision, and speech recognition. However, the simplest computing device has the capability to process very large datasets well beyond the ability of the human brain.

The biological background

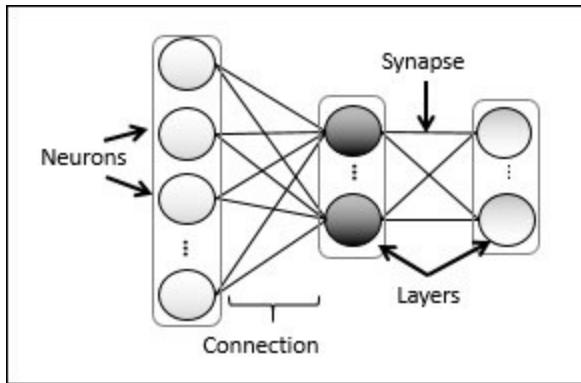
In biology, a neural network is composed of groups of neurons interconnected by synapses [10:2], as shown in the following image:



Visualization of biological neurons and synapses

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 1960s 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 [10:3].

Biological neurons communicate through electrical charges known as **stimuli**. This network of neurons can be represented as a simple schematic, as follows:



Representation of neuron layers, connections, and synapses

This representation categorizes groups of neurons as layers. The terminology used to describe the natural neural networks has a corresponding nomenclature for the artificial neural network:

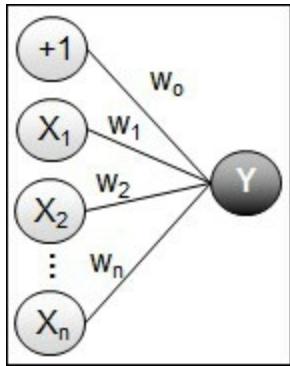
The biological neural network	The artificial neuron network
Axon	Connection
Dendrite	Connection
Synapse	Weight
Potential	Weighted sum
Threshold	Bias weight
Signal, Stimulus	Activation



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 feed-forward neural network or FFNN.

Mathematical background

The previous chapter, [Chapter 9, Regression and Regularization](#), describes the concept of the hyperplane, which segregates a set of labeled data points into distinct classes during training. The hyperplane is defined by the linear model (or margin) $w_T \cdot x + w_0 = 0$. The linear regression can be visualized as a simple connectivity model using neurons and synapses, as follows:



A two-layer basic neural network (no hidden layer)

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 phenomenon. Therefore, logistic regression is naturally used to compute the output of the perceptron. For a set of input variables $x = \{x_i\}_{0,n}$ and the weights $w = \{w_i\}_{1,n}$, the output y is computed as (M1):

$$y = \sigma(w_0 + w^T x) = \frac{1}{1 + e^{-(w_0 + w^T x)}}$$

Such an approach can be modeled as a FFNN, known as the multilayer perceptron [10:4].

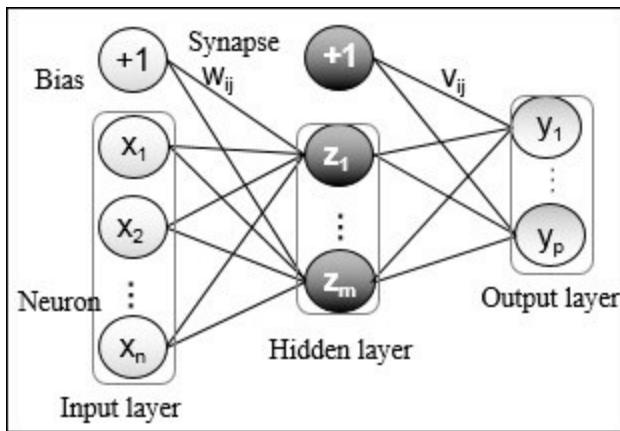
An FFNN can be regarded as a stack of layers of logistic regression with the output layer as a linear regression. The value of the variables in each hidden layer is computed as the sigmoid of the dot product of the connection weights and the output of the previous layer. Although it's interesting, the theory behind artificial neural networks is beyond the scope of this book [10:5].

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: $\{true, false\}$ or $\{-1, +1\}$. The original perceptron algorithm was defined as a single layer of neurons for which each value x 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 perceptron is 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 perceptron models. 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 (one hidden layer) perceptron

The three-layered perceptron requires two sets of weights: w_{ij} to process the output of the input layer to the hidden layer and v_{ij} between the hidden layer and the output layer. The intercept value, w_0 , in both linear and logistic regression, is represented with $+1$ in the visualization of the neural network

$(w_0 \cdot I + w_1 \cdot x_1 + w_2 \cdot x_2 + \dots)$.

Note

FFNN with no hidden layer:

An FFNN without a hidden layer is like a linear statistical model. The only transformation or connection between the input and output layer is 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.

Tip

Terminology:

Artificial neural networks encompass a large variety of learning algorithms, the multilayer perceptron being one of them. Perceptrons are 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.

Activation function

The perceptron is represented as a linear combination of weights, w_i , and input values, x_i , processed by the **output unit activation** function h , as shown here (M2):

$$\hat{y} = h\left(w_0 + \sum_{i=1}^n w_i x_i\right) = h(w_0 + w^T x)$$

The output activation function h must 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 (linear formula) of the regression mode
- *Sigmoid*, s , for hidden layers and output layer of the binomial classifier
- *Softmax* for the multinomial classification
- Hyperbolic tangent, $\tan h$, for classification using zero mean

The softmax formula is described in *Step 1 – Input forward propagation* under *Training epoch*.

Network topology

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:

- One input layer
- $(n-1)$ hidden layer
- One output layer

A fully connected neural network has all its input nodes connected to hidden layer neurons. Networks are characterized as partially connected neural networks if one or more of their input variables are not processed. This chapter deals with a fully connected neural network.

Note

Partially connected networks:

Partially connected networks are not as complex as they seem. They can be generated from fully connected networks by setting some of the weights to zero.

The structure of the output layer is highly dependent on the type of problems (regression or classification) you need to solve, also known as the operating mode of the multilayer perceptron. The type of problem at hand defines the number of output nodes [10:6], for example:

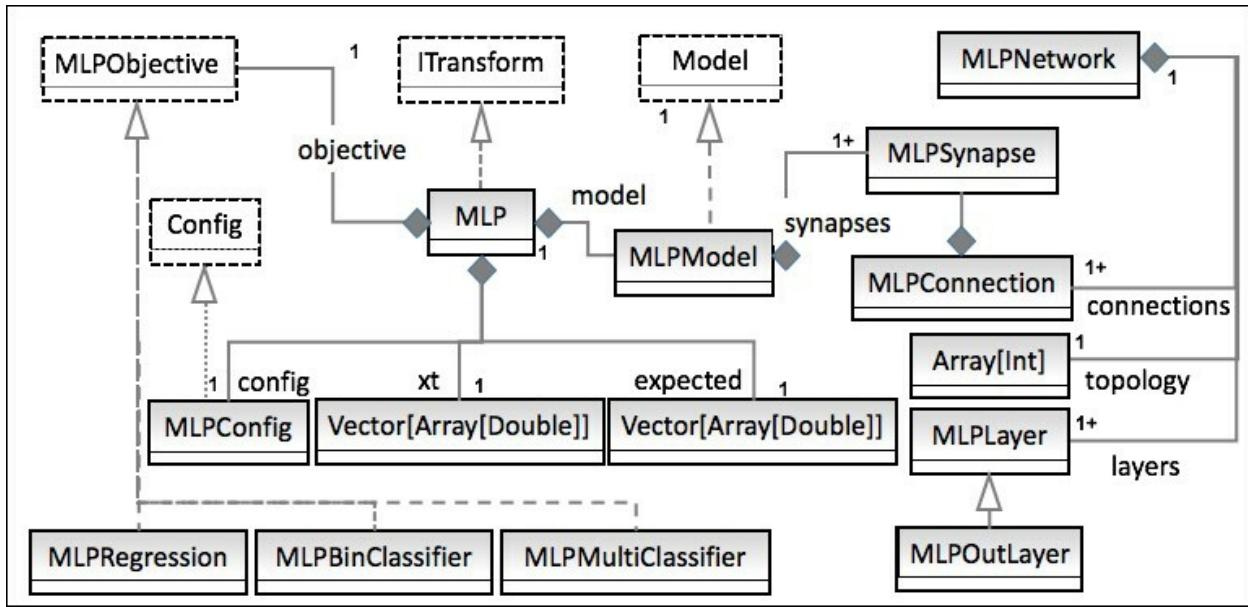
- A *one-variate regression* has one output node whose value is a real number $[0, 1]$
- A *multivariate regression* with n variables has n real output nodes
- A *binary classification* has one binary output node $\{0, 1\}$ or $\{-1, +1\}$
- A *multinomial or K-class classification* has K binary output nodes

Design

The implementation of the MLP classifier follows the same pattern as previous classifiers (refer to the *Design template for classifiers* section in the *Appendix*):

- A connectionist network `MLPNetwork` composed of a layer of neurons of the type `MLPLayer`, connected by synapses of type `MLPSynapse` contained by a connector of type `MLPConnection`.
- All the configuration parameters are encapsulated into a single configuration class, `MLPConfig`.
- A model `MLPModel` that consists of a sequence of connection synapses
- The multilayer perceptron class, `MLP`, is implemented as a data transformation `ITransform` for which the model is automatically extracted from a training set with labels.
- The multilayer perceptron class, `MLP`, takes four parameters: a configuration, a features-set or time series `xt` of type `Vector[Array[Double]]`, a labeled dataset `expected` of type `Vector[Array[Double]]`, and an activation function of type `Function1[Double, Double]`.

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 network and its components. The UML diagram omits the helper traits or classes such as `Monitor` or Apache Commons Math components.

Configuration

The `MLPConfig` configuration of the multilayer perceptron consists of the definition of the network configuration with its hidden layers, the learning and training parameters, and the activation function:

```
case class MLPConfig(  
    alpha: Double, //1  
    eta: Double,  
    numEpochs: Int,  
    eps: Double,  
    activation: Double => Double) extends Config {  
}
```

For the sake of readability, the name of the configuration parameters matches the symbols defined in the mathematical formulation (line 1):

- `alpha`: This is the momentum factor a , that smooth the computation of the gradient of the weights for online training. The momentum factor is used in the mathematical expression M10 in *Step 2 – Error backpropagation* under *Training epoch*.
- `eta`: This is the learning rate, γ , used in the gradient descent. The gradient descent updates the weights or parameters of a model by the quantity $\eta \cdot (predicted - expected).input$ as described in the mathematical formulation M9 in the *Step 2 – Error backpropagation* section under *The training epoch*. The gradient descent was introduced in *Training the model* section under *Let's kick the tires of [Chapter 1, Getting Started](#)*.
- `numEpochs`: This is the maximum number of epochs (or cycles or episodes) allowed for training the neural network. An epoch is the execution of the error backpropagation across the entire observations set.
- `eps`: This is the convergence criteria used as an exit condition for the training of the neural network when `error < eps`.
- `activation`: This is the activation function used for nonlinear regression applied to hidden layers. The default function is the sigmoid (or hyperbolic tangent) introduced for the logistic regression (refer to the

Logistic function section of [Chapter 9](#), Regression and Regularization).

Network components

The training and classification of MLP model relies on the network architecture. The `MLPNetwork` class is responsible for creating and managing the different components and the topology of the network: layers, synapses, and connections.

Network topology

The instantiation of the `MLPNetwork` class requires a minimum set of two parameters with an instance of the model as an optional third argument (line 2):

- A MLP execution configuration, `config`, introduced in the previous section.
- A `topology` defined as an array of the number of nodes for each layer: input, hidden, and output layers
- A `model` with type `Option[MLPModel]` if it has already been generated through training, `None` otherwise
- An implicit reference to the operating `mode` of the MLP:

```
class MLPNetwork(  
    config: MLPConfig,  
    topology: Array[Int],  
    model: Option[MLPModel] =None) (implicit mode: MLPMode) { /  
  
    val layers = topology.indices.map( n =>if (topology.length  
    val connections = zipWithShift1(layers).map{  
        case(src,dst) => new MLPConnection(config, src,dst,model)  
    } //4  
  
    def trainEpoch(x: Array[Double], y: Array[Double]): Double/  
    def getModel: MLPModel //6  
    def predict(x: Array[Double]): Array[Double] //7  
}
```

An MLP network has the following components, derived from the topology array:

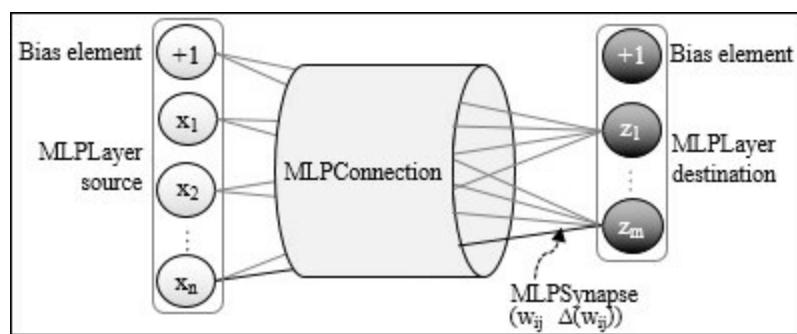
- Multiple layers of the `MLPLayers` class (line 3)
- Multiple connections of the `MLPConnection` class (line 4)

The topology is defined as an array of number of nodes per layer, starting with the input nodes. The array indices follow the forward path within the network. The size of the input layer is automatically generated from the observations as the size of the features vector. The size of the output layer is automatically extracted from the size of the output vector (line 3).

The constructor for `MLPNetwork` creates a sequence of layers by assigning and ordering an `MLPLayer` instance to each entry in the topology (line 3). The constructor creates *number of layers - 1* interlayer connections of type `MLPConnection` (line 4). The `zipWithShift1` method of the `TSeries` object zips a time series with its duplicated shift by one element.

The `trainEpoch` method (line 5) implement the training of this network for a single pass of the entire set of observations (refer to the *Putting it all together* section under *Training epoch*). The `getModel` method retrieves the model (synapses) generated through training of the MLP (line 6). The `predict` method computes the output value generated from the network using the forward propagation algorithm (line 7).

The following diagram visualizes the interaction between the different components of a model: `MLPLayer`, `MLPConnection`, and `MLPSynapse`:



Core components of the MLP network

Input and hidden layers

First, let's start with the definition of the layer class, `MLPLayer`: it is fully specified by its position (or rank) `id` in the network and the number of nodes, `numNodes`, it contains:

```
class MLPLayer(
    val id: Int,
    val numNodes: Int,
    val activation: Double => Double) //8
    (implicit mode: MLPMode) { //9

    val output = Array.fill(numNodes)(1.0) //10

    def setOutput(xt: Array[Double]): Unit =
        xt.copyToArray(output, 1) //11
    def activate(x: Double): Double = activation(x) //12
    def delta(loss: Array[Double], .
              srcOut: Array[Double],
              synapses: MLPConnSynapses): Delta //13
    def setInput(_x: Array[Double]): Unit //14
}
```

The `id` parameter is the order of the layer (0 for input, 1 for the first hidden layer, and $n-1$ for the output layer) in the network. The `numNodes` value is the number of elements or nodes, including the bias element, in this layer. The `activation` function is the last argument of the layer given a user-defined mode or objective (line 8). The operating `mode` must be provided implicitly prior to the instantiation of a layer (line 9).

The `output` vector for the layer is an uninitialized array of values updated during the forward propagation. It initializes the bias value with the value, *1.0* (line 9). The matrix of difference of weights, `delta`, associated with the `output` vector (line 10), is updated through the error backpropagation algorithm, described in *Step 2 – Error propagation* section under *Training epoch*. The `setOutput` method initializes the output values for the output and hidden layers during the backpropagation of the error on the output of the network's (*expected – predicted*) values (line 11).

The `activate` method invokes the activation method (*$\tan h$, sigmoid, ...*) defined in the configuration (line 12).

The `delta` method computes the correction to be applied to each weight or

synapse, as described in the *Step 2 – Error propagation* section under *Training epoch* (line 13).

The `setInput` method initializes the `output` values for the nodes of the input and hidden layers, except the bias element, with the value `x` (line 14). The method is invoked during the forward propagation of the input values:

```
def setInput(x: DblVec): Unit =  
    x.copyToArray(output, output.length -x.length)
```

The methods of the `MLPLayer` class for the input and hidden layers are overridden for the output layer of the `MLPOutLayer` type.

Output layer

Contrary to the hidden layers, the output layer does not have either an activation function or a bias element. The `MLPOutLayer` class has the following arguments: the order `id` in the network (as last layer of the network) and the number, `numNodes`, of output or nodes (line 15):

```
class MLPOutLayer(  
    id: Int,  
    numNodes: Int)(implicit obj: MLP.MLPObjective) //15  
extends MLPLayer(id, numNodes, identity) {  
  
    override def numNonBias: Int = numNodes  
    override def setOutput(xt: Array[Double]): Unit =  
        obj(xt).copyToArray(output)  
  
    override def delta(loss: Array[Double,  
        srcOut: Array[Double,  
        synapses: MLPConnSynapses]): Delta  
    ...  
}
```

The `numNonBias` method returns the actual number of output values from the network. The implementation of the `delta` method is described in the *Step 2 – Error propagation* section under *Training epoch*.

Synapses

A synapse is defined as a pair of real (floating point) 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), $?w_{ij}$

Its type is defined as `MLPSynapse`, as shown here:

```
type MLPSynapse = (Double, Double)
```

Connections

The connections are instantiated by selecting two consecutive layers of index n (with respect to $n+1$) as the source (with respect to destination). A connection between two consecutive layers implements the matrix of synapses, as $(w_{ij}, ?w_{ij})$ pairs. The `MLPConnection` instance is created with the following parameters (line 16):

- Configuration parameters, `config`
- The source layer, sometimes known as the ingress layer, `src`
- The destination (or egress) layer, `dst`
- A reference to the `model` if it has been already generated through training or `None` if the model has not been trained
- An implicitly defined operating mode or objective, `mode`

The `MLPConnection` class is defined as follows:

```
type MLPConnSynapses = Array[Array[MLPSynapse]]  
  
class MLPConnection(config: MLPConfig,  
                      src: MLPLayer,  
                      dst: MLPLayer,  
                      model: Option[MLPModel]) //16  
                      (implicit mode: MLP.MLPOjective) {  
  
  var synapses: MLPConnSynapses //17  
  def connectionForwardPropagation: Unit //18  
  def connectionBackpropagation(delta: Delta): Delta //19  
  ...  
}
```

The last step in the initialization of the MLP algorithm is the selection of the initial (usually random) values of the weights (synapses) (line 17).

The `MLPConnection` method implements the forward propagation of weights computation for this connection, `connectionForwardPropagation` (line 18), and the backward propagation of the delta error during training, `connectionBackpropagation` (line 19). These methods are described in the next section related to the training of the MLP model.

Weights initialization

The initialization values for the weights depends if it 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. [10:7]

Our implementation relies on the sigmoid activation function and uses *range* $[0, \text{BETA}/\sqrt{\text{numOutputs} + 1}]$ (line 20). However, the user can select a different range for random values, such as $[-r, +r]$ for the $\tan h$ activation function. The weight for the bias is obviously defined as $w_0 = +1$, and its weight adjustment is initialized as $?w_0 = 0$.

```
var synapses: MLPConnSynapses =
if(!model.isDefined) {
    val max = BETA/sqrt(src.output.length+1.0) //20
    Array.fill(dst.numNonBias) (
        Array.fill(src.numNodes) ((Random.nextDouble*max, 0.00)))
    )
}
else model.get.synapses(src.id) //21
```

The connection derives its weights or synapses from a model (line 21) if it has already been created through training.

Model

The `MLPNetwork` class defines the topological model of the multilayer perceptron. The weights or synapses are the attributes of the model of type `MLPModel` generated through training:

```
case class MLPModel (synapses: Vector[MLPConnSynapses])  
  extends Model[MLPModel]
```

The model can be stored in simple key-value pair JSON, CVS, or sequence file.

Tip

Encapsulation and the model factory:

The network components (connections, layers, and synapses) are implemented as top-level classes for clarity's 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. A factory design pattern would be perfectly appropriate to instantiate an `MLPNetwork` instance dynamically [10:8].

Once initialized, the MLP model is ready to be trained using a combination of forward propagation, output error backpropagation, and iterative adjustment of weights and gradients of weights.

Problem types (modes)

There are three distinct type of problems or operating modes associated with the multilayer perceptron:

- Binomial classification (binary) with two classes and one output
- Multinomial classification (multiclass) with n classes and outputs
- Regression

Each operating mode has distinctive error, hidden layer, and output layer activation functions, as illustrated in the following table:

Operating modes	Error function	Hidden layer activation function	Output layer activation function
Binomial classification	Cross-entropy	Sigmoid	Sigmoid
Multinomial classification	Sum of squares error or mean squared error	Sigmoid	Softmax
Regression	Sum of squares error or mean squared error	Sigmoid	Linear

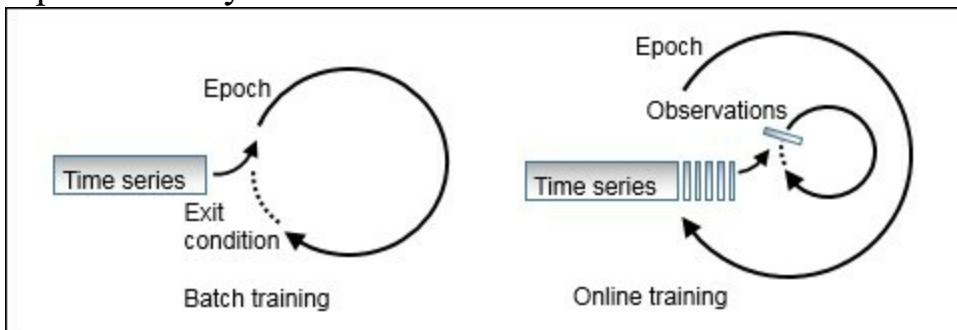
Operating modes of the multilayer perceptron

The cross-entropy is described by expressions M6 and M7, and the softmax uses formula M8 in the *Step 1 – Input forward propagation* section under *Training epoch*.

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

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 [10:9]. Techniques such as the momentum factor, which was described earlier, or any adaptive learning scheme, improves the performance and accuracy of the online training methodology.

The online training strategy is applied to all the test cases of this chapter.

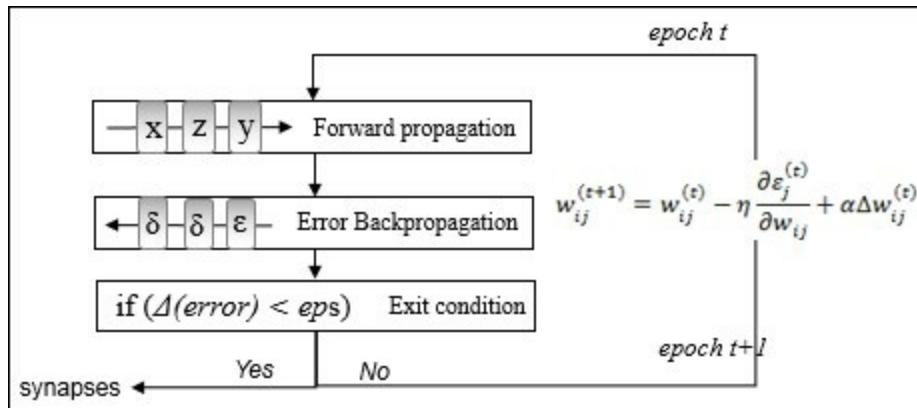
Training epoch

The training of the model processes the training observations iteratively multiple times. A training cycle or iteration is known as an **epoch**. The order of observations is shuffled for each epoch. The four steps of the training cycle are as follows:

1. Forward propagation of the input value for a specific epoch.
2. Computation and backpropagation of the output error.
3. Evaluation of the convergence criteria and exit if the criteria are 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 (predicted values) backward to the input layer through the hidden layers, if any are defined.

The three steps of the training cycle or training epoch is summarized in the following diagram:



Iterative implementation of the training for MLP

Let's apply the three steps of a training epoch in the `trainEpoch` method of the `MLPNetwork` class using a simple `foreach` Scala higher order function, as shown here:

```

def trainEpoch(
    x: Array[Double],
    y: Array[Double]): Double = {

    layers.head.setInput(x) //22
    connections.foreach( _.connectionForwardPropagation) //23

    val err = mode.error(y, layers.last.output)
    val bckIterator = connections.reverseIterator

    val z = y.zip(layers.last.output).map {
        case (a, b) => diff(a,b)
    }
    var delta = Delta(z) //24
    bckIterator.foreach( iter =>
        delta = iter.connectionBackpropagation(delta)
    ) //25
    err //26
}

```

You can certainly recognize the first two stages of the training cycle: forward propagation of the input and the backpropagation of the error of the online training of a single epoch.

The execution of the training of the network for one epoch, `trainEpoch`, initializes the input layer with observations `x` (line 22). The input values are propagated through the network by invoking `connectionForwardPropagation` for each connection (line 23). The `delta` error is initialized from the values in the `output` layer and the expected values, `y` (line 24).

The training method iterates through the connections backward to propagate the error through each connection by invoking the `connectionBackpropagation` method on the backward iterator, `bckIterator` (line 25). Finally, the training method returns the cumulative error, mean square error, or cross-entropy, according to the operating mode (line 26).

This approach is not that different than the beta (or backward) pass in the hidden Markov model, covered in the *Beta (backward) pass* section in [Chapter 7, Sequential Data Models](#).

Let's look at the implementation of the forward and backward propagation

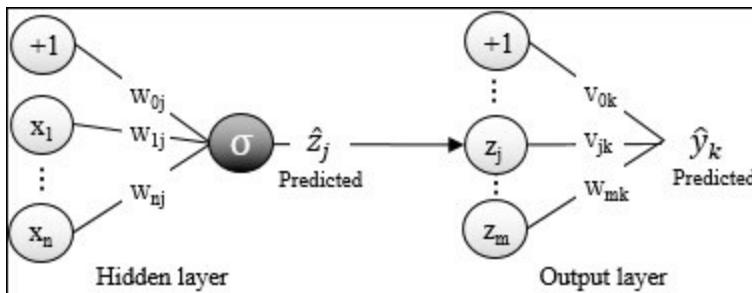
algorithm for each type of connection

- Input or hidden layer to hidden layer
- Hidden layer to output layer

Step 1 – input forward propagation

As mentioned earlier, the output values of a hidden layer are computed as the sigmoid or hyperbolic tangent of the dot product of the weights w_{ij} and the input values x_i .

In the following diagram, the MLP algorithm computes the linear product of the weights w_{ij} and input x_i for the hidden layer. The product is then processed by the activation function s (sigmoid or hyperbolic tangent). The output values, z_j , are then combined with the weights, v_{ij} , of the output layer, which doesn't have an activation function:



Distribution of weights in MLP hidden and output layers

The mathematical formulation of the output of a neuron, j , is defined as a composition of the activation function and the dot product of the weights w_{ij} and input values x_i .

M3: Computation (or prediction) of the output layer from the output values, z_j , of the preceding hidden layer and the weights, v_{kj} :

$$\tilde{y}_k = v_{0j} + \sum_{j=1}^m v_{kj} z_j$$

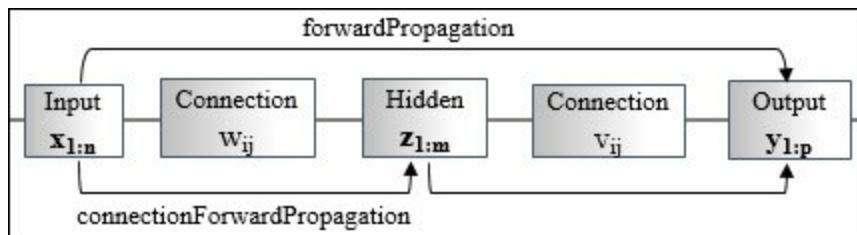
M4: Estimation of the output values for binary classification with an activation function s :

$$\tilde{z}_j = \sigma\left(w_{0j} + \sum_{i=1}^m w_{ij}x_i\right) = \frac{1}{1 + e^{-w_{0j} - \sum_{i=1}^m w_{ij}x_i}}$$

As seen in the network architecture section, the output values for the multinomial (or multiclass) classification with more than two classes are normalized using an exponential function described in the following softmax section.

Computational flow

The computation of the output values y from the input x is known as the input forward propagation. For the sake of simplicity, we represent the forward propagation between layers with the following block diagram:



A computation model of input forward propagation

This diagram conveniently illustrates a computational model for the input forward propagation, as the programmatic relation between the source and destination layers and their connectivity. The input x is propagated forward through each connection.

The `connectionForwardPropagation` method computes the dot product of the weights and the input values, and applies the activation function in the case of hidden layers, for each connection. Therefore, it is a member of the `MLPConnection` class.

The forward propagation of input values across the entire network is

managed by the `MLP` algorithm itself. The forward propagation of the input value is used in the classification or prediction, $y = f(x)$. It depends on the value weights, w_{ij} and v_{ij} , that need to be estimated through training. As you may have guessed, the weights define the model of a neural network similar to the regression models.

Let's look at the `connectionForwardPropagation` method of the `MLPConnection` class:

```
def connectionForwardPropagation: Unit = {
    val out = synapses.map(x => {
        val dotProd = margin(src.output, x.map(_.-_1)) //27
        dst.activate(dotProd) //28
    })
    dst.setOutput(out) //29
}
```

The first step is to compute the dot product, `dotProd` (refer to the *Time series in Scala* section of [Chapter 3, Data Pre-Processing](#)) of the output `out` of the current source layer for this connection, and the `synapses` (weights) (line 27). The activation function is computed by applying the `activate` method of the destination layer to the dot product (line 28). Finally, the computed value, `out` is used to initialize the output for the destination layer (line 29).

Error functions

As mentioned in the *Problem types (modes)* section, there are two approaches to computing the error or loss on the output values:

- SSE between the expected and predicted output values as defined in the following mathematical expression, M5
- Cross-entropy of the expected and predicted values described in the M6 and M7 mathematical formulas

M5: Sum of squared errors, e , and mean square error for predicted value $\sim y$ and expected values of y :

$$\mathcal{E} = \frac{1}{2} \sum_{i=0}^{n-1} \sum_{j=0}^{k-1} (y_{ij} - \tilde{y}_{ij})^2 \quad \bar{\mathcal{E}} = \frac{\mathcal{E}}{n}$$

M6: Cross-entropy for a single output value y :

$$ce = -\sum_{i=0}^{n-1} \left\{ y_i \log(\tilde{y}_i) + (1-y_i) \cdot \log(1-\tilde{y}_i) \right\}$$

M7: Cross-entropy for a multivariable output vector y :

$$ce = -\sum_{i=0}^{n-1} \sum_{j=0}^{k-1} \tilde{y}_{ij} \log(y_{ij})$$

The sum of square errors and mean squared error functions have been described in the *Time series in Scala* section of [Chapter 3, Data Pre-processing](#).

The `crossEntropy` method of the `Loss` object for a single variable is implemented as follows:

```
def crossEntropy(x: Double, y: Double): Double =
  -(x * log(y) + (1.0 - x) * log(1.0 - y))
```

The computation of the cross-entropy for multiple variable features as a signature similar to the single variable case is as follows:

```
def crossEntropy(
  xt: Array[Double],
  yt: Array[Double]): Double =
  yt.zip(xt).aggregate(0.0) ({case (s, (y, x)) => s - y * log(x)}, _ + _)
```

Operating modes

In the *Network architecture* section, you learned that the structure of the output layer depends on the type of problems that need to be resolved, also known as operating modes. Let's encapsulate the different operating modes

(binomial, multinomial classification, and regression) into a class hierarchy implementing the `MLPMode` trait.

The following code snippet implements the `MLPMode` trait:

```
trait MLPMode {
    def apply(output: Array[Double]): Array[Double] //30
    def error(
        labels: Array[Double],
        output: Array[Double]): Double = mse(labels, output)//31
}
```

The `MLPMode` trait has two methods. Their implementation is specific to the type of problem:

- `apply` is the transformation applied to the output values
- `error` is the computation of cumulative errors for the entire observations set

The `apply` method applies a transformation in the output layer as described in the last column of the operating modes table (line 30). The `error` function computes the cumulative error or loss in the output layer for all the observations, as described in the first column of the operating modes table (line 31).

The transformation in the output layer of the binomial (two-class) classifier, `MLPBinClassifier`, consists of applying the `sigmoid` function to each `output` value (line 32). The cumulative `error` is computed as the cross-entropy of the expected output, `labels`, and the predicted output (line 33):

```
class MLPBinClassifier extends MLPMode {
    override def apply(output: Array[Double]): Array[Double] =
        output.map(sigmoid(_)) //32

    override def error(
        labels: Array[Double],
        output: Array[Double]): Double =
        crossEntropy(labels.head, output.head) //33
}
```

The regression mode for the multilayer perceptron is defined according to the

operating modes table in the *Problem types (modes)* section:

```
class MLPRegression extends MLPMode {  
    override def apply(output: Array[Double]): Array[Double] =  
        output  
}
```

The multinomial classifier mode is defined by the `MLPMultiClassifier` class. It uses the `softmax` method to boost the `output` with the highest value, as shown here:

```
class MLPMultiClassifier extends MLPMode {  
    override def apply(output: Array[Double]):Array[Double] =  
        softmax(output)  
}
```

The `softmax` method is applied to the actual `output` value, not the bias. Therefore, the first node, $y(0)=+1$, has to be dropped before applying the `softmax` normalization.

Softmax

With a classification problem with K classes ($K > 2$), the output must be converted into a probability, $[0, 1]$. For problems that require many 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 [10:10].

M8: The softmax formula for multinomial ($K > 2$) classification is as follows:

$$\hat{y} = \frac{e^{-\hat{y}_k}}{\sum_i e^{-\hat{y}_i}}$$

Here is the simple implementation of the `softmax` method of the `MLPMultiClassifier` class:

```
def softmax(y: Array[Double]): Array[Double] = {  
    val softmaxValues = new Array[Double](y.length)  
    val expY = y.map(exp(_)) //34
```

```

    val expYSum = expY.sum //35
    expY.map(_ /expYSum).copyToArray(softmaxValues,1) //36
    softmaxValues
}

```

The `softmax` method implements the mathematical expression M8. First, the method computes the exponential values `expY` of the output values (line 34). The exponentially transformed outputs are then normalized by their sum, `expYSum` (line 35), to generate the array of `softmaxValues` output (line 36). Once again, there is no need to update the bias element, $y(0)$.

The second step in the training phase is to define and initialize the matrix of error delta values to be backpropogated between layers from the output layer back to the input layer.

Step 2 – error backpropagation

Error backpropagation is an algorithm that estimates the error for the hidden layer to compute the change in weights of the network. It takes the sum of squared errors on the output as input.

Tip

Convention for computing the cumulative error

Some authors refer to backpropagation as a training methodology for an MLP, which applies the gradient descent to the output error defined as either the sum of square errors, or the mean squared error for multinomial classification or regression. In this chapter, we keep the narrower definition of backpropagation as the backward computation of the sum of square errors.

Weights adjustment

The connection weights, \mathbf{v} and \mathbf{w} , are adjusted by computing the sum of the derivatives of the error, over the weights scaled with a learning factor. The gradient of weights is then used to compute the error of the output of the source layer [10:11].

The simplest algorithm to update the weights is the gradient descent [10:12]. The batch gradient descent was introduced in the *Training the model* section in the *Let's kick the tires* section of [Chapter 1, Getting Started](#).

The gradient descent is a very simple and robust algorithm. However, it can be slower in converging toward a global minimum than the conjugate gradient or the quasi-Newton method (refer to the *Summary of optimization techniques* section in the *Appendix*).

There are several methods available to speed up the convergence of the gradient descent toward a minimum: momentum factor and adaptive learning coefficient [10:13].

Large variations of the weights during training increases the number of epochs required for the model (connection weights) to converge. This is particularly true for a training strategy known as online training. The training strategies are discussed in the next section. The momentum factor, a , is used for the remaining section of the chapter.

Note

M9: Learning rate

The computation of neural network weights using gradient descent is as follows:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \eta \frac{\partial \mathcal{E}_j^{(t)}}{\partial w_{ij}}$$

Note

M10: Learning rate and momentum factor

The computation of neural network weights using gradient descent method with momentum coefficient, a , is as follows:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \eta \frac{\partial \mathcal{E}_j^{(t)}}{\partial w_{ij}} + \alpha \Delta w_{ij}^{(t)}$$

Note

The simplest version of the gradient descent algorithm (M9) is selected by simply setting the momentum factor, a , to zero in the generic mathematical expression (M10).

Error propagation

The objective of the training of a perceptron is to minimize the loss or cumulative errors for the entire input observations as either the sum of squared errors or the cross-entropy as computed at the output layer. The error, e_k , for each output neuron, y_k , is computed as the difference between a predicted output value and label output value. The error cannot compute the output values of the hidden layers, z_j , because the label values for those layers are unknown:

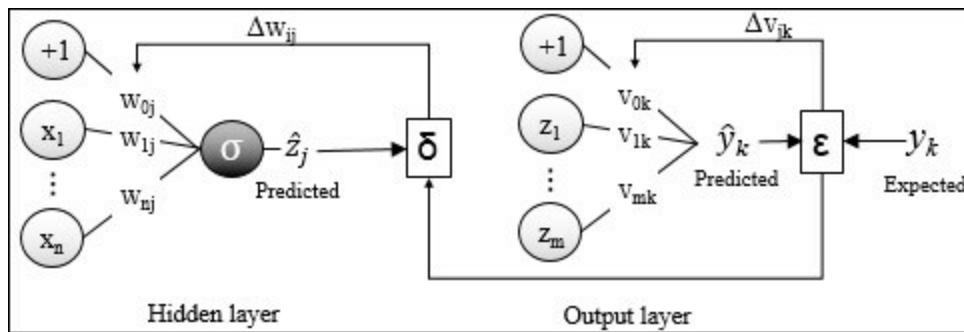


Illustration of the backpropagation algorithm

In the case of the sum of squared errors, the partial derivative of the cumulative error over each weight of the output layer is computed as the composition of the derivative of the square function, and the derivative of the dot product of weights and the input z .

As mentioned earlier, the computation of the partial derivative of the error

over the weights of the hidden layer is a bit tricky. Fortunately, the mathematical expression for the partial derivative can be written as the product of three partial derivatives:

- Derivative of the cumulative error, e , 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, s , is $s(1 - s)$
- Derivative of the output of the hidden layer, z_j , over the weights, w_{ij}

The decomposition of the partial derivative produces the following formulas for updating the synapses weights for the output and hidden neurons by propagating the error (or loss) e .

Note

M11: Output weights adjustment

Computation of delta, d , and weight adjustment, Δv , for the output layer with the predicted value \tilde{y} and expected value y and output z of the hidden layer:

$$\delta_{ih} = (\tilde{y}_i - y_i) \cdot z_h \quad \Delta v_{ih} = -\delta_{ih}$$

M12: Hidden weights adjustment

Computation of delta, d , and weight adjustment, Δw , for the hidden layer with the predicted value \tilde{y} and expected value y , output z of the hidden layer and the input value x :

$$\delta_{hi} = \sum_{j=0}^{k-1} \left\{ (\tilde{y}_j - y_j) \cdot v_{jh} \right\} \cdot z_h (1 - z_h) \cdot x_i \quad \Delta w_{hi} = -\eta \delta_{hi}$$

The matrix, dij , is defined by the `delta` matrix in the `Delta` class. It contains the basic parameters to be passed between layers, traversing the network from the output layer back to the input layer. The parameters are as follows:

- Initial `loss` or error computed at the output layer
- Matrix of the `delta` values from the current connection
- Weights or `synapses` of the downstream connection (or connection between the destination layer and the following layer)

```
case class Delta(
  loss: Array[Double],
  delta: DblMatrix = Array.empty[Array[Double]],
  synapses: MLPConnSynapses = Array.empty[Array[MLPSynapse]] )
)
```

The definition of the data class Delta follows:

The first instance of the `Delta` class is generated for the output layer using the expected values, y , propagated to the preceding hidden layer in the `MLPNetwork.trainEpoch` method (line 24):

```
val diff = (x: Double, y: Double) => x - y
Delta(zipToArray(y, layers.last.output)(diff))
```

The mathematical expression $M11$ is implemented by the `delta` method of the `MLPOutLayer` class:

```
def delta(error: Array[Double],
         srcOut: Array[Double],
         synapses: MLPConnSynapses): Delta = {
  val deltaMatrix = ArrayBuffer[Array[Double]]() //34
  val deltaValues = error./:(deltaMatrix) { (m, l) =>
    m += srcOut.map(_ * l)
  } //35
  new Delta(error, deltaValues.toArray, synapses) //36
}
```

The method generates the matrix of delta values associated with the output layer (line 34). The formula $M11$ is implemented by the fold over the output value, `srcOut` (line 35). The new delta instance is returned to the `trainEpoch` method of `MLPNetwork` and back-propagated to the preceding hidden layer (line 36).

The `delta` method of the `MLPLayer` class implements the mathematical

expression, M12:

```
def delta(
    prevDelta: ArrayBuffer[Double],
    srcOut: ArrayBuffer[Double],
    synapses: MLPConnSynapses): Delta = {

    val deltaMatrix = ArrayBuffer[(Double, ArrayBuffer[Double])]()
    val weights = transpose(synapses.map(_.map(_._1))).drop(1)//37

    val (loss, nDelta) = output.drop(1)
        .zipWithIndex./:(deltaMatrix){ // 38
            case (m, (zh, n)) => {
                val newDelta = margin(oldDelta, weights(n))*zh*(1.0-zh)
                m += (newDelta, srcOut.map(_ * newdelta) )
            }
        }.unzip

    new Delta(loss, nDelta)//39
}
```

The implementation of the `delta` method is like the `MLPOutLayer.delta` method. It extracts the weights, v , from the output layer through transposition (line 37). The values of the delta matrix in the hidden connection are computed by applying the formula, M12 (line 38). The new delta instance is returned to the `trainEpoch` method (line 39) to be propagated to the preceding hidden layer if one exists.

The computational model

The computational model for the error backpropagation algorithm is very similar to the forward propagation of the input. The main difference is that the propagation of d (delta) is performed from the output layer to the input layer. The following diagram illustrates the computational model of the backpropagation in the case of two hidden layers, z^s and z^t :

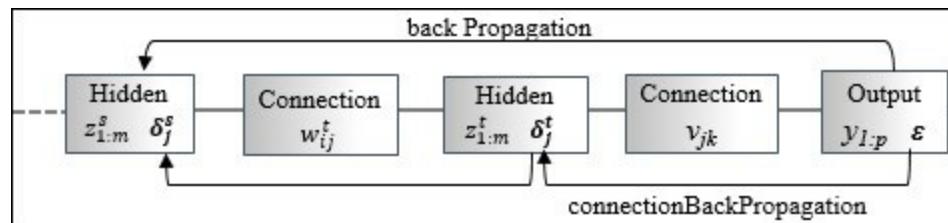


Illustration of the backpropagation of error, delta

The `connectionBackPropagation` method propagates the error back from the output layer or one of the hidden layers to the preceding layer. It is a member of the `MLPConnection` class. The backpropagation of the output error across the entire network is managed by the `MLP` class.

It implements the two sets of equations where `synapses(j)(i)._1` are the weights, w_{ji} , `dst.delta` is the vector of error derivative in the destination layer, and `src.delta` is the error derivative on the outputs in the source layer, as shown here:

```
def connectionBackpropagation(delta: Delta): Delta = { //40
    val inSynapses = //41
        if(delta.synapses.length > 0) delta.synapses
        else synapses

    val cDelta = dst.delta(delta.loss, src.output, inSynapses)//4
    synapses = synapses.indices.map( j =>//43
        synapses(j).indices.map(i =>{
            val ndw = config.eta * cDelta.delta(j)(i)
            val (w, dw) = synapses(j)(i)
            (w + ndw - config.alpha*dw, ndw)
        })
    }
    Delta(cDelta, synapses)
}
```

The `connectionBackPropagation` method takes the `delta` associated to the destination (output) layer as an argument (line 40). The output layer is the last layer of the network, therefore the synapses for the following connection is defined as an empty matrix, of length zero (line 41). The method computes the new delta matrix for the hidden layer using the error, `delta.loss`, and output from the source layer, `src.output` (line 42). The weights (synapses) are updated using the gradient descent with a momentum factor, as in the mathematical expression M10 (line 43).

Note

The adjustable learning rate:

The computation of the new weights of a connection for each new epoch can be further improved by making the learning adjustable.

Step 3 – exit condition

The convergence criterion consists of evaluating the cumulative error (or loss) relevant to the operating mode (or problem) against a predefined convergence, `eps`. The cumulative error is computed using either the sum of squares error formula (M5) or the cross-entropy formula (M6 and M7). An alternative approach is to compute the difference of the cumulative error between two consecutive epochs and apply the convergence criteria, `eps`, as the exit condition.

Putting it all together

The `MLP` class is defined as a data transformation of type `ITransform` using a `model` implicitly generated from a training set, `xt`, as described in the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#) (line 44).

The MLP algorithm takes the following parameters:

- `config`: The configuration of the algorithm
- `hidden`: Array of the size of the hidden layers, if any
- `xt`: The time series of features used to train the model
- `expected`: The labeled output values for training purposes
- `mode`: The implicit operating mode, objective of the algorithm
- `f`: The implicit conversion from a feature from type `T` to `Double`

The type `v` of the output of the prediction or classification method, `|>`, of this implicit transform is `Array[Double]` (line 45):

```
type v = Array[Double] //45

class MLP[T: ToDouble] (
  config: MLPConfig,
  hidden: Array[Int] = Array.empty[Int],
  xt: Vector[Array[T]],
  expected: Vector[Array[T]]) (implicit mode: MLPMode)
```

```

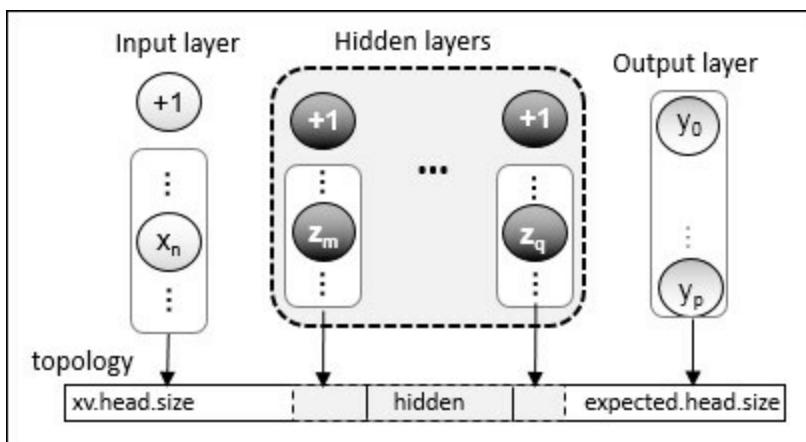
extends ITransform[Array[T], V] with Monitor[Double] { //44

  lazy val topology = if(hidden.length == 0)
    Array[Int](xt.head.length, expected.head.length)
  else Array[Int](xt.head.length) ++
    hidden ++
    Array[Int](expected.head.length) //46

  val model: Option[MLPModel] = train
  def train: Option[MLPModel] //47
  override def |> : PartialFunction[Array[T], Try[V]]
}

```

The topology is created from the input variables, `xt`, the `expected` values, and the configuration of `hidden` layers, if any (line 46). The generation of the topology from parameters of the `MLPNetwork` class is illustrated in the following diagram:



Topology encoding for multilayer perceptron

For instance, the `topology` of a neural network with three input variables, one output variable, and two hidden layers of three neurons each is specified as `Array[Int](4, 3, 3, 1)`. The model is generated through training by invoking the `train` method (line 47). Finally, the operator `|>` of `ITransform` trait is used for classification, prediction, or regression, depending on the selected operating mode (line 48).

Training and classification

Once the training cycle or epoch is defined, it is merely a matter of defining and implementing a strategy to create a model using a sequence of data or time series.

Regularization

There are two approaches to find the most appropriate network architecture for a given classification or regression problem:

- **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 *Ridge regression* section in [Chapter 9, 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 [10:14].

Model generation

The `MLPModel` instance is created (trained) during the instantiation of the multilayer perceptron. The constructor iterates through the training cycles (or epochs) over all the data points of the time series `xt`, until the cumulative is smaller than the convergence criteria, `eps`, as in the following code:

```

def train: Option[MLPModel] = {
  val network = new MLPNetwork(config, topology) //48
  val zi = xt.zip(expected) // 49
  var prevErr = Double.MaxValue

  (0 until config.numEpochs).find( n => { //50
    val err = fisherYates(xt.size)
    .map(zi(_))
    .map{ case(x, e) => {
      val z = x.map(implicitly[ToDouble[T]].apply(_))
      network.trainEpoch(z, e)
    } }.sum/st.size //51
  }

  val diffErr = abs(err - prevErr)
  prevErr = err
  diffErr < config.eps //52
}
.map(_ => network.getModel)
}

```

The `train` method instantiates an MLP network using the configuration and the `topology` as input (line 48). The method executes multiple epochs until either the gradient descent with momentum converges or the maximum number of allowed iterations is reached (line 50). At each epoch, the method shuffles the input values and labels using the *Fisher-Yates* algorithm, invokes the `MLPNetwork.trainEpoch` method, and computes the cumulative error, `err` (line 51). This implementation compares the change in the error value, `err`, against the convergence criteria, `eps`, as the exit condition (line 52).

Note

Tail recursive training of MLP:

The training of the multilayer is implemented as an iterative process. It can be easily substituted with a tail recursion using weights and cumulative errors as the argument of the recursion.

Lazy views are used to reduce the unnecessary creation of objects (line 49).

Tip

Exit condition:

In this implementation, the training initializes the model as none, if it does not converge before the maximum number of epochs has been reached. An alternative would be to generate a model even in the case of non-convergence and add an accuracy metric to the model as in our implementation of the support vector machine (refer to the *Training* section under *Support vector classifier* in [Chapter 12, Kernel Models and SVM](#)).

Once the model is created during the instantiation of the multilayer perceptron, it is available to predict or classify the class of a new observation.

Fast Fisher-Yates shuffle

The *Step 5—Implementing the classifier* section under *Let's kick the tires in Chapter 1, Getting Started*, describes a home-grown shuffling algorithm as an alternative to the `scala.util.Random.shuffle` method of the Scala standard library. This section describes an alternative shuffling mechanism known as the *Fisher-Yates* shuffling algorithm:

```
def fisherYates(n: Int): IndexedSeq[Int] = {  
  
    def fisherYates(seq: Seq[Int]): IndexedSeq[Int] = {  
        setSeed(System.currentTimeMillis)  
        (0 until seq.size).map(i => {  
            var randomIdx: Int = i + nextInt(seq.size-i)//53  
  
            seq(randomIdx) ^= seq(i)      //54  
            seq(i) = seq(randomIdx) ^ seq(i)  
            seq(randomIdx) ^= (seq(i))  
            seq(i)  
        })  
    }  
  
    if( n <= 0) Array.empty[Int]  
    else fisherYates((0 until n).toArray) //55  
}
```

The Fisher-Yates algorithm consists of creating an ordered sequence of integers (line 55), and swaps each integer with another integer, randomly selected from the remaining of the initial sequence (line 52). This

implementation is particularly fast because the integers are swapped in place using a bit operator, also known as **bitwise swap** (line 54).

Tip

Tail recursive implementation of Fisher-Yates:

The Fisher-Yates shuffling algorithm can be implemented using a tail recursion instead of an iteration.

Prediction

The data transformation `|>` implements the runtime classification/prediction. It returns the predicted value, normalized as a probability if the model was successfully trained; `None`, otherwise. The method invokes the forward prediction function of `MLPNetwork` (line 53):

```
override def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T] if isModel && x.size == dimension(xt) => {
    val z = x.map(implicitly[ToDouble[T]].apply(_))
    Try(MLPNetwork(config, topology, model).predict(z)) //56
  }
}
```

Tip

Implicit type conversion:

The input array of an element of type T must be converted implicitly to an array of an element of type `Double` using the `implicitly` keyword because the class is parameterized with a context bound.

The `predict` method of `MLPNetwork` computes the output values from an input `x` using the forward propagation as follows:

```
def predict(x: Array[Double]): Array[Double] = {
  layers.head.set(x)
  connections.foreach(_.connectionForwardPropagation)
```

```
    layers.last.output  
}
```

Model fitness

The fitness of a model measures how well the model fits the training set. A model with a high degree of fitness will probably overfit. The fitness method, `fit`, computes the mean squared errors of the predicted values against the labels (or expected values) of the training set. The method returns the percentage of observations for which the prediction value is correct, using the higher order `count` method:

```
def fit(threshold: Double): Option[Double] = model.map(m =>  
  xt.map(x => {  
    val z = x.map(implicitly[ToDouble[T]].apply(_))  
    MLPNetwork(config, topology, Some(m)).predict(z)  
  })  
  .zip(expected)  
  .count { case (y, e) => mse(y, e) < threshold }  
  /xt.size.toDouble  
)
```

Note

Model fitness versus accuracy:

The fitness of a model against the training set reflects the degree the model fits the training set. The computation of the fitness does not involve a validation set. Quality parameters such as accuracy, precision, or recall measure the reliability or quality of the model against a validation set.

Our `MLP` class is now ready to tackle some classification challenges.

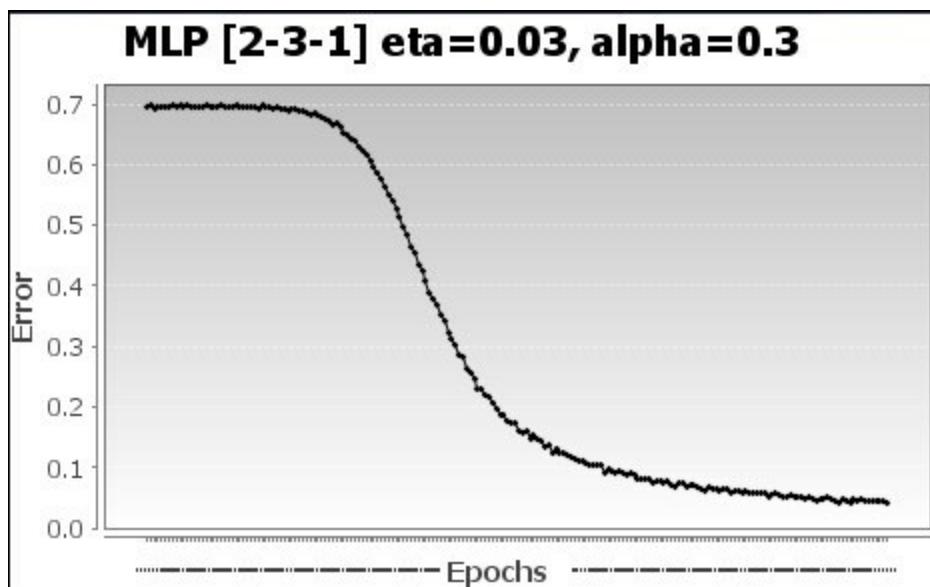
Evaluation

Before applying our multilayer perceptron to understand fluctuations in the currency market exchanges, let's get acquainted with some of the key learning parameters introduced in the first section.

Execution profile

Let's look at the convergence of the training of the multiple layer perceptron. The monitor trait (refer to the *Training* section under *Helper classes* in the *Appendix*) collects and displays some execution parameters. We selected to extract the profile for the convergence of the multiple layer perceptron using the difference of the backpropagation errors between two consecutive episodes (or epochs).

The test profiles the convergence of the MLP using a learning rate, $\eta = 0.03$, and a momentum factor of $\alpha = 0.3$ for a multilayer perceptron with two input values, one hidden layer with three nodes, and one output value. The test relies on synthetically generated random values:



Execution profile for cumulative errors for MLP

Impact of learning rate

The purpose of the first exercise is to evaluate the impact of the learning rate, ?, on the convergence of the training epoch, as measured by the cumulative errors of all output variables. The observations, `xt` (with respect to the labeled output, `yt`) are synthetically generated using several noisy patterns, functions `f1` (line 57), `f2` (line 58), as follows:

```
def f1(x: Double): Array[Double] = Array[Double]( //57
  0.1+ 0.5*Random.nextDouble, 0.6*Random.nextDouble)
def f2(x: Double): Array[Double] = Array[Double]( //58
  0.6 + 0.4*Random.nextDouble, 1.0 - 0.5*Random.nextDouble)

val HALF_TEST_SIZE = (TEST_SIZE>>1)
val xt = Vector.tabulate(TEST_SIZE)(n => //59
  if( n <HALF_TEST_SIZE) f1(n) else f2(n -HALF_TEST_SIZE))
val yt = Vector.tabulate(TEST_SIZE)(n =>
  if( n < HALF_TEST_SIZE) Array[Double](0.0)
  else Array[Double](1.0) ) //60
```

The input value, `xt`, is generated synthetically by the `f1` function for half of the dataset, then by the `f2` function for the other half (line 59). The generator for the expected values, `yt`, assigns the label `0.0` for the input value associated with the `f1` function and `1.0` for the input values associated with `f2` (line 60).

The test is run with a sample of size `TEST_SIZE` data points over a maximum of `NUM_EPOCHS` epochs, a single hidden layer of `HIDDENS.head` neurons with no softmax transformation, and the following MLP parameters:

```
val ALPHA = 0.3
val ETA = 0.03
val HIDDEN = Array[Int](3)
val NUM_EPOCHS = 200
val TEST_SIZE = 12000
val EPS = 1e-7

def testEta(eta: Double,
  xt: Vector[Array[Double]],
  yt: Vector[Array[Double]]):
```

```

Option[ (ArrayBuffer[Double], String) ] = {

implicit val mode = new MLPBinClassifier //61
val config = MLPConfig(ALPHA, eta, NUM_EPOCHS, EPS)
MLP[Double](config, HIDDEN, xt, yt)
  .counters("err").map( (_, s"eta=$eta")) //62
}

```

The `testEta` method generates the profile or errors given different values of `eta`.

The operating `mode` must be implicitly defined prior to the instantiation of the `MLP` class (line 61). It is set as a binomial classifier of type `MLPBinClassifier`. The execution profile data is collected by the `counters` method of the trait `Monitor` (line 62) (refer to the *Monitor* section of *Utility classes* in the *Appendix*).

The driver code for evaluating the impact of the learning rate on the convergence of the multilayer perceptron is quite simple:

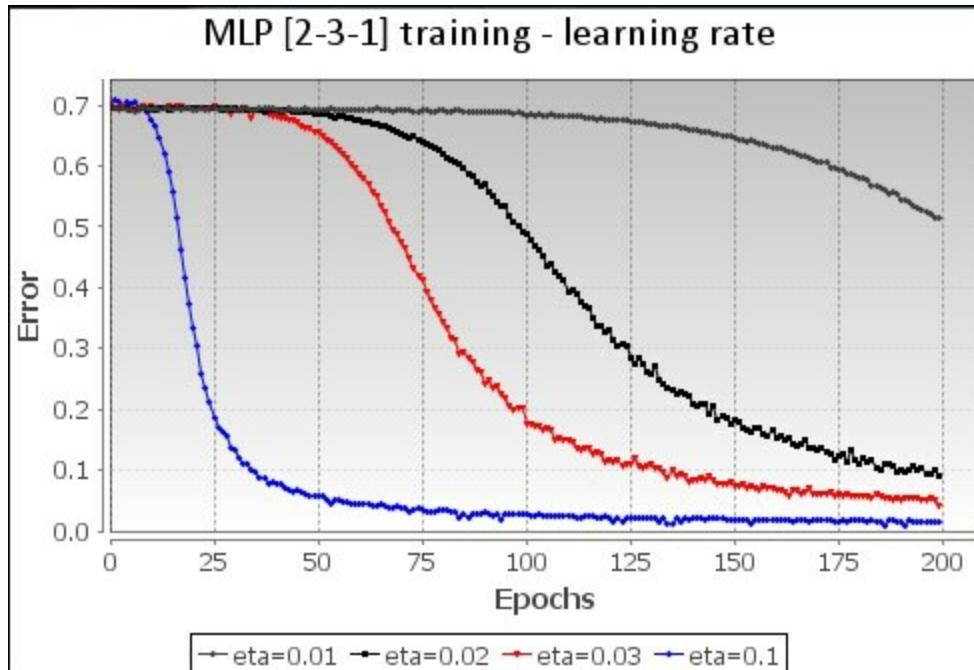
```

val etaValues = List[Double](0.01, 0.02, 0.03, 0.1)
val data = etaValues.flatMap( testEta(_, xt, yt))
  .map{ case(x, s) => (x.toVector, s) }

val legend = new Legend("Err",
  "MLP [2-3-1] training - learning rate", "Epochs", "Error")
LinePlot.display(data, legend, new LightPlotTheme)

```

The profile is created with JFreeChart library and displayed in the following chart:



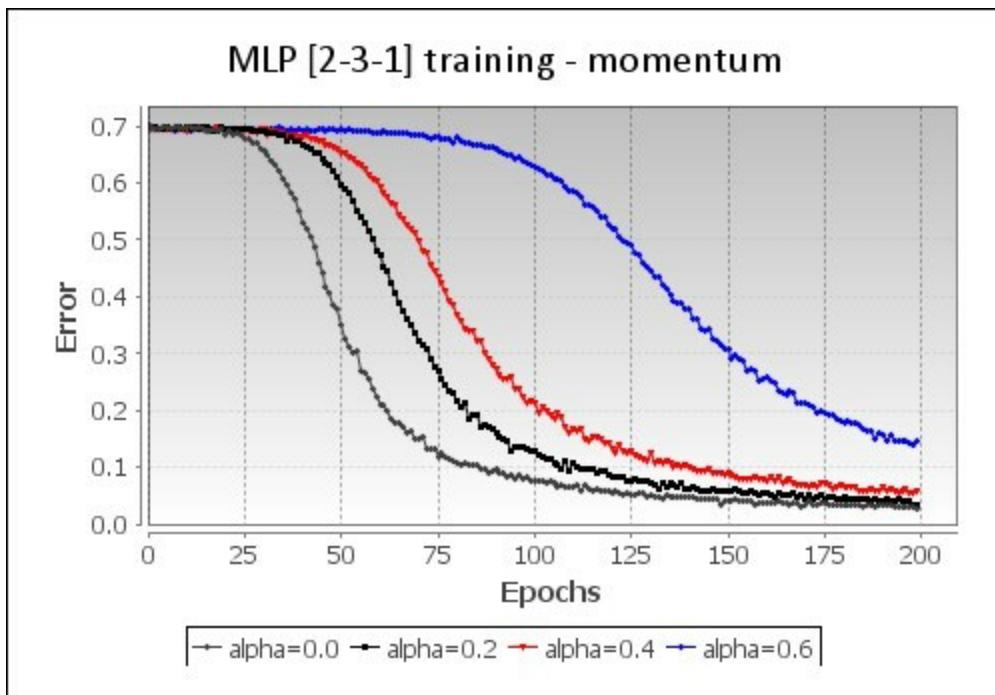
Impact of the learning rate on the MLP training

The chart illustrates that the MLP model training converges a lot faster with a larger value of learning rate. You need to keep in mind, however, that a very steep learning rate may lock the training process into a local minimum for the cumulative error, generating weights with lesser accuracy. The same configuration parameters are used to evaluate the impact of the momentum factor on the convergence of the gradient descent algorithm.

Impact of the momentum factor

Let's quantify the impact of the momentum factor, a , on the convergence of the training process toward an optimal model (synapse weights). The testing code is very similar to the evaluation of the impact of the learning rate.

The cumulative error for the entire time series is plotted in the following graph:

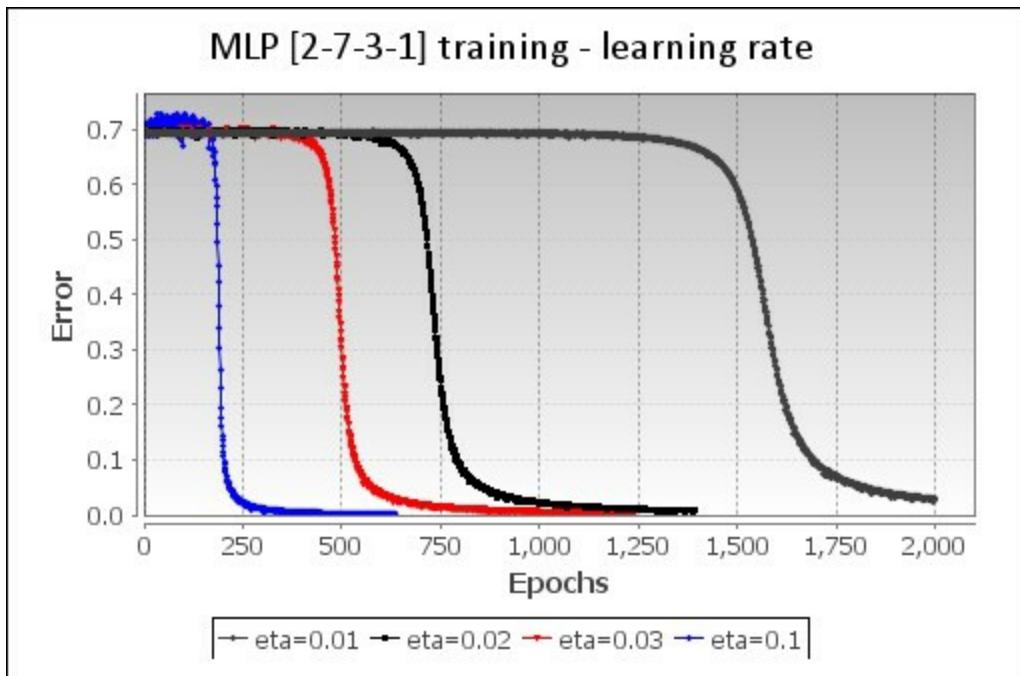


Impact of the momentum factor on the MLP training

The graph shows that the rate the mean square error 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.

Impact of the number of hidden layers

Let's consider a multilayer perceptron with two hidden layers (seven and three neurons). The execution profile for the training shows that the cumulative error of the output converges abruptly after several epochs for which the descent gradient failed to find a direction:



Execution profile of training of a MLP with two hidden layers

Let's apply our new-found knowledge regarding neural networks and the classification of variables to the exchange rates of certain currencies.

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 [10:15].

The objective of the test case is to understand the correlation factors between the exchange rate of some currencies, the spot price of gold and the S&P 500 index. For this exercise, we will use the following **exchange-traded funds (ETFs)** as proxies for the exchange rate of currencies:

- **FXA:** Rate of an Australian dollar in US dollars
- **FXB:** Rate of a British pound in US dollars
- **FXE:** Rate of the Euro in US dollars
- **FXC:** Rate of a Canadian dollar in US dollars
- **FXF:** Rate of a Swiss franc in US dollars
- **FXY:** Rate of a Japanese yen in US dollars
- **CYB:** Rate of a Chinese yuan in US dollars
- **SPY:** S&P 500 index
- **GLD:** The price of gold in US dollars

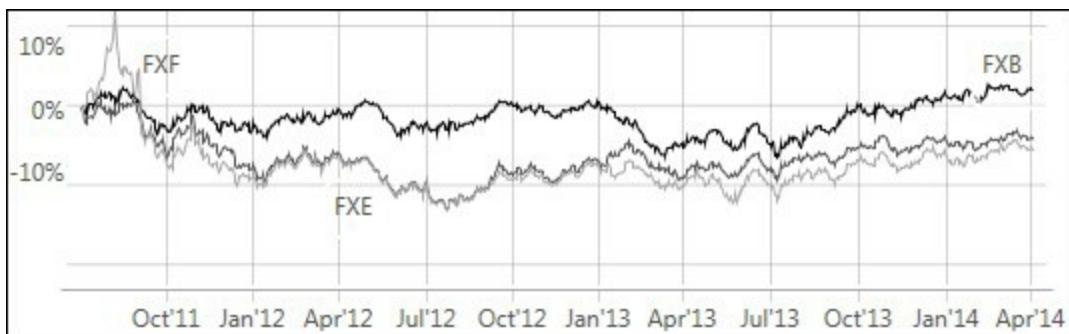
Practically, the problem to solve is to extract one or more regressive models that link one ETF, y , with a basket of other ETFs, $\{x_i\}$ $y=f(x_i)$. For example, is there a relation between the exchange rate of the Japanese yen (FXY) and a combination of the spot price for gold (GLD), the exchange rate of the Euro in US dollars (FXE), the exchange rate of the Australian dollar in US dollars (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 2.5 years. The first chart displays an initial group of potentially correlated ETFs:



An example of correlated currency-based ETFs

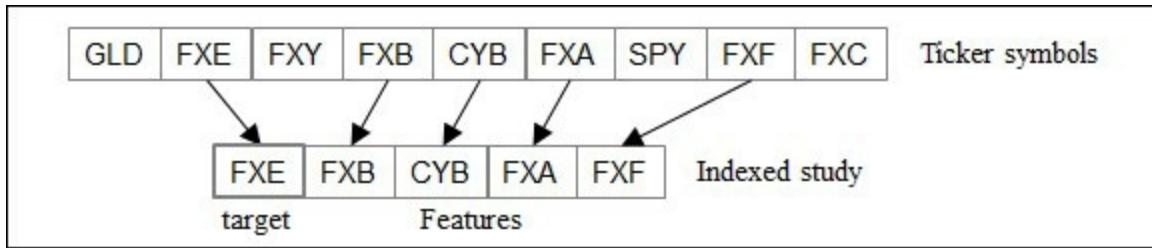
The second chart displays another group of currency-related ETFs that shares a similar price action behavior. Neural networks do not provide any analytical representation of their internal reasoning; therefore, a *visual* correlation can be extremely useful to novice engineers in validating their models:



\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, as follows:

```
val Path = "supervised/nnet/mlp"
val ALPHA = 0.8;
val ETA = 0.01
val NUM_EPOCHS = 250
val EPS = 1e-3
val THRESHOLD = 0.12
val hiddens = Array[Int](7, 7) //59
```

Besides the learning parameters, the network is initialized with multiple topology configuration (line 59).

Next, let's create the search space of the prices of all the ETFs used in the analysis:

```
val symbols = Array[String] (
  "FXE", "FXA", "SPY", "GLD", "FXB", "FXF", "FXC", "FXY", "CYB"
)
val STUDIES = List[Array[String]]( //60
  Array[String]("FXY", "FXC", "GLD", "FXA"),
  Array[String]("FXE", "FXF", "FXB", "CYB"),
  Array[String]("FXE", "FXC", "GLD", "FXA", "FXY", "FXB"),
  Array[String]("FXC", "FXY", "FXA"),
  Array[String]("CYB", "GLD", "FXY"),
  symbols
)
```

The purpose of the test is to evaluate and compare seven different portfolio or studies (line 60). The closing prices of all the ETFs over a period of 3 years

are extracted from the Google Financial tables, using the `GoogleFinancials` extractor for a basket of ETFs (line 61):

```
val prices = symbols.map(s =>
  DataSource(s"${getPath(s"dataPath/$s.csv") }"
    .map(_ .flatMap(_.get(close)))
    .filter(_.isSuccess).map(_.get)) //61
```

The next step consists of implementing the mechanism to extract the target and the features from a basket of ETFs, or studies introduced in the previous paragraph. Let's consider the following study as the list of ETF ticker symbols:

```
val study = Array[String] ("FXE", "FXF", "FXB", "CYB")
```

The first element of the study, `FXE`, is the labeled output; the remaining three elements are observed features. For this study, the network architecture has three input variables, `{FXF, FXB, CYB}`, and one output variable, `FXE`:

```
val obs = symbols.flatMap(index.get(_))
  .map(prices(_).toArray) //62
val xv = obs.drop(1).transpose //63
val expected = Array[Array[Double]](obs.head).transpose //64
```

The set of observations, `obs`, is built using an index (line 62). By convention, the first observation is selected as the label data and the remaining studies as the features for training. As the observations are loaded as an array of time series, the time features of the series are computed through `transpose` (line 63). The single output variable, `target`, must be converted into a matrix before transposition (line 64).

Ultimately, the model is built through instantiation of the `MLP` class:

```
implicit val mode = new MLPBinClassifier //65
val classifier = MLP[Double](config, hiddenLayers, xv, expected)
classifier.fit(THRESHOLD)
```

The objective or operating mode, `mode`, is implicitly defined as an `MLP` binary classifier, `MLPBinClassifier` (line 65). The `MLP.fit` method is defined in the *Training* section under *Training and Classification*.

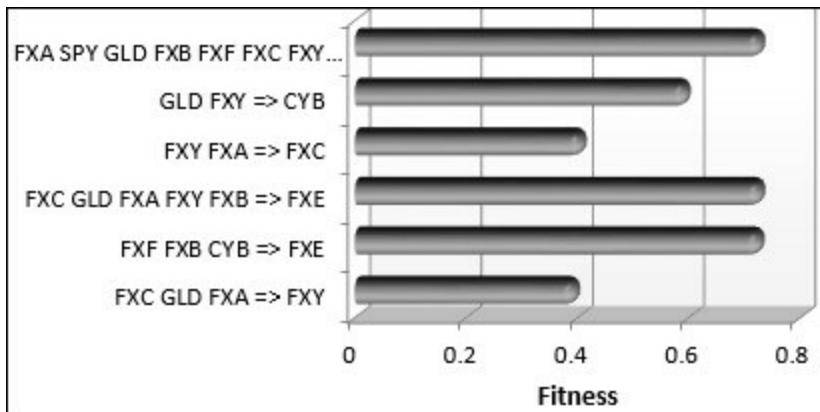
Models evaluation

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:

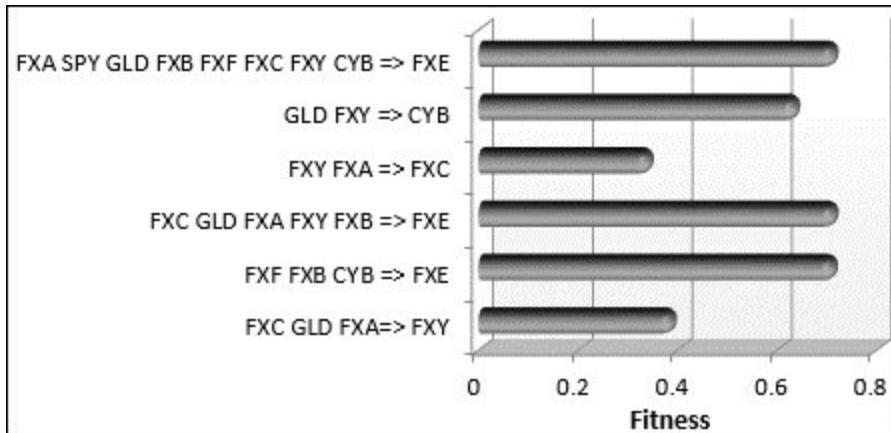
- Two hidden layers with four nodes each
- Three hidden layers with eight (with respect to five and six) nodes

The first chart visualizes the fitness of the six regression models with an architecture consisting of a variable number of inputs [2, 7], one output variable, and two hidden layers of four nodes each. The features (ETF symbols) are listed on the left-hand side of the arrow \Rightarrow along the y-axis. The symbol on the right-hand side of the arrow is the expected output value:



Accuracy of MLP with two hidden layers of four nodes each

The next chart displays the fitness 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 fittest regression models are as follows:

- $FXE = f(FXA, SPY, GLD, FXB, FXF, FXD, FXY, CYB)$
- $FXE = g(FXC, GLD, FXA, FXY, FXB)$
- $FXE = h(FXF, FXB, CYB)$

On the other hand, the prediction the Canadian dollar to the US dollar's exchange rate (FXC) using the exchange rate for the Japanese yen (FXY) and the Australian dollar (FXA) is poor with both configurations.

Tip

The empirical evaluation:

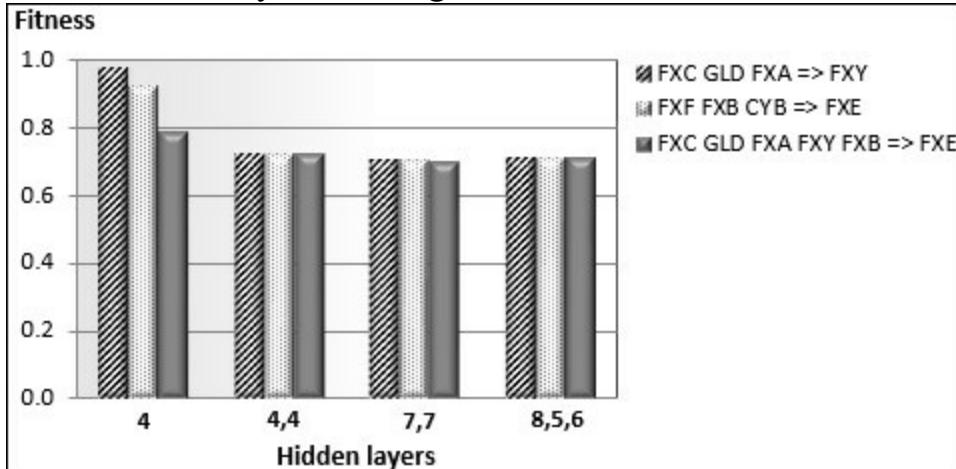
These empirical tests use a simple accuracy metric. A formal comparison of the regression models would systematically analyze every combination of input and output variables. The evaluation would also compute the precision, the recall, and the F1 score for each of those models (refer to the *Key metrics* section under *Validation* in the *Assessing a model* section in [Chapter 2, Data Pipelines](#)).

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: ($FXF, FXB, CYB \Rightarrow FXE$), ($FCX, GLD, FXA \Rightarrow FXY$), and ($FXC, GLD, FXA, FXY, FXB \Rightarrow FXE$). For this test, the accuracy is computed by selecting a subset of the training data as a test sample, for the sake of convenience. The objective of the test is to compare different network architectures using some metrics, not to estimate the absolute accuracy of each model.

The four network configurations are as follows:

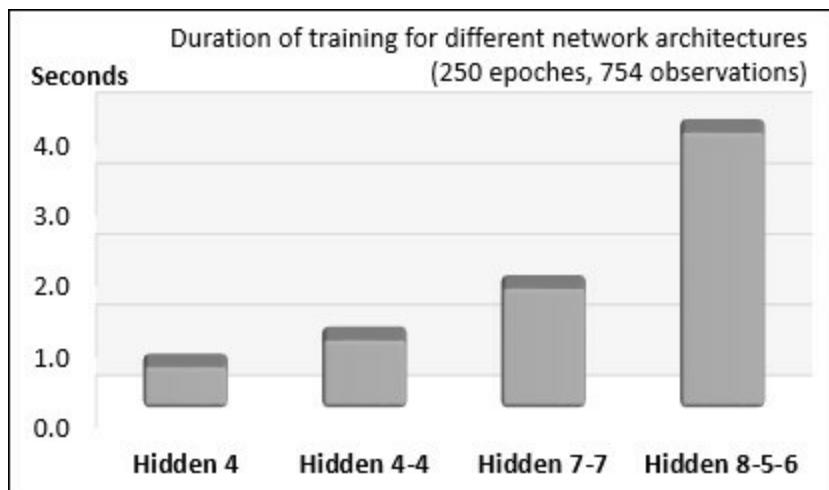
- A single hidden layer with four nodes
- Two hidden layers with four nodes each
- Two hidden layers with seven nodes each
- Three hidden layer with eight, five, and six nodes:



Impact of the number and configuration of hidden layers 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 shown in the following graph:



Impact of number and configuration of hidden layers on duration of training

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:

- 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 (refer to [Chapter 17, Apache Spark](#)).

However, as with any machine learning algorithms, neural networks have their detractors. The most documented limitations are as follows:

- Estimating the minimum size of the training set required to generate an accurate model and limiting computation time is not obvious.
- MLP models are black boxes for which the association between features and classes may not be easily described and understood.
- MLP requires a lengthy training process, especially using the batch training strategy. For example, a two-layer network has a time complexity (number of multiplications) of $O(n.m.p.N.e)$ for n input variables, m hidden neurons, p output values, N observations, and e epochs. It is not uncommon that a solution emerges after thousands of epochs. The online training strategy using a momentum factor tends to converge faster and require a smaller number of epochs than the batch

process.

Tip

GPU to the rescue:

The introduction of GPU as an alternative computing platform for machine learning has significantly improved the performance of neural networks.

- Tuning the configuration parameters, such as optimization of the learning rate and momentum factors, selection of the most appropriate activation method, and the cumulative error formula, can turn into a lengthy process.

Tip

Impact of optimization parameters on training

The configuration of the stochastic descent gradient or batched descent gradient used in minimizing the error during backpropagation has a significant impact on the time required to train an MLP model.

- A neural network cannot be incrementally retrained. Any new labeled data requires the execution of several training epochs, adding to the computation cost.

Summary

The multilayer perceptron is a non-parametric estimator that has been used for several decades in various industries, from insurance to image processing. It has been proven a reliable model for classification at the cost of lengthy execution.

In this chapter, you learned the origin of feedforward neural networks, how to implement the different steps of the training cycle with backpropagation in Scala, experiment with the different configuration parameters and apply the multilayer perceptron to the analysis of the fluctuations in the currencies market.

The multilayer perceptron, along with Monte Carlo sampling ([Chapter 8, Monte Carlo Inference](#)) and regularization ([Chapter 9, Regression and Regularization](#)), is a key ingredient of deep learning, which is the topic of the next chapter.

Chapter 11. Deep Learning

This chapter leverages the concepts and components of the multilayer perceptron described in the previous chapter and applies them to deep learning architectures. There has been quite a bit of buzz surrounding deep learning lately, although the algorithms presented in this chapter were introduced 20 to 30 years ago.

The recent advance in neural networks has as much to do with the availability of powerful hardware such as memory-based distributed computing and GPU as the academic research.

This chapter describes the following:

- *Sparse autoencoders* as a dimension reduction technique for non-linear problems
- *Binary restricted Boltzmann machines* as the core foundation of deep generative models for unsupervised learning
- *Convolutional neural networks* as an efficient alternative to the multilayer perceptron for supervised learning

The first two neural architectures do not require labeled data and rely on the input data itself to extract a model (weights).

The sections on the autoencoder and the restricted Boltzmann machine include an implementation in Scala. The convolutional neural network is presented for informational purposes.

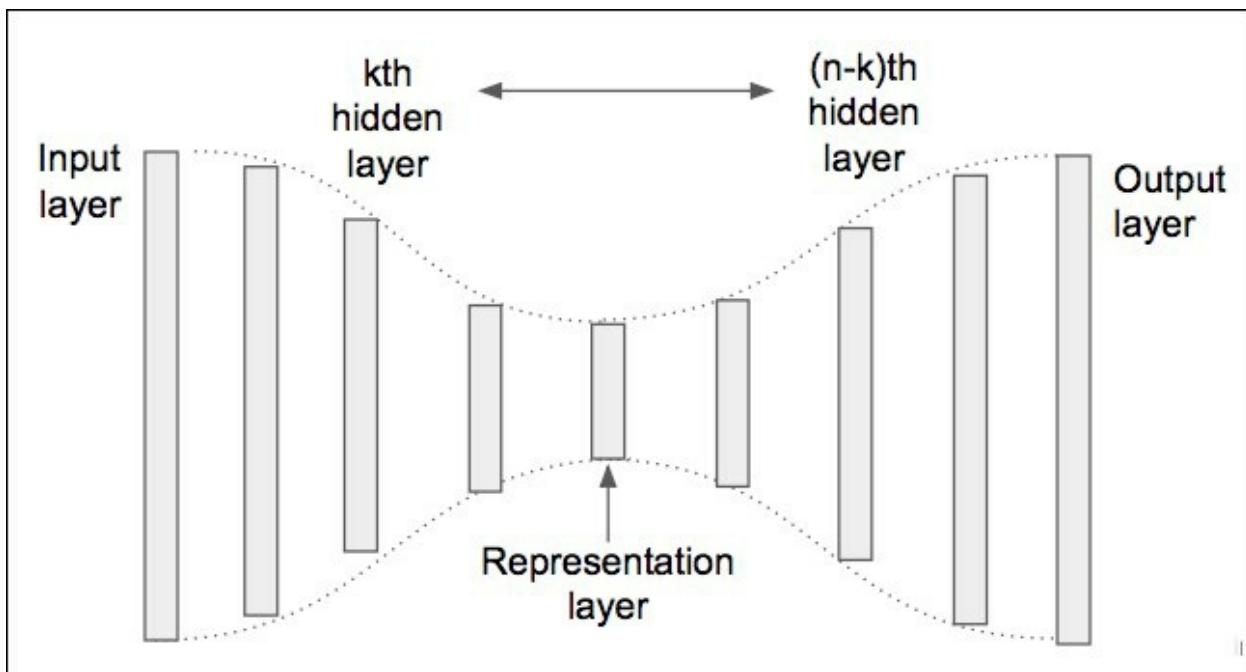
Sparse autoencoder

Autoencoders are fundamentally unsupervised learning models. They are widely used for feature extraction and dimension reduction. The simplest autoencoders are directly derived from the feed-forward neural network (see the *Feed-forward neural network* section of [Chapter 10, Multilayer Perceptron](#)).

The autoencoder attempts to reconstruct its input and therefore, the output and input layer have the same number of nodes or neurons. A conventional neural network such as the multilayer perceptron predicts a target or output vector y from an input vector x . An autoencoder predicts the output layer as the input layer x which constrains the network topology to be symmetric.

Undercomplete autoencoder

One useful application of autoencoders is the extraction of the features that are relevant to the training set (dimension reduction). The hidden layers are stacked using a symmetric pattern along a central hidden layer as described in the following diagram:



Architecture overview of an autoencoder

The topology of hidden layers in an autoencoder has the following characteristics:

- The input and output layers have the same number of neurons.
- The autoencoder has an odd number of layers.
- The first hidden layer has the same number of neurons as the last hidden layer. The second hidden layer has the same number of neurons as the one before the last hidden layer, and both contain fewer neurons than the first and last neuron, and so on.

The input and output layers represent the observations space while the

representation hidden layer is the features space, which has a significantly lower dimension.

The output of the autoencoder is the representation layer (or central hidden layer). The encoding function is the transformation f from input layer X to representation layer H . The decoding function is the transformation $\hat{?}$ from layer H to the output layer $X' \sim X$.

Deterministic autoencoder

Generic single hidden layer with f encoder and $?$ decoder:

$$x' = \emptyset(\psi(x)) \text{ loss} = \mathcal{L}(x, \emptyset(\psi(x)))$$

Linear single hidden layer h with activation s :

$$h = \emptyset(x) = \sigma(w^t \cdot x + b)$$

$$x' = \psi(h) = \sigma'(w'^t \cdot h + b)$$

$$\text{loss} = \|x - x'\|^2$$

Categorization

Autoencoders are feed-forward networks. The most common categories or configuration of autoencoders are [11:1]:

- **Undercomplete:** The dimension of the representation or hidden layers is less than the dimension of the input layer.
- **Complete:** The dimension of the hidden layers is equal to the dimension of the input layer.
- **Overcomplete:** The dimension of the hidden layers is greater than the input layer.
- **Regularized:** The loss function used in the training of the autoencoder has a regularized term independent of the size of the hidden layers.
- **Sparse:** The loss function has a penalty factor linked to the hidden layers. The penalty function is indeed a sparsity constraint parameter applied to the activation function.
- **Stochastic:** The encoding function $h = f(x)$ is a continuous probability density function.

Note

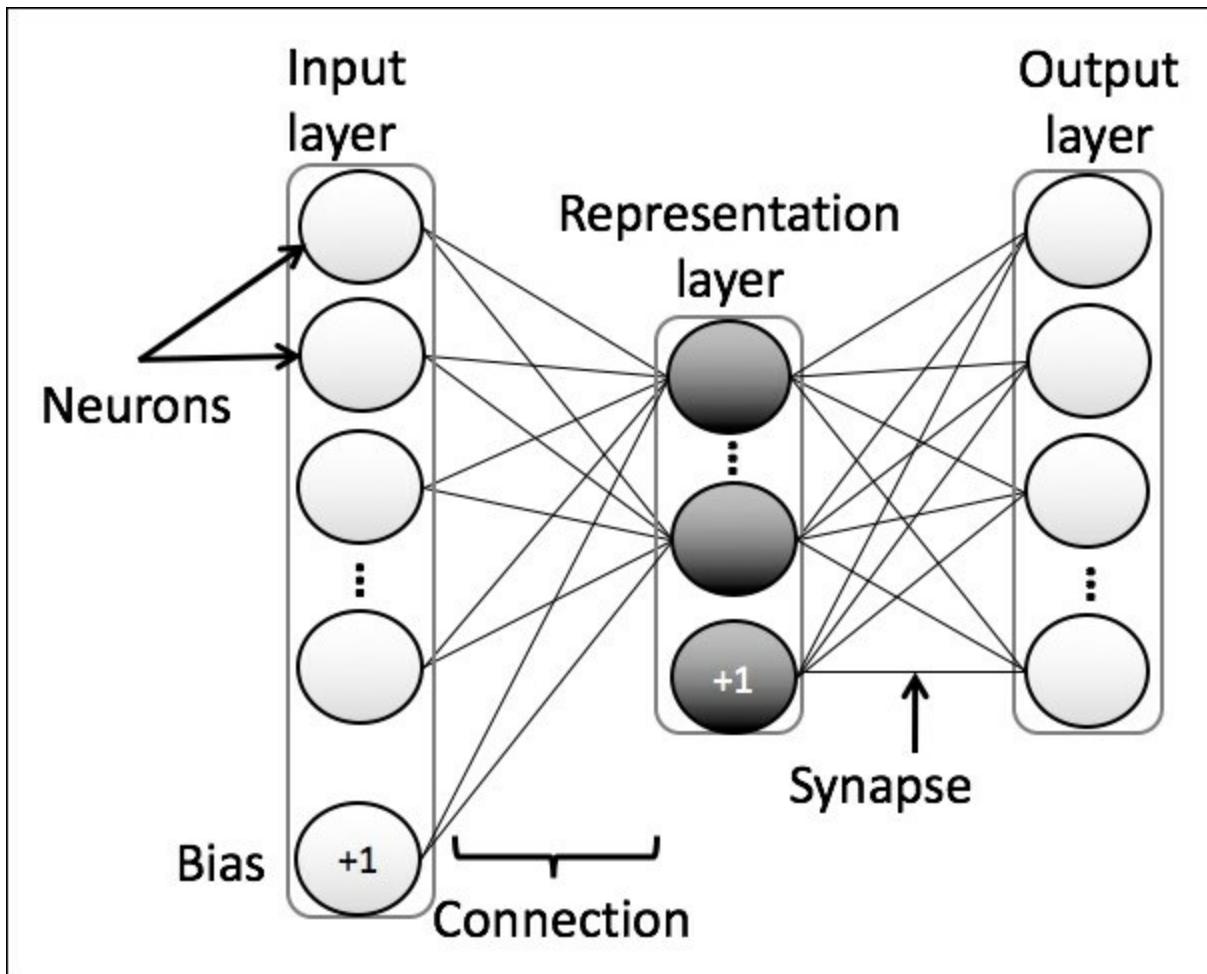
Relation to PCA

Undercomplete linear encoder function $h = f(x)$ using the mean squared error as loss function extracts correlation similar to the principal components.

The next section focuses on sparse, undercomplete autoencoders.

Feed-forward sparse, undercomplete autoencoder

Let's consider the simplest form of autoencoder: an undercomplete, feed-forward autoencoder with a sparsity penalty [11:2]. The output layer has the same neurons or variables as the input layer except for the bias (intercept) neuron with a weight +1 as shown here:



Architecture of single hidden layer autoencoder

The number of neurons in the hidden layer defines the internal representation of the input values. If there is no correlation between the input variables

(independent and identically distributed), the representation in the hidden layer would be close to the identical function. If the input variables are correlated then the reduction of the number of neurons in the representation (hidden) layer may expose some of these correlations.

One simple approach is to reduce the size of the hidden layer by creating a *sparsity constraint*. The sparsity constraint is implemented by enforcing the expectation of an activation function for a hidden neuron to be set to a fixed value, known as a *sparsity parameter*. In the case of back-propagation, the gradient descent adjusts the sparsity parameter λ iteratively with a parameter α whose value is close to 0. The iterative computation of the sparsity parameter resembles the exponential weighed average decaying formula.

Sparsity updating equations

For a sparsity constraint γ , a sparsity adjustment factor γ' , an activation function f , and a secondary learning rate β (M1):

$$\rho'_i = \lambda \cdot f(w_i^T \cdot x + b'_i) + (1 - \lambda) \cdot \rho$$

$$b'_i = b_i - \alpha \beta (\rho_i - \rho'_i)$$

The second formula adjusts the bias term for the input layer to enforce the value of the gradient decreases (convergence). If $\gamma' > \gamma$, then the activation should decrease toward its minimum value (-1 in the case of the hyperbolic tangent). Similarly, the activation should increase toward the maximum value (+1) if $\gamma > \gamma'$. A simple way to enforce this rule is to apply the correction to the bias term b' using a secondary attenuating or learning parameter β .

Tip

Overfitting versus sparsity

As the multilayer perceptron, the autoencoder is not immune to overfitting. Many hidden layers increase the degree of freedom of the model and its tendency to overfit. Adding sparsity to the original dataset is a traditional approach to reduce complexity in the model. One approach is to penalize non-sparse data points in the cost function.

Implementation

We consider the simple configuration of a sparse, undercomplete autoencoder with a single hidden (representation) layer. The implementation of the autoencoder leverages the Scala implementation of the multilayer perceptron.

First, let's define the configuration, `AEConfig`, of the autoencoder. The configuration extends the configuration `MLPConfig` of the multilayer perceptron with the sparsity constraint factor `lambda`, (line 3) and the attenuation (or secondary learning rate) `beta` (line 4):

```
class AEConfig (
    alpha: Double, //1
    eta: Double, //2
    val lambda: Double, //3
    val beta: Double, //4
    numEpochs: Int, //5
    eps: Double) //6
extends MLPConfig(alpha, eta, numEpochs, eps, tanh) //7
```

The momentum factor `alpha` (line 1), the learning rate `eta` (line 2), the number `numEpochs` of epochs (line 5), and the relative convergence criterion `eps` (line 6) completes the initialization of the neural neuron network (refer to the *Configuration* section of [Chapter 10, Multilayer Perceptron](#)). The default activation function for the autoencoder is the hyperbolic function `tanh` (line 7).

The connection between neurons in an autoencoder is defined in the class `AEConnection`. It is a specialization of the more generic MLP connection by overriding the forward-propagation method `connectionForwardPropagation` (line 9) with the sparsity penalization function, `penalize` (line 10). The array `rhoLayer` contains the sparsity constraint parameters `?` for each of the hidden neurons (line 8):

```
class AEConnection (
    config: AEConfig,
    src: MLPLayer,
    dst: MLPLayer,
```

```

model: Option[MLPModel])
extends MLPConnection(config, src, dst, model) with Sparsity {

    val lambda = config.lambda
    val beta = config.beta
    val rhoLayer = Array.ofDim[Double](src.output.length) //8

    override def connectionForwardPropagation(): Unit = { //9
        val _output: Array[Double] = synapses.indices.map(n => {
            val weights = synapses(n).map(_._1)
            weights(0) = penalize(weights.head, n) //10
            dst.activation(innerDouble(src.output, weights))
        }).toArray

        update(_output)
        dst.setOutput(_output)
    }
}

```

The class `AEConnection` reuses the method `connectionBackpropagation` that computes the backpropagation of the output error defined in `MLPConnection` class (refer to the *Connections* subsection in the *Network components* section in [Chapter 10, Multilayer Perceptron](#)).

The sparsity penalization function is implemented by the `Sparsity` trait as follows:

```

trait Sparsity {
    val lambda: Double
    val beta: Double
    val rhoLayer: Array[Double] //11
    val rhoLayerPrime: Array[Double] = rhoLayer //12
    val lambda_1 = 1.0 - lambda

    def update(activeValues: Array[Double]): Unit = //13
        rhoLayerPrime = activeValues.indices.map(n =>
            lambda*rhoLayer(n) + lambda_1*activeValues(n)
        ).toArray

    def penalize(bias: Double, index: Int): Double = //14
        bias - beta*(rhoLayer(index) - rhoLayerPrime(index))
}

```

The iterative formula M1 updates the array of sparsity parameters `rhoLayer` (line 11) by computing the new values of the sparsity parameters

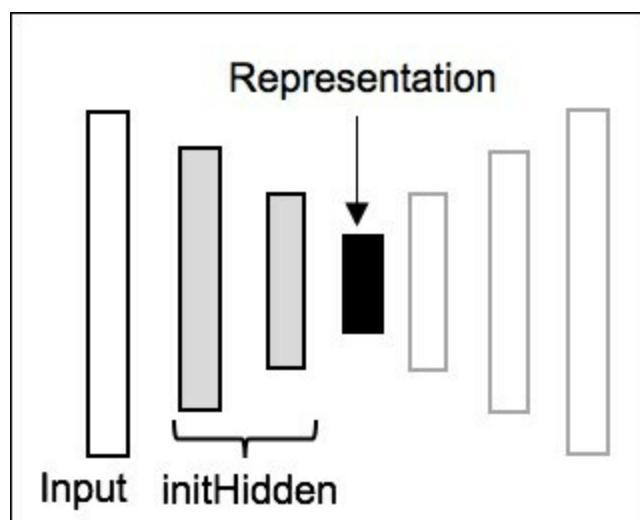
`rhoLayerPrime` (line 12).

The only reason for sub-classing the MLP network configuration, `MLPNetwork`, is to override the type of connection, as defined in the `AENetwork` class:

```
class AENetwork (
    config: AEConfig,
    topology: Array[Int],
    model: Option[MLPModel] = None)
extends MLPNetwork(config, topology, model) {

    override val connections = zipWithShift1(layers.toArray).map {
        case (src, dst) => AEConnection(config, src, dst, model)
    }
}
```

Finally, the `AE` class implements the generation of the unsupervised autoencoder model. Contrary to the `MLP`, the model is generated from unlabeled input data. The class implements the `ITransform` trait (refer to the *Implicit model* subsection in the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#)). The class argument `representation` defines the number of neurons in the central hidden layer (line 5). The argument `initHidden` specifies the configuration of the hidden layers that sit between the input layers and the representation layer (line 14), as illustrated in the following diagram:



Initialization of the topology of an autoencoder

The default configuration has only three layers: input, representation, and output. The topology leverages the symmetry of the autoencoder architecture (line 16):

```

class AE[@specialized(Double) T: ToDouble] (
    config: AEConfig,
    initHidden: Array[Int] = Array.empty[Int], //14
    representation: Int, //15
    xt: Vector[Array[T]])
extends ITransform[Array[T], Array[Double]] with Monitor[Double]

lazy val topology = //16
    if (initHidden.length == 0)
        Array[Int](xt.head.length, representation, xt.head.length)
    else
        Array[Int](xt.head.length) ++
        generateHiddenLayer(hidden, representation) ++
        Array[Int](xt.head.length)

val model: Option[MLPModel] = { //17
    val network = AENetwork(config, topology)
    var prevErr = Double.MaxValue

    (0 until config.numEpochs).find(n => {
        val err = fisherYates(xt.size)
        .map(n => {
            val z = xt(n).map(implicitly[ToDouble[T]].apply(_))
            network.trainEpoch(z, z) //18
        }).sum/xt.size

        val diffErr = err - prevErr
        prevErr = err
        abs(diffErr) < config.eps
    }).map(_ => network.getModel)
}

override def |>: PartialFunction[Array[T], Try[Array[Double]]] = {
    case x: Array[T] isModel && x.length == dimension(xt) =>
        Try( AENetwork(config, topology, model) //19
            .predict(x.map(implicitly[ToDouble[T]].apply(_)))
        )
}
}

```

The generation of the model for the autoencoder resembles the implementation of the `MLP.train` method (refer to the *Model generation*

subsection in the *Training and classification* section in [Chapter 10](#), *Multilayer Perceptron*). The only difference of the input value of type `T`, converted to a `Double`, is that `z` is used as both the input and the expected value in the method `trainEpoch` (line 18). The implementation of the predictive method `|>` is similar to the `MLP.>:`.

Restricted Boltzmann Machines (RBMs)

The RBM is a probabilistic, undirected, graphical model with an input layer of observable variables or features and a layer of latent, representative neurons. It is also interpreted as a stochastic neural network. RBMs can be stacked to compose a **deep belief network (DBM)** [11:3].

The simplest form of RBM architecture consists of one layer of input variables and a layer of latent variables.

Boltzmann machine

The *Boltzmann machine* is a symmetric network of binary vectors of stochastic processing nodes. It is an undirected structure, used to discover interesting features in datasets composed of binary vectors and the probability distribution of the input data. It is commonly associated with *Markov random fields* [11:4].

Note

For a neuron i , weights w_{ij} of connections from units j , and a bias unit b_i , an energy $E(x_i)$ and input x_i , probability for unit i :

$$E(x) = -\left(x^T W x + b^T x \right) = -\sum_{i < j}^{n-1} w_{ij} x_i x_j - \sum_{i=0}^{n-1} b_i x_i$$

Note

Boltzmann probability

$E(x=0)$ being the energy for negative outcome and $E(x=1)$ being the energy for the positive outcome with a temperature T :

$$p(x=1) = \frac{1}{1 + e^{-\frac{E(x=1) - E(x=0)}{T}}}$$

The Boltzmann machine is a very generic bidirectional network of connected neurons. For instance, neurons within a given layer could be interconnected adding an extra dimension to the mathematical representation of the network: tensors. Consequently, the learning process for such network architecture is computationally intensive and difficult to interpret.

One solution is to add two restrictions to the architecture of the Boltzmann machine architecture:

- Restrict the number of layers in the network to 2
- The network is a fully connected, bipartite, undirected graph

Such architecture is known as the RBM (or *harmonium*).

The type of visible and hidden features can be binary, categorical, or continuous (Gaussian). Therefore, there are potentially nine different kind of RBMs, although not all have meaningful applications.

Note

Deep Boltzmann machines (DBMs)

DBMs are built by stacking multiple layers of independent latent variables. DBMs belong to the category of deep generative models. They typically contain binary units, although they can be easily extended to support real values. The additional hidden layers preserve the bipartite property of the network.

The rest of the section deals exclusively with the binary RBM: a pair of two layers, one containing observed features, the other representing the latent variables.

Note

Deep belief network versus deep Boltzmann machine

Deep belief networks (DBNs) are bipartite generative models with multiple hidden layers. The most common architecture has real value as visible units and binary hidden units. Contrary to the DBN, DBN is a directed graphical model [11:5].

Binary restricted Boltzmann machines

RBMs are a category of unsupervised learning algorithm that is defined as an undirected graphical model.

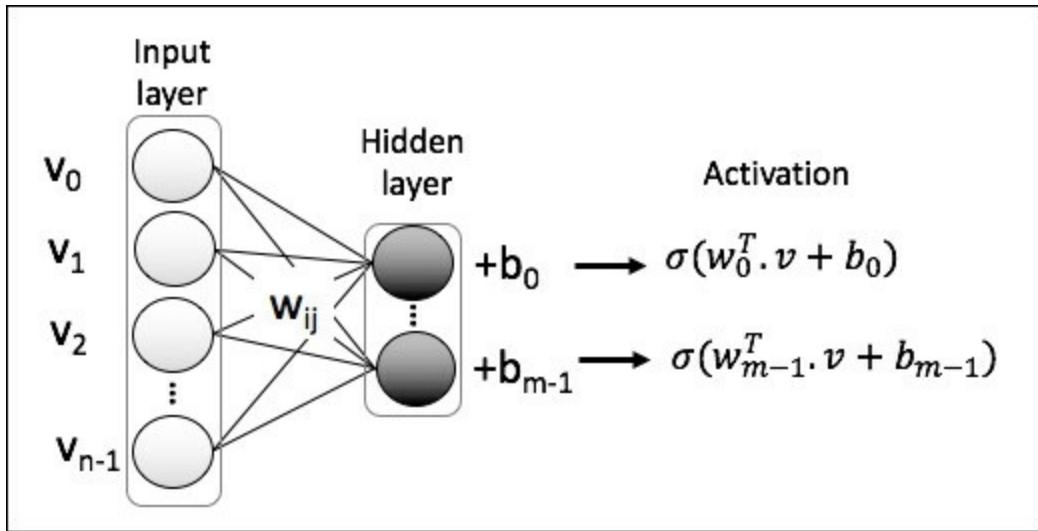
Conditional probabilities

Let's consider the simplest architecture of the RBM with n observed binary features (input layer), v_i and m latent variables and h_j (hidden layer). The dependencies between neurons from the input layer and the neuron from the output layer are defined by weights, w_{ij} . The output to the hidden layer is computed through the activation (sigmoid) function (refer to the *Network components* section of [Chapter 10, Multilayer Perceptron](#)).

The conditional probability of the input value v_j given the hidden layer h is given by the formula M1:

$$p(v|h) = \prod_{j=0}^{n-1} p(v_j | h) \quad p(v_j = 1 | h) = \sigma\left(\sum_{i=0}^{n-1} w_{ij} h_i + b_j\right)$$

The computation of the activation values (or forward pass) is illustrated by the following diagram:

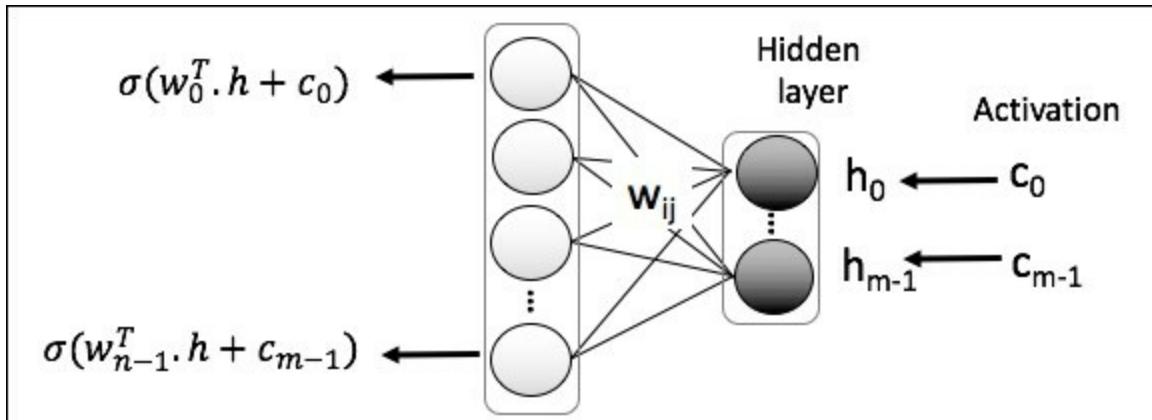


Forward pass for a two-layer restricted Boltzmann machine

Similarly, the conditional probability of the hidden value h_i given the input neuron v is computed by the following formula, M2:

$$p(h | v) = \prod_{i=0}^{m-1} p(h_i | v) \quad p(h_i = 1 | v) = \sigma\left(\sum_{j=0}^{m-1} w_{ij} v_j + c_i\right)$$

The computation of the activation values (or forward pass) is illustrated by the following diagram:



Backward pass for a two-layer restricted Boltzmann machine

Let's create a class, SCondBProb, that implements the sampling on the two

conditional probabilities defined by the formulas $M1$ and $M2$. The class has two arguments, the matrix of the `weights` of the connection between the data and hidden layers and the number of samples, `nSamples`, to be generated from the conditional probabilities (line 1):

```
type Weights = Array[Array[Double]]

class SCondProb(weights: Weights, nSamples: Int) { //1

  def probHV(h: Array[Double]): IndexedSeq[Double] =
    (0 until h.size).map(n => sigmoid(
      inner(h, weights(n+1)) + weights(0)(n) - weights(0)(n)
    ) //2

  def probVH(v: Array[Double]): IndexedSeq[Double] = {
    val tW = weights.transpose
    (0 until v.size).map(n => sigmoid(
      inner(v, tW(n+1)) + tW(0).head) - tWeights(0)(n)
    ) //3

  def sample(
    input: Array[Double],
    positive: Boolean): Array[Double]
}

The method probHV implements the formula  $M1$  to compute the conditional probability of an observed feature given a latent feature h (line 2). Similarly, the method probVH computes the conditional probability of a latent variable (formula  $M2$ ) given an observation v (line 3). Both methods apply the sigmoid function to the inner product of the weights and input values.
```

Sampling

Problems that require the computation of the sum or integral can be intractable if no known close form exists. Monte Carlo sampling provides a numeric approximation to the solution.

Note

Monte Carlo integration

In its simplest form, Monte Carlo integration consists of generating many data points within and beyond a curve, a surface or volume defined by a function, then computing the percentage of data points located within the curve, surface, or volume.

The simplest form of sampling the probability density function of a distribution is to apply it to uniformly generated random values. Let's consider an observation `in` (line 4). The method `sample` computes `nSamples` random values from either conditional probability `probHV` or `probVH` (line 5), selected through the Boolean argument, `positive`:

```
def sample(
    in: Array[Double], //4
    positive: Boolean): Array[Double] = {

    val rawOut = if(positive) probHV(in) else probVH(in)
    (0 until nSamples).map(_ => rawOut(nextInt(in.length))) //5
    ).toArray
}
```

Note

The sampling of the conditional probabilities is simple because the RBM assumes the variables (units) within a layer are independent. Consequently, the variables of an entire layer can be sampled at once (block sampling), which improves the efficiency of the computation. The block Gibbs sampler is an example of block sampling.

Log-likelihood gradient

The binary RBM unsupervised learning algorithm relies on a gradient-based optimization. The method consists of normalizing the likelihood $p(v|w)$ using a cumulative partition function, then computing the gradient of the logarithm of the likelihood as the difference between a positive term (or phase or energy) and a negative term.

Note

Log-likelihood gradient

For an observation v , model weight w , and a partition function Z :

$$p(v|w) = \frac{1}{Z(w)} \tilde{p}(v|w)$$
$$\frac{\partial \ln p(v|w)}{\partial w} = \frac{\partial \ln \tilde{p}(v|w)}{\partial w} - \frac{\partial Z(w)}{\partial w}$$

The fundamental question is how to compute the gradient of the logarithm of the conditional probability. One simple solution is to compute the joint probability $p(v, h|w)$ as a combination of the two conditional probabilities $p(v|h, w)$ and $p(h|v, w)$ using a sampling technique, and more specifically, a **Markov Chain Monte Carlo (MCMC)** algorithm [11:6].

MCMC algorithms such as the well-known *Metropolis-Hastings* are critical techniques in the data scientist tool box [11:7]. One of the simplest MCMC methods, known as the *Gibbs* sampler, is a good candidate to approximate the gradient of the logarithm of the likelihood.

A more recent approach, *contrastive divergence*, has emerged as a more efficient technique that extends the Gibbs sampling method.

Contrastive divergence

Contrastive divergence is an iterative method that consists of sampling the distribution of the input data, then converging the model (likelihood) toward the distribution of the input [11:8].

As described in the previous section, the gradient of the logarithm of the likelihood is computed as the difference between a positive and a negative term or energy. The description of the various steps of the derivation of the log likelihood is beyond the scope of this book. The formulas for the positive and negative terms (also known as energies) are defined as follows:

Note

Contrastive divergence

Approximation of gradient of logarithm likelihood (M3):

$$\frac{\partial \ln(\mathcal{L}(w|v))}{\partial w_{ij}} \triangleq \Delta_e = \frac{1}{n} (e_+ - e_-)$$

Note

Positive term/energy (M4):

$$e_+ = v^{(0)} \cdot p(v^{(0)} | h)$$

Note

Negative term (M5):

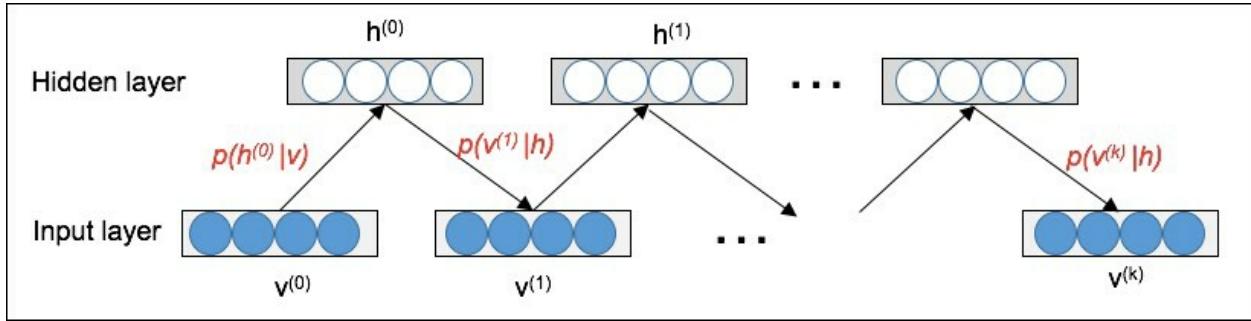
$$e_- = v^{(1)} \cdot p(v^{(1)} | h) = p(h^{(0)} | v) \cdot h^{(0)} = p(v^{(0)} | h)$$

As the algorithm progresses, the estimation of the negative energy becomes more and more accurate.

As mentioned in the previous section, sampling methods such as MCMC can provide a good approximation of an integral. Let's consider the sampling of the conditional probabilities to compute the positive (M4) and negative (M5) terms of the gradient of the logarithm of the likelihood.

The k-step contrastive divergence is initialized with a sampling of the input data (observations) using the Gibbs technique. At each step, the value $h(t)$ is sampled from $p(h|v(t))$ (positive energy) then the negative energy is

recomputed by sampling $v(t+1)$ from $p(v|h(t))$ as illustrated in the following diagram:



k-step contrastive divergence with Gibbs sampling

Let's implement the computation of the contrastive divergence as defined in the formulas $M3$, $M4$, and $M5$. The class `ContrastiveDivergence` requires the size of samples, `nSamples`, to be defined (line 6). The class processes variables that are contextually bound to the type `Double` (line 7). The method `apply` computes an approximation of the gradient of the logarithm of the likelihood from the current `weights` matrix and data `v0`:

```
type DblMatrix = Array[Array[Double]]

class ContrastiveDivergence[T: Double](nSamples: Int) { //6

  def apply(weights: Weights, v0: Array[Array[T]]): RealMatrix = {
    val dv0 = v0.map(_.map(implicitly[ToDouble[T]].apply(_))) //7

    val pSampler = new SCondProb(weights, nSamples) //8

    val h0 = dv0.map( pSampler.sample(_, true) ) //9
    val posEnergy = multiplyTranspose(h0, dv0) //12

    val v1 = h0.map( pSampler.sample(_, false) ) //10
    val h1 = v1.map( pSampler.sample(_, true) ) //11
    val negEnergy = multiplyTranspose(v1, h1) //13

    posEnergy.subtract(negEnergy)
      .scalarMultiply(1.0/v0.head.length) //14
  }
}
```

The `apply` method initializes the sampler, `pSampler`, for the two conditional probabilities (line 8) and generates samples for the density function of the

conditional probability $p(h|v_0)$ (line 9), $p(v|h_0)$ (line 10), and $p(h|v_I)$ (line 11). It computes the positive term using the formula $M4$ (line 12) and the negative term, formula $M5$ (line 13). Finally, the method executes the formula $M3$ (line 14).

The operations on matrices use the Apache Commons Math library (refer to the Appendix) introduced in the *Tools and frameworks* section of [Chapter 1, Getting Started](#), as illustrated by the implementation of the method `multiplyTranspose`:

```
def multiplyTranspose(
    input1: DblMatrix, input2: DblMatrix
): DblMatrix = {
    val realMatrix1 = new Array2DRowRealMatrix(input1)
    val realMatrix2 = new Array2DRowRealMatrix(input2)
    realMatrix1.transpose.multiply(realMatrix2).getData
}
```

Configuration parameters

The data class `RBMConfig` encapsulates the configuration parameters used in the training of the binary RBM, that is, the learning rate (line 15), the maximum number of iterations allowed during training (line 16), the tolerance for the convergence criterion (line 17), and the `penalty` function used to compute the cost or `loss` at each iteration of the training of the RBM:

```
case class RBMConfig(
    learningRate: Double, //15
    maxIters: Int, //16
    tol: Double, //17
    penalty: Double => Double) //18

    final def loss: Double = penalty(learningRate)
}
```

Unsupervised learning

The elements of the RBM algorithm are in place to implement the training method. The class `RBM` has five arguments:

- `numInputs`: Dimension of the observations (number of neurons in the input layer) – line 19
- `numLatents`: Dimension of the latent features (number of neurons in the hidden layer) – line 20
- `config`: Configuration parameters for the training of RBM – line 21
- `nSamples`: Size of sampling of the conditional probabilities – line 22
- `input`: Input data of element type bound to `Double` – line 23:

```

class RBM[@specialized(Double) T: Double] (
    numInputs: Int, //19
    numLatents: Int, //20
    config: RBMConfig, //21
    nSamples: Int, //22
    input: Array[Array[T]]) //23
extends ITransform[Array[T], Try[Array[Double]]] //24

val model: Option[Weights] = train
val cd = new ContrastiveDivergence[T](nSamples) //25

override def |>: PartialFunction[Array[T], Try[Array[Double]]]
private def train: Option[Weights]
}

```

The class implements the `ITransform` trait (refer to the *Implicit model* subsection in the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#)) (line 24). The divergence is instantiated with the proper sample size (line 25).

The iterative learning algorithm is implemented as the Scala tail recursive method, `train` (line 27). The matrix of `weights` (type `Weights`) is arbitrarily initialized with a Gaussian distribution (line 26).

Each recursion computes the approximation of the gradient of the logarithm of the likelihood `dWeights` (line 28). It applies the `divergence` algorithm to the `weights` computed in the previous recursion (line 30) and applies the existing gradient `prevDW` (line 31) to update the gradient `dWeights`:

```

def train: Option[Weights] = {
  var weights = Array.fill(numInputs)(//26
    Array.fill(numLatents)(gauss(0.0, 0.001, nextDouble)))
}

```

```

@scala.annotation.tailrec //27
def train(  

    x: Array[Array[T]],  

    prevDW: RealMatrix, //30  

    count: Int): Boolean = {  

    val divergence = cd(weights, x) //31  

    val diff = divergence.subtract(prevDW)  

        .scalarMultiply(config.cost)) //30  

    val dWeights = diff.scalarMultiply(config.learningRate)//28  

weights = weights.add(dWeights).getData //32  

    if(mse(prevDW, dWeights) < config.tol) //33  

        Some(weights)  

    else if(count > config.maxIters) //34  

        None  

    else  

        train(x, dWeights, count+1) //35  

    }  

train(input, new Array2DRowRealMatrix(weights), 0)
}

```

The learning algorithm applies the different exit strategies once the `weights` are updated with the recomputed gradient (line 32) as follows:

- The recursion terminates successfully if the mean square error, `mse`, between the previous and current gradient meets the convergence criterion, `tol` (line 33)
- The recursion fails as soon as the maximum number of iterations, `maxIters`, is reached (line 34)
- The recursion continues with the new gradient `dWeights` otherwise (line 35)

Tip

Gradient ascent

The algorithm relies on the gradient ascent. Therefore, the weights are updated by adding the recomputed gradient to the existing weights in line 30.

The implementation of the mean square error method consists of computing

the difference between two consecutive matrices of gradient and summing the square of the elements of the resulting matrix:

```
def mse (pDW: RealMatrix, dW: RealMatrix): Double =  
    sqrt( pDW.subtract(dW).getData  
.map( _ .reduceLeft(_*_)).sum )
```

This computation of the error using changes in gradient and implemented in the previous code snippet is quite simple. It approximates the second derivative or change in the gradient. The reader is invited to experiment with different error formulas which take into account changes in the loss function. The selection of the convergence criterion according to the learning rate is quite challenging [11:9].

Tip

RBM for continuous variables

The binary RBM described in this chapter can be extended to model a continuous distribution. It simply required the normalization of the input data and assigning the visible variable the probability 1. The conditional probability $p(v_j=1|h)$ used in the binary RBM becomes the state of the continuous variable v_j .

We use the overridden method `:>` to compute a conditional probability $p(h|v)$:

```
def |> : PartialFunction[Array[T], Try[Array[Double]]] = {  
  case x: Array[T] if isModel && x.length==input.head.length =>  
    val z = x.map(implicitly[ToDouble[T]].apply(_))  
    val probSampler = new SCondProb(model.get, nSamples)  
    Try( probSampler.probHV(z).toArray )  
}
```

The method returns the conditional probability as the value of the activation function of the dot product of the input `x` and the `model`.

Convolution neural networks

This section is provided as a brief introduction to convolution neural networks without Scala implementation. In a nutshell, a convolutional neural network is a feed-forward network with multiple hidden layers, some of them relying on convolution instead of matrix or tensor multiplication ($w^T \cdot x$).

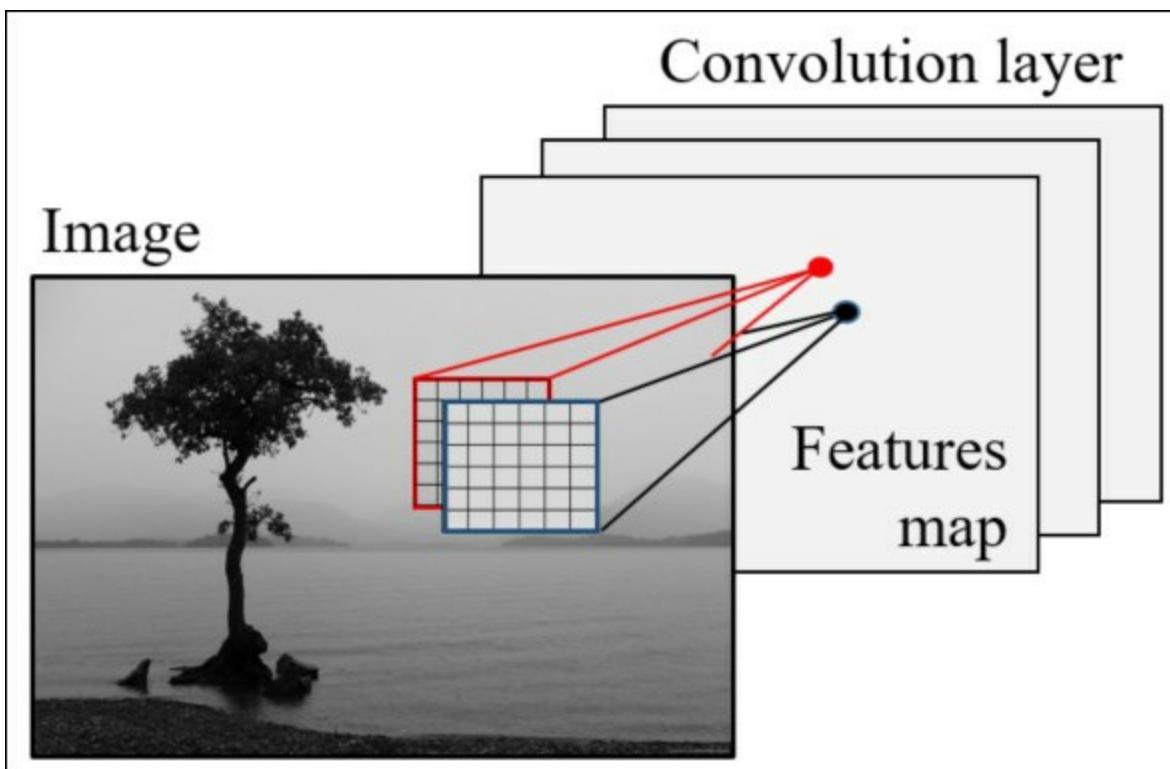
Up to this point, the layers of the perceptron were organized as a fully connected network. The number of synapses or weights increases significantly as the number and size of hidden layers increases. For instance, a network for a feature set with a dimension 6, three hidden layers of 64 nodes each, and 1 output value requires $(7 \times 64 + 2 \times 65 \times 64 + 65 \times 1) = 8833$ weights!

Applications such as image or character recognition require a very large feature set, making training a fully connected layered perceptron very computationally intensive. Moreover, these applications need to convey spatial information such as the proximity of pixels as part of the features vector.

A recent approach, known as *convolution neural networks*, consists of limiting the number of nodes in the hidden layers an input node is connected to. In other words, the methodology leverages spatial localization to reduce the complexity of connectivity between the input and the hidden layer [11:10]. The subset of input nodes connected to a single neuron in the hidden layer is known as the *local receptive fields*.

Local receptive fields

The neuron of the hidden layer is learning from the local receptive fields or sub-image of n by n pixels, each of those pixels being an input value. The next local receptive field, shifted by one pixel in any direction, is connected to the next neuron in the first hidden layer. The first hidden layer is known as the **convolution layer**. An illustration of the mapping between the input (image) and the first hidden layer (convolution layer) follows:



Generation of convolution layer from an image

It would make sense that each n by n local receptive field has a bias element (+1) that connects to the hidden neuron. However, the extra complexity does not lead to a more accurate model, and therefore, the bias is shared across the neurons of the convolution layer.

Weight sharing

The local receptive fields representing a small section of the image are generated by shifting the fields by one pixel (*UP, DOWN, LEFT, or RIGHT*). Therefore, the weights associated with the local receptive fields are also shared across the neurons in the hidden layer. As a matter of fact, the same feature, such as an image color or edge, can be detected in many pixels across the image. The maps between the input features and neurons in the hidden layer, known as *feature maps*, share weights across the convolution layer. The output is computed using the activation function.

Note

Tanh versus sigmoid activation

The sigmoid is predominantly used in the examples related to the multilayer perceptron, as the activation function for the hidden layer. The hyperbolic tangent function is commonly used for convolution networks.

Convolution layers

The output computed from the feature maps is expressed as a convolution that is similar to the convolution used in discrete Fourier transformed based filters (refer to mathematical expression M11 in the *DFT-based filtering* subsection in the *Fourier analysis* section in [Chapter 3](#), *Data Pre-processing*). The activation function that computes the output in the hidden layer must be modified to consider the local receptive fields.

Note

Activation convolution neural network

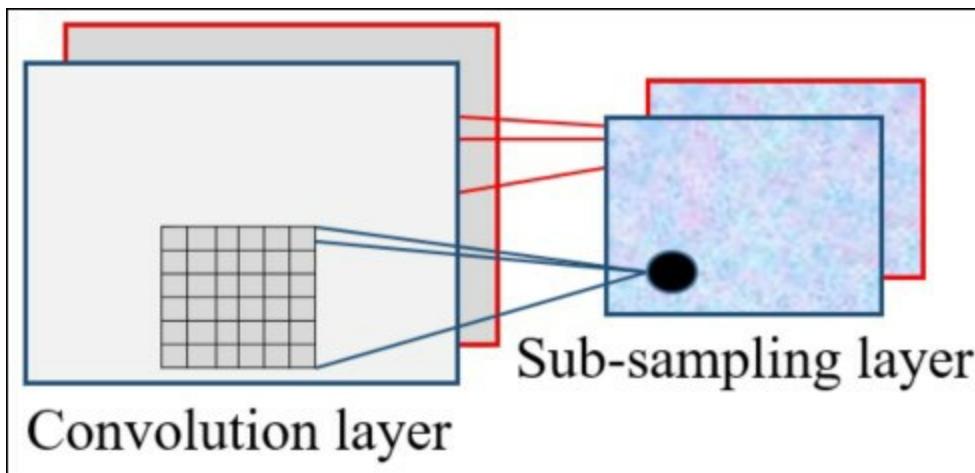
Output value z_j for a shared bias w_0 , an activation function s , some local receptive fields of n by n pixels, input values x_{ij} and weights w_{uv} associated to a feature map:

$$\tilde{z}_j = \sigma \left(w_0 + \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} w_{u,v} x_{j+u,i+v} \right)$$

The next step in building the neural network would be to use the output of the convolution layer to a fully connected hidden layer. However, the feature maps in the convolution layer are usually similar enough that they can be reduced to a smaller set of outputs through an intermediate layer known as sub-sampling layers [11:11].

Sub-sampling layers

Each feature map in the convolution layer is reduced or condensed into a smaller feature map. The layer composed of these smaller feature maps is known as the sub-sampling layer. The purpose of the sampling is to reduce the sensitivity of the weights to any minute changes in the image between adjacent pixels. The sharing of weights reduces the sensitivity to any non-significant changes in the image:

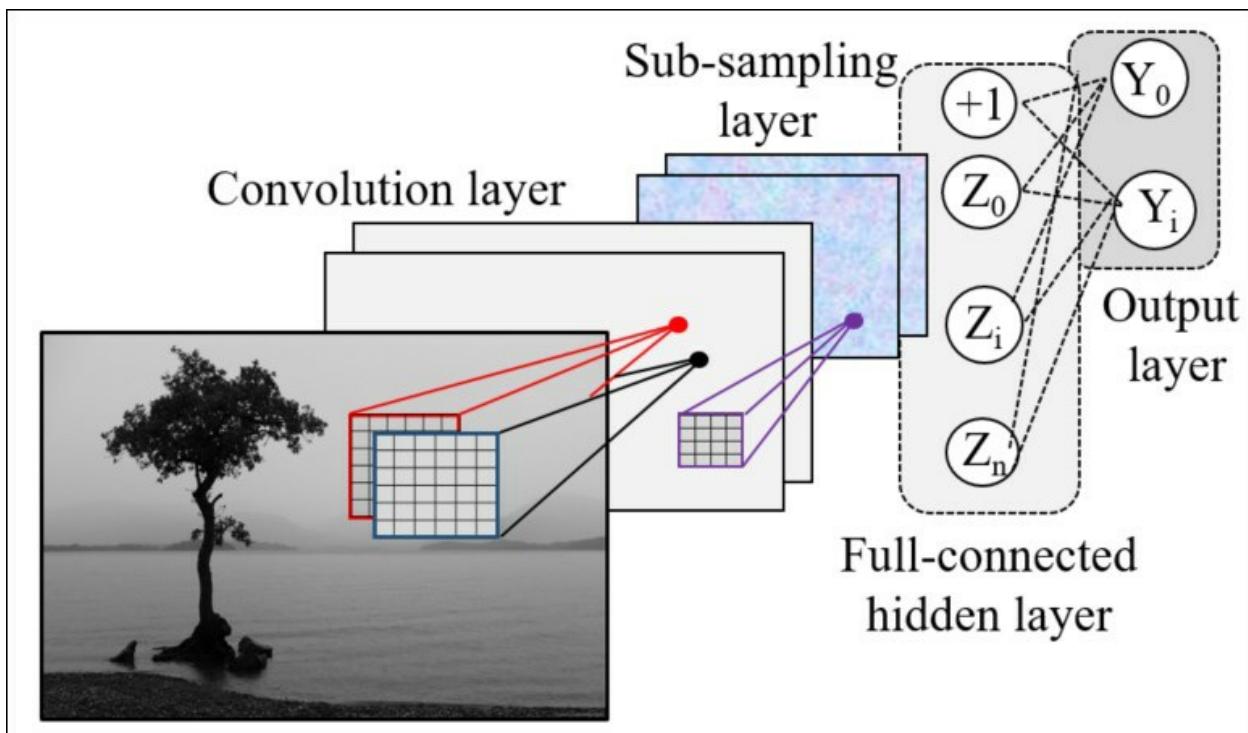


Connectivity between feature map from convolution to sub-sampling layer

The sub-sampling layer is sometimes referred to as the *pooling layer*.

Putting it all together

The last layers of the convolution neural network are the fully connected hidden layer and output layer, subjected to the same transformative formulas as the traditional multilayer perceptron. The output values may be computed using a linear product or a *Softmax* function:



Overview of a convolution neural network

The error backpropagation algorithm described in the *Step 2: error propagation* section of [Chapter 10, Multilayer Perceptron](#) should be modified to support the feature map [9:17].

Tip

Architecture of convolution network

Deep convolution neural networks have multiple sequences of convolution

layer and sub-sampling layers, and may have more than one fully connection hidden layer.

Summary

This concludes the second chapter dedicated to neural networks. There are many more deep learning architectures such as recurrent neural networks as an alternative to hidden Markov models or deep (stacked) belief networks.

In this chapter, you learned about feature compression through the implementation of an under-complete, sparse autoencoder and a binary RBM in Scala. An introduction to convolution-based feed-forward neural networks concludes this brief overview of deep learning.

The next chapter describes one last type of discriminative, gradient-based model: support vector machines.

Chapter 12. Kernel Models and SVM

In the *Binomial classification* section of [Chapter 9, Regression and Regularization](#), you learned the concept of hyperplanes that segregate observations into two classes. These hyperplanes are also known as linear decision boundaries. In the case of the logistic regression, the datasets must be linearly separated. This constraint is particularly an issue for problems with many features that are nonlinearly dependent (high dimension models).

Support vector machines (SVMs) overcome this limitation by estimating the optimal separating hyperplane using kernel functions.

This chapter introduces kernel functions; binary support vectors classifiers, one-class SVMs for anomaly detection, and support vector regression.

In this chapter, you will answer the following questions:

- What is the purpose of kernel functions?
- What is the concept behind the maximization of margin?
- What is the impact of some of the SVM configuration parameters and the kernel method on the accuracy of the classification?
- How do you apply the binary support vector classifier to estimate the risk for a public company to curtail or eliminate its dividend?
- How do you detect outliers with a one-class support vector classifier?
- How does the support vector regression compare to the linear regression?

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 [12:1].

Overview

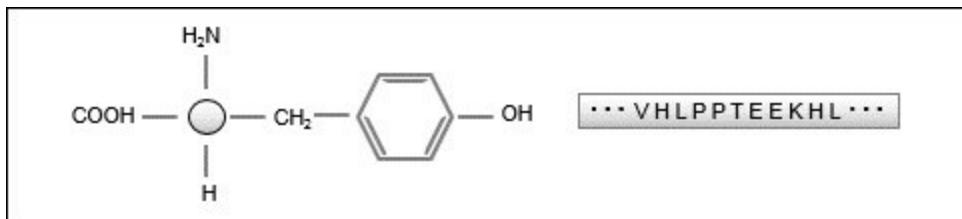
The concept of kernel methods may be a bit odd at first to a novice. Let's consider the simple case of the classification of proteins. Proteins have different lengths and composition, but this does not prevent scientists from classifying them [12:2].

Note

Proteins:

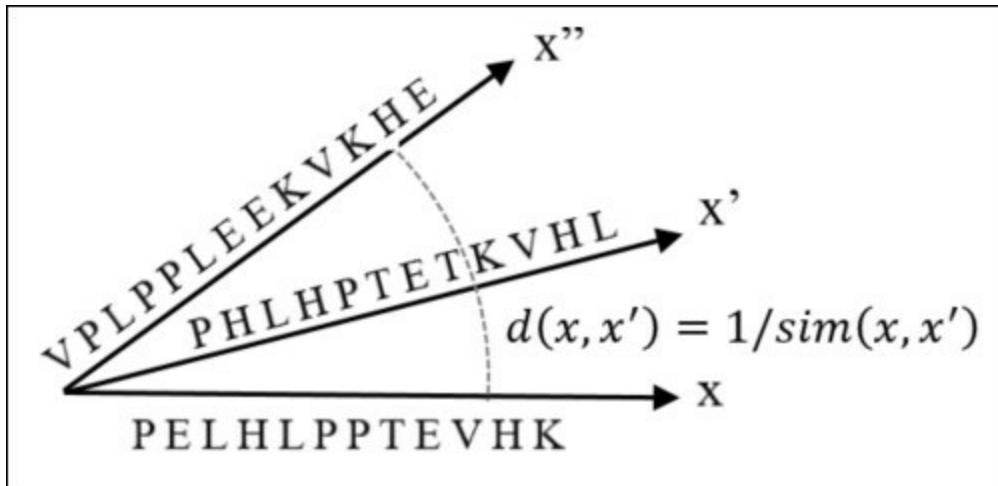
Proteins are polymers of amino acids joined together by peptide bonds. They are composed of a carbon atom bonded to a hydrogen atom, another amino acid, or a carboxyl group.

A protein is represented using a traditional molecular notation with 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 requires 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 to establish that they belong to the same class. You only need to compare each element of each sequence, one by one, and compute the similarity. For the same reason, the estimation of the similarity does not require the two proteins to have the same length.

In this example, we do not have to assign a numerical value to each element of the annotation. Let's consider an element of the protein annotation as its character c and position p (for example, K, 4). The dot product of the two protein annotations x and x' of the respective lengths n and n' are defined as the number of identical elements (character and position) between the two annotations divided by the maximum length between the two annotations (M1):

$$sim(x_{cp}, x'_{c'p'}) = \frac{1}{mx} \sum_{i=1}^{mx} (c = c') \cap (p = p') \quad mx = \max(n, n')$$

The computation of the similarity for the three proteins produces the result as follows: $sim(x, x') = 6/12 = 0.50$, $sim(x, x'') = 3/13 = 0.23$, $sim(x', x'') = 4/13 = 0.31$.

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.

Note

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 when trying 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.

Note

RBF terminology:

In this presentation and the library used in its implementation, the radial basis function is a synonym to the Gaussian kernel function. However, RBF also refers to the family of exponential kernel functions that encompasses Gaussian, Laplacian, and exponential functions.

The simple linear model for regression consists of the dot product of the regression parameters (weights) and the input data (refer to the *Ordinary least squares regression* section of [Chapter 9, Regression and Regularization](#)).

The model is, in fact, the linear combination of weights and linear combination of inputs. The concept can be extended by defining a general regression model as the linear combination of nonlinear functions, known as basis functions (M2):

$$f(x | w) = w_0 + \sum_{d=1}^D w_d \phi_d(x) \quad \phi_d : \mathbb{R} \rightarrow \mathbb{R}$$

The most commonly used basis functions are the power and Gaussian functions. The kernel function is described as the dot product of the two vectors of the basis function $f(x), f(x')$ of two features vector x and x' . A partial list of kernel methods is as follows:

M3: The generic kernel function

$$K(x, x') = \phi(x) \cdot \phi(x') = \sum_{d=1}^D \phi_d(x) \phi_d(x')$$

M4: The linear kernel:

$$K(x, x') = x^T x' = \sum_{d=1}^D x_i x'_i$$

M5: The polynomial kernel with the slope ?, degree n , and constant c :

$$K(x, x') = (\gamma x^T x' + c)^n \quad \gamma > 0, c \geq 0$$

M6: The sigmoid kernel with the slope ? and constant c :

$$K(x, x') = \tanh(\gamma x^T x' + c) \quad \gamma > 0, c \geq 0$$

M7: The radial basis function kernel with the slope ?:

$$K(x, x') = e^{-\gamma \|x-x'\|^2} \quad \gamma > 0$$

M8: The Laplacian kernel with the slope ?:

$$K(x, x') = e^{-\gamma \|x-x'\|} \quad \gamma > 0$$

M9: 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 [12:3].
- **Smoothing kernels:** This is the nonparametric formulation, averaging density with the nearest neighbor observations [12:4].
- **Reproducible kernel Hilbert spaces:** This is the dot product of finite or infinite basis functions [12:5].

The kernel functions play a very important role in support vector machines for nonlinear problems.

Kernel monadic composition

The concept of kernel function is actually derived from differential geometry and, more specifically, from manifold introduced in the *Non-linear models* section in [Chapter 5, Dimension Reduction](#).

Tip

Optional reading:

This section discusses the concept of monadic representation of kernel. It highlights the functional programming capabilities of Scala and is not required to understand the key concepts in this chapter.

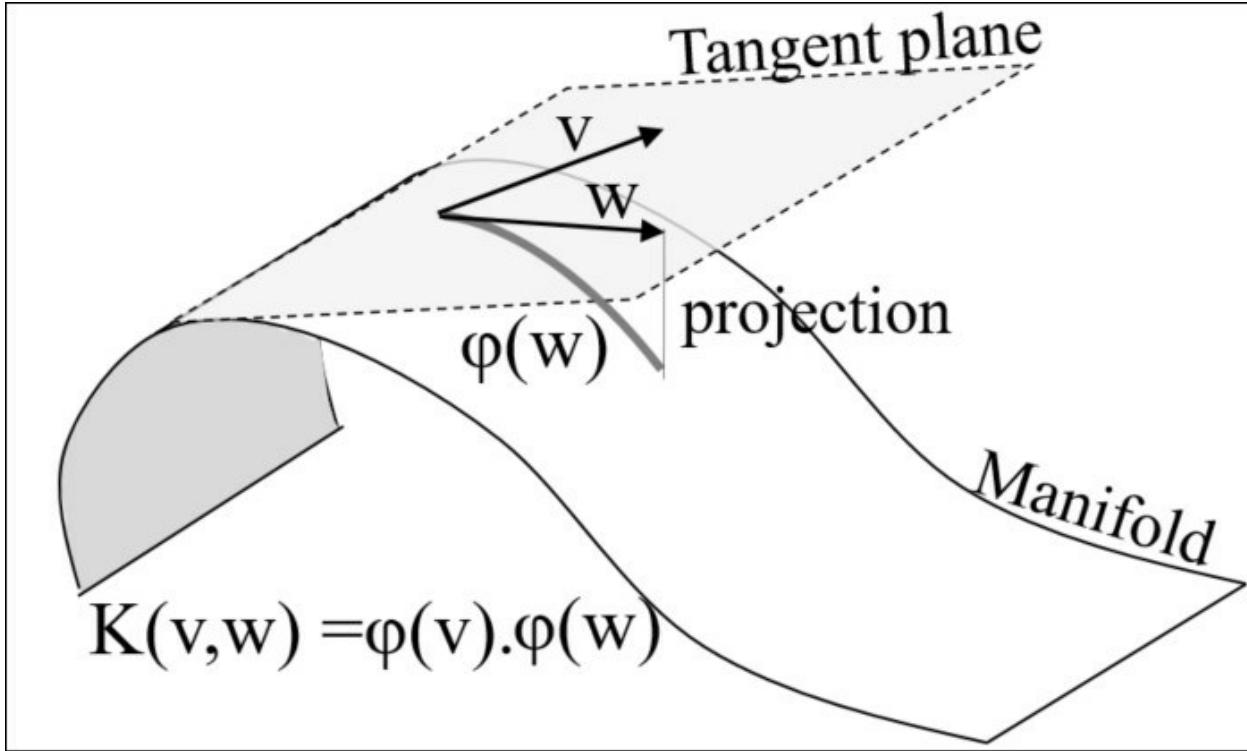
A manifold is a low dimension features space embedded in the observations space of higher dimension. The dot product (or margin) of two observations, known as the *Riemann metric*, is computed on a Euclidean tangent space.

Note

Heat kernel function:

The kernel function on a manifold is computed by solving a heat equation that uses the *Laplace-Beltrami operator*. The *heat kernel* is the solution of the heat differential equation. It associates the dot product to an exponential map.

The kernel function is the composition of the dot product on the tangent space projected on the manifold using an exponential map, as visualized in the following figure:



Visualization of manifold, Riemann metric, and projection of inner product

A kernel function is the composition *g o f* of two functions:

- A function *h* that implements the Riemann *metric* or similarity between two vectors, *v* and *w*
- A function *g* that implements the projection of the similarity *h(v, w)* to the manifold (exponential map)

The class `KF` implements the kernel function as a composition of the function of `g` and `h`:

```

type F1 = Double => Double
type F2 = (Double, Double) => Double

case class KF[G] (val g: G, h: F2) {
  def metric(v: DblVec, w: DblVec)
    (implicit gf: G => F1): Double = //1
    g(v.zip(w).map{ case(_v, _w) => h(_v, _w) }.sum) //2
}
  
```

The class `KF` is parameterized with a type, `G`, that can be converted to a

`Function1[Double, Double]`. Therefore, the computation of the `metric` (dot product) requires an implicit conversion from `G` to `Function1` (line 1). The `metric` is computed by zipping the two vectors, mapping with the `h` similarity function, and summing the resulting vector (line 2).

Let's define the monadic composition for the class `KF`:

```
val kfMonad = new Monad[KF] {
  override def map[G, H](kf: KF[G])(f: G => H): KF[H] =
    KF[H](f(kf.g), kf.h) //3
  override def flatMap[G, H](kf: KF[G])(f: G => KF[H]): KF[H] =
    KF[H](f(kf.g).g, kf.h)
}
```

The creation of the instance, `kfMonad`, overrides the `map` and `flatMap` method defined in the generic `_Monad` trait described in the *Abstraction* section of [Chapter 1, Getting Started](#). The implementation of the `unit` method is not essential to the monadic composition and is therefore omitted.

The function argument of the `map` and `flatMap` methods applies to the exponential map function `g` only (line 3). The composition of two kernel functions $kf1 = g1 \circ h$ and $kf2 = g2 \circ h$ produces a kernel function $kf3 = g2 \circ (g1 \circ h) = (g2 \circ g1) \circ h = g3 \circ h$.

Note

Interpretation of kernel functions monadic composition:

The visualization of the monadic composition of kernel functions on the manifold is quite intuitive. The composition of two kernel functions consists of composing their respective projection or exponential map functions `g`. The function `g` is directly related to the curvature of the manifold around the data point for which the metric is computed. The monadic composition of the kernel functions attempts to adjust the exponential map to fit the curvature of the manifold.

The next step is to define an `implicit` class to convert a kernel function, of the type `KF`, to its monadic representation so it can access the `map` and `flatMap` methods (line 4):

```

implicit class kF2Monad[G](kf: KF[G]) { //4
  def map[H](f: G => H): KF[H] = kfMonad.map(kf)(f)
  def flatMap[H](f: G => KF[H]): KF[H] = kfMonad.flatMap(kf)(f)
}

```

Let's implement the radial basis function, `RBF`, and the polynomial kernel function, `Polynomial`, by defining their respective `g` and `h` functions. The parameterized type for the kernel function is simply `Function1[Double, Double]`:

```

class RBF(s2: Double) extends KF[F1] {
  (x: Double) => exp(-0.5*x*x/s2),
  (x: Double, y: Double) => x - y

class Polynomial(d: Int) extends KF[F1] {
  (x: Double) => pow(1.0+x, d),
  (x: Double, y: Double) => x*y

```

Here is an example of composition of two kernel functions, a RBF kernel `kf1` with standard deviation of 0.6 (line 5) and a polynomial kernel `kf2` with a degree 3 (line 6).

```

val v = Vector[Double](0.5, 0.2, 0.3)
val w = Vector[Double](0.1, 0.7, 0.2)

val composed = for {
  kf1 <- new RBF(0.6) //5
  kf2 <- new Polynomial(3) //6
} yield kf2
composed.metric(v, w) //7

```

Finally, the `metric` is computed on the `composed` kernel functions (line 7).

Tip

Kernel functions in SVM:

Our implementation of the support vector machine uses the kernel function included in the LIBSVM library.

The support vector machine (SVM)

An 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 9, Regression and Regularization](#)). The main difference is that the support vector machine computes the optimum separating hyperplane between groups or classes of observations. The hyperplane is indeed the equation that represents the model generated through training.

Tip

Optional mathematical formulation:

SVMs are formulated as a convex optimization problem. The mathematical foundation of the related algorithms is described in this chapter for reference and is not required for understanding the kernel and SVM models.

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:

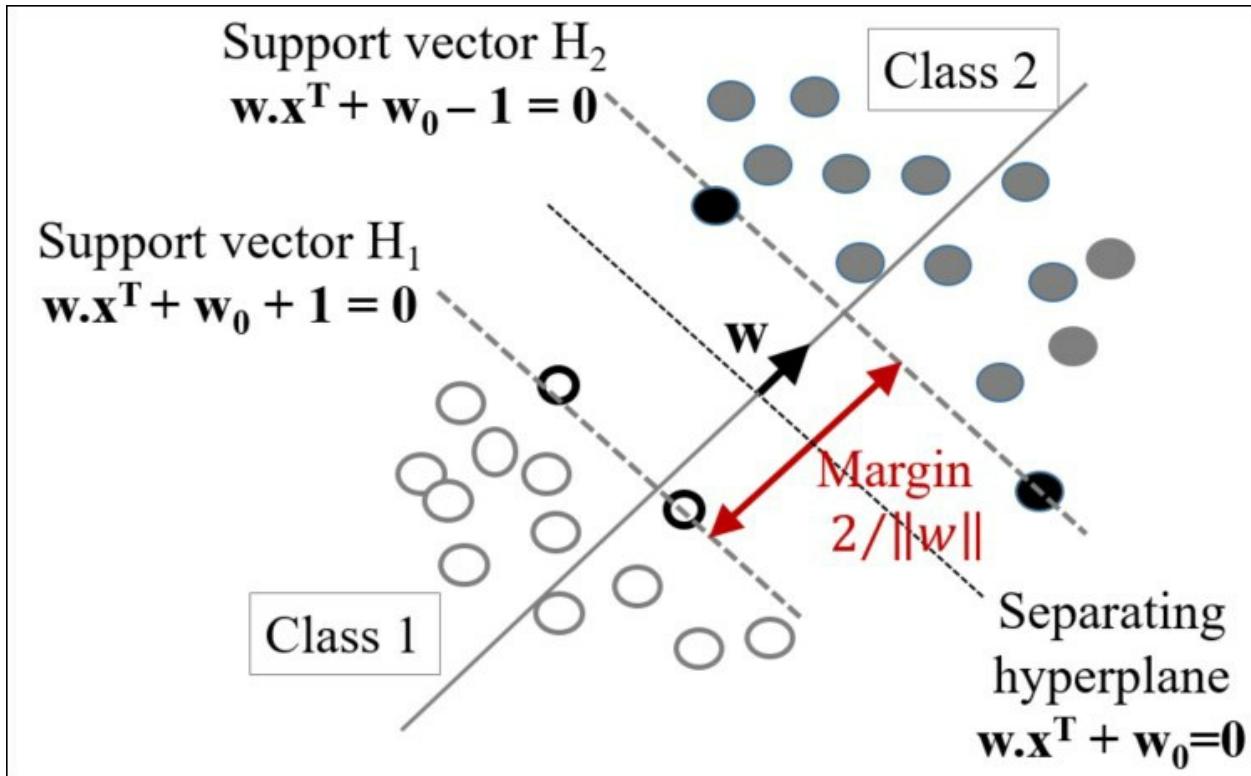
- Naturally segregated in the features space (the separable case)
- Intermingled with overlap (the non-separable case)

It is easy to understand the concept of an optimal separating hyperplane in case 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 two-dimensional (x, y) set of observations with two classes, C_1 and C_2 . The label y has the value -1 or +1.

The equation for the separating hyperplane is defined by the linear equation, $y = w \cdot x^T + w_0$, 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 ($H_2 : w \cdot x^T + w_0 - 1$). The planes H_1 and H_2 are the support vectors:



Visualization of the hard margin in the support vector machine

In the separable case, the support vectors fully segregate the observations into two distinct classes. The margin between the two support vectors is the same for all the observations and is known as the **hard margin**.

Separable case:

M1: Support vectors equation w is represented as:

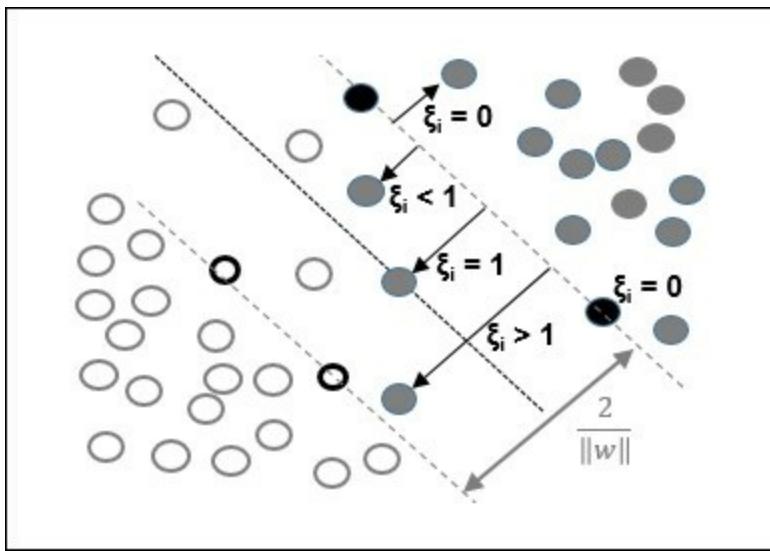
$$y_i(w^T x + w_0) \geq 1 \quad \forall i$$

M2: 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 non-separable case (soft margin)

In the non-separable 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_i , also known as the slack variable, increases if the outlier is further away from the support vector:



Visualization of the hard margin in the support vector machine

The observations that belong to the appropriate (or own) class do not have to be penalized. The condition is similar to the hard margin, which means that the slack ξ_i is null. This technique penalizes the observations that belong to the class, but are located beyond its support vector; the slack ξ_i 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.

Note

Non-separable case:

M3: Optimization of the soft-margin for a linear SVM with C formulation:

$$\min_{w, \xi} \left\{ \frac{w^T w}{2} + C \sum_{i=0}^{n-1} \xi_i \right\}$$

$$\xi_i \geq 0, \quad y_i (w^T x + w_0) \geq 1 - \xi_i \quad \forall i$$

Note

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 9, Regression and Regularization](#)). 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 $\gamma = 1/2C$.

The problem can be reformulated as the minimization of a function known as *the primal problem* [12:6].

M4: Primal problem formulation of the support vector classifier using the L_2 regularization:

$$\min_{w, w_0} \left\{ \frac{w^T w}{2} + C \sum_{i=0}^{n-1} \mathcal{L}_i \right\} \mathcal{L}_i = |1 - y_i (w^T x + w_0)|$$

The C penalty factor is the inverse of the L_2 regularization factor. The loss function L is 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 non-separable case.

The γ -SVM (or Nu-SVM) is an alternative formulation to the C-SVM. The formulation is more descriptive than C-SVM; γ represents the upper bound of the training observations that are poorly classified and the lower bound of the

observations on the support vectors [12:7].

M5: ?-SVM formulation of a linear SVM using the L₂ regularization:

$$\min_{w, \rho, \xi} \left\{ \frac{w^T w}{2} - \rho + \frac{1}{vn} \sum_{i=0}^{n-1} \xi_i \right\}$$
$$\xi_i \geq 0, \quad y_i (w^T x + w_0) \geq \rho - \xi_i \quad \forall i$$

Here, ? is a margin factor used as an optimization variable.

The C-SVM formulation is used throughout the chapters for the binary, one class supports vector classifier as well as the support vector regression.

Note

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 $O(N^2)$ (refer to *Performance Considerations*).

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 non-separable) 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:

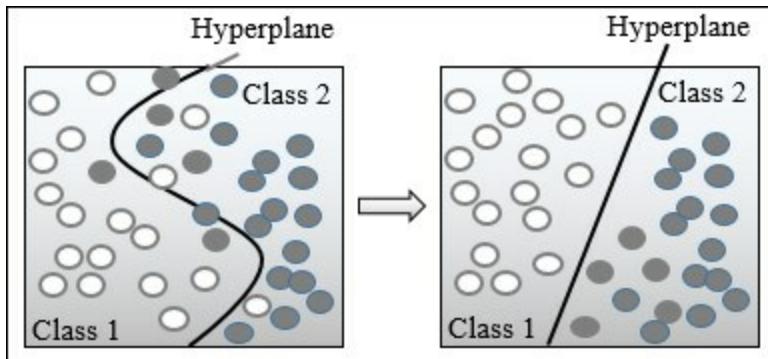


Illustration of the kernel trick in SVM

The nonlinear SVM is implemented using a basis function, $f(x)$. The

formulation of the nonlinear C-SVM resembles the formulation of the linear case. The only difference is the constraint along the support vector, using the basis function, f (M6):

$$y_i(w^T \phi(x) + w_0) \geq 1 - \xi_i \quad \xi_i \geq 0 \quad \forall i$$

The minimization of $w^T f(x)$ in the preceding equation requires the computation of the inner product $f(x)^T f(x)$. 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, $y_i f(x_i)$, 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 [12:8].

M7: Optimal hyperplane for the SVM dual problem:

$$w^* = \sum_{i=0}^{n-1} \alpha_i y_i \phi(x_i)$$

M8: Hard margin formulation for the SVM dual problem:

$$y_i(w^T \phi(x) + w_0) = y_i \left(\sum_{i=0}^{n-1} \alpha_i y_i K(x_i, x) + w_0 \right) \geq 1$$

$$K(x_i, x) = \phi(x_i) \phi(x) \quad \forall_i$$

The kernel trick

The transformation $(x, x') \Rightarrow K(x, x')$ maps a nonlinear problem into a linear problem in a higher dimensional space. It is known as the *kernel trick*.

Let's consider, for example, the polynomial kernel defined in the first section

with a degree $d=2$ and coefficient of $C_0 = 1$ in a two-dimension space. The polynomial kernel function on two vectors, $x=[x_1, x_2]$ and $x'=[x'_1, x'_2]$, is decomposed into a linear function in a dimension 6 space:

$$\begin{aligned}
 K(x_i, x) &= (1 + x^T x')^2 \\
 &= 1 + 2x_1 x'_1 + 2x_2 x'_2 + 2x_1 x'_1 x_2 x'_2 + (x_1 x'_1)^2 + (x_2 x'_2)^2 \\
 &= \phi_1(x) \phi_1(x') + \phi_2(x) \phi_2(x') + \phi_3(x) \phi_3(x') + \dots \\
 \phi_2(x) &= 1, \phi_2(x) = \sqrt{2}x_1, \phi_3(x) = \sqrt{2}x_2, \phi_4(x) = x_1^2 \dots
 \end{aligned}$$

Tip

Convexity:

Support vector machines can formulate the training of a model as a nonlinear optimization for which the objective function is convex. The drawback is that the kernelization of the SVM may result in many basis functions (higher dimensions). One solution is to reduce the number of basis functions through parameterization, so these functions can adapt to different training sets.

Support vector classifier (SVC)

SVMs can be applied to classification, anomaly 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 (two-class) support vector classifier. The implementation uses the *LIBSVM* library created by *Chih-Chung Chang* and *Chih-Jen Lin* from the National Taiwan University [12:9].

LIBSVM

The library was originally written in C before being ported to Java. It can be downloaded from <http://www.csie.ntu.edu.tw/~cjlin/libsvm> as a ZIP or tar.gz file. The library includes the following classifier modes:

- Support vector classifiers (C-SVC, ?-SVC, and one-class SVC)
- Support vector regression (?-SVR and e-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>.

Note

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 [12:10]. *Thorsten Joachims'* *SVMLight* [12:11]

Spark/MLlib includes two Scala implementations of SVM using resilient distributed datasets (refer to [Chapter 17, Apache Spark](#)). 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`: Defines the parameters of the model created during training
- `svm_node`: Models the element of the sparse matrix Q, used in the maximization of the margins
- `svm_parameters`: 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`: 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`: Implements algorithms used in training, classification, and regression

The library also includes template programs for training, prediction, and normalization of datasets.

Tip

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 of limitations, LIBSVM is a 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.

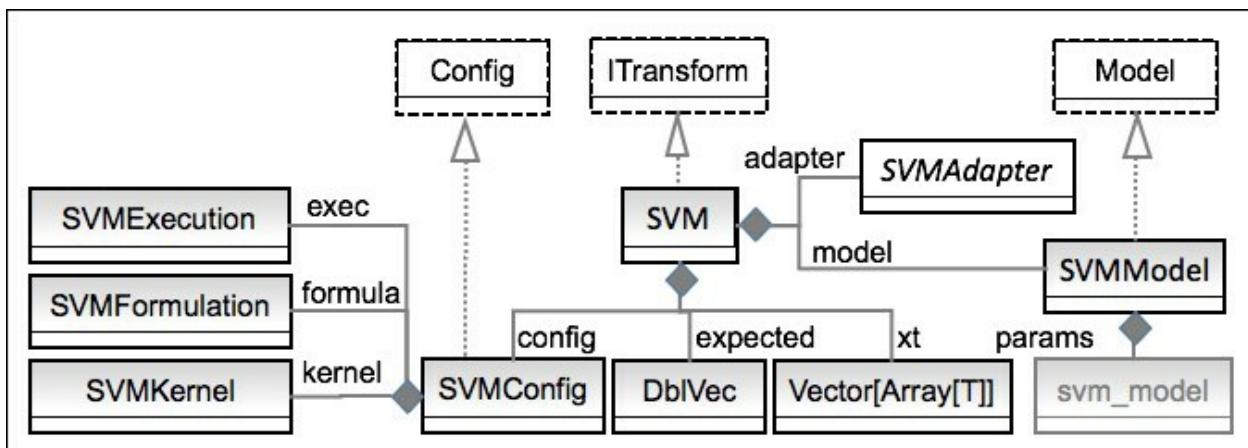
Design

The implementation of the support vector machine algorithm uses the design template for classifiers (refer to the *Design template for classifier* section of the *Appendix*).

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.
- An object `SVMAAdapter` interfaces with the internal LIBSVM data structures and methods.
- The support vector machine class `SVM` is implemented an implicit data transformation of type `ITransform`. It has three parameters: the configuration wrapper of the type `SVMConfig`, the features/time series of the type `XVSeries`, 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 non-separable 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:



UML class diagram for the support vector machine

The UML diagram omits the helper traits and classes such as `Monitor` or Apache commons math components.

Configuration parameters

LIBSVM exposes several 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

The 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 by a class hierarchy with `SVMFormulation` as base trait:

```
sealed trait SVMFormulation extends SVMConfigItem {  
    def update(param: svm_parameter): Unit  
}
```

The list of formulations 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 must 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 γ -SVM, l -SVM, and e -SVM respectively for regression models.

The SVM kernel function

Next, you need to specify the kernel functions by defining and implementing the `SVMKernel` trait:

```
sealed trait SVMKernel extends SVMConfigItem {  
    override def update(param: svm_parameter): Unit  
}
```

Once again, there are a limited number of kernel functions supported in LIBSVM. Therefore, the hierarchy of kernel functions is sealed. The following code snippet configures the radius basis function kernel, `RbfKernel`, as an example of 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 exp(-gamma*sqrt(sum));
```

5. Add the implementation of the Laplace kernel function in the `svm.k_function` method by modifying the existing implementation of RBF (`distanceSqr`).
6. Rebuild the `libsvm.jar` file.

The SVM execution

The `SVMExecution` class defines the configuration parameters for the execution of the training of the model, namely, the convergence factor, `eps` for the optimizer (line 2), the size of the cache `cacheSize` (line 1), 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 //1
    param.eps = eps //2
  }
}
```

Cross-validation is performed only if the `nFolds` value is greater than 1. 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) //3
kernel.update(param) //4
exec.update(param) //5
}

```

The class `SVMConfig` delegates the selection of the formula to the `SVMFormulation` class (line 3), selection of the kernel function to `SVMKernel` class (line 4), and the execution parameters to `SVMExecution` class (line 5). The sequence of update calls initializes the LIBSVM list of configuration parameters.

Interface to LIBSVM

We need to create an adapter object to encapsulate the invocation to LIBSVM. The object `SVMAdapter` hides the LIBSVM internal data structures, `svm_model` and `svm_node`:

```

object SVMAdapter {
  type SVMNodes = Array[Array[svm_node]]
  class SVMProblem(numObs: Int, expected: Array[Double]) //6

  def createSVMNode(dim: Int,
                    x: Array[Double]): Array[svm_node] //7
  def predictSVM(model: SVMModel, x: Array[Double]): Double //8
  def crossValidateSVM(
    problem: SVMProblem, //9
    param: svm_parameter,
    nFolds: Int,
    expected: Array[Double])
  def trainSVM(
    problem: SVMProblem, //10
    param: svm_parameter): svm_model
}

```

The object `SVMAdapter` is a single-entry point to LIBSVM for training, validating an SVM model, and executing predictions:

- `SVMProblem` wraps the definition of the training objective or problem in LIBSVM, using the labels or `expected` values (line 6)
- `createSVMNode` creates a new computation node for each observation `x` (line 7)

- `predictSVM` predicts the outcome of a new observation x given a model, `svm_model` generated through training (line 8)
- `crossValidateSVM` validates the model, `svm_model` with a `nFold` training – validation sets (line 9)
- `trainSVM` executes the training configuration, problem (line 10).

Tip

svm_node:

The LIBSVM `svm_node` Java class is defined as a pair of index of the feature in the observation array and its value:

```
public class svm_node implements java.io.Serializable {
    public int index;
    public double value;
}
```

The `SVMAdapter` methods are described in the next section.

Training

The model for SVM is defined with the following two components:

- `svm_model` that is the SVM model parameters defined in LIBSVM
- `accuracy` that is the accuracy of the model computed during cross-validation:

```
case class SVMModel(svmmodel: svm_model, accuracy: Double) extends Model {
    lazy val residuals: Array[Double] = svmmodel.sv_coef(0)
}
```

The residuals $r = y - f(x)$ are computed in the LIBSVM library.

Tip

Accuracy in SVM model:

You may wonder why the value of the accuracy is a component of the model. The accuracy component of the model provides the client code with a quality metric associated with the model. Integrating the accuracy into the model allows the user to make informed decision in accepting or rejecting the model. The accuracy is stored in the model file for subsequent analysis.

Next, let's create the first support vector classifier for the two-class problems. The SVM class implements the monadic data transformation `ITransform` that generates implicitly a model from a training set as described in the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#) (line 11).

The constructor for SVM follows the template described in the *Design template for classifiers* section of *Appendix*. It takes a configuration of type `SVMConfig`, a set of observations `xt` of type `Array[T]`, and a set of labels (expected values) of type, `DblVec` as arguments (line 12):

```
class SVM[T: ToDouble] (
  config: SVMConfig,
  xt: Vector[Array[T]],
  expected: DblVec) //12
extends ITransform[Array[T], Double] { //11

  val normEPS = config.eps*1e-7 //13
  val model: Option[SVMModel] = train //14

  def accuracy: Option[Double] = model.map(_.accuracy) //15
  def mse: Option[Double] //16
  def margin: Option[Double] //17
}
```

The `normEPS` is used for rounding errors in the computation of the margin (line 13). The model of type `SVMModel` is generated through training by the `SVM` constructor (line 14). The last four methods are used to compute the parameters of the model `accuracy` (line 15), the mean square of errors, `mse` (line 16), and the `margin` (line 17).

Let's look at the training method, `train`:

```
def train: Option[SVMModel] = Try {
  val problem = new SVMProblem(xt.size, expected.toArray) //18
  val dim = dimension(xt)
  xt.indices.foreach(
```

```

    n => problem.update(n, createNode(dim, x(n)).map((c(_)))) //12
)
new SVMModel(
  trainSVM(problem, config.param), accuracy(problem) //20
)
}.toOption

```

The `train` method creates a `SVMProblem` that provides LIBSVM with training components (line 18). The purpose of the `SVMProblem` class is to manage the definition of training parameters implemented in LIBSVM, as follows:

```

class SVMProblem(numObs: Int, expected: Array[Double]) {
  val problem = new svm_problem //21
  problem.l = numObs
  problem.y = expected
  problem.x = new SVMNodes(numObs)

  def update(n: Int, node: Array[svm_node]): Unit =
    problem.x(n) = node //22
}

```

The arguments of the `SVMProblem` constructor, the number of observations and the labels or expected values are used to initialize the corresponding data structure, `svm_problem` in LIBSVM (line 21). The method `update` maps each observation, defined as an array of `svm_node` to the problem (line 22).

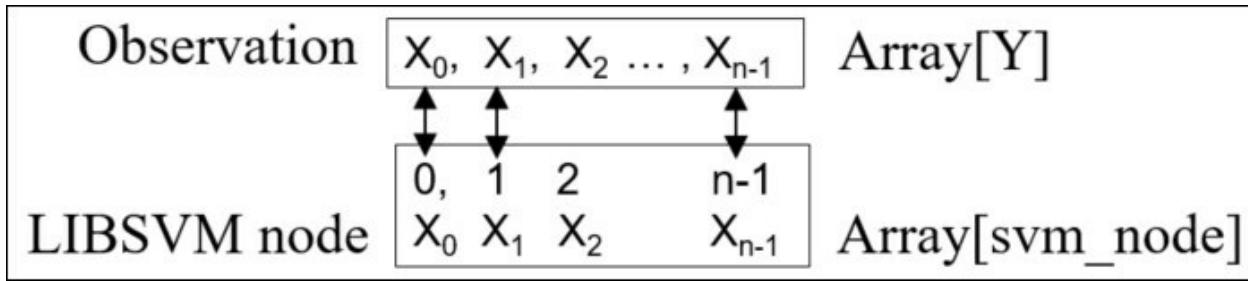
The method `createNode` creates an array of `svm_node` from an observation. A `svm_node` in LIBSVM is the pair of the index `j` of a feature in an observation (line 23) and its value `y` (line 24):

```

def createNode(dim: Int, x: Array[Double]): Array[svm_node] =
  x.indices./:(new Array[svm_node](dim)){(newNode, j) =>
    val node = new svm_node
    node.index= j //23
    node.value = x(j) //24
    newNode(j) = node
    newNode
}

```

The mapping between an observation and a LIBSVM node is illustrated in the following diagram:



Indexing of observations using LIBSVM

The method `trainSVM` pushes the training request with a well-defined problem and configuration parameters to LIBSVM by invoking the `svm_train` method (line 26):

```
def trainSVM(problem: SVMProblem,
              param: svm_parameter): svm_model =
  svm.svm_train(problem.problem, param) //26
```

The `accuracy` is the ratio of true positive plus the true negative over the size of the test sample (refer to the *Key metrics* section of [Chapter 2, Data Pipelines](#)). 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:

```
def accuracy(problem: SVMProblem): Double =
  if( config.isCrossValidation ) {
    val target = crossValidateSVM(problem, config.param, //27
                                  config.nFolds, expected.size)

    target.view.zip(expected.view)
      .count{ case(x, y) => abs(x-y) < config.eps } //28
      .toDouble/expected.size
  }
  else 0.0
```

The call to the `crossValidateSVM` method of `SVMAdapter` forwards the configuration and execution of the cross validation with `config.nFolds` (line 27):

```

def crossValidateSVM(
    problem: SVMProblem,
    param: svm_parameter,
    nFolds: Int,
    size: Int) = {
    val target = Array.fill(size)(0.0)
    svm.svm_cross_validation(
        problem.problem, param, nFolds, target)
}
target
}

```

The Scala `filter` weeds out the observations that were poorly predicted (line 28). This minimalist implementation is good enough to start exploring the support vector classifier.

Classification

The implementation of the classification method `|>` for the `SVM` class follows the same pattern as the other classifiers. It invokes the `predictSVM` method in `SVMAdapter` that forwards the request to LIBSVM (line 29):

```

override def |> : PartialFunction[Array[T], Try[Double]] = {
    case x: Array[T] if(x.size == dimension(xt) && isModel) =>
        Try(predictSVM(model.get, x.map(c(_)))) //29
}

```

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 `SVM` class, as follows:

```

def margin: Option[Double] = model.map{ m =>
    1.0/sqrt(m.residuals.reduce(_ * _))
}

```

The first instruction computes the sum of the squares, `wNorm`, of the residuals $r = y - f(x|w)$. The margin is ultimately computed if the sum of squares is significant enough to avoid rounding errors.

The margin is evaluated using an artificially generated time series and labeled data. First, we define the method to evaluate the margin for a specific value of the penalty (inversed regularization coefficient) factor C :

```
val GAMMA = 0.8; val CACHE_SIZE = 1<<8
val NFOLDS = 2; val EPS = 1e-5

def evalMargin(features: Vector[Array[Double]],
              expected: DblVec,
              c: Double): Int = {
  val execEnv = SVMExecution(CACHE_SIZE, EPS, NFOLDS)
  val config = SVMConfig(new CSVCFormulation(c),
                        new RbfKernel(GAMMA),
                        execEnv
                      )
  val svc = SVM[Double](config, features, expected)
  svc.margin.map(_.toString) //30
}
```

The method `evalMargin` uses the execution parameters, `CACHE_SIZE`, `EPS`, and `NFOLDS`. The execution displays the value of the margin for different values of C (line 30). The method is invoked iteratively to evaluate the impact of the penalty factor on the margin extracted from the training of the model. The test uses a synthetic time series to highlight the relation between C and the margin. The synthetic time series created by the method generate consists of two training sets of an equal size, N :

- Data points generated as $y = x(1 + r/5)$ for the label 1 , r being a randomly generated number over the range $[0, 1]$ (line 31)
- Randomly generated data point $y = r$ for the label of -1 (line 32)

Consider the following code:

```
def generate: Option[(Vector[Array[Double]], Array[Double])] = {
  val z = Vector.tabulate(N)(i => {
    val ri = i*(1.0 + 0.2*nextDouble)
    Array[Double](i, ri) //31
  }) ++
  Vector.tabulate(N)(i => Array[Double](i, i*nextDouble))

  normalizeArray(z).map(
    (_, Array.fill(N)(1) ++ Array.fill(N)(-1)))
  .toOption //32
```

}

The `evalMargin` method is executed for values C ranging from 0.1 to 5:

```
generate.map(y =>
  (0.1 until 5.0 by 0.1)
    .flatMap(evalMargin(y._1, y._2, _)).mkString("\n")
)
```

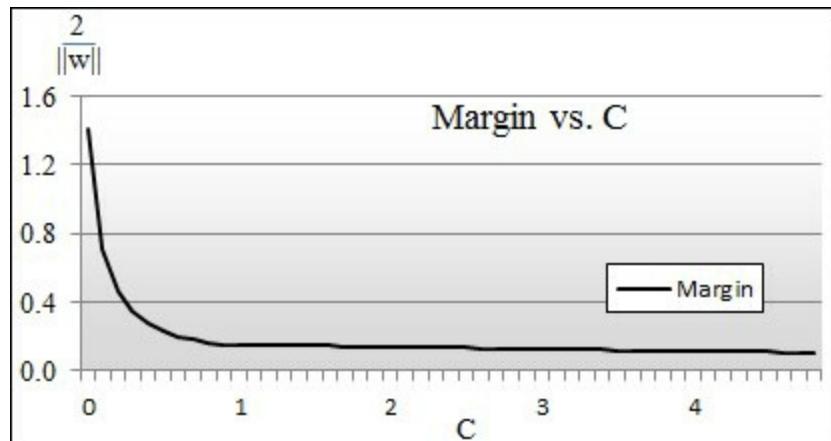
Tip

val versus final val:

There is a difference between a `val` and a `final val`. A non-final 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 factor for a support vector classifier

As expected, the value of the margin decreases as the penalty term C

increases. The C penalty factor is related to the L_2 regularization factor λ as $C \sim 1/\lambda$. A model with a large value of C has a high variance and a low bias, while a small value of C will produce lower variance and a higher bias.

Note

Optimizing C-penalty:

The optimal value for C is usually evaluated through cross-validation by varying C in incremental powers of 2: $2^n, 2^{n+1} \dots [12:12]$.

Kernel evaluation

The next test consists of comparing the impact of the kernel function on the accuracy of the prediction. Once again, a synthetic time series is generated to highlight the contribution of each kernel. The test code uses the runtime prediction or classification method `|>` to evaluate the different kernel functions. Let's create a method to evaluate and compare these kernel functions. All we need is the following (line 33):

- A training set, `xt`, of type `Vector[Array[Double]]`
- A test set, `test`, of type `Vector[Array[Double]]`
- A set of `labels` for the training set, taking the value 0 or 1
- A kernel function `kF`

Consider the following code:

```
val C = 1.0
def accuracy(  
    xt: Vector[Array[Double]],  
    test: Vector[Array[Double]],  
    labels: DblVec,  
    kF: SVMKernel): Double = { //33  
  
    val config = SVMConfig(new CSVCF(C), kF) //34  
    val svc = SVM[Double](config, xt, labels)  
    val pfnSvc = svc |> //35  
  
    test.zip(labels).count{ case(x, y) =>
```

```

    if(pfnSvc.isDefinedAt(x)) pfnSvc(x).get == y else false
  }.toDouble/test.size //36
}

```

The configuration, `config`, of the SVM uses C penalty factor 1, the C-formulation, and the default execution environment (line 34). The predictive partial function `pfnSvc` (line 35) is used to compute the predictive values for the test set. Finally, the method `accuracy` counts the number of successes for which the predictive values match the labeled or expected values. The accuracy is computed as the ratio of the successful prediction over the size of the test sample (line 36).

In order to compare the different kernels, let's generate three datasets of the size $2N$ for a binomial classification using the pseudo-random data generation method `genData`:

```

def genData(
  variance: Double,
  mean: Double): Vector[Array[Double]] = {

  val rGen = new Random(System.currentTimeMillis)
  Vector.tabulate(N) { _ =>
    rGen.setSeed(rGen.nextLong)
    Array[Double](rGen.nextDouble, rGen.nextDouble)
      .map(variance*_ - mean) //37
  }
}

```

The random value is computed through a transformation $f(x) = variance * x = mean$ (line 37). The training and test sets consist of the aggregation of two classes of data points:

- Random data points with `variance a` and `mean b` associated to the label 0.0
- Random data points with `variance a` and `mean 1-b` associated to the label 1.0

Consider the following code for the training set:

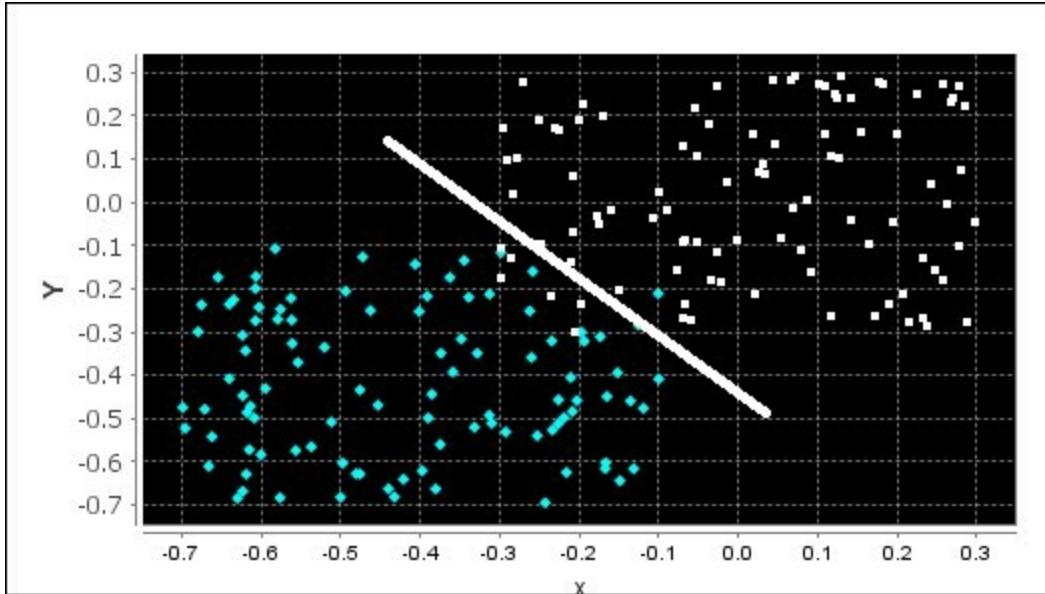
```

val trainSet = genData(a, b) ++ genData(a, 1-b)
val testSet = genData(a, b) ++ genData(a, 1-b)

```

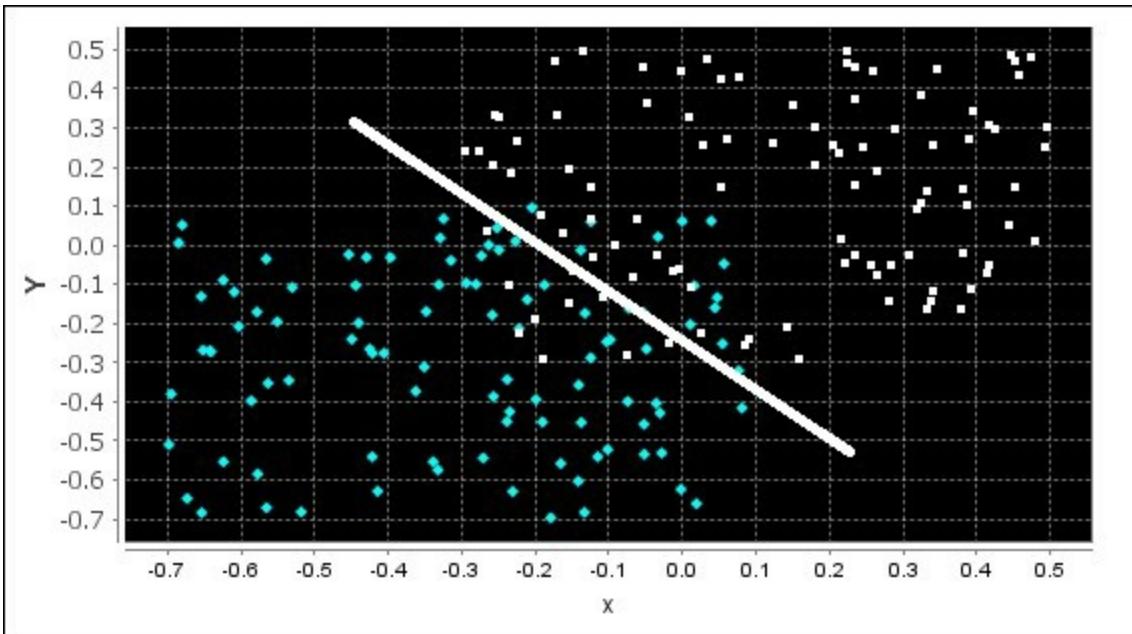
The parameters a , b are selected from two groups of training data points with various degree of separation to illustrate the separating hyperplane.

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:



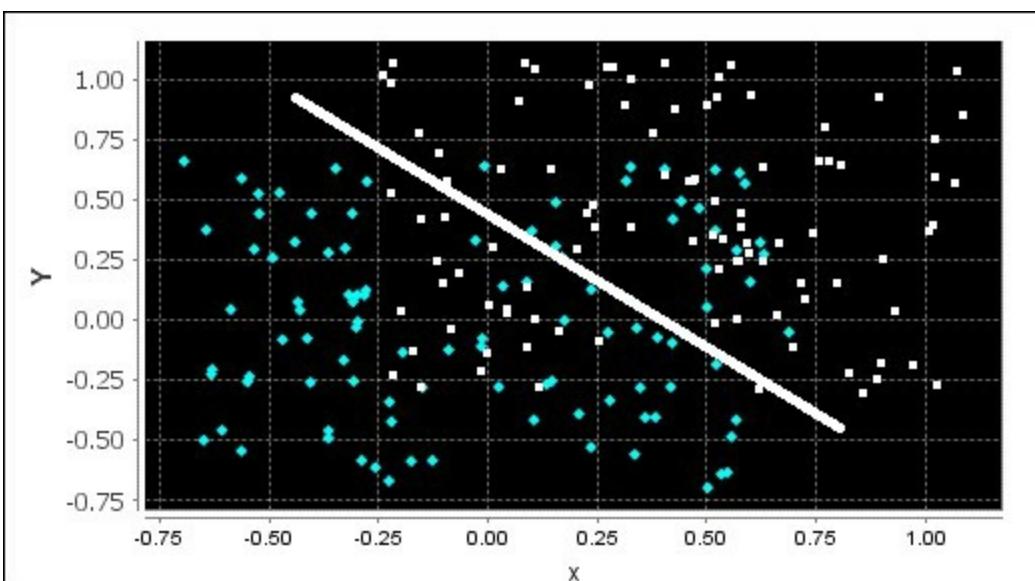
Scatter plot for training and testing sets with $a = 0.6$ and $b = 0.3$

The following chart describes the medium margin; the parameters $a = 0.8$ and $b = 0.3$ generate two groups of observations with some overlap:



Scatter plot for training and testing sets with $a = 0.8$ and $b = 0.3$

The following chart describes the low margin; the two groups of observations in this last training are generated with $a = 1.4$ and $b = 0.3$ and show a significant overlap:



Scatter plot for training and testing sets with $a = 1.4$ and $b = 0.3$

The test set is generated in a similar fashion as the training set, as they are

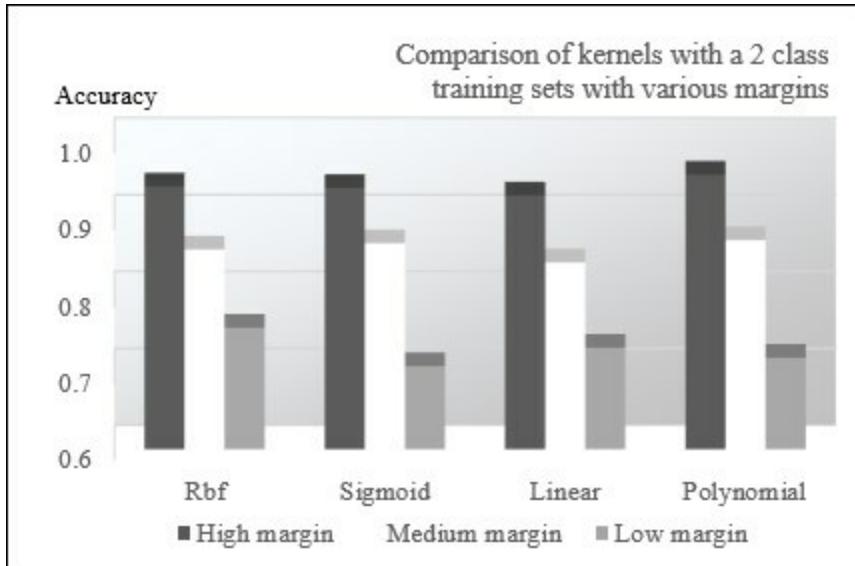
extracted from the same data source:

```
val GAMMA = 0.8; val COEF0 = 0.5; val DEGREE = 2 //38
val N = 100

def compareKernel(a: Double, b: Double) {
    val labels = Vector.fill(N)(0.0) ++ Vector.fill(N)(1.0)

    evalKernel(trainSet, testSet, labels, new RbfKernel(GAMMA))
    evalKernel(trainSet, testSet, labels,
               new SigmoidKernel(GAMMA))
    evalKernel(trainSet, testSet, labels, LinearKernel)
    evalKernel(trainSet, testSet, labels,
               new PolynomialKernel(GAMMA, COEF0, DEGREE))
}
```

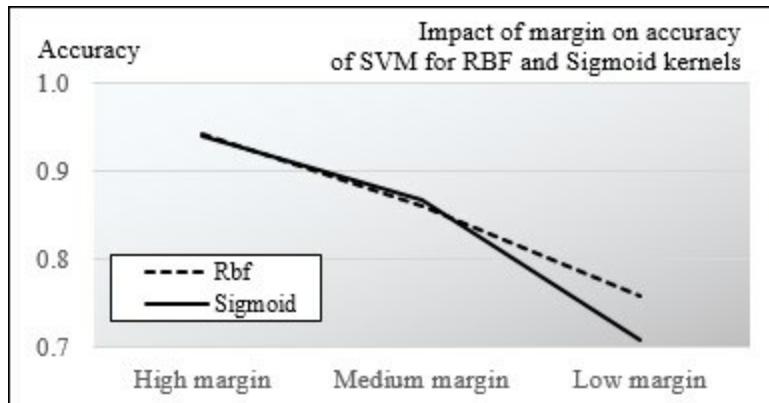
The parameters for each of the four kernel functions are arbitrary selected from textbooks (line 38). The `evalKernel` method defined earlier is applied to the three training sets: high margin ($a = 1.4$), medium margin ($a = 0.8$), and low margin ($a = 0.6$), with each of the four kernels (RBF, sigmoid, linear, and polynomial). The accuracy is assessed by counting the number of observations correctly classified for all the classes for each invocation of the predictor, `|>:`



Comparative chart of kernel functions using synthetic data

Although the different kernel functions do not differ in terms of their impact

on the accuracy of the classifier, you can observe that the RBF and polynomial kernels produce results that are slightly more accurate. As expected, the accuracy decreases as the margin decreases. A decreasing margin is a sign that the cases are not easily separable, affecting the accuracy of the classifier:



Impact of the margin value on the accuracy of RBF and Sigmoid kernel functions

In summary, there are four steps in creating an SVC-based model:

- Select a features-set.
- Select the C-penalty (inverse regularization).
- Select the kernel function.
- Tune the kernel parameters

Note

Test case design:

The test to compare the different kernel methods is highly dependent on the distribution or mixture of data in the training and test sets. The synthetic generation of data in this test case is used for illustrating the margin between classes of observations. Real-world datasets may produce different results.

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.

We need to select any subset of the following financial technical analysis metrics (refer to the *Metrics* section of the *Appendix*):

- 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 values are normalized with values 0 and 1

The labels or expected 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 two-fold validation configuration:

```

val path = "supervised/svm/dividends2.csv"
val C = 1.0
val GAMMA = 0.5
val EPS = 1e-2
val NFOLDS = 2

val extractor = relPriceChange :: debtToEquity :::
  dividendCoverage :: cashPerShareToPrice :: epsTrend :::
  shortInterest :: dividendTrend :::
  List[Array[String] =>Double] () //39

val pfnSrc = DataSource(path, true, false, 1) |> //40
val config = SVMConfig(new CSVCFormulation(C),
  new RbfKernel(GAMMA), SVMExecution(EPS, NFOLDS))

for {
  path <- getPath(dataPath)
  pfnSrc <- DataSource(path, true, false, 1) //40
  input <- pfnSrc.|>(extractor) //41
  obs <- getObservations(input) //42
} yield {
  svc = SVM[Double](config, obs, input.last.toVector)
  show(s"${svc.toString}\naccuracy=${svc.accuracy.getOrElse(-1)}")
}
  
```

The first step is to define the `extractor`, that, is the list of fields to retrieve from the file, `dividends2.csv` (line 39). The partial function `pfnSrc` generated by the `DataSource` transformation class (line 40) converts the input file into a set of typed fields (line 41). An observation is an array of fields. The sequence of observations, `obs`, are generated from the input fields by transposing the matrix observations x features (line 42):

```

def getObservations(input: Vector[Array[Double]]):
  Try[Vector[Array[Double]]] = Try {
    transpose( input.dropRight(1).map(_.toArray) ).toVector
  }
  
```

```
}
```

The test computes the model parameters and the accuracy from the cross-validation during the instantiation of SVM.

Tip

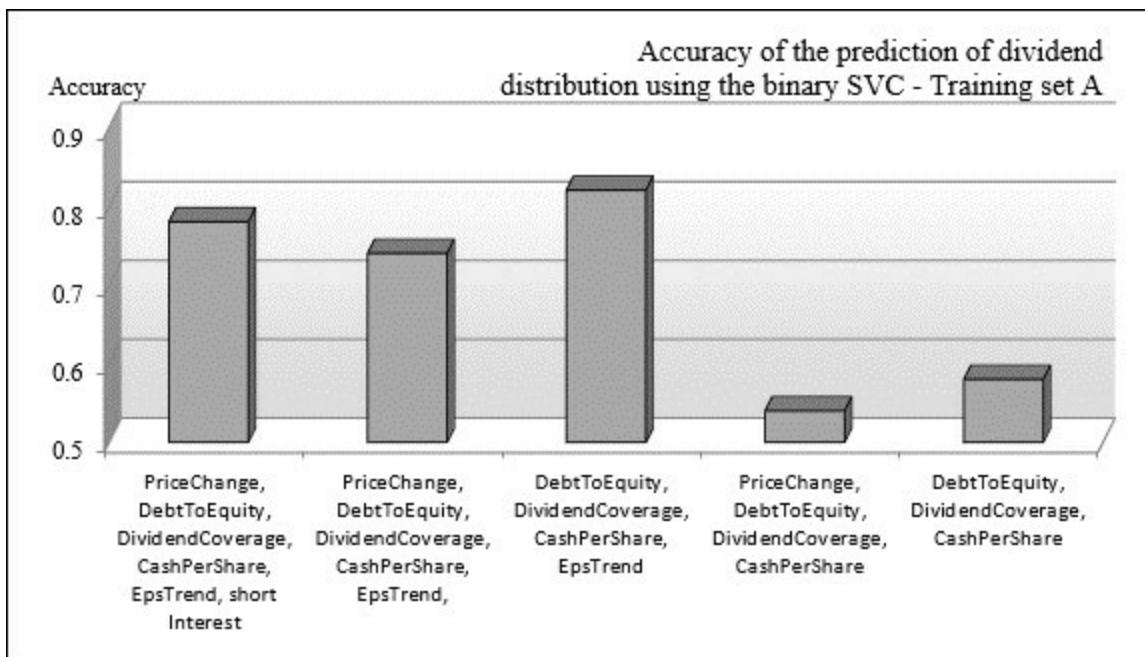
LIBSVM scaling:

LIBSVM supports feature normalization known as scaling, prior to training. The main advantage of scaling is to avoid attributes in greater numeric ranges dominating those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during the calculation. In our examples, we use the normalization method of the time series `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 :: ....
```

The following bar chart displays the various values of the accuracy of the different SVM models, using the training set A.



Comparative study of trading strategies using binary SVC

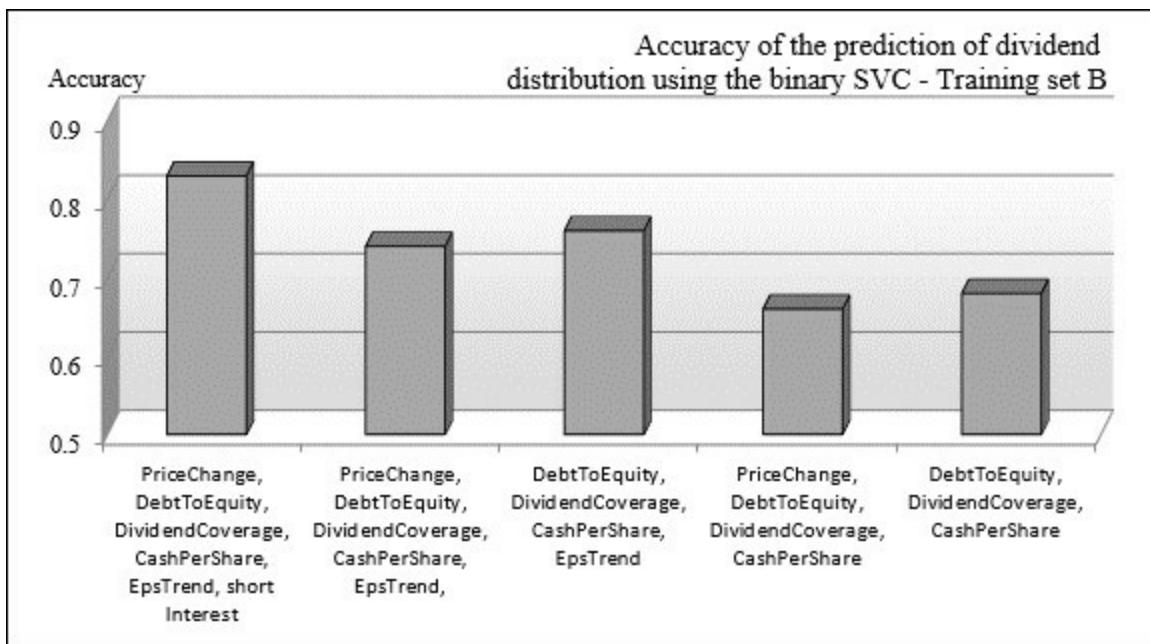
The test demonstrates that the selection of the proper features set is the most critical step in applying the SVM, or any other model for that matter, to classification problems. In this case, the accuracy is also affected by the small size of the training set. The increase in the number of features also reduces the contribution of each specific feature to the loss function.

Note

N-fold cross-validation:

The cross-validation in this test example uses only two folds because the number of observations is small, and you want to make sure that any class contains at least a few observations.

The same process is repeated for the test B whose purpose is to classify companies with decreasing dividends and companies with stable or increasing dividends, as shown in the following graph:



Comparative study of trading strategies using binary SVC

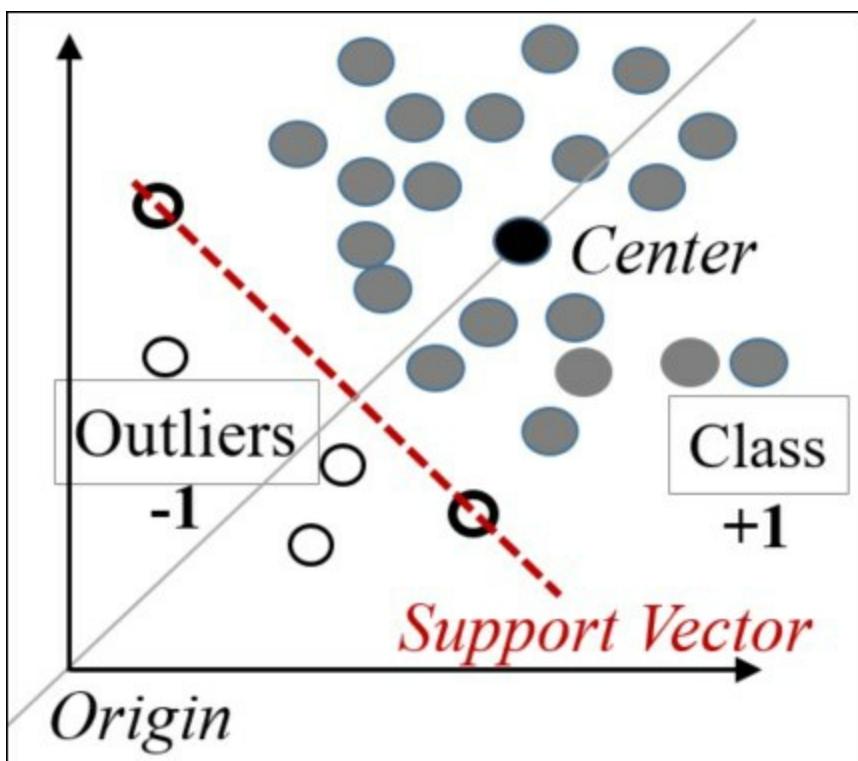
The difference, in terms of accuracy of prediction, between the first three features set and the last two features set in the preceding graph is more pronounced in test A than in test B. In both tests, the feature `eps` (earning per share) trend improves the accuracy of the classification. It is a particularly 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 is the risk a company eliminates its dividend altogether? Such a scenario is rare, and those cases are outliers. A one-class support vector classifier can be used to detect outliers or anomalies [12:13].

Anomaly detection with one-class SVC

The design of the one-class SVC is an extension of the binary SVC. The main difference is that a single class contains most of the baseline (or normal) observations. A reference point, known as the SVC origin, replaces the second class. The outliers (or abnormal) observations reside beyond (or outside) the support vector of the single class:



Visualization of the one-class SVC

The outlier observations have a labeled value of -1, while the remaining training sets are labeled +1. 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/svm/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 binomial SVC. The only difference is the definition of the output labels; -1 for companies eliminating dividends and +1 for all other companies:?

```
val NU = 0.2
val GAMMA = 0.5
val EPS = 1e-3
val NFOLDS = 2

val extractor = relPriceChange :: debtToEquity :::
  dividendCoverage :: cashPerShareToPrice :: epsTrend :::
  dividendTrend :: List[Array[String] =>Double]()

val filter = (x: Double) => if(x == 0) -1.0 else 1.0 //43
val pfnsrC = DataSource(path, true, false, 1) |>
val config = SVMConfig(new OneSVCFormulation(NU), //44
  new RbfKernel(GAMMA), SVMExecution(EPS, NFOLDS))
for {
  pfnsrC <- pfnsrC
  input <- pfnsrC(extractor)
  obs <- getObservations(input)
} yield {
  svc = SVM[Double](
    config, obs, input.last.map(filter(_)).toVector
  )
}
```

The labels or expected data are generated by applying a binary filter to the last field, `dividendTrend` (line 43). The formulation in the configuration has

the type `OneSVCFormulation` (line 44).

The model is generated with an accuracy of 0.821. This level of accuracy should not be a surprise; the outliers (companies that eliminated their dividends) are added to the original dividend `.csv` file. These outliers differ significantly from the baseline observations (companies who have reduced, maintained, or increased their dividend) in the original input file.

Where the labeled observations are available, the one-class SVM is an excellent alternative to clustering techniques.

Note

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:

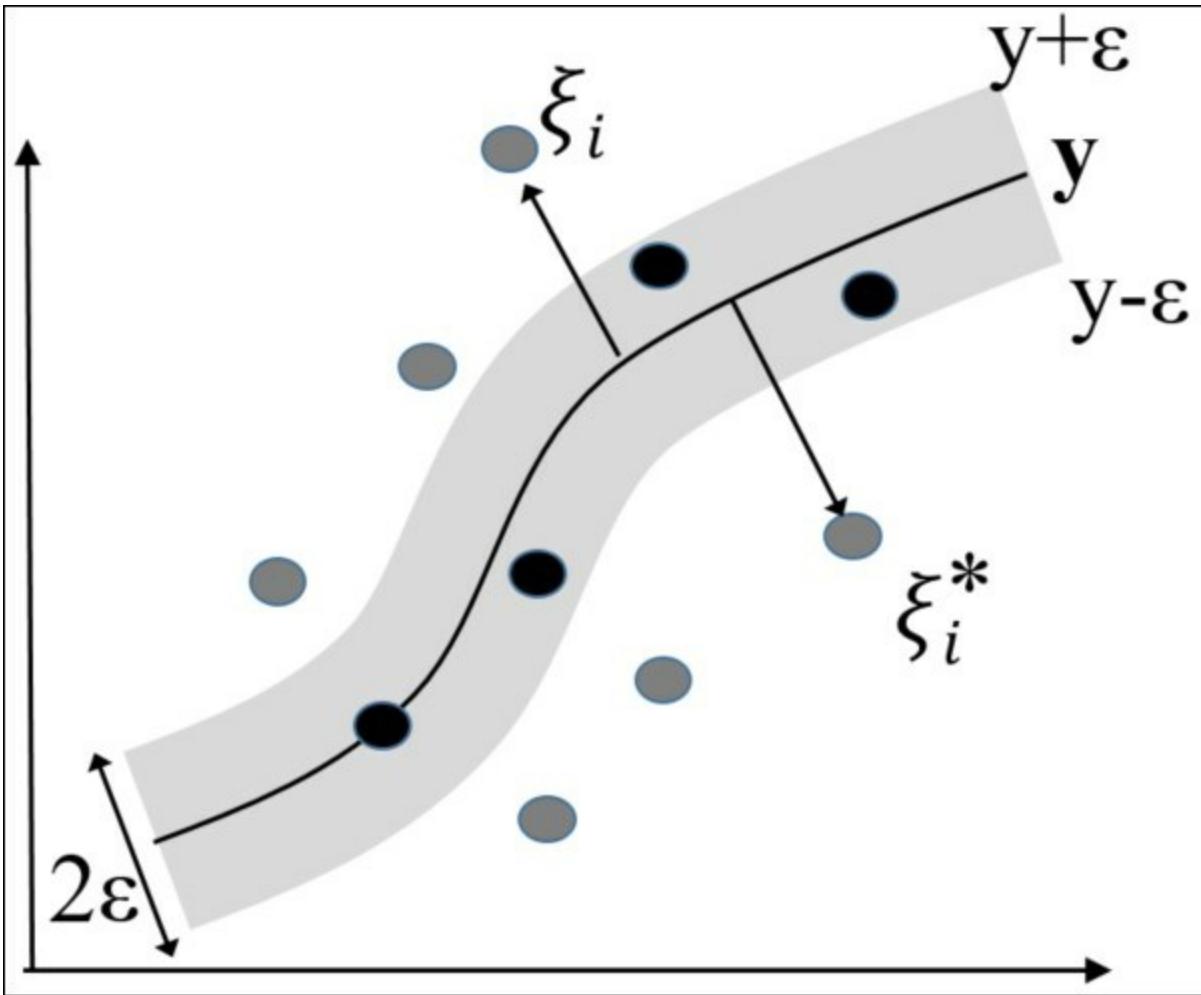
- e-SVR (sometimes called C-SVR)
- ?-SVR

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, e . 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 [12:14].

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 $2e$ (e is known as the insensitive error). The insensitive error plays a similar role to the margin in the SVC:



Visualization of the support vector regression and insensitive error

For the mathematically inclined, the maximization of the margin for nonlinear models introduces a pair of slack variables. As you may remember, the C-support vector classifiers use a single slack variable. The preceding diagram illustrates the minimization formula.

M9: e-SVR formulation:

$$\begin{aligned} \min_{w, \xi, \xi^*} & \left\{ \frac{w^T w}{2} + C \sum_{i=0}^{n-1} (\xi_i + \xi_i^*) \right\} \\ -\epsilon - \xi_i^* & \leq w^T \phi(x_i) + w_0 - y_i \leq \epsilon + \xi_i \quad \forall i \end{aligned}$$

Here, e is the insensitive error function.

M10: The e-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 regression* section of [Chapter 9, Regression and Regularization](#)).

SVR versus linear regression

This test consists of reusing the example that illustrates the presented in single-variate linear regression (refer to the *One-variate linear regression* section of [Chapter 9, 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:

- 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)
- Extract the labeled data (*SPY price*) from the data source (*DataSource*), which is the Yahoo financials CSV-formatted data file
- 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

- Create the observations as a time series of indexes, `xt`
- Instantiate the SVR with the index of trading session as features and the SPY adjusted closing price as the labeled output
- Run the prediction methods for both SVR and the linear regression and compare the results of the linear regression and SVR, `collect`:

```

val C = 12
val GAMMA = 0.3
val EPSILON = 2.5

val config = SVMConfig(
    new SVRFormulation(C, EPSILON), new RbfKernel(GAMMA) //45
)

for {
    src <- DataSource(path, false, true, 1)
    price <- src.get(close)
    (xt, y) <- getLabeledData(price.size) //46
    linRg <- SingleLinearRegression[Double](price, y) //47
} yield {
    val svr = SVM[Double](config, xt, price)
    collect(svr, linRg, price)
}

```

The formulation in the configuration has the type `SVRFormulation` (line 45). The `DataSource` class extracts the price of the SPY ETF. The method `getLabeledData` generates the input features `xt` and the labels (or expected values) `y` (line 46):

```

type LabeledData = (Vector[Array[Double]], DblVec)

def getLabeledData(numObs: Int): Try[LabeledData] = Try {
    val y = Vector.tabulate(numObs)(_.toDouble)
    val xt = Vector.tabulate(numObs)(Array[Double](_))
    (xt, y)
}

```

The single variate linear regression, `SingleLinearRegression`, is instantiated using the input `price` and labels `y` as input (line 47).

Finally, the method `collect` executes the two regression partial functions; `pfSvr` and `pfLinr`:

```

def collect(
    svr: SVM[Double],
    linr: SingleLinearRegression[Double],
    price: DblVec): List[Vector[DblPair]] = {

  val pfSvr = svr |>
    val pfLinr = linr |>
    (0 until price.size-80).foreach( n =>
      for {
        val z = Array[Double](n.toDouble)
        if( pfSvr.isDefinedAt(z)); x <- pfSvr(z)
        if( pfLinr.isDefinedAt(n)); y <- pfLinr(n)
        } yield { ... }
    )
  ...
}

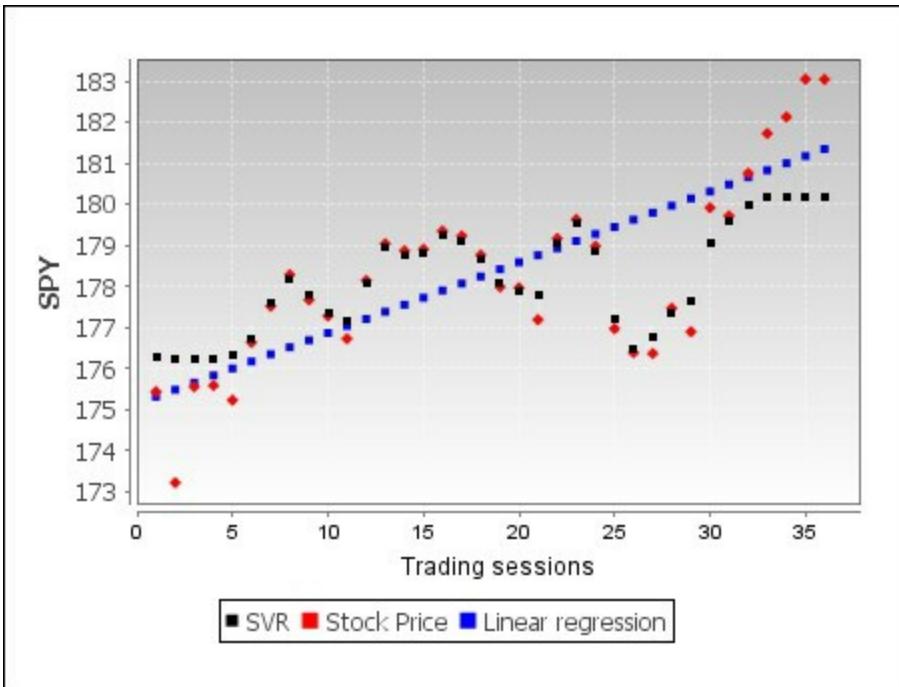
```

Note

isDefinedAt:

It is a good practice to check if a partial function is defined for a specific value of the argument. This preemptive approach allows the developer to select an alternative method or a full function. It is an efficient alternative to catching a `MathErr` exception.

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 of linear regression and SVR

The support vector regression provides a more accurate prediction than the linear regression model. You can also observe that the L₂ 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 L₂-norm penalty factor as $\gamma = 1/C$.

Tip

SVR and L₂ regularization:

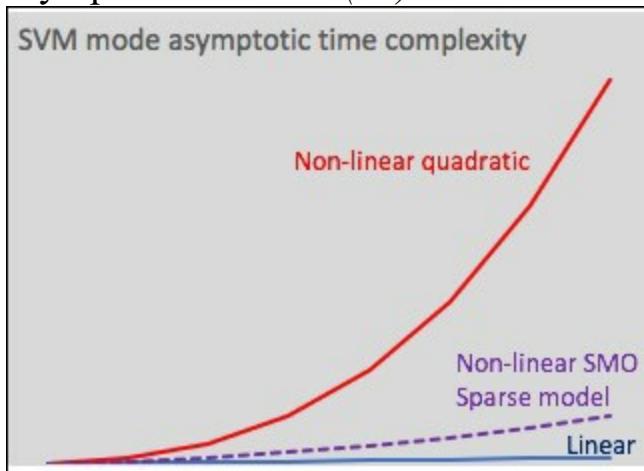
You are invited to run the use case with a different value of C to quantify the impact of the L₂ regularization on the predictive values of the SVR.

There is no need to compare SVR with logistic regression, as logistic regression is a classifier. However, SVM is related to logistic regression; the hinge loss in SVM is similar to the loss in the logistic regression [12:15].

Performance considerations

As with most discriminative models, the performance of the support vector machine obviously depends on the optimizer selected to maximize the margin during training. Let's look at the time complexity for different configuration and applications of SVM:

- A linear model (SVM without kernel) has an asymptotic time complexity $O(N)$ for training N labeled observations
- Nonlinear models with quadratic kernel methods (formulated as a quadratic programming problem) have 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 $O(N^2)$:



Graph asymptotic time complexity for various SVM implementations

The time and space complexity of the kernelized support vector machine has been the source of recent research [12:16] [12:17].

Summary

Maximizing margin classifiers, such as SVM, are a robust alternative to logistic regression for non-linear models for which appropriate kernel functions exists. Moreover, SVM is less demanding of computation resources for very large datasets.

In a nutshell, this chapter introduces you to the basic concept of kernel functions and the theory and application of SVM classifiers as applied to financial instruments. The chapter concludes with the one-class SVM classification for detecting outliers and an overview of the support vector regression models.

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.

This chapter concludes our overview of discriminative, supervised machine learning models. The next couple of chapters deal with a new universe: evolutionary models and reinforcement learning.

Chapter 13. Evolutionary Computing

There's a lot more to evolutionary computing than genetic algorithms. The first foray into evolutionary computing was motivated by the need to address different types of large combinatorial problems also known as **NP problems**. This field of research was pioneered by **John Holland** [10:1] and **David Goldberg** [10:2] to leverage the theory of evolution and biology to solve combinatorial problems. 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 purpose and foundation of genetic algorithms as well as their benefits and limitations

From a practical perspective, you will learn how to:

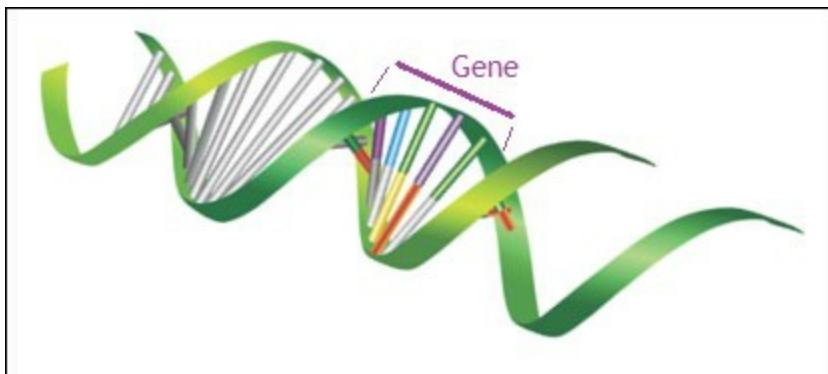
- Apply genetic algorithms to leverage a technical analysis of market price and volume movement to predict future returns
- Evaluate or estimate the search space
- Encode solutions in the binary format using either hierarchical or flat addressing
- Tune some genetic operators
- Create and evaluate fitness 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 fish, aerodynamic for birds, or stealth skills for predators. The following diagram shows a gene:



Visualization of a DNA strand

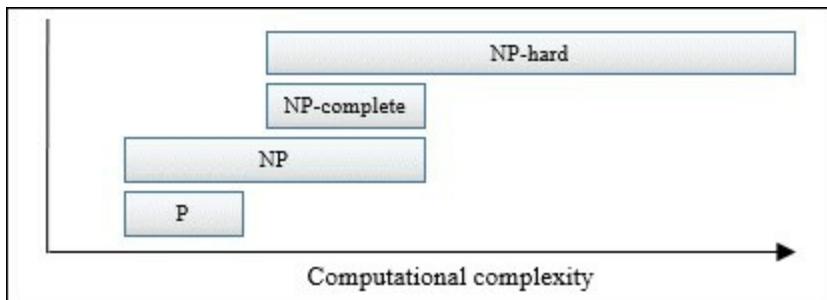
The (genetic) **population** of organisms varies over time. The number of individuals within a population changes, sometimes dramatically. These variations are usually associated with the abundance or lack of predators and prey as well as changing environments. Only the fittest organisms within the population are able to survive over time by adapting effectively to sudden changes in living environments and adjusting to unforeseen constraints.

NP problems

Nondeterministic Polynomial (NP) problems relate to the theory of computation and, more precisely, time and space complexity. 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 problem 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.

The computational complexity of the different types of NP problems is illustrated as follows:



Categorization of NP problems using computational complexity

Problems such as the traveling salesman, floor shop scheduling, the computation of a graph K-minimum spanning tree, map coloring, or cyclic ordering have a search execution time that is a nondeterministic polynomial, ranging from $n!$ to 2^n for a population of n elements [10:4].

NP problems cannot always be solved using analytical methods because of the computation overhead—even in the case of a model, it relies on differentiable functions. Genetic algorithms were invented by *John Holland* in the 1970s, and they derived their properties from Darwin's Theory of Evolution 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 a 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 genes from parents are passed on to their offspring. The success of an organism in its life measures its fitness [10:5].

Genetic algorithms use reproduction to evolve a population of possible solutions to a problem.

Genetic algorithms and machine learning

The practical purpose of a genetic algorithm as an optimization technique is to solve problems by finding the most relevant or fittest solution among a set or group of solutions. Genetic algorithms have many applications in machine learning, as follows:

- **Discrete model parameters:** Genetic algorithms are particularly effective in finding the set of discrete parameters that maximizes?. For example, the colorization of a black and white movie relies on a large but finite set of transformations from shades of grey to the RGB color scheme. The search space is composed of the different transformations and the objective function is the quality of the colorized version of the movie.
- **Reinforcement learning:** Systems that select the most appropriate rules or policies to match a given dataset rely on genetic algorithms to evolve a 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 15, 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 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 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 cases where the variable is continuous, the conversion is known as **quantization** or **discretization**. Each type of variable has a unique encoding scheme, as follows:

- Boolean values are easily encoded with 1 bit: 0 for false and 1 for true.
- Continuous variables are quantized or discretized like 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:

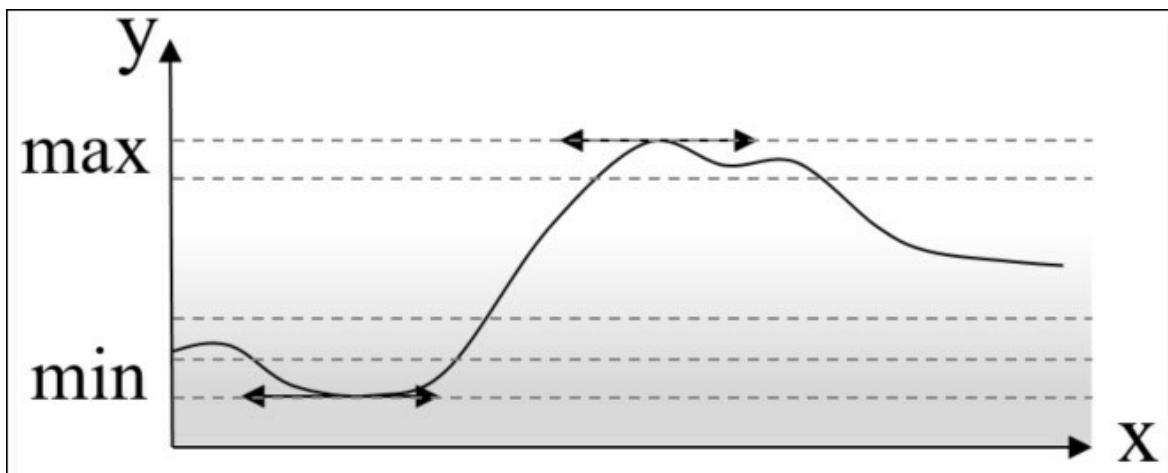


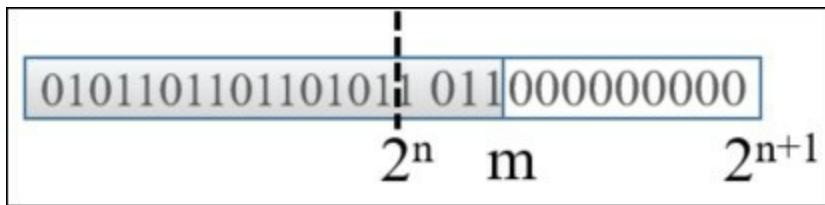
Illustration of quantization of continuous variable $y = f(x)$

The step size of the discretization is computed as (M1):

$$step = \frac{max - min}{2^n}$$

The step size of the quantization 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 must be accounted for. However, there is no guarantee that the number of variables will coincide with the bit boundary:



Padding for base 2 representation of values

In this case, the next exponent, $n+1$, defines the minimum number of bits required to represent the set of values: $n = \log_2(m).toInt + 1$. A discrete variable with 19 values requires five 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 an art as it is a science. For each encoding function, you need a decoding function to convert the binary representation back to actual values.

Predicate encoding

A predicate for a variable x is a relation defined as ' x operator [target]', for instance, 'unit cost < [\$9]', 'temperature = [82F]', or 'movie rating is [3 stars]'.

The simplest encoding scheme for predicates is as follows:

- Variables are encoded as a category or type (for example, temperature, barometric pressure, and so on) because there is a finite number of variables in any model
- Operators are encoded as discrete type
- Values are encoded as either discrete or continuous values

Note

Encoding format for predicates:

There are many approaches for encoding a predicate in a binary string. For instance, the format $\{operator, left-operand, right-operand\}$ 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

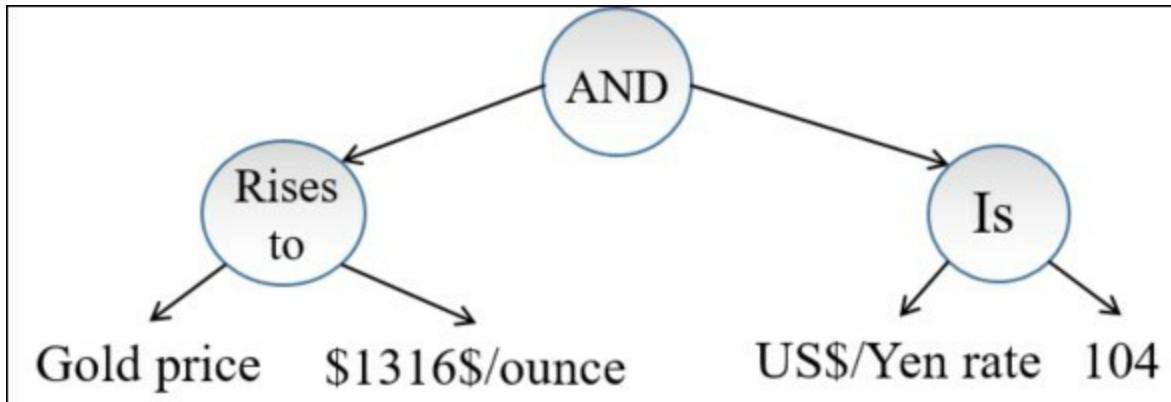
The solution encoding approach describes the solution to a problem as an unordered sequence of predicates. Let's consider the following rule:

```
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:

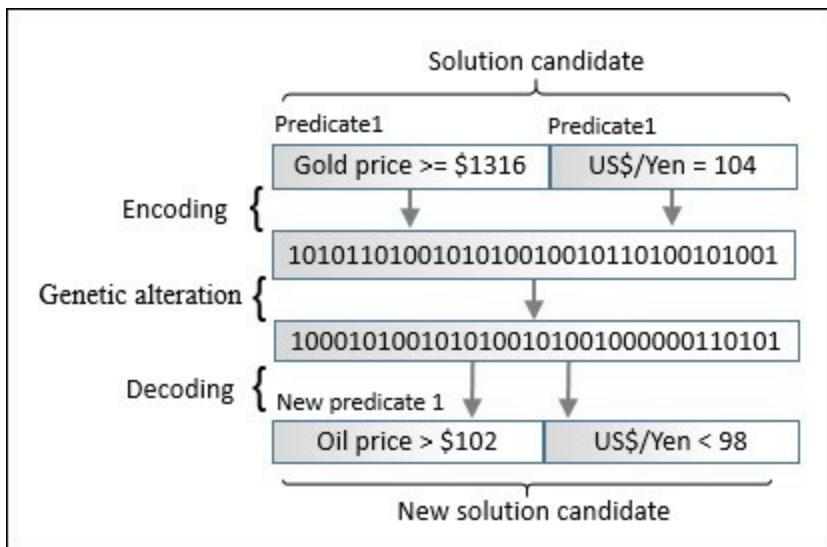
- Boolean operators (for example, *AND*) and predicates
- Each predicate is defined as a tuple $\{variable, operator, target value\}$

The tree representation for the search space is shown in the following diagram:



Graph representation of encoded rules

The binary string representation is decoded back to its original format for further computation:



Encoding, alteration, and decoding predicates

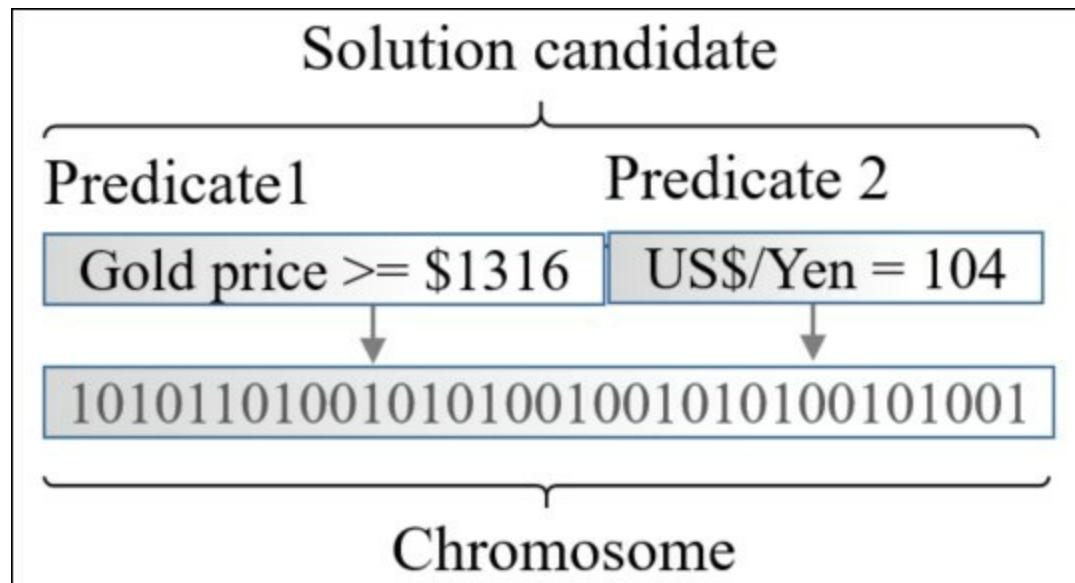
The encoding scheme

There are two approaches to encoding such a candidate solution or a chain of predicates:

- Flat encoding of a chromosome
- Hierarchical coding of a chromosome as a composition of genes

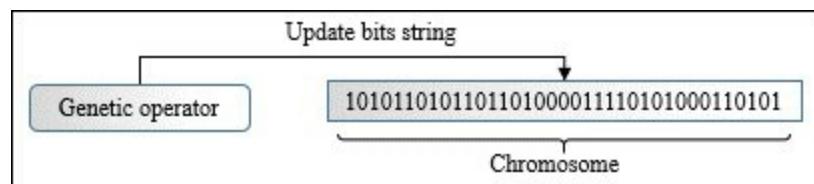
Flat encoding

The flat encoding approach consists of encoding the set of predicates into a single chromosome (bits string) representing a specific solution candidate to the optimization problem. The identity of the predicates is not preserved:



Flat addressing schema for chromosomes

A genetic operator manipulates the bits of the chromosome regardless of whether the bits refer to a specific predicate:

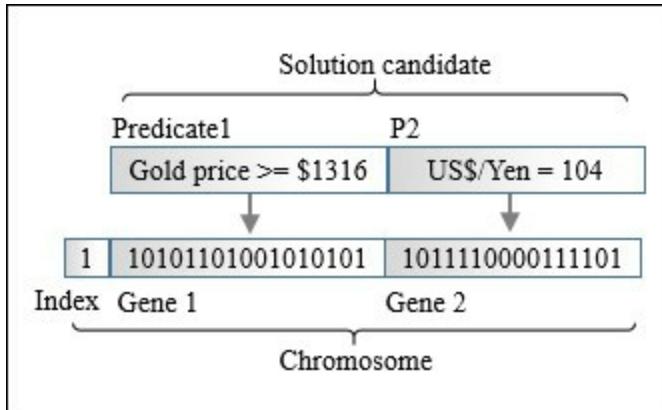


Chromosome encoding with flat addressing

Hierarchical encoding

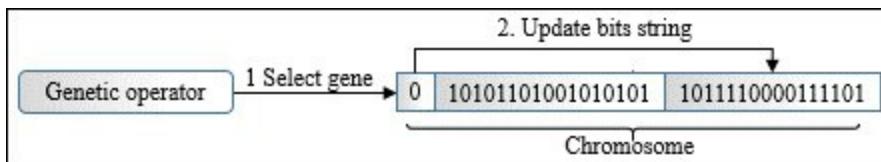
In this configuration, the characteristic of each predicate is preserved during the encoding process. Each predicate is converted into a gene represented by a bit string. The genes are aggregated to form the chromosome. An extra field is added to the bit string or chromosome for the selection of the gene. This

extra field consists of the index or the address of the gene:



Hierarchical addressing schema for chromosomes

A generic operator selects the predicate it needs to manipulate first. Once the target gene is selected, the operator updates the binary string associated to the gene, as follows:



Chromosome encoding with flat addressing

The next step is to define the genetic operators that manipulate or update the binary string representing either a chromosome or individual genes.

Genetic operators

The implementation of the reproduction cycle attempts to replicate the natural reproduction process [10:7]. The reproduction cycle that controls the population of chromosomes consists of three genetic operators:

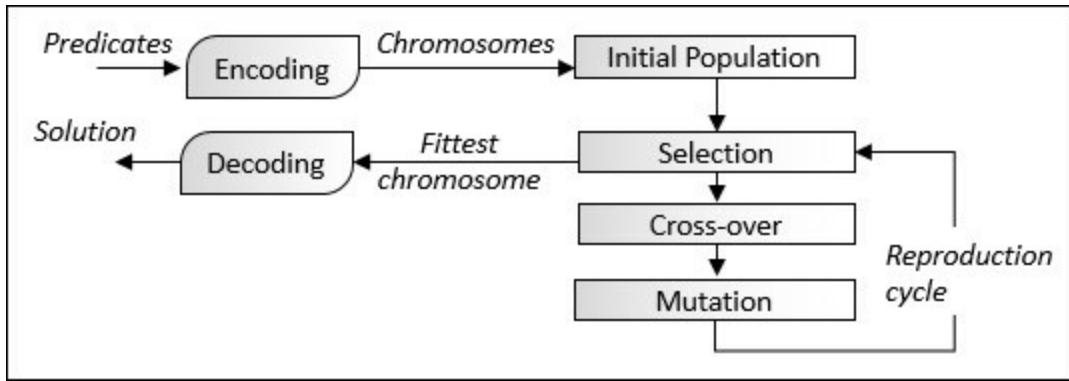
- **Selection:** This operator ranks chromosomes according to a fitness function or criteria. It eliminates the weakest or less-fit chromosomes and controls the population growth.
- **Crossover:** This operator pairs chromosomes to generate offspring chromosomes. These offspring chromosomes are added to the population along with their parent chromosomes.
- **Mutation:** This operator introduces a minor alteration in the genetic code (binary 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.

Tip

Transposition operator:

Some implementations of genetic algorithms use a fourth operator, genetic transposition, in cases where 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:



Basic workflow for the execution of genetic algorithms

The reproduction cycle can be implemented as a recursion or iteratively. The encoding and decoding operations are defined as a pair of transform and inverse transform from a Scala object to a bit string and vice-versa.

Tip

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 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].

Note

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 converging. However, for some problems, you may need to scale or magnify the relative fitness to preserve a meaningful difference in the fitness score between the chromosomes [10:9].

The following implementation relies on rank-based selection. The selection process consists of the following steps:

1. Apply the fitness function to each chromosome j in the population, f_j .
2. Compute the total fitness score for the entire population, $\sum f_j$.
3. Normalize the fitness score of each chromosome by the sum of the fitness scores of all the chromosomes, $f'_j = f_j / \sum f_j$.
4. Sort the chromosomes by their descending fitness score, $f'_j < f'_{j-1}$.
5. Compute the cumulative fitness score for each chromosome, $C_j = f'_j + \sum f'_k$.
6. Generate the selection probability (for the rank-based formula) as a random value, $p \in [0, 1]$.
7. Eliminate the chromosome, k , with a low unfitness score $f_k < p$ or high fitness cost, $f_k > 1 - p$.
8. Reduce the size of the population if it exceeds the maximum allowed number of chromosomes.

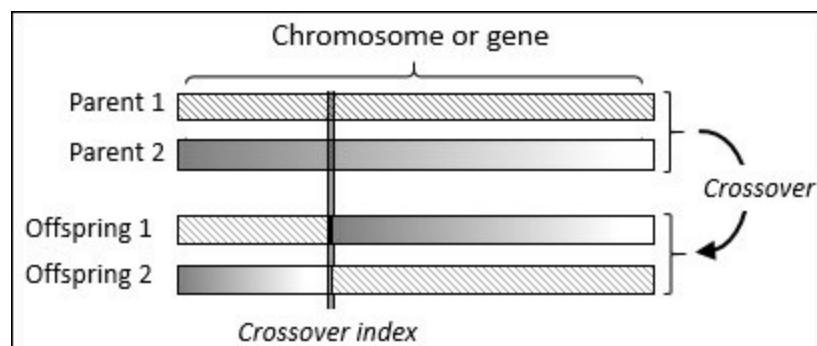
Note

Natural selection

You should not be surprised by the need to control the size of the population of chromosomes. After all, nature does not allow any species to grow beyond a certain point 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 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 technique. 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 operation

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 is 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 $xi = p \cdot gene_length$, 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.

Tip

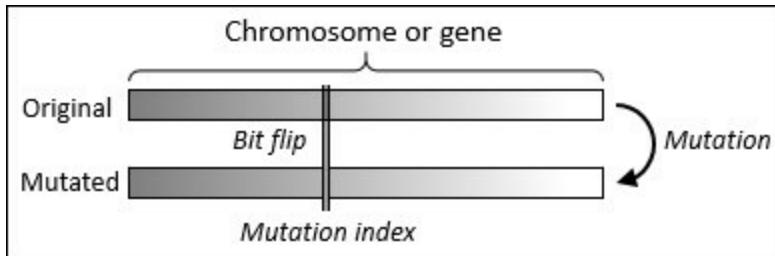
Preserving parent chromosomes

You may wonder why the parents are not removed from the population once the offspring chromosomes are created. This is because there is no guarantee that any of the offspring chromosomes are a better fit.

Mutation

The objective of genetic mutation is to prevent the reproduction cycle from converging 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 binary string representation of the chromosome, as illustrated in the following diagram:



A chromosome's mutation operation

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 $mi = p.\text{num_genes}$.
4. Compute the index of the bit in the gene to be mutated
 $xi = p.\text{genes_length}$.
5. Perform a flip XOR operation on the selected bit.

Note

Tuning considerations

Tuning 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 function:

- **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.

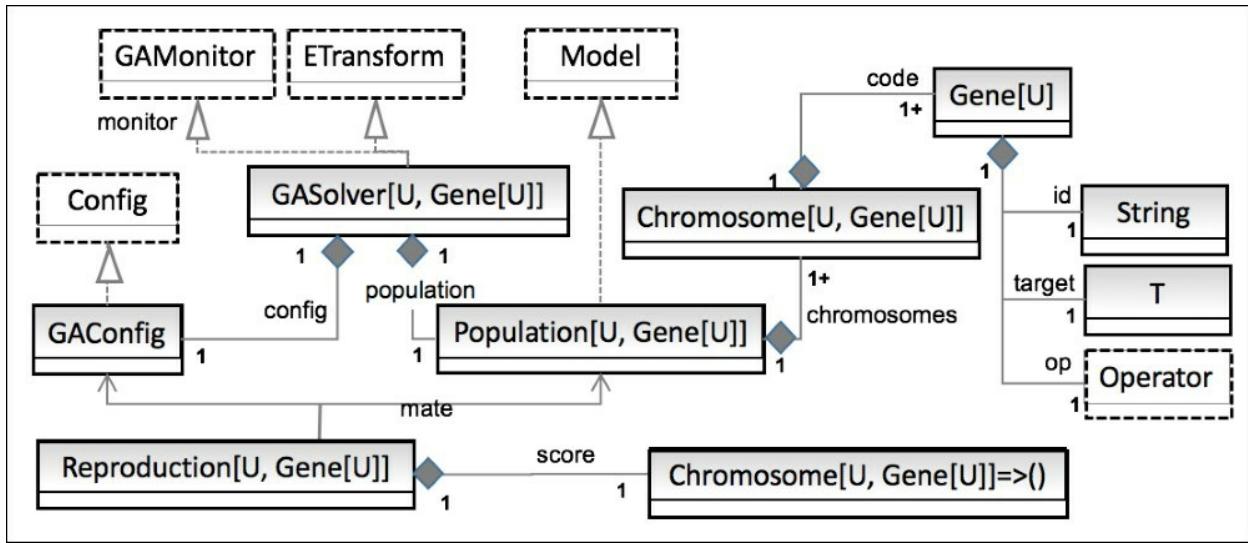
Software design

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 of the *Appendix*).

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 type `GAConfig` and the initial population. This class implements an explicit monadic data transformation of type `ETransform`.
- The configuration class `GAConfig` consists of the GA execution and reproduction configuration parameters.
- The reproduction (of type `Reproduction`) controls the reproduction cycle between consecutive generations of chromosomes through the `mate` method.
- The monitoring trait `GAMonitor` tracks the progress of the optimization and evaluates the exit condition for each reproduction cycle.

The following UML class diagram describes the relationship 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 elements 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.

Tip

The case for mutability

It is a good Scala programming practice to stay away from mutable collections. However, in this case, the number of chromosomes can be very large. Most implementations of genetic algorithms update the population potentially three times per reproduction cycle, generating many objects and taxing the Java garbage collector.

Population

The `Population` class takes two arguments:

- `limit`: This is the maximum size of the population
- `chromosomes`: This is the pool of chromosomes 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 (line 1), `+-` for the crossover of all the chromosomes (line 2), and `^` for the mutation of each chromosome (line 3). Consider the following code:

```
type Pool[U, T <: Gene[U]] = ArrayBuffer[Chromosome[U, T]]
```

```
class Population[U: ToDouble, T <: Gene[U]] {
```

```

    limit: Int, val chromosomes: Pool[U, T]
) {
  def select(score: Chromosome[U, T] => (), cutoff: Double) //1
  def +- (xOver: Double) //2
  def ^ (mu: Double) //3
  ...
}

```

The `limit` value specifies the maximum size of the population during optimization. It defines the hard limit or constraints on the population growth.

Chromosomes

The chromosome is the second level of containment in the genotype hierarchy. The `Chromosome` class takes a list of genes as a parameter (`code`). The signature of the crossover and mutation methods, `+-` and `^`, is 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[U: ToDouble, T <: Gene[U]] (val code: List[T]) {
  var cost: Double = nextDouble //4
  def +- (that: Chromosome[T], idx: GeneticIndices) :
    (Chromosome[T], Chromosome[T])
  def ^ (idx: GeneticIndices): Chromosome[T]
  ...
}

```

The algorithm assigns the (un)fitting score or a `cost` value to each chromosome to enable the ranking of chromosomes in the population and ultimately the selection of the fittest chromosomes (line 4).

Note

Fitness versus cost

The machine learning algorithms used the loss function or its variant as an objective function to be minimized. This implementation of the GA uses `cost`

scores to be consistent with the concept of minimization of the, loss, or penalty function.

Genes

Finally, the reproduction process executes the genetic operators on each gene:

```
class Gene[U: ToDouble] (
    val id: String,
    val target: Double,
    op: Operator)
    (implicit quant: Quantization[U], encoding: Encoding) { //5

    lazy val bits: BitSet = apply(target, op)
    def apply(value: Double, op: Operator): BitSet //6
    def unapply(bitSet: BitSet): (Double, Operator) //7

    def +- (index: Int, that: Gene): Gene //8
    def ^ (index: Int): Unit //9
    ...
}
```

The `Gene` class takes three arguments and two implicit parameters:

- `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.
- `quant`: This is the quantization or discretization class that converts a double value to an integer to be converted into bits and vice versa (line 5).
- `encoding`: This is the encoding or bits layout of the gene as a value and operator pair (globally).

The `apply` method encodes a pair of value and operator into a bit set (line 6). The `unapply` method is the reverse operation of `apply`: it decodes a bit set into a pair of value and operator (line 7).

Note

unapply()

The method `unapply` reverses the state transition performed by the `apply` method. For example, if the `apply` method populates a collection, the `unapply` method clears the collection from its elements.

The implementation of the cross-over (line 8) and mutation (line 9) operators on a gene is similar to the same operations on the container chromosome.

The quantization is implemented as a case class:

```
case class Quantization[U: ToDouble] (toInt: U=>Int, toU: Int=>U)
```

The first function, `toInt`, converts a real value to an integer and `toU` converts the integer back to a real value of type `U`. 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 type `java.util.BitSet`) using the quantization function `Quantization.toInt`.

The layout of a gene is defined by the `Encoding` class as follows:

```
class Encoding(nValueBits: Int, nOpBits: Int) {  
    val rValue = Range(0, nValueBits)  
    val length = nValueBits + nOpBits  
    val rOp = Range(nValueBits, length)  
}
```

The `Encoding` class specifies the bits layout of the gene as the number of bits, `nValueBits`, to encode the value and the number of bits, and `nOpBits` to encode the operator. The class defines the range `rValue` for the value and the range `rOp` for the operator. The client code has to supply the implicit instance of the `Encoding` class.

The bit set, `bits`, of the gene is encoded through the `apply` method:

```
def apply(value: Double, op: Operator): BitSet = {  
    val bitset = new BitSet(encoding.length)
```

```

encoding.rOp foreach(i => //10
  if(((op.id>>i) & 0x01) == 0x01) bitset.set(i)
)

encoding.rValue foreach(i => //11
  if(((quant.toInt(value)>>i) & 0x01) == 0x01) bitset.set(i)
)
bitset
}

```

The bits layout of the gene is created using `java.util.BitSet`. The operator, `op` is encoded first through its identifier `id` (line 10). The `value` is quantized by invoking the method `toInt` then encoded into a bit set (line 11).

The `unapply` method decodes the gene from a bit set or bit string to a pair of value and operator. The method uses the quantization instance to cover bits into value and an auxiliary function, `convert`, which is described along with its implementation in the source code accompanying this book (line 12):

```

def unapply(bits: BitSet): (Double, Operator) =
  (quant.toU(convert(encoding.rValue, bits)),
   op(convert(encoding.rOp, bits))) //12

```

The `Operator` trait defines the signature of any operator. Each domain-specific problem requires a unique set of operations: Boolean, numeric, or string manipulation:

```

trait Operator {
  def id: Int
  def apply(id: Int): Operator
}

```

The preceding operator has two methods: an identifier `id` and an `apply` method that converts an index to an operator.

Selection

The first genetic operator of the reproduction cycle is the selection process. The `select` method of the `Population` class implements the steps of the selection phase on the population of chromosomes in the most efficient manner, as follows:

```
def select(
    score: Chromosome[U, T] => Unit,
    cutOff: Double
): Unit = {
    val cumul = chromosomes.map(_.cost).sum/ScalingFactor //13
    chromosomes foreach(_ /= cumul) //14

    val _chromosomes = chromosomes.sortBy(_.cost)//15
    val cutOffSize = (cutOff*_chromosomes.size).floor.toInt //16
    val popSize = if(limit < cutOffSize) limit else cutOffSize

    chromosomes.clear //17
    chromosomes += _chromosomes.take(popSize) //18
}
```

The `select` method computes the cumulative sum, `cumul`, of the `cost` (line 13) for the entire population. It normalizes the cost of each chromosome (line 14), orders the population by decreasing value (line 15), and applies a soft limit function, `cutOff`, on population growth (line 16). The next step reduces the size of the population to the lowest of the two limits: the hard limit, `limit`, or the soft limit, `cutOffSize`. Finally, the existing chromosomes are cleared (line 17) and updated with the next generation (line 18).

Tip

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 a parameter and returns the `cost` 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 genetic algorithms are:

- `xOver`: This is the crossover ratio (or probability) and has a value in the interval $[0, 1]$
- `mu`: This is the mutation ratio
- `maxCycles`: This is the maximum number of reproduction cycles
- `softLimit`: This is the soft constraint on the population growth

Consider the following code:

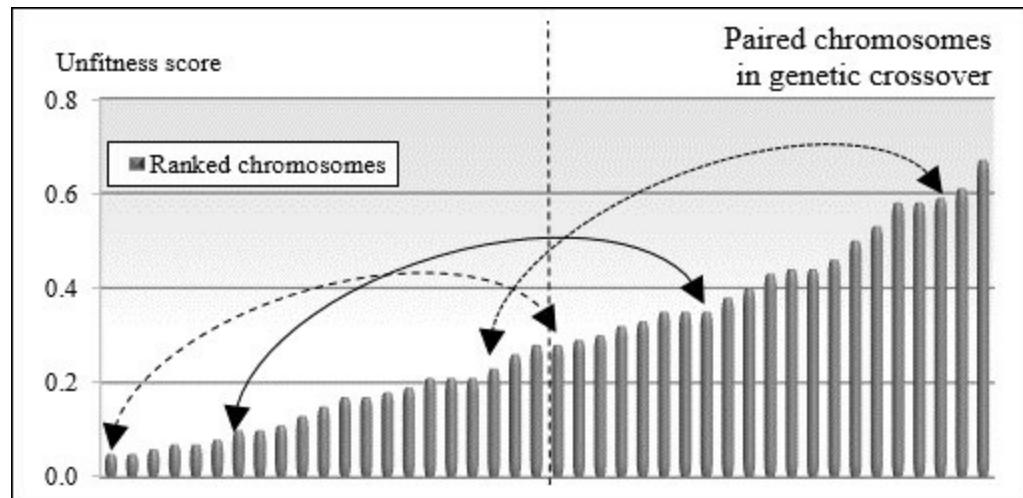
```
class GAConfig(  
    val xover: Double,  
    val mu: Double,  
    val maxCycles: Int,  
    val softLimit: Int => Double) extends Config {  
    val mutation = (cycle : Int) => softLimit(cycle)  
}
```

Crossover

As mentioned earlier, the genetic crossover operator couples two chromosomes to generate two offspring chromosomes that compete with all the other chromosomes in the population, including their own parents, in the selection phase of the next reproduction cycle.

Population

We use the notation `+-` as the implementation of the crossover operator in Scala. There are several options to select pairs of chromosomes for crossover. This implementation ranks the chromosomes by their *fitness* (or inverse `cost`) value and then divides the population into two halves. Finally, it pairs chromosomes of identical rank from each half as illustrated in the following diagram:



Pairing of chromosomes within a population prior to crossover

The crossover implementation, `+-`, selects the parent chromosome candidates for crossover using the pairing scheme described earlier. Consider the following code:

```
def +- (xOver: Double): Unit = if( size > 1) {  
    val mid = size>>1
```

```

    val bottom = chromosomes.slice(mid, size) //19
    val gIdx = geneticIndices(xOver) //20

    val offSprings = chromosomes.take(mid) .zip(bottom)
        .map{ case (t, b) => t +- (b, gIdx) }
        .unzip //21
    chromosomes +== offSprings._1 ++ offSprings._2 //22
}

```

This method splits the population into two subpopulations of equal size (line 19) and applies the Scala `zip` and `unzip` methods to generate the set of pairs of offspring chromosomes (line 20). The crossover operator, `+-`, is applied to each chromosome pair to produce an array of pairs of `offsprings` (line 21). Finally, the crossover method adds offspring chromosomes to the existing population (line 22). The crossover value, `xOver`, is a probability randomly generated over the interval `[config.xOver, 1]`.

The `GeneticIndices` case class defines two indices of the bit whenever a crossover or a mutation occurs. The first index, `chOpIdx`, is the absolute index of the bit affected by the genetic operation in the chromosome (line 23). The second index, `geneOpIdx`, is the index of the bit within the gene subjected to crossover or mutation (line 24):

```

case class GeneticIndices(
    chOpIdx: Int, //23
    geneOpIdx: Int //24
)

```

The `geneticIndices` method computes the relative indices of the crossover bit in the chromosomes and genes:

```

def geneticIndices(prob: Double): GeneticIndices = {
    var idx = (prob*chromosomeSize).floor.toInt //25
    val chIdx = if(idx == chromosomeSize) chromosomeSize-1
                else idx //25

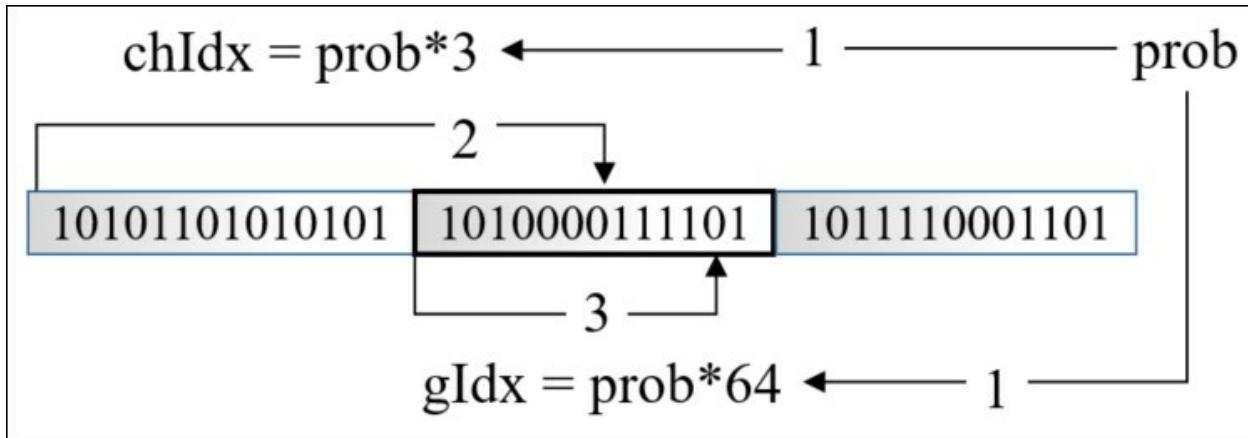
    idx = (prob*geneSize).floor.toInt
    val gIdx = if(idx == geneSize) geneSize-1 else idx //26
    GeneticIndices(chIdx, gIdx)
}

```

The first indexer `chIdx` is the index or rank of the gene within the

chromosome, to be affected by the genetic operator (line 25). The second indexer `gIdx` is the relative index of the bit within the gene (line 26).

Let's consider a chromosome composed of 2 genes with 63-bits/elements each, as illustrated in the following diagram:



The `geneticIndices` method computes:

- The index, `chIdx` of the gene within the chromosome and the index `gIdx` of the bit within the gene
- The genetic operator selects the gene of index `chIdx` (that is the second gene) to be altered
- The genetic operator altered the chromosome at the bit of index `gIdx` (that is $chIdx * 64 + gIdx$)

Chromosomes

We already defined the class `Chromosome` in the previous section. Let's look at the declaration and implementation of the genetic operators that apply to chromosomes:

```
final val CostFactor = 500

var cost: Double = CostFactor *(1.0 + nextDouble) //27

def +- (that: Chromosome[U,T], indices: GeneticIndices): //28
       (Chromosome[U,T], Chromosome[U,T])
def ^ (indices: GeneticIndices): Chromosome[U,T] //29
```

```

def /= (normalizeFactor: Double): Unit =    //30
    cost /= normalizeFactor
def decode(implicit d: Gene[U]=>T): List[T] = code.map(d(_)) //31

```

The cost (or unfitness) of a chromosome is initialized as a random value between QUANT and 2*QUANT (line 27). The genetic cross-over operator `+-` generates a pair of two offspring chromosomes (line 28). The genetic mutation operator `^` creates a slightly modified (1 or 2 bits) clone of this chromosome (line 29). The method `/=` normalizes the cost of the chromosome (line 30). The `decode` method converts the gene to a logic predicate or rule using an implicit conversion `d` between a gene and its subclass (line 31).

Tip

Cost initialization

There is no absolute rule to initialize the cost of the chromosomes from an initial population. However, it is recommended to differentiate chromosomes by using non-zero random values with a large range as their cost.

Implementing crossover for a pair of chromosomes using hierarchical encoding follows two steps:

- Find the gene on each chromosome that corresponds to the crossover index, `indices.chOpIdx`, and then swap the remaining genes
- Split and splice the gene crossover at `xoverIdx`

Consider the following code:

```

def +- (that: Chromosome[U, T], indices: GeneticIndices):
    (Chromosome[U,T], Chromosome[U,T]) = {
        val xoverIdx = indices.chOpIdx //32
        val xGenes = spliceGene(indices, that.code(xoverIdx)) //33

        val offSprng1 = code.slice(0, xoverIdx) :::
            xGenes._1 :: that.code.drop(xoverIdx+1) //34
        val offSprng2 = that.code.slice(0, xoverIdx) :::
            xGenes._2 :: code.drop(xoverIdx+1)
        (Chromosome[U,T](offSprng1), Chromosome[U,T](offSprng2)) //35
    }

```

```
}
```

The crossover method computes the index `xoverIdx` of the bit that defines the crossover in each parent chromosome (line 32). The genes `code(xoverIdx)` and `that.code(xoverIdx)` are swapped and spliced by the `spliceGene` method to generate a spliced gene (line 33):

```
def spliceGene(indices: GeneticIndices, thatCode: T): (T, T) = {
  ((this.code(indices.chOpIdx) +- (thatCode, indices)),
   (thatCode +- (code(indices.chOpIdx), indices)) )
}
```

The offspring chromosomes are gathered by collating the first `xOverIdx` genes of the parent chromosome, the crossover gene, and the remaining genes of the other parent (line 34). The method returns a pair of offspring chromosomes (line 35).

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[U], indices: GeneticIndices): Gene[U] = {
  val clonedBits = cloneBits(bits) //36

  (indices.geneOpIdx until bits.size).foreach(n =>
    if( that.bits.get(n) ) clonedBits.set(n)
    else clonedBits.clear(n) //37
  )
  val (target, operator) = decode(clonedBits) //38
  new Gene[U](id, target, operator)
}
```

The bits of the gene are cloned (line 36) and then spliced by exchanging their bits along the crossover point `indices.geneOpIdx` (line 37). The `cloneBits` function duplicates a bit string, which is then converted into a `(target value, operator)` tuple using the `decode` method (line 38). We omit these two methods because they are not critical to the understanding of the algorithm.

Mutation

The mutation of the population uses the same algorithmic approach as the crossover operation.

Population

The mutation operator `^` invokes the same operator for all the chromosomes in the population and then adds the mutated chromosomes to the existing population, so that they can compete with the original chromosomes. We use the notation `^` to define the mutation operator to remind the reader that the mutation is implemented by flipping one bit:

```
def ^ (prob: Double): Unit =  
    chromosomes ++= chromosomes.map(_ ^ geneticIndices(prob))
```

The mutation parameter `prob` is used to compute the absolute index of the mutating gene, `geneticIndices(prob)`.

Chromosomes

The implementation of the mutation operator `^` on a chromosome consists of mutating the gene of the index `indices.chOpIdx` (line 39) and then updating the list of genes in the chromosome (line 40). The method returns a new chromosome (line 41) that will compete with the original chromosome:

```
def ^ (indices: GeneticIndices): Chromosome[U, T] = { //39  
    val mutated = code(indices.chOpIdx) ^ indices  
    val xs = (0 until code.size).map(i =>  
        if(i== indices.chOpIdx) mutated  
        else code(i)  
    ).toList //40  
    Chromosome[U,T](xs) //41  
}
```

Genes

Finally, the mutation operator flips (XOR) the bit at the index `indices.geneOpIdx`:

```
def ^ (indices: GeneticIndices): Gene[U] = {
    val idx = indices.geneOpIdx
    val clonedBits = cloneBits(bits) //42

    clonedBits.flip(idx) //43
    val (target, operator) = decode(clonedBits) //44
    new Gene[U](id, target, operator) //45
}
```

The `^` method mutates the cloned bit string, `clonedBits` (line 42) by flipping the bit at the index `indices.geneOpIdx` (line 43). It decodes and converts the mutated bit string by converting it into a (target value, operator) tuple (line 44). The last step creates a new gene from the target-operator tuple (line 45).

Reproduction

Let's wrap the reproduction cycle into a `Reproduction` class that uses the scoring function `score`:

```
class Reproduction[U: ToDouble, T <: Gene[U]]  
  (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[U, T],  
        config: GAConfig,  
        cycle: Int): Boolean = (population.size: @switch) match {  
  case 0 | 1 | 2 => false //46  
  case _ => {  
    rand.setSeed(rand.nextInt + System.currentTimeMillis)  
  
    population.select(score, config.softLimit(cycle)) //47  
    population +- rand.nextDouble*config.xover //48  
    population ^ rand.nextDouble*config.mu //49  
    true  
  }  
}
```

The `mate` method returns `false` (that is, the reproduction cycle aborts) if the population size is less than 3 (line 46). The chromosomes in the current population are ranked by increasing cost. Chromosomes with a high cost or low fitness are discarded to comply with the soft limit, `softLimit`, on the population growth (line 47). The randomly generated probability is used as input to the cross over operation on the entire remaining population (line 48) and input to the mutation of the remaining population (line 49):

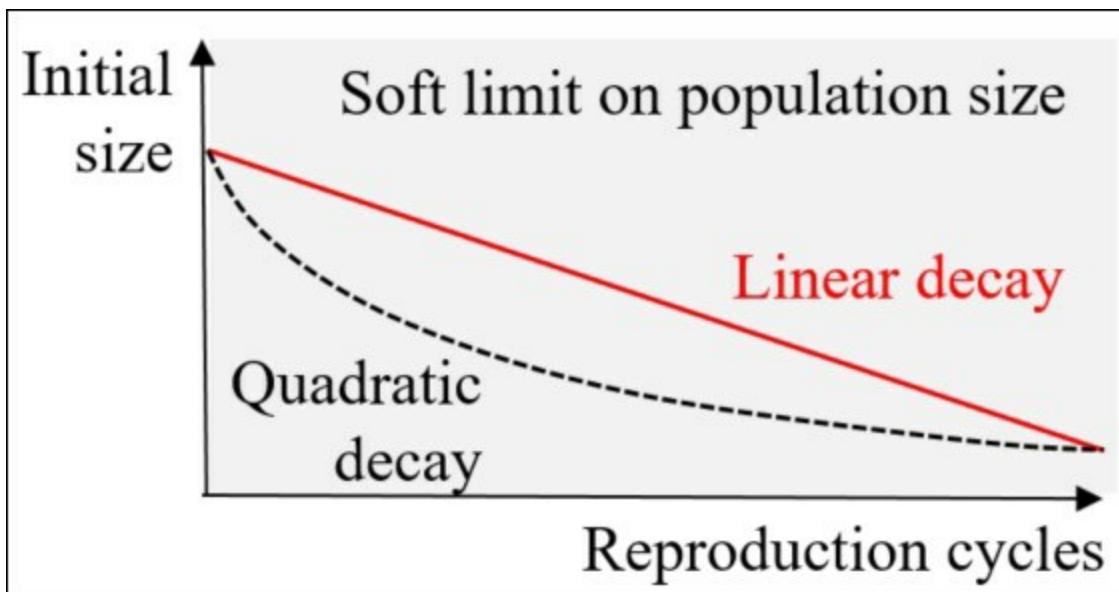


Illustration of the linear and quadratic soft limit for population growth

Solver

The `GASolver` class manages the reproduction cycles and the population of chromosomes. The solver is defined as a data transformation of type `ETransform` using an explicit configuration of type `GAConfig` as described in the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#) (line 50).

The `GASolver` class implements the `GAMonitor` trait to monitor population diversity, manage the reproduction cycle, and control the convergence of the optimizer (line 51).

The genetic algorithm-based solver has three arguments:

- `config`: This is the configuration of the execution of the genetic algorithm
- `score`: This is the scoring function of a chromosome
- `tracker`: This is the optional tracking function to initialize the monitoring function of `GAMonitor`:

```
class GASolver[U: ToDouble, T <: Gene[U]] (
    config: GAConfig,
    score: Chromosome[U, T] => Unit,
    tracker: Option[Population[U, T] => Unit] = None)
extends ETransform[Population[U, T], Population[U, T]] //50
    with GAMonitor[U, T] { //51

    type W = Population[U, T] //52
    val monitor: Option[W => Unit] = tracker //53

    def |>(initialize: => W): Try[W] = this.|>(initialize()) //5
    override def |>: PartialFunction[W, Try[W]] //55
}
```

This explicit data transformation must initialize the type `w` of the input element and output element (line 52) for the prediction or optimization method `|>`. The optional monitor function is used for debugging/tracing purposes (line 53). The optimizer takes an initial population as input and generates a very small population of the fittest chromosomes from which the

best solution is extracted (line 55).

The population is generated by the method `|> (=> Population[U, T])`, which takes the constructor of the `Population` class as argument (line 54). Let's briefly look at the monitoring trait, `GAMonitor`, assigned to the genetic algorithm. The trait has two attributes:

- `monitor`: This is an abstract value to be initialized by classes implementing this trait (line 55).
- `state`: This is the current state of the execution of the genetic algorithm. The initial state of the genetic algorithm is `GA_NOT_RUNNING` (line 56):

```
trait GAMonitor[U, T <: Gene] extends Monitor {
  self: {
    def |> :PartialFunction[Population[U, T],
                           Try[Population[U, T]]]
  } => //55
  val monitor: Option[Population[U, T] => Unit] //56
  var state: GASTate = GA_NOT_RUNNING //57

  def isReady: Boolean = state == GA_NOT_RUNNING
  def start: Unit = state = GA_RUNNING
  def isComplete(
    population: Population[U, T],
    remainingCycles: Int
  ): Boolean = { ... } //58
}
```

The `state` of the genetic algorithm can only be updated in the `|>` method through an instance of the `GAMonitor` class. (line 55)

Here is a subset of the possible state of the execution of the genetic algorithm:

```
sealed abstract class GASTate(desc: String)
case class GA_FAILED(desc: String) extends GASTate(desc)
object GA_RUNNING extends GASTate("Running")
```

The solver invokes the `isComplete` method to test the convergence of the optimizer at each reproduction cycle (line 58).

There are two options for estimating that the reproducing cycle is converging:

- **Greedy**: In this approach, the objective is to 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

Let's consider the following implementation of the genetic algorithm solver:

```

override def |> : PartialFunction[W, Try[W]] = {
  case population: U if(population.size > 1 && isReady) => {
    start //59
    val reproduction = Reproduction[U,T](score) //60

    @tailrec
    def reproduce(population: W, n:Int): W = { //61
      if( !reproduction.mate(population, config, n) ||
          isComplete(population, config.maxCycles -n) )
        population
      else
        reproduce(population, n+1)
    }

    reproduce(population, 0)
    population.select(score, 1.0) //62
    Try(population)
  }
}

```

The optimizing method initializes the state of execution (line 59) and the components of the `reproduction` cycle (line 60). The reproduction cycle (or epoch) is implemented as a tail recursion, which tests if the last reproduction cycle failed or if the optimization has converged towards a solution (line 61). Finally, the remaining, fittest, chromosomes are reordered by invoking the `select` method of the `Population` class (line 62).

GA for trading strategies

Let's apply our fresh expertise in genetic algorithms to evaluating different strategies to trade securities using trading signals. Knowledge of 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 the technical analysis of securities and financial markets, described briefly in the *Technical analysis* section of the *Appendix*.

The problem is to find the best trading strategy to predict the increase or decrease of the price of a security given a set of trading signals. A trading strategy is defined as a set of trading signals ts_j that are triggered or fired when a variable $x = \{x_j\}$, derived from financial metrics such as the price of the security or the daily or weekly trading volume, exceeds, equals, or is below a predefined target value, a_j (refer to the *Trading signals and strategy* section of the *Appendix*).

The number of variables that can be derived from the 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 dataset (minimize a cost or unfitness function)
- Tuning those trading signals with heuristics derived from personal experience and expertise

Note

Alternative to GA

The problem described earlier can certainly be solved using one of the machine learning algorithms introduced in the previous chapters. It is just a matter of defining a training set and formulating the problem as minimizing the loss function between the predictor and the training score.

The following table lists the trading classes with their counterpart in the genetic world:

Generic classes	Corresponding securities trading classes
Operator	SOperator
Gene	Signal
Chromosome	Strategy
Population	StrategiesFactory

Definition of trading strategies

The components of a trading strategy include financial operators, the objective, or cost function, the market signals, and the encoding technique.

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 a `SOperator` instance):

```
class SOperator(_id: Int) extends Operator {
    override def id: Int = _id
    override def apply(idx: Int): SOperator = SOPRATORS(idx)
}
```

The operators used by trading signals are the logical operators: `<` (`LESS_THAN`), `>` (`GREATER_THAN`), and `=` (`EQUAL`), as follows:

```
final object LESS_THAN extends SOperator(1)
final object GREATER_THAN extends SOperator(2)
final object EQUAL extends SOperator(3)
```

Each operator of type `SOperator` is associated with a scoring function by the `map operatorFuncMap`. The scoring function computes the cost (or unfitness) of the signal against a real value or a time series:

```
val operatorFuncMap = Map[Operator, (Double, Double) => Double] (
    LESS_THAN -> ((x: Double, target: Double) => target - x),
    ...
)
```

The `select` method of `Population` computes the `cost` value of a signal by quantifying the truthfulness of the predicate. For instance, the unfitness value for a trading signal, $x > 10$, is penalized as $5 - 10 = -5$ for $x = 5$ and credited as $14 - 10 = 4$ if $x = 14$. In this regard, the unfitness value is like the cost or

loss in a discriminative machine learning algorithm.

The cost function

Let's consider the following trading strategy defined as a set of two signals to predict the sudden relative decrease Δp of the price of a security:

- Relative volume v^m with a condition $v^m < \alpha$
- Relative volatility v^l with the condition $v^l > \beta$

Have a look at the following graphs:

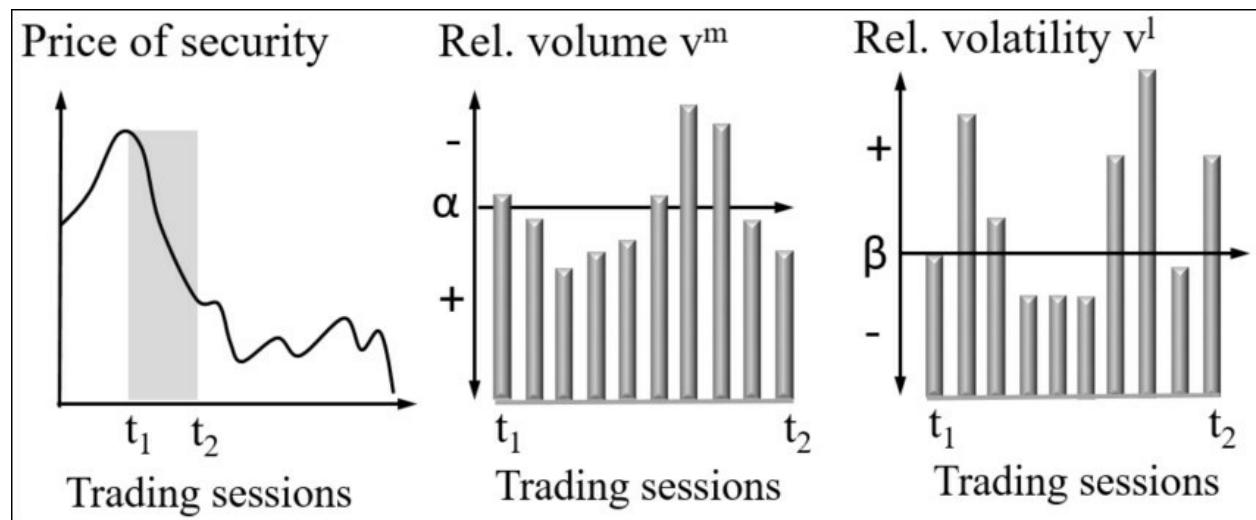


Chart of the price, relative volume, and relative volatility of a security

As the goal is to model a sudden crash in stock price, we should reward trading strategies that predict a steep decrease in the stock price and penalize 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 vm and relative volatility vl , n trading sessions, the cost or unfitness function C , and given a relative variation of stock price and a penalization $w = -\Delta p$ (M2):

$$w_t = -\Delta p_t$$

$$C(p, v^m, v^I | \alpha, \beta) = \sum_{t=0}^{n-1} (\alpha - v_t^m) w_t + (v_t^I - \beta) w_t$$

Market signals

Let's subclass the `Gene` class to define a market signal of type `Signal` as follows:

```
final class Signal(id: String,
  target: Double,
  op: Operator,
  xt: DblVec,
  weights: DblVec)
(implicit quant: Quantization[Double], encoding: Encoding)
  extends Gene[Double](id, target, op)
```

The `Signal` class requires the following arguments:

- An identifier `id` for the feature
- A `target` value
- An operator `op`
- A time series `xt` of type `DblVec`
- The optional `weights` associated to each data point of the time series `xt`
- An implicit `Quantization` instance, `quant`
- An implicit `encoding` scheme

The main purpose of the `Signal` class is to compute its `score` as a chromosome. The chromosome updates its `cost` by summing the score or weighted score of the signals it contains. The score of the trading signal is simply the summation of the penalty or truthfulness of the signal for each entry of the time series, `ts`:

```
override def score: Double =
  if (!operatorFuncMap.contains(op)) Double.MaxValue
  else
    operatorFuncMap.get(op).map(
      0 until xt.length).map(n =>
```

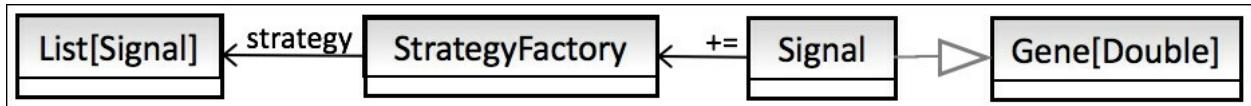
```

    weights(n) * _ (xt(n), target)).sum).getOrElse(-1.0)
)

```

Trading strategies

A trading strategy is an unordered list of trading signals. It makes sense to create a factory class to generate trading strategies. The `StrategyFactory` class creates strategies of type `List[Signal]` from an existing pool of signals of subtype `Gene`:



Factory pattern for trading signals

The `StrategyFactory` class has two arguments: the number of signals, `nSignals`, in a trading strategy and the implicit `Quantization` and `Encoding` instances (line 63):

```

class StrategyFactory(nSignals: Int) //63
  (implicit quant: Quantization[Double], encoding: Encoding) {

  val signals = ListBuffer[Signal]()
  lazy val strategies: Pool[Double, Signal] //64
  def +=(id: String, target: Double,
         op: SOperator, xt: DblVec, weights: DblVec)
  ...
}

```

The `+=` method takes five arguments: the identifier `id`; the target value; the 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. The `StrategyFactory` class generates all possible sequences of signals as trading strategies as lazy values to avoid unnecessary regeneration of the pool on demand (line 64) as follows:

```

lazy val strategies: Pool[Double, Signal] = {
  implicit val ordered = Signal.orderedSignals //70

  val XSS = new Pool[Double, Signal] //65

```

```

val treeSet = new TreeSet[Signal] ++= signals.toList //66
val subsetsIterator = treeSet.subsets(nSignals) //67

while( subsetsIterator.hasNext) {
    val signalList = subsetsIterator.next.toList //68
    XSS.append(Chromosome[Signal](signalList)) //69
}
XSS
}

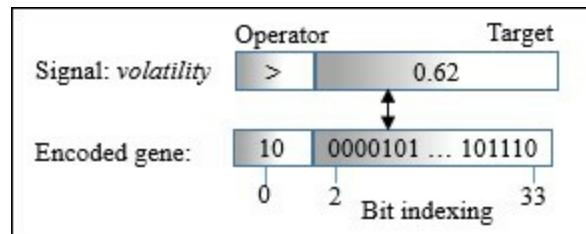
```

The implementation of the `strategies` value creates a `Pool` of signals (line 65) by converting the list of signals to a `treeset` (line 66). It breaks down the tree set into unique subtrees of `nSignals` nodes each. It instantiates a `subsetsIterator` iterator to traverse the sequence of subtrees (line 67) and converts them into a list (line 68) as arguments of the new chromosome (trading strategy) (line 69). The procedure to order the signals, `orderedSignals`, in the tree set has to be implicitly defined (line 70):

```
val orderedSignals = Ordering.by((sign: Signal) => sign.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 two-bit representation for the operator:



Encoding the trading signal: $volatility > 0.62$

Tip

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:

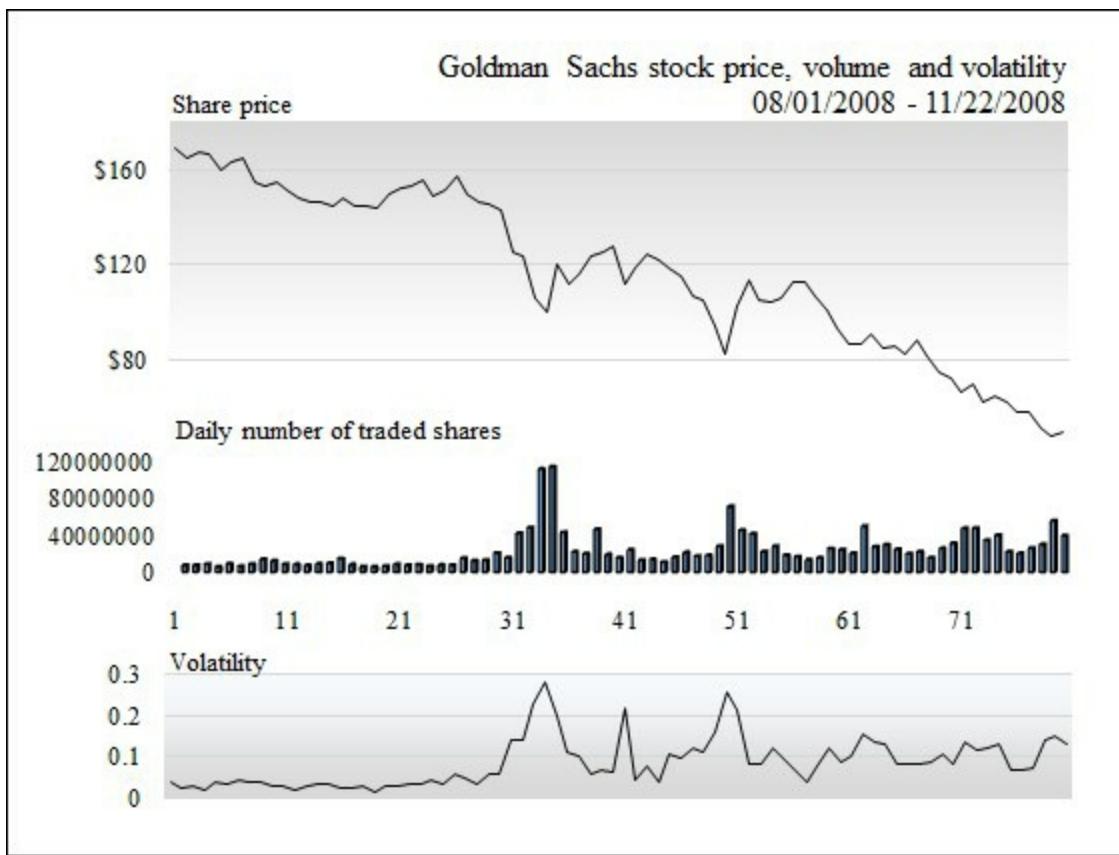
- **encoding e:** `(x: Double) => (x*100000).toInt`
- **decoding d:** `(x: Int) => x*1e-5`

Tip

All values are normalized; so, there is no risk of overflowing the 32-bit representation.

Test case – Fall 2008 market crash

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:



Sudden decrease in Goldman-Sachs stock price in Sept – Nov 2008

- Besides the variation of the price of the stock between two consecutive trading sessions (`dPrice`), the model uses the following parameters (or trading signals):
- `dVolume`: This is the relative variation of the volume between two consecutive trading sessions
- `dVolatility`: This is the relative variation of volatility between two consecutive trading sessions

- `volatility`: This is the relative volatility within a trading session
- `vPrice`: This is the relative difference of the stock opening and closing price

The naming convention for the trading data and metrics is described in the *Trading data* section under *Technical analysis in the Appendix*.

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.

Creating trading strategies

The input to the genetic algorithm is the population of trading strategies. Each strategy consists of a combination of three trading signals and each trading signal is a tuple (signal id, operator, and target value).

The first step is to extract the model parameters as illustrated for the variation of the stock price volume, volatility, and relative volatility between two consecutive trading sessions (line 71):

```
val NUM_SIGNALS = 3

def createStrategies: Try[Pool[Double, Signal]] = for {
  src <- DataSource(path, false, true, 1) //71
  price <- src.get(adjClose)
  dPrice <- delta(price, -1.0)
  volume <- src.get(volume)
  dVolume <- delta(volume, 1.0)
  volatility <- src.get(volatility)
  dVolatility <- delta(volatility, 1.0)
  vPrice = src.get(vPrice)
} yield { //72
```

```

val factory = new StrategyFactory(NUM_SIGNALS) //73
val weights = dPrice //74

factory += ("dvolume", 1.1, GREATER_THAN, dVolume,weights)
factory += ("volatility", 1.3, GREATER_THAN,
            volatility.drop(1), weights)
factory += ("vPrice", 0.8, LESS_THAN,
            vPrice.drop(1), weights)
factory += ("dVolatility", 0.9, GREATER_THAN,
            dVolatility, weights)
factory.strategies
}

```

The purpose is to generate the initial population of strategies that compete to become relevant to the decline of the price of stocks of Goldman Sachs. The initial population of trading strategies is generated by creating a combination from four trading signals weighted by the variation in stock price: $?(volume) > 1.1$, $?(volatility) > 1.3$, $?(close-open) < 0.8$, and $volatility > 0.9$.

The `delta` method computes the variation of a trading variable between consecutive trading sessions. It invokes the `TSeries.zipWithShift` method introduced in the *Time series in Scala* section of [Chapter 3, Data Pre-Processing](#):

```

def delta(xt: DblVec, a: Double): Try[DblVec] = Try {
  zipWithShift(xt, 1).map{case(x, y) => a*(y/x - 1.0)}
}

```

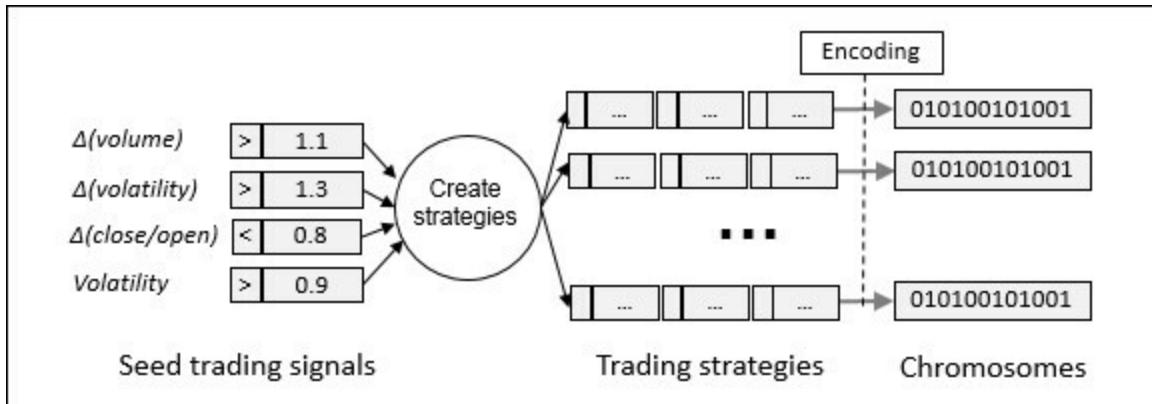
The trading strategies are generated by the `StrategyFactory` class introduced in the previous section (line 73). The weights for the trading strategies are computed as the difference `dPrice` of the price of the stock between two consecutive trading sessions (line 74). The option of unweighted trading strategies is selected by replacing the `weights` with the average price variation as follows:

```

val avWeights = dPrice.sum/dPrice.size
val weights = Vector.fill(dPrice.size)(avWeights)

```

The generation of the initial population of trading strategies is illustrated in the following diagram:



Design for the generation of the initial population of trading strategies

Configuring the optimizer

The configuration parameters for the execution of the genetic algorithm is categorized as:

- Tuning parameters such as crossover, mutation ratio, or soft limit on population growth
- Data representation parameters such as quantization and encoding
- Scoring scheme

The four configuration parameters for the GA are the maximum number of reproduction cycles (`MaxCycles`) allowed in the execution, the crossover (`xover`), the mutation ratio (`Mu`), and the soft limit function (`softLimit`) to control the population growth. The soft limit is implemented as a linearly decreasing function of the number of cycles (`n`) to retrain the growth of the population as the execution of the genetic algorithm progresses:

```

val Xover = 0.8    //Probability(ratio) for cross-over
val Mu = 0.4      //Probability(ratio) for mutation
val MaxCycles = 400 //Max. number of optimization cycles

val CutoffSlope = -0.003          //Slope linear soft limit
val CutoffIntercept = 1.003        //Intercept linear soft limit
val softLimit = (n: Int) => CutoffSlope *n + CutoffIntercept

```

The trading strategies are converted into chromosomes through `encoding` (line 75). A quantization scheme, `quant`, has to be implicitly defined in order

to encode the target value in each trading signal (line 76):

```
implicit val encoding = defaultEncoding //75

val RCoef = 1024.0 //Quantization ratio
implicit val quant = Quantization[Double](
  (x: Double) => (x*RCoef).float.toInt, (n: Int) =>n/RCoef
) //76
```

The `scoring` function computes the `cost` or unfitness of a trading strategy (chromosome) by applying the `score` function to each of the three trading signals (genes) it contains (line 77):

```
val scoring = (chr: Chromosome[Double, Signal]) => {
  val signals: List[Gene[Double]] = chr.code
  chr.cost = log(signals.map(_.score).sum + 1.0) //77
}
```

Finding the best trading strategy

The trading strategies generated by the factory in the `createStrategies` method are fed to the genetic algorithm as the `initial` population (line 79). The upper limit to the population growth is set at eight times the size of the initial population (line 78):

```
createStrategies.map(strategies => {
  val limit = strategies.size <<3 //78
  val initial = Population[Double, Signal](limit, strategies) //79

  val config = GAConfig(Xover, MU, MaxCycles, softLimit) //80
  val solver = GASolver[Double, Signal](
    config, scoring, Some(tracker) //81
  )
  (solver |> initial).map(
    _.fittest.map(_.symbolic).getOrElse("NA")
  ) match {
    case Success(results) => show(results)
    case Failure(e) => error("GAEval: ", e)
  } //82
})
```

The configuration `config` (line 80), the `scoring` function, and optionally a tracker function is all you need to create and execute the genetic algorithm,

`solver` (line 81). The partial function generated by the `|>` operator transforms the `initial` population of trading strategies into the two `fittest` strategies (line 82).

The documented source code for the monitoring function, tracker, and miscellaneous methods is available online at the Packt Publishing website.

Tests

The cost function C (or unfitness) score of each trading strategy is weighted for the rate of decline of the price of the Goldman Sachs stock. Let's run two tests:

- Evaluation of the configuration of the genetic algorithm with the score weighted by the price variation
- Evaluation of the genetic algorithm with an unweighted scoring function

The weighted score

The score is weighted by the variation of the price of the stock GS. The test uses three different sets of crossover and mutation ratios: (0.6, 0.2), (0.3, 0.1), and (0.2, 0.6). The best trading strategy for each scenario are as follows:

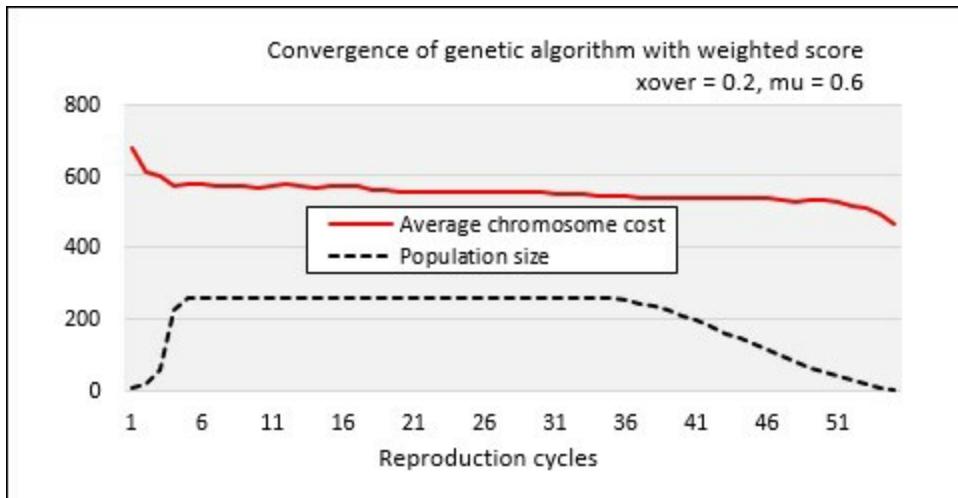
- 0.6-0.2: change < 0.82 dVolume > 1.17 volatility > 1.35 cost= 0.0
fitness: 1.0E10
- 0.3-0.1: change < 0.42 dVolume > 1.61 volatility > 1.08 cost= 59.18
fitness: 0.016
- 0.2-0.6: change < 0.87 dVolume < 8.17 volatility > 3.91 cost= 301.3
fitness: 0.003

The fittest trading strategy for each case does not differ much from the initial population for one or several of the following reasons:

- The initial guess for the trading signals was good
- The size of the initial population is too small to generate genetic diversity
- The test does not consider the rate of decline of the stock price

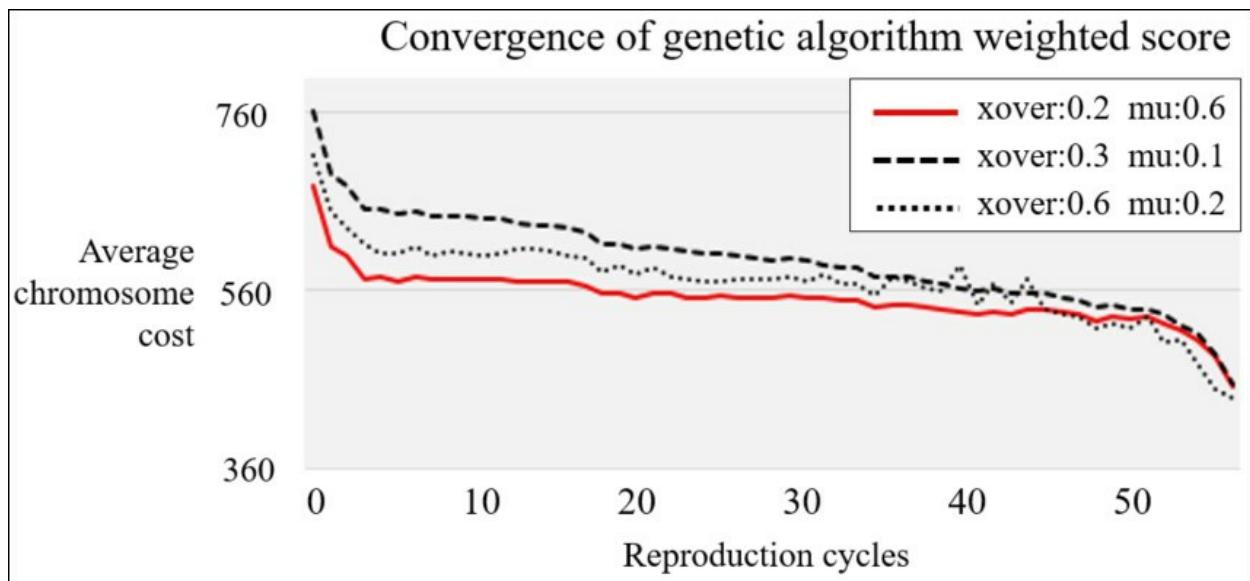
The execution of the genetic algorithm with cross-over = 0.2 and mutation = 0.6 produces a trading strategy that is inconsistent with the first two cases. One possible explanation is the fact that the cross-over applies always to the first of the three genes, forcing the optimizer to converge towards a local minimum.

Let's examine the behavior of the genetic algorithm during execution. We are particularly interested in the convergence of the average chromosome unfitness score. The average chromosome unfitness is the ratio of the total unfitness score for the population over the size of the population: Have a look at the following graph:



Convergence of genetic algorithm for crossover ratio 0.2 and mutation 0.6 with weighted score

The GA converges quite quickly and then stabilizes. The size of the population increases through crossover and mutation operations until it reaches the maximum of 256 trading strategies. The soft limit or constraint on the population size kicks in after 23 trading cycles. The test is run again with different values of crossover and mutation ratio, as shown in the following graph:



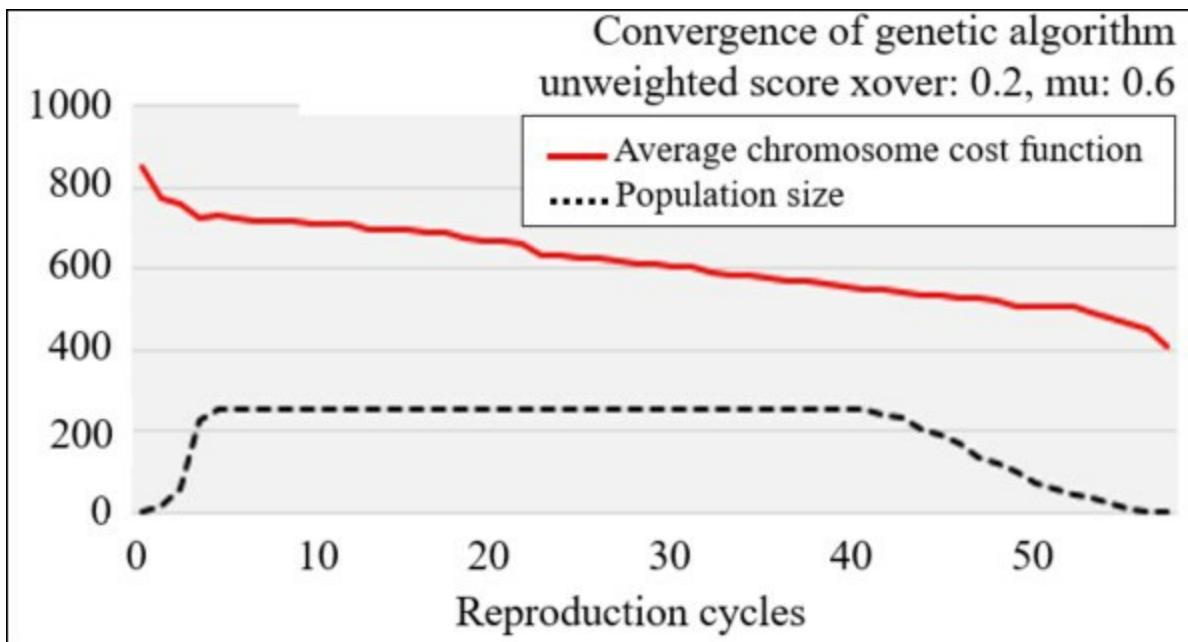
Impact of crossover and mutation ratio on the convergence of genetic algorithm with weighted score

The profile of the execution of the genetic algorithm is not overly affected by the different values of crossover and mutation ratios. The chromosome unfitness score for the high crossover ratio, 0.6, oscillates as the execution progresses. In some cases, the unfitness score between chromosomes is so small that the GA recycles the same few trading strategies.

The quick decline in the unfitness of the chromosomes is consistent with the fact that some of the fittest strategies were part of the initial population. It should, however, raise some concerns that the GA locked on a local minimum early on.

The unweighted score

The execution of a test that is similar to the previous one with the unweighted trading strategies (trading strategies that use the average price variation) scoring formula produces some interesting results, as shown in the following graph:



Convergence of genetic algorithm for crossover ratio 0.4 and mutation 0.4 with unweighted score

The profile for the size of the population is like the test using weighted scoring. However, the chromosome average cost pattern is somewhat linear. The unweighted (or averaging) adds the rate of decline of the stock price into the score (cost).

Tip

Complexity of scoring function

The complexity of the scoring (or computation of the cost) formula increases of the odds the genetic algorithm not converging properly. The possible solutions to the convergence problem are as follows:

- Make the weighting function additive (less complex)
- Increase the size and diversity of the initial population

Advantages and risks of genetic algorithms

It should be clear by now that genetic algorithms provide scientists with a powerful optimization tool for problems that:

- Are poorly understood.
- May have more than one good enough solution.
- Have discrete, discontinuous, and non-differentiable functions.
- Can be easily integrated with the rules or policies engine (see the *Learning classifiers systems* section in [Chapter 15, Reinforcement Learning](#)).
- Do not require deep domain knowledge. The genetic algorithm generates new solution candidates through genetic operators without the need to specify constraints and initial conditions.
- Do not require knowledge of numerical methods such as the *Newton-Raphson*, *conjugate gradient*, or *L-BFGS* as optimization techniques, which frighten those with little inclination for mathematics.

However, evolutionary computation is not suitable for problems for which:

- A fitness or scoring function cannot be quantified or even defined
- There is a need to find the global minimum or maximum, not just a good enough solution
- The execution time must be either predictable or very short (real-time optimization) or both

Summary

This concludes our journey into the unsettling world of evolutionary computing. There is a lot more to evolutionary computing than genetic algorithms such as artificial life, swarm intelligence, or differential evolution. Moreover, our description of genetic algorithm did not include genetic programming that applies genetic operators to trees and directed graphs of expressions.

This chapter dealt with a review of the different NP problems, the key components and genetic operators, an application of the fitness score to financial trading strategy, and the subtle variation in encoding predicates. The chapter concluded with an overview of the advantages and risks of genetic algorithms.

Genetic algorithms are an important element of a special class of reinforcement learning introduced in the *Learning classifiers systems* section of [Chapter 15, Reinforcement learning](#). The next two chapters describe the most common reinforcement learning techniques starting with Bayesian inference algorithms, [Chapter 14, Multiarmed Bandits](#).

Chapter 14. Multiarmed Bandits

This chapter is the first installment in our description of the reinforcement learning technique. In the context of a problem with multiple solutions, multiarmed bandit techniques attempt to acquire behavioral knowledge on many solutions (exploration) while at the same time applying the most rewarding solution (exploitation) to maximize success. The balancing act between experimenting and acquiring new knowledge and leveraging previously acquired knowledge is the core concept behind multiarmed bandit techniques.

This chapter covers the following topics:

- Exploration versus exploitation trade-off
- Minimization of cumulative regret
- Epsilon-greedy algorithm
- Upper confidence bound technique
- Context free Thompson sampling

K-armed bandit

The K-armed bandit is a metaphor representing a casino slot machine with k pull levers (or arms). The user or customer pulls any one of the levers to win a predefined reward. The objective is obviously to select the lever that will provide the user with the highest reward:



2-Arm bandit

Although the challenge could be defined as an optimization problem, it is a classification problem. There is no ability to assign any of the K levers a specific reward; therefore, the model is generated through reinforcement learning [14:1].

The basic concept of reinforcement learning is illustrated in the following diagram:

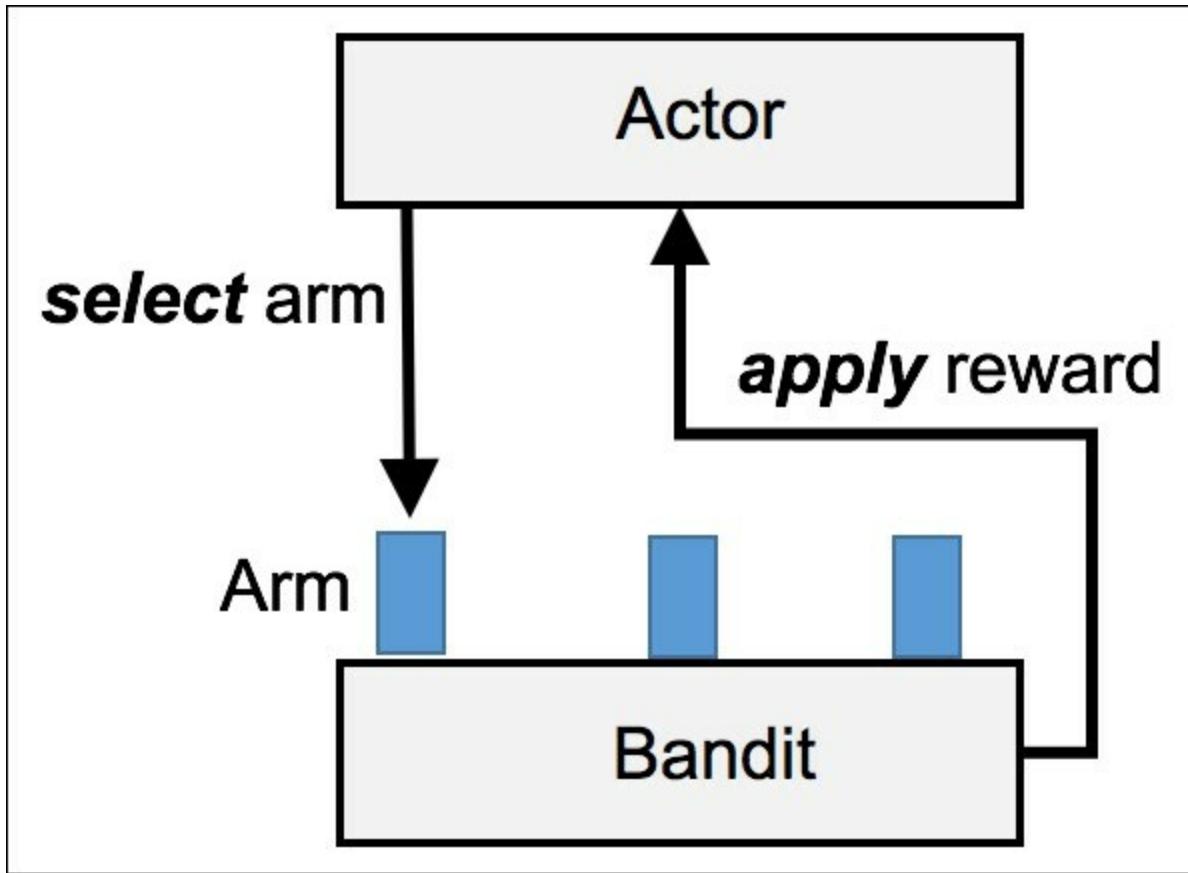


Illustration of action and reward for a multiarmed bandit

The actor selects and plays the arm with the highest estimate reward, collects the reward, and re-computes the statistics or performance for the selected arm.

Note

Markov decision process

The K-armed bandit problem can be defined as the one state **Markov decision process (MDP)** (see the *Markov decision process* section in [Chapter 15, Reinforcement Learning](#)).

Exploration-exploitation trade-offs

It is impossible to estimate which arm provides the higher estimated reward without playing all of them. This phase is known as exploration. It is required to collect information or knowledge on each of the arms to estimate the best arm.

Once the best arm is found, it can be used or exploited to maximize reward. This phase is known as exploitation. The chance of selecting the best arm during exploration is very low. The cost of exploration known as regret increases with the number of arms. The exploration ends when an arm emerges as the most rewarding.

Expected cumulative regret

There is an obvious penalty in exploring the potential reward for alternative arms: it is the cost (or regret) of an action that generates a less than optimal reward. The objective is to maximize the reward or, equivalently, minimize the regret overtime.

Let's consider T consecutive actions on any of the k arms; let μ_t be the reward for each of the action t with $0 \leq t < T$. If μ^* is the mean value of the reward for the optimal arm, then the expected value for the cumulative regret R after T actions is defined as:

$$E[R] = T\mu^* - \sum_{t=0}^{T-1} \mu_{a_t}$$

Bayesian Bernoulli bandits

The simplest form of bandit follows the Bernoulli distribution of successes and failures [14:2]. The algorithm predicts that a given arm will be selected as a winner.

The probability $p(X=1)$ becomes:

- $(\text{successes} + 1) / (\text{successes} + \text{failures} + 1)$ if the prediction is correct
- $\text{successes} / (\text{successes} + \text{failure} + 1)$ if the prediction is incorrect

Let's define the arm of a Bernoulli bandit as the trait `Arm`. It has four fields:

- `id`: Identifier of this arm of the bandit (line 1)
- `successes`: Number of times this arm has been correctly selected (level with the highest reward) (line 2)
- `failures`: Number of times this arm has been incorrectly selected (line 3)

The code is as follows:

```
trait Arm {  
    val id: Int //1  
    var successes: Int = _ //2  
    var failures: Int = _ //3  
  
    def mean: Double = successes.toDouble / (successes+failures) //4  
  
    def apply(winner: Arm): Unit = //5  
        if (id == winner.id) {  
            winner.successes += 1  
            failures += 1  
        } else {  
            winner.failures += 1  
            success += 1  
        }  
}
```

The mean is computed as a Bernoulli probability $p(x = 1)$ (line 4). The apply

method updates the number of successes and failures for the selected arm that returns a reward +1 and the other arm that return a reward 0 (line 5).

Note

A/B testing

A/B testing is considered a special case of K-armed bandit. This experimental method consists of pursuing an exploration strategy during testing/evaluation and then switches to a permanent exploitation strategy with the selected configuration (arm).

Let's implement the K-armed bandit by creating the trait `ArmedBandit`. The trait is parameterized by the type of arm. The K-armed bandit is an implicit transform that takes an arm as input (type `U`) and returns the current cumulative regret as a Double (line 1). The trait has two attributes:

- The list of `arms` associated with the bandit, as an abstract value (line 2)
- The cumulative regret variable `cumulRegret` (line 3)

The code is as follows:

```
trait ArmedBandit[U <: Arm] extends ITransform[U, Double]{ //1
    protected[this] val arms: List[U] //2
    protected[this] var cumulRegret: Double = _ //3

    def play: U //4
    def apply(successArm: U): Unit //5

    override def |> : PartialFunction[U, Try[Double]] = { //6
        case successArm: U =>
            val playedArm = select //7
            this(successArm)
            cumulRegret += playedArm.mean - successArm.mean //8
            Try(cumulRegret)
    }
}
```

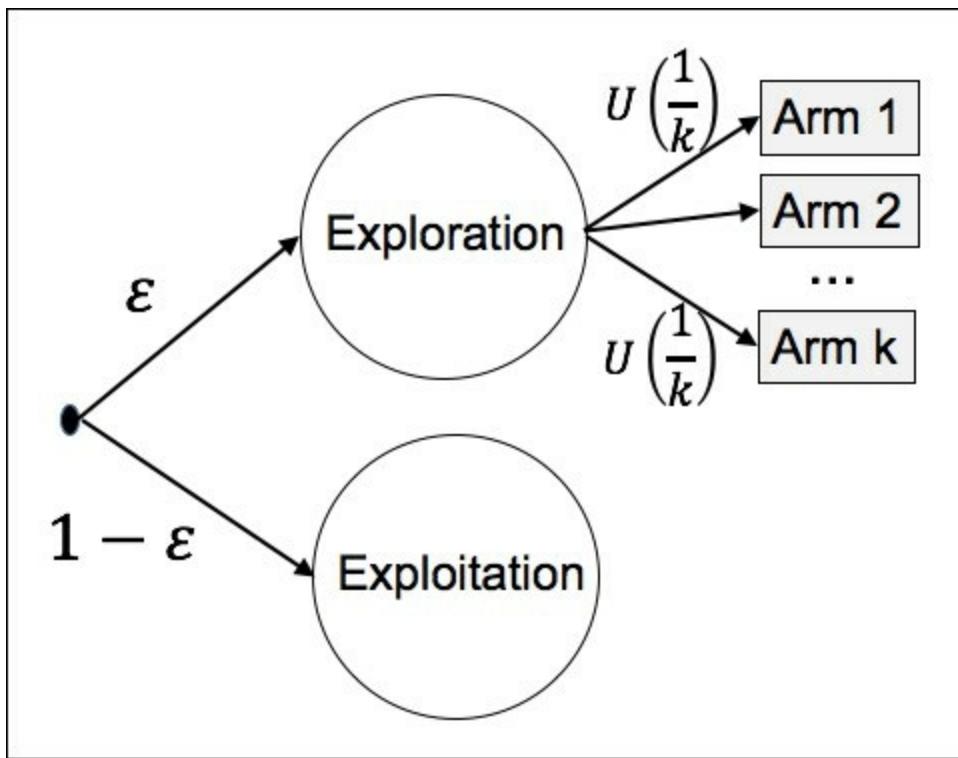
The abstract method `play` relies on a combination of exploration and exploitation to pull the appropriate arm (line 4). The method `apply` updates the parameters of each arm with the reward [success or failure] (line 5).

The overridden method `|>` (line 6) implements the feedback loop of selecting an arm using either exploration or exploitation (line 7), compared to the successful arm (arm with the reward +1) to compute the cumulative regret (line 8).

Epsilon-greedy algorithm

The Epsilon-greedy algorithm is the simplest methodology to balance exploration and exploitation. The algorithm consists of selecting an arm using a uniform random distribution U , ϵ percentage of the time. The arm with the highest estimated reward is selected $1 - \epsilon$ percentage of the time [14:3].

This method relies on a single parameter, ϵ , to maximize the total cumulative reward:



Exploration-exploitation balancing in epsilon-greedy algorithm

So far, we have seen that the selection of the best arm (the arm with the highest expected reward) is only driven by the history of rewards. This configuration is known as a context-free multi-arm bandit. This approach does not consider any impact an action on over a system. An arm represents one of the configuration parameters of a system which state is altered by actions (state - transition). For instance, one of the weights of a logistic

regression may also be modeled as an bandit arm. The state or set of configuration parameters is known as a context and the methodology is referred to a contextual multi-arm bandit.

The expected cumulative regret is computed as:

$$E[R] = T \varepsilon \left(\mu^* - \frac{1}{K} \right) \sum_{j=0}^{K-1} \mu_j$$

Let's implement the ϵ -greedy selection with the class `EGreedy` parameterized with a subtype of `Arm`. The class implements the generic `ArmedBandit` trait. It takes two arguments: the exploration/exploitation parameter `epsilon` (line 9) and the list of `arms` associated with the bandit (line 10):

```
class EGreedy[U <: Arm] (
  epsilon: Double, //9
  arms: List[U]) extends ArmedBandit[U] { //10

  override def play: U =
    if(nextDouble < epsilon) arms(nextInt(arms.size)) //11
    else arms.sortBy(_.mean).head //12

  override def apply(selectedArm: U): Unit = //13
    arms.foreach(_.update(selectedArm))
}
```

The `play` method is overridden and relies on the combination of exploration (line 11) and exploitation to pull the appropriate arm (line 12). The exploitation formula consists of ranking the arms by their mean reward in increasing value and returns the first element. The `apply` method updates the Bernoulli parameters for each arm with the appropriate reward [success or failure] (line 13).

The next step is to evaluate the epsilon-greedy selection algorithm in the context of cumulative regret.

Let's consider `numArms` arms with default type (line 14). We simulate the sequence of actions on the bandit by rewarding systematically the arm with index 3 for the first five actions (line 15), then the arm with index 5 for the

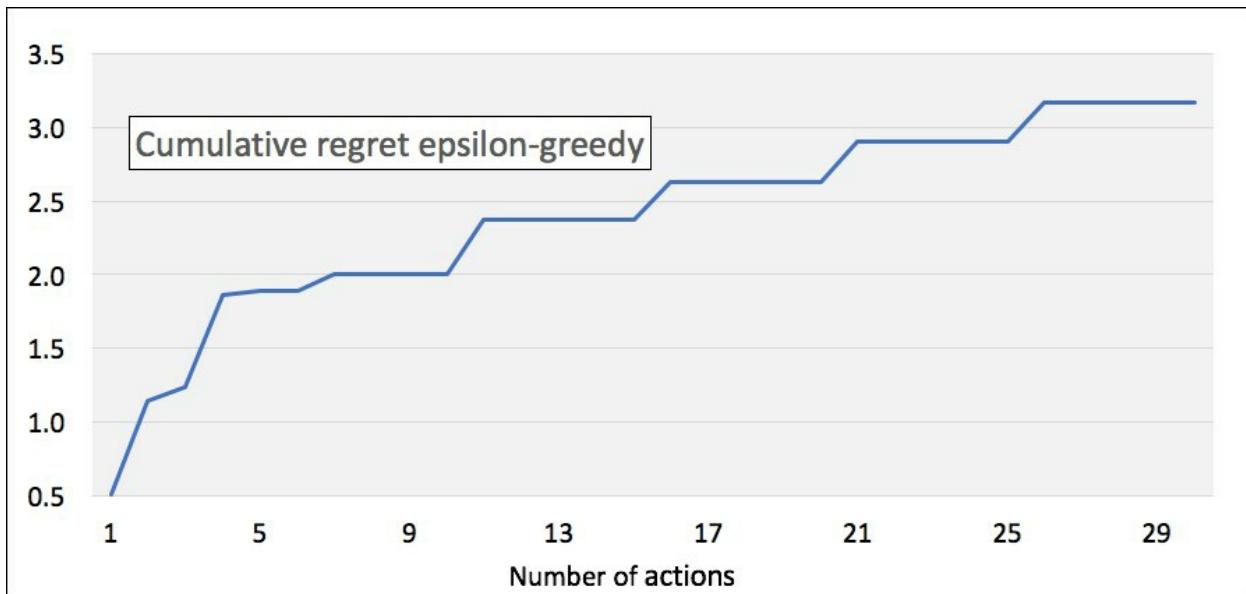
following 25 actions (line 16):

```
val epsilon = 0.3
val numArms = 10
val arms = List.tabulate(numArms) (n => //14
    new Arm { override val id: Int = n }
)

val epsilonGreedy = new EGreedy[Arm] (epsilon, arms)
val pfnCumulRegret = epsilonGreedy.|>

val simulated = (0 until 30).map( n =>
    if(n < 5) pfnCumulRegret(arms(3)) //15
    else pfnCumulRegret(arms(5)) //16
)
simulated.map( _.foreach(show_) ) //17
```

Let's plot the cumulative regret along the sequence of actions (line 17):



Cumulative regret for a 10-arm bandit using epsilon-0.3 greedy selection

The cumulative regret increases significantly in the first few actions during the pure exploration phase. However, the cumulative regret increase moderates as we assign or force the arm of index 3, then the arm of index 5, to be the arm with the highest reward (exploitation driven by a single arm). The occasional exploration (uniformly random value $<$ epsilon) causes a small increase in cumulative regret during the (predominantly) exploitation

phase.

Thompson sampling

Thompson sampling is a simple strategy, introduced 80 years ago, that has received renewed attention in recent years. It is widely used in advertising displays, marketing surveys, and financial analysis. Thompson sampling is also a Bayesian strategy, known as *probability matching*: The probability of selecting the arm n is the probability that n is the arm with the maximum reward [14:4].

The strategy can be summarized as:

- Assign a uniform distribution for each arm, prior to the selection
- Select arm n with a posterior probability that increases with the probability that n is optimal (probability matching)

Bandit context

So far, we have discussed K-armed bandits that do not maintain a state or context. It is assumed that all the arms are identical and only parameterized by their mean reward (successes and failures in the case of Bernoulli bandits). However, real-world applications, such as product recommendations or advertising targeting, require arms (a product or advertising display) to have their own set of parameters or characteristics. In this case, the characteristics of each arm have an impact on the selection of the optimal arm.

The characteristics of the arm are known as a *context*. The arms of context-free bandits have no independent parameters or characteristics. Contextual bandits have parameterized arms. Arms are instances of a model in which parameters are constantly updated after each action. The optimal arm of a contextual bandit is the arm with the optimal context [14:5].

Note

Optimal arm in contextual bandits

Let $w_k = w_{jk}$ be the parameters for the k_{th} arm. Optimal arm m for a bandit with a context f .

$$m = \arg \max_{0 \leq j < k} f(\cdot | w_j)$$

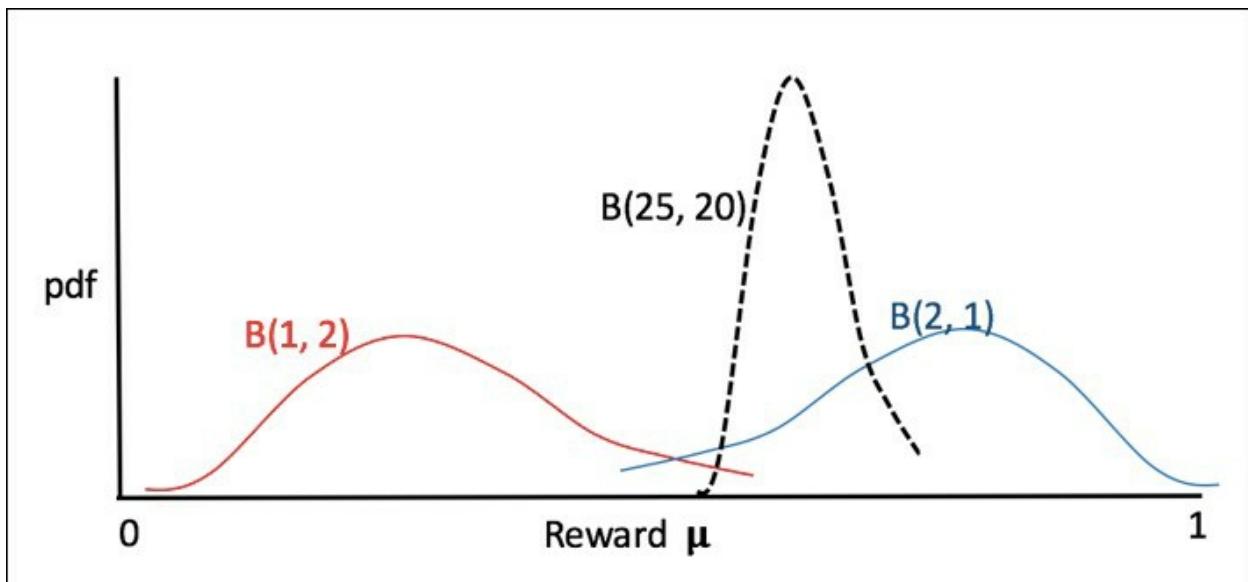
For instance, the predictive function f for display advertising with x_i being the advertiser parameters can be defined by the following logistic regression model with weight w_i (refer to the *The Logistic Regression* section in [Chapter 9, Regression and Regularization](#)):

$$f(x | w) = \frac{1}{1 + e^{-\sum x_i w_j}}$$

From now on, this chapter deals exclusively with context-free bandits.

Prior/posterior beta distribution

The probability of any arm being optimal is unknown before any action is taken. The information regarding any given arm (success and failure) becomes more accurate after each action. In other words, the variance on the mean reward, successes/(successes + failures) for any given arm decreases as the number of times this arm is selected increases:



Probability density function for Beta distribution

For each action, we compute the probability of a given arm being selected before the action (prior) and the probability of the same arm being selected after the action (posterior). These probabilities are computed using the Beta distribution with successes as the shape factor and successes + failures as the scale factor.

Note

Context-free Thompson sampling

Let $s_{i,t}$ be the number of successes for arm i for action t , $f_{i,t}$ the number of

failures for arm i for action t , $p_{i,t}$ the prior probability for the arm i for action t .

- Compute prior probability as $p_{i,t} = \text{Beta}(s_{i,t} + 1, f_{i,t+1})$ $i: 0, k-1$
- Select arm $j = \arg \max\{p_{j,t}\}$
- Observer or collect reward $r_{j,t} = \{0, 1\}$
- Compute posterior probability with $s_{i,t+1} = s_{i,t} + r_{i,t}$ and $f_{i,t+1} = f_{i,t} - r_{i,t}$

Implementation

The first step is to extend the trait Arm to support sampling from the Beta distribution: `BetaArm` (line 1):

```
trait BetaArm extends Arm { //1
    var pdf = new Beta(1, 1) //2

    override def apply(winner: Arm): Unit = { //3
        super(winner)
        pdf = new Beta(successes+1, failures+1) //4
    }
    def sample: Double = pdf.draw
}
```

The Beta distribution, `pdf`, is initialized as a uniform distribution or Beta distribution with a shape and scale value of 1 (line 2). The `update` method that reports the successful arm (the arm with the highest reward) for the current action (line 3) is overridden to update the shape and scale parameters (line 4).

The `CFThompsonSampling` class implements context-free Thompson sampling. It is quite simple as most of the functionality is inherited from the epsilon-greedy selection algorithm (line 5). It takes the same parameters, `epsilon` and the bandit defined as a list of arms for arm `BetaArm`:

```
class CFThompsonSampling[U <: BetaArm] (
    epsilon: Double,
    arms: List[U]) extends EGreedy[U](epsilon, arms) { //5

    override def play: U =
        if(nextDouble < epsilon) arms(nextInt(arms.size)) //6
        else arms.sortBy(_.sample).head //7
}
```

The `play` method in Thompson sampling differs slightly from the epsilon-greedy selection algorithm:

- The exploration action is identical to the epsilon-greedy (line 6)

- The exploitation action sorts the arms by their sampled value from the Beta distribution (descending order) then selects the arm with the highest mean reward (line 7)

Tip

The select method executes a sort of asymptotic time $O(n \log n)$. The algorithm is easy to understand, but is computationally inefficient. A better alternative involves creating a max-heap structure to update the order of the arms.

Tip

Efficient implementation of exploration

Simulated exploration and exploitation

Let's evaluate the impact of Epsilon on the cumulative regret. This simplistic evaluation relies on the simulation of exploration and exploitation phases. The concept is to assign the best arm (the arm with the highest simulated reward) using the following simulation:

- Each arm sequentially for the first 40 actions (simulated exploration)
- One specific arm (index 6) for the remaining actions (simulated exploitation):

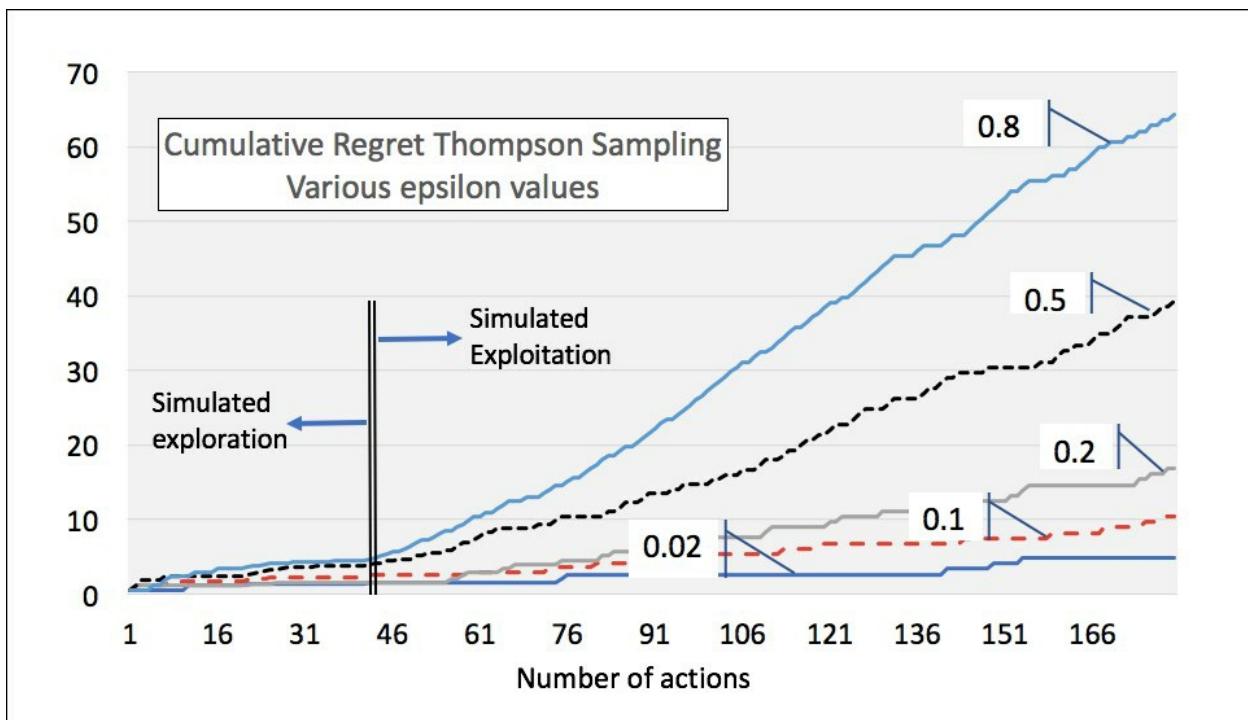
```
val epsilon = 0.01
val numArms = 20
val numActions = 180
val startPureExploration = numArms<<1
val arms = List.tabulate(numArms) (n =>
    new BetaArm { override val id: Int = n } //8
)

val cFTS = new CFThompsonSampling[BetaArm] (epsilon, arms)
val pfnCFTS = cFTS.|> //9

val simulated = (0 until numActions).map( n =>
    if(n >= startPureExploration)
        pfnCFTS(arms(6)) //11
    else
        pfnCFTS(arms(n % numArms)) //10
)
simulated.map( _.foreach( show(_) ) )
```

Context-free Thompson sampling is from a list of arms of type `BetaArm` (line 8). The partial function, `pfnCFTS` (line 9), takes the played arm as input and returns the cumulative estimate regret. The simulation applies a simple round-robin algorithm for the first `startPureExploration` actions (line 10), and then exploits one given arm (index 6) for the remaining actions (line 11).

The experiment is performed for various values of epsilon 0.02, 0.1, 0.2, 0.5, and 0.8. We should expect that the cost of randomization (cumulative regret) increases with the value of *epsilon* as illustrated with the following plot:



Simulated cumulative regret for Thompson sampling with various epsilon values

The results of the experiment confirm our assumption regarding the impact of the ratio exploration/exploitation on the cumulative regret. The five experiments have a similar profile of cumulative regret during the simulated exploration phase (the first 40 actions) and diverge significantly during the simulated exploitation phase.

Tip

Typical cumulative regret behavior

The previous experiment relies on simulation for exploration and exploitation to highlight the impact of the epsilon threshold value on the cumulative regret. You should expect very different behavior in real-world applications.

The Thompson sampling concept is most valuable when applied to the contextual bandit problem [14:6]. Although interesting, the contextual Thompson sampling topic is beyond the scope of this book.

Contrary to the Upper Bound Confidence algorithm described in the next chapter, the Thompson attempts to match the asymptotic lower bound of the expected cumulative regret.

Upper bound confidence

The UCB approach assumes that the expected reward of an action is linearly dependent on the d-dimension context.

Confidence interval

Intuitively, the confidence on the reward for a given arm increases as the arm is played. The variance of the reward is significantly high when the arm has been rarely played. The variance or confidence interval symbolizes the uncertainty on the reward of the arm. As the arm gets played, the confidence interval decreases.

The goal of the exploration is to play arms with a large confidence interval around the mean value of their reward so they can be a potential candidate for exploitation.

The following diagram illustrates the process of exploration [14:7]:

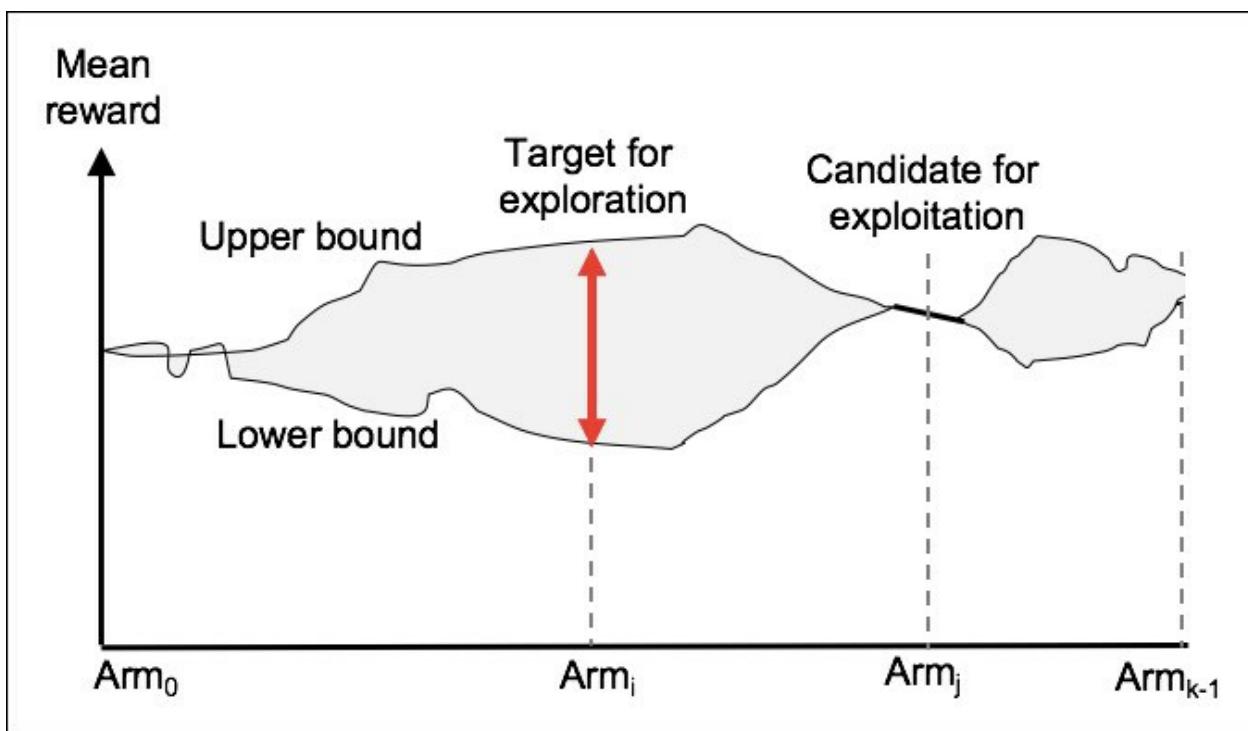


Illustration of confidence interval for k arms during the exploitation-exploration cycle

The exploration phase favors the arm i being played to reduce its confidence interval. The exploration phase uses arm j because it has the highest mean

reward with a very small confidence factor.

The challenge is to score each arm with a formula that considers the mean reward (exploitation) and confidence interval (exploration). The optimistic solution uses the upper bound of the confidence interval to score and then rank the arms.

UCB1 is a simple implementation of the UCB algorithm. Its regret grows only logarithmically with the number of actions taken.

The algorithm has two steps:

1. For each action on arm j , compute the mean reward μ_j and the number of times n_j the arm j has been played.
2. Play the arm l that maximizes the mean + upper bound value of the confidence interval.

Note

UCB for Bernoulli variables

For t actions, n_j number of time arm j has been selected, μ_j mean reward for the arm j , and then the next action selects arm l :

$$l = \arg \max_{0 \leq j < k} f \left\{ \mu_j + \sqrt{2 \frac{\log t}{n_j}} \right\}$$

Implementation

Let's first define the type for the arm suitable to the UCB algorithm. The simplest design is to sub-class the generic `Arm` (with the trait `CountedArm`) (line 1) and add a counter, `cnt`, (line 2) for the number of times the arm is played:

```
trait CountedArm extends Arm { //1
    var cnt: Int = _ //2

    def play: Unit = cnt += 1 //3
    def score(numActions: Int): Double =
        mean + sqrt(2.0*log(numActions)/cnt) //4
}
```

The method `play` is invoked each time this counted arm is played. It updates the counter `cnt` for the number of times this arm is played (line 3). The `score` method implements the formula for the Bernoulli UCB algorithm (line 4).

The `UCB1` algorithm is a special case of K-armed bandit, therefore it extends the trait `ArmedBandit` (line 5):

```
class UCB1[U <: CountedArm] (
    override val arms: List[U]
) extends ArmedBandit[U] { //5

    var numActions: Int = _

    override def play: U = {
        numActions += 1 //6
        arms.sortBy(_.score(numActions)).head //7
    }

    override def apply(successArm: U): Unit =
        arms.foreach(_.update(successArm))
}
```

The overridden method, `play`, updates the total number of actions (line 6), ranks the arms by decreasing score, and plays the arm with the highest score or estimated reward (line 7).

Note

Thompson sampling versus UCB1

Unlike Thompson sampling, the UCB1 algorithm is deterministic. However, it requires scoring every arm for each experiment or play, while Thompson sampling allows the batching of plays and relies on random distribution sampling. Finally, UCB1 achieves the asymptotic upper bound for the expected cumulative regret while the Thompson sampling converges towards the lower bound.

Summary

This concludes the first of two chapters dedicated to reinforcement learning. In this chapter, we learned to balance exploration (learning) and exploitation (executing) by:

- Managing and reducing the confidence interval across the arms
- Applying the simple epsilon-greedy selection for exploring underplayed arms
- Leveraging the concept of probability matching through Thompson sampling for context-free bandits
- Using Upper Confidence Bounds to model the confidence interval as a function of the number of plays

The K-armed bandit problem is a viable solution for simple problems in which the interaction between the actor (player) and the environment (bandit) relies on a single state and immediate reward.

The next chapter introduces alternative approaches to multiarmed bandits for more complex, multi-state problems using value-actions and the Markovian decision process.

Chapter 15. Reinforcement Learning

You may wonder, at this stage of the book, how robotics, gaming, and autonomous systems leverage machine learning. The answer lies in a field of AI known as reinforcement learning. For those with no familiarity with reinforcement learning, I highly recommend you read the seminal book on reinforcement learning by **R. Sutton** and **A. Barto** [11:1] if you are interested to know about its origin, purpose, and scientific foundation.

The first part of this chapter focuses on the **Q-learning algorithm**. The second part is dedicated to **Learning Classifier Systems (LCS)**, which combine reinforcement learning techniques with evolutionary computing, introduced in the previous chapter. Learning classifiers are an interesting breed of algorithm that is not commonly included in literature dedicated to machine 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

Note

Learning classifier systems implementation

The section on **Learning Classifier Systems (LCS)** is mainly informative with an overview of the principles behind these models and a skeleton of implementation.

Reinforcement learning

The need for an alternative to traditional learning techniques arose with the design of the first autonomous systems.

Understanding the challenge

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 a knowledge not only of the current state of the environment but also of 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 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 the event 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 times, 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

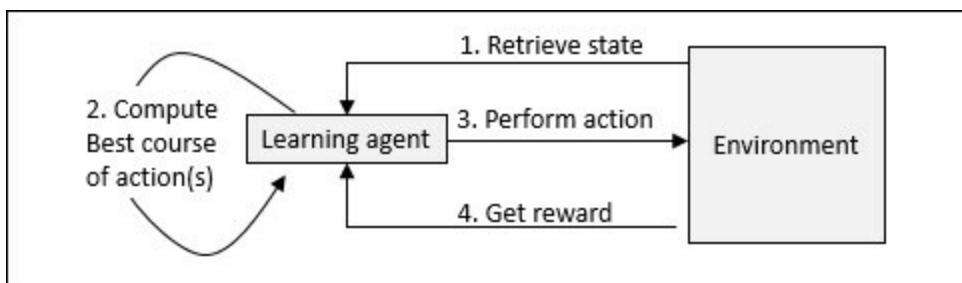
Reinforcement learning introduces a new terminology as listed here, quite different from that of older machine learning techniques:

- **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 reward than any other state. A high cumulative reward prevents the best policy from being dependent on the initial state during training.
- **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 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 action and being rewarded or penalized for it [11:3]. You can visualize these agents as robots navigating through an unfamiliar terrain or a maze. Robots use reinforcement learning as part of their reasoning process after all. The following diagram gives the overview architecture of the reinforcement learning agent:



The four state transitions of reinforcement learning

The agent collects the state of the environment, and selects and then executes the most appropriate action. The environment responds to the action by changing its state and rewarding or punishing the agent for the action.

The four steps of an episode or learning cycle are as follows:

1. The learning agent retrieves or is notified of a new state of the environment.
2. The agent evaluates and selects the action that may provide the highest reward.
3. The agent executes the action.
4. The agent collects the reward or penalty and applies it to calibrate the learning algorithm.

Note

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 should 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](#)).

Tip

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 a *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 the policy.

Intuitively, the value of a policy is the sum of all the rewards collected because of the sequence of actions taken by the agent. In practice, an action over the policy 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 .

Note

Transition and rewards matrices

Transition and emission matrices were introduced in the *The hidden Markov model* section in [Chapter 7, Sequential Data Models](#).

The optimum policy, p^* , 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

$A = \{a_i\}$	Actions on the environment
$\pi_t = p(a_t s_t)$	Policy (or strategy) of the agent
$V^p(s_t)$	Value of the policy at the state
$p_t = p(s_{t+1} s_t, a_t)$	State transition probabilities from state s_t to state s_{t+1}
$r_t = p(r_{t+1} s_t, s_{t+1}, a_t)$	Reward of an action 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_p of a policy p at state s_t is the maximum expected reward R_t given the state s_t .

M1: Cumulative reward R_t and value function $V^p(s_t)$ for the state s_t given a policy p and a discount rate γ :

$$R_t \sum_{k=0}^{+\infty} \gamma^k r_{t+k}$$

$$V^\pi(s_t) = E\{R_t | s_t\}$$

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^p of a policy p can be formulated using the Markovian state transition probabilities p_t .

M2: Value function $V^p(s_t)$ for a state s_t and future state s_k with a reward r_k using the transition probability p_k , given a policy p and a discount rate γ :

$$V^\pi(s_t) = \sum_{a \in A} \pi_a \sum_k \left\{ p_k (r_k + \gamma \cdot V^\pi(s_k)) \right\}$$

$$V^*(s_t) = \max_{\pi} V^\pi(s_t)$$

$V^*(s_t)$ is the optimal value of state s_t across all the policies. The equations are known as the Bellman optimality equations.

Note

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 transitions 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:

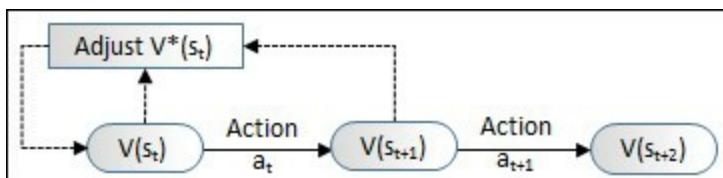


Illustration of the temporal difference algorithm

The iterative feedback loop used to adjust the value action on the state plays a role similar to the back propagation of errors in artificial neural networks or the minimization of the loss function in supervised learning. The adjustment algorithm has to:

- Discount the estimated value of the next state using the discount rate ?
- Strike a balance between the impact of the current state and the next state on updating the value at time t using the learning rate α

The iterative formulation of the first Bellman equation predicts $V^p(s_t)$, 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 d_t .

M3: Formula for tabular temporal difference d_t , for a value function $V(s_t)$ at state s_t , a learning rate α , a reward r_t , and a discount rate γ :

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

$$\hat{V}^\pi(s_t) = V^\pi(s_t) + \alpha \delta_t$$

An alternative to evaluating a policy using the value of the state, $V^p(s_t)$, is to use the value of taking an action on a state s_t known as the value of action (or action-value) $Q^p(s_t, a_t)$.

M4: Definition of the value Q of action at state s_t as the expectation of a reward R_t for an action at state s_t :

$$Q_t^\pi = Q^\pi(s_t, a_t) = E(R_t | s_t, a_t)$$

There are two methods to implement the temporal difference algorithm:

- **On-policy:** This is the value for the next best action that uses the policy
- **Off-policy:** This is the value for the next best action that does not use the policy

Let's consider the temporal difference algorithm using an off-policy method and its most commonly used implementation: Q-learning.

Action-value iterative update

Q-learning is a model-free learning technique using an off-policy method. It optimizes the action-selection policy by learning an action-value function.

Like any machine learning technique that relies on convex optimization, the Q-learning algorithm iterates through actions and states using the quality function, as described in the following mathematical formulation.

The algorithm predicts and discounts the optimum value of action, $\max\{Q_t\}$, 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.

M5: Q- learning action-value updating formula for a given policy p , set of states $\{s_t\}$, a set of actions $\{a_t\}$ associated to each state, s_t , a learning rate α , and a discount rate γ :

$$\tilde{Q}_t^\pi = Q_t^\pi + \alpha \left[r_{t+1} + \gamma \max_{a_{t+1}} Q_{t+1}^\pi - Q_t^\pi \right] Q_t^\pi = Q^\pi(s_t, a_t)$$

- A value of 1 for the learning rate α discards the previous state, while a value of 0 discards learning
- A value of 1 for the discount rate γ uses long-term rewards only, while a value of 0 uses the short-term reward only

Q-learning estimates the cumulative reward discounted for future actions.

Tip

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 the management and optimization of a portfolio of exchange-traded funds.

Implementation

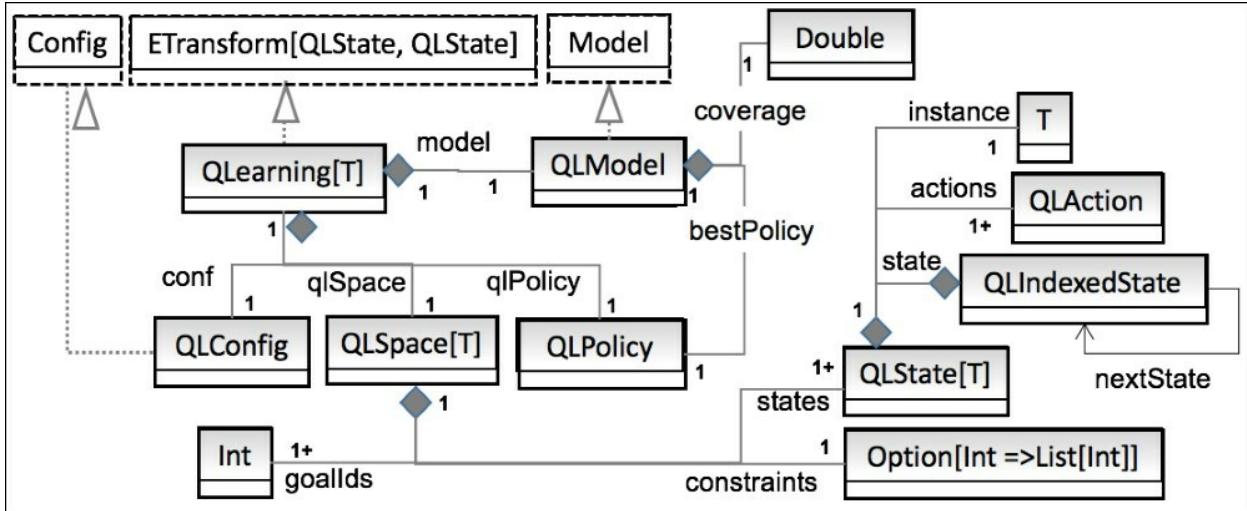
The first step toward the implementation of Q-learning in Scala is to identify the various components of Q-learning and their interaction.

Software design

The key components of the implementation of the Q-learning algorithm are defined as follows:

- The `QLearning` class implements training and prediction methods. It defines a data transformation of type `ETransform` using an explicit configuration of type `QLConfig`.
- The `QLSpace` class has two components: a sequence of states of type `QLState` and the identifier, `id`, of one or more goal states within the sequence.
- A state, `QLState`, contains a sequence of `QLAction` instances used in its transition to another state and a reference to the object or instance for which the state is to be evaluated and predicted.
- An indexed state, `QLIndexedState`, indexes a state in the search toward the goal state.
- An optional `constraint` function limits the scope of the search for the next most rewarding action from the current 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 key components of the Q-learning algorithm:



UML components diagram of the Q-learning algorithm

The states and actions

The `QLAction` class specifies the transition of one state with an identifier `from` to another state with the identifier `to`, as shown here:

```
class QLAction(from: Int, 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 identifier `id`, the list of actions to transition to some other states, and a property `prop` of parameterized type, as shown in the following code:

```
Case class QLState[T](id: Int,
  actions: List[QLAction] = List.empty,
  instance: T)
```

The state might not have any actions. This is usually the case of the goal or absorbing state. In this case, the list is empty. The parameterized `instance` is a reference to the object for which the state is computed.

The next step consists of creating the graph or search space.

The search space

The search space is the container responsible for any sequence of states. The `QLSpace` class takes the following parameters:

- The sequence of all the possible `states`
- The ID of one or several states that have been selected as `goals`

Tip

Why multiple goals?

There is absolutely no requirement that a state space must have a single goal. You can describe a solution to a problem as reaching a threshold or meeting one of several conditions. Each condition can be defined as a state goal.

The `QLSpace` class is implemented as follows:

```
class QLSpace[T](states: Seq[QLState[T]], goals: Array[Int]) {  
  
    val statesMap = states.map(st =>(st.id, st)) //1  
    val goalStates = new HashSet[Int]() ++ goals //2  
  
    def maxQ(state: QLState[T],  
            policy: QLPolicy): Double //3  
    def init(state0: Int) QLState[T] //4  
    def nextStates(st: QLState[T]): Seq[QLState[T]] //5  
  
}
```

The constructor for the `QLSpace` class generates a map, `statesMap`. It retrieves the state using its `id` (line 1), and the array of goals, `goalStates` (line 2). Furthermore, the `maxQ` method computes the maximum action-value, `maxQ`, for a state given a policy (line 3). The implementation of the method `maxQ` is described in the next section.

The `init` method selects an initial state for training episodes (line 4). The state is randomly selected if the `state0` argument is invalid:

```
def init(state0: Int): QLState[T] =  
  if(state0 < 0) {  
    val seed = System.currentTimeMillis + nextLong  
    states((new Random(seed)).nextInt(states.size-1))  
  }  
  else states(state0)
```

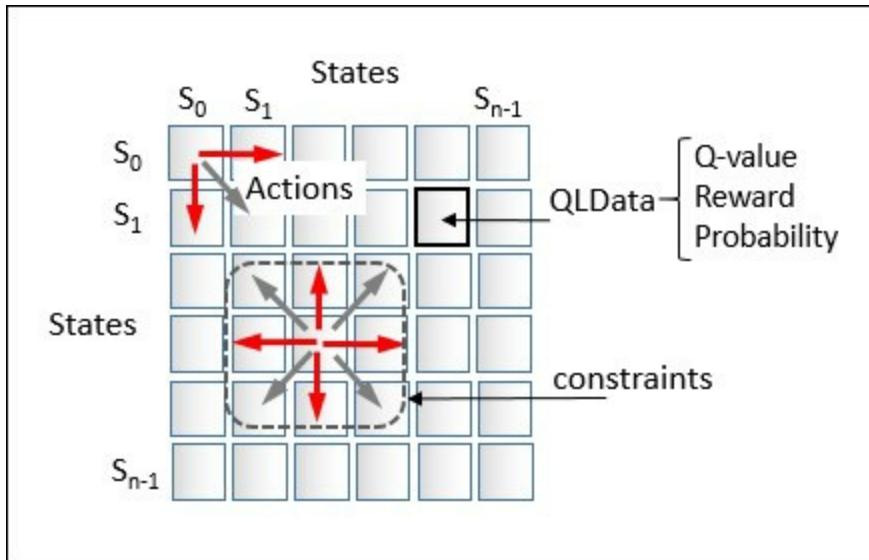
Finally, the `nextStates` method retrieves the list of states resulting from the execution of all the actions associated to the `st` state (line 5).

The search space `QLSpace` is created by the factory method `apply` defined in the `QLSpace` companion object, as shown here:

```
def apply[T] (  
  goals: Array[Int],  
  instances: Seq[T],  
  constraints: Option[Int => List[Int]]): QLSpace[T] = { //6  
  
  val r = Range(n, instances.size)  
  val states = instances.zipWithIndex.map{ case(x, n) =>  
    val validStates = constraints.map(_(n)).getOrElse(r)  
    val actions = validStates.view  
      .map(QLAction(n, _)).filter(n != _.to) //7  
    QLState[T](n, actions, x)  
  }  
  new QLSpace[T](states, goals)  
}
```

The `apply` method creates a list of states using the `instances` set, the `goals` and the constraining function, `constraints`, as input (line 6). Each state creates its list of `actions`. The actions are generated from this state to any other states (line 7).

The search space of states is illustrated in the following diagram:



State transition matrix with QLData (Q-value, reward, probability)

The function `constraints` limits the scope of the actions that can be triggered from any given state, as illustrated in the preceding diagram.

The policy and action-value

Each action has an action-value, a reward, and a potential probability. The probability variable is introduced simply to model the hindrance or adverse condition for an action to be executed. If the action does not have any external constraint, the probability is 1. If the action is not allowed, the probability is 0.

Tip

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 two values, reward, probability, and a variable `value` for the Q-value, as shown here:

```
class QLData(val reward: Double, val probability: Double) {  
    var value: Double = 0.0  
    def estimate: Double = value*probability  
}
```

Note

Reward and punishment

The probability in the `QLData` class represents the hindrance or difficulty to reach one state from another state. Nothing prevents you from using the probability as a negative reward or punishment. However, its proper definition is to create a soft constraint of a state transition for a small subset of a state. For most applications, the overwhelming majority of state transitions have a probability of 1.0 and therefore rely on the reward to guide the search toward the goal.

The `estimate` method adjusts the Q-value, `value`, with the probability to reflect any external condition that can impede the action.

Tip

Mutable data

You might wonder why the `QLData` class defines `value` as a variable instead as a value as recommended by the best Scala coding practices [11:4]. The reason is that an instance of an immutable class would be created for each action or state transition that requires updating the variable `value`.

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:

```
case class QLInput(from: Int, to: Int,
                     reward: Double, 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 the sequence of input define the policy of type `QLPolicy`. It is merely a data container, as shown here:

```
class QLPolicy(input: Seq[QLInput]) {
    val numStates = Math.sqrt(input.size).toInt //8

    val qlData = input.map(qlIn =>
        new QLData(qlIn.reward, qlIn.prob)) //9

    def setQ(from: Int, to: Int, value: Double): Unit =
        qlData(from*numStates + to).value = value //10

    def Q(from: Int, to: Int): Double =
        qlData(from*numStates + to).value //11
    def EQ(from: Int, to: Int): Double =
        qlData(from*numStates + to).estimate //12
    def R(from: Int, to: Int): Double =
        qlData(from*numStates + to).reward //13
    def P(from: Int, to: Int): Double =
        qlData(from*numStates + to).probability //14
}
```

The number of states, `numStates`, is the square root of the number of elements of the initial input matrix; `input` (line 8). The constructor initializes the `qlData` matrix of type `QLData` with the input data, `reward` and `probability` (line 9). The `QLPolicy` class defines the short-cut methods to update (line 10) and retrieves (line 11) the `value`, and retrieves the `estimate` (line 12), the `reward` (line 13), and the `probability` (line 14).

The Q-learning components

The `QLearning` class encapsulates the Q-learning algorithm, and more specifically the action-value updating equation. It is a data transformation of type `ETransform` with the explicit configuration of type `QLConfig` (line 16) (refer to the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#)):

```
class QLearning[T] (
    conf: QLConfig,
    qlSpace: QLSpace[T],
    qlPolicy: QLPolicy) //15
extends ETransform[QLState[T], QLState[T]](conf) { //16

    type U = QLState[T] //17
    type V = QLState[T] //18

    val model: Option[QLModel] = train //19
    def train: Option[QLModel]
    def nextState(iSt: QLIndexedState[T]): QLIndexedState[T]
    override def |> : PartialFunction[U, Try[V]]
    ...
}
```

The constructor takes the following parameters (line 15):

- Configuration of the algorithm, `config`
- Search space, `qlSpace`
- Policy, `qlPolicy`

`model` is generated or trained during the instantiation of the class (refer to the *Design template for classifier* section in the *Appendix*.) (line 19). The Q-learning algorithm is implemented as an explicit data transformation; therefore, the type `U` of the input element and the type `V` of the output element to the predictor `|>` are initialized as `QLState` (lines 17, 18).

The configuration of the Q-learning algorithm, `QLConfig`, specifies the learning rate, `alpha`; the discount rate, `gamma`; the maximum number of states (or length) of an episode, `episodeLength`; the number of episodes (or epochs) used in training, `numEpisodes`; and the minimum coverage, `minCoverage`, required to select the best policy as follows:

```

case class QLConfig(
  alpha: Double,
  gamma: Double,
  episodeLength: Int,
  numEpisodes: Int,
  minCoverage: Double) extends Config

```

The `QLearning` class has two constructors defined in its companion object that initialize the policy either from an input matrix of states or from a function that computes the reward and probabilities:

1. The client code specifies the function, `input`, to initialize the state of the Q-learning algorithm from the input data.
2. The client code specifies the functions to generate the `reward` and `probability` for each action or state transition.

The first constructor for the `QLearning` class, `apply`, passes the initialization of `states => Seq[QLInput]`, the sequence of references of `instances` associated to the states, and the scope constraining function, `constraints`, as arguments, beside the configuration and the goals (line 20):

```

def apply[T] (
  config: QLConfig, //20
  goals: Array[Int],
  input: => Seq[QLInput],
  instances: Seq[T],
  constraints: Option[Int => List[Int]]): QLearning[T] = {

  new QLearning[T](config,
    QLSpace[T](goals, instances, constraints),
    new QLPolicy(input))
}

```

The second constructor passes the input data, `xt` (line 21), the `reward` function (line 22), and the `probability` function (line 23), as well as the sequence of references of `instances` associated to the states and the scope constraining function, `constraints`, as arguments:

```

def apply[T] (
  config: QLConfig,
  goals: Array[Int],
  xt: DblVec, //21
  reward: (Double, Double) => Double, //22
  ...
)

```

```

probability: (Double, Double) => Double, //23
instances: Seq[T].
constraints: Option[Int =>List[Int]] =None) : QLearning[T] ={

    val r = xt.indices
    val input = ArrayBuffer[QLInput]() //24
    r.foreach(i =>
        r.foreach(j =>
            input.append(
                QLInput(i, j, reward(xt(i), xt(j)), probability(xt(i), xt(j)))
            )
        )
    )
    new QLearning[T](config,
        QLSpace[T](goals, instances, constraints),
        new QLPolicy(input))
}

```

The reward and probability matrices are used to initialize the `input` state (line 24).

The Q-learning training

Let us look at the computation of the best policy during training. First, we need to define a model class, `QLModel`, with the optimum policy (or path), `bestPolicy`, and its `coverage` as parameters:

```

case class QLModel(bestPolicy: QLPolicy, coverage: Double)
    extends Model[QLModel]

```

The creation of `model` consists of executing multiple episodes to extract the best policy. The training is executed in the `train` method: Each episode starts with a randomly selected state, as shown in the following code:

```

def train: Option[QLModel] = Try {
    val completions = (0 until config.numEpisodes)
        .map(epoch => if(train(-1)) 1 else 0).sum //25
    completions.toDouble/config.numEpisodes //26
}
.filter(_ > conf.minCoverage)
    .map(new QLModel(qlPolicy, _)).toOption

```

The method `train` iterates through the generation of the best policy starting

from a randomly selected state `config.numEpisodes` times (line 25). The state `coverage` is computed as the percentage of times the search ends with the goal state (line 26). The training succeeds only if the `coverage` exceeds a threshold value, `config.minAccuracy`, specified in the configuration.

Tip

Quality of the model

The implementation uses the accuracy to measure the quality of the model or best policy. The F1 measure (refer to the *Assessing a model* section in [Chapter 2, Data Pipelines](#), is not appropriate because there are no false positives.

The `train (state0: Int)` method does the heavy lifting at each episode (or epoch). It triggers the search by selecting either the initial state `state0` or a random generator `r` with a new seed, if `state0` is `< 0`, as shown in the following code:

```
case class QLIndexedState[T] (state: QLState[T], iter: Int)
```

The utility class `QLIndexedState` keeps track of the `state` at a specific iteration, `iter`, within an episode or epoch:

```
def train(state0: Int): Boolean = {  
  
    @tailrec  
    def search(iSt: QLIndexedState[T]): QLIndexedState[T] = ...  
  
    val finalState = search(  
        QLIndexedState(qlSpace.init(state0), 0)  
    )  
    if( finalState.index == -1) false //28  
    else qlSpace.isGoal(finalState.state) //29  
}
```

The implementation of `search` for the goal state from a predefined or random `state0` is a textbook implementation of the Scala tail recursion. Either the recursive search ends if there are no more states to consider (line 28) or the goal state is reached (line 29).

Tail recursion to the rescue

Tail recursion is a very effective construct to apply an operation to every item of a collection [11:5]. It optimizes the management of the function stack frame during the recursion. The annotation triggers a validation of the condition necessary for the compiler to optimize the function calls, as shown here:

```
@tailrec
def search(iSt: QLIndexedState[T]): QLIndexedState[T] = {
    val states = qlSpace.nextStates(iSt.state) //30

    if( states.isEmpty || iSt.iter >= config.episodeLength) //31
        QLIndexedState(iSt.state, -1)

    else {
        val state = states.maxBy(s =>
            qlPolicy.R(iSt.state.id, s.id) //32
        )
        if( qlSpace.isGoal(state) )
            QLIndexedState(state, iSt.iter) //33

        else {
            val fromId = iSt.state.id
            val r = qlPolicy.R(fromId, state.id)
            val q = qlPolicy.Q(fromId, state.id) //34

            val nq = q + config.alpha*
                (r + config.gamma*qlSpace.maxQ(state, qlPolicy)-q) //35
            qlPolicy.setQ(fromId, state.id, nq) //36
            search(QLIndexedState(state, iSt.iter+1))
        }
    }
}
```

Let us dive into the implementation for the Q action-value updating equation. The method `search` implements the mathematical expression M5 for each recursion.

The recursion uses the utility class `QLIndexedState` (state, iteration number in the episode) as an argument. First, the recursion invokes the `nextStates` method of `QLSpace` (line 30) to retrieve all the states associated with the current state, `iSt.state`, through its actions, `st.actions`, as shown here:

```
def nextStates(st: QLState[T]): Seq[QLState[T]] =  
  if( st.actions.isEmpty ) Seq.empty[QLState[T]]  
  Else st.actions.map(ac => statesMap.get(ac.to) )
```

The search completes and returns the current state, if the length of the episode (maximum number of states visited) is reached, the goal is reached, or there is no further state to transition to (line 31). Otherwise, the recursion computes the state to which the transition generates the higher reward R from the current policy (line 32). The recursion returns the state with the highest reward if it is one of the goal states (line 33). The method retrieves the current q action value (line 34) and r reward matrices from the policy, and then applies the equation to update the action-value (line 35). The method updates the action-value Q with the new value nq (line 36).

The action-value updating equation requires the computation of the maximum action-value associated with the current state, which is performed by the `maxQ` method of the `QLSpace` class:

```
def maxQ(state: QLState[T], policy: QLPolicy): Double = {  
  val best = states.filter(_ != state) //37  
    .maxBy(st => policy.EQ(state.id, st.id)) //38  
  policy.EQ(state.id, best.id)  
}
```

The method `maxQ` filters out the current state (line 37) and then extracts the best state, which maximizes the policy (line 38).

Note

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, independently of the fact that the goal state is reached for every episode.

Validation

A commercial application may require multiple types of validation mechanism regarding the states' transition, reward, probability, and Q-value matrices.

One critical validation is to verify that the user-defined `constraints` function does not create a dead-end in the search or training of Q-learning. The function `constraints` establishes the list of states that can be accessed from a given state through actions. If the constraints are too tight, some of the possible search paths may not reach the goal state. Here is a simple validation of the `constraints` function:

```
def validateConstraints(
    numStates: Int,
    constraints: Int => List[Int]): Boolean =
  (0 until numStates).exists( constraints(_) .isEmpty )
```

The prediction

The last functionality of the `QLearning` class is the prediction using the model created during training. The method `|>` predicts the optimum state transition (or action) from a given state, `state0`:

```
override def |> : PartialFunction[U, Try[V]] = {
  case state0: U if(isModel) => Try {
    if(state0.isGoal) state0 //39
    else nextState(QLIndexedState[T](state0, 0)).state) //40
  }
}
```

The data transformation `|>` returns itself if the input state, `state0`, is the goal (line 39) or computes the best outcome, `nextState`, (line 40) using another tail recursion, as follows:

```
@tailrec
def nextState(iSt: QLIndexedState[T]): QLIndexedState[T] = {
  val states = qlSpace.nextStates(iSt.state) //41
  if( states.isEmpty || iSt.iter >= config.episodeLength)
```

```

    iSt //42
} else {
    val fromId = iSt.state.id
    val qState = states.maxBy(s => //43
        model.map(_.bestPolicy.EQ(fromId, s.id)).getOrElse(-1.0))
    nextState(QLIndexedState[T](qState, iSt.iteR+1)) //44
}
}

```

The method `nextState` executes the following sequence of invocations:

1. Retrieves the eligible states that can be transitioned to from the current state, `iSt.state` (line 41).
2. Returns the states if there is no more states or if the method does not converge within the maximum number of allowed iterations, `config.episodeLength` (line 42).
3. Extracts the state, `qState` with the most rewarding policy (line 43).
4. Increments the iteration counter, `iSt.iteR` (line 44).

Tip

Exit condition

The prediction ends when no more states are available or the maximum number of iterations within the episode is exceeded. You can define a more sophisticated exit condition. The challenge is that there is no explicit error or loss variable/function that can be used except the temporal difference error.

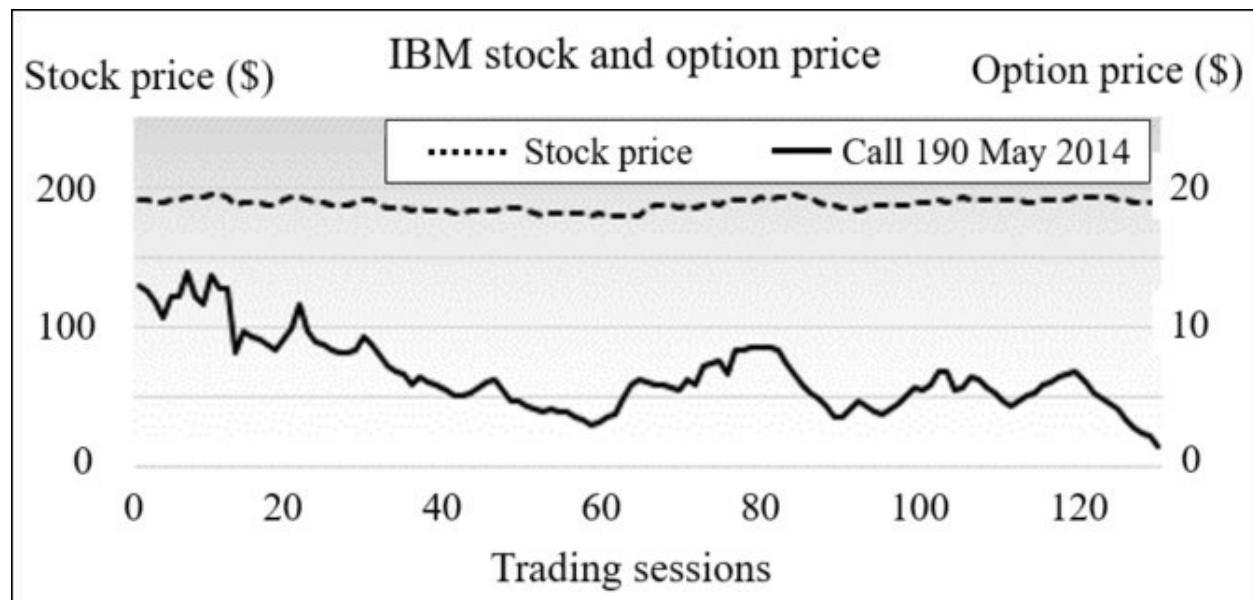
The prediction method `|>` returns either the best possible state, or `None` if the model cannot be created during training.

Option trading using Q-learning

The Q-learning algorithm is used in many financial and market trading applications [11:6]. Let 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 the *Appendix*.) 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, the 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:



IBM stock and call \$190 May 2014 pricing in May-Oct 2013

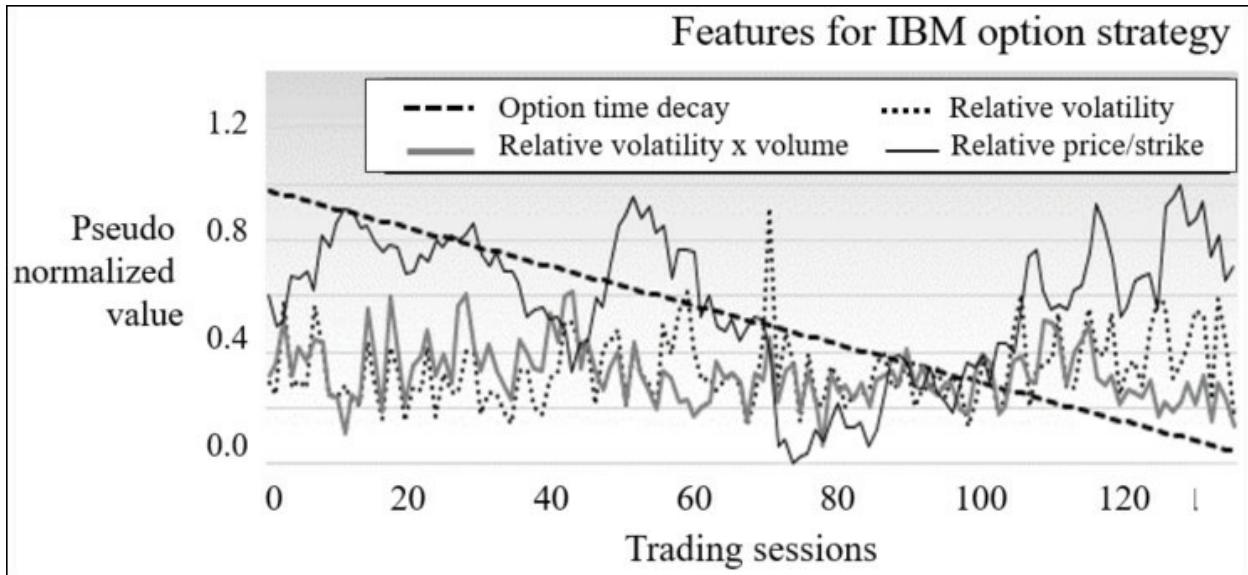
The price of an option depends on the following parameters:

- Time to expiration of the option (time decay)
- The price of the underlying security
- The volatility of returns of the underlying asset

A pricing model usually does not consider the variation in trading volume of the underlying security. Therefore, 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** (`timeToExp`): This is the time to expiration once normalized over [0, 1].
- **Relative volatility** (`volatility`): This is the relative variation of the price of the underlying security within a trading session. It is different from the more complex volatility of returns defined in the Black-Scholes model, for example.
- **Volatility relative to volume** (`vltyByVol`): This is the relative volatility of the price of the security adjusted for its trading volume.
- **Relative difference between the current price and strike price** (`priceToStrike`): This measures the ratio of the difference between price and strike price to the strike price.

The following graph shows the four normalized features for the IBM option strategy:



Normalized relative stock price volatility, volatility relative to trading volume, and price relative to strike price for IBM stock

The implementation of the option trading strategy using Q-learning consists of the following steps:

1. Describing the property of an option.
2. Defining the function approximation.
3. Specifying the constraints on the state transition.

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 (line 45) as follows:

```
class OptionProperty(timeToExp: Double, volatility: Double,
    vltyByVol: Double, priceToStrike: Double) { //45

    val toArray = Array[Double](
        timeToExp, volatility, vltyByVol, priceToStrike
    )
}
```

```
)  
}
```

Tip

Modular design:

The implementation avoids sub-classing 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 the data, `src`.
- The minimum time decay or time to expiration, `minTDecay`: Out-of-the-money options expire worthless and in-the-money options have a very different price behavior as they get closer to the expiration date (refer to the *Options trading* section in the *Appendix*). Therefore, the last `minTDecay` trading sessions prior to the expiration date are not used in the training of the model.
- The number of steps (or buckets), `nSteps`: It is used in approximating the values of each feature. For instance, an approximation of four steps creates four buckets `[0, 25]`, `[25, 50]`, `[50, 75]`, and `[75, 100]`.

The implementation of the `OptionModel` class is as follows:

```
class OptionModel(symbol: String, strikePrice: Double,  
                 ds: DataSource, minExpT: Int, nSteps: Int) {  
  
    val propsList = (for {  
        src <- ds  
    }  
    )
```

```

    price <- src.get(adjClose)
    volatility <- src.get(volatility)
    nVolatility <- normalize(volatility)
    vltyByVol <- src.get(volatilityByVol)
    nVltyByVol <- normalize(vltyByVol)
    priceToStrike <- normalize(price.map(p =>(1.0 - strikePrice/p
} yield {

nVolatility.zipWithIndex.:/:(List[OptionProperty] ()) { //46
  case(xs, (v,n)) => {
    val normDecay = (n+minExpT).toDouble/(price.size+minExpT)
    new OptionProperty(
      normDecay, v, nVltyByVol(n), priceToStrike(n)
    ) :: xs
  }
}.drop(2).reverse //48
})
.getOrElse(List.empty[OptionProperty].)

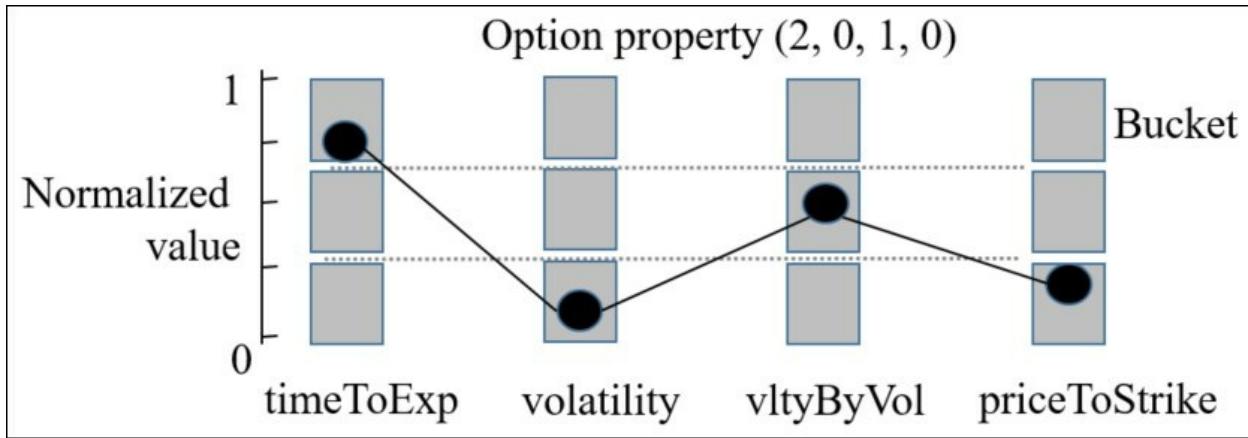
def quantize(o: Array[Double]): Map[Array[Int], Double]
}

```

The factory uses the `zipWithIndex` Scala method to represent the index of the trading sessions (line 46). All feature values are normalized over the interval $[0, 1]$, including the time decay (or time to expiration) of the `normDecay` option (line 47). The instantiation of the `OptionModel` class generates a list of `OptionProperty` elements if the constructor succeeds (line 48); it generates an empty list otherwise.

Quantization

The four properties of the option are continuous values, normalized as a probability $[0, 1]$. The states in the Q-learning algorithm are discrete and require a quantization or categorization known as a *function approximation*, although a function approximation scheme can be quite elaborate [11:9]. Let us settle for a simple linear categorization as illustrated in the following diagram:



Quantization of the state of a traded option

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 $34 = 81$ states or potentially $38-34 = 6480$ actions! The maximum number of states for one buckets function approximation and n features is l^n with a maximum number of $l^{2n} - l^n$ actions.

Note

Quantization or function approximation guidelines

The design of the function to approximate the state of options must address the following two conflicting requirements:

- Accuracy demands a fine-grained approximation
- Limited computation resources restrict the number of states, and therefore, the level of approximation

The `quantize` method of the `OptionModel` class converts the normalized value of each option property of features into an array of bucket indices. It returns a map of profit and loss for each bucket keyed on the array of bucket indices, as shown in the following code:

```
def quantize(o: DblArray): Map[Array[Int], Double] = {
  val mapper = HashMap[Int, Array[Int]]() //49
  val acc = propsList.view //50
```

```

    .map( _.toArray)
    .map( toArrayInt( _ ) ) //51
    .map(ar => {
      val enc = encode(ar) //52
      mapper.put(enc, ar)
      enc
    }).zip(o)./:(_acc) {
      case (_acc, (t,y)) => { //53
        _acc += (t, y)
        _acc
      }
    }
  acc.map {case (k, (v,w)) => (k, v/w) } //54
    .map {case( k,v) => (mapper(k), v) }.toMap
}

```

The method creates a `mapper` instance to index the array of buckets (line 49). An accumulator, `acc`, of type `NumericAccumulator` extends `Map[Int, (Int, Double)]` and computes the tuple (number of occurrences of features on each bucket, the sum of the increase or decrease of the option price) (line 50). The `toArrayInt` method converts the value of each option property (`timeToExp`, `volatility`, and so on) into the index of the appropriate bucket (line 51). The array of indices is then encoded (line 52) to generate the `id` or index of a state. The method updates the accumulator with the number of occurrences and the total profit and loss for a trading session for the option (line 53). It finally computes the reward on each action by averaging the profit and loss on each bucket (line 54).

A view is used in the generation of the list of `OptionProperty` to avoid unnecessary object creation.

The source code for the methods `toArrayInt` and `encode` and the `NumericAccumulator` are documented and available online.

Putting it all together

The final piece of the puzzle is the code that configures and executes the Q-learning algorithm on one or several options on a security, IBM:

```
val STOCK_PRICES = getPath("rl/IBM.csv")
val OPTION_PRICES = getPath("rl/IBM_O.csv")
val QUANTIZER = 4
val src = DataSource(STOCK_PRICES, false, false, 1) //55

val model = for {
    option <- Try( createOptionModel(src) ) //56
    oPrices <- DataSource(OPTION_PRICES, false).extract //57
    _model <- createModel(option, oPrices) //58
} yield _model
```

The preceding implementation creates the Q-learning model with the following steps:

1. Extract the historical prices for the IBM stock by instantiating a data source, `src` (line 55).
2. Create an `option` model (line 56).
3. Extract the historical prices, `oPrices` for option *Call \$190 May 2014* (line 57).
4. Create the model `_model` with a predefined goal, `goalstr` (line 58):

```
val STRIKE_PRICE = 190.0
val MIN_TIME_EXPIRATION = 6

def createOptionModel(src: DataSource): OptionModel =
    new OptionModel("IBM", STRIKE_PRICE, src,
                    MIN_TIME_EXPIRATION, QUANTIZER)
```

Let's look at the method `createModel`, which takes the option-pricing model, `option`, and the historical prices for options, `oPrice`, as arguments:

```
val LEARNING_RATE = 0.2
val DISCOUNT_RATE = 0.7
val MAX_EPISODE_LEN = 128
val NUM_EPISODES = 80
```

```

def createModel(option: OptionModel, oPrices: DblArray,
               alpha: Double, gamma: Double): Try[QLModel] = Try {

    val qPriceMap = option.quantize(oPrices) //59
    val numStates = qPriceMap.size

    val qPrice = qPriceMap.values.toVector //60
    val profit = zipWithShift(qPrice, 1).map{case (x, y) => y-x}//61
    val maxProfitIndex = profit.zipWithIndex.maxBy(_._1)._2 //62

    val reward = (x: Double, y: Double) => exp(30*(y - x)) //63
    val probability = (x: Double, y: Double) =>
        if(y < 0.3*x) 0.0 else 1.0 //64

    if( !validateConstraints(profit.size, neighbors)) //65
        throw new IllegalStateException(" ... ")

    val config = QLConfig(
        alpha, gamma, MAX_EPISODE_LEN, NUM_EPISODES //66
    )
    val instances = qPriceMap.keySet.toSeq.drop(1)

    QLearning[Array[Int]](config, Array[Int](maxProfitIndex),
                           profit, reward, probability, instances, Some(neighbors))
        .getModel //67
}

```

The method quantizes the option prices map, `oPrices` (line 59), extracts the historical option prices, `qPrice` (line 60), computes the profit as the difference in the price of the option between two consecutive trading sessions (line 61), and computes the index, `maxProfitIndex`, of the trading session with the highest profile (line 62). The state with the index `maxProfitIndex` is selected as the goal.

The input matrix is automatically generated using the `reward` and `probability` function. The `reward` function rewards the state transition proportionally to the profit (line 63). The `probability` function punishes the state transition for which the loss $y - x$ is greater than $0.3 * x$ by setting the probability value to 0 (line 64).

Note

Initialization of rewards and probabilities

In our example, the reward and probabilities matrices are automatically generated through two functions. An alternative approach consists of initializing these two matrices using either historical data or educated guesses.

The method `validateConstraints` of the companion object `QLearning` validates the constraints function, `neighbors`, as described in the section, *Validation* (line 65).

The last two steps consist of creating a configuration, `config`, for the Q-learning algorithm (line 66) and training the model by instantiating the `QLearning` class with the appropriate parameters, including the method, `neighbors`, which defines the neighboring states for any give state (line 67). The method, `neighbors`, is described in the documented source code available online.

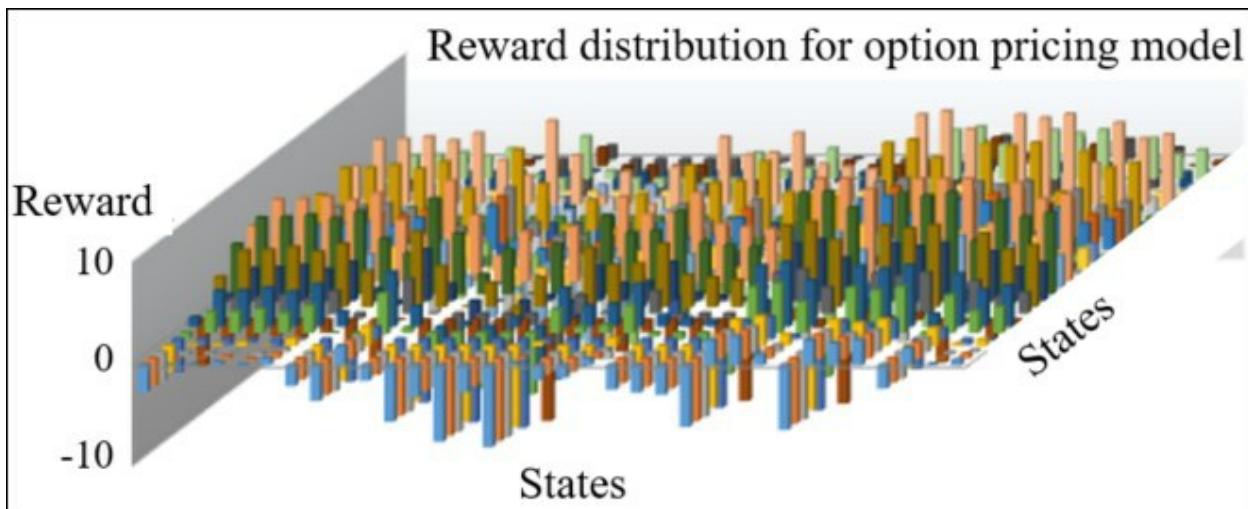
Note

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 rewards matrix:



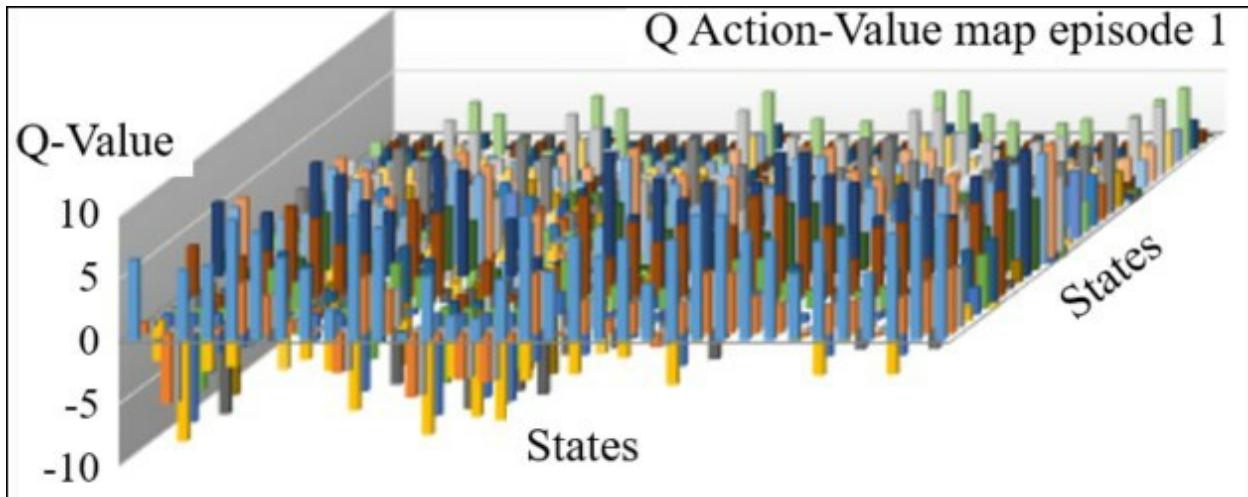
Rewards matrix for the option-pricing Q-learning strategy

The graph visualizes the distribution of the rewards computed from the profit and loss of the option. The xy plane represents the actions between states. The states' IDs are listed on x and y -axes. The z -axis measures the actual value of the reward associated with each action.

The reward reflects the fluctuation in the price of the option. The price of an option has a higher volatility than the price of the underlying security.

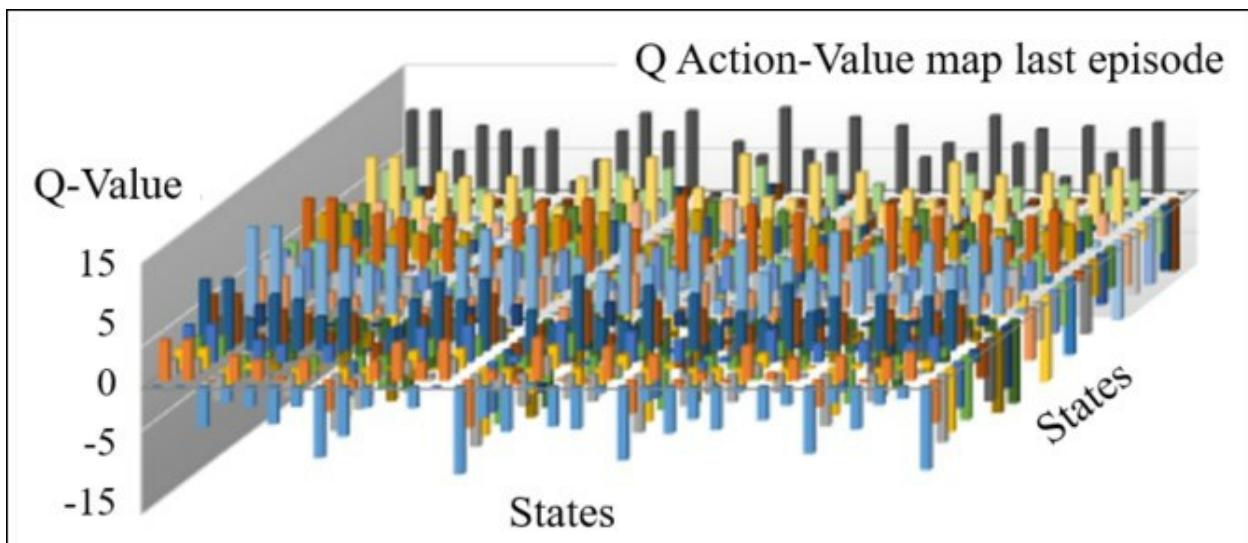
The xy reward matrix R is rather highly distributed. Therefore, we select a small value for the learning rate, 0.2, to reduce the impact of the previous state on the new state. The value for the discount rate, 0.7, accommodates the fact that the number of states is limited. There is no reason to compute the future discounted reward using a long sequence of states. The training of the

policies generates the following action-value matrix Q of 34 states by 34 states after the first episode:



Q Action-value matrix for the first episode (epoch)

The distribution of the action-values between states at the end of the first episode reflects the distribution of the reward across state-to-state action. The first episode consists of a sequence of nine states from an initial randomly selected state to the goal state. The action-value map is compared with the map generated after 20 episodes in the following graph:

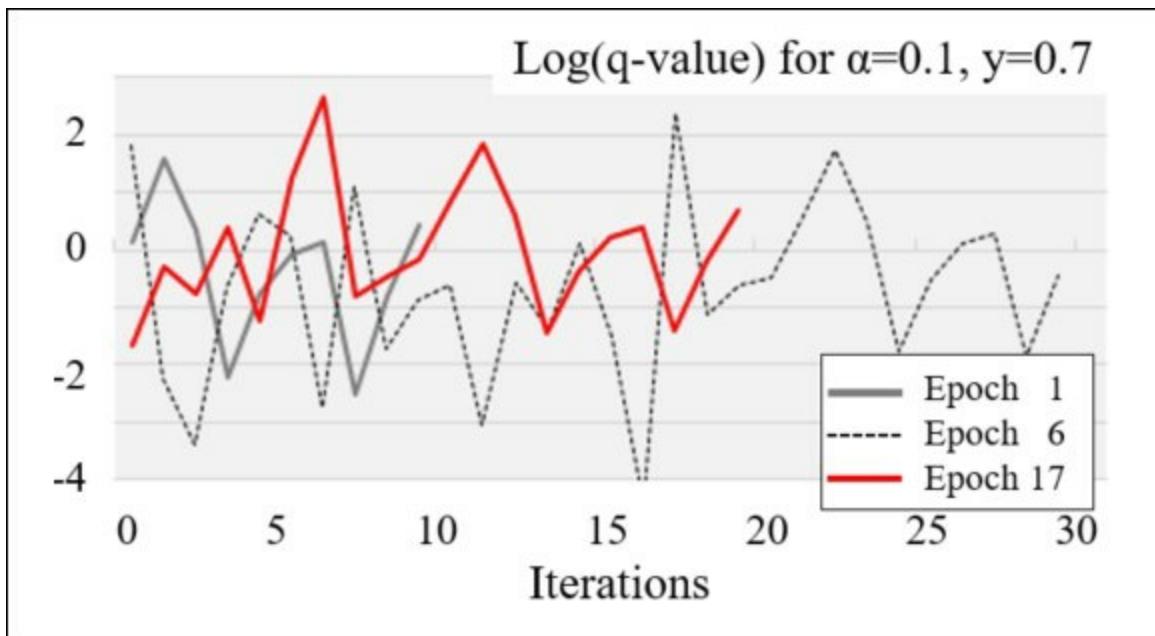


Q Action-value matrix for the last episode (epoch)

The action-value map at the end of the last episode shows some clear patterns. Most of the rewarding actions transition from many 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 of most buckets having at least one data point.
- However, a loose function approximation or quantization tends to group quite different states into the same bucket.
- A bucket with a very low number can potentially mischaracterize one property or feature of a state.

The next test is to display the profile of the log of the Q-value (`QLData.value`) as the recursive search (or training) progress for different episodes or epochs. The test uses a learning rate $\alpha = 0.1$ and a discount rate $\gamma = 0.9$:

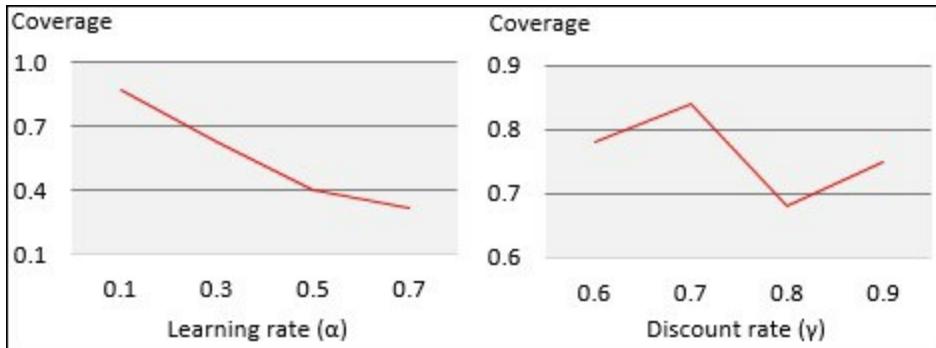


Profile of the log (Q-Value) for different epochs during Q-learning training

The preceding chart illustrates the fact that the Q-value for each profile is independent of the order of the epochs during training. However, the length of the profile (or the number of iterations to reach the goal state) depends on

the initial state selected randomly in this example.

The last test consists of evaluating the impact of the learning rate and discount rate on the coverage of the training:



Training coverage versus learning rate and discount rate

The coverage (the percentage of episodes or epochs for which the goal state is reached) decreases as the learning rate increases. The result confirms the general rule of *using learning rate < 0.2*. A similar test to evaluate the impact of the discount rate on the coverage is inconclusive.

Pros and cons of reinforcement learning

Reinforcement learning algorithms are ideal for the following problems:

- Online learning
- The training data is small or 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 because the maintenance of the states, action graph, and rewards matrix becomes challenging
- The execution is not always predictable in terms of scalability and performance

Learning classifier systems

J. Holland introduced the concept of **Learning Classifier Systems (LCS)** more than 30 years ago as an extension to evolutionary computing [11:10].

Learning classifier systems are a kind of rule-based system with general mechanisms for processing rules in parallel, for the adaptive generation of new rules, and for testing the effectiveness of new rules.

However, the concept started to get the attention of computer scientists only a few years ago, with the introduction of several variants of the original concept, including **Extended Learning Classifiers (XCS)**. Learning classifier systems are interesting because they combine rules, reinforcement learning, and genetic algorithms.

Note

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 (LCS) merge the concepts of reinforcement learning, rule-based policies, and evolutionary computing. This unique class of learning algorithms represents the merging of the following research fields [11:11]:

- Reinforcement learning
- Genetic algorithms and evolutionary computing
- Supervised learning
- Rule-based knowledge encoding

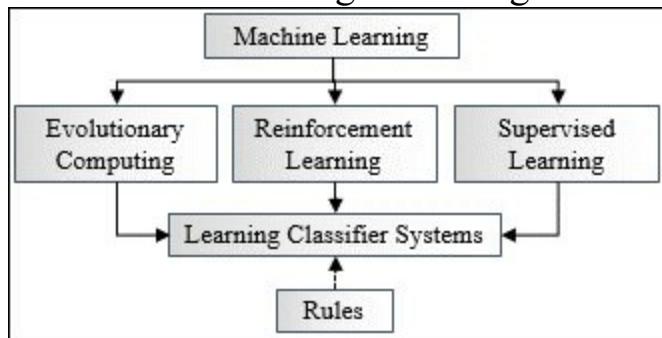


Diagram of the scientific disciplines required for learning classifier systems

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:** This evolves over time. In some cases, a domain expert creates a primitive set of rules (core knowledge). In other cases, the rules are randomly generated prior to the execution of the learning classifier system.
- **A genetic algorithm-based discovery engine:** This generates new classifiers or rules from the existing population. This component is also known as the **rules discovery module**. The rules rely on the same pattern of evolution of organisms introduced in the previous chapter. The rules are encoded as strings or bit strings to represent a condition (predicate) and action.

- **A performance or evaluation function:** This measures the positive or negative impact of the actions from the fittest classifiers or policies.
- **A reinforcement learning component:** This rewards or punishes the classifiers that contribute to the action, as seen in the previous section. 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.

Combining learning and evolution

Learning classifier systems are particularly appropriate to problems in which the environment is constantly changing, and are a combination of a learning strategy and an evolutionary approach to building and maintaining 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 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 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 the 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:

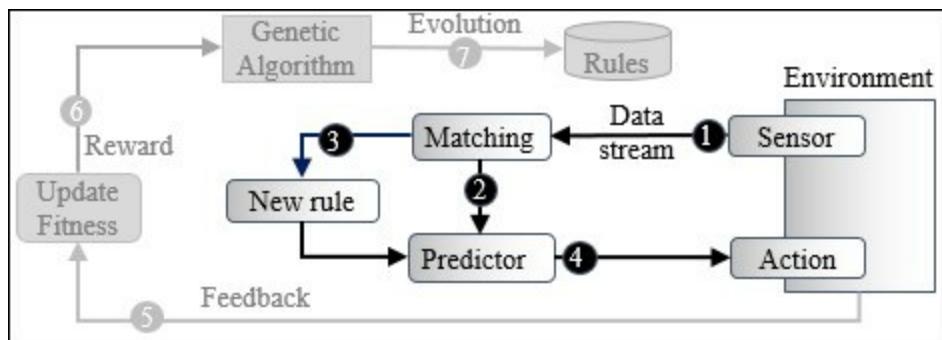
- **Environment:** This is the environment variable in the context of reinforcement learning.
- **Agent:** An agent used in reinforcement learning.
- **Predicate:** A clause or fact using the format: variable- operator- value, and usually implemented as (operator, variable value); for example, Temperature- exceeds - 87F or (Temperature, 87F), Hard drive – failed or (Status hard drive, FAILED), and so on. It is encoded as a gene 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 (predicate 1, predicate 2, 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 a sequence of actions; for example, IF gold price < \$1140 THEN sell stock of oil and gas producing companies.
- **Classifier:** This is a rule in the context of an LCS.
- **Rule fitness or score:** This is identical to the definition of fitness or score in the genetic algorithm. In the context of an LCS, it is the probability of a rule being invoked and fired in response to a change in the environment.
- **Sensors:** Environment variables monitored by an 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:** This is the process of creating new rules to match a new condition (sensor) in the environment. It generates rules by either using a random generator or mutating existing rules.
- **Predictor:** This is an algorithm to find the action with the maximum number of occurrences within a set of matching rules.

Extended learning classifier systems

Similar to reinforcement learning, the XCS algorithm has an *exploration* phase and an *exploitation* phase. The exploitation process consists of leveraging the existing rules to influence the target environment in a profitable or rewarding manner:

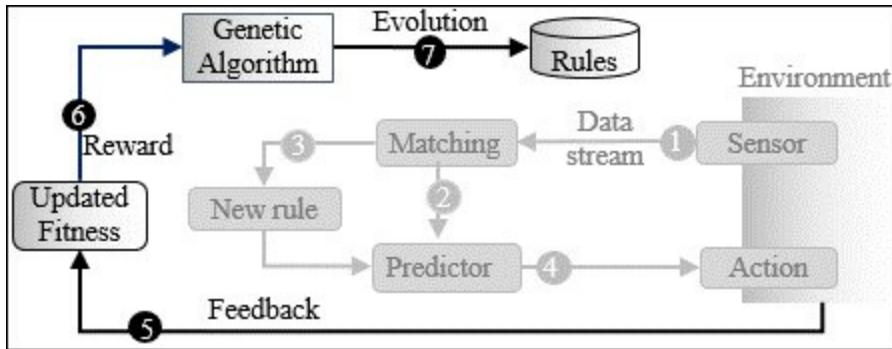


Exploitation component of the XCS algorithm

The following list describes each numbered block:

1. Sensors acquire new data or events from the system.
2. Rules for which the condition matches the input event are extracted from the current population.
3. A new rule is created if no match is found in the existing population. This process is known as covering.
4. The chosen rules are ranked by their fitness values, and the rules with the highest predicted outcome are used to trigger the action.

The purpose of exploration components is to increase the rule base as a population of the chromosomes that encode these rules:



Exploration components of the XCS algorithm

The following list describes each numbered block of the block diagram:

1. Once the action is performed, the system rewards the rules for which the action has been executed. The reinforcement learning module assigns credit to these rules.
2. Rewards are used to update the rule fitness, applying evolutionary constraints to the existing population.
3. The genetic algorithm updates the existing population of classifiers/rules using operators such as crossover and mutation.

XCS components

This section describes the key classes of the XCS. The implementation leverages the existing design of the genetic algorithm and 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 (ETF)** in an ever-changing financial environment. Unlike 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 should be constantly adjusted in response to any specific change in the environment or market condition that affects the total value of the portfolio, as shown in 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	Stock feed 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 can be defined by a domain expert.

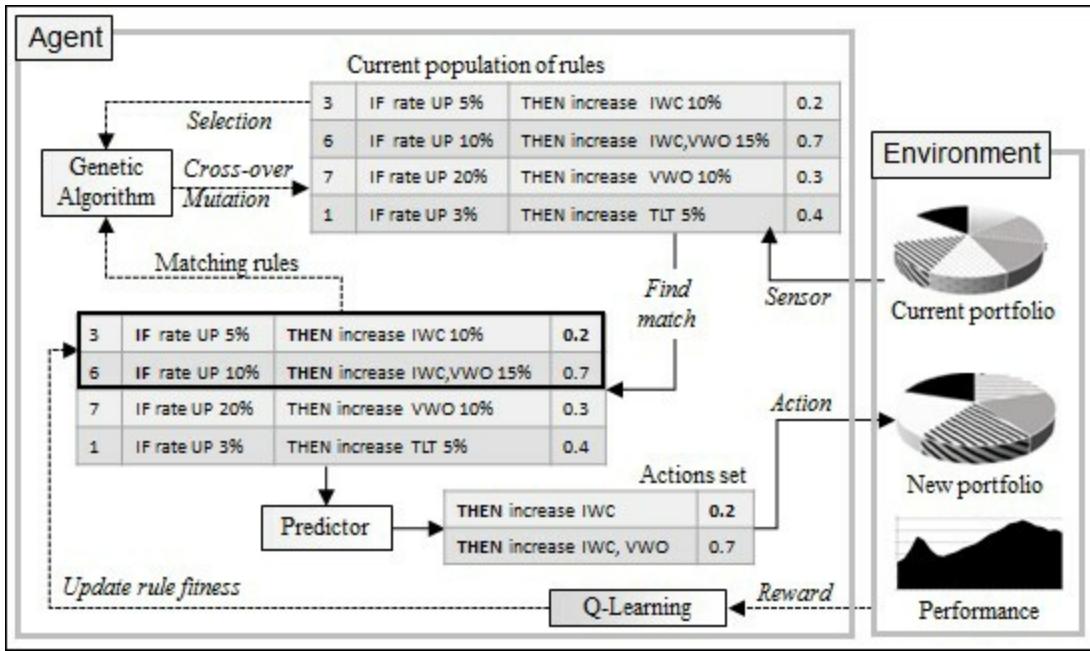
Tip

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 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 is used to evolve the population of rules:



Overview of the XCS algorithm to optimize the portfolio allocation

The agent responds to the change in the allocation of ETFs in the portfolio by matching one of the existing rules.

Let's build the XCS agent from the ground up.

XCS core data

There are three types of data that are manipulated by the XCS agent:

- **Signal:** This is the trading signal.
- **XcsAction:** This is the action on the environment. It subclasses a **Gene** defined in the genetic algorithm.
- **XcsSensor:** This is the sensor or data from the environment.

The **Gene** class was introduced for the evaluation of the genetic algorithm in the *Trading signals* section in [Chapter 13, Evolutionary Computing](#). The agent creates, modifies, and deletes actions. It makes sense to define these actions as mutable genes, as follows:

```
class XcsAction(sensorId: String, target: Double)
  (implicit quantize: Quantization, encoding: Encoding) //1
```

```
extends Gene(sensorId, target, EQUAL)
```

The quantization and encoding of the `XCSAction` into a `Gene` has to be explicitly declared (line 1). The `XcsAction` class has the identifier of the 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 defined as follows:

```
val vwoTo80 = new XcsAction("VWO", 80.0)
```

The only type of action allowed in this scheme is setting a value using the `EQUAL` operator. You can create actions that support other operators, such as `+=`, which is used to increase an existing value. These operators need to implement the operator trait, explained in the *Trading operators* section in [Chapter 13, Evolutionary Computing](#).

Finally, the `XcsSensor` class encapsulates the `sensorId` identifier for the variable and `value` of the sensor, as shown here:

```
case class XcsSensor(sensorId: String, value: Double)  
val new10ytb = XcsSensor("10yTBYield", 2.76)
```

Tip

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 a variable/sensor ID and values as arguments.

XCS rules

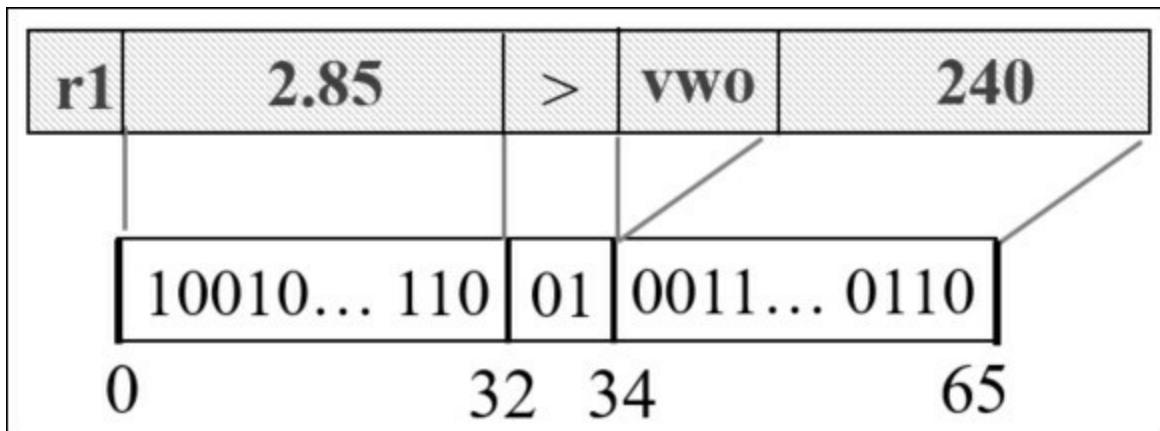
The next step consists of defining a rule, of type `XcsRule`, as a pair of two genes: a signal and an action, as shown in the following code:

```
Case class XcsRule(signal: Signal, 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("10ybt", 2.84, GREATER_THAN)
val action = new XcsAction("vwo", 240)
val r1 = XcsRule(signal, action)
```

The agent encodes the rule as a chromosome using two bits to represent the operator and 32 bits for values, as shown in the following diagram:



XCS rule encoding scheme

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.

Tip

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 `new10ybt` sensor (line 2) against the signal in the current population of `s10ybt1` (line 3) and `s10ybt2` (line 4) rules that use

the same sensor or variable `10ytb`, as follows:

```
val new10ytb = new XcsSensor("10ytb", 2.76) //2
val s10ytb1 = Signal("10ytb", 2.5, GREATER_THAN) //3
val s10ytb2 = Signal("10ytb", 2.2, LESS_THAN) //4
```

In this case, the agent selects the 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) //5
val r11: XcsRule(s10yTB6, act6)
val r46: XcsRule(s10yTB7, act12) //6
```

The action with the most references, `act12` (lines 5, 6), is executed. The Q-learning algorithm computes the reward from the profit or loss incurred by the portfolio following the execution of the selected 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 rules in the population, until either a new genetic rule, modified through crossover, and mutation or a rule, created through covering, triggers a more rewarding action on the environment.

Covering

The purpose of the covering phase is to generate new rules if no rule matches the input or sensor. The `cover` method of an `XcsCover` singleton generates a new `XcsRule` instance given a sensor and an existing set of actions, as shown here:

```
val MAX_NUM_ACTIONS = 2048
def cover(
    sensor: XcsSensor,
    actions: List[XcsAction])
  (implicit quant: Quantization, encoding: Encoding): List[XcsRule]

  actions./:(List[XcsRule]()) ((xs, act) => {
    val signal = Signal(sensor.id, sensor.value,
      new SOperator(nextInt(Signal.numOperators)))
    new XcsRule(signal, XcsAction(act, Random)) :: xs
  })
```

```
    })  
}
```

You might wonder why the `cover` method uses a set of actions as arguments knowing that covering consists of creating new actions. The method mutates (operator `^`) an existing action to create a new one instead of using a random generator. This is one of the advantages of defining an action as a gene. One of the constructors of `XcsAction` executes the mutation, as follows:

```
def apply(action: XcsAction, r: Random): XcsAction =  
  (action ^ r.nextInt(XCSACTION_SIZE))
```

The index of the operator type, `r`, 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 of the rules
- `Xcs`: These are the rules for the matching, covering, and generation of actions

The extended learning classifier is a data transformation of type `ETransform` with explicit configuration of type `XcsConfig` (line 8) (refer to the *Monadic data transformation* section in [Chapter 2, Data Pipelines](#)).

The following code snippet is a skeleton implementation of the extended learning classifier class `Xcs`:

```
class Xcs(config: XcsConfig,  
  population: Population[Signal],  
  score: Chromosome[Signal]=> Unit,  
  input: Array[QLInput]) //7  
extends ETransform[XcsConfig, XcsConfig](config) { //8  
  
  type U = XcsSensor //9  
  type V = List[XcsAction] //10
```

```

    val solver = GASolver[Signal](config.gaConfig, score)
    val features = population.chromosomes.toSeq
    val qLearner = QLearning[Chromosome[Signal]]( //11
        config.qlConfig, extractGoals(input), input, features
    )

    override def |> : PartialFunction[U, Try[V]]
    ...
}

```

The XCS algorithm is initialized with a configuration, `config`; an initial set of rules, `population`; a fitness function, `score`; and an `input` to the Q-learning policy to generate a reward matrix for `qlLearner` (line 7). Being an explicit data transformation, the type `U` of the input element and the type `V` of the output element to the predictor `|>` are initialized as `XcsSensor` (line 9) and `List[XcsAction]` (line 10).

The goals and number of states are extracted from the input to the policy of the Q-learning algorithm.

In this implementation, the generic algorithm, `solver`, is mutable. It is instantiated along with the `Xcs` container class. The Q-learning algorithm uses the same design, as any classifier, as immutable. The model of Q-learning is the best possible policy to reward rules. Any changes in the number of states or the rewarding scheme require a new instance of the learner.

Benefits and limitations of learning classifier systems

Learning classifier systems and XCS provide many benefits, 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 the exploration and exploitation of the knowledge base

However, the scientific community has been slow to recognize the merits of

these techniques. The wider adoption of learning classifier systems is hindered by the following factors:

- The large number of parameters used in both the exploration and exploitation phases adds to the sheer complexity of the algorithm.
- There are too many competitive variants of learning classifier systems
- There is no clear unified theory to validate the concept of evolutionary policies or rules. After all, these algorithms are a combination of standalone techniques. The accuracy and performance of the execution of many variants of learning classifier systems depend on each component as well as the interaction between components.
- Their execution is not always predictable in terms of scalability and performance.

Summary

This concludes our foray into the world of reinforcement learning algorithms. We hope that this chapter provides adequate answers to those looking to understand the Q-learning algorithm, how to implement it in Scala, and how to apply it to leverage financial instruments. The chapter concludes with an overview of learning classifier systems.

This chapter completes our overview of some of the most frequently applied machine learning methods. It is important to acknowledge that the book does not pretend to cover all the types of machine learning algorithm such as k-nearest neighbors, decision trees, or random forests.

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 last part of the book: scalability.

Chapter 16. Parallelism in Scala and Akka

Data analysts, scientists, and software engineers have been facing a serious challenge: the explosion of the amount of data required to build reliable models. After all, how valuable is a data mining application if the model does not scale?

The challenge of big data is addressed through a two-facet strategy: improving the efficiency of existing data mining and machine learning solutions, and leveraging scalable infrastructure (frameworks, programming languages, GPUs, and so on).

This chapter covers the Scala parallel collections, the Actor model, and the Akka framework. The next chapter introduces the Apache Spark framework and its collection of machine learning algorithms.

The following are the topics addressed in this chapter:

- Introduction to Scala parallel collections
- Evaluation of the performance of a parallel collection on a multicore CPU
- The Actor model and reactive systems
- Clustered and reliable distributed computing using Akka
- Design of computational workflow using Akka routers

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.

The following stack representation illustrates the interdependencies between frameworks:

Spark

Partitioner, Accumulator: *org.apache.spark.broadcast*
Resilient datasets: *org.apache.spark.rdd*.
Data frame: *org.apache.spark.sql*.
Caching, Shuffling: *org.apache.spark*.
Listeners: *org.apache.spark.scheduler*.
Serialization: *org.apache.spark.serializer*

Akka

Actors, Supervisors: *akka.actors*.
Remote actors: *akka.remote*
Type actors: *akka.actors*.
Mailbox management: *akka.mailbox*.
Clusters: *akka.cluster*.
Dispatchers: *akka.dispatch*
Events management: *akka.event*.
Routing, Broadcast: *akka.routing*
Persistency: *akka.persistence*.

Scala

Scheduler: *scala.actors.scheduler*
Concurrency: *scala.concurrent*
Par. collections: *scala.collection.parallel*

JVM

Threads, executors: *java.util.concurrent.**

Stack representation of Scalable frameworks using Scala

The next chapter introduces the Apache Spark framework.

Each layer adds a new functionality to the previous one to increase

scalability. The **Java Virtual Machine (JVM)** runs as a process within a single host. Scala concurrent classes support effective deployment of an application by leveraging multi core CPU capabilities without the need to write multithreaded applications. Akka extends the Actor paradigm to clusters with advanced messaging and routing options. Finally, Apache Spark leverages Scala higher-order collection methods and the Akka implementation of the Actor model to provide large-scale data processing systems with better performance and reliability, through its resilient distributed datasets and in-memory persistency.

Scala

The Scala standard library offers a rich set of tools, such as parallel collections and concurrent classes to scale number-crunching applications. Although these tools are very effective in processing medium-sized datasets, they are unfortunately quite often discarded by developers in favor of more elaborate frameworks.

Object creation

Although code optimization and memory management is beyond the scope of this chapter, it is worthwhile to remember that a few simple steps can be taken to improve the scalability of an application. One of the most frustrating challenges in using Scala to process large datasets is the creation of a large number of objects and the load on the garbage collector.

A partial list of remedial actions is as follows:

- Limiting unnecessary duplication of objects in an iterated function 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

Streams

Some problems require the pre processing and training of very large datasets, resulting in significant memory consumption by the JVM. Streams are list-like collections in which elements are instantiated or computed lazily. Streams share the same goal of postponing computation and memory allocation as views.

Memory on demand

One very important application of Streams for scientific computing and machine learning is the implementation of iterative or recursive computation for very large or infinite data Streams.

Let's consider the simple computation of the mean of a very large dataset. The following is the formula for it:

$$\mu_n = \frac{1}{n} \sum_{i=0}^{n-1} x_i$$

This formula has two major problems:

- It requires us to allocate memory for the entire dataset x_i
- The sum of all elements may overflow if the dataset is very large

A simple solution is to iterate the value of the mean for each new element of the dataset as follows:

$$\mu_n = \left(1 - \frac{1}{n}\right) \mu_{n-1} + \frac{x_n}{n}$$

As you can see in the following code, the iterative formula for the computation of the mean of a very large or infinite dataset can be

implemented using Streams and tail recursion (*line 1*). At each iteration (or recursion), the new mean is updated from the existing mean (*line 2*), the count is incremented (*line 3*), and the next value is accessed through the tail of the Stream (*line 4*):

```
def mean(strm: => Stream[Double]): Double = {  
    @scala.annotation.tailrec //1  
    def mean(z: Double,  
             count: Int,  
             strm: Stream[Double]): (Double, Int) =  
  
        if(strm.isEmpty) (z, count)  
        else  
            mean((1.0 - 1.0/count)*z + strm.head/count, //2  
                  count+1, //3  
                  strm.tail) //4  
  
    mean(0.0, 1, strm)._1  
}
```

Tip

Stream as by-name parameter

Passing the Stream by value holds the Stream tail. One easy solution is to pass the Stream by name, as a function. In this case, the mean is computed before it is passed to the method.

Design for reusing Streams memory

Let's consider the computation of the loss function in machine learning. An observation of type `DataPoint` is defined as a features vector x and a label or expected value y :

```
case class DataPoint(x: DblVec, y: Double)
```

We can create a loss function, `LossFunction` that processes a very large dataset on a platform with limited memory. The optimizer responsible for the minimization of the loss or error invoked the loss function at each iteration or

recursion, as described in the following diagram:

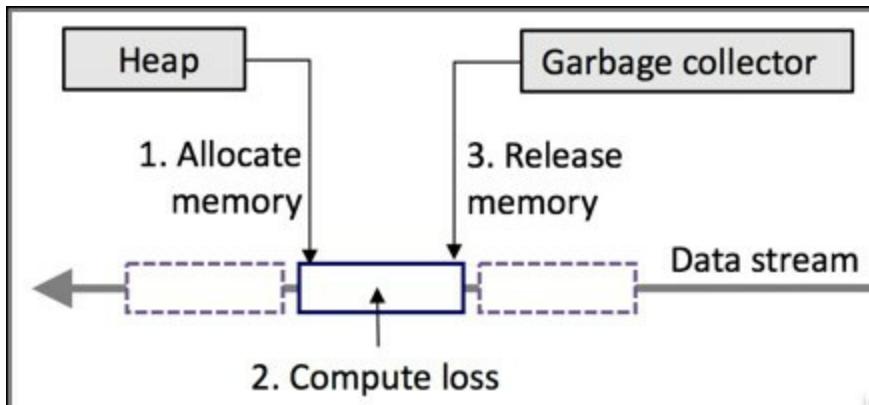


Illustration of lazy memory management with Scala Streams

The memory management for the Stream consists of the following steps:

1. Allocate a memory block to load a slice of the Stream for which the loss function is compute.
2. Apply the loss function to the slice of the Stream.
3. Release the memory block associated to the slice once the loss function is complete.
4. Repeat step 1 to 3 for the next slice on the Stream.

The challenge is to make sure the memory allocated for a slice of the Stream is actually released once it is no longer needed (the computation of the loss for the observations contained in the slice is completed). This is accomplished by allocating each slice with a weak reference.

Note

Spark Streaming

The architecture of Spark Streaming described in the *Spark Streaming* section of [Chapter 17, Apache Spark MLlib](#) used a similar design principle as our reusable Stream memory.

Let's implement our design. The constructor of the `LossFunction` class has three arguments (line 6 of the code shown as follows):

- The computation f of the loss for each data point
- The weights of the model
- The size of the entire Stream, `dataSize`

Here is the code:

```
type StreamLike = WeakReference[Stream[DataPoint]] //5
type DblVec = Vector[Double]

class LossFunction(
    f: (DblVec, DblVec) => Double,
    weights: DblVec,
    dataSize: Int) { //6

    var nElements = 0
    def compute(stream: () => StreamLike): Double =
        compute(stream().get, 0.0) //7

    def sqrLoss(xs: List[DataPoint]): Double = xs.map(dp => {
        val z = dp.y - f(weights, dp.x)
        z * z
    }).reduce(_ + _) //8
}
```

The `LossFunction` for the Stream is implemented as the tail recursion, `compute` (line 7). The recursive method updates the reference of the Stream. The type of reference of the Stream is `WeakReference` (*line 5*), so the garbage collection can reclaim the memory associated with the slice for which the loss has been computed. In this example, the loss function is computed as a sum of square error (*line 8*).

The `compute` method manages the allocation and release of slices of the Stream:

```
@tailrec
def compute(strm: Stream[DataPoint], loss: Double): Double ={
    if( nElements >= dataSize) loss
    else {
        val step =
            if(nElements + STEP > dataSize) dataSize - nElements
            else STEP
        nElements += step

        val newLoss = sqrLoss(strm.take(step).toList) //9
    }
}
```

```
    compute(strm.drop(STEP), loss + newLoss) //10
}
}
```

The dataset is processed in two steps:

- The driver allocates (that is, `take`) a slice of the Stream of observations and then computes the cumulative loss for all the observations in the slice (line 9)
- Once the computation of the loss for the slice is completed, the memory allocated to the weak reference is released (that is `drop`) (line 10)

Tip

Alternative to weak references

There are alternatives to weak references to the Stream for forcing the garbage collector to reclaim the memory blocks associated with each slice of observations:

- Define the stream reference as *def*
- Wrap the reference into a method: the reference is then accessible to the garbage collector when the wrapping method returns
- Use a List Iterator.

The average memory allocated during the execution of the `LossFunction` for the entire Stream is the memory needed to allocate a single slice.

Parallel collections

The Scala standard library includes parallelized collections, whose purpose is to shield developers from the intricacies of concurrent thread execution and race condition. Parallel collections are a very convenient approach to encapsulate concurrency constructs to a higher level of abstraction [16:1].

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 lends 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*).

Tip

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

The main purpose of parallel collections is to improve the performance of execution through concurrency. The first step is to either select an existing benchmark or create our own.

Tip

Scala library benchmark

The Scala standard library has a trait, `testing.Benchmark`, for testing using the command line [16:2]. All you need to do is insert your function or code in the `run` method:

```
object test with Benchmark { def run { /* ... */ }}
```

Let us create a parameterized class, `Parallelism`, to evaluate the performance of operations on parallel collections:

```
abstract class Parallelism[U](times: Int) {
    def map(f: U => U)(nTasks: Int): Double //1
    def filter(f: U => Boolean)(nTasks: Int): Double //2
    def timing(g: Int => Unit): Long
}
```

The user has to supply the data transformation `f` for the `map` (*line 1*) and `filter` (*line 2*) operations of parallel collection as shown in the preceding code as well as the number of concurrent tasks `nTasks`. The `timing` method collects the duration of the `times` execution of a given operation `g` on a parallel collection:

```
def timing(g: Int => Unit): Long = {
    var startTime = System.currentTimeMillis
    Range(0, times).foreach(g)
    System.currentTimeMillis - startTime
}
```

Let's define the mapping and reducing operation for the parallel arrays for which the benchmark is defined as follows:

```
class ParallelArray[U] (
    u: Array[U], //3
    v: ParArray[U], //4
    times: Int) extends Parallelism[T](times)
```

The first argument of the benchmark constructor is the default array of the Scala standard library (*line 3*). The second argument is the parallel data structure (or class) associated to the array (*line 4*).

Let's compare the parallelized and default array on the `map` and `reduce` methods of `ParallelArray` as follows:

```
def map(f: U => U)(nTasks: Int): Unit = {
    val pool = new ForkJoinPool(nTasks)
    v.tasksupport = new ForkJoinTaskSupport(pool)

    val duration = timing(_ => u.map(f)).toDouble //5
```

```

    val ratio = timing(_ => v.map(f))/duration //6
    show(s"$numTasks, $ratio")
}

```

The user has to define the mapping function, `f`, and the number of concurrent tasks, `nTasks`, available to execute a `map` transformation on the array `u` (line 5) and its parallelized counterpart `v` (line 6). The `reduce` method follows the same design as shown in the following code:

```

def reduce(f: (U,U) => U)(nTasks: Int): Unit = {
    val pool = new ForkJoinPool(nTasks)
    v.tasksupport = new ForkJoinTaskSupport(pool)

    val duration = timing(_ => u.reduceLeft(f)).toDouble //7
    val ratio = timing(_ => v.reduceLeft(f) )/duration //8
    show(s"$numTasks, $ratio")
}

```

The user-defined function `f` is used to execute the `reduce` action on the array `u` (line 7) and its parallelized counterpart `v` (line 8).

The same template can be used for other higher Scala methods, such as `filter`.

The absolute timing of each operation is completely dependent on the environment. It is far more useful to record the ratio of the duration of execution of operation on the parallelized array, over the single thread array.

The benchmark class, `ParallelMap`, used to evaluate `ParHashMap` is similar to the benchmark for `ParallelArray`, as shown in the following code:

```

class ParallelMap[U] (
    u: Map[Int, U],
    v: ParMap[Int, U],
    times: Int) extends ParBenchmark[T](times)

```

For example, the `filter` method of `ParMapBenchmark` evaluates the performance of the parallel map `v` relative to single threaded map `u`. It applies the filtering condition to the values of each map as follows:

```

def filter(f: U => Boolean)(nTasks: Int): Unit = {

```

```

val pool = new ForkJoinPool(nTasks)
v.tasksupport = new ForkJoinTaskSupport(pool)

val duration = timing(_ => u.filter(e => f(e._2))).toDouble
val ratio = timing(_ => v.filter(e => f(e._2)))/duration
show(s"$nTasks, $ratio")
}

```

Performance evaluation

The first performance test consists of creating a single-threaded and a parallel array of random values and executing the evaluation methods, `map` and `reduce`, on using an increasing number of tasks, as follows:

```

val sz = 1000000; val NTASKS = 16
val data = Array.fill(sz)(nextDouble)
val pData = ParArray.fill(sz)(nextDouble)
val times: Int = 50

val bench = new ParallelArray[Double](data, pData, times)
val mapper = (x: Double) => sin(x*0.01) + exp(-x)
Range(1, NTASKS).foreach(bench.map(mapper)(_))
val reducer = (x: Double, y: Double) => x+y
Range(1, NTASKS).foreach(bench.reduce(reducer)(_))

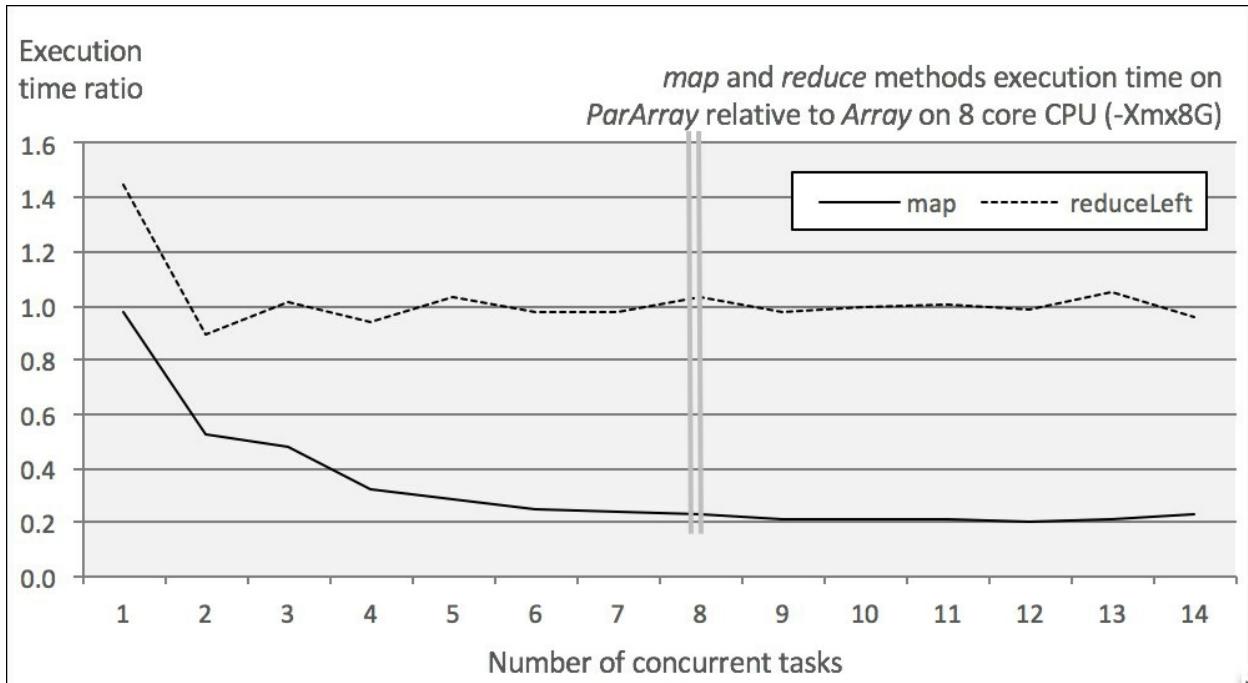
```

Tip

Measuring performance

The code has to be executed within a loop and the duration has to be averaged over a large number of executions to avoid transient actions such as initialization of the JVM process or collection of unused memory (GC).

The following graph shows the output of the performance test:



Impact of concurrent tasks on the performance on Scala parallelized map and reduce

The test executes the mapper and reducer functions 1 million times on an 8-core CPU with 8 GB of available memory on 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 = 10000000
val mData = new HashMap[Int, Double]
Range(0, sz).foreach( mData.put(_, nextDouble()) ) //9
val mParData = new ParHashMap[Int, Double]
```

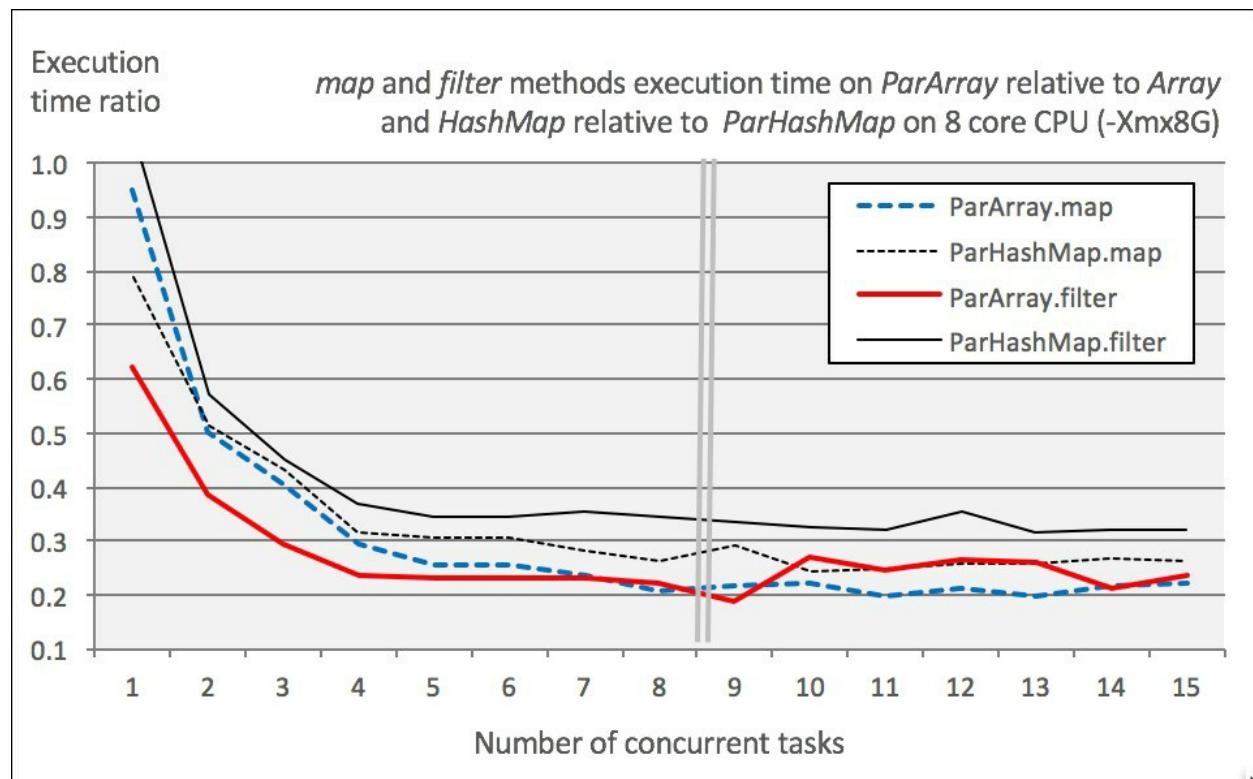
```

Range(0, sz).foreach( mParData.put(_, nextDouble))

val bench = new ParallelMap[Double](mData, mParData, times)
Range(1, NTASKS).foreach(bench.map(mapper) (_)) //10
val filterer = (x: Double) => (x > 0.8)
Range(1, NTASKS).foreach( bench.filter(filterer) (_)) //11

```

The test initializes a `HashMap` instance and its parallel counter `ParHashMap` with 1 million random values (*line 9*). The benchmark, `bench`, processes all the elements of these hash maps with the `mapper` instance introduced in the first test (*line 10*) and a filtering function, `filterer` (*line 11*), with `NTASKS = 6`. The output is as shown here:



Impact of concurrent tasks on the performance on Scala parallelized array and hash map

The impact of the parallelization of collections is very similar across methods and across collections. The performance of all 4 parallel collections increases 3 to 5 fold as the number of concurrent tasks (threads) increases. It is interesting to notice that the performance of these 4 collections stabilizes for tests with more than 4 tasks. *Core parking* is partially responsible for this

behavior. Core parking disables a few CPU cores in an effort to conserve power, and in the case of a single application, consumes almost all CPU cycles.

Tip

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

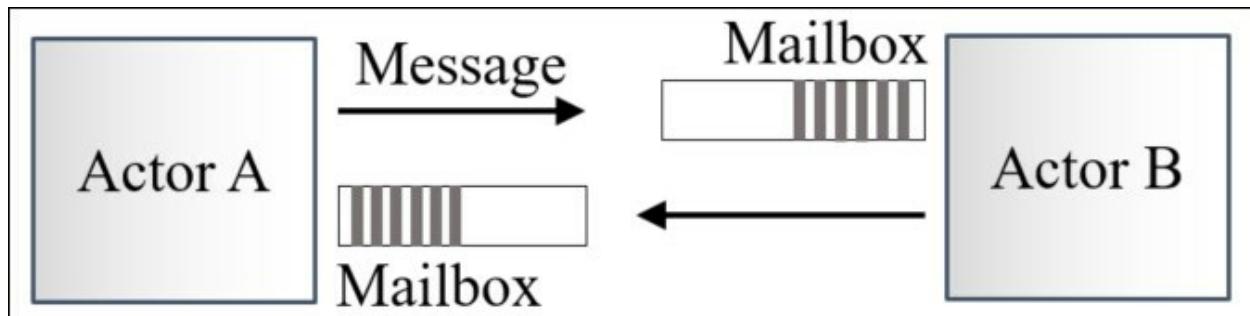
The Actor model

The Actor model, originally introduced in the **Erlang** programming language, addresses these issues [16:3]. The purpose of using the Actor model is two-fold:

- 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 are 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]` [16:4]. 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[16:5].

Note

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. You must be mindful that blocking the underlying threads might starve other Actors from CPU cycles. It is recommended you either configure the runtime system to use a large thread pool, or you 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 sub-datasets.
2. Processing each dataset independently and concurrently.
3. Aggregating all the resulting datasets.

These steps are defined through a monad associated with a collection in the *Abstraction* section under *Why Scala?* in [Chapter 1, Getting started](#):

1. The `apply` method creates the sub-collection or partitions for the first step, for example, `def apply[T](a: T): List[T]`.
2. A map-like operation defines the second stage. The last step relies on the monoidal associativity of the Scala collection, for example, `def ++(a: List[T], b: List[T]): List[T] = a ++ b`.
3. The aggregation, such as `reduce`, `fold`, `sum`, and so on, consists of flattening all the sub-results 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 [16:6].

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 a typed Actor, message dispatching, routing, load balancing, and partitioning, as well as supervision and configurability [16:7].

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

Note

Typed and untyped Actors

Untyped Actors can process messages of any type. If the type of the message is not matched by the receiving Actor, it is discarded. Untyped Actors can be regarded as contract-less Actors. They are the default Actors in Scala.

Typed Actors are similar to Java remote interfaces. They respond to a method invocation. The invocation is declared publicly, but the execution is

delegated asynchronously to the private instance of the target Actor [16:8].

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 an explicit or implicit monadic data transformation as described in the *Monadic data transformation* section of [Chapter 2, Data Pipelines](#).

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 [16:9]. 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.

Tip

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:

```
sealed abstract class Message(val i: Int)
case class Start(i: Int = 0) extends Message(i)    //1
case class Activate(i: Int, x: DblVec) extends Message(i) //2
case class Completed(i: Int, x: DblVec) extends Message(i)//3
```

Let's define the messages that control the execution of the algorithm. We need at least the following message types or case classes:

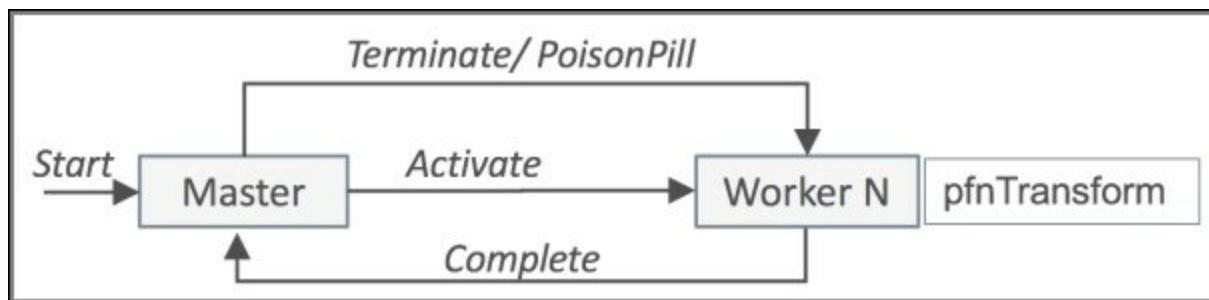
- `Start` is sent by the client code to the master to start the computation (line 1).
- `Activate` is sent by the master to the workers to activate the

computation. This message contains the time series, x , to be processed by the worker Actors. It also contains the reference to `sender` (master Actor) (line 2).

- `Completed` is sent by each worker back to `sender`. It contains the variance of the data in the group (line 3).

The master stops a worker using a `PoisonPill` message. The different approaches to terminate an Actor are described in the *The master Actor* section.

The hierarchy of the `Message` class is sealed to prevent third-party developers from adding another `Message` type. The worker responds to the `activate` message by executing a data transformation of type `ITransform`. The messages exchanged between master and worker Actors are shown in the following diagram:



Sketch design of the master-slave communication in an Actor framework

Note

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 partitioned datasets created by the master Actor, as follows:

```
type DblVec = Vector[Double]
type PfnTransform = PartialFunction[DblVec, Try[DblVec]]

class Worker(id: Int, fct: PfnTransform) extends Actor { //1

    override def receive = {
        case msg: Activate => //2
            sender ! Completed(msg.id+id, fct(msg.xt).get)
    }
}
```

The `Worker` class constructor takes `fct`, the partial function, 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, `fct` is completed.

The workflow controller

In the *Scalability* section in [Chapter 1, Getting Started](#), we introduced the concepts of workflow and controller, to manage the training and classification process as a sequence of the 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`, implemented as a partial function
- The number of partitions, `nPartitions`, to break down a time series for concurrent processing

The `Controller` class can be defined as follows:

```
abstract class Controller (
    val xt: DblVec,
    val fct: PfnTransform,
    val nPartitions: Int)
extends Actor with Monitor[Double] { //3

    def partition: Iterator[DblVec] = { //4
```

```

        val sz = (xt.size.toDouble/nPartitions).ceil.toInt
        xt.grouped(sz)
    }
}

```

The controller is responsible for splitting the time series into several partitions and assigning each partition to a dedicated worker (line 4).

The master Actor

Let's define a master Actor class, `Master`. The three methods to override are as follows:

- `preStart` is a method invoked to initialize all the necessary resources such as file or database connection before the Actor executes (line 9)
- `receive` is a partial function that de-queues and processes the messages from the mailbox
- `postStop` cleans up resources such as releasing memory and closing database connections, sockets, or file handles (line 10)

The `Master` class can be defined as follows:

```

abstract class Master( //5
  xt: DblVec,
  fct: PfnTransform,
  nPartitions: Int)
extends Controller(xt, fct, nPartitions) {

  val aggregator = new Aggregator(nPartitions) //6
  val workers = List.tabulate(nPartitions)(n =>
    context.actorOf(Props(new Worker(n, fct)),
      name = s"worker_$n")
  ) //7
  workers.foreach( context.watch( _ ) ) //8

  override def preStart: Unit = /* ... */ //9
  override def postStop: Unit = /* ... */ //10
  override def receive
}

```

The `Master` class has the following parameters (line 5):

- `xt`: This is the time series to transform
- `fct`: This is the transformation function
- `nPartitions`: This is the number of partitions

An aggregating class `Aggregator`, collects, and computes (reducer) the results from each worker (line 6). The aggregator manages the state of the computation of the worker nodes with the methods `+=` (adding the results from a worker), `clear` (resetting the state), and `completed` (testing whether every worker returns their computed values):

```
class Aggregator(partitions: Int) {
    val state = ListBuffer[DblVec]()

    def +=(x: DblVec): Boolean = {
        state.append(x)
        state.size == partitions
    }
    def clear: Unit = state.clear
    def completed: Boolean = state.size == partitions
}
```

The worker Actors are created through the `actorOf` factory method of the `ActorSystem` context (line 7). The worker Actors are attached to the context of the master Actor so it can be notified when the workers terminate (line 8).

The `receive` message handler processes only two types of messages: `Start` from the client code and `Completed` from the workers, as shown in the following code:

```
override def receive = {
    case s: Start => start //11
    case msg: Completed => //12
        if( aggregator += msg.xt) //13
            workers.foreach( context.stop(_) ) //14
    case Terminated(sender) => //15
        if( aggregator.completed ) {
            context.stop(self) //16
            context.system.shutdown
        }
}
```

The `Start` message triggers the partitioning of the input time series into

partitions (line 11):

```
def start: Unit = workers.zip(partition.toVector)
    .foreach {case (w, s) => w ! Activate(0, s)} //16
```

The partitions are then dispatched to each worker with the `Activate` message (line 16).

Each `worker` sends a `Completed` message back to the master upon the completion of their task (line 12). The master aggregates the results from each worker (line 13). Once all the workers have completed their task, they are removed from the master's context (line 14). The master terminates all the workers, through a `Terminated` message (line 15), and finally terminates itself through a request to its `context` to stop it (line 16).

The previous code snippet uses two different approaches to terminate an Actor. There are four different methods of shutting down an Actor, as mentioned here:

- `actorSystem.shutdown`: This method is used by the client to shut down the parent Actor system
- `actor ! PoisonPill`: This method is used by the client to send a poison pill message to the Actor
- `context.stop(self)`: This method is used by the Actor to shut itself down within its context
- `context.stop(childActorRef)`: This method is used by the Actor to shut itself down through its reference

Master with routing

The previous design makes sense only if each worker has a unique characteristic that requires direct communication with the master. This is not the case in most applications. The communication and internal management of the worker can be delegated to a router. The implementation of the master routing capabilities is very similar to the previous design, as shown in the following code:

```
class MasterWithRouter (
```

```

xt: DblVec,
fct: PfnTransform,
nPartitions: Int) extends Controller(xt, fct, nPartitions) {

    val aggregator = new Aggregator(nPartitions)
    val router = { //17
        val routerConfig = RoundRobinRouter(nPartitions, //18
            supervisorStrategy = this.supervisorStrategy)
        context.actorOf(
            Props(new Worker(0, fct)).withRouter(routerConfig)
        )
    }
    context.watch(router)
    override def receive { ... }
}

```

The only difference is that the `context.actorOf` factory creates an extra Actor, `router`, along with the workers (line 17). This particular implementation relies on round-robin assignment of the message by the router to each worker (line 18). Akka supports several routing mechanisms that select a random Actor, or the Actor with the smallest box, or the first to respond to a broadcast, and so on.

Note

Router supervision

The router Actor is a parent of the worker Actors. It is by design a supervisor of the worker Actors, which are its children Actors. Therefore, the router is responsible for the life cycle of the worker Actors, which includes their creation, restarting, and termination.

The implementation of the `receive` message handler is almost identical to the message handler in the master without routing capabilities, except the termination of the workers through the router (line 19):

```

override def receive = {
    case Start => start
    case msg: Completed =>
        if( aggregator += msg.xt) context.stop(router) //19
}

```

The message handler, `start` has to be modified to broadcast the `Activate` message to all the workers through the `router` using this code:

```
def start: Unit =  
    partition.toVector.foreach {router ! Activate(0, _)}
```

Distributed discrete Fourier transform

Let's select the **discrete Fourier transform (DFT)** on a time series, `xt`, as our data transformation. We discussed it in the *Discrete Fourier transform* section in [Chapter 3, Data Pre-processing](#). 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 DFT, as follows:

```
type Reducer = List[DblVec] => immutable.Seq[Double]  
class DFTMaster(  
    xt: DblVec,  
    nPartitions: Int,  
    reducer: Reducer) //20  
extends Master(xt, DFT[Double].|>, nPartitions)
```

The `reducer` method aggregates or reduces the results of the DFT (frequencies distribution) from each worker (line 20). In the case of the DFT, the `reducer` method, `fReduce` transposes the list of frequencies distribution then summed the amplitude for each frequency (line 21):

```
def fReduce(buf: List[DblVec]): immutable.Seq[Double] =  
    buf.transpose.map(_.sum).toSeq //21
```

Let's look at the test code:

```
val NUM_WORKERS = 4  
val NUM_DATAPOINTS = 1000000  
val h = (x:Double) => 2.0*cos(Math.PI*0.005*x) +  
    cos(Math.PI*0.05*x) + 0.5*cos(Math.PI*0.2*x) +  
    0.3* nextDouble //22  
  
val actorSystem = ActorSystem("System") //23  
val xt = Vector.tabulate(NUM_POINTS)(h(_))
```

```

val controller = actorSystem.actorOf(
    Props(new DFTMasterWithRouter(xt, NUM_WORKERS, fReduce)) ,
    "MasterWithRouter"
) //24
controller ! Start(1) //25

```

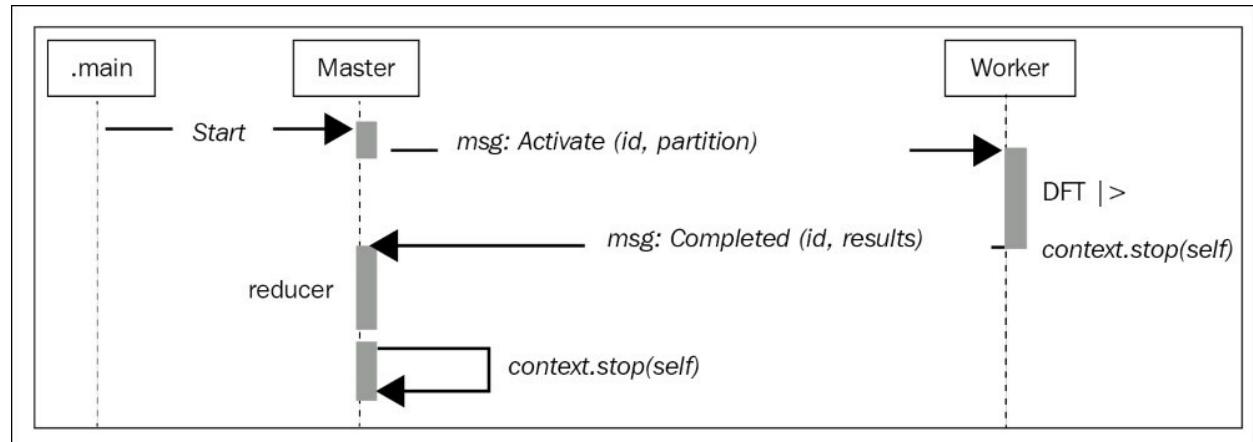
The input time series is synthetically generated by the noisy sinusoidal function, h (line 22). The function h has three distinct harmonics, 0.005, 0.05, and 0.2, so the results of the transformation can be easily validated. The Actor system, `ActorSystem`, is instantiated (line 23) and the master Actor is generated through the `Akka ActorSystem.actorOf` factory (line 24). The main program sends a `Start` message to the master to trigger the distributed computation of the discrete Fourier transform (line 25).

Tip

Action Instantiation:

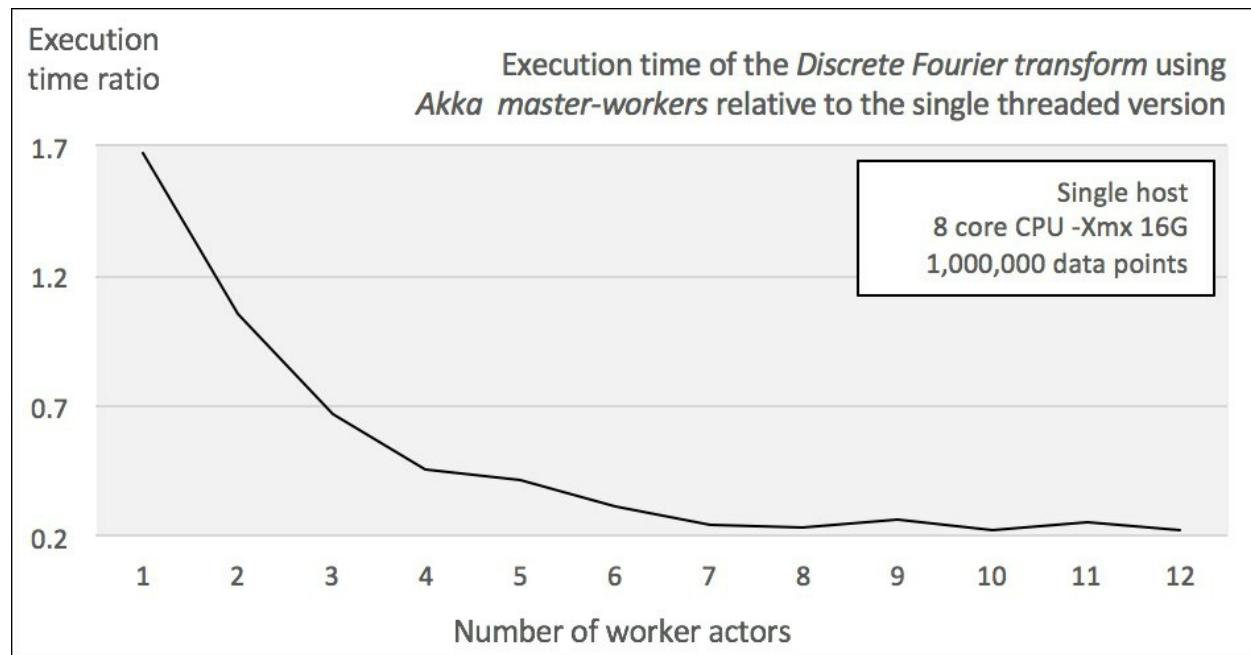
Although the `scala.actor.Actor` class can be instantiated using the constructor, `akka.actor.Actor` is instantiated using a context, `ActorSystem`; a factory, `actorOf`; and a configuration object, `Props`. This second approach has several benefits, including decoupling the deployment of the Actor from 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 DFT 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:



Impact of the number of worker (slave) Actors on the performance of the discrete Fourier transform

The single-threaded version of the DFT 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 the execution of the Fourier transform. However, the master worker model is far more efficient with three or more worker Actors.

Limitations

The master-worker implementation has a few problems:

- 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 certain 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 [16:10].

There are two options to retrieve results from futures:

- Blocking execution using `scala.concurrent.Await`
- Callback functions, `onComplete`, `onSuccess`, and `onFailure`

Note

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.

Let's re-implement the normalization of cross-validation groups by their variance, which we introduced in the previous section, using futures to support concurrency. The first step is to import the appropriate classes for execution of the main Actor and futures, as follows:

```
import akka.actor.{Actor, ActorSystem, ActorRef, Props} //26
import akka.util.Timeout    //27
import scala.concurrent.{Await, Future}   //28
```

The Actor classes are provided by the package `akka.actor`, instead of the `scala.actor._` package because of Akka's extended Actor model (line 26).

The future-related classes, `Future` and `Await`, are imported from the `scala.concurrent` package, which is similar to the `java.concurrent` package (line 28). The `akka.util.Timeout` class is used to specify the maximum duration the Actor has to wait for the completion of the futures (line 27).

There are two options for a parent Actor or the main program to manage the futures it creates:

- **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, which 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`, which manages futures instead of workers or routing Actors, as follows:

```
abstract class TFuture(  
    xt: DblVec,  
    fct: PfnTransform,  
    nPartitions: Int)  
  (implicit timeout: Timeout) //29  
  extends Controller(xt, fct, nPartitions) {  
  
  override def receive = {  
    case s: Start => compute(transform) //30  
  }  
}
```

The `TransformFutures` class requires the same parameters as the `Master` Actor: a time series, `xt`; a data transformation, `fct`; and the number of partitions, `nPartitions`. The `timeout` parameter is an implicit argument of the `Await.result` method, and therefore, needs to be declared as an

argument (line 29). The only message, `start`, triggers the computation of the data transformation of each future, and then the aggregation of the results (line 30). The `transform` and `compute` methods have the same semantics as those in the master-workers design.

Note

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 the 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[DblVec]] = {
  val futures = new Array[Future[DblVec]](nPartitions) //31

  partition.zipWithIndex.foreach { case (x, n) => { //32
    futures(n) = Future[DblVector] { fct(x).get } //33
  }}
  futures
}
```

An array of `futures` (one future per partition) is created (line 31). The `transform` method invokes the partitioning method `partition` (line 32) and then initializes the future with the partial function `fct` (line 33):

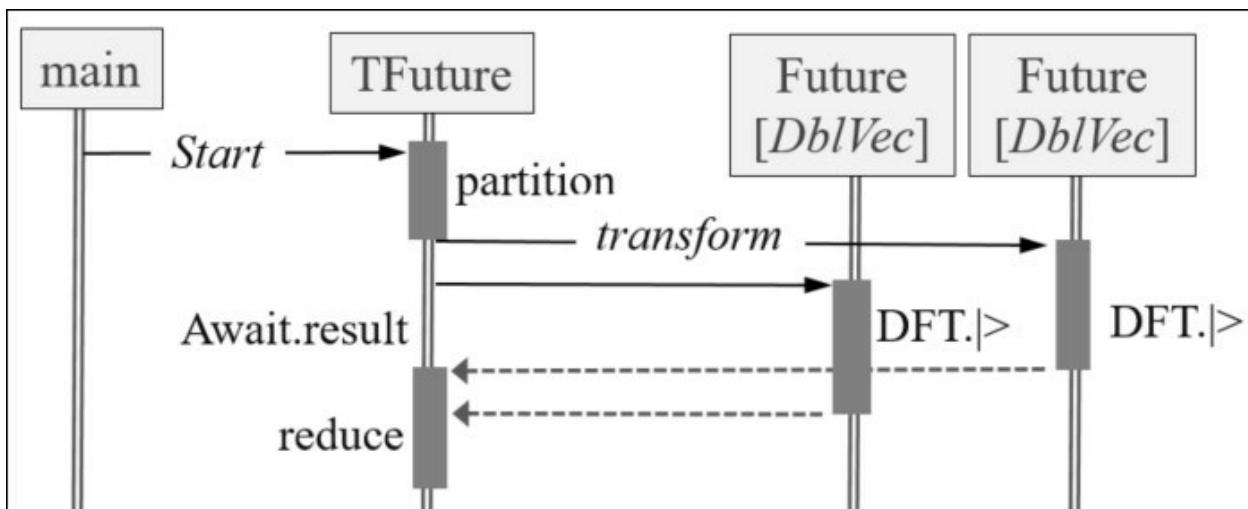
```
def compute(futures: Array[Future[DblVec]]): Seq[Double] =
  reduce(futures.map(Await.result(_, timeout.duration))) //34
```

The `compute` method invokes a user-defined function, `reduce`, on the `futures`. The execution of the Actor is blocked until the `Await` class method (line 34)

`scala.concurrent.Await.result` returns the result of each future computation. In the case of the DFT, the list of frequencies is transposed before the amplitude of each frequency is summed (line 35):

```
def reduce(data: Array[DblVec]): Seq[Double] =
  data.view.map(_.toArray)
    .transpose.map(_.sum) //35
    .take(SPECTRUM_WIDTH).toSeq
```

The following sequential diagram illustrates the blocking design and the activities performed by the Actor and the futures:



Sequential diagram for Actor blocking on future results

Future callbacks

Callbacks are an excellent alternative to having the Actor block 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 an

argument with an implicit reference to the execution context, as shown in the following code:

```
val f: Future[T] = future { execute task } f onComplete {
  case Success(s) => { ... }
  case Failure(e) => { ... }
}
```

You can surely recognize the `{Try, Success, Failure}` monad.

An alternative implementation is to invoke the `onSuccess` and `onFailure` methods that use partial functions as arguments to implement the callbacks, as follows:

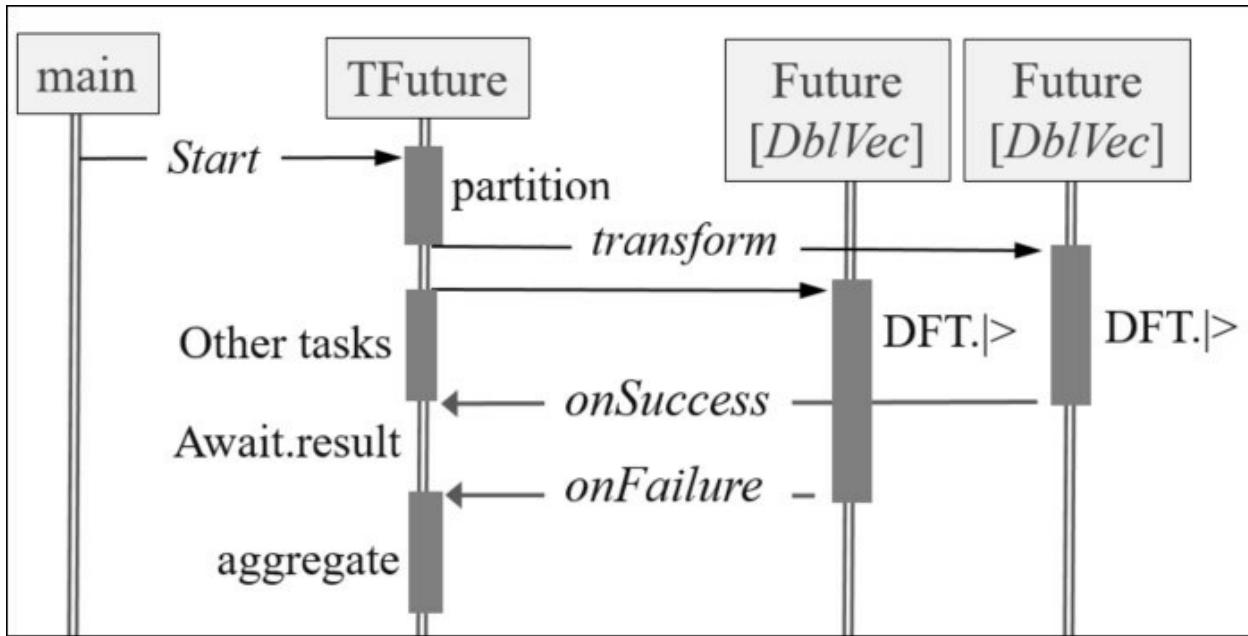
```
f onFailure { case e: Exception => { ... } }
f onSuccess { case t => { ... } }
```

The only difference between blocking one future data transformation and handling callbacks is the implementation of the `compute` method or reducer. The class definition, message handler, and initialization of futures are identical, as shown in the following code:

```
def compute(futures: Array[Future[DblVec]]): Seq[Double] = {
  val buffer = new ArrayBuffer[DblVec]

  futures.foreach( f => {
    f onSuccess { //36
      case data: DblVec => buffer.append(data)
    }
    f onFailure { case e: Exception => /* .. */ } //37
  })
  buffer.find( _.isEmpty).map( _ => reduce(buffer)) //38
}
```

Each future calls the master Actor back with either the result of the data transformation, the `onSuccess` message (line 36), or an exception, the `onFailure` message (line 37). If every future succeeds, the values of every frequency for all the partitions are summed (line 38). The following sequential diagram illustrates the handling of the callback in the master Actor:



Sequential diagram for Actor handling future result with Callbacks

Note

Execution context

- The application of futures 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 it all together

Let's reuse the DFT. 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 DFT, `DFTFuture`, as follows:

```
class DFTFuture (
```

```

xt: DblVec,
partitions: Int)(implicit timeout: Timeout)
extends TFuture[xt, DFT[Double].|>, partitions] {

override def reduce(data: Array[DblVec]): Seq[Double] =
  data.map(_.toArray).transpose//39
    .map(_.sum) //40
    .take(SPECTRUM_WIDTH).toSeq //41
}

```

The `reduce` method converts an array of vectors into a matrix of type `Array[Array[Double]]`, transpose the matrix (line 39) then computes the sum of frequencies count (line 40). The method returns the first `SPECTRUM_WIDTH` frequencies (line 41).

The only purpose of the `DFTFuture` class is to define the aggregation method, `reduce`, for the discrete Fourier transform. Let's reuse the same test case as in the *Distributed discrete Fourier transform* section under *Master-workers*:

```

import akka.pattern.ask

val duration = Duration(8000, "millis")
implicit val timeout = new Timeout(duration)
val master = actorSystem.actorOf( //42
  Props(new DFTFuture(xt, NUM_WORKERS)), "DFT"
)
val future = master ? Start(0)//43
Await.result(future, timeout.duration) //44
actorSystem.shutdown//45

```

The master Actor is initialized as of the `TFuture` type with the input time series, `xt`; `DFT`, `DFT`; and the number of workers or partitions `nPartitions` as arguments (line 42). The program creates a `future` instance, by sending (`ask`) the `Start` message to `master` (line 43). The program blocks until the completion of the future (line 44), and then shuts down the Akka Actor system (line 45).

Summary

This concludes the first of the two chapters dedicated to scalability in using Scala to build machine learning-based applications.

This first part dealt with the parallel collections in Scala as a trivial but effective way to make your application scalable. You also learned about the benefits of asynchronous concurrency in Scala, the Akka framework, and the essential elements of the Actor model and composed futures as applied to improve the performance of a distributed application.

The Akka framework is the underlying mechanism used in Apache Spark to distribute runtime execution and exchange/broadcast data. The Apache Spark framework is the topic of the next chapter.

Chapter 17. Apache Spark MLlib

The previous [Chapter 16](#), *Parallelism with Scala and Akka*, provided the reader with different options to make computing-intensive applications scalable. These solutions are generic and do not address the specific needs of the data scientist. Optimization algorithms such as minimization of loss function or dynamic programming methods such as the Viterbi algorithm require support for caching data and broadcasting of model parameters. The Apache Spark framework addresses these shortcomings.

This chapter describes the key concepts behind the Apache Spark framework and its application to large scale machine learning problems. The reader is invited to dig into the wealth of books and papers on this topic. This last chapter describes four key characteristics of the Apache Spark framework:

- MLlib functionality as illustrated with the K-means algorithm
- Reusable ML pipelines, introduced in Spark 2.0
- Extensibility of existing Spark functionality using Kullback-Leibler divergence as an example
- Spark streaming library

Overview

Apache Spark is a fast and general-purpose cluster computing system, initially developed as **AMPLab / UC Berkeley** as part of the **Berkeley Data Analytics Stack (BDAS)**, http://en.wikipedia.org/wiki/UC_Berkeley. It provides high-level APIs for the following programming languages that make large, concurrent parallel jobs easy to write and deploy [17:01]:

- **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>

Note

Link to latest information

The URLs as any reference to Apache Spark may change in future versions.

The core element of Spark is the **resilient distributed dataset (RDD)**, which is a collection of elements partitioned across the nodes of a cluster and/or CPU cores of servers. An RDD can be created from a local data structure such as a list, array, or hash table, from the local filesystem or the **Hadoop distributed file system (HDFS)** [17:02].

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:

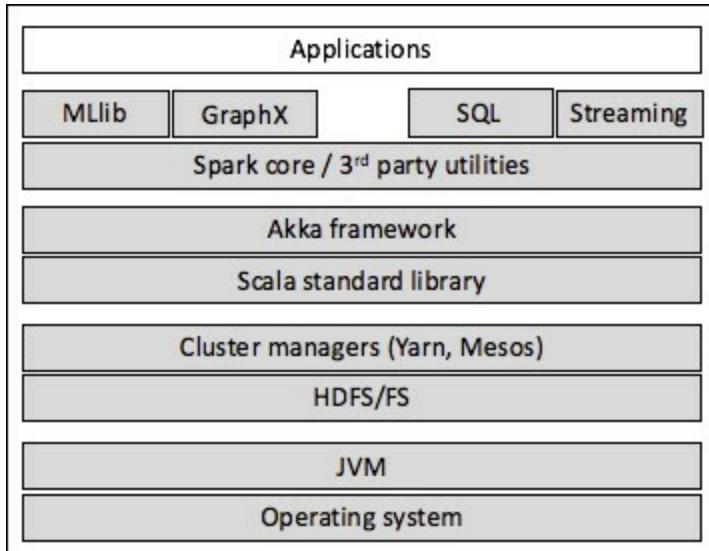
- **Transformation:** This operation converts, manipulates, and filters the elements of an RDD on each partition
- **Action:** This operation aggregates, collects, or reduces the elements of the RDD from all partitions

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 filesystem
- Mesos or Yarn for management of cluster and shared pool of data nodes

The Spark ecosystem can be represented as stacks of technology and frameworks, as seen in the following diagram:



Apache Spark framework ecosystem

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 [17:12].

Apache Spark core

The RDD is the core data structure of the Apache Spark architecture. RDDs store and preserve data distributed and partitioned over multiple processors and servers so operations can be executed concurrently.

Data frames have been added, later on, to extend RDDs with SQL functionality. The original Apache Spark machine learning library, MLlib, uses RDDs that operate at a lower level (API). The more recent ML library allows data scientists to describe transformation and actions using SQL.

Note

Deprecation RDD-based API for MLlib

The RDD-based classes and methods in MLlib have moved to maintenance mode in Spark 2.0 and will be completely removed in Spark 3.0

Why Spark?

The introduction of the Hadoop ecosystem more than 10 years ago, opened the door to large-scale data processing and analytics. The Hadoop framework relies on a very effective distributed filesystem, HDFS, suitable for processing a large number of files containing sequential data. However, this reliance on the distributed filesystem for managing data becomes a bottleneck when executing iterative computation such as dynamic programming and machine learning algorithms.

Apache Spark lays out an efficient, resilient memory management layer on top of HDFS. This design provides developers with a very significant performance improvement, tenfold to hundred-fold depending on the type of applications.

Spark provides data engineers with a wide array of prebuilt transformation and actions inspired by the Scala API. Scala developers can leverage their knowledge of the language. However, Spark API is also available in Java and Python.

Design principles

The performance of Spark relies on four core design principles [17:03]:

- In-memory persistency
- Laziness in scheduling tasks
- Transformation 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 de-serialized 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
```

Note

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` (line 1). 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, and then `g` initializes `x` to compute its power to `m<<1` (line 2).

Spark applies the same principle to RDDs by executing the transformation only when an action is performed. In other words, Spark postpones memory allocation, parallelization, and computation until the driver code gets the result through the execution of an action. The cascading effect of invoking all these transformations 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>	Transform an RDD by executing the function <code>f</code> on each element of the collection

<code>filter(f)</code>	<code>filter(f)</code>	Transform an RDD by selecting the element for which the function <code>f</code> returns <code>true</code>
<code>flatMap(f)</code>	<code>flatMap(f)</code>	Transform an RDD by mapping each element to a sequence of output items
<code>mapPartitions(f)</code>		Execute the <code>map</code> method separately on each partition
<code>sample</code>		Sample a fraction of the data with or without a replacement using a random generator
<code>groupByKey</code>	<code>groupBy</code>	Called on (K, V) to generate a new $(K, Seq(V))$ RDD
<code>union</code>	<code>union</code>	Create a new RDD as union of this RDD and the argument
<code>distinct</code>	<code>distinct</code>	Eliminate duplicate elements from this RDD
<code>reduceByKey(f)</code>	<code>reduce</code>	Aggregate or reduce the value corresponding to each key using the function <code>f</code>
<code>sortByKey</code>	<code>sortWith</code>	Reorganize (K, V) in an RDD by the ascending, descending, or otherwise

		specified order of the keys, K
join		Join a RDD (K,V) with a RDD (K,W) to generate a new RDD (K, (V,W))
coGroup		Implement a join operation, but generate a 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)	Aggregate all the elements of the RDD across all the partitions and return a Scala object to the driver
collect	collect	Collect and return all the elements of the RDD across all the partitions as a list in the driver
count	count	Return the number of elements in the RDD to the driver
first	head	Return the first element of the RDD to the driver
take (n)	take (n)	Return the first n elements of the RDD to the driver

		Return an array of random elements from the RDD back to the driver
		Write the elements of the RDD as a text file in either the local filesystem or HDFS
		Generate a (K, Int) RDD with the original keys, K, and the count of values for each key
foreach	foreach	Execute a T=> Unit function on each element of the RDD

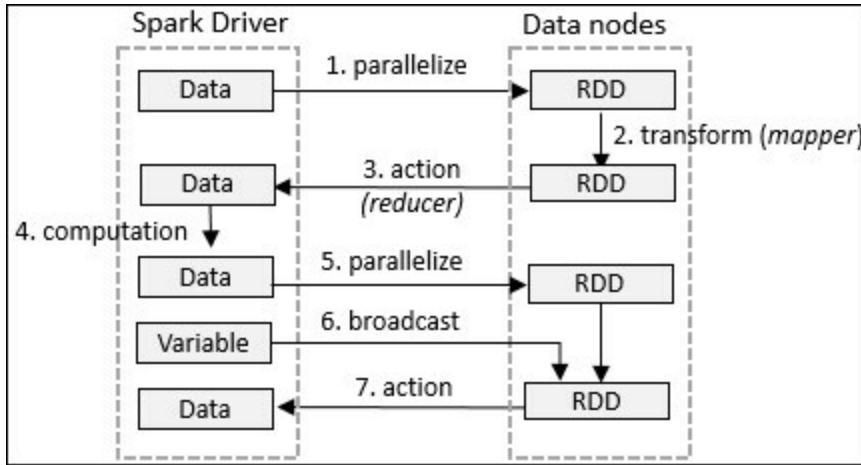
Scala methods such as `fold`, `find`, `drop`, `flatten`, `min`, `max`, and `sum` are not currently implemented in Spark. Other Scala methods such as `zip` have to be used carefully, as there is no guarantee that the order of the two collections in `zip` is maintained between partitions.

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 into the driver.

Note

Similarity with Akka

If you look closely, the management of datasets and RDDs by the Spark driver is not very different from that by Akka master and Actors actors of

futures.

Experimenting with Spark

Spark's in-memory computation enables iterative computing found in the training of machine learning models or execution dynamic programming or optimization algorithms. Spark runs on Windows, Linux, and macOS 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 in this chapter is 2.1.

Note

JVM and Scala compatible versions

At the time of writing, the version of Spark 2.11 required Java 1.7 or higher and Scala 2.11.5 or higher. Future releases may require Scala 2.12.

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>.
5. Next, you need to build, or assemble, the Apache Spark libraries from the top-level directory using either *Maven* or *sbt*.
6. Set the following Maven options to support build, deployment, and

execution:

```
MAVEN_OPTS="-Xmx4g -XX:MaxPermSize=512M  
           -XX:ReservedCodeCacheSize=512m"  
mvn [args] -DskipTests clean package
```

Example:

Building on Hadoop 2.7 using Yarn clusters manager and Scala 2.11:

```
mvn -Pyarn -pHadoop-2.7 -Dhadoop.version=2.7.0 -Dscala=2.11  
     -DskipTests clean package
```

7. Use the following command for simple build tool:

```
sbt/sbt [args] assembly
```

Example:

Building on Hadoop 2.7 using Yarn clusters manager and Scala 2.11:

```
sbt -Pyarn -pHadoop 2.7 -Dscala=2.11 assembly
```

Note

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.

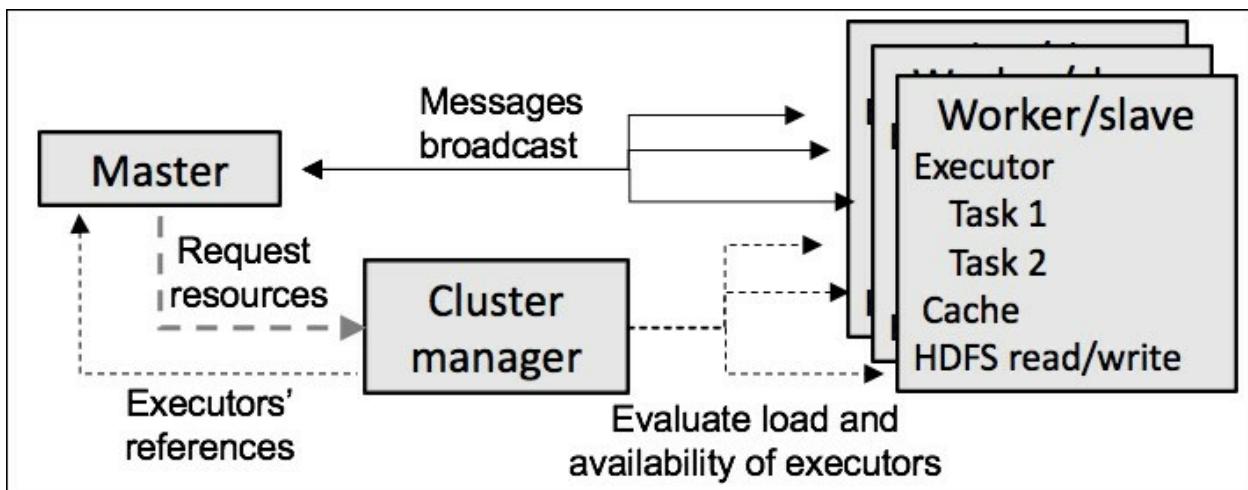
Apache supports multiple deployment modes:

- **Standalone mode:** The driver and executors run as master and slave Akka actors, bundled with the default Spark distribution JAR file.
- **Local mode:** This is a standalone mode running on a single host. The slave Actors are deployed across multiple core within the same host.
- **Yarn clusters manager:** Spark relies on the Yarn resource manager running on Hadoop version 2 and higher. The Spark driver can run either on the same JVM as the client application (client mode) or on the

same JVM as the master (cluster mode).

- **Apache Mesos resource manager:** This deployment allows dynamic and scalable partitioning. Apache Mesos is an open source, general purpose cluster manager that has to be installed separately (refer to <http://mesos.apache.org/>). Mesos manages the abstracted hardware artifacts such as memory or storage.

The communication between a master node (or driver), a cluster manager, and set of slave (or worker) nodes is illustrated in the following diagram:



Master-worker communication through a cluster manager

Tip

Installation under Windows

Hadoop relies on some UNIX/Linux utilities that need to be added to the development environment when running on Windows. The file `winutils.exe` has to be installed and added to the `HADOOP_PATH` environment variable.

Using Spark shell

Use any of the following methods to use the Spark shell:

- The shell is an easy way to get your feet wet with Spark-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]
  --num-executors 10
--confspark.executor.extraJavaOptions="-XX:MaxPermSize=512m"
--executor-memory 12G
myApplication.jar
```

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 to execute the following command line:

```
./bin/spark-submit
--class application_class
--master spark://162.198.11.201:7077
--total-executor-cores 80
--confspark.executor.extraJavaOptions="-XX:MaxPermSize=512m"
--driver-library-path xxx // path for binary libraries
  --executor-memory 24G
myApplication.jar
```

The following screenshot captures the output for launching a Spark shell. The standard output dumps:

- IP port of the web-based UI to monitor staging and execution on RDDs and Data frames: `10.1.2.194:4040`
- The deployment mode: standalone, local deployment for which the number of cores [*] is computed dynamically
- Session-based context:

Partial screenshot of Spark shell command line output

Tip

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 the `ADD_JARS` environment variable as `ADD_JARS = path1/jar1, path2/jar2`.

MLlib library

MLlib is a scalable machine learning library built on top of Spark. The machine learning library is composed of two distinct packages, which are [17:03]:

1. `org.apache.spark.mllib`: RDD-based library of some common machine learning algorithms. This package will be deprecated in future releases.
2. `org.apache.spark.ml`: Library of machine learning algorithms that leverages datasets and data frames structures. The package supports tasks pipeline and stages that are described and illustrated in the next section.

Overview

The main components of the MLlib package are as follows:

- Classification algorithms, including logistic regression, Naïve Bayes, and support vector machines
- Clustering and unsupervised learning techniques such as K-means
- 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.

Creating RDDs

The transformation and actions are performed on RDDs. Therefore, the first step is to create a mechanism to facilitate the generation of RDDs from a time series.

Let's create an `RDDSource` singleton with a `convert` method that transforms a time series, `xt`, into an RDD, as shown here:

```
def convert(
    xt: immutable.Vector[DblArray],
    rddConfig: RDDConfig)
  (implicit sc: SparkContext): RDD[Vector] = {

  val rdd: RDD[Vector] =
    sc.parallelize(xt.toVector.map(new DenseVector(_))) //3
    rdd.persist(rddConfig.persist) //4
    if(rddConfig.cache) rdd.cache //5
    rdd
}
```

The last argument of the method `convert`, `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(cache: Boolean, 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 it into a vector (`SparseVector` or `DenseVector`) (line 3).
2. Specify the persistency model or the storage level if the default level needs to be overridden for the RDD (line 3).
3. Specify whether the RDD has to persist in memory (line 5).

Note

Alternative creation of an RDD

An RDD can be generated from data loaded from either the local filesystem or HDFS using the `SparkContext.textFile` method that returns an RDD of string.

Once the RDD is created, it can be used as an input for any algorithm defined as a sequence of transformation and actions. Let's experiment with the implementation of the K-means algorithm in Spark/MLlib.

K-means using MLlib

The first step is to create a `KmeansConfig` class to define the configuration of the Apache Spark K-means algorithm, as follows:

```
Class KmeansConfig(  
  K: Int, maxIters:  
    Int, numRuns: Int =1) {  
  val kmeans: KMeans =  
    (newKMeans).setK(K) //6  
      .setMaxIterations(maxIters) //7  
      .setRuns(numRuns) //8  
}
```

The minimum set of initialization parameters for MLlib K-means algorithm is as follows:

- Number of clusters, `K` (line 6)
- Maximum number-iterations for the reconstruction of the total error, `maxIters` (line 7)
- Number of training runs, `numRuns` (line 8)

The `SparkKMeans` class wraps the `Spark KMeans` into a data transformation of type `ITransform` described in the *Monadic data transformation* section of [Chapter 2, Data Pipelines](#). The class follows the design template for classifier as explained in the *Design template for classifiers* section in [Appendix](#):

```
class Kmeans( //9  
  kmeansConfig: KmeansConfig,  
  rddConfig: RDDConfig,  
  xt: Vector[Array[Double]])(implicit sc: SparkContext)  
extends ITransform[Array[Double], Int] { //10  
  
  val model: Option[KMeansModel] = train //11  
  override def |> : PartialFunction[Array[Double], Try[Int]] //12  
  def train: Option[KMeansModel]  
}
```

The constructor takes three arguments: The Apache Spark `KMeans` configuration, `config`; the RDD configuration, `rddConfig`; and the input time

series to clustering, `xt` (line 9). The return type of the `ITransform` partial function `|>` is defined as an `Int` (line 10).

The generation of `model` merely consists of converting the time series `xt` into an RDD using `rddConfig` and invoking `MLlib KMeans.run` (line 11). Once created, the model of clusters (`KMeansModel`) is available for predicting a new observation, `x`, (line 12) as follows:

```
override def: PartialFunction[Array[Double], Try[V]] = {  
    case x: Array[Double] if(x.nonEmpty && model.isDefined) =>  
        Try[Int](model.get.predict(new DenseVector(x)))  
}
```

The prediction method, `|>`, returns the index of the cluster of observations.

Finally, let's write a simple client program to exercise the `Kmeans` model using the volatility of the price of a stock and its daily trading volume. The objective is to extract clusters with features `{volatility, volume}`, each cluster representing a specific behavior of the stock:

```
val K = 8; val RUNS = 16; val MAXITERS = 200  
val CACHE = true  
  
val sparkConf = new SparkConf().setMaster("local[8]")  
    .setAppName("Kmeans")  
    .set("spark.executor.memory", "2048m") //13  
implicit val sc = new SparkContext(sparkConf) //14  
  
extract.map{ case (vty,vol) => { //15  
    val vtyVol = zipToSeries(vty, vol)  
    val conf = KmeansConfig(K,MAXITERS,RUNS) //16  
    val rddConf = RDDConfig(CACHE, MEMORY_ONLY) //17  
    val pfFnKmeans = Kmeans(conf,rddConf,vtyVol) |> //18  
    val obs = Array[Double](0.23, 0.67)  
    val clusterId = pfFnKmeans(obs)  
}}
```

The first step is to define the minimum configuration for context `sc` (line 13) and initialize it (line 14). The two variables, `volatility` and `volume`, are used as features for K-means and extracted from a CSV file (line 15):

```
type DblVec = Vector[Double]  
def extract: Option[(DblVec, DblVec)] = {
```

```

val extractors = List[Array[String] => Double] (
  volatility, volume
)
val pfnSrc = DataSource(PATH, true) |>
  pfnSrc( extractors ) match {
  case Success(x) => Some((x(0).toVector, x(1).toVector))
  case Failure(e) => { error(e.toString); None }
}
}

```

The execution is to create a configuration `config` for the K-means (line 16) and another configuration for the Spark RDD `rddConfig`, (line 17). The partial function `pfnKMeans` that implements the K-means algorithm is created with the K-means, RDD configurations, and the input data `vtyVol` (line 18).

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 months, 72 month, 96 months, and 120 months.

Tip

Meaningful performance test

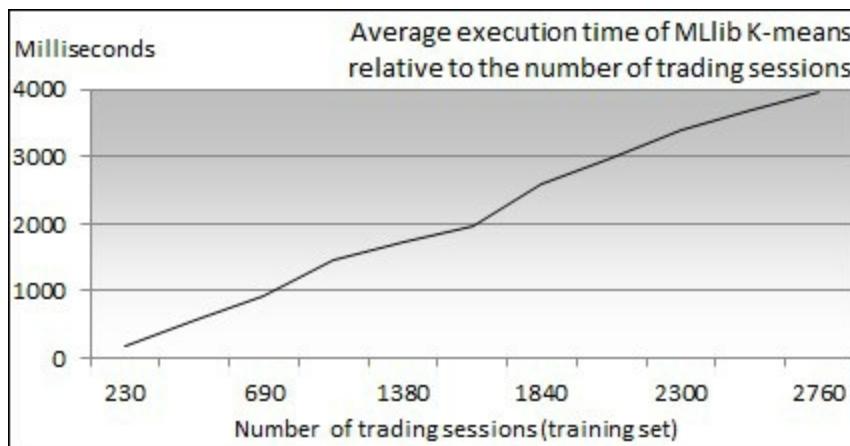
The scalability test should be performed with a large number of data points (normalized volatility, normalized volume), in excess of 1 million in order to estimate the asymptotic time complexity.

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.

The following diagram illustrates the average duration of the execution of the K-means algorithm in MLlib with a variable number of data points (trading sessions):



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.

Note

Performance 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.

Reusable ML pipelines

ML pipelines have been introduced in Apache Spark 1.4.0. An ML pipeline is a sequence of tasks that can be used to cleanse, filter, train, classify observations, detect anomalies, generate, validate models, and predict outcomes [17:04].

Contrary to the MLlib package classes that rely on RDDs, ML pipeline uses data frame or datasets as input and output of tasks.

Note

Data frame versus Dataset

The class `Dataset` was introduced in Spark 2.0. Dataset instances are typed (that is, `Dataset[T]`) while data frames are untyped.

This section is a very brief overview of ML pipelines.

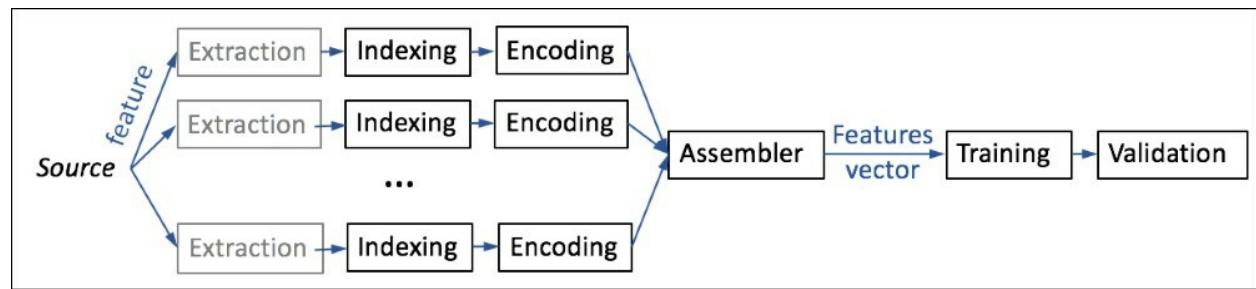
The key ingredients of an ML pipeline are [17:05]:

- *Transformers* are algorithms that can transform one data frame into another data frame. Transformers are stateless.
- *Estimators* are algorithms that can fit on a data frame to produce a Transformer (that is, `Estimator.fit`).
- *Pipelines* are estimators that weave or chain multiple transformers and estimators together to specify an ML workflow.
- *Pipeline stages* are the basic element of the ML workflow. Transformers, estimators, and pipelines are pipeline stages.
- *Parameters* encapsulate the tuning and configuration parameters required for each transformers and estimators.

Reusable ML transforms

We apply ML pipelines to the training and validation of a model to classify sales data. We illustrate the modularity and reusability of the pipeline of ML tasks available in Spark 2.0 by extracting, encoding, and indexing categorical features, assembled into a sparse vector, and finally training and validating a model.

Categorical features can be quite challenging. The process of extracting, indexing, and encoding each feature as well as assembling a vector (usually sparse) feature is error prone. This problem is the perfect candidate for reusability. The purpose is to create a reusable pre-defined sequence of pipeline stages as follows:



ML pipeline for the encoding, training, and validation of categorical features

The Apache Spark package `org.apache.spark.ml.feature`, contains an extensive list feature representation or transfers from *hashing*, *bucketizing*, N-Grams, tokenizer to scaler. Our encoding sub-pipeline leverages three features transforms, which are:

- `StringIndexer`: This is a class used for indexing. It maps a feature label to a categorical index used in the encoding transform
- `OneHotEncoder`: This is a class used for encoding. It maps a feature categorical index to binary vectors
- `VectorAssembler`: This is a class used for feature assembling. It merges the different binary vectors columns into a single vector

Note

Parsing observations

The extraction or parsing of observations is not included in our pipeline. It will be treated in the *Streaming* section of this chapter.

Encoding features

Categorical features have multiple values or instances that are represented as a string. Numerical values are categorized or bucketed through a conversion to a string.

Once converted to a string, the categorical values are converted into category indices using the `StringIndex` transformer. The resulting sequence of indices is ranked by decreasing order of frequency of the value in the training or validation set.

Next, the indices associated to features are encoded into a vector of binary value (0, 1) through the `OneHotEncoder` transformer. A feature instance is encoded as 1 if it is defined in the data point, it is 0 otherwise.

Finally, the vector of binary values of all the features are aggregated or assembled through the transformer `VectorAssembler`.

Let's consider the following simple use case. A sales report lists the following attributes: the `date` an item is sold, its identification `id`, the sales `region` and name of the sales person or `agent` as illustrated in the following record:

```
date: 07/10/2014
id: 23c9a89d
region: 17
sales person code: aa4
```

The first step is to create the sequence of pipeline stages that indexes and encode each feature and assemble the features vector. The trait `DataEncoding` has an abstract value that defines the name of the column which data is to be used in the training (line 1):

```

trait DataEncoding{
    protected[this] val cols: Array[String] // 1
    private lazy val vectorizedCols= cols.map(vector(_))

    lazy val stages = cols.map(col =>new StringIndexer().setInputCol(
        .setOutputCol(s"${col}Index")) ++ // 2
        cols.map(col =>new OneHotEncoder().setInputCol(s"${col}Index"
            .setOutputCol(s"${col}Vector")) ++
        Array[PipelineStage](new VectorAssembler() // 3
            .setInputCols(vectorizedCols)
            .setOutputCol("features"))
    private def vector(col: String): String = s"${col}Vector"
}

```

The pipeline `stages` are defined with an input column and an output column. The name of the column output from the indexer and column input into the encoder is automatically generated by appending `Index` to the input column name, `col` (line 2). The encoder outputs a column and its name follows the same convention. The input column to the vector assembler is arbitrarily named by appending the suffix `Vector` to its name (line 3).

Tip

Naming convention for Dataset columns

Apache Spark datasets and data frames represent categorical features as columns. The source code becomes more readable and easier to maintain if a column of the dataset and its corresponding categorical feature share the same name.

The output data frame contains the columns created through the different pipeline stages such as string indexing or one encoding of these categorical features.

The schema for the output frame is described as follows:

```

root
|--- date: string (nullable = true)
|--- asset: string (nullable = true)
|--- region: integer (nullable = true)
|--- agent: string (nullable = true)
|--- dateIndex: double (nullable = true)
|--- assetIndex: double (nullable = true)
|--- regionIndex: double (nullable = true)
|--- agentIndex: double (nullable = true)
|--- dateVector: vector (nullable = true)
|--- assetVector: vector (nullable = true)
|--- regionVector: vector (nullable = true)
|--- agentVector: vector (nullable = true)
|--- features: vector (nullable = true)
|--- rawPrediction: vector (nullable = true)
|--- probability: vector (nullable = true)
|--- prediction: double (nullable = true)

```

Schema for the data frame output from predictive model

Let's consider the following two records for the sales report:

	date	asset	region	agent
07/05/2014	300ec90b	27	aa5	
08/03/2014	23c9ab02	7	a08	

Sample of content of data frame of observations

The string indexer generates the following values for each of the fields of the sales report as follows:

dateIndex	assetIndex	regionIndex	agentIndex
4.0	4.0	19.0	6.0
3.0	1.0	13.0	2.0

Schema for the data frame output from string indexer

The value for each column (or categorical feature) is encoded as shown:

dateVector	assetVector	regionVector	agentVector	features
(16, [4], [1.0]) (14, [4], [1.0]) (20, [19], [1.0]) (15, [6], [1.0]) (65, [4, 20, 49, 56], ...)				
(16, [3], [1.0]) (14, [1], [1.0]) (20, [13], [1.0]) (15, [2], [1.0]) (65, [3, 17, 43, 52], ...)				

Schema for the data frame output from encoding (OneHotEncoding)

The last column, `features`, represents the final encoded feature vector for this record or observation.

Training the model

Once the observations vectors are encoded, they can be used to train a model, using an estimator. The estimator, implemented by the trait `ModelEstimator`, is parameterized by the type of model or machine learning algorithm (line 4). The algorithm that generates the model is specified at runtime by overriding the abstract value, `estimator`:

```
trait ModelEstimator[T <: Model[T]] { // 4
  protected[this] val estimator: Estimator[T] // 5
  def apply() // 6
```

```

trainSet: DataFrame,
stages: Array[PipelineStage]
): PipelineModel = {
  val pipeline = new Pipeline().setStages(stages ++
    Array[PipelineStage](estimator)) // 7
  pipeline.fit(train)
}

def trainWithSummary( ... )
}

```

The model is trained by invoking the method `apply` (line 6). It takes two parameters:

- The data frame, `trainSet`, which contains the labeled training set, previously encoded
- The encoding `stages` generated in the previous section

The estimator is converted into a pipeline stage that extends encoding stages (line 7). Finally, the pipeline model is generated by fitting the training set, `trainSet`, into the `pipeline` (line 8).

Tip

Estimator semantic

The definition of estimator is borrowed from the *Scikit-learn* library. It refers to classification, regressive, or predictive models.

The `trainWithSummary` method implements the execution of the training of the model along with the generation of two quality metrics, which are:

- F_1 -score
- Area under the Receiver Operating Characteristic (AuROC or AUC)

The **receiver operating characteristic (ROC)** evaluates the performance of a binary classifier by plotting the **true positive rate (TPR)** against the **false positive rate (FPR)** for a set of discriminative classification threshold [17:06].

The TPRs and FPRs are computed by the following formulas:

$$TPR = \frac{TP}{TP + FN} \quad TFP = \frac{TP}{TP + FP}$$

Note

Sensitivity and fallout

True positive rate (resp. false positive rate) is also known as sensitivity (resp. fallout).

The area under the ROC curve computes the integral under the ROC curve. It can be interpreted as the probability that the binary classifier ranks a randomly chosen positive observation higher than a randomly chosen negative observation.

Tip

Metrics

The different model training metrics such as F_1 -score or AuROC were introduced in the *Assessing a model*' section in [Chapter 2, Data Pipelines](#).

There are many possible implementations of the computation of AuROC [17:07]. Our implementation leverages the package `org.apache.spark.ml.classification` of the Apache Spark library.

The method `trainWithSummary` takes two arguments:

- The labeled training data frame, `trainSet` (line 9)
- All the `stages` need to train the model (line 10)

The method invokes the `apply` method to generate the pipeline model (line 11):

```

def trainWithSummary(
    trainSet: DataFrame,           // 9
    stages: Array[PipelineStage]   // 10
  ): Option[(Double, Double)] = {
  this(trainSet, stages).stages.last match {           // 11
    case lrModel: LogisticRegressionModel =>
      val binarySummary = lrModel.summary
        .asInstanceOf[BinaryLogisticRegressionSummary]

      val f1 = binarySummary.fMeasureByThreshold // 12
        .select("F-Measure").head.getDouble(0)
      (f1, binarySummary.areaUnderROC)
    case _ => None
  }
}

```

The two-quality metrics, f (F_1 -score) and **areaUnderROC** (**AuROC**) are extracted from the binary summary (line 12).

Tip

BinaryLogisticRegressionSummary

The class `BinaryLogisticRegressionSummary` is one of the classes from `org.apache.spark.ml.classification` packages that computes the different quality metrics for training and validation of models. The class `BinaryLogisticRegressionTrainingSummary` extends `BinaryLogisticRegressionSummary` with the method `objectiveHistory` that encapsulates the statistics related to the training run. Developers can also implement the generic `LogisticRegressionSummary` trait for multinomial models.

The next section implements the predictive model.

Predictive model

The prediction of new observation(s) using a trained model can be executed with or without cross-validation. The base class `Predictor` defines the parameters shared between the different versions of the predictive model:

- The estimator, `estimate`, is used to train or generate the model of type `T` (line 13).
- The array of name of columns, `cols`, representing the model features (line 14).
- The name of the file, `trainFile`, containing the labeled data (line 15):

```
abstract class Predictor[T <: Model[T]] (
    estimate: Estimator[T],           // 13
    cols: Array[String],             // 14
    trainFile: String                // 15
) extends DataEncoding with SessionLifecycle {

    override protected[this] val colNames = cols
    protected[this] val trainDf = csv2DF(trainFile) // 16
    override protected val estimator: Estimator[T] = estimate

    def classify(model: PipelineModel, testFile: String): Data
        model.transform(csv2DF(testFile)) //17
}
```

A predictor uses the pipeline stages created in the trait `DataEncoding`. It also inherits the basic life cycle components of a Spark session, such as Spark context, SQL context, and conversion of the generation of a data frame from the content of an input CSV file (method `csv2DF`) (line 16).

The trait `SessionLifecycle` is not described in this section. The classification method, `classify`, computes the prediction as a data frame using an input file, `testFile` and the pipeline model, `model`, generated through training (line 17).

Let's define a simple predictor as the class `simplePredictor`. The class takes the same three arguments as the generic predictor. It also implements the `ModelEstimator` trait that is used to rate the model used in the prediction (line 18):

```
Class simplePredictor[T <: Model[T]] (
    estimate: Estimator[T],
    cols: Array[String],
    trainFile: String
) extends Predictor[T](estimate, cols, trainFile)
    With ModelEstimator[T] { //18
        def apply(): PipelineModel = this(trainDf, stages)
```

```
}
```

The `apply` method returns the pipeline model used in the classification.

Let's apply our new-found knowledge to classify some sales reports. The training and test data is extracted from the resources file. In our example, we select the logistic regression as our estimator (line 19). The predictor requires the sequence of fields used from both the training and test set:

```
(for {
    trainPath<- ResourcesLoader.getPath(trainFile)
    testPath<- ResourcesLoader.getPath(testFile)
} yield {
    val predictor = new SimplePredictor[LogisticRegressionModel] (
        new LogisticRegression().setMaxIter(5).setRegParam(0.1), // 1
        Array[String]("date", "asset", "region", "agent"),
        trainPath
    )

    (predictor, predictor.classify(predictor, testPath)) // 20
}).map {
    case (predictor, output) => {
        val result = output.select("prediction")
            .collect
            .map(_.getDouble(0))

        predictor.stop
        result
    }
}
```

The model is trained by invoking the `apply` method on the predictor instance, `predictor()` (line 20). The predicted values are actually computed through the `classify` method.

Tip

Configuring a transformer or estimator

There are two ways to configure a ML transformer or estimator:

- Specify the parameters on the transformer or estimator, (that is, `new`

```
    LogisticRegression().setMaxIter(30)).
```

- Specify the parameters through the parameter map, (that is, new ParamMap().put(lr.maxIter-> 30)).

Training summary statistics

The next test is to generate the two basic statistics, F1-score and AuROC, related to the training phase. The implementation is quite similar to the prediction test with the exception that the method `trainingWithSummary` (line 21) is invoked on a `validatedPredictor` instance:

```
(for{
    trainPath<- ResourcesLoader.getPath(trainFile)
    testPath<- ResourcesLoader.getPath(testFile)
} yield {
    val lr = new LogisticRegression().setMaxIter(5)
        .setRegParam(0.1)

    val paramsMap = new ParamMap().put(lr.maxIter-> 30)
        .put(lr.regParam-> 0.1)
    val validator = new ValidatedPredictor[LogisticRegressionModel]
        lr, columns, trainPath
    )
    val (f1, auROC) = validator.trainingWithSummary //21
        .getOrElse((Double.NaN, Double.NaN))
    validator.stop //22
})
```

Tip

Stopping context

In our configuration, the estimators implement the interface that manages the life cycle of Spark context and sessions. Therefore, they are responsible for creating and stopping these contexts (line 22).

Validating the model

The validation of a model is a critical step in the generation of a model. The

validated predictive model, `ValidatedPredictor`, uses a similar interface as its sibling, `SimplePredictor`. However, it adds the functionality of the cross validation, implemented in the `CrossValidation` trait, as follows. The cross validation of a model is associated to the estimator used to train the model. Therefore, it implements the `ModelEstimator` trait (line 23):

```
trait CrossValidation[T <: Model[T]] extends ModelEstimator[T] {
    protected[this] val numFolds: Int          //24

    protected def apply(
        trainDf: DataFrame,
        stages: Array[PipelineStage],
        grid: Array[ParamMap]
    ): CrossValidatorModel= {           //25

        val pipeline= new Pipeline().setStages(stages ++
            Array[PipelineStage](estimator)) //26

        new CrossValidator() {
            .setEstimator(pipeline)
            .setEstimatorParamMaps(grid)
            .setEvaluator(new BinaryClassificationEvaluator)
            .setNumFolds(numFolds)
            .fit(trainDf)      //27
        }
    }

    protected def evaluate(
        trainDf: DataFrame,
        stages: Array[PipelineStage],
        grid: Array[ParamMap]
    ): Evaluator= this(trainDf, stages, grid).getEvaluator //28
}
```

The cross-validation component is created by specifying the number of folds (line 24). The `apply` method executes the cross-validation and returns a `CrossValidatorModel` instance (line 25). The `estimator` is appended to the existing data encoding `pipeline` (line 26).

The `CrossValidator` class generates the model through training, by invoking the `fit` method on the training set data frame (line 27). The minimum set of parameters required to generate the model are:

- The data frame containing the input observations, `trainDf`

- The sequence of stages to transform features (line and train the model: pipeline)
- The parameters for the grid search: grid

The evaluator of type `BinaryClassificationEvaluator` is used to collect information and statistics relevant to the execution of the training phase. It is optional.

The `evaluate` method produces an `evaluator` that encapsulates the computation of metrics associated with each prediction. It takes the same sequence of arguments as the method training method `apply` (line 28).

Note

`BinaryClassificationEvaluator`

This class evaluates a binary classifier by comparing raw predicted values with labeled data. The main method, `evaluate`, computes a metric of type `Double` to be specified by the user.

The methods of the trait `CrossValidation` are declared protected to reduce the scope of their access to the predictor class, `ValidatedPredictor`.

Grid search

Grid search consists of tuning the hyper-parameters of an estimator (regression, prediction, or classification). The key components of a grid search are:

- Estimator
- Set of hyper-parameters
- Search or sampling methods
- Scoring or model evaluation function
- Optional cross-validation design

The hyper-parameters are specific to each estimator, from the L₂

regularization factor in SVM to tolerance on reconstruction errors in K-means. The list of potential scoring functions includes *AuROC*, *F₁-score*, and loss function. A search method can be as simple as sampling different values from the set of hyper-parameters or as complex as a genetic algorithm.

Apache Spark MLlib provides some basic support for a grid search in the `ml.tuning` package. The class `ParamGridBuilder` creates a search grid as a set of hyper-parameters of type `Param` for each common type (`Long`, `Double`,). The parameters are used to drive the cross validation (`CrossValidator` and `CrossValidatorModel`).

As its name indicates, the `ValidatedPredictor` inherits from the `Predictor` class by adding cross validation capabilities, through the implementation of the `CrossValidation` trait (line 30). A validated predictor instance is created from:

- An estimator, `estimate`
- List of column names, `cols`
- A training set of observations, `trainSet`
- The number of folds, `numFolds` used in the cross-validation (line 29):

```
class ValidatedPredictor[T <: Model[T]] {  
    estimate: Estimator[T],  
    cols: Array[String],  
    trainSet: String,  
    override val numFolds: Int = 2          //29  
} extends Predictor[T](estimate, cols, trainSet)  
    with CrossValidation[T] {           //30  
  
    def apply(grid: Array[ParamMap]): CrossValidatorModel =  
        this(trainDf, stages, grid)           //31  
  
    def classify(grid: Array[ParamMap], testSet: String): DataF  
        this(trainDf, stages, grid).transform(csv2DF(testSet))  
  
    def evaluate(grid: Array[ParamMap]): Evaluator =  
        evaluate(trainDf, stages, grid)         //32  
  
    final def trainingSummary: Option[(Double, Double)] =  
        trainSummary(trainDf, stages)  
}
```

The methods `apply` for training (line 31) and `evaluate` for prediction (line 32) invoke the same methods in their parent class; these methods are independent from the cross-validation.

Let's use the methods of `ValidatedPredictor` to train a logistic regression model and compute the F1-measure and AuROC for the training set:

```
(for{
    trainPath<- ResourcesLoader.getPath(trainFile)
    testPath<- ResourcesLoader.getPath(testFile)
} yield {
    val lr = new LogisticRegression().setMaxIter(5)
        .setRegParam(0.1) //33
    val grid = Array[ParamMap](new ParamMap()
        .put(lr.maxIter-> 30)
        .put(lr.regParam-> 0.1))//34

    val validator = new ValidatedPredictor[LogisticRegressionModel]
        lr, columns, trainPath
    ) //35
    (validator, validator.classify(grid, testPath)) //36
})
.map {
    case (validator, output) => {
        val(f1, auROC) = validator.trainingWithSummary//37
            .getOrElse((Double.NaN, Double.NaN))

        validator.stop
        (f1, auROC)
    }
}
```

The training and test path are extracted using a resources loader, then the test case creates the logistic regression model (line 33) and the parameters for the grid search (line 34). Once the validator is properly instantiated (line 35), the test observations are classified using the `grid` search parameters (line 36). The metrics, `f1` and `auROC`, are computed through the `traningWithSummary` method described in the previous section (line 37).

Apache Spark and ScalaTest

Debugging an Apache Spark application using *ScalaTest* is rather trivial in the case of a single unit test.

The four key steps to write a unit test using ScalaTest are:

1. Specify your Spark context configuration *SparkConf*.
2. Create the Spark context.
3. Add test code related to your application.
4. Clean up resources (Spark context, Akka context, File handles...).

However, there are cases when you may need to create and close the Spark context used across multiple tests or on a subset of the tests. The challenge is to make sure that the Spark context is actually close when it is no longer needed. The ScalaTest methods `beforeAll` (line 38) and `afterAll` (line 39) of the trait `BeforeAndAfterAll` are used to manage the context life cycle of your test application:

```
trait MyTest extends BeforeAndAfterAll{
  self: Suite =>

  override def beforeAll: Unit = {    //38
    val sparkSession= SparkSession.builder()
      .appName(AppNameLabel)
      .config(new SparkConf())
      .set("spark.default.parallelism", "24")
      .set("spark.rdd.compress", "true")
      .set("spark.executor.memory", "32g")
      .set("spark.shuffle.spill", "true")
      .set("spark.shuffle.spill.compress", "true")
      .set("spark.io.compression.codec", "lzf")
      .setMaster("spark://server.domain:7077")
    ).getOrCreate()                      //40
    super.beforeAll
  }

  override def a fterAll: Unit = {    //39
    sparkSession.stop
    super.afterAll
  }
}
```

```
 }  
 }
```

The initialization of the Spark session consists of specifying the configuration for the sequence of tests. The Spark session is created only if it is not already defined within the scope of the test using the `getOrCreate` method (line 40).

Extending Spark

As an open source platform, Apache Spark MLlib can be easily customized, and extended to address specific problems related to requiring the implementation of new statistics, mathematical, or machine learning solutions.

The purpose of this section is to create a Kullback-Leibler divergence as a Spark *evaluator*. The implementation follows two steps:

- Describe and implement the Kullback-Leibler divergence using Spark 2.0
- Convert the class into a Spark *evaluator*

Kullback-Leibler divergence

The *Kullback-Leibler* has been briefly introduced in [Chapter 5, Dimension Reduction – Divergences](#) in the context of evaluating the similarity of the frequencies distribution between two datasets. The Kullback-Leibler divergence is a concept borrowed from *information theory* and is commonly associated with *Information Gain*. The Kullback-Leibler divergence is also known as the *relative entropy or information divergence* between two distributions. It measures the dissimilarity of the distribution of random values (that is, probability density function).

Let's consider two continuous probability distributions P and Q with density functions p and q , the Kullback-Leibler divergence from the prior probability distribution Q to the posterior probability distribution P is defined as [17:08]:

$$D_{KL}(P \| Q) = - \int_{-\infty}^{+\infty} p(x) \cdot \log \frac{p(x)}{q(x)}$$

The formula can be interpreted as the *information lost* when the distribution Q is used to approximate the distribution P .

Note

Kullback-Leibler measure

Although Kullback-Leibler divergence is not actually a metric, it can be used to evaluate the difference of probabilistic measures.

It is obvious that the computation is not symmetric:

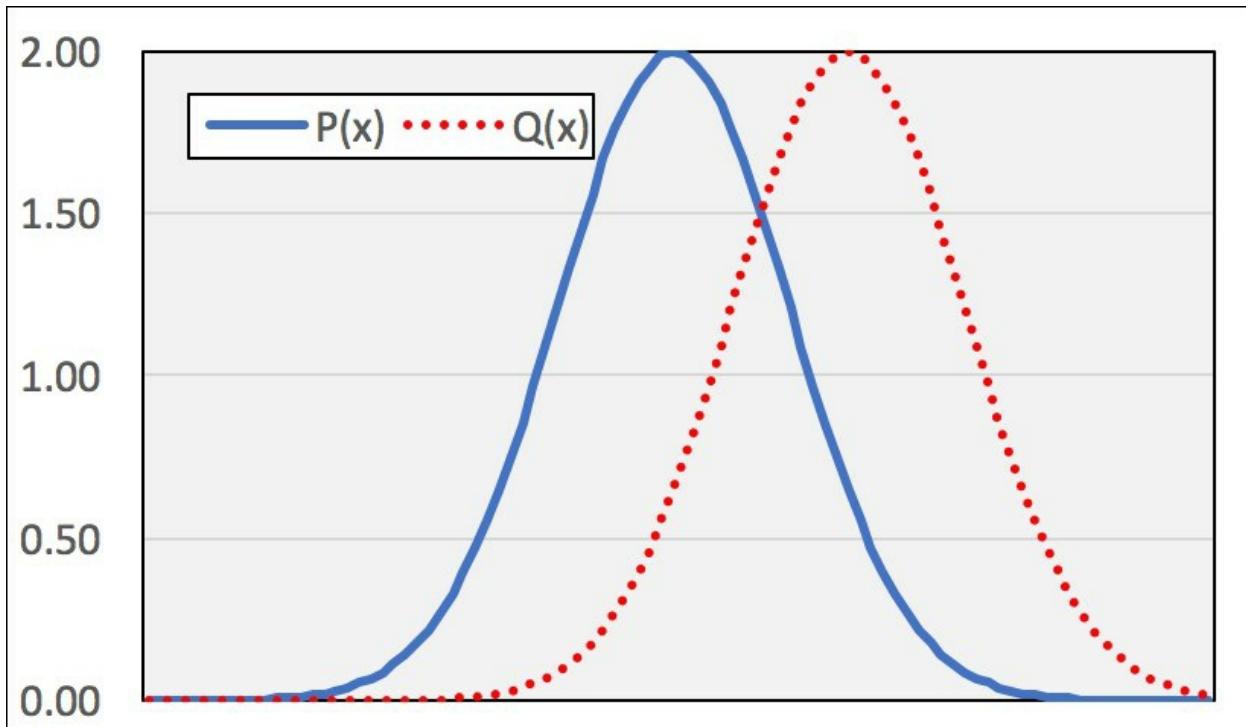
$$D_{KL}(P \| Q) \neq D_{KL}(Q \| P)$$

The Kullback-Leibler divergence is used to compute the *Mutual Information*,

which is one of the feature extraction methods. It is part of the family of F-Divergences.

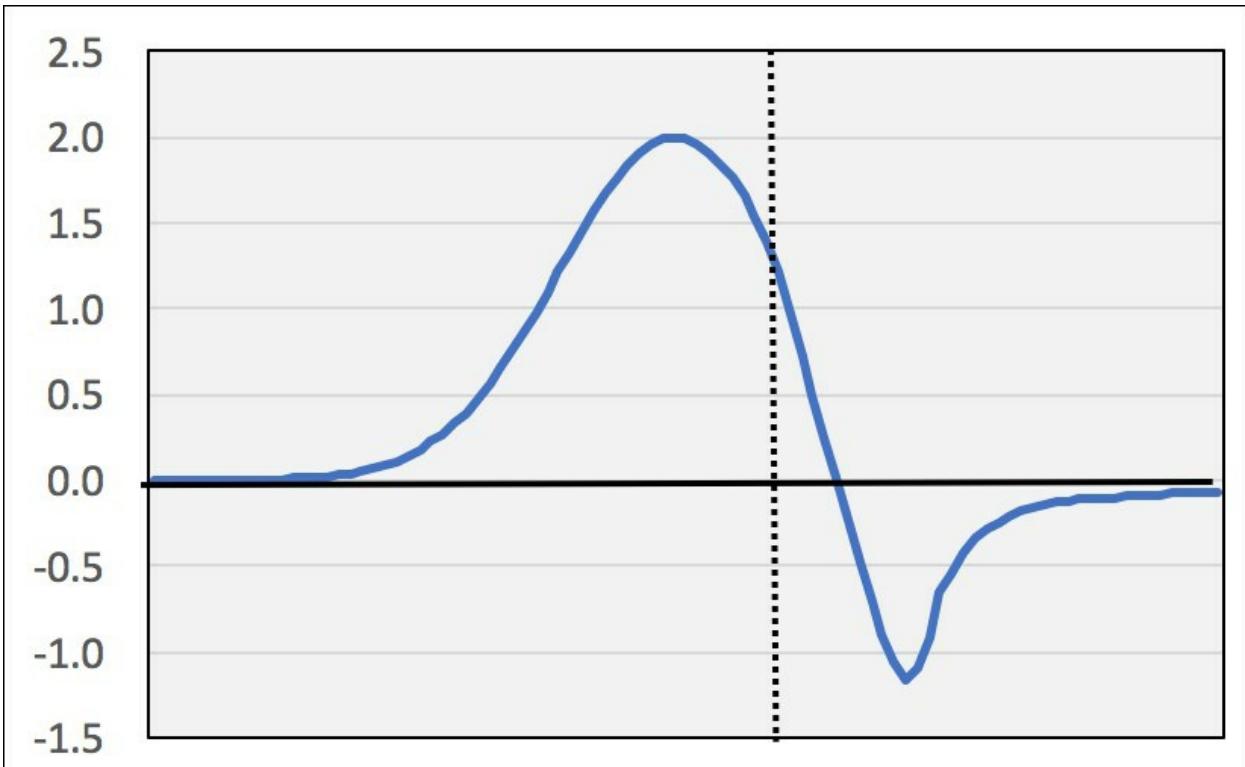
The objective of this section is to implement Kullback-Leibler divergence in Spark in order to evaluate the similarity between a probability distribution and given dataset.

Let's illustrate the Kullback-Leibler divergence with Gaussian distributions:



Mixture model of two Gaussian distribution with identical variance

The following plot illustrates the computation of the Kullback-Leibler divergence for two Gaussian distributions that differ only by their mean:



Kullback-Leibler divergence of two Gaussian distribution with identical variance

Tip

Kullback-Leibler for feature extraction

The Kullback-Leibler is used to estimate the difference between the distribution of the value of features. Two features with small KL divergence are very likely redundant. In practice, the mutual information exclusion, built using the KL divergence is more commonly used to reduce the dimension of a model [17:09].

Implementation

Let's consider a set of continuous probability distributions as defined by their density functions. The test case consists of finding the probability distribution that is the best possible fit to a given dataset.

The Kullback-Leibler divergence is implemented by the class `KullbackLeibler` which takes an iterable collection of probability density functions, `pdfs` as argument (line 1):

```
class KullbackLeibler(  
    pdfs: Iterable[(String, Double => Double)]    //1  
) (implicit sparkSession: SparkSession) {           //2  
    val sparkContext = sparkSession.sparkContext  
  
    type DataIter = Iterator[(Double, Double)]  
    type DataSeq = Seq[(Double, Double)]  
  
    type BrdcastData = Iterable[(String, Double=>Double)]  
  
    def kl(data: DataSet): Seq[Double] = kl(data.rdd) //3  
  
    def fittestDistribution(fileName: String): String = { //4  
        val rdd = sparkContext.textFile(fileName).map(_.split(",")).  
            .map(arr => (arr.head.toDouble, arr.last.toDouble))  
        )  
        kl(rdd).zip(pdfs).minBy(_._1)._2._1  
    }  
  
    def kl(data: RDD[(Double, Double)]): Seq[Double]  
}
```

This implementation of the Kullback-Leibler divergence assumes that a Spark session has already been implicitly created (line 2).

The public method `kl` computes the KL divergence for each of the `pdfs` against a given dataset (line 3). It returns a sequence of KL values.

The more generic method, `fittestDistribution`, extracts the fittest distributions among the `pdfs`, given a dataset residing in a HDFS file,

`filename` (line 4). The actual computation of the KL divergence is implemented in the private method `kl`.

The method takes a RDD of pair $\{x, y\}$ values as argument (line 5). The value y is to be compared with density function of each of the probability distributions to be evaluated:

```

def kl(data: RDD[(Double, Double)]): Seq[Double] = { //5
    val pdfs_broadcast =
        sparkContext.broadcast[BroadcastData](pdfs) //6

    val kl = data.mapPartitions((it: DataIterator) => //7
        pdfs_broadcast.value.map { //8
            case (key, pdf) =>
                def exec(xy: DataSeq, pdf: Double=>Double): Double = {
                    -xy./(0.0) {
                        case (s, (x, y)) => {
                            val px = pdf(x)
                            val z = if (abs(y) < Eps) px/Eps else px/y
                            s + px * (if (z < Eps) LogEps else log(z))
                        }
                    }
                }
            }

            (key, exec(it.toSeq, pdf)) //9
        }.iterator
    ).collect

    (0 until kl.size by pdfs.size)
        .:(Array.fill(pdfs.size)(0.0))((sum, n) => { //10
            (0 until pdfs.size).foreach(j => sum.update(j, kl(n+j)._2))
            sum
        }).map(_ / kl.length) //11
}

```

The computation of the Kullback-Leibler divergence is parallelized by invoking `mapPartitions` that executes the method `exec` on each partition (line 6). The input `data` of type `DataSeq` is converted into an iterator of type `DataIterator` (line 7) assigned to each partition. A copy of the collection of probability density functions, `pdfs`, is duplicated and broadcast to all the partitions. (line 6):

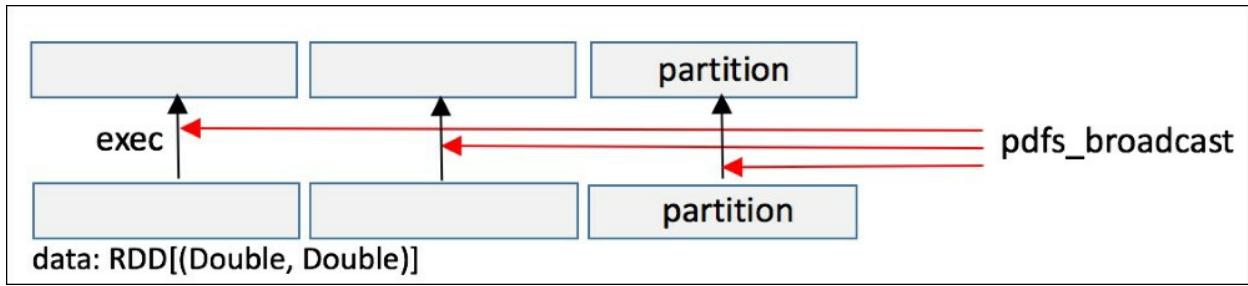


Illustration of `mapPartitions` with broadcasting

The Spark executors running on data nodes retrieve the probability density functions by calling the `value` method (line 8):

Tip

Partitions higher order method

The `RDD` class methods `mapPartitions`, `mapPartitionsWithIndex`, and `foreachPartition` generate an `RDD` by applying a function on each partition of the input `RDD`. This set of higher order methods can be an excellent alternative to `PairRDDFunctions` such as `aggregateByKey`, `countByKey`, `groupByKey`, `coGroup`, or `combineByKey`. `PairRDDFunctions` methods produce a significant read and write shuffling.

The method `exec` computes the Kullback-Leibler divergence between each of the probability density function `pdf` and the `RDD` partition `xy` (line 9). The result of the computation of KL divergence on each partition is collected and then reduced on the Spark driver, using a `foldLeft` (line 10). Finally, the KL divergence is normalized by the number of `pdfs` (line 11).

Let's put the Kullback-Leibler computation into practice, comparing multiple Gaussian distribution with identical variance. The objective is to evaluate the impact of the mean of the Gaussian distribution on the KL divergence. The steps are:

- Generate 5,000 data points sampled from a Gaussian distribution, `NormalDistribution`, with a given mean μ and standard deviation (line 12)

- Compute the KL divergence between the sample and Gaussian distributions with identical variance, but increasing value of the mean x , from μ to $\mu + 5.0$ (line 13)

The following code snippet implements the computation of the Kullback-Leibler divergence for a set of Gaussian distribution:

```

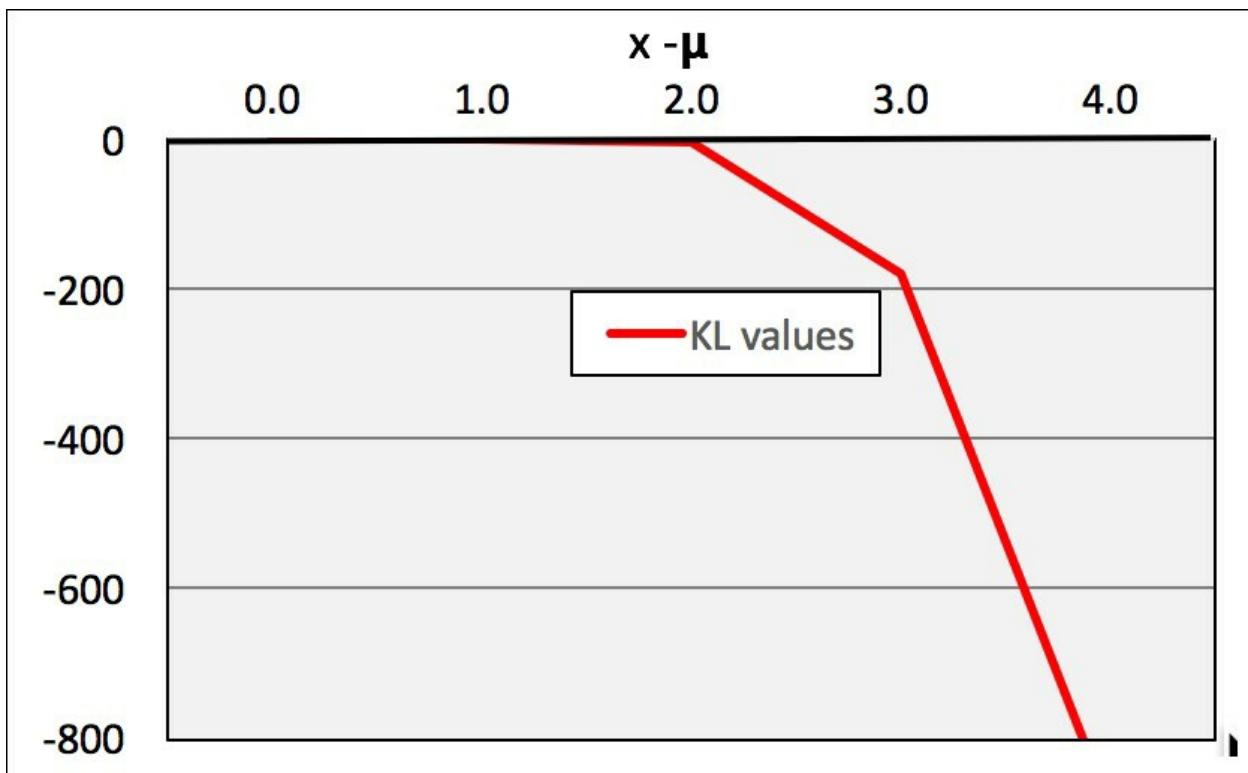
val normalGenerator = new NormalDistribution
val normalData= toDSPairDouble(5000)((n: Int) => {
    val x = n.toDouble * 0.001
    (x, normalGenerator.density(x)) //12
} )

val mu = 2.0
val Inv2PI = 1.0 / sqrt(2.0 * PI)
val pdf= (x: Double) => {
    val z = x -mu
    Inv2PI * exp(-z*z)
}

Val kullbackLeibler = KullbackLeibler(s"Normal mu=$mu", pdf)
kullbackLeibler.kl(normalData).head //13

```

The following plot illustrates the impact of the divergence of the mean of Gaussian distribution on the KL divergence:



Kullback-Leibler divergence values for Gaussian distribution of same variance and varying means

As expected, the KL divergence increases significantly as the difference between mean x of the Gaussian distribution and the mean μ of the sample increases.

Kullback-Leibler evaluator

In this last step, we convert our class Kullback-Leibler as an evaluator, by implementing the following abstract components of the `Evaluator` trait:

- Unique identifier related to the evaluator `uid`
- Method to copy the parameters map `copy`
- Method to evaluate a given dataset

The first step is to update the declaration of the `KullbackLeibler` class:

```
class KullbackLeibler(  
    pdfs: Iterable[(String, Double => Double)],  
    override val uid: String = "KullbackLeibler0.99.2"  
) (implicit sparkSession: SparkSession) extends Evaluator {
```

For the sake of simplicity, we will provide a default unique identifier.

The implementation of the method `copy` is rather simple: It uses the `defaultCopy` method of the `Evaluator` trait:

```
override def copy(extra: ParamMap): KullbackLeibler =  
    defaultCopy(extra)
```

The last step consists of implementing the `evaluate` method:

```
override def evaluate(dataset: Dataset[_]): Double = { //14  
    dataset match {  
        case input: Dataset[(Double, Double)]  
            k1(input.rdd).head //15  
        case _ => Double.NaN  
    }  
}
```

The method evaluates a dataset against a probability distribution. The method throws an `IllegalArgumentException` exception if the constructor defines more than one probability density function (line 14). Finally, the method invokes the `k1` method to compute the KL divergence (line 15).

Tip

KL divergence as an estimator

It is conceivable to define the KL divergence as an estimator. In this case, the KL divergence acts as a regressive model, measuring the fitness of a probability distribution to a given dataset.

Streaming engine

The Apache Spark streaming component is an integral part of the framework. It does not require any specific installation or configuration. Apache Spark In-memory capabilities are a good solution to problems dealing with large scale real-time processing.

There are numerous articles and books related to the Apache Spark streaming library. This section introduces some basic concepts in the context of machine learning algorithms.

Why streaming?

Many applications require real-time or pseudo real-time processing of data from weather reporting, automated manufacturing processing, ATMs, advertising targeting, to financial markets analysis. The implementation of such systems is challenging because of its stringent requirements:

- **Low latency:** Response time is sometimes computed in milliseconds
- **Continuous traffic:** Never ending stream of data
- **No downtime:** Fault-tolerant design to avoid loss of information

It is not uncommon that these requirements are formalized into contractual obligations such as a **service level agreement (SLA)** with significant monetary implications.

Machine learning techniques are not immune from the constraints of real-time processing. Some models require a constant partial or full update as the most recent observations may have an outsize impact on the accuracy of the classifier or regression algorithm. Predictive models or evaluators are commonly used in real-time environments.

Tip

Back-pressure

Apache Spark 2.0 supports back-pressure management from consumer to producer to prevent buffer overflows and avoid throttling. The Spark back-pressure model is derived from Akka reactive streams. The Spark configuration parameters regarding back-pressure are:

- `spark.streaming.backpressure.enabled` Enable/disable streaming internal backpressure
- `spark.streaming.backpressure.initialRate` Initial rate at which a receiver receives the first batch

Batch and real-time processing

The batch processing of tasks is the process of executing a sequence of tasks that implements one or multiple processing units. Contrary to long held perceptions, batched tasks can be:

- Mission critical
- Executed either sequentially and in parallel
- Scheduled
- Highly distributed
- Fault-tolerant
- Complex

Real-time data processing is a design of executing tasks on ever-changing, continuous data. Contrary to batch data processing, there is an expectation of responsiveness measured as latency. Real-time data processing is not literally applicable in the real world because of the constraints of computing resources. It is approximated by breaking down real-time data streams into mini-batches that can be executed within a time window usually governed by a service level agreement.

Note

Spark fault tolerance

Spark may decide to rerun a task on another node if:

- The current node crashes
- The current node is slow
- If a value needs to be cached, if cache has run out of memory

A discretized stream is specific to the source of the streamed data. The class `StreamingContext` is responsible for read data and converts into a stream of RDDs of type `InputDStream`.

Apache Spark 2.0+ streaming library, supports the following adapters:

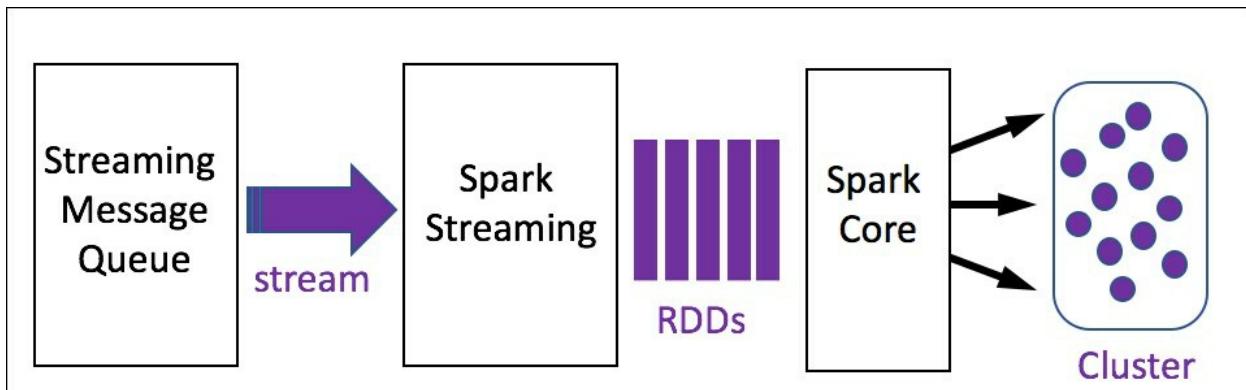
- `StreamingContext.fileStream`: Input stream from a HDFS files
- `StreamingContext.textFileStream`: Input stream from HDFS text files
- `StreamingContext.queueStream`: Input stream from a queue of RDDs
- `StreamingContext.receiverStream`: Input stream from a generic receiver
- `StreamingContext.socketStream`: Input stream of type `ReceiveInputDStream` from a socket
- `KafkaUtils.createStream`: Input stream of type `ReceiveInputDStream` ingested from a Kafka broker
- `FumeUtils.createStream`: Input stream ingested from Fume sinks

Architecture overview

As described in the previous section, absolute real-time processing is not practical in the realm of the physical, computing world. A common approach to get around the limitation of computing power is to break down streamed data into mini-batches that can be queued and processed in parallel. This is the approach taken by Apache Spark. A simple and typical data flow has the following components [17:10]:

1. A real-time publisher-subscriber messaging platform organizes input data into channels and topics to which downstream applications subscribe to. Apache Kafka, Fume, and Impala are commonly used streaming messaging systems.
2. The Spark streaming engine consumes events or data and converts into RDDs.
3. The Spark core engine distributes and processes RDDs across multiple hosts and executors within a cluster.

The following diagram illustrates the various processing units of Apache Spark streaming:



Simplified view of data streaming using Apache Spark

Discretized streams

Apache Spark 2.0+ streaming library structures data streams as mini-batches of RDDs known as **discretized streams (DStreams)**. A streaming job can be easily represented as the following hierarchical decomposition:

- A stream job contains multiple discretized streams
- A discretized stream contains multiple RDDs
- An RDD contains multiple observations

Note

MLlib and streaming

Spark 2.x MLlib package has a sub-set of estimators or models with predefined streaming capabilities such as

`StreamingLogisticRegressionWithSGD` or `StreamingKMeans`.

Use case – continuous parsing

Being the first data processing unit (or transform) in most machine learning related pipelines, the parsing of a stream of observations has the potential to become a bottleneck, creating havoc in downstream pipeline stages.

Let's create a simple parser for streaming data, `StreamingExtractor`. The objective is to extract an input of type `T` into a sequence of fields of type `U`.

The extracting function, `extractor`, is supplied as an argument of the streaming extractor (line 1):

```
class StreamingExtractor[T, U] (
    extractor: T => Seq[U] //1
) (implicit sparkContext: SparkContext) extends Serializable {

    def extract(
        inputStream: InputDStream[T],
        f: RDD[Seq[U]] => Unit
    ): Unit =//2
        inputStream.map(extractor(_)).foreachRDD(f(_)) //3

    def textFile(fileName: String): RDD[String] =
        sparkContext.textFile(fileName)
}
```

The method `extract` (line 2) takes two arguments:

- A discretized stream of parameterized type `InputDStream[T]`
- A function `f` to process each batch of RDDs

The method applies the `extractor` for each RDDs of the input stream, and then invokes the parameter function `f` to each sequence of fields.

Let's consider the following streaming extractor. It converts an input (text line) into a sequence of fields of type `Double` (line 4):

```
val extractor = new StreamingExtractor[String, Double] (
    _.split(",") .map(_.toDouble) //4
```

```
)
```

The input stream is actually defined as a Scala mutable queue of RDDs of elements of type String. The `input` stream is created by loading the content of multiple files, `fileNames` (line 5). The queue is actually built using a left fold (line 6):

```
val input = fileNames.map(fileName =>      //5
    extractor.textFile(s"resources/$fileName"))
.:/ (mutable.Queue[RDD[String]]() ) (
    (qStream, rdd) =>qStream += rdd  //6
)
```

The discretized stream, `dStream`, is generated as a queue stream using the streaming context (line 7):

```
val dStream=
    streamingContext.queueStream[String](input,true) //7

extractor.extract(dStream, (rddStr: RDD[Seq[String]]) =>
    logger.info(rddStr.collect.mkString("\n"))           //8
)

streamingContext.start//9

streamingContext.stop(true, true)
streamingContext.awaitTerminationOrTimeout(100L) //10
```

The post-processing function (second argument of the `extract` method) simply wrote the fields into a log file (line 8). Practically, a filtering or data cleansing method would be used after each input line is decomposed into fields.

The extraction of streamed data does not start until the context is started (line 9). In true Apache Spark fashion, the different processing units that execute on the data stream, are defined as a **lazy direct acyclic graph (DAG)**.

Finally, the current execution thread has to be blocked until either the streaming process is completed or the execution time exceeds the predefined time out (line 10).

Checkpointing

As mentioned in the introduction of this section, streaming engines require a solid fault tolerance strategy to recover from unexpected errors during streaming. The previous implementation of streaming extractor does not take into account that some of the input data may not be available.

One option is to set multiple checkpoints along the data flow so any recovery strategy will restart the computation from the state saved at the last successful checkpoint. In our case, a checkpoint is set if one of the CSV input files fails to load (line 11). If the execution fails, a new streaming context, `newStreamingContext` is created with a state initialized from the checkpoint (line 12), when the application is restarted. The original streaming context is used if the checkpoint information cannot be successfully retrieved (line 13).

The original streaming context, `streamingContext`, is preserved as long as no failure occurs at any checkpoint (line 14):

```
...
Val checkpointDir = "resources/checkpoint"
...

val newStreamingContext =
  if( fileNames.exists(!new java.io.File(_).exists)) //11
    StreamingContext.getOrCreate(checkpointDir,           //12
      () =>streamingContext)//13
  else {
    streamingContext.checkpoint(checkpointDir) //14
    streamingContext
  }
}

newStreamingContext.start

newStreamingContext.stop(true, true)
newStreamingContext.awaitTerminationOrTimeout(streamer.timeOut)
```

Tip

Useful checkpointing

Although instructive, the selection of the checkpoint in the previous example is not optimized: The test for the existing input file to be streamed should have been done prior to creating the first steaming context.

Checkpoints have to be used judiciously because they require reliable storage and extra processing time. Moreover, the storage for the checkpoint data has to be fault-tolerant. Any failure to access the checkpoint stored data makes the process irrelevant.

Performance evaluation

There are numerous configuration parameters that can be set to optimize the execution of Spark jobs. The topic of tuning and the resolution of performance bottlenecks on Spark clusters deserves at the minimum, a dedicated chapter.

This section does not address Mesos-and Yarn-specific configurations as they are not related to machine learning and are beyond the scope of this book [7:11].

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 more than 60 configuration parameters as of the last count. Fortunately, the majority of those parameters have relevant default values.

However, there are a 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 generally a good compromise.
- Number of concurrent tasks to use across all the partitions for shuffle-related operations, they use keys such as `reduceByKey`:
`spark.default.parallelism`. The recommended formula is *parallelism = total number of cores x 2.5*. The value of the parameter can be overridden with the `spark.reducer.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.
- Memory of the GC young generation objects by tuning
`spark.memory.fraction`. Reserve the appropriate amount of memory for cached objects using `spark.memory.storageFraction`.
- The rate limit for receivers `spark.streaming.receiver.maxRate` has to be set up in case the back pressure is not enabled.

Performance considerations

This section 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 2.0+:

- 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.
- Estimate the memory consumption for very large objects before caching using the `SizeEstimator.estimate` method.
- Prefer arrays and primitive types instead of Java or Scala standard library.
- Use the *G1GC* JVM garbage collector and set the extra JVM parameters `G1HeapRegionSize` appropriately.
- Look into defining your own partitioning function to reduce large key-value pair datasets. The convenience of `reduceByKey`, `aggregateByKey`, or `groupByKey` has its price. The ratio of number of partitions to number of cores has an impact on the performance of a reducer-using key. Using partitions methods such as `mapPartitions` can be a good alternative to pair RDD higher order functions.
- 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.
- Optimize JVM execution by specifying JVM command-line arguments using the parameter `spark.executor.extraJavaOptions` for the executors on data nodes and the parameter `spark.driver.extraJavaOptions` for the driver process.
- Enable back pressure control by setting `spark.streaming.backpressure.enabled` as true.
- Rely on shared *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
- 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 library for pseudo-real time computation or infinite loop
- Apache Spark leverages Scala functional capabilities and a large number of open source Java libraries
- Spark can leverage the *Mesos* or *Yarn* cluster managers, 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 and/or Scala.
- Tuning the performance of an Apache Spark cluster can be daunting, at times. It is worthwhile deploying applications on clusters in which the number and characteristic of the data nodes does change significantly over time: developers underestimate the number of configuration parameters that require tuning for each new deployment configuration.

Summary

This chapter should be regarded as an invitation to explore the capabilities of the Apache Spark framework in a single host than a large deployment environment.

Beyond the introduction to the key components of the Apache Spark framework and the concept of resilient distributed datasets, data frames and datasets, we learned how to leverage RDDs for data clustering using the K-means algorithm and create data frames and reusable ML pipelines for encoding observations and training models.

We also experimented with extending the Spark library with new functionality such as the Kullback-Leibler divergence and leveraging the Spark streaming library for pseudo-real-time data processing as applied to the data parsing.

Appendix A. Basic Concepts

Machine learning algorithms make significant use of linear algebra and optimization techniques. Describing the concept and the implementation of linear algebra, calculus, and optimization algorithms in detail would have added significant complexity to the book and distracted the reader from the essence of machine learning.

The appendix lists the basic set of elements of linear algebra and optimization mentioned throughout the book. It also summarizes the coding practices and acquaints the reader with basic knowledge in financial analysis.

Scala programming

Here is a partial list of coding practices and design techniques used throughout the book.

List of libraries and tools

The precompile Scala for Machine learning is `ScalaML-2.11-0.99.2.jar` located in directory `$ROOT/project/target/scala-2.11`.

Here is the complete list of recommended tools and libraries used in creating and running the Scala for machine learning:

- Java JDK 1.7 or 1.8 for all chapters
- Scala 2.11.8 or higher for all chapters
- Scala IDE for Eclipse 4.0 or higher
- IntelliJ IDEA Scala plug in 13.0 or higher
- Sbt 0.13.1 or higher
- Apache Commons Math 3.5 or higher for [Chapter 3](#), *Data Pre-processing*, [Chapter 4](#), *Unsupervised Learning*, and [Chapter 12](#), *Kernel Models and SVM*
- JFChart 1.0.1 in Chapter 1, *Getting Started*, [Chapter 2](#), *Data Pipelines*, [Chapter 5](#), *Dimension Reduction*, and [Chapter 9](#), *Regression and Regularization*
- Itib CRF 0.2 (including LBGFS and Colt libraries) in [Chapter 7](#), *Sequential Data Models*
- LIBSVM 0.1.6 in [Chapter 8](#), *Monte Carlo Inference*
- Akka framework 2.3 or higher in [Chapter 16](#), *Parallelism in Scala and Akka*
- Apache Spark/MLlib 2.0 or higher in [Chapter 17](#), *Apache Spark MLlib*
- Apache Maven 3.5 or higher (required for Apache Spark 2.0 or higher)

Tip

Note for Spark developers

The version of the Scala library and compiler JAR bundled with the assembly JAR for Apache Spark contains a version of the Scala standard library and compiler JAR file that may conflict with an existing Scala library (that is, Eclipse default ScalaIDE library).

The `lib` directory contains the following JAR files related to the 3rd part libraries or frameworks used in the book: colt, CRF, LBFGS, and LIBSVM

Code snippets format

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

The following are the comments:

```
/**  
This class is defined as ...  
*/  
// MathRuntime exception has to be caught here!
```

The validation of class parameters and method arguments are as follows:

```
class Columns(cols: List[Int] . ...) {  
require (cols.size > 0, "Cols is empty")
```

The code for class qualifiers such as `final` and `private` are as follows:

```
final protected class MLP[T: ToDouble] ...
```

The code for method qualifiers and access control (`final` and `private`) is as follows:

```
final def inputLayer: MLPLayer  
private def recurse: Unit  
private[this] val eps = 1e-6
```

The code for serialization is as follows:

```
class Config extends Serializable {...}
```

The code for validation of partial functions is as follows:

```
val pfn: PartialFunction[U, V]  
pfn.isDefinedAt(u)
```

The validation of intermediate state is as follows:

```
assert( p != None, " ... ")
```

The following are the Java style exceptions:

```
try { ... }
catch { case e: ArrayIndexOutOfBoundsException => ... }
if (y < EPS)
    throw new IllegalStateException( ... )
```

The following are the Scala style exceptions:

```
Try(process(args)) match {
  case Success(results) => ...
  case Failure(e) => ...
}
```

The following are the nonessential annotation:

```
@inline def mean = ...
@implicitNotFound("Conversion $T to Array[Int] undefined")
@throws(classOf[IllegalStateException])
```

The following is the logging and debugging code:

```
m_logger.debug( ... )
Console.println(...)
```

Auxiliary and nonessential methods

Best practices

Let's walkthrough the practices in detail.

Encapsulation

One important objective in creating an API is to reduce the access to the supporting or helper class. There are two options to encapsulate helper classes:

- **Package scope:** The supporting classes are first-level class with protected access
- **Class or object scope:** The supported classes are nested in the main class

The algorithms presented in the 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 as scope (`protected`):

```
protected class A[T] (val x: X, val y: Y,...) { ... }
object A {
  def apply[T] (x: X, y: Y, ...): A[T] = new A(x, y,...)
  def apply[T] (x: , ...): A[T] = new A(x, y0, ...)
}
```

For example, the `SVM` class that implements the support vector machine is defined as follows:

```
final protected class SVM[T: ToDouble] (
  config: SVMConfig,
  xt: Vector[Array[T]],
  expected: DblVec)
extends ITtransform[Array[T], Array[Double]] {
```

The companion object, `SVM`, is responsible for defining all the constructors (instance factories) relevant to the protected class `SVM`:

```
def apply[T: ToDouble] (
    config: SVMConfig,
    xt: Vector[Array[T]],
    expected: DblVec): SVM[T] =
new SVM[T](config, xt, expected)
```

Companion objects versus case classes

In the preceding example, the constructors are explicitly defined in the companion object. Although the invocation of the constructor is very similar to the instantiation of case classes, there is a major difference: the scala compiler generates several methods to manipulate an instance as regular data (`equals`, `copy`, and `hash`).

Case classes should be reserved for single state data objects (no methods).

Enumerations versus case classes

It is quite common to read or hear discussions regarding the relative merit of enumerations and pattern matching with case classes in Scala. [A:1] As a very general guideline, enumeration values can be regarded as lightweight case classes or case classes can be considered as heavyweight enumeration values.

Let's take an example of Scala enumeration, which consists of evaluating the uniform distribution of `scala.util.Random`:

```
object A extends Enumeration {
    type TA = Value
    val A, B, C = Value
}

import A._
val counters = Array.fill(A.maxId+1) (0)
(0 until 1000).foreach(_ => nextInt(10) match {
    case 3 => counters(A.id) += 1
    ...
})
```

```
    case _ => ...
})
```

The pattern matching is very similar to the Java switch statement.

Let's consider the following example of pattern matching using case classes that select a mathematical formula according to the input:

```
package AA {
  sealed abstract class A(val level: Int)
  case class AA extends A(3) { def f = (x:Double) => 23*x}
  ...
}

import AA._
def compute(a: A, x: Double): Double = a match {
  case a: AA => a.f(x)
  ...
}
```

The pattern matching is performed using the default equals method, where the byte code is automatically generated for each case class. This approach is far more flexible than the simple enumeration at the cost of extra computation cycles.

The advantages of using enumerations over case classes are as follows:

- Enumerations involve less code for a single attribute comparison
- Enumerations are more readable, especially for Java developers

The advantages of using case classes are as follows:

- Case classes are data objects and support more attributes than enumeration IDs
- Pattern matching is optimized for sealed classes as the Scala compiler is aware of the number of cases

Briefly, you should use enumeration for single value constant and case classes for matching data objects.

Overloading

Contrary to C++, Scala does not actually overload operators. Here is the meaning of the very few operators used in code snippets:

- `+=` adds an element to a collection or container
- `+` sums two elements of the same type

Design template for immutable classifiers

The machine learning algorithms described in *Scala for Machine Learning* use the following design pattern and components:

- The set of configuration and tuning parameters for the classifier is defined in a class inheriting from `Config` (that is, `SVMConfig`).
- The classifier implements a monadic data transformation of type `ITransform` for which the model is implicitly generated from a training set (that is, `SVM[T]`). The classifier requires at least three parameters, which are as follows:
 - A configuration for the execution of the training and classification tasks
 - An input data set `xt` of type `Vector[T]`
 - A vector of labels or expected values
- A model of type inherited from `Model`. The constructor is responsible for creating the model through training (that is, `SVMMModel`)

For example, the key components of the support vector machine package are the classifier `SVM`:

```
final protected class SVM[T: ToDouble] (
  config: SVMConfig,
  xt: Vector[Array[T]],
  val labels: DblVec)
extends ITransform[Array[T], Array[Double]] {

  val model: Option[SVMMModel] =
    override def |>: PartialFunction[Array[T], Array[Double]] =
    ...
}
```

The training set is created by combining or zipping the input dataset `xt` with

the labels or expected values expected. Once trained and validated, the model is available for prediction or classification.

The design has the main advantage of reducing the lifecycle of a classifier: a model is either defined, available for classification, or is not created. The configuration and model classes are implemented as follows:

```
case class SVMConfig(  
    formulation: SVMFormulation,  
    kernel: SVMKernel,  
    svmExec: SVMExecution) extends Config  
  
class SVMModel(val svmmode1: svm_model) extends Model
```

Note

Implementation considerations

The validation phase is omitted in most of the practical examples throughout the book for the sake of readability.

Utility classes

Let's look at the utility classes in detail.

Data extraction

The CSV file is the most common format used to store historical financial data. It is the default format for importing data used throughout the book. The data source relies on the `DataSourceConfig` configuration class as follows:

```
case class DataSourceConfig(  
    path: String,  
    normalize: Boolean,  
    reverseOrder: Boolean,  
    headerLines: Int = 1)
```

The parameters for the `DataSourceConfig` class are as follows:

- `path`: 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 files were CSV files
- `normalize`: Flag to specify whether the data must be normalized [0, 1]
- `reverseOrder`: Flag to specify whether the order of the data in the file should be reversed (that is, time series) if `true`
- `headerLines`: Number of lines for the column headers and comments

The data source, `DataSource` implements data transformation of type `ETransform` using an explicit configuration, `DataSourceConfig` as described in the *Monadic data transformation* section of [Chapter 2, Data Pipelines](#):

```
type Fields = Array[String]  
type U = List[Fields => Double]  
type V = Vector[Array[Double]]  
  
final class DataSource(  
    config: DataSourceConfig,  
    srcFilter: Option[Fields => Boolean] = None)  
extends ETransform[U, V](config) {
```

```

override def |> : PartialFunction[U, Try[V]] {
  ...
}

```

The `srcFilter` argument specifies the filter or condition of some of the row fields to skip the dataset (that is, missing data or incorrect format). Being an explicit data transformation, constructor for the `DataSource` class has to initialize the input type `U` and output type `V` of the extracting method `|>`. The method takes the extractor from a row of literal values to double floating-point values:

```

override def |> : PartialFunction[U, Try[V]] = {
  case fields: U if(!fields.isEmpty) =>load.map(data =>{ //1
    val convert = (f: Fields =>Double) => data._2.map(f(_))

    if( config.normalize) //2
      fields.map(t => new MinMax[Double](convert(t)) //3
          .normalize(0.0, 1.0).toArray ).toVector //4
    else fields.map(convert(_)).toVector
  })
}

```

The data is loaded from the file using the helper method `load` (line 1). The data is normalized if required (line 2) by converting each literal to a floating-point value using an instance of the `MinMax` class (line 3). Finally, the `MinMax` instance normalizes the sequence of floating point values (line 4).

The `DataSource` class implements a significant set of methods that are documented in the source code available online.

Financial data sources

The examples in the book rely on three different sources of financial data using the CSV format as follows:

- `YahooFinancials`: This is used for Yahoo schema for historical stock and ETF price
- `GoogleFinancials`: This is used for Google schema for historical stock and ETF price

- **Fundamentals:** This is used for fundamental financial analysis ration (CSV file)

Let's illustrate the extraction from data source using `YahooFinancials` as an example:

```
object YahooFinancials extends Enumeration {
    type YahooFinancials = Value

    val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME, ADJ_CLOSE = Value
    val adjClose = ((s:Array[String]) =>
        s(ADJ_CLOSE.id).toDouble) //5
    val volume = (s: Fields) => s(VOLUME.id).toDouble
    ...
    def toDouble(value: Value): Array[String] => Double =
        (s: Array[String]) => s(value.id).toDouble
}
```

Let's look at an example of application of `DataSource` transformation: loading stock historical data from Yahoo finance site. The data is downloaded as a CSV formatted file. Each column is associated to an extractor function (line 5):

```
val symbols = Array[String] ("CSCO", ...) //6
val prices = symbols
    .map(s => DataSource(s"$path$s.csv", true, true, 1)) //7
    .map(_ |> adjClose) //8
```

The list of stocks for which the historical data has to be downloaded is defined as an array of symbols (line 6). Each symbol is associated to a CSV file (that is, `CSCO => resources/CSCO.csv`) (line 7). Finally, the `YahooFinancials` extractor for the `adjClose` price is invoked (line 8).

The format for the financial data extracted from the Google financial pages are similar to the format used with the Yahoo financial pages:

```
object GoogleFinancials extends Enumeration {
    type GoogleFinancials = Value
    val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME = Value
    val close = (s:Array[String]) => s(CLOSE.id).toDouble //5
    ...
}
```

The `YahooFinancials`, `GoogleFinancials`, and `Fundamentals` classes implement a significant number of methods that are documented in the source code available online.

Documents extraction

The `DocumentsSource` class is responsible for extracting the most date, title, and content of a list of text documents or text files. The class does not support HTML documents. The `DocumentsSource` class implements a monadic data transformation of type `ETransform` with an explicit configuration of type `SimpleDateFormat`:

```
type U = Option[Long] //2
type V = Corpus[Long] //3
class DocumentsSource(
    dateFormat: SimpleDateFormat,
    val pathName: String)
extends ETransform[U, V](dateFormat) {

    override def |> : PartialFunction[U, Try[V]] = { //4
        case date: U if fileList.isDefined =>
            Try(if(date.isEmpty) getAll else get(date))
    }

    def get(t: U): V = getAll.filter(_.date == t.get)
    def getAll: V //5
    ...
}
```

The `DocumentsSource` class takes two arguments—the format of the date associated to the document and the name of the path the documents are located in (line 1). Being an explicit data transformation, the constructor for the `DocumentsSource` class should initialize the input type `U` (line 2) as a date converted to a `long` and output type `V` (line 3) as a `Corpus` for the extracting method `|>`.

The extractor `|>` generates a corpus associated to a specific date converted to a `long` type (line 4). The `getAll` method does the heavy lifting for extracting or sorting the document (line 5).

The implementation of the `getAll` method as well as other methods of the `DocumentsSource` class is described in the documented source code available online.

DMatrix class

Some of the discriminative learning models require operations performed on rows and columns of matrix. The `DMatrix` class facilitates the read and write operations on columns and rows:

```
class DMatrix(
    val nRows: Int,
    val nCols: Int,
    val data: Array[Double]) {

    def apply(i: Int, j: Int): Double = data(i*nCols+j)
    def row(iRow: Int): Array[Double] = {
        val idx = iRow*nCols
        data.slice(idx, idx + nCols)
    }

    def col(iCol: Int): IndexedSeq[Double] =
        (iCol until data.size by nCols).map( data(_) )
    def diagonal: IndexedSeq[Double] =
        (0 until data.size by nCols+1).map( data(_) )
    def trace: Double = diagonal.sum
    ...
}
```

The `apply` method returns an element of the matrix. The `row` method returns a row array and the `col` method returns the indexed sequence of column elements. The `diagonal` method returns the indexed sequence of diagonal elements and the `trace` method sums the diagonal elements.

The `DMatrix` class supports normalization of elements, rows, and columns, transposition, update of elements, columns, and rows. The `DMatrix` class implements a significant number of methods that are documented in the source code available online.

Counter

The `Counter` class implements a generic mutable counter for which the key is a parameterized type. The number of occurrences of a key is managed by a mutable hash map as follows:

```
class Counter[T] extends HashMap[T, Int] {
    def += (t: T): type.Counter = super.put(t, getOrElse(t, 0)+1)
    def + (t: T): Counter[T] = {
        super.put(t, getOrElse(t, 0)+1); this
    }
    def ++ (cnt: Counter[T]): type.Counter = {
        cnt./:(this) ((c, t) => c + t._1); this
    }
    def / (cnt: Counter[T]): HashMap[T, Double] = map {
        case (str, n) => (str, if( !cnt.contains(str) )
            throw new IllegalStateException(" ... ")
            else n.toDouble/cnt.get(str).get )
    }
    ...
}
```

The `+=` operator updates the counter of key `t`, and returns itself. The `+` operator updates, and then duplicates the updated counters. The `++` operator updates this counter with another counter. The `/` operator divides the count for each key by the counts of another counter.

The `Counter` class implements a significant set of methods that are documented in the source code available online.

Monitor

The `Monitor` class has two purposes:

- Log information and error messages using the methods `show` and `error`
- Collect and display variables related to the recursive or iterative execution of an algorithm

The data is collected at each iteration or recursion, then displayed as a time series with iterations as x axis values, as shown in the following code:

```
trait Monitor[T] {
    protected val logger: Logger
```

```
lazy val _counters = HashMap[String, ArrayBuffer[T]]()

def counters(key: String): Option[ArrayBuffer[T]]
def count(key: String, value: T): Unit
def display(key: String, legend: Legend)
  (implicit f: T => Double): Boolean
def show(msg: String): Int = show(msg, logger)
def error(msg: String): Int = error(msg, logger)
...
}
```

The `counters` method produces an array associated with a specific key. The `count` method updates the data associated with a key. The `display` method plots the time series. Finally, the methods `show` and `error` send information and error messages to the standard output.

The documented source code for the implementation of the `Monitor` class is available online.

Mathematics

This section describes very briefly some of the mathematical concepts used in the book.

Linear algebra

Many algorithms used in machine learning such as minimization of a convex loss function, principal component analysis, or least squares regression involves invariably manipulation and transformation of matrices. There are many good books on the subject, from the inexpensive [A:2] to the sophisticated [A:3].

QR decomposition

The QR decomposition (also known as QR factorization) is the decomposition of a matrix A into a product of an orthogonal matrix Q and upper triangular matrix R. $A = QR$ and $Q^T Q = I$ [A:4].

The decomposition is unique if A is a real, square, and invertible matrix. In the case of a rectangle matrix A, m by n with $m > n$ the decomposition is implemented as the dot product of two vector of matrices: $A = [Q_1, Q_2] \cdot [R_1, R_2]^T$ where Q_1 is an m by n matrix, Q_2 is an m by n matrix, R_1 is n by n and an upper triangle matrix, R_2 is an m by n null matrix.

QR decomposition is a reliable method for solving large systems of linear equations for which the number of equations (rows) exceeds the number of variables (columns). Its asymptotic computational time complexity for a training set of m dimension and n observations is $O(mn^2 - n^3 / 3)$.

It is used to minimize the loss function for ordinary least squares regression, which is covered in the *Ordinary least squares regression* section in [Chapter 9, Regression and Regularization](#).

LU factorization

LU factorization is a technique to solve a matrix equation $A \cdot x = b$, where A is a nonsingular matrix and x and b are two vectors. The technique consists of

decomposing the original matrix A as the product of simple matrix $A = A_1 A_2 \dots A_n$.

- Basic LU factorization defines A as the product of a lower unit triangular matrix L and an upper triangular matrix, U : $A = LU$
- LU factorization with pivot define A as the product of a permutation matrix, P , a unit lower triangular matrix L and an upper triangular matrix, U : $A = PLU$

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 , $L^T A = LDL^T$.

Cholesky factorization

The Cholesky factorization (also known as 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 lower triangle matrix L and its conjugate transpose L^T , $A = LL^T$.

The asymptotic computational time complexity for the Cholesky factorization is $O(mn^2)$, where m is the number of features (model parameters) and n the number of observations. The Cholesky factorization is used in linear least squares, Kalman filter (in the *Kalman filter: Recursive algorithm* section in [Chapter 3](#), *Data Preprocessing*), and nonlinear Quasi-Newton optimizer.

Singular Value Decomposition (SVD)

The SVD of real matrices defines a $m \times n$ real matrix A as the product of a m square real unitary matrix, U a $m \times n$ rectangular diagonal matrix S and the transpose V^T matrix of a real matrix, $A = USV^T$.

The columns of the matrix U and V are the orthogonal bases and the value of

the diagonal matrix S is the singular values [A:4]. The asymptotic computational time complexity for the singular value decomposition for n observations and m features is $O(mn^2 - n^3)$. The singular value decomposition is used in minimizing the total least squares and solving homogeneous linear equations.

Eigenvalue decomposition

The Eigen-decomposition of a real square matrix A is the canonical factorization as $Ax = ?x$.

? is the eigenvalue (scalar) corresponding to 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 wherein the elements are the eigenvalues associated to the eigenvectors [A:5], [A:6]. The Eigen-decomposition is used in Principal Components Analysis (in the *Principal Components Analysis* section in [Chapter 5, Dimension Reduction](#).)

Algebraic and numerical libraries

There are many more Open Source algebraic libraries available to developers as API besides the Apache Commons Math used in [Chapter 3, Data Pre-processing](#), [Chapter 5, Dimensional Reduction](#), and [Chapter 6, Naive Bayes Classifiers](#), and Apache Spark/Mlib in [Chapter 17, Apache Spark Mlib](#).

- jBlas 1.2.3 (Java) is created by *Mikio Braun* under BSD revised license. The library provides Java and Scala developers with a high-level Java interface to BLAS and LAPACK—<https://github.com/mikiobraun/jblas>
- Colt 1.2.0 (Java) is a high performance scientific library developed at the CERN under the European Organization for Nuclear Research licence—<http://acs.lbl.gov/ACSSoftware/colt/>
- AlgeBird 2.10 (Scala) is developed at Twitter under Apache Public License 2.0. It defines abstract linear algebra concepts using monoid and monads. The library is an excellent example of high-level functional programming using Scala—<https://github.com/twitter/algebroid>
- Breeze 0.8 (Scala) is a numerical processing library using Apache Public

License 2.0 originally created by *David Hall*. It is a component of the ScalaNLP suite of machine learning and numerical computing libraries —<http://www.scala-nlp.org/>

The Apache Spark/MLlib framework bundles jBlas, Colt, and Breeze. The liblinear framework for conditional random fields uses colt linear algebra components.

Tip

Alternative to Java/Scala libraries

If your application or project needs a high performance numerical processing tool under limited resources (CPU and RAM memory), using a C/C++ compiled library is an excellent alternative if portability is not a constraint. The binary functions are accessed through the **Java Native Interface (JNI)**.

First order predicate logic

Propositional logic is the formulation of axioms or proposition. 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 of the stock price > 0.76 or Minimization of loss function != converge*

Propositional logic is subject to the rule of Boolean calculus. Let's consider three propositions P, Q, and R and the three Boolean operators NOT, AND, and OR:

- 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 a propositional logic [A:7]. The most common formulations of the first order logic are:

- Rules IF P THEN action
- Existential operators

First order logic is used to describe the classifiers in the learning classifier systems (in the *Learning classifiers systems* section in [Chapter 13, Evolutionary Computation](#)).

Jacobian and Hessian matrices

Let's consider a function with n variables x_i and m output y_j : $\{x_i\} \rightarrow \{y_j = f_j(x)\}$. The Jacobian matrix [A:8] is the matrix of the first order partial derivatives of an output value 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 continuously, twice differentiable function:

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \dots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Consider the following example:

$$f(x, y) = x^2y + e^{-y} \quad J(f) = \begin{bmatrix} 2xy, x^2 - e^{-y} \end{bmatrix} \quad H(f) = \begin{bmatrix} 2y & 2x \\ 2x & e^{-y} \end{bmatrix}$$

Summary of optimization techniques

Optimization is critical to the efficiency and to some extent extends the accuracy of the machine learning algorithms. Some basic knowledge in this field goes a long way to build practical solutions for large datasets.

Gradient descent methods

There are many optimization techniques that either rely on exclusively on the first order derivative or provide a linear algebra alternative to the computation of the gradient.

Steepest descent

The *steepest descent* (also known as gradient descent) method is one of the simplest techniques for finding a local minimum of any continuous, differentiable function F or the global minimum for any define, differentiable, convex function [A:9]. The value of a vector or data point x_{t+1} at iteration $t+1$ is computed from the previous value x_t using the gradient ∇F of function F and the slope ?:

$$x_{(t+1)} = x_{(t)} - \gamma \nabla F(a)$$

The steepest gradient algorithm is used for solving a system of nonlinear equations, minimization of the loss function in the logistic regression (refer to the *Numerical optimization* section in [Chapter 9, Regression and Regularization](#)), support vector classifier (refer to the *Non-separable case* section in [Chapter 12, Kernel Models and Support Vector Machine](#)), and multilayer perceptron (refer to the *Training strategy and classification* section in [Chapter 10, Multilayer Perceptron](#)).

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 symmetric square matrices. The solution x^* of the equation $A.x = b$ is expanded as the weighted summation of n basis orthogonal directions p_i (also known as *conjugate directions*):

$$Ax = b \rightarrow \sum_{i=0}^{n-1} \alpha_i p_i x^* = b; p_i \cdot p_j = 0$$

The solution x^* is extracted by computing the i^{th} conjugate vector p_i and then computing the coefficients α_i .

Stochastic gradient descent

The *stochastic gradient* method is a variant of the steepest descent that minimizes the convex function by defining the objective function F as the sum of differentiable, basis function f_i as follows:

$$F(x) = \sum_{i=0}^{n-1} f_i(x), x_{t+1} = x_t - \alpha \sum_{i=0}^{n-1} \nabla f_i(x)$$

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 basic functions [A:10]. The stochastic gradient descent is usually faster than other gradient descents or quasi-new methods in converging toward a solution for a convex function. The stochastic gradient descent is used in the logistic regression, support vector machines, and back-propagation neural networks.

Stochastic gradient is particularly suitable for discriminative models with large datasets [A:11]. Spark/Mlib makes extensive use of the stochastic gradient method.

Note

Batch gradient descent

The batch gradient descent is introduced and implemented in the *Step 5: Implementing the classifier* section under *Let's kick the tires* in [Chapter 1, Getting Started](#).

Quasi-Newton algorithms

Quasi-Newton algorithms are variations on Newton's methods of finding the value of a vector or data point that maximizes or minimizes a function F (1st order derivative is null) [A:12].

Newton's method is a well-known and simple optimization method for finding the solution of equations $F(x) = 0$ for which F is continuous and 2nd order differentiable. It relies on the Taylor series expansion to approximate the function F with a quadratic approximation on 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:

$$F(x_t + \Delta x) - F(x_t) \approx F'(x_t) \cdot \Delta x + F''(x_t) \cdot \frac{\Delta x^2}{2} \rightarrow x_{t+1} = x_t - \frac{F'(x_t)}{F''(x_t)}$$

Contrary to the Newton method, Quasi-Newton methods do not require that the 2nd order derivative, Hessian matrix, of the objective function be computed. It just has to be approximated [A:13]. There are several approaches to approximate the computation of the Hessian matrix.

BFGS

The **Broyden-Fletcher-Goldfarb-Shanno (BFGS)** is a Quasi-Newton iterative numerical method to solve unconstrained nonlinear problems. The Hessian matrix H_{t+1} at iteration t is approximated using the value of the previous iteration t as $H_{t+1} = H_t + U_t + V_t$ applied to the Newton equation for the direction p_t :

$$H_t p_t = -\nabla F(x_t), \quad x_{t+1} = x_t + \alpha_t p_t$$

The BFGS is used in the minimization of the cost function for the conditional random field and L₁ and L₂ regression.

L-BFGS

The performance of the BFGS algorithm is related to the caching of the approximation of the Hessian matrix in memory (U, V) at the cost of high memory consumption.

The Limited memory BFGS 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 gradient ΔG_t at iteration t, then computes these values for the next step $t+1$:

$$x_{t+1} = x_t + \Delta x_t; \quad \nabla F(x_t) + \Delta G_t = \Delta(\nabla F(x_t))$$

It is supported in Apache Commons Math 3.3+, Apache Spark/MLlib 1.0+, Colt 1.0+, and Liitb CRF libraries. L-BFGS is used in the minimization of the loss function in the Conditional Random Fields (refer to the *Conditional random fields* section in [Chapter 7, Sequential Data Models](#)).

Nonlinear least squares minimization

Let's consider the classic minimization of the least squares of a nonlinear function $y = F(x, w)$ with w_i parameters for observations $\{y, x_i\}$. 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 (also known as Jacobian):

$$w_{(t+1)} = w_{(t)} - \left\| \frac{\partial r_i(w_{(t)})}{\partial w_i} \right\|_{ij}^{-1} r(w_{(t)})$$

The Gauss-Newton algorithm is used in Logistic Regression (refer to the *Logistic regression* section in [Chapter 9, Regression and Regularization](#)).

Levenberg-Marquardt

The Levenberg-Marquardt algorithm is an alternative to the Gauss-Newton for solving nonlinear least squares and curve fitting problems. The method consists of adding the gradient (Jacobian) terms to the residuals r_i to approximate the least squares error:

$$\mathcal{L}(w + \delta) \approx \sum_{i=0}^{m-1} \left(r_i(w) - \frac{\partial F(x_i, w)}{\partial w} \delta \right)^2$$

The algorithm is used in the training of the logistic regression (refer to the *Logistic regression* section in [Chapter 9, Regression and Regularization](#)).

Lagrange multipliers

The Lagrange multipliers methodology is an optimization technique to find local optima of a multivariate function subject to equality constraints [A:14]. The problem is stated as follows:

maximize $f(x)$ subject to $g(x) = c$, c is a constant, x is a variable or features vector.

The 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 the gradient of \mathcal{L} over the variables x_i and λ . The Lagrange multipliers are computing by maximizing \mathcal{L} :

$$\mathcal{L}(x, \lambda) = f(x) + \lambda(g(x) - c)$$

$$\nabla_{x, \lambda} \mathcal{L}(x, \lambda) = 0$$

$$\nabla \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial x_i}, \frac{\partial \mathcal{L}}{\partial \lambda} \right]$$

Consider the following example:

$$f(x, y) = x^2 + y^2 \text{ subject } x - y = 2$$

$$\frac{\partial \mathcal{L}}{\partial x} = 2x + \lambda, \frac{\partial \mathcal{L}}{\partial y} = 2y - \lambda, \frac{\partial \mathcal{L}}{\partial \lambda} = x - y - 2$$

$$x = 1, y = -1, \lambda = -2$$

Lagrange multipliers are used in minimizing the loss function in the nonseparable case of linear support vector machines (refer to the *Non-separable case* section of [Chapter 12, Kernel Models and Support Vector Machines](#)).

Overview dynamic programming

The purpose of dynamic programming is to break down an optimization problem into a sequence of steps known as substructures [A:15]. There are two types of problems for which dynamic programming is suitable.

The solution of a global optimization problem can be broken down into optimal solutions for its subproblems. The solution of the subproblems is known as optimal substructures. Greedy algorithms or the computation of the minimum span of a graph are examples of the decomposition into optimal substructures. Such algorithms can be implemented either recursively or iteratively.

The solution of the global problem is applied recursively to the subproblems, where the number of subproblems is small. This approach is known as dynamic programming using overlapping substructures. Forward-backward passes on hidden Markov models, the Viterbi algorithm (refer to the *Viterbi algorithm* section in [Chapter 12, Sequential Data Models](#)) or the back-propagation of error in a multilayer perceptron (refer to the *Step 3 - Error backpropagation* section in [Chapter 10, Multilayer Perceptron](#)) are good examples of overlapping substructures.

The mathematical formulation of dynamic programming solutions is specific to the problem it attempts to resolve. Dynamic programming techniques are also commonly used in mathematical puzzles such as the tour of Hanoi.

Finances 101

The exercises presented throughout the 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 examining 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 the book. Here are the definitions of the most commonly used metrics [A:16]:

- **Earnings per share (EPS):** This is the ratio of net earnings over number of outstanding shares.
- **Price/Earnings Ratio (PE):** This is the ratio of market price per share over 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 PE per share over annual growth of earnings per share.
- **Operating Income:** This is the difference between operating revenue and operating expenses.
- **Net Sales:** This is the difference between revenue or gross sales and cost of goods or cost of sales.
- **Operating Profit Margin:** This is the ratio of 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.
- **Short Interest Ratio:** This is the ratio of short interest over total number of shares floated.
- **Cash per Share:** This is the ratio of 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 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 12 months to gross dividends paid to common shareholders, expressed as percent.
- **Growth Domestic Product (GDP):** This is the aggregate measure of the economic output of a country. It measures the sum of value added in 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 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 for forecasting 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 publicity known about that asset as processed by market participants.

Terminology

- **Bearish or bearish position:** A bear position attempts to profit by betting the prices of the security will fall.
- **Bullish or bullish position:** A bull position attempts to profit by betting the price of the security will rise.
- **Long position:** A long position attempts to profit by betting the price of the security will rise.
- **Neutral position:** A neutral position attempts to profit by betting that 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 formula.
- **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 signal or indicator.
- **Relative strength index:** The **Relative strength index (RSI)** is an oscillator that computes the average of the 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:** A short position attempts to profit by betting the price

of the security will fall.

- **Support:** The support level is the lower limit of the price range of a security over a period. The price bounces back as soon as it reaches the support level.
- **Technical indicator:** The 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 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, upward or downward.
- **Volatility:** This is the variance or standard deviation of the price of a security over a period.

Trading data

The raw trading data extracted from Google or Yahoo financials pages consists of the following:

- **adjClose (or close):** This is the adjusted or nonadjusted price of a security at the closing of the trading session
- **open:** This is the price of the security at the opening of the trading session
- **high:** This is the highest price of the security during the trading session
- **low:** This is the lowest price of the security during the trading session

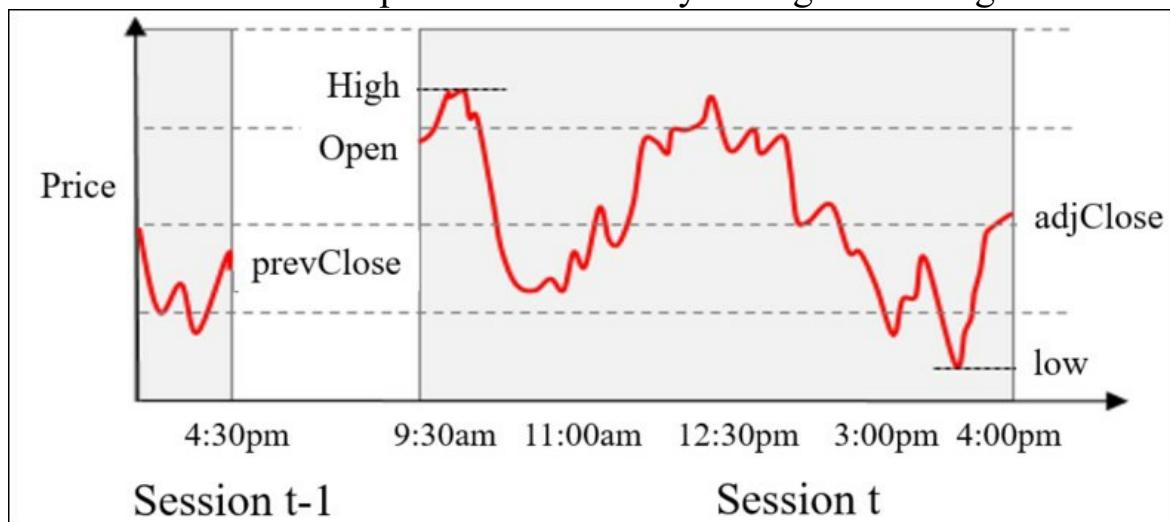


Illustration of a daily open and close price of a stock

We can derive the following metrics from the raw trading data:

- **Price volatility:** $\text{volatility} = 1.0 - \text{high}/\text{low}$
- **Price variation:** $v\text{Price} = \text{adjClose} - \text{open}$
- **Price difference (or change) between two consecutive sessions:** $d\text{Price} = \text{adjClose} - \text{prevClose} = \text{adjClose}(t) - \text{adjClose}(t-1)$
- **Volume difference between two consecutive sessions:** $d\text{Volume} = \text{volume}(t)/\text{volume}(t-1) - 1.0$
- **Volatility difference between two consecutive sessions:** $d\text{Volatility} = \text{volatility}(t)/\text{volatility}(t-1) - 1.0$
- **Relative price variation over the last T trading days:** $r\text{Price} = \text{price}(t)/\text{average(price over } T) - 1.0$
- **Relative volume variation over the last T trading days:** $r\text{Volume} = \text{volume}(t)/\text{average(volume over } T) - 1.0$
- **Relative volatility variation over the last T trading days:** $r\text{Volatility} = \text{volatility}(t)/\text{average(volatility over } T) - 1.0$

Trading signal and strategy

The purpose is to create a set variable x derived from price and volume, $x = f(\text{price}, \text{volume})$ that generates predicates, $x \ op \ c$ for which op is a Boolean operator such as $>$ or $=$ that compare the value of x to a predetermined threshold c .

Let's consider one of the most common technical indicators derived from a price: the relative strength index, RSI or the normalized RSI, $nRSI$ for which the formulation is provided next as a reference.

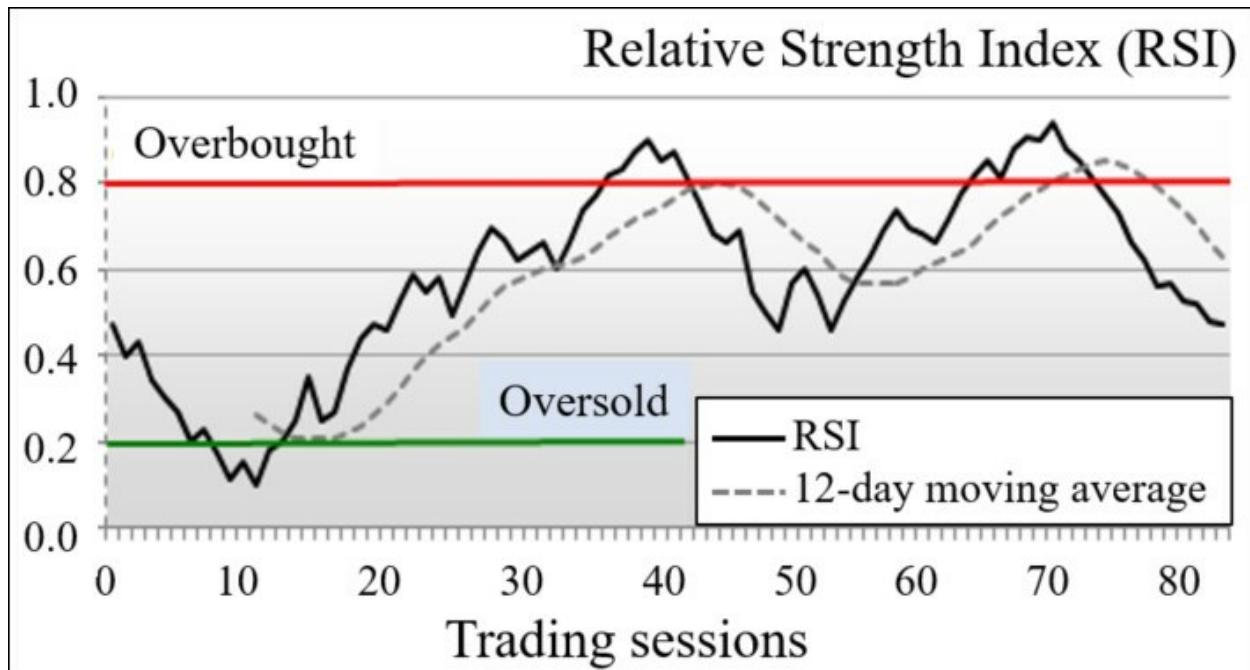
Note

Relative strength Index

RSI for a period of T sessions, with po opening price, pc closing price is as follows:

$$U = p(t) - p(t-1), D = 0$$

A *trading signal* is a predicate using a technical indicator, $nRSIT(t) < 0.2$. In trading terminology, a *signal* is emitted for any period t for which the predicate is true:



Visualization of oversold and overbought position using the relative strength index

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, I_{10} is the current rate of the 10-year Treasury bond, and RSI_{sp500} is the relative strength index of the S&P 500, then we could conclude that the increase of the exchange rate of US\$ to the Japanese yen is maximized for the following trading strategy:

$$\{G < \$1170 \text{ and } I_{10} > 3.9\% \text{ and } RSI_{sp500} > 0.6 \text{ and } RSI_{sp500} < 0.8\}$$

Price patterns

Technical analysis assumes that historical prices 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:

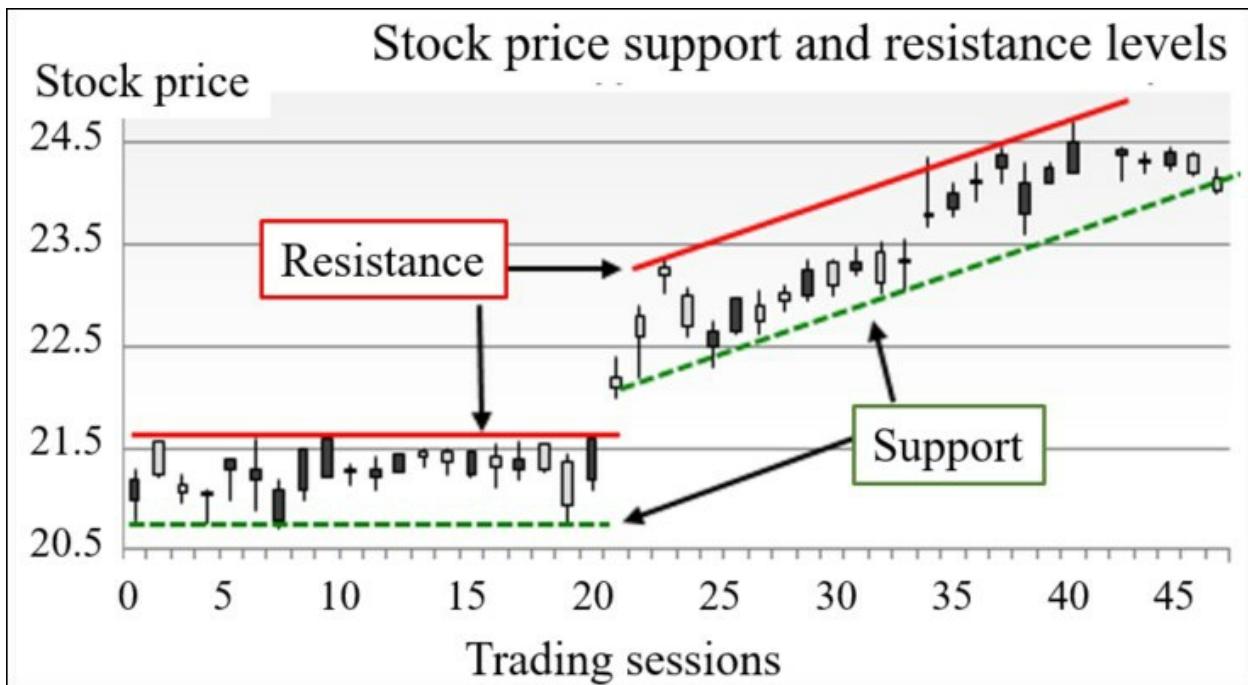


Illustration of trend and support resistance levels in technical analysis

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. Buyers of calls expect that the price of the security will increase substantially over a 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. 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 contact 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 occur is the loss of premium or \$270 when the option expires worthless. 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 $1030/270 = 380$ percent. The action of buying the stock at \$24, then selling at \$36 would have generated a return on investment of $36/24 - 1 = 50$ percent. This example is simple and does not consider the transaction fee or margin cost [A:20]:

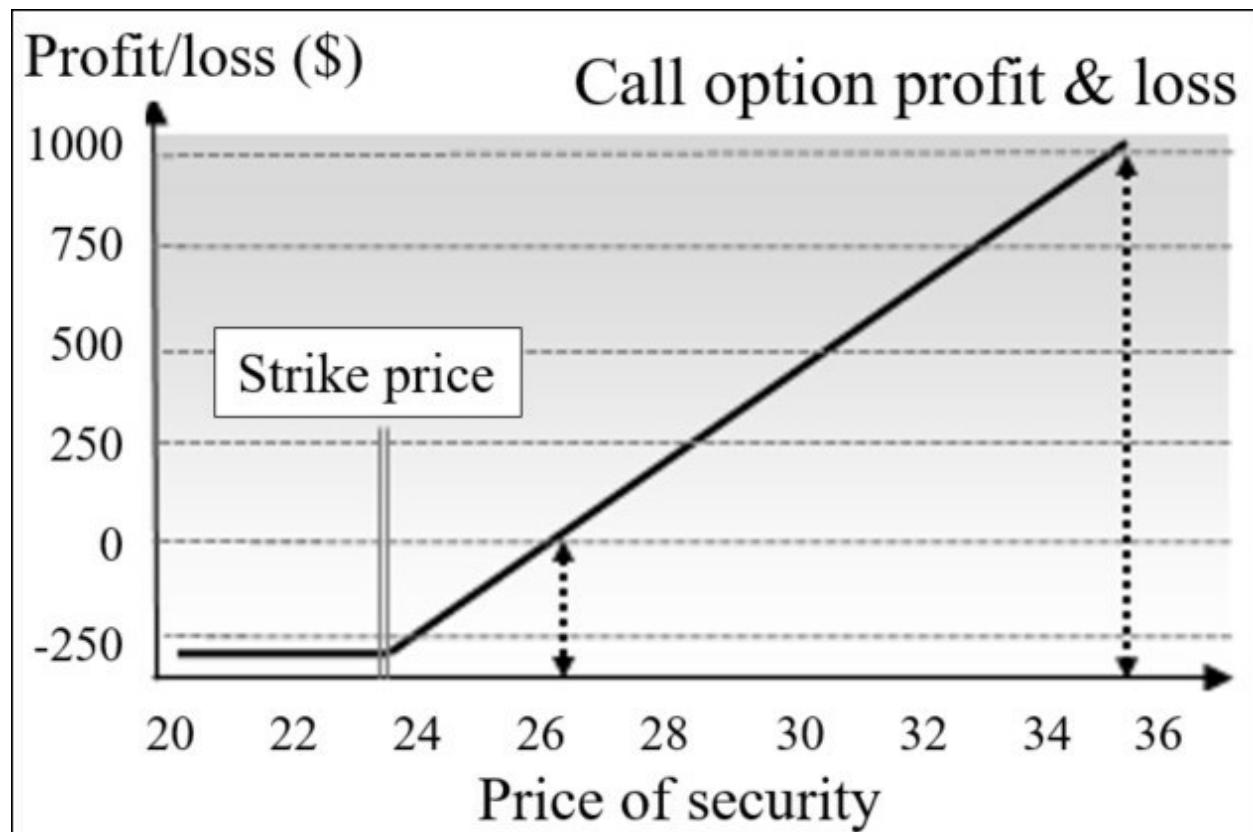


Illustration of the pricing of a call option

Financial data sources

There are numerous sources of financial data available to experiment with machine learning and validate models [A:21]:

- Yahoo finances (stocks, ETFs, and Indices): <http://finance.yahoo.com>
- Google finances (stocks, ETFs, and Indices):
<https://www.google.com/finance>
- NASDAQ (stocks, ETFs, and Indices): <http://www.nasdaq.com>
- European Central Bank (European bonds and notes): <http://www.ecb.int>
- TrueFx (Forex): <http://www.truefx.com>
- Quandl (economics and financials statistics): <http://www.quandl.com>
- Dartmouth University (portfolio and simulation):
<http://mba.tuck.dartmouth.edu>

Suggested online courses

- Practical Machine Learning, J. Leek, R. Peng, B. Caffo, Johns Hopkins University: <https://www.coursera.org/jhu>
- Probabilistic Graphical Models, D. Koller, Stanford University: <https://www.coursera.org/course/pgm>
- Machine Learning, A. Ng, Stanford University: <https://www.coursera.org/course/ml>

References

- [A:1] *Daily scala: Enumeration* J. Eichar 2009 <http://daily-scala.blogspot.com/2009/08/enumerations.html>
- [A:2] *Matrices and Linear Transformations 2nd edition* C. Cullen, Dover Books on Mathematics 1990
- [A:3] *Linear Algebra: A Modern Introduction* D. Poole, BROOKS/COLE CENGAGE Learning 2010
- [A:4] *Matrix decomposition for regression analysis* D. Bates 2007 <http://www.stat.wisc.edu/courses/st849-bates/lectures/Orthogonal.pdf>
- [A:5] *Eigenvalues and Eigenvectors of Symmetric Matrices* I. Mateev 2013 <http://www.slideshare.net/vanchizzle/eigenvalues-and-eigenvectors-of-symmetric-matrices>
- [A:6] *Linear Algebra Done Right 2nd edition §5 Eigenvalues and Eigenvectors* S. Axler, Springer 2000
- [A:7] *First Order Predicate Logic* S. Kaushik, CSE India Institute of Technology, Delhi http://www.cse.iitd.ac.in/~saroj/LFP/LFP_2013/L4.pdf
- [A:8] *Matrix Recipes* J. Movellan 2005
- [A:9] *Gradient descent: Wikipedia, the free encyclopedia* Wikimedia foundation http://en.wikipedia.org/wiki/Gradient_descent
- [A:10] *Large Scale Machine Learning: Stochastic Gradient Descent Convergence* A. Ng Stanford University <https://class.coursera.org/ml-003/lecture/107>
- [A:11] *Large-Scala Machine Learning with Stochastic Gradient Descent* L. Bottou 2010 <http://leon.bottou.org/publications/pdf/compstat-2010.pdf>

[A:12] *Overview of Quasi-Newton optimization methods* Dept. Computer Science, University of Washington

[A:13] *Lecture 2-3: Gradient and Hessian of Multivariate Function* M. Zibulevsky 2013 <http://www.youtube.com>

[A:14] *Introduction to the Lagrange Multiplier* ediwm.com Video
<http://www.noodle.com/learn/details/334954/introduction-to-the-lagrange-multiplier>

[A:15] *A brief introduction to Dynamic Programming (DP)* A. Kasibhatla, Nanocad Lab

http://nanocad.ee.ucla.edu/pub/Main/SnippetTutorial/Amar_DP_Intro.pdf

[A:16] *Financial ratios* Wikipedia
http://en.wikipedia.org/wiki/Financial_ratio

[A:17] *Getting started in Technical Analysis §1 Charts: Forecasting Tool or Folklore?* 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 Your 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
http://en.wikipedia.org/wiki/List_of_financial_data_feeds

Appendix B. References

The author does not make any guarantee that web references and links listed for some references are still accessible.

Chapter 1

[1:1] One Div Zero Monads are Elephants Part 2, *J. Iry Blog - 2007*:

<http://james-iry.blogspot.com/2007/10/monads-are-elephants-part-2.html>

[1:2] *Monad Design for the Web §7 A Review of Collections as Monads*, L.G. Meredith - Artima - 2012

[1:3] *Scalable Component Abstractions*, M. Odersky M. Zenger - 2005

<http://lamp.epfl.ch/~odersky/papers/ScalableComponent.pdf>

[1:4] *Akka Essentials §1 Introduction to Akka*, M. K. Gupta - Packt Publishing 2012

[1:5] Introduction to Machine Learning §1.2 Examples of Machine Learning Applications E. Alpaydin – MIT Press 2007

[1:6] Pattern Recognition and Machine Learning §1.2 Probability Theory, C. Bishop – Springer 2006

[1:7] Apache Commons Math 3.3, The Apache Software Foundation -
<http://commons.apache.org/proper/commons-math/>

[1:8] *JFreeChart version 1.0.1*, Object Refinery Limited 2013 -
<http://www.jfree.org/jfreechart/>

[1:9] Scalanlp/Breeze Wiki, <https://github.com/scalanlp/breeze/wiki>

[1:10] Programming in Scala 2nd edition §19 Type Parameterization, M. Odersky, L. Spoon, B. Venners - Artima 2008

[1:11] Effective Scala, M. Eriksen - Twitter – 2012 -
<http://twitter.github.io/effectivescala>

[1:12] *Large-Scale Machine Learning with Stochastic Gradient Descent*, L.

Bottou – 2010- <http://leon.bottou.org/publications/pdf/compstat-2010.pdf>

[1:13] [http://www.scala-lang.org/api/2.12.0/scala/util/Random\\$.html](http://www.scala-lang.org/api/2.12.0/scala/util/Random$.html)

Chapter 2

[2:1] Scientific modeling Wikipedia the Free encyclopedia, Wikimedia Foundation http://en.wikipedia.org/wiki/Scientific_modelling

[2:2] Inside F# Brian's thoughts on F# and .NET: Pipelining in F# - 2008
<http://lorgonblog.wordpress.com/2008/03/30/pipelining-in-f/>

[2:3] Programming in Scala *2nd edition §17 Collections*, M. Odersky, L. Spoon, B. Venners - Artima 2008

[2:4] Programming in Scala *2nd edition §12.5 Traits as stackable modification*, M. Odersky, L. Spoon, B. Venners - Artima 2008

[2:5] Dependency Injection in Scala: Cake Pattern V. Mencik, J. Janecek, M. Prihoda - Czech Scala Enthusiasts 2013:

<http://www.slideshare.net/czechscala/dependency-injection-in-scala-part>

[2:6] Dependency Injection in Scala: Extending the Cake pattern *A. Warski Blog of Adam Warski 2010*: <http://www.warski.org/blog/2010/12/di-in-scala-cake-pattern>

[2:7] Introduction to Machine Learning *§14.2 Cross-Validation and Resampling Methods* E. Alpaydin – MIT Press 2004-2007

[2:8] *Machine learning: A Probabilistic Perspective* §1.1 Introduction Example: Polynomial Curve Fitting K. Murphy – MIT Press 2012

[2:9] The Elements of Statistical Learning: Data Mining, Inference and Prediction §7.2 Bias, Variance and Model Complexity T. Hastie R. Tibshirani, J. Friedman - Springer 2001

Chapter 3

[3:1] *Programming in Scala 3rd edition §21.6 Context bounds*, M. Odersky, L. Spoon, B. Venners - Artima 2016

[3:2] *Moving Averages*, Wikipedia, the free encyclopedia Wikimedia Foundation - http://en.wikipedia.org/wiki/Moving_average_model

[3:3] *Forecasting with weighted moving averages* P. Kumar, Kushmanda Manpower 2011

[3:4] *Practical Guide to Data Smoothing & Filtering*, T. Van Den Bogert - 1996 -<http://isbweb.org/software/sigproc/bogert/filter.pdf>

[3:5] *Spectral density estimation*, Wikipedia, the free encyclopedia Wikimedia Foundation - http://en.wikipedia.org/wiki/Spectral_estimation

[3:6] *The Fast Fourier Transform and Its Applications* E O. Brigham - Prentice-Hall 1988

[3:7] *Fourier Transform Tutorial: Learn difficult engineering concepts through interactive flash programs* - <http://www.fourier-series.com/f-transform/>

[3:8] *The Cooley-Tukey Fast Fourier Transform Algorithm*, C. S. Burrus OpenStax -<http://cnx.org/content/m16334/latest/>

[3:9] *Apache Commons Math 3.3 API*, Apache Software Foundation - <http://commons.apache.org/proper/commons-math/apidocs/index.html>

[3:10] *Scala for the Impatient §15.6 Specialization for primitive types*, C. Horstmann - Addison-Wesley 2012

[3:11] *An introduction to the Kalman Filter*, G Welch, G Bishop - University of North Carolina 2006 -
http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf

[3:12] *Advanced Robotics: lecture 22, HMMs, Kalman filters*, P. Abbeel - University of California, Berkeley 2009 -
<http://www.eecs.berkeley.edu/~pabbeel/cs287-fa09/lecture-notes/lecture22-6pp.pdf>

[3:13] *Stochastic models, Estimation and Control*, P. Maybeck - Academic Press 1979

[3:14] *10-Year Treasury Note*, Investopedia -
<http://www.investopedia.com/terms/1/10-yeartreasury.asp>

[3:15] *Nonlinear Filtering of Non-Gaussian Noise*, K. Plataniotis, D. Andoutsos, A. Venetsanopoulos - Journal of Intelligent and Robotic Systems 19: 207-213 Kluwer Academic Publishers 1997

Chapter 4

[4:1] *Unsupervised Learning*, P. Dayan - The MIT Encyclopedia of the Cognitive Sciences, Wilson & Kiel editors 1998 -
<http://www.gatsby.ucl.ac.uk/~dayan/papers/dun99b.pdf>

[4:2] *Learning Vector Quantization (LVQ): Introduction to Neural Computation*, J. Bullinaria – 2007 -
http://www.cs.bham.ac.uk/~pxt/NC/lvq_jb.pdf

[4:3] *The Elements of Statistical Learning: Data Mining, Inference and Prediction §14.3 Cluster Analysis*, T. Hastie, R. Tibshirani, J. Friedman - Springer 2001

[4:4] *Efficient and Fast Initialization Algorithm for K-means Clustering*, International Journal of Intelligent Systems and Applications – M. Agha, W. Ashour - Islamic University of Gaza 2012 - <http://www.mecs-press.org/ijisa/ijisa-v4-n1/IJISA-V4-N1-3.pdf>

[4:5] *A Comparative Study of Efficient Initialization Methods for the K-Means Clustering Algorithms*, M E. Celebi, H. Kingravi, P Vela – 2012 -
<http://arxiv.org/pdf/1209.1960v1.pdf>

[4:6] *Machine Learning: A Probabilistic Perspective: §25.1 Clustering Introduction*, K. Murphy – MIT Press 2012

[4:7] *Maximum Likelihood from Incomplete Data via the EM Algorithm* - Journal of the Royal Statistical Society Vo. 39 No .1 A. P. Dempster, N. M. Laird, and D. B. Rubin. 1977 -
<http://web.mit.edu/6.435/www/Dempster77.pdf>

[4:8] *Machine Learning: A Probabilistic Perspective §11.4 EM algorithm*, K. Murphy – MIT Press 2012

[4:9] *The Expectation Maximization Algorithm A short tutorial*, S. Borman – 2009 - http://www.seanborman.com/publications/EM_algorithm.pdf

[4:10] *Apache Commons Math library 3.3:*
org.apache.commons.math3.distribution.fitting, The Apache Software Foundation - <http://commons.apache.org/proper/commons-math/javadocs/api-3.6/index.html>

[4:11] Pattern Recognition and Machine Learning §9.3.2 An Alternative View of EM- Relation to K-means, *C. Bishop –Springer 2006*

[4:12] Machine Learning: A Probabilistic Perspective §11.4.8 Online EM, K. Murphy – MIT Press 2012

[4:13] *Function approximation – Wikipedia the free encyclopedia* Wikimedia Foundation - https://en.wikipedia.org/wiki/Function_approximation

[4:14] *Function Approximation with Neural Networks and Local Methods: Bias, Variance and Smoothness*, S. Lawrence, A.Chung Tsoi, A. Back - University of Queensland Australia 1998 - <http://machine-learning.martinsewell.com/ann/LaTB96.pdf>

Chapter 5

[5:1] *CFCS: Entropy and Kullback-Leibler Divergence*, M. Osborne - University of Edinburg 2008 -
<http://www.inf.ed.ac.uk/teaching/courses/cfcs1/lectures/entropy.pdf>

[5:2] *Jensen-Shannon divergence*, Wikipedia the free encyclopedia
Wikimedia Foundation - https://en.wikipedia.org/wiki/Jensen-Shannon_divergence

[5:3] *Machine Learning: A Probabilistic Perspective* §2.3.8 Mutual Information, K. Murphy – MIT Press 2012

[5:4] *Learning with Bregman Divergences*, I. Dhillon, J. Ghosh - University of Texas at Austin - <http://www.cs.utexas.edu/users/inderjit/Talks/bregtut.pdf>

[5:5] *A Tutorial on Principal Components Analysis*, L. Smith, - 2002
http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf

[5:6] *Fast Cross-validation in Robust PCA*, S. Engelen, M. Hubert - COMPSTAT 2004 symposium, Partial Least Squares Physica-Verlag/Springer
<http://wis.kuleuven.be/stat/robust/papers/2004/fastcvpcaCOMPSTAT2004.pdf>

[5:7] *A survey of dimension reduction techniques*, I. Fodor - Center for Applied Scientific Computing Lawrence Livermore National Laboratory 2002 - <https://e-reports-ext.llnl.gov/pdf/240921.pdf>

[5:8] Multiple Correspondence Analysis - Wikipedia

https://en.wikipedia.org/wiki/Multiple_correspondence_analysis

[5:9] *Dimension Reduction for Fast Similarity Search in Large Time Series Databases*. E. Keogh, K. Chakrabarti, M. Pazzani, S. Mehrotra. - Dept. of Information and Computer Science, University of California Irvine 2000 - <http://www.ics.uci.edu/~pazzani/Publications/dimen.pdf>

[5:10] *Manifold learning with applications to object recognition*, D Thompson - Carnegie-Mellon University Course AP 6 -
<https://www.cs.cmu.edu/~efros/courses/AP06/presentations/ThompsonDimen>

[5:11] *Manifold learning: Theory and Applications*, Y. Ma, Y. Fu - CRC Press 2012

Chapter 6

[6:1] *Probabilistic Graphical Models: Overview and Motivation*, D. Koller - Stanford University - <http://www.youtube.com>

[6:2] *Introduction to Machine Learning §3.2 Bayesian Decision Theory*, E. Alpaydin - MIT Press 2004

[6:3] *Machine Learning: A Probabilistic Perspective §10 Directed graphical models*, K. Murphy - MIT Press 2012

[6:4] Probabilistic Entity-Relationship Models, PRMs, and Plate Models, *D. Heckerman, C. Meek, D. Koller -Stanford University -*
<http://robotics.stanford.edu/~koller/Papers/Heckerman+al:SRL07.pdf>

[6:5] Think Bayes Bayesian Statistics Made Simple §1 Bayes's Theorem, *A. Downey - Green Tea Press 2010 -*
<http://greenteapress.com/thinkbayes/html/index.html>

[6:6] *Machine Learning: A Probabilistic Perspective Information §2.8.3 Theory-Mutual Information*, K. Murphy – MIT Press 2012

[6:7] *Introduction to Information Retrieval §13.2 Naïve Bayes text classification*, C.D. Manning, P. Raghavan and H. Schütze, - Cambridge University Press 2008

[6:8] *Hidden Naïve Bayes*, H. Zhang, J. Su - University of New Brunswick L Jiang University of Geosciences Wuhan, 2004 -
<http://www.cs.unb.ca/profs/hzhang/publications/AAAI051ZhangH1.pdf>

[6:9] Pattern Recognition and Machine Learning §2.3.6 Bayesian inference for the Gaussian, C. Bishop –Springer 2006

[6:10] Pattern Recognition and Machine Learning §2.1 Binary Variables, C. Bishop –Springer 2006

[6:11] Machine Learning Methods in Natural Language Processing, *M. Collins - MIT CSAIL -2005* -
http://www.cs.columbia.edu/~mcollins/papers/tutorial_colt.pdf

[6:12] *Dbpedia: Wikipedia, the free encyclopedia* Wikimedia Foundation -
<http://en.wikipedia.org/wiki/DBpedia>

[6:13] *Introduction to Information Retrieval §20 Web crawling and indexes*, C.D. Manning, P. Raghavan and H. Schütze, - Cambridge University Press, 2008

[6:14] *Introduction to Information Retrieval §25 Support vector machines and machine learning on documents*, C.D. Manning, P. Raghavan and H. Schütze, - Cambridge University Press, 2008

Chapter 7

[7:1] *A tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*, L. Rabiner - Proceedings of the IEEE Volume 77, Feb 1989 - <http://www.cs.ubc.ca/~murphyk/Bayes/rabiner.pdf>

[7:2] *CRF Java library v 1.3* S. Sarawagi - Indian Institute of Technology, Bombay 2008 - <http://crf.sourceforge.net/>

[7:3] *Introduction to Machine Learning §13.2 Discrete Markov Processes* E. Alpaydin - The MIT Press 2004

[7:4] *A Revealing Introduction to Hidden Markov Models* M. Stamp - Dept. of Computer Science, San Jose State University 2012 - <http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf>

[7:5] *A brief introduction to Dynamic Programming* A. Kasibhatla - Nanocad Lab University of California, Los Angeles 2010 - http://nanocad.ee.ucla.edu/pub/Main/SnippetTutorial/Amar_DP_Intro.pdf

[7:6] Pattern Recognition and Machine Learning §13.2.1 Maximum Likelihood for the HMM C. Bishop –Springer 2006

[7:7] *Dynamic Programing in Machine learning – Part 3: Viterbi Algorithm and Machine Learning* E. Nichols - Nara Institute of Science and Technology - <https://www.youtube.com>

[7:8] *American Association of Individual Investors (AAII)* - <http://www.aaii.com>

[7:9] *Conditional Random Fields: Probabilistic Models for Segmenting and Labelling Sequence Data* – J. Lafferty - Carnegie Mellon University A. McCallum, University of Massachusetts F. Pereira - University of Pennsylvania 2001

[7:10] *Machine Learning for Multimedia Content Analysis §9.6 Conditional*

Random Fields case study Y. Gong, W. Xu - Springer 2007

[7:11] *Machine Learning: A Probabilistic Perspective* §19.6.2.4 Conditional random fields Natural language parsing K Murphy - MIT Press 2012

[7:12] *Conditional Random Field* §3 Various Interfaces KReSIT - IIT Bombay 2004 -

<http://crf.sourceforge.net/introduction/interfaces.html#FeatureGenerator>

[7:13] *Distributed Training for Conditional Random Fields* X. Lin, L Zhao, D Yu, X. Wu Key - Laboratory of Machine Perception and Intelligence School of Electronics Engineering and computer science China 2010 -

<http://www.klmp.pku.edu.cn/Paper/UsrFile/97.pdf>

Chapter 8

[8:1] *Machine Learning: A Probabilistic Perspective* §23 Monte-Carlo inference K Murphy - MIT Press 2012

[8:2] *Machine Learning: An Algorithmic Perspective* §15.1.2 Gaussian Random Numbers S. Marsland – Chapman & Hall/CRC 2015

[8:3] *Monte Carlo Method* Wikipedia the free encyclopedia Wikimedia Foundation - https://en.wikipedia.org/wiki/Monte_Carlo_method

[8:4] *Monte Carlo Integration* - Hitotsubashi University 2009 - <http://ta.twi.tudelft.nl/mf/users/oosterle/oosterlee/lec8-hit-2009.pdf>

[8:5] *Bootstrap: A Statistical Method* K. Singh, M. Xie – Rutgers University - <http://stat.rutgers.edu/home/mxie/RCPapers/bootstrap.pdf>

[8:6] *Machine Learning: A Probabilistic Perspective* §24 Markov Chain Monte-Carlo inference K Murphy - MIT Press 2012

[8:7] Pattern Recognition and Machine Learning §11.2.2 The Metropolis-Hastings Algorithm C. Bishop – Springer 2006

[8:8] Bayesian Inference: Gibbs Sampling I. Yildirim - University of Rochester 2012 - <http://www.mit.edu/~ilkery/papers/GibbsSampling.pdf>

Chapter 9

[:1] *Machine Learning Lecture 3 (CS 229)* A. Ng – Stanford University 2008

[9:2] *Matrix decompositions for regression analysis §1.2 The QR decomposition* D Bates – 2007 - <http://www.stat.wisc.edu/courses/st849-bates/lectures/Orthogonal.pdf>

[9:3] *Matrix decompositions for regression analysis §3.3 The Singular value decomposition* D Bates – 2007 - <http://www.stat.wisc.edu/courses/st849-bates/lectures/Orthogonal.pdf>

[9:4] *Gradient Descent for Linear Regression* A. Ng Stanford - University Coursera NL lecture 9 - <https://class.coursera.org/ml-003/lecture/9>

[9:5] *Stochastic gradient descent to find least square in linear regression* - Qize Study and Research 2014 -
<http://qizeresearch.wordpress.com/2014/05/23/stochastic-gradient-descent-to-find-least-square-in-linear-regression/>

[9:6] *Apache Commons Math Library 3.3 §1.5 Multiple linear regression* - The Apache Software Foundation -
<http://commons.apache.org/proper/commons-math/userguide/stat.html>

[9:7] *Lecture 2: From linear Regression to Kalman Filter and Beyond* S. Sarkka - Dept. of Biomedical Engineering and Computational Science, Helsinki University of Technology 2009 -
http://www.lce.hut.fi/~ssarkka/course_k2009/slides_2.pdf

[9:8] *Introductory Workshop on Time Series Analysis* S McLaughlin - Mitchell Dept. of Political Science University of Iowa 2013 -
<http://qipsr.as.uky.edu/sites/default/files/mitchelltimeserieslecture102013.pdf>

[9:9] Pattern Recognition and Machine Learning §3.1 *Linear Basis Function Models* C. Bishop - Springer 2006

[9:10] *Machine Learning: A Probabilistic Perspective* §13.1 $L1$
Regularization basics - K Murphy - MIT Press 2012

[9:11] *Feature selection, L1 vs. L2 regularization, and rotational invariance*
A. Ng, - Computer Science Dept. Stanford University -
<http://www.machinelearning.org/proceedings/icml2004/papers/354.pdf>

[9:12] Lecture 5: *Model selection and assessment* H. Bravo, R. Irizarry -
Dept. of Computer Science, University of Maryland 2010 -
<http://www.cbcn.umd.edu/~hcorrada/PracticalML/pdf/lectures/selection.pdf>

[9:13] *Machine learning: a probabilistic perspective* §9.3 *Generalized linear models* - K Murphy - MIT Press 2012

[9:14] *An Introduction to Logistic and Probit Regression Models* C. Moore -
University of Texas 2013 -
http://www.utexas.edu/cola/centers/prc/_files/cs/Fall2013_Moore_Logistic_P1

Chapter 10

[10:1] Neural Network: A Review *M. K. Gharate - PharmaInfo.net 2007*

[10:2] *Parallel Distributed Processing* R. Rumelhart, J. McClelland - MIT Press 1986

[10:3] Pattern Recognition and Machine Learning Chap 5 Neural Networks: Introduction C. Bishop –Springer 2006

[10:4] Neural Network Models §3.3 Mathematical Model, §4.6 Lyapunov Theorem for Neural Networks *P. De Wilde - Springer 1997*

[10:5] *Modern Multivariate Statistical Techniques §10.7 Multilayer Perceptrons (introduction)* A.J. Izenman - Springer 2008

[10:6] *Algorithms for initialization of neural network weights* A. Pavelka, A Prochazka - Dept. of Computing and Control Engineering. Institute of Chemical Technology -

http://dsp.vscht.cz/konference_matlab/matlab04/pavelka.pdf

[10:7] Design Patterns: Elements of Reusable Object-Oriented Software
\$Object creation pattern: builder *E. Gamma, R. Helm, R. Johnson, J. Vlissides - Addison Wesley 1995*

[10:8] *Introduction to Machine Learning: Linear Discrimination §10.7.2 Multiple Classes.* E. Alpaydin - MIT Press 2007

[10:9] Pattern Recognition and Machine Learning §5.3 Neural Networks: Error Backpropagation *C. Bishop –Springer 2006*

[10:10] Pattern Recognition and Machine Learning §5.2.4 Neural Networks: Gradient descent optimization *C. Bishop –Springer 2006*

[10:11] *Introduction to Machine Learning §11.8.1 Multilayer Perceptrons: Improving Convergence.* E. Alpaydin - MIT Press 2007

[10:12] The general inefficiency of batch training for gradient descent training *D. R. Wilson, Fonix Corp. T. R. Martinez - Brigham Young University Elsevier 2003* -

<http://axon.cs.byu.edu/papers/Wilson.nn03.batch.pdf>

[10:13] Regularization in Neural Networks CSE 574, §5 *S. Srihari - University of New York, Buffalo* -

<http://www.cedar.buffalo.edu/~srihari/CSE574/Chap5/Chap5.5-Regularization.pdf>

[10:14] *Stock Market Value Prediction Using Neural Networks* M.P. Naeini, H. Taremian, H.B. Hashemi - 2010 International Conference on Computer Information Systems and Industrial Management Applications IEEE -

http://people.cs.pitt.edu/~hashemi/papers/CISIM2010_HBHashemi.pdf

Chapter 11

[11:1] *Deep Learning §14 Autoencoders* I. Goodfellow, Y. Bengio, A. Courville – MIT Press 2016

[11:2] *Introduction to Autoencoders*, P. Galeone - 2016 -
<https://pgaleone.eu/neural-networks/2016/11/18/introduction-to-autoencoders/>

[11:3] An introduction to Restricted Boltzmann machines A Fisher, C. Igel – University of Copenhagen - <http://image.diku.dk/igel/paper/AItRBM-proof.pdf>

[11:4] *Machine Learning: A Probabilistic Perspective* §27.7 Restricted Boltzmann Machines, K Murphy - MIT Press 2012

[11:5] *Deep Learning §20.3 Deep Belief Networks*, I. Goodfellow, Y. Bengio, A. Courville – MIT Press 2016

[11:6] *Deep Learning 18.2 Stochastic Maximum Likelihood and Contrastive Divergence* I. Goodfellow, Y. Bengio, A. Courville -MIT Press 2016

[11:7] *Markov Chain Monte Carlo* Wikipedia -
https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo

[11:8] *Notes on Contrastive Divergence*, O. Woodford -
<http://www.robots.ox.ac.uk/~ojw/files/NotesOnCD.pdf>

[11:9] *Convergence Analysis of the Contrastive Divergence Algorithm*, X. Ma, X. Wang – Hindawi 2015 -
<https://www.hindawi.com/journals/mpe/2015/350102/>

[11:10] An Introduction to Convolution Neural Network – *Stanford University 2013* -
http://white.stanford.edu/teach/index.php/An_Introduction_to_Convolutional_

[11:11] Exploring Convolutional Neural Network Structures and Optimization Techniques for Speech Recognition, *O. Abdel-Hamid, L. Deng, D. Yu – Microsoft Research, Interspeech 2013 -*
http://research.microsoft.com/pubs/200804/CNN-Interspeech2013_pub.pdf

Chapter 12

[12:1] *Machine Learning: A Probabilistic Perspective* §14.1 Kernels
Introduction K. Murphy – MIT Press 2012

[12:2] *An introduction into protein-sequence annotation* A. Muller - Dept. of Biological Sciences, Imperial College Center for Bioinformatics 2002 -
<http://www.sbg.bio.ic.ac.uk/people/mueller/introPSA.pdf>

[12:3] Pattern Recognition and Machine Learning §6.4 Gaussian processes C. Bishop –Springer 2006

[12:4] *Introduction to Machine Learning* §Nonparametric Regression: Smoothing Models. E. Alpaydin - MIT Press 2007

[12:5] *The Elements of Statistical Learning*: Data Mining, Inference and Prediction §5.8 Regularization and Reproducing Kernel Hilbert Spaces T. Hastie, R. Tibshirani, J. Friedman - Springer 2001

[12:6] *The Elements of Statistical Learning*: Data Mining, Inference and Prediction §12.3.2 The SVM as a penalization method. T. Hastie, R. Tibshirani, J. Friedman - Springer 2001

[12:7] *A Short Introduction to Learning with Kernels* B. Scholkopf, - Max Planck Institut für Biologische Kybernetik A. Smola Australian National University 2005 - <http://alex.smola.org/papers/2003/SchSmo03c.pdf>

[12:8] *Machine Learning for Multimedia Content Analysis* §10.1.5 SVM Dual Y. Gong, W. Xu- Springer 2007

[12:9] *LIBSVM: A Library for Support Vector Machines* C-C Chang, C-J Lin - 2003 - <http://www.csie.ntu.edu.tw/~cjlin/libsvm/>

[12:10] jLibSvm: Efficient Training of Support Vector Machines in Java D. Soergel - <https://github.com/davidsoergel/jlibsvm>

[12:11] *SVMlight Support Vector Machine implementation in C* T. Joachims - Dept. of Computer Science, Cornell University 2008 -
<http://svmlight.joachims.org/>

[12:12] *Machine Learning: A Probabilistic Perspective §14.5.3 Kernels, Choosing* C K. Murphy – MIT Press 2012

[12:13] *Data Mining: Anomaly, Outlier, Rare Event Detection* G. Nico – 2014 - http://gerardnico.com/wiki/data_mining/anomaly_detection

[12:14] *Machine Learning: A Probabilistic Perspective §14.5.1 Kernels: SVMs for regression* K. Murphy - MIT Press 2012

[12:15] Pattern Recognition and Machine Learning §7.1.2 Sparse Kernel Machines; Relation to logistic regression C. Bishop - Springer 2006

[12:16] *Very Large SVM Training using Core Vector Machines* I. Tsang, J. Kwok, P-M, Cheung - Dept. of Computer Science, The Hong Kong University of Science and Technology -
<http://www.gatsby.ucl.ac.uk/aistats/fullpapers/172.pdf>

[12:17] *Training Linear SVMs in Linear Time* T. Joachims - Dept. of Computer Science Cornell University -
http://www.cs.cornell.edu/people/tj/publications/joachims_06a.pdf

Chapter 13

[13:1] Adaptation in Natural and Artificial Systems: An introductory Analysis with Application to Biology, Control and Artificial Intelligence J. Holland –1992 MIT Press

[13:2] Genetic Algorithms in Search, Optimization and Machine Learning D. Goldberg - Addison-Wesley 1989

[13:3] *What is Evolution?* Stated Clearly – 2013 - <http://www.youtube.com>

[13:4] Complexity and Approximation: Combinatorial optimization problems and their approximability properties §Compendium of NP optimization problems G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A Marchetti-Spaccamela, M. Protazi - 1999 -

<http://www.csc.kth.se/~viggo/wwwcompendium/>

[13:5] *Introduction to Evolutionary Computing* §2 *What is an Evolutionary Algorithm?* A. Eiben, J.E. Smith – Springer 2003

[13:6] *Machine Learning: A Probabilistic Perspective* §16.6 Ensemble learning K. Murphy – MIT Press 2012

[13:7] *Adaptation in Natural and Artificial Systems: An introductory Analysis with Application to Biology, Control and Artificial Intelligence* §6 *Reproductive Plans and Genetic Operators* J. Holland –MIT Press 1992

[13:8] *Introduction to Genetic Algorithms Tutorial IX Selection* M. Obitko - <http://www.obitko.com/tutorials/genetic-algorithms/selection.php>

[13:9] *Introduction to Genetic Algorithms -§Scaling of Relative Fitness* E.D Goodman, - Michigan State University 2009 World Summit on Genetic and Evolutionary Computation, Shanghai -

http://www.egr.msu.edu/~goodman/GECSummitIntroToGA_Tutorial-goodman.pdf

[13:10] *The Lokta-Volterra equation*: Wikipedia the free encyclopedia
Wikimedia Foundation - http://en.wikipedia.org/wiki/Lotka-Volterra_equation

[13:11] *A comprehensive Survey of Fitness Approximation in Evolutionary Computation* Y. Jin - Honda Research Institute Europe 2003 -
<http://epubs.surrey.ac.uk/7610/2/SC2005.pdf>

[13:12] *Stock price prediction using genetic algorithms and evolution strategies* G. Bonde, R. Khaled Institute of Artificial Intelligence - University of Georgia - <http://worldcomp-proceedings.com/proc/p2012/GEM4716.pdf>

Chapter 14

[14:1] *Introduction to Machine Learning §Reinforcement learning: Single State Case: K-Armed Bandit*, E. Alpaydin - MIT Press 2007

[14:2] *Algorithms for the multiarmed bandit problem* V. Kuleshov, D. Precup
McGill University – 2000, <https://www.cs.mcgill.ca/~vkules/bandits.pdf>

[14:3] *Multiarmed Bandits and Exploration Strategies*, S. Raja -
<https://sudeepraja.github.io/Bandits/>

[14:4} *A Tutorial on Thompson Sampling*, D. Russo, B. Van Roy, A. Kazerouni, I. Osband, Z. Wen – Stanford University, Columbia University, Google Deepmind, Adobe Research 2017 -
http://web.stanford.edu/~bvr/pubs/TS_Tutorial.pdf

[14:5] *Analysis of Thompson Sampling for the Multiarmed Bandit Problem*, S. Agrawal, N. Goyal – Microsoft Research India – 2012 -
<http://proceedings.mlr.press/v23/agrawal12/agrawal12.pdf>

[14:6] Generalized Thompson Sampling for Contextual Bandits, L.Li
Microsoft Research – 2013 - <https://arxiv.org/pdf/1310.7163.pdf>

[14: 7] *Bandit Algorithms Continued: UCB1*: N. Welsh – 2010 -
<https://www.cs.bham.ac.uk/internal/courses/robotics/lectures/ucb1.pdf>

Chapter 15

[15:1] *Reinforcement Learning: An introduction* R.S. Sutton, A. Barto - MIT Press 1998

[15:2] *Reinforcement Learning and Plan Recognition* M. Veloso - Computer Science Dept. Carnegie Mellon University 2001 -
<http://www.cs.cmu.edu/~reids/planning/handouts/RL-HMM.pdf>

[15:3] *Reinforcement learning: A Brief Tutorial* D. Precup - Reasoning and Learning Lab, McGill University 2005 -
<http://www.iro.umontreal.ca/~lisa/seminaires/14-09-2005.pdf>

[15:4] *Programming in Scala 2nd Edition §18 Stateful Objects* M. Odersky, L. Spoon, B. Venners - Artima 2008

[15:5] *Scala for the Impatient §15.6 Annotations for Optimizations* C. Horstmann - Addison-Wesley 2012

[15:6] *Reinforcement Learning for automatic financial trading: Introduction and some applications* F. Bertoluzzo, M. Corazza, Ca'Foscari - University of Venice 2012 -
http://www.unive.it/media/allegato/DIP/Economia/Working_papers/Working

[15:7] *The Options Institute tutorial* - Chicago Board of Options Exchange -
<http://www.cboe.com/LearnCenter/Tutorials.aspx#basics>

[15:8] *Black-Scholes model* Wikipedia the Free encyclopedia Wikimedia Foundation - http://en.wikipedia.org/wiki/Black-Scholes_model

[15:9] *Value Function Approximation in Reinforcement Learning using the Fourier Basis* G. Konidaris, S. Osentoski, P. Thomas -
<http://lis.csail.mit.edu/pubs/konidaris-aaai11a.pdf>

[15:10] *A mathematical framework for studying learning in classifier systems.* J. Holland Physica D, volume 2, §1-3 1986

[15:11] *Introduction to Learning Classifier Systems Tutorial* J. Bacardit, N. Krasnogor - G53 Bioinformatics University of Nottingham -
<http://www.exa.unicen.edu.ar/escuelapav/cursos/bio/17.pdf>

[15:12] *Learning Classifier Systems: A Gentle Introduction*: P. L.Lanzi - Politecnico Di Milano GECCO 2014 -
<http://www.slideshare.net/pierluca.lanzi/gecco2014-learning-classifier-systems-a-gentle-introduction>

[15:13] *Automated Stock Trading and Portfolio Optimization Using XCS Trader and Technical Analysis* A. Chauban - Schools of Informatics, University of Edinburgh 2008 -
<http://www.inf.ed.ac.uk/publications/thesis/online/IM080575.pdf>

Chapter 16

[16:1] *Parallel Collections: Overview* – Scala Documentation A. Prokopec, H. Miller - <http://docs.scala-lang.org/overviews/parallel-collections/overview.html>

[16:2] *Parallel Collections: Measuring Performance*, Scala Documentation A. Prokopec, H. Miller - <http://docs.scala-lang.org/overviews/parallel-collections/overview.html>

[16:3] *The Actor Model: Towards Better Concurrency* D. Bereznitsky - Java Edge 2009 - <http://www.slideshare.net/drорbr/the-actor-model-towards-better-concurrency>

[16:4] *Scala for the impatient* §14.17 Partial Functions C Horstmann - Addison-Wesley 2012

[16:5] *Programming in Scala; a Comprehensive Step-by-Step Guide 2nd edition* §30.3 *Treating native threads as actors* M. Odersky, L. Spoon, B. Venners - Artima 2008

[16:6] *Deprecating the Observer Pattern* I. Maier, T Rompf, M. Odersky - Ecole Polytechnique Federale de Lausanne

[16:7] *Introducing Akka* J. Boner - Typesafe 2012 - <http://www.slideshare.net/jboner/introducing-akka>

[16:8] *Akka Essentials* §4 *Typed Actors* M. K. Gupta - Packt Publishing 2012

[16:9] *Akka Essentials* §6 *Supervision and Monitoring: Supervision* M. K. Gupta - Packt Publishing 2012

[16:10] *Concurrent Programming with Futures, Finagle* - Twitter Inc. 2004 - twitter.github.io/finagle/guide/Futures.html

Chapter 17

[17:01] Hands-on Tour of Apache Spark in 5 Minutes – HortonWorks 2015 -
<http://hortonworks.com/hadoop-tutorial/hands-on-tour-of-apache-spark-in-5-minutes/>

[17:02] *Apache Spark Programming Guide* - The Apache Software Foundation 2016 - <http://spark.apache.org/docs/latest/programming-guide.html>

[17:03] *MLlib: Scalable Machine Learning on Spark -Workshop* - X. Meng – 2015 - <http://stanford.edu/~rezab/sparkworkshop/slides/xiangrui.pdf>

[17:04] *Introduction to ML with Apache Spark MLlib* - Taras Matyashovsky 2016 - <http://www.slideshare.net/tmatyashovsky/introduction-to-ml-with-apache-spark-mllib>

[17:05] *ML Pipelines: A New High-Level API for MLlib* – E. Sparks, J. Bradley, S. Venkataman, X. Meng – Databricks 2015 -
<https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib.html>

[17:06] *Receiver Operating Characteristic* – Wikipedia
https://en.wikipedia.org/wiki/Receiver_operating_characteristic

[17:07] *Advanced Analytics with Spark – Chapter 3 Recommending Music/ and the Audioscrobbler Data Set / Computing AUC* S. Ryza, U. Laserson, S. Owen, J. Wills - O'Reilly 2015

[17:08] *Shannon Entropy and Kullback-Leibler Divergence* – C. Shalizi - Carnegie Mellon University 2006 -
<http://www.stat.cmu.edu/~cshalizi/754/2006/notes/lecture-28.pdf>

[17:09] *An Introduction to Feature Extraction* - I. Guyon, A. Elisseeff -
<http://clopinet.com/fextract-book/IntroFS.pdf>

[17:10] *Learning Real-time Processing with Spark Streaming Chapter 2 Architecture and Components of Spark and Spark Streaming* S. Gupta – Packt publishing 2015

[17:11] *Apache Spark Cluster Mode Overview* - The Apache Software Foundation 2016 - <http://spark.apache.org/docs/latest/cluster-overview.html>

[17:12] Apache Spark community mailing list <dev@spark.apache.org> List of Spark related meetup - <http://spark.meetup.com>

Index

A

- A/B testing / [Bayesian Bernoulli bandits](#)
- accuracy / [Key quality metrics](#)
 - versus model fitness / [Model fitness](#)
- action-value iterative update / [Action-value iterative update](#)
- activation convolution neural network / [Convolution layers](#)
- activation function / [Activation function](#)
- Actors
 - scalability / [Scalability with Actors](#)
- adaptive modeling / [Model categorization](#)
- aggregation effect / [Tuning the number of clusters](#)
- Akka / [Scala as a scalable language](#)
- Akka framework
 - about / [Akka](#)
 - URL, for downloading / [Akka](#)
 - master-workers design / [Master-workers](#)
 - futures / [Futures](#)
- AlgeBird
 - reference link / [Algebraic and numerical libraries](#)
- Algebroid / [Other libraries and frameworks](#)
- algebraic libraries
 - about / [Algebraic and numerical libraries](#)
- alpha (forward pass) / [Alpha \(forward pass\)](#)
- Analysis of Variance (ANOVA) / [Challenging model complexity](#)
- anomaly / [Anomaly detection with one-class SVC](#)
- anomaly detection
 - with one-class SVC / [Anomaly detection with one-class SVC](#)
- Apache Commons Math
 - reference link / [Leveraging Java libraries](#)
 - about / [Apache Commons Math](#)
 - description / [Description](#)

- licensing / [Licensing](#)
 - installation / [Installation](#)
 - URL, for download / [Installation](#)
- Apache Commons Math library
 - exceptions, handling / [Training workflow](#)
- Apache Mesos resource manager
 - URL / [Deploying Spark](#)
- Apache Public License 2.0
 - URL / [Licensing](#)
- Apache Spark / [Scala as a scalable language](#)
 - about / [Apache Spark core, Extending Spark](#)
 - need for / [Why Spark?](#)
 - design principles / [Design principles](#)
 - experimenting with / [Experimenting with Spark](#)
 - deploying / [Deploying Spark](#)
 - URL / [Deploying Spark](#)
 - shell, using / [Using Spark shell](#)
 - Kullback-Leibler divergence / [Kullback-Leibler divergence](#)
 - Kullback-Leibler evaluator / [Kullback-Leibler evaluator](#)
- area under PRC / [Area under PRC](#)
- area under ROC / [Area under ROC](#)
- areaUnderROC (AuROC)
 - about / [Training the model](#)
- attributes
 - about / [What is a model?](#)
- Auto-Regressive Integrated Moving Average (ARIMA) / [Alternative preprocessing techniques](#)
- Auto-Regressive Moving Average (ARMA) / [Alternative preprocessing techniques](#)
- autoencoders
 - sparse autoencoder / [Sparse autoencoder, Categorization](#)
 - undercomplete autoencoder / [Undercomplete autoencoder, Categorization, Feed-forward sparse, undercomplete autoencoder](#)
 - topology of hidden layers, characteristics / [Undercomplete autoencoder](#)
 - deterministic autoencoder / [Deterministic autoencoder](#)
 - complete autoencoder / [Categorization](#)

- overcomplete autoencoder / [Categorization](#)
- regularized autoencoder / [Categorization](#)
- stochastic autoencoder / [Categorization](#)
- feed-forward sparse autoencoder / [Feed-forward sparse, undercomplete autoencoder](#)
- implementing / [Implementation](#)
- autonomous systems
 - about / [Understanding the challenge](#)

B

- batch EM / [Online EM](#)
- batch training
 - versus online training / [Online versus batch training](#)
 - about / [Online versus batch training](#)
- Baum-Welch estimator (EM) / [Baum-Welch estimator \(EM\)](#)
- Bayesian Bernoulli bandits / [Bayesian Bernoulli bandits](#)
- Bayesian networks
 - about / [Probabilistic graphical models](#)
- Bellman optimality equations / [Bellman optimality equations](#)
- Bernoulli model / [Multivariate Bernoulli classification](#)
- best practices, Scala programming
 - encapsulation / [Encapsulation](#)
 - class constructor template / [Class constructor template](#)
 - companion objects, versus case classes / [Companion objects versus case classes](#)
 - enumeration, versus case classes / [Enumerations versus case classes](#)
 - overloading / [Overloading](#)
 - design template, for immutable classifiers / [Design template for immutable classifiers](#)
- beta (backward pass) / [Beta \(backward pass\)](#)
- bias-variance decomposition / [Bias-variance decomposition](#)
- BinaryClassificationEvaluator
 - about / [Validating the model](#)
- BinaryLogisticRegressionSummary
 - about / [Training the model](#)
- binary restricted Boltzmann machines
 - about / [Binary restricted Boltzmann machines](#)
 - conditional probabilities / [Conditional probabilities](#)
 - sampling / [Sampling](#)
 - log-likelihood gradient / [Log-likelihood gradient](#)
 - contrastive divergence / [Contrastive divergence](#)
 - configuration parameters / [Configuration parameters](#)
 - unsupervised learning / [Unsupervised learning](#)

- binary SVC
 - about / [The binary SVC](#)
 - LIBSVM / [LIBSVM](#)
 - design / [Design](#)
 - configuration parameters / [Configuration parameters](#)
 - interface, creating to LIBSVM / [Interface to LIBSVM](#)
 - training / [Training](#)
 - classification / [Classification](#)
 - margin / [C-penalty and margin](#)
 - C-penalty / [C-penalty and margin](#)
 - C-penalty, optimizing / [C-penalty and margin](#)
 - kernel, evaluation / [Kernel evaluation](#)
 - application, to risk analysis / [Application to risk analysis](#)
- binomial logistic regression / [Step 6 – evaluating the model](#)
- bitwise swap / [Fast Fisher-Yates shuffle](#)
- Bloomberg / [Naïve Bayes and text mining](#)
- Boltzmann machine / [Boltzmann machine](#)
- bootstrapping
 - with replacement / [Bootstrapping with replacement](#)
 - overview / [Overview](#)
 - resampling / [Resampling](#)
 - implementation / [Implementation](#)
 - pros and cons / [Pros and cons of bootstrap](#)
- Box-Muller transform
 - about / [Box-Muller transform](#)
- Breeze / [Other libraries and frameworks](#)
- Bregman distance
 - about / [The divergences](#)
- Broyden-Fletcher-Goldfarb-Shanno (BFGS) / [BFGS](#)
- Broyden-Fletcher-Goldfarb-Shannon method / [Numerical optimization](#)

C

- cake pattern
 - about / [Scala as an object oriented language, Step 3 – instantiation, Composing mixins to build workflow](#)
- canonical forms / [The hidden Markov model \(HMM\)](#)
 - evaluation / [The hidden Markov model \(HMM\)](#)
 - training / [The hidden Markov model \(HMM\)](#)
 - decoding / [The hidden Markov model \(HMM\)](#)
- category M
 - about / [Abstraction](#)
- centroid / [K-means](#)
- Cholesky factorization
 - about / [Cholesky factorization](#)
- chromosomes
 - about / [Evolutionary computing](#)
- class constructor template / [Class constructor template](#)
- classification / [Step 4 – Classification, Classification](#)
- classification, multilayer perceptron (MLP)
 - about / [Training and classification](#)
 - regularization / [Regularization](#)
 - model generation / [Model generation](#)
 - Fast Fisher-Yates shuffle / [Fast Fisher-Yates shuffle](#)
 - prediction / [Prediction](#)
 - model fitness / [Model fitness](#)
- cluster analysis
 - about / [K-mean clustering](#)
- clustering
 - about / [K-mean clustering](#)
- clustering algorithms
 - K-means / [K-mean clustering, K-means](#)
 - Expectation-Maximization / [K-mean clustering](#)
- clusters
 - defining / [Defining clusters](#)
 - initializing / [Initializing clusters](#)

- clusters assignment, K-means algorithm / [Step 2 – Clusters assignment](#)
- clusters configuration, K-means algorithm / [Step 1 – Clusters configuration](#)
- CNBC / [Naïve Bayes and text mining](#)
- Colt
 - reference link / [Algebraic and numerical libraries](#)
- companion objects
 - versus case classes / [Companion objects versus case classes](#)
- complete autoencoder / [Categorization](#)
- complex Fourier transform / [Fourier analysis](#)
- conditional independence
 - about / [Probabilistic graphical models](#)
- conditional random field (CRF)
 - about / [Conditional random fields](#), [Introduction to CRF](#)
 - linear chain CRF / [Linear chain CRF](#)
 - potential functions (f_i) / [Linear chain CRF](#)
 - identity potential functions / [Linear chain CRF](#)
 - transition feature functions / [Linear chain CRF](#)
 - state feature functions / [Linear chain CRF](#)
 - versus hidden Markov model (HMM) / [Comparing CRF and HMM](#)
- configuration parameters, binary SVC
 - about / [Configuration parameters](#)
 - SVM formulation / [The SVM formulation](#)
 - SVM kernel function / [The SVM kernel function](#)
 - SVM execution / [The SVM execution](#)
- confusion matrix
 - about / [F-score for multinomial classification](#)
- conjugate gradient / [Conjugate gradient](#)
- connectionism / [The biological background](#)
- constructive tuning / [Regularization](#)
- consumer price index (CPI)
 - about / [Introducing the multinomial Naïve Bayes](#)
- context bounds / [Context bounds](#)
- context free Thompson sampling / [Prior/posterior beta distribution](#)
- continuation-passing style (CPS)
 - about / [Beyond Actors – reactive programming](#)
- continuous-time Kalman filter / [Benefits and drawbacks](#)

- contract
 - about / [Error handling](#)
- contravariant vectors
 - about / [Manifolds](#)
- control learning
 - about / [A solution – Q-learning](#)
- convexity / [The kernel trick](#)
- convex minimization / [L_n roughness penalty](#)
- convolution layer / [Local receptive fields](#)
- convolution neural networks
 - about / [Convolution neural networks](#)
 - local receptive fields / [Local receptive fields](#)
 - weight sharing / [Weight sharing](#)
 - convolution layers / [Convolution layers](#)
 - sub-sampling layers / [Sub-sampling layers](#)
 - implementing / [Putting it all together](#)
- Cooley-Tukey algorithm / [Discrete Fourier transform \(DFT\)](#)
- cosine distance / [Measuring similarity](#)
- Counter class / [Counter](#)
- covariant functor
 - about / [Functors](#)
- covariant functor
 - about / [Functors](#)
- covariant vectors
 - about / [Manifolds](#)
- cross-validation
 - about / [Cross-validation](#)
 - 1-fold cross-validation / [One-fold cross-validation](#)
 - K-fold cross-validation / [K-fold cross-validation](#)
- crossover
 - about / [Crossover](#)
 - population / [Population](#)
 - chromosomes / [Chromosomes](#)
 - genes / [Genes](#)
- curse of dimensionality
 - about / [Curse of dimensionality](#)
- curve-fitting algorithms / [Alternative preprocessing techniques](#)

D

- Darwinian process
 - about / [The origin](#)
- data
 - about / [Modeling](#)
 - profiling / [Profiling data](#)
- data clustering / [Clustering](#)
- data partitioning / [Clustering](#)
- data scientist
 - about / [Defining a methodology](#)
- data segmentation / [Clustering](#)
- DataSourceConfig class / [Data extraction](#)
- Davidon-Fletcher-Powell method / [Numerical optimization](#)
- DBpedia / [Basics information retrieval](#)
- decision boundary / [Visualizing model features](#)
- decoding, hidden Markov model (HMM)
 - about / [Decoding \(CF-3\)](#)
 - Viterbi algorithm / [The Viterbi algorithm](#)
- deep belief network (DBM) / [Restricted Boltzmann Machines \(RBMs\)](#)
- Deep belief networks (DBNs) / [Boltzmann machine](#)
- Deep Boltzmann machines (DBMs) / [Boltzmann machine](#)
- def
 - overriding, with val / [Understanding the problem](#)
- density estimation / [Unsupervised learning](#)
- dependency injection
 - about / [Scala as an object oriented language, Understanding the problem](#)
- deployment modes, Apache Spark
 - standalone mode / [Deploying Spark](#)
 - local mode / [Deploying Spark](#)
 - Yarn clusters manager / [Deploying Spark](#)
 - Apache Mesos resource manager / [Deploying Spark](#)
- descriptive models / [Model categorization](#)
- design

- versus model / [Model versus design](#)
- designing / [Model versus design](#)
- design principles, Spark
 - about / [Design principles](#)
 - in-memory persistency / [In-memory persistency](#)
 - laziness / [Laziness](#)
 - transforms and actions / [Transforms and actions](#)
 - shared variables / [Shared variables](#)
- design template
 - for immutable classifiers / [Design template for immutable classifiers](#)
- destructive tuning / [Regularization](#)
- deterministic autoencoder / [Deterministic autoencoder](#)
- DFT-based filtering / [DFT-based filtering](#)
- differential operator / [Differential operator](#)
- dimension reduction / [Dimension reduction](#)
- directed graphical models
 - about / [Probabilistic graphical models](#)
- Dirichlet Latent Allocation / [Retrieving textual information](#)
- Discrete Fourier transform (DFT)
 - about / [Discrete Fourier transform \(DFT\)](#)
- discrete Fourier transform (DFT) / [Performance](#)
 - about / [Distributed discrete Fourier transform](#)
- discrete Kalman filter
 - about / [The discrete Kalman filter](#)
 - state space estimation / [The state space estimation](#)
 - recursive algorithm / [The recursive algorithm](#)
- discrete Markov chain
 - about / [The Markov property](#)
- discretization / [Value encoding](#)
- discretized streams (DStreams)
 - about / [Discretized streams](#)
- discriminative kernels
 - about / [Common discriminative kernels](#)
- discriminative models / [Discriminative models](#)
- divergences
 - about / [The divergences](#)

- Kullback-Leibler (KL) divergence / [The divergences](#)
- Jensen-Shannon metric / [The divergences](#)
- mutual information / [The divergences](#)
- Bregman distance / [The divergences](#)
- DMatrix class / [DMatrix class](#)
- DNA
 - about / [Evolutionary computing](#)
- DocumentsSource class / [Documents extraction](#)
- domain
 - about / [Defining a methodology](#)
- dynamic programming
 - overview / [Overview dynamic programming](#)

E

- eclipse Scala IDE
 - about / [Eclipse Scala IDE](#)
 - reference link / [Eclipse Scala IDE](#)
- Eigen-decomposition
 - about / [Eigenvalue decomposition](#)
- encapsulation
 - about / [Encapsulation](#)
- encoding scheme
 - flat encoding / [Flat encoding](#)
 - hierarchical encoding / [Hierarchical encoding](#)
- enumerations
 - versus case classes / [Enumerations versus case classes](#)
- epoch
 - about / [Training epoch](#)
- epoch training, multilayer perceptron (MLP)
 - about / [Training epoch](#)
 - input forward propagation / [Step 1 – input forward propagation](#)
 - error backpropagation / [Step 2 – error backpropagation](#)
 - exit condition / [Step 3 – exit condition](#)
 - implementing / [Putting it all together](#)
- Epsilon-greedy algorithm / [Epsilon-greedy algorithm](#)
- error backpropagation, multilayer perceptron (MLP)
 - about / [Step 2 – error backpropagation](#)
 - weights adjustment / [Weights adjustment](#)
 - error propagation / [Error propagation](#)
 - computational model / [The computational model](#)
- error handling
 - about / [Error handling](#)
- error insensitive zone / [Overview](#)
- Euclidean distance / [Measuring similarity](#)
- European Central Bank
 - URL / [Financial data sources](#)
- evaluation, hidden Markov model (HMM)

- about / [Evaluation \(CF-1\)](#)
 - alpha (forward pass) / [Alpha \(forward pass\)](#)
 - beta (backward pass) / [Beta \(backward pass\)](#)
- evaluation, multilayer perceptron (MLP)
 - about / [Evaluation](#)
 - execution profile / [Execution profile](#)
 - learning rate, impact / [Impact of learning rate](#)
 - momentum factor, impact / [Impact of the momentum factor](#)
 - number of hidden layers, impact / [Impact of the number of hidden layers](#)
 - test case / [Test case](#)
- exception handling / [Implementation](#)
- exchange-traded funds (ETFs) / [Test case](#)
- expectation-maximization (EM) / [Training \(CF-2\)](#)
- Expectation-Maximization algorithm
 - about / [Expectation-Maximization \(EM\)](#)
 - Gaussian mixture model / [Gaussian mixture model](#)
 - overview / [EM overview](#)
 - implementation / [Implementation](#)
 - classification / [Classification](#)
 - testing / [Testing](#)
- explicit model
 - about / [Monadic data transformation](#)
- explicit models
 - about / [Explicit models](#)
- exponential moving average
 - about / [Exponential moving average](#)
- extended Kalman filter (EKF) / [Benefits and drawbacks](#)
- Extended Kalman Filters (EKF) / [The discrete Kalman filter](#)
- Extended Learning Classifiers (XCS)
 - about / [Learning classifier systems](#)

F

- 1-fold cross-validation / [One-fold cross-validation](#)
- F-score
 - for binomial classification / [F-score for binomial classification](#)
 - for multinomial classification / [F-score for multinomial classification](#)
- F1-measure / [Key quality metrics](#)
- F1-score / [Key quality metrics](#)
- False Negatives (FNs) / [Key quality metrics](#)
- false positive rate (FPR) / [Area under ROC](#)
 - about / [Training the model](#)
- False Positives (FPs) / [Key quality metrics](#)
- Fast Fisher-Yates shuffle / [Fast Fisher-Yates shuffle](#)
- Fast Fourier Transform (FFT) / [Discrete Fourier transform \(DFT\)](#)
- feature extraction
 - about / [Extracting features](#)
- feature functions / [Linear chain CRF](#)
- features
 - about / [What is a model?](#)
- features extraction
 - automation / [Test case 2 – features selection](#)
- features selection
 - about / [Selecting features](#)
- federal fund rate (FDR)
 - about / [Introducing the multinomial Naïve Bayes](#)
- feed-forward neural networks (FFNN)
 - about / [Feed-forward neural networks \(FFNN\)](#)
 - biological background / [The biological background](#)
 - mathematical background / [Mathematical background](#)
 - without hidden layers / [The multilayer perceptron \(MLP\)](#)
- feed-forward sparse autoencoder / [Feed-forward sparse, undercomplete autoencoder](#)
- filtering
 - versus smoothing / [The discrete Kalman filter](#)

- filtering algorithms / [Modularizing](#)
- final val
 - versus val / [C-penalty and margin](#)
- finances 101
 - about / [Finances 101](#)
 - fundamental analysis / [Fundamental analysis](#)
 - technical analysis / [Technical analysis](#)
 - options trading / [Options trading](#)
 - financial data sources / [Financial data sources](#)
- financial data sources
 - about / [Financial data sources](#)
- financial metrics
 - earnings per share (EPS) / [Fundamental analysis](#)
 - Price/Earnings Ratio (PE) / [Fundamental analysis](#)
 - Price/Sales Ratio (PS) / [Fundamental analysis](#)
 - Price/Book Value Ratio (PB) / [Fundamental analysis](#)
 - Price to Earnings/Growth (PEG) / [Fundamental analysis](#)
 - Operating Income / [Fundamental analysis](#)
 - Net Sales / [Fundamental analysis](#)
 - Operating Profit Margin / [Fundamental analysis](#)
 - Net Profit Margin / [Fundamental analysis](#)
 - Short Interest / [Fundamental analysis](#)
 - Short Interest Ratio / [Fundamental analysis](#)
 - Cash per Share / [Fundamental analysis](#)
 - Pay-out Ratio / [Fundamental analysis](#)
 - Annual Dividend yield / [Fundamental analysis](#)
 - Dividend Coverage Ratio / [Fundamental analysis](#)
 - Growth Domestic Product (GDP) / [Fundamental analysis](#)
 - Consumer Price Index (CPI) / [Fundamental analysis](#)
 - Federal Fund rate / [Fundamental analysis](#)
- first-order discrete Markov chain
 - about / [The first-order discrete Markov chain](#)
- first order predicate logic / [First order predicate logic](#)
- fitness function
 - about / [Fitness score](#)
 - fixed fitness function / [Fitness score](#)
 - evolutionary fitness function / [Fitness score](#)

- approximate fitness function / [Fitness score](#)
- fixed lag smoothing / [Fixed lag smoothing](#)
 - complex strategies / [Fixed lag smoothing](#)
- Fn score / [Key quality metrics](#)
- forms, models
 - parameteric / [What is a model?](#)
 - differential / [What is a model?](#)
 - probabilistic / [What is a model?](#)
 - graphical / [What is a model?](#)
 - directed graphs / [What is a model?](#)
 - numerical methods / [What is a model?](#)
 - chemistry / [What is a model?](#)
 - taxonomy / [What is a model?](#)
 - grammar / [What is a model?](#)
 - lexicon / [What is a model?](#)
 - inference logic / [What is a model?](#)
- Fourier analysis
 - about / [Fourier analysis](#)
 - Discrete Fourier transform (DFT) / [Discrete Fourier transform \(DFT\)](#)
 - DFT-based filtering / [DFT-based filtering](#)
 - market cycles, detecting / [Detection of market cycles](#)
- Fourier transform / [Fourier analysis](#)
- frameworks
 - about / [Tools and frameworks](#)
 - Java / [Java](#)
 - Scala / [Scala](#)
 - SBT / [Simple build tool](#)
 - Apache Commons Math / [Apache Commons Math](#)
 - JFreeChart / [JFreeChart](#)
 - libraries / [Other libraries and frameworks](#)
 - tools / [Other libraries and frameworks](#)
- frequency domain / [Discrete Fourier transform \(DFT\)](#)
- function language
 - Scala / [Scala as a functional language](#)
- functor
 - about / [Scala as a functional language](#)

- functors
 - about / [Functors](#)
- fundamental analysis
 - about / [Fundamental analysis](#)
- futures
 - about / [Futures](#)
 - blocking / [Blocking on futures](#)
 - callbacks / [Future callbacks](#)
 - implementing / [Putting it all together](#)

G

- Gauss-Newton method / [Numerical optimization](#), [Training workflow](#)
- Gauss-Newton technique
 - about / [Gauss-Newton](#)
- Gaussian distribution / [Z-score and Gauss](#)
- Gaussian mixture / [Class likelihood](#)
- Gaussian mixture model
 - about / [Gaussian mixture model](#)
- Gaussian noise / [The transition equation](#)
- Gaussian sampling
 - about / [Gaussian sampling](#)
 - Box-Muller transform / [Box-Muller transform](#)
- generalized autoregressive conditional heteroscedasticity (GARCH) / [Alternative preprocessing techniques](#)
- generalized linear models (GLM) / [Logistic regression](#)
- generative models / [Generative models](#)
- Generic message handler
 - about / [Blocking on futures](#)
- genetic algorithm, implementations
 - about / [Implementation](#)
 - software design / [Software design](#)
 - key components / [Key components](#)
 - selection process / [Selection](#)
 - population growth, controlling / [Controlling population growth](#)
 - configuration / [GA configuration](#)
 - crossover / [Crossover](#)
 - mutation / [Mutation](#)
 - reproduction / [Reproduction](#)
 - solver / [Solver](#)
- genetic algorithms (GA)
 - about / [Genetic algorithms and machine learning](#)
 - for trading strategies / [GA for trading strategies](#)
 - advantages / [Advantages and risks of genetic algorithms](#)
 - risks / [Advantages and risks of genetic algorithms](#)

- genetic algorithms components
 - about / [Genetic algorithm components](#)
 - genetic encoding / [Encodings](#)
 - genetic operators / [Genetic operators](#)
 - fitness function / [Fitness score](#)
- genetic encoding
 - about / [Encodings](#)
 - value encoding / [Value encoding](#)
 - predicate encoding / [Predicate encoding](#)
 - solution encoding / [Solution encoding](#)
 - encoding scheme / [The encoding scheme](#)
- genetic operators
 - about / [Genetic operators](#)
 - selection / [Genetic operators](#), [Selection](#)
 - crossover / [Genetic operators](#), [Crossover](#)
 - mutation / [Genetic operators](#), [Mutation](#)
- Gibbs sampling / [Test](#)
- G measure / [Key quality metrics](#)
- GNU Lesser General Public License (LGPL)
 - about / [Licensing](#)
- goal state
 - about / [Putting it all together](#)
- Google finances
 - reference link / [Financial data sources](#)
- Googles Breeze
 - reference link / [Abstraction](#)
- gradient descent methods
 - steepest descent / [Steepest descent](#)
 - conjugate gradient / [Conjugate gradient](#)
 - stochastic gradient descent / [Stochastic gradient descent](#)
- graph-structured CRF
 - about / [Introduction to CRF](#)
- graphical models
 - about / [Probabilistic graphical models](#)
- GraphX
 - about / [Overview](#)
- greedy approach

- about / [Solver](#)
- grid search
 - about / [Grid search](#)
- growth domestic product (GDP)
 - about / [Introducing the multinomial Naïve Bayes](#)

H

- Hadoop Distributed File System (HDFS) / [Step 2 – loading data](#)
- Hadoop distributed file system (HDFS)
 - about / [Overview](#)
- hard margin / [The separable case \(hard margin\)](#)
- heat kernel function / [Kernel monadic composition](#)
- Hessian matrix / [Jacobian and Hessian matrices](#)
- hidden layers / [The multilayer perceptron \(MLP\)](#)
- hidden Markov model (HMM)
 - about / [The hidden Markov model \(HMM\)](#)
 - notation / [Notation](#)
 - lambda model / [The lambda model](#)
 - design / [Design](#)
 - evaluation (CF-1) / [Evaluation \(CF-1\)](#)
 - training (CF-2) / [Training \(CF-2\)](#)
 - decoding (CF-3) / [Decoding \(CF-3\)](#)
 - implementing / [Putting it all together](#)
 - ViterbiPath class / [Putting it all together](#)
 - ViterbiPath singleton / [Putting it all together](#)
 - test case / [Test case 1 – Training, Test case 2 – Evaluation](#)
 - as filtering technique / [HMM as filtering technique](#)
 - versus conditional random field (CRF) / [Comparing CRF and HMM](#)
 - performance consideration / [Performance consideration](#)
- Hidden Naïve Bayes (HNB) / [Training](#)
- higher kinded types (HKTs)
 - about / [Higher kinded types](#)
- hinge loss / [The non-separable case \(soft margin\)](#)
- HMM constructor
 - config argument / [Putting it all together](#)
 - xt argument / [Putting it all together](#)
 - form argument / [Putting it all together](#)
 - quantize argument / [Putting it all together](#)

I

- identically distributed population (i.i.d) / [The purpose of sampling](#)
- IITB CRF Java library
 - evaluation / [Training the CRF model](#)
- immutable statistics / [Immutable statistics](#)
- immutable transformations
 - about / [Explicit models](#)
- implementation, regularized CRF
 - about / [Implementation](#)
 - CRF classifier, configuring / [Configuring the CRF classifier](#)
 - CRF model, training / [Training the CRF model](#)
 - CRF model, applying / [Applying the CRF model](#)
- implicit model
 - about / [Monadic data transformation](#)
- implicit models
 - about / [Implicit models](#)
- incremental EM / [Online EM](#)
- initialization
 - about / [Genetic operators](#)
- input forward propagation, multilayer perceptron (MLP)
 - about / [Step 1 – input forward propagation](#)
 - computational flow / [Computational flow](#)
 - error functions / [Error functions](#)
 - operating modes / [Operating modes](#)
 - softmax / [Softmax](#)
- input value
 - about / [Error handling](#)
- IntelliJ IDEA Scala plugin
 - about / [IntelliJ IDEA Scala plugin](#)
 - reference link / [IntelliJ IDEA Scala plugin](#)
- intercept / [Step 5 – implementing the classifier](#)
- iterators / [Lazy views](#)

J

- Jacobian J matrix / [Numerical optimization](#)
- Jacobian matrix / [Jacobian and Hessian matrices](#)
- Java
 - about / [Java](#)
 - reference link / [Overview](#)
- Java libraries
 - leveraging / [Leveraging Java libraries](#)
- Java Native Interface (JNI) / [Algebraic and numerical libraries](#)
- Java Virtual Machine (JVM)
 - about / [Overview](#)
- JBlas
 - reference link / [Leveraging Java libraries](#)
- jBlas
 - reference link / [Algebraic and numerical libraries](#)
- Jensen-Shannon metric
 - about / [The divergences](#)
- JFreeChart
 - about / [JFreeChart, Plotting data](#)
 - description / [Description](#)
 - licensing / [Licensing](#)
 - installation / [Installation](#)
 - URL, for installation / [Installation](#)

K

- K-armed bandit
 - about / [K-armed bandit](#)
 - exploration-exploitation trade-offs / [Exploration-exploitation trade-offs](#)
 - expected cumulative regret / [Expected cumulative regret](#)
 - Bayesian Bernoulli bandits / [Bayesian Bernoulli bandits](#)
 - Epsilon-greedy algorithm / [Epsilon-greedy algorithm](#)
- K-fold cross-validation / [K-fold cross-validation](#)
- K-means
 - with MLlib / [K-means using MLlib](#)
- K-means algorithm
 - defining / [Defining the algorithm](#)
 - steps / [Defining the algorithm](#)
 - exit condition / [Step 2 – Clusters assignment](#)
- K-means clustering
 - about / [K-means](#)
 - similarity measures / [Measuring similarity](#)
 - evaluation, setting up / [Evaluation](#)
 - results, evaluating / [The results](#)
 - number of clusters, tuning / [Tuning the number of clusters](#)
 - output, validating / [Validation](#)
- K-means components
 - creating / [Creating K-means components](#)
- Kalman filter / [The discrete Kalman filter](#)
 - recursive characteristic / [The discrete Kalman filter](#)
 - optimal characteristic / [The discrete Kalman filter](#)
 - non-linear systems / [The discrete Kalman filter](#)
- Kalman smoothing / [Kalman smoothing](#)
- Kernel functions
 - about / [Kernel functions](#)
 - overview / [Overview](#)
 - discriminative kernels / [Common discriminative kernels](#)
 - monadic composition / [Kernel monadic composition](#)

- monadic composition, interpretation / [Kernel monadic composition](#)
 - in SVM / [Kernel monadic composition](#)
- kernel functions, types
 - linear kernel (dot product) / [Common discriminative kernels](#)
 - polynomial kernel / [Common discriminative kernels](#)
 - radial basis function (RBF) / [Common discriminative kernels](#)
 - sigmoid kernel / [Common discriminative kernels](#)
 - Laplacian kernel / [Common discriminative kernels](#)
 - log kernel / [Common discriminative kernels](#)
- Kernel PCA
 - about / [Kernel PCA](#)
- kernels, types
 - probabilistic kernels / [Common discriminative kernels](#)
 - smoothing kernels / [Common discriminative kernels](#)
 - reproducible kernel Hilbert spaces / [Common discriminative kernels](#)
- kernel trick
 - about / [The kernel trick](#)
- key components
 - population / [Population](#)
 - chromosomes / [Chromosomes](#)
 - genes / [Genes](#)
- Kohonen's self-organizing maps / [Manifolds](#)
- Kullback-Leibler (KL) divergence
 - about / [The divergences, The Kullback-Leibler divergence](#)
 - overview / [Overview](#)
 - implementation / [Implementation](#)
 - testing / [Testing](#)
- Kullback-Leibler divergence
 - about / [Kullback-Leibler divergence](#)
 - implementation / [Implementation](#)
- Kullback-Leibler evaluator
 - about / [Kullback-Leibler evaluator](#)

L

- L1 regularization / [Challenging model complexity](#)
- Lagrange multipliers / [Max-margin classification](#), [Lagrange multipliers](#)
- Laplacian Eigenmaps
 - about / [Manifolds](#)
- Laplacian kernel / [Common discriminative kernels](#)
- Lasso regularization / [Ln roughness penalty](#)
- Latent Dirichlet Allocation (LDA)
 - about / [Probabilistic graphical models](#)
- lazy direct acyclic graph (DAG)
 - about / [Use case – continuous parsing](#)
- lazy value trigger / [Step 3 – instantiation](#)
- lazy views / [Lazy views](#)
- LDL decomposition
 - about / [LDL decomposition](#)
- Learning Classifier Systems (LCS)
 - about / [Learning classifier systems](#), [Introduction to LCS](#)
 - components / [Introduction to LCS](#)
 - learning strategy, combining with evolutionary approach / [Combining learning and evolution](#)
 - terminology / [Terminology](#)
 - extended learning classifier systems / [Extended learning classifier systems](#)
 - XCS components / [XCS components](#)
 - portfolio management, application / [Application to portfolio management](#)
 - XCS core data / [XCS core data](#)
 - XCS rules / [XCS rules](#)
 - covering phase / [Covering](#)
 - implementation, example / [Example of implementation](#)
 - benefits / [Benefits and limitations of learning classifier systems](#)
 - limitations / [Benefits and limitations of learning classifier systems](#)
- learning vector quantization (LVQ)
 - about / [K-mean clustering](#)

- least squares problem / [Numerical optimization](#)
- lemmatization / [Basics information retrieval](#)
- Levenberg-Marquardt algorithm
 - about / [Levenberg-Marquardt](#)
- Levenberg-Marquardt method / [Numerical optimization](#), [Training workflow](#)
- Levenberg-Marquardt optimizer / [Alternative preprocessing techniques](#)
- LIBSVM
 - about / [LIBSVM](#)
 - URL / [LIBSVM](#)
 - reference / [LIBSVM](#)
 - need for / [LIBSVM](#)
 - Java code / [LIBSVM](#)
 - svm_node / [Interface to LIBSVM](#)
 - scaling / [Application to risk analysis](#)
- LIBSVM, Java class
 - svm_model / [LIBSVM](#)
 - svm_node / [LIBSVM](#)
 - svm_parameters / [LIBSVM](#)
 - svm_problem / [LIBSVM](#)
 - svm / [LIBSVM](#)
- linear algebra
 - about / [Linear algebra](#)
 - QR decomposition / [QR decomposition](#)
 - LU factorization / [LU factorization](#)
 - LDL decomposition / [LDL decomposition](#)
 - Cholesky factorization / [Cholesky factorization](#)
 - SVD / [Singular Value Decomposition \(SVD\)](#)
 - Eigen-decomposition / [Eigenvalue decomposition](#)
 - algebraic libraries / [Algebraic and numerical libraries](#)
 - numerical libraries / [Algebraic and numerical libraries](#)
- linear chain CRF
 - about / [Introduction to CRF](#), [Linear chain CRF](#)
- linear chain structured graph CRF
 - about / [Introduction to CRF](#)
- linear kernel (dot product) / [Common discriminative kernels](#)
- linear regression

- about / [Linear regression](#)
 - univariate linear regression / [Univariate linear regression](#)
 - ordinary least squares regression (OLS) / [Ordinary least squares \(OLS\) regression](#)
 - concept / [Test case 2 – features selection](#)
 - versus support vector regression (SVR) / [SVR versus linear regression](#)
- linear SVM
 - about / [The linear SVM](#)
 - separable case (hard margin) / [The separable case \(hard margin\)](#)
 - non-separable case (soft margin) / [The non-separable case \(soft margin\)](#)
- Local Linear Embedding
 - about / [Manifolds](#)
- logistic regression
 - about / [Logistic regression](#)
 - logistic function / [Logistic function](#)
 - design / [Design](#)
 - training workflow / [Training workflow](#)
 - classification / [Classification](#)
- logistic regression, test case
 - about / [Let's kick the tires](#)
 - workflow, writing / [Writing a simple workflow](#)
 - issues, scoping / [Step 1 – scoping the problem](#)
 - data, loading / [Step 2 – loading data](#)
 - data, preprocessing / [Step 3 – preprocessing data, Immutable normalization](#)
 - immutable normalization / [Immutable normalization](#)
 - patterns, discovering / [Step 4 – discovering patterns](#)
 - data, analyzing / [Analyzing data](#)
 - data, plotting / [Plotting data](#)
 - model features, visualizing / [Visualizing model features](#)
 - label, visualizing / [Visualizing label](#)
 - classifier, implementing / [Step 5 – implementing the classifier](#)
 - optimizer, selecting / [Selecting an optimizer](#)
 - model, training / [Training the model](#)
 - observations, classifying / [Classifying observations](#)

- model, evaluating / [Step 6 – evaluating the model](#)
- log kernel / [Common discriminative kernels](#)
- loss function / [Selecting an optimizer](#)
- loss function approach
 - about / [Solver](#)
- Lotka-Volterra equation
 - about / [Selection](#)
- LU factorization
 - about / [LU factorization](#)

M

- machine learning
 - need for / [Why machine learning?](#)
 - classification / [Classification](#)
 - prediction / [Prediction](#)
 - optimization / [Optimization](#)
 - regression / [Regression](#)
 - about / [Genetic algorithms and machine learning](#)
- machine learning algorithms
 - taxonomy / [Taxonomy of machine learning algorithms](#)
 - unsupervised learning / [Unsupervised learning](#)
 - supervised learning / [Supervised learning](#)
 - discriminative models / [Discriminative models](#)
 - semi-supervised learning / [Semi-supervised learning](#)
 - reinforcement learning / [Reinforcement learning](#)
- macro formulas
 - for multinomial precision / [F-score for multinomial classification](#)
 - for recall / [F-score for multinomial classification](#)
/ [Area under ROC](#)
- Manhattan distance / [Measuring similarity](#)
- manifolds
 - about / [Manifolds](#)
- Markov chain
 - about / [The hidden Markov model \(HMM\)](#)
- Markov Chain Monte Carlo (MCMC)
 - about / [Markov Chain Monte Carlo \(MCMC\)](#)
 - overview / [Overview](#)
 - Metropolis-Hastings (MH) / [Metropolis-Hastings \(MH\)](#)
 - implementation / [Implementation](#)
 - testing / [Test](#)
/ [Log-likelihood gradient](#)
- Markov decision process (MDP) / [K-armed bandit](#)
- Markov decision processes
 - about / [Markov decision processes](#)

- Markov property / [The Markov property](#)
 - first-order discrete Markov chain / [The first-order discrete Markov chain](#)
- Markov property
 - about / [The Markov property](#)
- master-workers design
 - about / [Master-workers](#)
 - messages exchange / [Messages exchange](#)
 - worker Actors / [Worker Actors](#)
 - workflow controller / [The workflow controller](#)
 - master Actor / [The master Actor](#)
 - with routing / [Master with routing](#)
 - DFT / [Distributed discrete Fourier transform](#)
 - limitations / [Limitations](#)
- mathematical abstractions
 - supporting / [Supporting mathematical abstractions](#)
 - variable declaration / [Step 1 – variable declaration](#)
 - model definition / [Step 2 – model definition](#)
 - instantiation / [Step 3 – instantiation](#)
- mathematical notations / [Mathematical notations for the curious](#)
- mathematics
 - linear algebra / [Linear algebra](#)
 - first order predicate logic / [First order predicate logic](#)
 - Hessian matrix / [Jacobian and Hessian matrices](#)
 - Jacobian matrix / [Jacobian and Hessian matrices](#)
 - optimization techniques / [Summary of optimization techniques](#)
- max-margin classification
 - about / [Max-margin classification](#)
- mean / [Immutable statistics](#)
- mean square error (MSE) / [Bias-variance decomposition](#)
- measurement equation / [The state space estimation](#), [The measurement equation](#)
- measurement noise covariance / [The measurement equation](#)
- memory management
 - about / [Explicit models](#)
- methodology
 - defining / [Defining a methodology](#)

- Metropolis-Hastings (MH) / [Metropolis-Hastings \(MH\)](#)
- mixin composition
 - for ITransform / [Instantiating the workflow](#)
- mixins
 - composing, to build workflow / [Composing mixins to build workflow](#)
 - about / [Composing mixins to build workflow](#)
- mixins linearization
 - about / [Understanding the problem](#)
- MLlib library
 - about / [Overview, MLlib library](#)
 - components / [Overview](#)
 - RDDs, creating / [Creating RDDs](#)
 - using, for K-means / [K-means using MLlib](#)
 - tests / [Tests](#)
- model
 - about / [What is a model?](#)
 - versus design / [Model versus design](#)
- model assessment
 - about / [Assessing a model](#)
 - validation / [Validation](#)
 - area under curves / [Area under the curves](#)
 - cross-validation / [Cross-validation](#)
 - bias-variance decomposition / [Bias-variance decomposition](#)
 - overfitting / [Overfitting](#)
- model categorization
 - about / [Model categorization](#)
 - predictive models / [Model categorization](#)
 - descriptive models / [Model categorization](#)
- model complexity
 - challenging / [Challenging model complexity](#)
- model fitness
 - versus accuracy / [Model fitness](#)
- modeling
 - about / [Modeling](#)
 - / [Model versus design](#)
- models

- forms / [What is a model?](#)
- model validation
 - about / [Validation](#)
 - key quality metrics / [Key quality metrics](#)
 - F-score, for binomial classification / [F-score for binomial classification](#)
 - F-score, for multinomial classification / [F-score for multinomial classification](#)
- modules
 - defining / [Defining modules](#)
- monad
 - about / [Scala as a functional language](#)
- monadic composition
 - about / [Monads](#)
- monadic data transformation
 - about / [Monadic data transformation](#)
- monads
 - about / [Monads, Monads to the rescue](#)
- Monitor class / [Monitor](#)
- Monte Carlo approximation
 - about / [Monte Carlo approximation](#)
 - overview / [Overview](#)
 - implementation / [Implementation](#)
- Monte Carlo EM / [Online EM](#)
- Monte Carlo integration / [Sampling](#)
- morphism
 - about / [Error handling](#)
- moving averages
 - about / [Moving averages](#)
 - simple moving average / [Simple moving average](#)
 - weighted moving average / [Weighted moving average](#)
 - exponential moving average / [Exponential moving average](#)
 - on multi-dimensional time series / [Exponential moving average](#)
- multi-class scoring / [F-score for binomial classification](#)
- multilayer perceptron (MLP)
 - about / [The multilayer perceptron \(MLP\)](#)
 - activation function / [Activation function](#)

- network topology / [Network topology](#)
 - design / [Design](#)
 - configuration / [Configuration](#)
 - network components / [Network components](#)
 - model / [Model](#)
 - problem types (modes) / [Problem types \(modes\)](#)
 - online training, versus batch training / [Online versus batch training](#)
 - epoch, training / [Training epoch](#)
 - training / [Training and classification](#)
 - classification / [Training and classification](#)
 - evaluation / [Evaluation](#)
 - limitations / [Benefits and limitations](#)
 - benefits / [Benefits and limitations](#)
- multinomial Naïve Bayes
 - about / [Introducing the multinomial Naïve Bayes](#)
 - formalism / [Formalism](#)
 - frequentist perspective / [The frequentist perspective](#)
 - predictive model / [The predictive model](#)
 - zero-Frequency problem / [The zero-frequency problem](#)
- multivariate Bernoulli classification
 - about / [Multivariate Bernoulli classification](#)
 - model / [Model](#)
 - implementation / [Implementation](#)
- mutation
 - about / [Mutation](#)
 - population / [Population](#)
 - chromosome / [Chromosomes](#)
 - genes / [Genes](#)
- mutual information
 - about / [The divergences](#), [The mutual information](#)

N

- n-fold cross-validation / [Application to risk analysis](#)
- NASDAQ
 - URL / [Financial data sources](#)
- natural language processing (NLP) / [The feature functions model](#)
- natural selection
 - about / [Selection](#)
- Naïve Bayes
 - pros and cons / [Pros and cons](#)
- Naïve Bayes classifier
 - used, for text mining / [Naïve Bayes and text mining](#)
- Naïve Bayes classifiers
 - about / [Naïve Bayes classifiers](#)
 - multinomial Naïve Bayes / [Introducing the multinomial Naïve Bayes](#)
 - implementation / [Implementation](#)
 - design / [Design](#)
 - training / [Training](#)
 - classification / [Classification](#)
 - F1 Validation / [F1 Validation](#)
 - features extract / [Features extraction](#)
 - testing / [Testing](#)
- Naïve Bayes models
 - about / [Probabilistic graphical models](#)
- network components, multilayer perceptron (MLP)
 - about / [Network components](#)
 - network topology / [Network topology](#)
 - hidden layers / [Input and hidden layers](#)
 - input layers / [Input and hidden layers](#)
 - output layers / [Output layer](#)
 - synapses / [Synapses](#)
 - connections / [Connections](#)
 - weights initialization / [Weights initialization](#)
- news

- macro trends / [Naïve Bayes and text mining](#)
 - micro updates / [Naïve Bayes and text mining](#)
- Nondeterministic Polynomial (NP) problems
 - about / [NP problems](#)
 - categories / [NP problems](#)
- nonlinear least squares minimization
 - about / [Nonlinear least squares minimization](#)
 - Gauss-Newton technique / [Gauss-Newton](#)
 - Levenberg-Marquardt algorithm / [Levenberg-Marquardt](#)
- non linear models
 - about / [Nonlinear models](#)
 - Kernel PCA / [Kernel PCA](#)
 - manifolds / [Manifolds](#)
- nonlinear SVM
 - about / [The nonlinear SVM](#)
 - max-margin classification / [Max-margin classification](#)
 - kernel trick / [The kernel trick](#)
- normalization
 - about / [Immutable normalization](#)
- normalized inner product / [Measuring similarity](#)
- null frequencies
 - handling / [Implementation](#)
- numerical libraries
 - about / [Algebraic and numerical libraries](#)
- numerical optimization
 - about / [Numerical optimization](#)
 - Newton (or second-order techniques) / [Numerical optimization](#)
 - Quasi-newton (or first-order techniques) / [Numerical optimization](#)
- Nyquist / [Discrete Fourier transform \(DFT\)](#)

O

- object oriented language
 - Scala / [Scala as an object oriented language](#)
- observation
 - about / [Extracting features](#)
- one-class SVC
 - anomaly detection / [Anomaly detection with one-class SVC](#)
- online EM / [Online EM](#)
- online training
 - versus batch training / [Online versus batch training](#)
 - about / [Online versus batch training](#)
- operations, time series
 - transpose operator / [Transpose operator](#)
 - differential operator / [Differential operator](#)
- optimization techniques
 - gradient descent methods / [Steepest descent](#)
 - Quasi Newton algorithms / [Quasi-Newton algorithms](#)
 - nonlinear least squares minimization / [Nonlinear least squares minimization](#)
 - Lagrange multipliers / [Lagrange multipliers](#)
 - dynamic programming, overview / [Overview dynamic programming](#)
- options trading
 - about / [Options trading](#)
- option trading
 - Q-learning, used / [Option trading using Q-learning](#)
 - option property / [Option property](#)
 - option model / [Option model](#)
 - quantization / [Quantization](#)
- ordinary least squares regression (OLS)
 - about / [Ordinary least squares \(OLS\) regression](#)
 - design / [Design](#)
 - implementation / [Implementation](#)
 - test case / [Test case 1 – trending, Test case 2 – features selection](#)

- output unit activation function / [Activation function](#)
- output value
 - about / [Error handling](#)
- overcomplete autoencoder / [Categorization](#)
- overfitting / [Overfitting](#)
 - empirical estimation / [Bias-variance decomposition](#)
 - versus sparsity / [Sparsity updating equations](#)
- overloading / [Overloading](#)

P

- padding / [Value encoding](#)
- parallel collections
 - about / [Parallel collections](#)
 - processing / [Processing a parallel collection](#)
 - benchmark framework / [Benchmark framework](#)
 - performance evaluation / [Performance evaluation](#)
- Parallel Colt
 - reference link / [Leveraging Java libraries](#)
- parent chromosomes
 - preserving / [Crossover](#)
- partial functions
 - reusability / [Error handling](#)
 - about / [Error handling](#)
 - runtime validation / [Error handling](#)
 - versus partially applied functions / [DFT-based filtering](#)
- Partial Least Square Regression (PLSR) / [Validation](#)
- partially applied functions
 - versus partial functions / [DFT-based filtering](#)
- partially connected networks / [Network topology](#)
- particle filter / [Alternative preprocessing techniques](#)
- penalized least squares regression / [Ln roughness penalty](#)
- performance evaluation, Spark
 - about / [Performance evaluation](#)
 - tuning parameters / [Tuning parameters](#)
 - considerations / [Performance considerations](#)
 - pros and cons / [Pros and cons](#)
- plate model
 - about / [Probabilistic graphical models](#)
- polynomial kernel / [Common discriminative kernels](#)
- population growth
 - controlling / [Controlling population growth](#)
- pre-processing techniques
 - alternative techniques / [Alternative preprocessing techniques](#)

- precision / [Key quality metrics](#)
- precision-recall curve (PRC) / [Area under PRC](#)
- Predicted Residual Error Sum of Squares (PRESS) / [Validation](#)
- predictive models / [Model categorization](#)
- price pattern
 - about / [Price patterns](#)
- principal components analysis (PCA)
 - about / [Principal components analysis \(PCA\)](#)
 - algorithm / [Algorithm](#)
 - covariance matrix / [Algorithm](#)
 - implementation / [Implementation](#)
 - test case / [Test case](#)
 - evaluation / [Evaluation](#)
 - extending / [Extending PCA](#)
 - validation / [Validation](#)
 - categorical features / [Categorical features](#)
 - performance / [Performance](#)
- probabilistic graphical models
 - about / [Probabilistic graphical models](#)
 - directed graphical models / [Probabilistic graphical models](#)
 - Bayesian networks / [Probabilistic graphical models](#)
 - Naïve Bayes models / [Probabilistic graphical models](#)
- probabilistic kernels / [Common discriminative kernels](#)
- projection
 - about / [Functors](#)
- proposal distribution / [Overview](#)
- Proteins / [Overview](#)
- protein sequence annotation / [Overview](#)
- Python
 - reference link / [Overview](#)

Q

- Q-learning, implementation
 - about / [Implementation](#)
 - software design / [Software design](#)
 - states / [The states and actions](#)
 - actions / [The states and actions](#)
 - search space / [The search space](#)
 - action-value / [The policy and action-value](#)
 - policy / [The policy and action-value](#)
 - components / [The Q-learning components](#)
 - training / [The Q-learning training](#)
 - tail recursion / [Tail recursion to the rescue](#)
 - validation / [Validation](#)
 - prediction / [The prediction](#)
- Q-learning algorithm
 - about / [A solution – Q-learning](#)
 - terminology / [Terminology](#)
 - concept / [Concept](#)
 - value of policy / [Value of policy](#)
 - Bellman optimality equations / [Bellman optimality equations](#)
 - temporal difference, for model free learning / [Temporal difference for model-free learning](#)
 - action-value iterative update / [Action-value iterative update](#)
 - used, for option trading / [Option trading using Q-learning](#)
- QR Decomposition
 - about / [QR decomposition](#)
- QStar class / [The Viterbi algorithm](#)
- Quandl
 - URL / [Financial data sources](#)
- quantization / [Value encoding, Quantization](#)
- Quasi Newton algorithms
 - BFGS / [BFGS](#)
 - L-BFGS / [L-BFGS](#)

R

- radial basis function (RBF) / [Common discriminative kernels](#)
 - terminology / [Common discriminative kernels](#)
- recall / [Key quality metrics](#)
- receiver operating characteristic (ROC)
 - about / [Training the model](#)
- Receiver Operating Characteristics (ROC) / [Area under ROC](#)
- recombination
 - about / [Evolutionary computing](#)
- reconstruction error minimization
 - about / [Step 3 – Reconstruction error minimization](#)
 - tail recursive implementation / [Tail recursive implementation](#)
 - iterative implementation / [Iterative implementation](#)
- recursive algorithm
 - about / [The recursive algorithm](#)
 - prediction / [Prediction](#)
 - correction / [Correction](#)
 - Kalman smoothing / [Kalman smoothing](#)
 - fixed lag smoothing / [Fixed lag smoothing](#)
 - experimentation / [Experimentation](#)
 - benefits / [Benefits and drawbacks](#)
 - drawbacks / [Benefits and drawbacks](#)
- regression
 - about / [Regression](#)
- regression model / [Design](#)
- regularization
 - about / [Regularization, Ln roughness penalty](#)
 - Ln roughness penalty / [Ln roughness penalty](#)
 - in machine learning / [Ln roughness penalty](#)
 - model estimation / [Ln roughness penalty](#)
 - feature selection / [Ln roughness penalty](#)
 - overfitting / [Ln roughness penalty](#)
 - computation / [Ln roughness penalty](#)
 - ridge regression / [Ridge regression](#)

- regularized autoencoder / [Categorization](#)
- regularized CRF
 - text analytics / [Regularized CRF and text analytics](#)
 - feature functions model / [The feature functions model](#)
 - design / [Design](#)
 - implementation / [Implementation](#)
 - testing / [Tests](#)
- Reinforcement learning
 - Q-learning, implementation / [Implementation](#)
 - Q-learning, used for option trading / [Option trading using Q-learning](#)
- reinforcement learning / [Model categorization](#), [Reinforcement learning](#), [K-armed bandit](#)
 - about / [Reinforcement learning](#)
 - Q-learning algorithm / [A solution – Q-learning](#)
 - implementing / [Putting it all together](#)
 - evaluation / [Evaluation](#)
 - pros and cons / [Pros and cons of reinforcement learning](#)
- relative fitness degradation
 - about / [Selection](#)
- relative strength index (RSI) / [Terminology](#)
- replicate / [Resampling](#)
- reproducible kernel Hilbert spaces / [Common discriminative kernels](#)
- resampling / [Overview](#)
- resilient distributed dataset (RDD)
 - about / [Overview](#), [Apache Spark core](#), [Using Spark shell](#)
 - creating / [Creating RDDs](#)
- Restricted Boltzmann machines (RBMs) / [Restricted Boltzmann Machines \(RBMs\)](#)
- reusable ML pipelines
 - about / [Reusable ML pipelines](#)
 - Apache Spark application, debugging with ScalaTest / [Apache Spark and ScalaTest](#)
- reusable ML transforms
 - about / [Reusable ML transforms](#)
 - features, encoding / [Encoding features](#)
 - model, training / [Training the model](#)

- predictive model / [Predictive model](#)
- summary statistics, training / [Training summary statistics](#)
- model, validating / [Validating the model](#)
- grid search / [Grid search](#)
- ridge regression / [Ln roughness penalty](#)
 - about / [Ridge regression](#)
 - design / [Design](#)
 - implementation / [Implementation](#)
 - test case / [Test case](#)

S

- @specialized annotation / [Discrete Fourier transform \(DFT\)](#)
- sampling
 - purpose / [The purpose of sampling](#)
- Scala
 - about / [Why Scala?](#), [Scala](#), [Scala](#)
 - used, as functional language / [Scala as a functional language](#)
 - abstraction / [Abstraction](#)
 - HKTs / [Higher kinded types](#)
 - functors / [Functors](#)
 - monads / [Monads](#)
 - used, as object oriented language / [Scala as an object oriented language](#)
 - used, as scalable language / [Scala as a scalable language](#)
 - eclipse Scala IDE / [Eclipse Scala IDE](#)
 - IntelliJ IDEA Scala plugin / [IntelliJ IDEA Scala plugin](#)
 - time series / [Time series in Scala](#)
 - object, creating / [Object creation](#)
 - Streams / [Streams](#)
 - parallel collections / [Parallel collections](#)
 - reference link / [Overview](#)
- scalability
 - with Actors / [Scalability with Actors](#)
 - Actor model / [The Actor model](#)
 - partitioning / [Partitioning](#)
 - reactive programming / [Beyond Actors – reactive programming](#)
- scalable language
 - Scala / [Scala as a scalable language](#)
- ScalaNLP / [Other libraries and frameworks](#)
 - URL / [Algebraic and numerical libraries](#)
- Scala programming
 - about / [Scala programming](#)
 - libraries / [List of libraries and tools](#)
 - tools / [List of libraries and tools](#)

- code snippet fromat / [Code snippets format](#)
- Scala reactive library
 - example, reference link / [Beyond Actors – reactive programming](#)
- Scala standard library
 - reference link / [Scala](#)
- Scalaz
 - reference link / [Abstraction](#)
- scientific model
 - about / [What is a model?](#)
- selection process
 - about / [Selection](#)
- self-reference
 - about / [Composing mixins to build workflow](#)
- semi-supervised learning / [Semi-supervised learning](#)
- Sequential Minimal Optimization (SMO) / [The non-separable case \(soft margin\)](#), [LIBSVM](#)
- service level agreement (SLA)
 - need for / [Why streaming?](#)
- shared variables
 - about / [Shared variables](#)
 - broadcast values / [Shared variables](#)
 - accumulator variables / [Shared variables](#)
- shrinkage
 - about / [Ln roughness penalty](#)
- sigmoid activation
 - versus tanh / [Weight sharing](#)
- sigmoid kernel / [Common discriminative kernels](#)
- similarity
 - visualization / [Overview](#)
- similarity measures
 - Manhattan distance / [Measuring similarity](#)
 - Euclidean distance / [Measuring similarity](#)
 - normalized inner product / [Measuring similarity](#)
- simple build tool (sbt)
 - about / [Deploying Spark](#)
- Simple Build Tool (SBT)
 - about / [Simple build tool](#)

- reference link / [Simple build tool](#)
- simple moving average
 - about / [Simple moving average](#)
- singular value decomposition (SVD) / [Performance](#)
- Singular Value Decomposition (SVD)
 - about / [Singular Value Decomposition \(SVD\)](#)
- smoothing
 - versus filtering / [The discrete Kalman filter](#)
- smoothing kernels / [Common discriminative kernels](#)
- soft margin / [The non-separable case \(soft margin\)](#)
- software developer
 - about / [Defining a methodology](#)
- source code, Scala
 - about / [Source code](#)
 - conventions / [Convention](#)
 - context bounds / [Context bounds](#)
 - presentation / [Presentation](#)
 - primitives / [Primitives and implicits](#)
 - implicits / [Primitives and implicits](#)
 - immutability / [Immutability](#)
- SparkSQL
 - about / [Overview](#)
- Spark Streaming
 - about / [Design for reusing Streams memory](#)
- sparse autoencoder
 - about / [Sparse autoencoder](#)
 - / [Categorization](#)
- sparsity
 - versus overfitting / [Sparsity updating equations](#)
- sparsity updating equations / [Sparsity updating equations](#)
- spectral density estimation / [Fourier analysis](#)
- Spectral theory / [Fourier analysis](#)
- stackable trait injection
 - about / [Composing mixins to build workflow](#)
- state space estimation
 - about / [The state space estimation](#)
 - transition equation / [The state space estimation](#), [The transition](#)

equation

- measurement equation / [The state space estimation](#), [The measurement equation](#)
- steepest descent / [Steepest descent](#)
- stemming / [Basics information retrieval](#)
- steps, K-means algorithms
 - clusters configuration / [Step 1 – Clusters configuration](#)
 - clusters assignment / [Step 2 – Clusters assignment](#)
 - reconstruction error minimization / [Step 3 – Reconstruction error minimization](#)
 - classification / [Step 4 – Classification](#)
- stepwise EM / [Online EM](#)
- stimuli / [The biological background](#)
- stochastic autoencoder / [Categorization](#)
- stochastic gradient descent / [Stochastic gradient descent](#)
- Stochastic Gradient Descent (SGD) / [Selecting an optimizer](#)
- streaming engine
 - about / [Streaming engine](#)
 - need for / [Why streaming?](#)
 - batch processing / [Batch and real-time processing](#)
 - real-time processing / [Batch and real-time processing](#)
 - architecture / [Architecture overview](#)
 - discretized streams (DStreams) / [Discretized streams](#)
 - continuous parsing, use case / [Use case – continuous parsing](#)
 - checkpointing / [Checkpointing](#)
- Streams
 - about / [Streams](#)
 - memory, allocating / [Memory on demand](#)
 - memory, reusing designs / [Design for reusing Streams memory](#)
- streams / [Lazy views](#)
- subject-matter expert
 - about / [Defining a methodology](#)
- sum of the squared error (SSE) / [Online versus batch training](#)
- supervised learning
 - about / [Supervised learning](#)
 - generative models / [Generative models](#)
- support vector classifier (SVC)

- about / [Support vector classifier \(SVC\)](#)
 - binary SVC / [The binary SVC](#)
- support vector machine (SVM)
 - about / [The support vector machine \(SVM\)](#)
 - optional mathematical formulation / [The support vector machine \(SVM\)](#)
 - linear SVM / [The linear SVM](#)
 - nonlinear SVM / [The nonlinear SVM](#)
 - support vector classifier (SVC) / [Support vector classifier \(SVC\)](#)
 - anomaly detection, with one-class SVC / [Anomaly detection with one-class SVC](#)
 - support vector regression (SVR) / [Support vector regression \(SVR\)](#)
 - performance considerations / [Performance considerations](#)
- support vector regression (SVR)
 - about / [Support vector regression \(SVR\)](#)
 - overview / [Overview](#)
 - versus linear regression / [SVR versus linear regression](#)
 - L2 regularization / [SVR versus linear regression](#)
- SVM dual problem / [Max-margin classification](#)
- SVM model
 - accuracy / [Training](#)

T

- tagging / [The feature functions model](#)
- tagging model / [Basics information retrieval](#)
- tail recursion
 - about / [Tail recursion to the rescue](#)
- tanh
 - versus sigmoid activation / [Weight sharing](#)
- technical analysis
 - about / [Technical analysis](#)
 - terminology / [Terminology](#)
 - trading data / [Trading data](#)
 - trading signal / [Trading signal and strategy](#)
 - trading strategy / [Trading signal and strategy](#)
 - price patterns / [Price patterns](#)
- temporal difference
 - for model free learning / [Temporal difference for model-free learning](#)
- tensors
 - about / [Manifolds](#)
- test case, multilayer perceptron (MLP)
 - about / [Test case](#)
 - implementation / [Implementation](#)
 - models evaluation / [Models evaluation](#)
 - hidden layers' architecture, impact / [Impact of hidden layers' architecture](#)
- testing, regularized CRF
 - about / [Tests](#)
 - convergence profile, training / [The training convergence profile](#)
 - training set size impact, evaluating / [Impact of the size of the training set](#)
 - L2 regularization factor, evaluating / [Impact of L2 regularization factor](#)
- text analytics
 - about / [Regularized CRF and text analytics](#)

- text mining
 - with Naïve Bayes classifier / [Naïve Bayes and text mining](#)
 - information retrieval / [Basics information retrieval](#)
 - implementation / [Implementation](#)
 - documents, analyzing / [Analyzing documents](#)
 - relative terms frequency, extracting / [Extracting relative terms frequency](#)
 - features, generating / [Generating the features](#)
 - testing / [Testing](#)
 - textual information, retrieving / [Retrieving textual information](#)
 - classifier, evaluating / [Evaluating text mining classifier](#)
- theory of evolution
 - about / [Evolution](#)
 - origin / [The origin](#)
 - Nondeterministic Polynomial (NP) problems / [NP problems](#)
 - evolutionary computing / [Evolutionary computing](#)
- Thompson sampling
 - about / [Thompson sampling](#)
 - Bandit context / [Bandit context](#)
 - prior/posterior beta distribution / [Prior/posterior beta distribution](#)
 - implementation / [Implementation](#)
 - simulated exploration / [Simulated exploration and exploitation](#)
 - simulated exploitation / [Simulated exploration and exploitation](#)
 - versus UCB1 algorithm / [Implementation](#)
- time-domain function / [Discrete Fourier transform \(DFT\)](#)
- time dependency model / [The measurement equation](#)
- time series
 - about / [Time series in Scala](#)
 - context bound / [Context bounds](#)
 - types / [Types and operations](#)
 - operations / [Types and operations](#)
 - lazy views / [Lazy views](#)
- trading data
 - about / [Trading data](#)
- trading signal
 - about / [Trading signal and strategy](#)
- trading strategies

- GA / [GA for trading strategies](#)
 - definition / [Definition of trading strategies](#)
 - operators / [Trading operators](#)
 - cost function / [The cost function](#)
 - market signals / [Market signals](#)
 - about / [Trading strategies](#)
 - signal encoding / [Signal encoding](#)
 - test case / [Test case – Fall 2008 market crash](#)
 - creating / [Creating trading strategies](#)
 - optimizer, configuring / [Configuring the optimizer](#)
 - finding / [Finding the best trading strategy](#)
 - tests / [Tests](#)
 - weighted score / [The weighted score](#)
 - unweighted score / [The unweighted score](#)
- trading strategy
 - about / [Trading signal and strategy](#)
- training, hidden Markov model (HMM)
 - about / [Training \(CF-2\)](#)
 - Baum-Welch estimator (EM) / [Baum-Welch estimator \(EM\)](#)
- training, Naïve Bayes classifiers
 - about / [Training](#)
 - Likelihood class / [Class likelihood](#)
 - binomial model / [Binomial model](#)
 - multinomial model / [Multinomial model](#)
 - classifier components / [Classifier components](#)
- training files
 - raw dataset / [Training the CRF model](#)
 - tagged dataset / [Training the CRF model](#)
- training workflow, logistic regression
 - about / [Training workflow](#)
 - optimizer, configuring / [Step 1 – configuring the optimizer](#)
 - Jacobian matrix, computing / [Step 2 – computing the Jacobian matrix](#)
 - convergence of optimizer, managing / [Step 3 – managing the convergence of optimizer](#)
 - least squares problem, defining / [Step 4 – defining the least squares problem](#)

- sum of square errors, minimizing / [Step 5 – minimizing the sum of square errors](#)
 - testing / [Test](#)
- traits
 - about / [Composing mixins to build workflow](#)
- transition equation
 - about / [The state space estimation](#), [The transition equation](#)
- transpose operator / [Transpose operator](#)
- transposition operator
 - about / [Genetic operators](#)
- TrueFx (Forex)
 - URL / [Financial data sources](#)
- True Negatives (TNs) / [Key quality metrics](#)
- true positive rate (TPR)
 - about / [Training the model](#)
- True Positives (TPs) / [Key quality metrics](#)
- tuning, genetic algorithm / [Mutation](#)
- tuning parameters
 - about / [Performance evaluation](#)
- Twitters Algebird
 - reference link / [Abstraction](#)
- two-step lag smoothing / [Experimentation](#)

U

- UCB1 algorithm
 - versus Thompson sampling / [Implementation](#)
- unapply method
 - about / [Genes](#)
- undercomplete autoencoder / [Undercomplete autoencoder](#),
[Categorization](#), [Feed-forward sparse](#), [undercomplete autoencoder](#)
- univariate linear regression
 - about / [Univariate linear regression](#)
 - implementation / [Implementation](#)
 - test case / [Test case](#)
- unsupervised learning / [Unsupervised learning](#)
 - data clustering / [Clustering](#)
 - dimension reduction / [Dimension reduction](#)
- upper bound confidence
 - about / [Upper bound confidence](#)
 - confidence interval / [Confidence interval](#)
 - for Bernoulli variables / [Confidence interval](#)
 - implementation / [Implementation](#)
- utility classes, Scala programming
 - about / [Utility classes](#)
 - data extraction / [Data extraction](#)
 - financial data sources / [Financial data sources](#)
 - documents extraction / [Documents extraction](#)
 - DMatrix class / [DMatrix class](#)
 - Counter class / [Counter](#)
 - Monitor class / [Monitor](#)

V

- val
 - def, overriding with / [Understanding the problem](#)
 - versus final val / [C-penalty and margin](#)
- variables
 - about / [What is a model?](#)
- variance / [Immutable statistics](#)
- variance-bias trade-off / [Bias-variance decomposition](#)
- vector quantization
 - about / [K-mean clustering](#)
- views / [Lazy views](#)
- Viterbi algorithm / [The Viterbi algorithm](#)
 - psi matrix / [The Viterbi algorithm](#)
 - qStar matrix / [The Viterbi algorithm](#)
 - delta matrix / [The Viterbi algorithm](#)

W

- weight decay / [Ln roughness penalty](#)
- weighted moving average
 - about / [Weighted moving average](#)
- While loop / [Discrete Fourier transform \(DFT\)](#)
- white noise / [The transition equation](#)
- WordNet / [Basics information retrieval](#)
- workflow
 - instantiating / [Instantiating the workflow](#)
 - modularizing / [Modularizing](#)
- workflow computational model
 - about / [Workflow computational model](#)

Y

- 1-year Treasury bill (1yTB)
 - about / [Introducing the multinomial Naïve Bayes](#)
- Yahoo finances
 - reference link / [Financial data sources](#)

Z

- Z-score / [Z-score and Gauss](#)