

2ND EDITION

Hands-On Machine Learning with C++

Build, train, and deploy end-to-end machine learning
and deep learning pipelines



KIRILL KOLODIAZHNYI

Hands-On Machine Learning with C++

Build, train, and deploy end-to-end machine learning and deep learning pipelines

Kirill Kolodiaznyi



Hands-On Machine Learning with C++

Copyright © 2025 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 or its dealers and distributors, will be held liable for any damages caused or alleged to have been 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.

Group Product Manager: Ali Abidi

Publishing Product Manager: Tejaswini R

Book Project Manager: Hemangi Lotlikar

Senior Editor: Tazeen Shaikh

Technical Editor: Sweety Pagaria

Copy Editor: Safis Editing

Proofreader: Tazeen Shaikh

Indexer: Rekha Nair

Production Designer: Jyoti Kadam

Senior DevRel Marketing Executive: Vinishka Kalra

First published: May 2020

Second edition: January 2025

Production reference: 1061224

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80512-057-5

www.packtpub.com

*To my wife, Anna, and daughter, Veronica, for their help and patience through
the book creation process.*

– Kirill Kolodiazhnyi

Contributors

About the author

Kirill Kolodiaznyi is a seasoned software engineer with expertise in custom software development. He has several years of experience building machine learning models and data products using C++. He holds a bachelor's degree in computer science from the Kharkiv National University of Radio Electronics.

About the reviewers

Harshit Jain is a passionate AI engineer with extensive experience in developing cutting-edge AI solutions. He has worked on global projects, collaborating with international teams to drive innovative AI advancements and social causes, such as wildlife conservation and environmental sustainability. With a strong foundation in problem-solving and deep AI knowledge, Harshit has contributed to transformative projects in data analysis, image recognition, and advanced language models. An avid traveler and car enthusiast, he enjoys exploring new places, cooking, and discovering cultures through food and literature.

Lalithkumar Prakashchand is a seasoned software engineer with over a decade of experience in developing high-performance, enterprise-level applications. Specializing in microservices, distributed computing, machine learning, and AI, he has made significant contributions to leading tech companies such as Meta and Careem. Lalithkumar has been instrumental in building robust microservices and optimizing core platform capabilities, leading to substantial improvements in system performance and user engagement. He is honored to contribute as a reviewer for *Hands-On Machine Learning with C++*, bringing his extensive expertise to ensure the content is of the highest quality for its readers.

Hemanath Kumar J is a seasoned data enthusiast with extensive experience in developing and implementing machine learning models, GenAI models, data visualization, and analytics solutions. With a diverse background in transportation, education, finance, and healthcare, he has consistently delivered solutions with data-driven strategies that enhanced decision-making processes with high accuracy and operational efficiency. As a technical reviewer for Packt Publications, he brings his comprehensive expertise to *Hands-On Machine Learning with C++*, ensuring accuracy and clarity. He would like to acknowledge his family, mentors, and friends for their unwavering support and encouragement throughout this project.

Table of Contents

Preface	xv
---------	----

Part 1: Overview of Machine Learning

1

Introduction to Machine Learning with C++	3
Understanding the fundamentals of ML	4
Venturing into the techniques of ML	4
Dealing with ML models	5
Model parameter estimation	7
An overview of linear algebra	7
Learning the concepts of linear algebra	8
Basic linear algebra operations	9
Tensor representation in computing	11
Linear algebra API samples	12
An overview of linear regression	26
Solving linear regression tasks with different libraries	27
Solving linear regression tasks with Eigen	30
Solving linear regression tasks with Blaze	31
Solving linear regression tasks with ArrayFire	32
Linear regression with Dlib	34
Summary	34
Further reading	35

2

Data Processing	37
Technical requirements	38
Parsing data formats to C++ data structures	38
Reading CSV files with the Fast-CPP-CSV-Parser library	40
Preprocessing CSV files	42
Reading CSV files with the mlpack library	43
Reading CSV files with the Dlib library	44
Reading JSON files with the nlohmann-json library	45
Writing and reading HDF5 files with the HighFive library	53
Initializing matrix and tensor objects from C++ data structures	56
Working with the Eigen library	56
Working with the Blaze library	57

Working with the Dlib library	57	Deinterleaving in OpenCV	64
Working with the ArrayFire library	57	Deinterleaving in Dlib	65
Working with the mlpack library	58	Normalizing data	66
Manipulating images with the OpenCV and Dlib libraries	58	Normalizing with Eigen	67
Using OpenCV	59	Normalizing with mlpack	68
Using Dlib	61	Normalizing with Dlib	69
Transforming images into matrix or tensor objects of various libraries	64	Normalizing with Flashlight	69
		Summary	70
		Further reading	71

3

Measuring Performance and Selecting Models	73
---	-----------

Technical requirements	73	Model selection with the grid search technique	89
Performance metrics for ML models	74	Cross-validation	89
Regression metrics	74	Grid search	90
Classification metrics	76	mlpack example	91
Understanding bias and variance characteristics	81	Optuna with Flashlight example	93
Bias	82	Dlib example	98
Variance	84	Summary	99
Normal training	85	Further reading	100
Regularization	87		

Part 2: Machine Learning Algorithms

4

Clustering	103
-------------------	------------

Technical requirements	103	Chebyshev distance	105
Measuring distance in clustering	104	Types of clustering algorithms	106
Euclidean distance	104	Partition-based clustering algorithms	107
Squared Euclidean distance	104	Distance-based clustering algorithms	107
Manhattan distance	105	Graph theory-based clustering algorithms	108

Spectral clustering algorithms	109	Examples of using the Dlib library for dealing with the clustering task samples	119
Hierarchical clustering algorithms	110	K-means clustering with Dlib	119
Density-based clustering algorithms	111	Spectral clustering with Dlib	121
Model-based clustering algorithms	112	Hierarchical clustering with Dlib	123
Examples of using the mlpack library for dealing with the clustering task samples	114	Newman modularity-based graph clustering algorithm with Dlib	125
GMM and EM with mlpack	114	Chinese Whispers – graph clustering algorithm with Dlib	127
K-means clustering with mlpack	115		
DBSCAN with mlpack	117		
MeanShift clustering with mlpack	118	Plotting data with C++	129
		Summary	131
		Further reading	132

5

Anomaly Detection **133**

Technical requirements	134	Density estimation approach	141
Exploring the applications of anomaly detection	134	Using multivariate Gaussian distribution for anomaly detection	141
Learning approaches for anomaly detection	136	Examples of using different C++ libraries for anomaly detection	145
Detecting anomalies with statistical tests	136	C++ implementation of the isolation forest algorithm for anomaly detection	146
Detecting anomalies with the Local Outlier Factor method	137	Using the Dlib library for anomaly detection	153
Detecting anomalies with isolation forest	139	Summary	165
Detecting anomalies with One-Class Support Vector Machine	140	Further reading	165

6

Dimensionality Reduction **167**

Technical requirements	167	Dimensionality reduction methods	170
An overview of dimension-reduction methods	168	Exploring linear methods for dimension-reduction	170
Feature selection methods	169	Principal component analysis	170

Singular value decomposition	172	Sammon mapping	179
Independent component analysis	172	Distributed stochastic neighbor embedding	180
Linear discriminant analysis	174	Autoencoders	182
Factor analysis	175		
Multidimensional scaling	176	Understanding dimension-reduction algorithms with various C++ libraries	182
Exploring non-linear methods for dimension-reduction	177	Using the Dlib library	183
Kernel PCA	178	Using the Tapkee library	191
Isomap	178	Summary	200
		Further reading	201

7

Classification		203	
Technical requirements	204	Multi-class classification	215
An overview of classification methods	204	Examples of using C++ libraries for dealing with the classification task	217
Exploring various classification methods	205	Using the mlpack library	218
Logistic regression	205	Using the Dlib library	224
KRR	208	Using the Flashlight library	229
SVM	209	Summary	234
kNN method	213	Further reading	235

8

Recommender Systems		237	
Technical requirements	237	Relevance of recommendations	247
An overview of recommender system algorithms	238	Assessing system quality	247
Non-personalized recommendations	240	Understanding the collaborative filtering method	248
Content-based recommendations	241	Examples of item-based collaborative filtering with C++	253
User-based collaborative filtering	242	Using the Eigen library	253
Item-based collaborative filtering	243	Using the mlpack library	260
Factorization algorithms	244	Summary	262
Similarity or preferences correlation	245	Further reading	263
Data scaling and standardization	246		
Cold start problem	246		

9**Ensemble Learning 265**

Technical requirements	265	Using the random forest method for creating ensembles	276
An overview of ensemble learning	266	Examples of using C++ libraries for creating ensembles	280
Using a bagging approach for creating ensembles	268	Ensembles with Dlib	281
Using a gradient boosting method for creating ensembles	271	Ensembles with mlpack	284
Using a stacking approach for creating ensembles	275	Summary	293
		Further reading	294

Part 3: Advanced Examples**10****Neural Networks for Image Classification 297**

Technical requirements	298	What is deep learning?	328
An overview of neural networks	298	Examples of using C++ libraries to create neural networks	329
Neurons	299	Dlib	329
The perceptron and neural networks	300	mlpack	332
Training with the backpropagation method	303	Flashlight	335
Loss functions	311		
Activation functions	313	Understanding image classification using the LeNet architecture	337
Regularization in neural networks	320	Reading the training dataset	339
Neural network initialization	322	Neural network definition	344
Delving into convolutional networks	323	Network training	347
Convolution operator	323	Summary	351
Pooling operation	325	Further reading	352
Receptive field	326		
Convolution network architecture	327		

11**Sentiment Analysis with BERT and Transfer Learning 353**

Technical requirements	354	Sentiment analysis example with BERT	360
An overview of the Transformer architecture	354	Exporting the model and vocabulary	360
Encoder	356	Implementing the tokenizer	362
Decoder	357	Implementing the dataset loader	366
Tokenization	358	Implementing the model	371
Word embeddings	359	Training the model	373
Using the encoder and decoder parts separately	359	Summary	375
		Further reading	376

Part 4: Production and Deployment Challenges**12****Exporting and Importing Models 379**

Technical requirements	379	Delving into the ONNX format	395
ML model serialization APIs in C++ libraries	380	Using the ResNet architecture for image classification	396
Model serialization with Dlib	380	Loading images into onnxruntime tensors	402
Model serialization with Flashlight	384	Reading the class definition file	404
Model serialization with mlpack	386	Summary	405
Model serialization with PyTorch	388	Further reading	406

13**Tracking and Visualizing ML Experiments 407**

Technical requirements	408	MLflow	410
Understanding visualization and tracking systems for experiments	408	Experiment tracking with MLflow's REST API	411
TensorBoard	409	Implementing MLflow's REST C++ client	411

Logging metric values and running parameters	416	Experiments tracking process	421
Integrating experiment tracking into linear regression training	418	Summary	427
		Further reading	428

14

Deploying Models on a Mobile Platform	429		
Technical requirements	430	The Android Studio project	434
Developing object detection on Android	430	The Kotlin part of the project	436
The mobile version of the PyTorch framework	431	The native C++ part of the project	440
Using TorchScript for a model snapshot	433	Summary	465
		Further reading	466
Index	467		
Other Books You May Enjoy	484		

Preface

C++ can make your **machine learning (ML)** models run faster and more efficiently. This book teaches you the basics of ML and shows you how to use C++ libraries. It explains how to create supervised and unsupervised ML models.

You'll get hands-on with tuning and optimizing a model for different use cases, assisting you with model selection and the measurement of performance. The book covers techniques such as product recommendations, ensemble learning, anomaly detection, sentiment analysis, and object recognition using modern C++ libraries. Further, you'll learn how to handle production and deployment challenges on mobile platforms, and how the ONNX model format can help you in such tasks.

This new edition is updated with key topics such as sentiment analysis implementation using transfer learning and transformer-based models and tracking and visualizing ML experiments with MLflow. Also, there is an additional section about using Optuna for hyperparameter selection. The section about model deployment into mobile platforms is extended with a detailed explanation of real-time object detection for Android with C++.

By the end of this C++ book, you will have real-world ML and C++ knowledge, as well as the skills to use C++ to build powerful ML systems.

Who this book is for

If you want to get started with ML algorithms and techniques using the popular C++ language, then this book is for you. Aside from being a useful first course in ML with C++, this book will also appeal to data analysts, data scientists, and ML developers looking to implement different ML models in production using C++, which can be useful for some specific platforms, for example, embedded devices. Working knowledge of the C++ programming language, linear algebra, and basic calculus understanding are needed to get started with this book.

What this book covers

Chapter 1, Introduction to Machine Learning with C++, guides you through the necessary fundamentals of ML, including linear algebra concepts, ML algorithm types, and their building blocks.

Chapter 2, Data Processing, shows you how to load data from different file formats for ML model training and how to initialize dataset objects in various C++ libraries.

Chapter 3, Measuring Performance and Selecting Models, shows you how to measure the performance of various types of ML models, how to select the best set of hyperparameters to achieve better model performance, and how to use the grid search method in various C++ and external libraries for model selection.

Chapter 4, Clustering, discusses algorithms for grouping objects by their essential characteristics, shows why we usually use unsupervised algorithms for solving such types of tasks, and lastly, outlines the various types of clustering algorithms, along with their implementations and usage in different C++ libraries.

Chapter 5, Anomaly Detection, discusses the basics of anomaly and novelty detection tasks and guides you through the different types of anomaly detection algorithms, their implementation, and their usage in various C++ libraries.

Chapter 6, Dimensionality Reduction, discusses various algorithms for dimensionality reduction that preserve the essential characteristics of data, along with their implementation and usage in various C++ libraries.

Chapter 7, Classification, shows you what a classification task is and how it differs from a clustering task. You will be guided through various classification algorithms, their implementation, and their usage in various C++ libraries.

Chapter 8, Recommender Systems, gives you familiarity with recommender system concepts. You will be shown the different approaches to dealing with recommendation tasks, and you will see how to solve such types of tasks using the C++ language.

Chapter 9, Ensemble Learning, discusses various methods of combining several ML models to get better accuracy and to deal with learning problems. You will encounter ensemble implementations with the usage of different C++ libraries.

Chapter 10, Neural Networks for Image Classification, gives you familiarity with the fundamentals of artificial neural networks. You will encounter the essential building blocks, the required math concepts, and learning algorithms. You will be guided through different C++ libraries that provide functionality for neural network implementations. Also, this chapter will show you the implementation of a deep convolutional network for image classification with the PyTorch library.

Chapter 11, Sentiment Analysis with BERT and Transfer Learning, introduces you to **large language models (LLMs)**, and briefly describes how they work. It will also show how to use the transfer learning technique to use pre-trained LLMs to implement sentiment analysis with the PyTorch library.

Chapter 12, Exporting and Importing Models, shows you how to save and load model parameters and architectures using various C++ libraries. Also, you will see how to use the ONNX format to load and use a pre-trained model with the C++ API of the Caffe2 library.

Chapter 13, Tracking and Visualizing ML Experiments, shows you how to use the MLflow toolkit to track and visualize your ML experiments. Visualization is essential for understanding patterns, relationships, and trends in experiments. Experiment tracking allows you to compare results, identify best practices, and avoid repeating mistakes.

Chapter 14, Deploying Models on a Mobile Platform, guides you through the development of applications for object detection on devices' camera images using neural networks for the Android platform.

To get the most out of this book

To be able to compile and run the examples included in this book, you will need to configure a particular development environment. All code examples have been tested with the Ubuntu Linux version 22.04 distributions. The following list outlines the packages you'll need to install on the Ubuntu platform:

- `unzip`
- `build-essential`
- `gdb`
- `git`
- `libfmt-dev`
- `wget`
- `cmake`
- `python3`
- `python3-pip`
- `python-is-python3`
- `libblas-dev`
- `libopenblas-dev`
- `libfftw3-dev`
- `libatlas-base-dev`
- `liblapacke-dev`
- `liblapack-dev`
- `libboost-all-dev`
- `libopencv-core4.5d`
- `libopencv-imgproc4.5d`
- `libopencv-dev`

- libopencv-highgui4.5d
- libopencv-highgui-dev
- libhdf5-dev
- libjson-c-dev
- libx11-dev
- openjdk-8-jdk
- openjdk-17-jdk
- ninja-build
- gnuplot
- vim
- python3-venv
- libcpuinfo-dev
- libspdlog-dev

You will need a `cmake` package with a version not less than 2.27. To get it on Ubuntu 22.04, you have to download it manually and install it. For example, it can be done as follows:

```
wget https://github.com/Kitware/CMake/releases/download/v3.27.5/cmake-  
3.27.5-Linux-x86_64.sh \  
    -q -O /tmp/cmake-install.sh \  
    && chmod u+x /tmp/cmake-install.sh \  
    && mkdir /usr/bin/cmake \  
    && /tmp/cmake-install.sh --skip-license --prefix=/usr/bin/cmake \  
    && rm /tmp/cmake-install.sh  
  
export PATH="/usr/bin/cmake/bin:${PATH}"
```

Also, you need to install the additional packages for Python, which can be done with the following commands:

```
pip install pyyaml  
pip install typing  
pip install typing_extensions  
pip install optuna  
pip install torch==2.3.1 \  
    --index-url https://download.pytorch.org/whl/cpu  
pip install transformers  
pip install mlflow==2.15.0
```

Besides the development environment, you'll have to check out requisite third-party libraries' source code samples and build them. Most of these libraries are actively developed so you provide particular versions (Git tags) that you need to check out for our coding samples compatibility. The following table shows you the libraries you have to check out, their repository URLs, and the tag or the hash number of the commit to check out:

Library repository	Branch name/Tag	Commit
https://bitbucket.org/blaze-lib/blaze.git	v3.8.2	
https://github.com/arrayfire/arrayfire	v3.8.3	
https://github.com/flashlight/flashlight.git	v0.4.0	
https://github.com/davisking/dlib	v19.24.6	
https://gitlab.com/conradsnicta/armadillo-code	14.0.x	
https://github.com/xtensor-stack/xtl	0.7.7	
https://github.com/xtensor-stack/xtensor	0.25.0	
https://github.com/xtensor-stack/xtensor-blas	0.21.0	
https://github.com/nlohmann/json.git	v3.11.3	
https://github.com/mlpack/mlpack	4.5.0	
https://gitlab.com/libeigen/eigen.git	3.4.0	
https://github.com/BlueBrain/HighFive	v2.10.0	
https://github.com/yhirose/cpp-httplib	v0.18.1	
https://github.com/Kolkir/plotcpp		c86bd4f5d9029986f0d5f368450 d79f0dd32c7e4
https://github.com/ben-strasser/fast-cpp-csv-parser		4ade42d5f8c454c6c57b3dce9c51c 6dd02182a66

Library repository	Branch name/Tag	Commit
https://github.com/lisitsyn/tapkee		Ba5f052d2548ec03dcc6a4ac0e d8deeb79f1d43a
https://github.com/Microsoft/onnxruntime.git	v1.19.2	
https://github.com/pytorch/pytorch	v2.3.1	

Notice that it makes sense to compile and install PyTorch last, due to possible conflicts with the protobuf library version used by onnxruntime.

Also, for the last chapter, you may want to install the Android Studio IDE. You can download it from the official site at <https://developer.android.com/studio>. Besides the IDE, you'll also need to install and configure the Android SDK, NDK, and Android-based version of the OpenCV library. The following versions of tools are required:

Name	Versions
OpenCV	4.10.0
Android command-line tools for Linux	9477386
Android NDK	26.1.10909125
Android platform	35

You can configure these tools with the Android IDE or command-line tools as follows:

```
wget https://github.com/opencv/opencv/releases/download/4.10.0/opencv-4.10.0-android-sdk.zip
unzip opencv-4.10.0-android-sdk.zip

wget https://dl.google.com/android/repository/commandlinetools-linux-9477386_latest.zip
unzip commandlinetools-linux-9477386_latest.zip

./cmdline-tools/bin/sdkmanager --sdk_root=$ANDROID_SDK_ROOT "cmdline-tools;latest"
./cmdline-tools/latest/bin/sdkmanager --licenses
./cmdline-tools/latest/bin/sdkmanager "platform-tools" "tools"
./cmdline-tools/latest/bin/sdkmanager "platforms;android-35"
./cmdline-tools/latest/bin/sdkmanager "build-tools;35.0.0"
./cmdline-tools/latest/bin/sdkmanager "system-images;android-35;google_apis;arm64-v8a"
./cmdline-tools/latest/bin/sdkmanager --install "ndk;26.1.10909125"
```

Another way to configure the development environment is through the use of Docker. Docker allows you to configure a lightweight virtual machine with particular components. You can install Docker from the official Ubuntu package repository. Then, use the scripts provided with this book to automatically configure the environment. You will find the `build-env` folder in the `examples` repository. The following steps show how to use Docker configuration scripts:

1. Configure your GitHub account first. Then, you will be able to configure GitHub authenticating with SSH, as described in the article *Connecting to GitHub with SSH* (<https://docs.github.com/en/authentication/connecting-to-github-with-ssh>); this is the preferred way. Or you can use HTTPS and provide your username and password each time a new repository is cloned. If you use 2FA to secure your GitHub account, then you'll need to use a personal access token instead of a password, as explained in the article *Creating a personal access token* (<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>).
2. Run the following commands to create the image, run it, and configure the environment:

```
cd docker  
docker build -t buildenv:1.0 .
```

3. Use the following command to start a new Docker container and share the book examples sources with it:

```
docker run -it -v [host_examples_path]:[container_examples_path]  
[tag name] bash
```

Here, `host_examples` is the path to the checked-out example sources from <https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition.git> and `container_examples_path` is the destination mounting path in the container, for example, `/samples`.

After running the preceding command, you will be in the command-line environment with the necessary configured packages, compiled third-party libraries, and access to the programming examples package. You can use this environment to compile and run the code examples in this book. Each programming example is configured to use the CMake build system so you will be able to build them all in the same way. The following script shows a possible scenario of building a code example:

```
cd Chapter01  
mkdir build  
cd build  
cmake ..  
cmake --build . --target all
```

This is the manual approach. We also provide ready-to-use scripts to build each example. These scripts are placed in the `build_scripts` folder of the repository. For example, the build script for the first chapter is `build_ch1.sh`, which can be run directly from this folder.

If you are going to configure your build environment manually, take care of the `LIBS_DIR` variable that should point to the folder where all third-party libraries are installed; using the provided build scripts for the Docker environment, it will point to `$HOME/development/libs`.

Also, you can configure your local machine environment to share X Server with a Docker container to be able to run graphical UI applications from this container. It will allow you to use, for example, the Android Studio IDE or a C++ IDE (such as Qt Creator) from the Docker container, without local installation. The following script shows how to do this:

```
xhost +local:root
docker run --net=host -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-
unix -it -v [host_examples_path]:[container_examples_path] [tag name]
bash
```

To be more comfortable with understanding and building the code examples, we recommend you carefully read the documentation for each third-party library, and take some time to learn the basics of the Docker system and of development for the Android platform. Also, we assume that you have sufficient working knowledge of the C++ language and compilers and that you are familiar with the CMake build system.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the following section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter/X handles. Here is an example: “The `Dlib` library doesn't have many classification algorithms.”

A block of code is set as follows:

```
std::vector<fl::Tensor> fields{train_x, train_y};  
auto dataset = std::make_shared<fl::TensorDataset>(fields);  
int batch_size = 8;  
auto batch_dataset = std::make_shared<fl::BatchDataset>(dataset,  
batch_size);
```

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]  
exten => s,1,Dial(Zap/1|30)  
exten => s,2,Voicemail(u100)  
exten => s,102,Voicemail(b100)  
exten => i,1,Voicemail(s0)
```

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

```
$ mkdir css  
$ cd css
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “In the **one-against-all** strategy for N classes, N classifiers are trained, each of which separates its class from all other classes.”

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Hands-On Machine Learning with C++*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781805120575>

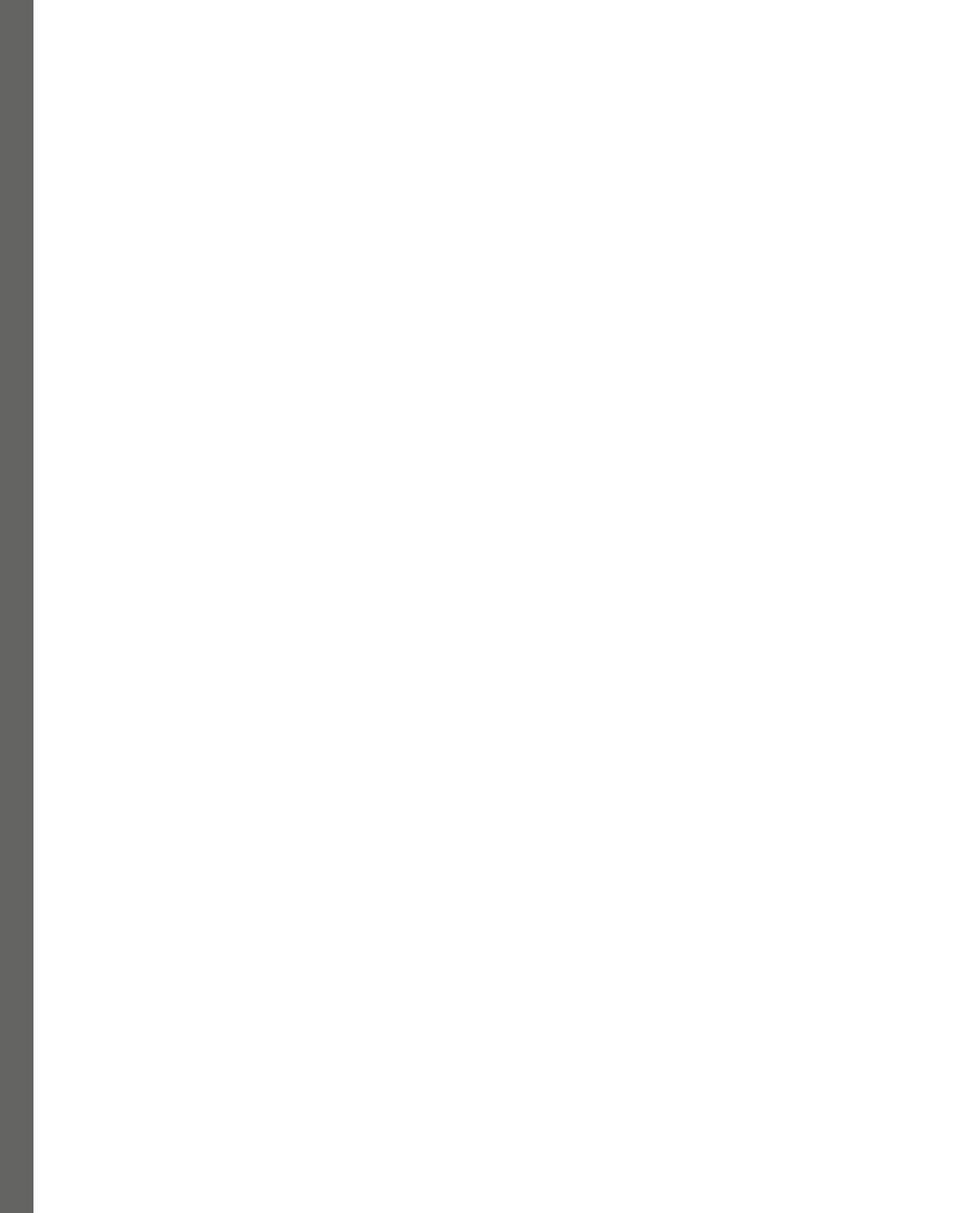
2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Overview of Machine Learning

In this part, we will delve into the basics of machine learning with the help of examples in C++ and various machine learning frameworks. We'll demonstrate how to load data from various file formats and describe model performance measuring techniques and the best model selection approaches.

This part comprises the following chapters:

- *Chapter 1, Introduction to Machine Learning with C++*
- *Chapter 2, Data Processing*
- *Chapter 3, Measuring Performance and Selecting Models*



1

Introduction to Machine Learning with C++

There are different approaches to making computers solve tasks. One of them is to define an explicit algorithm, and another one is to use implicit strategies based on mathematical and statistical methods. **Machine learning (ML)** is one of the implicit methods that uses mathematical and statistical approaches to solve tasks. It is an actively growing discipline, and a lot of scientists and researchers find it to be one of the best ways to move forward toward systems acting as human-level **artificial intelligence (AI)**.

In general, ML approaches have the idea of searching patterns in a given dataset as their basis. Consider a recommendation system for a news feed, which provides the user with a personalized feed based on their previous activity or preferences. The software gathers information about the type of news article the user reads and calculates some statistics. For example, it could be the frequency of some topics appearing in a set of news articles. Then, it performs some predictive analytics, identifies general patterns, and uses them to populate the user's news feed. Such systems periodically track a user's activity, update the dataset, and calculate new trends for recommendations.

ML is a rapidly growing field with diverse applications across various industries. In healthcare, it analyzes medical data to detect patterns and predict diseases and treatment outcomes. In finance, it aids in credit scoring, fraud detection, risk assessment, portfolio optimization, and algorithmic trading, enhancing decision-making and operations. E-commerce benefits from recommendation systems that suggest products based on customer behavior, boosting sales and satisfaction. Autonomous vehicles use ML for environmental perception, decision-making, and safe navigation.

Customer service is improved with chatbots and virtual assistants that handle queries and tasks. Cybersecurity leverages ML to detect and prevent cyberattacks by analyzing network traffic and identifying threats. Language translation tools use ML for accurate and efficient text translation. Image recognition, powered by computer vision algorithms, identifies objects, faces, and scenes in images and videos, supporting applications such as facial recognition and content moderation. Speech recognition in voice assistants such as Siri, Google Assistant, and Alexa relies on ML for understanding and responding to user commands. These examples illustrate the vast potential of ML in shaping our lives.

This chapter describes what ML is and which tasks can be solved with ML, and discusses different approaches used in ML. It aims to show the minimally required math to start implementing ML algorithms. It also covers how to perform basic **linear algebra** operations in libraries such as `Eigen`, `xtensor`, `ArrayFire`, `Blaze`, and `Dlib`, and also explains the linear regression task as an example.

The following topics will be covered in this chapter:

- Understanding the fundamentals of ML
- An overview of linear algebra
- An overview of a linear regression example

Understanding the fundamentals of ML

There are different approaches to creating and training ML models. In this section, we show what these approaches are and how they differ. Apart from the approach we use to create an ML model, there are also parameters that manage how this model behaves in the training and evaluation processes. Model parameters can be divided into two distinct groups, which should be configured in different ways. The first group of parameters is the model weights in ML algorithms that are used to adjust the model's predictions. They are assigned numerical values during the training process, and these values determine how the model makes decisions or predictions based on new data. The second group is the model hyperparameters that control the behavior of an ML model during training. They are not learned from the data like other parameters in the model but, rather, are set by the user or algorithm before training begins. The last crucial part of the ML process is the technique that we use to train a model. Usually, the training technique uses some numerical optimization algorithm that finds the minimal value of a target function. In ML, the target function is usually called a loss function and is used for penalizing the training algorithm when it makes errors. We discuss these concepts more precisely in the following sections.

Venturing into the techniques of ML

We can divide ML approaches into two techniques, as follows:

- **Supervised learning** is an approach based on the use of labeled data. Labeled data is a set of known data samples with corresponding known target outputs. Such data is used to build a model that can predict future outputs.
- **Unsupervised learning** is an approach that does not require labeled data and can search hidden patterns and structures in an arbitrary kind of data.

Let's have a look at each of the techniques in detail.

Supervised learning

Supervised ML algorithms usually take a limited set of labeled data and build models that can make reasonable predictions for new data. We can split supervised learning algorithms into two main parts, classification and regression techniques, described as follows:

- Classification models predict some finite and distinct types of categories—this could be a label that identifies whether an email is spam or not, or whether an image contains a human face or not. Classification models are applied in speech and text recognition, object identification on images, credit scoring, and others. Typical algorithms for creating classification models are **support vector machine (SVM)**, decision tree approaches, **k-nearest neighbors (KNN)**, logistic regression, Naive Bayes, and neural networks. The following chapters describe the details of some of these algorithms.
- Regression models predict continuous responses such as changes in temperature or values of currency exchange rates. Regression models are applied in algorithmic trading, forecasting of electricity load, revenue prediction, and others. Creating a regression model usually makes sense if the output of the given labeled data is real numbers. Typical algorithms for creating regression models are linear and multivariate regressions, polynomial regression models, and stepwise regressions. We can use decision tree techniques and neural networks to create regression models too.

The following chapters describe the details of some of these algorithms.

Unsupervised learning

Unsupervised learning algorithms do not use labeled datasets. They create models that use intrinsic relations in data to find hidden patterns that they can use for making predictions. The most well-known unsupervised learning technique is **clustering**. Clustering involves dividing a given set of data into a limited number of groups according to some intrinsic properties of data items. Clustering is applied in market research, different types of exploratory analysis, **deoxyribonucleic acid (DNA)** analysis, image segmentation, and object detection. Typical algorithms for creating models for performing clustering are k-means, k-medoids, Gaussian mixture models, hierarchical clustering, and hidden Markov models. Some of these algorithms are explained in the following chapters of this book.

Dealing with ML models

We can interpret ML models as functions that take different types of parameters. Such functions provide outputs for given inputs based on the values of these parameters. Developers can configure the behavior of ML models for solving problems by adjusting model parameters. Training an ML model can usually be treated as a process of searching for the best combination of its parameters. We can split the ML model's parameters into two types. The first type consists of parameters internal to the model, and we can estimate their values from the training (input) data. The second type consists of parameters external to the model, and we cannot estimate their values from training data. Parameters that are external to the model are usually called **hyperparameters**.

Internal parameters have the following characteristics:

- They are necessary for making predictions
- They define the quality of the model on the given problem
- We can learn them from training data
- Usually, they are a part of the model

If the model contains a fixed number of internal parameters, it is called **parametric**. Otherwise, we can classify it as **non-parametric**.

Examples of internal parameters are as follows:

- Weights of **artificial neural networks (ANNs)**
- Support vector values for SVM models
- Polynomial coefficients for linear regression or logistic regression

ANNs are computer systems inspired by the structure and function of biological neural networks in the human brain. They are composed of interconnected nodes, or neurons, that process and transmit information. ANNs are designed to learn patterns and relationships from data, allowing them to make predictions or decisions based on new inputs. The learning process involves adjusting the weights and biases of the connections between neurons to improve the accuracy of the model.

On the other hand, hyperparameters have the following characteristics:

- They are used to configure algorithms that estimate model parameters
- The practitioner usually specifies them
- Their estimation is often based on using heuristics
- They are specific to a concrete modeling problem

It is hard to know the best values for a model's hyperparameters for a specific problem. Also, practitioners usually need to perform additional research on how to tune required hyperparameters so that a model or a training algorithm behaves in the best way. Practitioners use rules of thumb, copying values from similar projects, as well as special techniques such as grid search for hyperparameter estimation.

Examples of hyperparameters are as follows:

- C and sigma parameters used in the SVM algorithm for a classification quality configuration
- The learning rate parameter that is used in the neural network training process to configure algorithm convergence
- The k value that is used in the KNN algorithm to configure the number of neighbors

Model parameter estimation

Model parameter estimation usually uses some optimization algorithm. The speed and quality of the resulting model can significantly depend on the optimization algorithm chosen. Research on optimization algorithms is a popular topic in industry, as well as in academia. ML often uses optimization techniques and algorithms based on the optimization of a loss function. A function that evaluates how well a model predicts the data is called a **loss function**. If predictions are very different from the target outputs, the loss function will return a value that can be interpreted as a bad one, usually a large number. In such a way, the loss function penalizes an optimization algorithm when it moves in the wrong direction. So, the general idea is to minimize the value of the loss function to reduce penalties. There is no single universal loss function for optimization algorithms. Different factors determine how to choose a loss function. Examples of such factors are as follows:

- Specifics of the given problem—for example, whether it is a regression or a classification model
- Ease of calculating derivatives
- Percentage of outliers in the dataset

In ML, the term **optimizer** is used to define an algorithm that connects a loss function and a technique for updating model parameters in response to the values of the loss function. So, optimizers tune ML models to predict target values for new data in the most accurate way by fitting model parameters. Optimizers or optimization algorithms play a crucial role in training ML models. They help find the best parameters for a model, which can improve its performance and accuracy. They have wide applications in various fields such as image recognition, natural language processing, and fraud detection. For example, in image classification tasks, optimization algorithms can be used to train deep neural networks to accurately identify objects in images. There are many optimizers: gradient descent, Adagrad, RMSProp, Adam, and others. Moreover, developing new optimizers is an active area of research. For example, there is the *ML and Optimization* research group at Microsoft (located in Redmond) whose research areas include combinatorial optimization, convex and non-convex optimization, and their application in ML and AI. Other companies in the industry also have similar research groups; there are many publications from Facebook Research, Amazon Research, and OpenAI groups.

Now, we will learn about what ML is and what its main conceptual parts are. So let's learn the most important part of its mathematical basement: linear algebra.

An overview of linear algebra

The concepts of linear algebra are essential for understanding the theory behind ML because they help us understand how ML algorithms work under the hood. Also, most ML algorithm definitions use linear algebra terms.

Linear algebra is not only a handy mathematical instrument but also the concepts of linear algebra can be very efficiently implemented with modern computer architectures. The rise of ML, and especially deep learning, began after significant performance improvement of the modern **graphics processing unit (GPU)**. GPUs were initially designed to work with linear algebra concepts and massively parallel computations used in computer games. After that, special libraries were created to work with general linear algebra concepts. Examples of libraries that implement basic linear algebra routines are Cuda and OpenCL, and one example of a specialized linear algebra library is cuBLAS. Moreover, it became more common to use **general-purpose graphics processing units (GPGPUs)** because these turn the computational power of a modern GPU into a powerful general-purpose computing resource.

Also, **central processing units (CPUs)** have instruction sets specially designed for simultaneous numerical computations. Such computations are called **vectorized**, and common vectorized instruction sets are AVx, SSE, and MMx. There is also the term **single instruction multiple data (SIMD)** for these instruction sets. Many numeric linear algebra libraries, such as Eigen, xtensor, VienaCL, and others, use them to improve computational performance.

Learning the concepts of linear algebra

Linear algebra is a big area. It is the section of algebra that studies objects of a linear nature: vector (or linear) spaces, linear representations, and systems of linear equations. The main tools used in linear algebra are determinants, matrices, conjugation, and tensor calculus.

To understand ML algorithms, we only need a small set of linear algebra concepts. However, to do research on new ML algorithms, a practitioner should have a deep understanding of linear algebra and calculus.

The following list contains the most valuable linear algebra concepts for understanding ML algorithms:

- **Scalar:** This is a single number.
- **Vector:** This is an array of ordered numbers. Each element has a distinct index. Notation for vectors is a bold lowercase typeface for names and an italic typeface with a subscript for elements, as shown in the following example:

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix}$$

- **Matrix:** This is a two-dimensional array of numbers. Each element has a distinct pair of indices. Notation for matrices is a bold uppercase typeface for names and an italic but not bold typeface with a comma-separated list of indices in subscripts for elements, as shown in the following example:

$$\mathbf{A} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix}$$

- **Tensor:** This is an array of numbers arranged in a multidimensional regular grid, and represents generalizations of matrices. It is like a multidimensional matrix. For example, tensor A with dimensions $2 \times 2 \times 2$ can look like this:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Linear algebra libraries and ML frameworks usually use the concept of a tensor instead of a matrix because they implement general algorithms, and a matrix is just a special case of a tensor with two dimensions. Also, we can consider a vector as a matrix of size $n \times 1$.

Basic linear algebra operations

The most common operations used for programming linear algebra algorithms are the following ones:

- **Element-wise operations:** These are performed in an element-wise manner on vectors, matrices, or tensors of the same size. The resulting elements will be the result of operations on corresponding input elements, as shown here:

$$\mathbf{A} + \mathbf{B} = \mathbf{C}, C_{i,j} = A_{i,j} + B_{i,j}$$

$$\mathbf{A} - \mathbf{B} = \mathbf{C}, C_{i,j} = A_{i,j} - B_{i,j}$$

$$\mathbf{A} * \mathbf{B} = \mathbf{C}, C_{i,j} = A_{i,j} * B_{i,j}$$

$$\mathbf{A}/\mathbf{B} = \mathbf{C}, C_{i,j} = A_{i,j}/B_{i,j}$$

The following example shows the element-wise summation:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix}$$

- **Dot product:** There are two types of multiplication for tensors and matrices in linear algebra—one is just element-wise, and the second is the dot product. The dot product deals with two equal-length series of numbers and returns a single number. This operation applied on matrices or tensors requires that the matrix or tensor A has the same number of columns as the number of rows in the matrix or tensor B . The following example shows the dot-product operation in the case when A is an $n \times m$ matrix and B is an $m \times p$ matrix:

$$\mathbf{A} \cdot \mathbf{B} = \mathbf{C}, C_{i,j} = \sum_{k=1}^m A_{i,k} B_{k,j}, i = 1, \dots, n, j = 1, \dots, p$$

- **Transposing:** The transposing of a matrix is an operation that flips the matrix over its diagonal, which leads to the flipping of the column and row indices of the matrix, resulting in the creation of a new matrix. In general, it is swapping matrix rows with columns. The following example shows how transposing works:

$$(\mathbf{A})^T_{i,j} = \mathbf{A}_{j,i} \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

- **Norm:** This operation calculates the size of the vector; the result of this is a non-negative real number. The norm formula is as follows:

$$\|x\|_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}}$$

The generic name of this type of norm is an L^P norm for $p \in R, p \geq 1$. Usually, we use more concrete norms such as an L^2 norm with $p = 2$, which is known as the Euclidean norm, and we can interpret it as the Euclidean distance between points. Another widely used norm is the squared L^2 norm, whose calculation formula is $x^T x$. The squared L^2 norm is more suitable for mathematical and computational operations than the L^2 norm. Each partial derivative of the squared L^2 norm depends only on the corresponding element of x , in comparison to the partial derivatives of the L^2 norm, which depends on the entire vector; this property plays a vital role in optimization algorithms. Another widely used norm operation is the L^1 norm with $p = 1$, which is commonly used in ML when we care about the difference between zero and nonzero elements. The L^1 norm is also known as the **Manhattan distance**.

- **Inverting:** The inverse matrix is such a matrix that $A^{-1}A = I$, where I is an identity matrix. The **identity matrix** is a matrix that does not change any vector when we multiply that vector by that matrix.

We have considered the main linear algebra concepts as well as operations on them. Using this math apparatus, we can define and program many ML algorithms. For example, we can use tensors and matrices to define training datasets for training, and scalars can be used as different types of coefficients. We can use element-wise operations to perform arithmetic operations with a whole dataset (a matrix or a tensor). For example, we can use element-wise multiplication to scale a dataset. We usually use transposing to change a view of a vector or matrix to make them suitable for the dot-product operation. The dot product is usually used to apply a linear function with weights expressed as matrix coefficients to a vector; for example, this vector can be a training sample. Also, dot-product operations are used to update model parameters expressed as matrix or tensor coefficients according to an algorithm.

The norm operation is often used in formulas for loss functions because it naturally expresses the distance concept and can measure the difference between target and predicted values. The inverse matrix is a crucial concept for the analytical solving of linear equations systems. Such systems often appear in different optimization problems. However, calculating the inverse matrix is very computationally expensive.

Tensor representation in computing

We can represent tensor objects in computer memory in different ways. The most obvious method is a simple linear array in computer memory (**random-access memory**, or **RAM**). However, the linear array is also the most computationally effective data structure for modern CPUs. There are two standard practices to organize tensors with a linear array in memory: **row-major ordering** and **column-major ordering**.

In row-major ordering, we place consecutive elements of a row in linear order one after the other, and each row is also placed after the end of the previous one. In column-major ordering, we do the same but with the column elements. Data layouts have a significant impact on computational performance because the speed of traversing an array relies on modern CPU architectures that work with sequential data more efficiently than with non-sequential data. CPU caching effects are the reasons for such behavior. Also, a contiguous data layout makes it possible to use SIMD vectorized instructions that work with sequential data more efficiently, and we can use them as a type of parallel processing.

Different libraries, even in the same programming language, can use different ordering. For example, Eigen uses column-major ordering, but PyTorch uses row-major ordering. So, developers should be aware of internal tensor representation in libraries they use, and also take care of this when performing data loading or implementing algorithms from scratch.

Consider the following matrix:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

Then, in the row-major data layout, members of the matrix will have the following layout in memory:

0	1	2	3	4	5
a11	a12	a13	a21	a22	a23

Table 1.1 – The row-major data layout example

In the case of the column-major data layout, order layout will be next, as shown here:

0	1	2	3	4	5
a11	a21	a12	a22	a13	a23

Table 1.2 – The column-major data layout example

Linear algebra API samples

Let's consider some C++ linear algebra **application programming interfaces (APIs)** and look at how we can use them for creating linear algebra primitives and performing algebra operations with them.

Using Eigen

Eigen is a general-purpose linear algebra C++ library. In Eigen, all matrices and vectors are objects of the `Matrix` template class, and the vector is a specialization of the matrix type, with either one row or one column. Tensor objects are not presented in official APIs but exist as submodules.

We can define the type for a matrix with known 3 x 3 dimensions and a floating-point data type like this:

```
typedef Eigen::Matrix<float, 3, 3> MyMatrix33f;
```

We can define a column vector in the following way:

```
typedef Eigen::Matrix<float, 3, 1> MyVector3f;
```

Eigen already has a lot of predefined types for vector and matrix objects—for example, `Eigen::Matrix3f` (floating-point 3 x 3 matrix type) or `Eigen::RowVector2f` (floating-point 1 x 2 vector type). Also, Eigen is not limited to matrices whose dimensions we know at compile time. We can define matrix types that will take the number of rows or columns at initialization during runtime. To define such types, we can use a special type variable for the `Matrix` class template argument named `Eigen::Dynamic`. For example, to define a matrix of doubles with dynamic dimensions, we can use the following definition:

```
typedef Eigen::
    Matrix<double, Eigen::Dynamic, Eigen::Dynamic>
    MyMatrix;
```

Objects initialized from the types we defined will look like this:

```
MyMatrix33f a;
MyVector3f v;
MyMatrix m(10,15);
```

To put some values into these objects, we can use several approaches. We can use special predefined initialization functions, as follows:

```
a = MyMatrix33f::Zero(); // fill matrix elements with zeros
a = MyMatrix33f::Identity(); // fill matrix as Identity matrix
v = MyVector3f::Random(); // fill matrix elements with random values
```

We can use the *comma-initializer* syntax, as follows:

```
a << 1,2,3,
    4,5,6,
    7,8,9;
```

This code construction initializes the matrix values in the following way:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

We can use direct element access to set or change matrix coefficients. The following code sample shows how to use the `()` operator for such an operation:

```
a(0,0) = 3;
```

We can use the object of the `Map` type to wrap an existent C++ array or vector in the `Matrix` type object. This kind of mapping object will use memory and values from the underlying object, and will not allocate the additional memory and copy the values. The following snippet shows how to use the `Map` type:

```
int data[] = {1,2,3,4};
Eigen::Map<Eigen::RowVectorxi> v(data,4);
std::vector<float> data = {1,2,3,4,5,6,7,8,9};
Eigen::Map<MyMatrix33f> a(data.data());
```

We can use initialized matrix objects in mathematical operations. Matrix and vector arithmetic operations in the `Eigen` library are offered either through overloads of standard C++ arithmetic operators such as `+`, `-`, or `*`, or through methods such as `dot()` and `cross()`. The following code sample shows how to express general math operations in `Eigen`:

```
using namespace Eigen;
auto a = Matrix2d::Random();
auto b = Matrix2d::Random();
auto result = a + b;
result = a.array() * b.array(); // element wise multiplication
result = a.array() / b.array();
a += b;
result = a * b; // matrix multiplication
//Also it's possible to use scalars:
a = b.array() * 4;
```

Notice that, in `Eigen`, arithmetic operators such as `+` do not perform any computation by themselves. These operators return an *expression object*, which describes what computation to perform. The actual computation happens later when the whole expression is evaluated, typically in the `=` arithmetic operator. It can lead to some strange behaviors, primarily if a developer uses the `auto` keyword too frequently.

Sometimes, we need to perform operations only on a part of the matrix. For this purpose, `Eigen` provides the `block` method, which takes four parameters: `i`, `j`, `p`, `q`. These parameters are the block size, `p`, `q`, and the starting point, `i`, `j`. The following code shows how to use this method:

```
Eigen::Matrixxf m(4,4);
Eigen::Matrix2f b = m.block(1,1,2,2); // copying the middle
                                         //part of matrix
m.block(1,1,2,2) *= 4; // change values in original matrix
```

There are two more methods to access rows and columns by index, which are also a type of `block` operation. The following snippet shows how to use the `col` and `row` methods:

```
m.row(1).array() += 3;
m.col(2).array() /= 4;
```

Another important feature of linear algebra libraries is broadcasting, and Eigen supports this with the `colwise` and `rowwise` methods. Broadcasting can be interpreted as a matrix by replicating it in one direction. Take a look at the following example of how to add a vector to each column of the matrix:

```
Eigen::Matrixxf mat(2,4);
Eigen::Vectorxf v(2); // column vector
mat.colwise() += v;
```

This operation has the following result:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} .\text{colwise}() + \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \end{bmatrix}$$

Using xtensor

The `xtensor` library is a C++ library for numerical analysis with multidimensional array expressions. Containers of `xtensor` are inspired by NumPy, the Python array programming library. ML algorithms are mainly described using Python and NumPy, so this library can make it easier to move them to C++. The following container classes implement multidimensional arrays in the `xtensor` library.

The `xarray` type is a dynamically sized multidimensional array, as shown in the following code snippet:

```
std::vector<size_t> shape = { 3, 2, 4 };
xt::xarray<double, xt::layout_type::row_major> a(shape);
```

Dynamic size for the `xarray` type means that this shape can be changed at compilation time.

The `xtensor` type is a multidimensional array whose range is fixed at compilation time. Exact dimension values can be configured in the initialization step, as shown in the following code snippet:

```
std::array<size_t, 3> shape = { 3, 2, 4 };
xt::xtensor<double, 3> a(shape);
```

The `xtensor_fixed` type is a multidimensional array with a dimension shape fixed at compile time, as shown in the following code snippet:

```
xt::xtensor_fixed<double, xt::xshape<3, 2, 4>> a;
```

The `xtensor` library also implements arithmetic operators with expression template techniques such as Eigen (this is a common approach for math libraries implemented in C++). So, the computation happens lazily, and the actual result is calculated when the whole expression is evaluated.

Lazy computation, also known as **lazy evaluation** or **call-by-need evaluation**, is a strategy in programming where the evaluation of an expression is delayed until its value is actually needed.

This contrasts with eager evaluation, where expressions are evaluated immediately upon encountering them. The container definitions are also expressions. There is also a function to force an expression evaluation named `xt::eval` in the `xtensor` library.

There are different kinds of container initialization in the `xtensor` library. Initialization of `xtensor` arrays can be done with C++ initializer lists, as follows:

```
xt::xarray<double> arr1{{1.0, 2.0, 3.0},  
{2.0, 5.0, 7.0},  
{2.0, 5.0, 7.0}}; // initialize a 3x3 array
```

The `xtensor` library also has builder functions for special tensor types. The following snippet shows some of them:

```
std::vector<uint64_t> shape = {2, 2};  
auto x = xt::ones(shape); // creates 2x2 matrix of 1s  
auto y = xt::zero(shape); // creates zero 2x2 matrix  
auto z = xt::eye(shape); // creates 2x2 matrix with ones  
//on the diagonal
```

Also, we can map existing C++ arrays into the `xtensor` container with the `xt::adapt` function. This function returns the object that uses the memory and values from the underlying object, as shown in the following code snippet:

```
std::vector<float> data{1,2,3,4};  
std::vector<size_t> shape{2,2};  
auto data_x = xt::adapt(data, shape);
```

We can use direct access to container elements, with the `()` operator, to set or change tensor values, as shown in the following code snippet:

```
std::vector<size_t> shape = {3, 2, 4};  
xt::xarray<float> a = xt::ones<float>(shape);  
a(2,1,3) = 3.14f;
```

The `xtensor` library implements linear algebra arithmetic operations through overloads of standard C++ arithmetic operators such as `+`, `-`, and `*`. To use other operations such as dot-product operations, we have to link an application with the library named `xtensor-blas`. These operators are declared in the `xt::linalg` namespace.

The following code shows the use of arithmetic operations with the `xtensor` library:

```
auto a = xt::random::rand<double>({2,2});  
auto b = xt::random::rand<double>({2,2});  
auto c = a + b;  
a -= b;  
c = xt::linalg::dot(a,b);  
c = a + 5;
```

To get partial access to the `xtensor` containers, we can use the `xt::view` function. The `view` function returns a new tensor object that shares the same underlying data with the original tensor but with a different shape or strides. This allows you to access the data in the tensor in a different way, without actually changing the underlying data itself. The following sample shows how this function works:

```
xt::xarray<int> a{
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12},
    {13, 14, 15, 16}
};

auto b = xt::view(a, xt::range(1, 3), xt::range(1, 3));
```

This operation takes a rectangular block from the tensor, which looks like this:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 7 \\ 10 & 11 \end{bmatrix}$$

The `xtensor` library implements automatic broadcasting in most cases. When the operation involves two arrays of different dimensions, it transmits the array with the smaller dimension across the leading dimension of the other array, so we can directly add a vector to a matrix. The following code sample shows how easy it is:

```
auto m = xt::random::rand<double>({2, 2});
auto v = xt::random::rand<double>({2, 1});
auto c = m + v;
```

Using Blaze

Blaze is a general-purpose high-performance C++ library for dense and sparse linear algebra. There are different classes to represent matrices and vectors in Blaze.

We can define the type for a matrix with known dimensions and a floating-point data type, like this:

```
typedef blaze::
    StaticMatrix<float, 3UL, 3UL, blaze::columnMajor>
    MyMatrix33f;
```

We can define a vector in the following way:

```
typedef blaze::StaticVector<float, 3UL> MyVector3f;
```

Also, Blaze is not limited to matrices whose dimensions we know at compile time. We can define matrix types that will take the number of rows or columns at initialization during runtime. To define such types, we can use the `blaze::DynamicMatrix` or `blaze::DynamicVector` classes. For example, to define a matrix of doubles with dynamic dimensions, we can use the following definition:

```
typedef blaze::DynamicMatrix<double> MyMatrix;
```

Objects initialized from the types we defined will look like this:

```
MyMatrix33f a;
MyVector3f v;
MyMatrix m(10, 15);
```

To put some values into these objects, we can use several approaches. We can use special predefined initialization functions, as follows:

```
a = blaze::zero<float>(3UL, 3UL); // Zero matrix
a = blaze::IdentityMatrix<float>(3UL); // Identity matrix

blaze::Rand<float> rnd;
v = blaze::generate(3UL, [&](size_t) { return rnd.generate(); });
// Random generated vector
// Matrix filled from the initializer list
a = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

// Matrix filled with a single value
a = blaze::uniform(3UL, 3UL, 3.f);
```

This code construction initializes the matrix values in the following way:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

We can use direct element access to set or change matrix coefficients. The following code sample shows how to use the `()` operator for such an operation:

```
a(0, 0) = 3;
```

We can use the object of the `blaze::CustomVector` type to wrap an existent C++ array or vector into the `Matrix` or `Vector` type object. This kind of mapping object will use memory and values from the underlying object, and will not allocate the additional memory and copy the values. The following snippet shows how to use this approach:

```
std::array<int, 4> data = {1, 2, 3, 4};
blaze::CustomVector<int,
                    blaze::unaligned,
                    blaze::unpadded,
                    blaze::rowMajor>
v2(data.data(), data.size());

std::vector<float> mdata = {1, 2, 3, 4, 5, 6, 7, 8, 9};
blaze::CustomMatrix<float,
                     blaze::unaligned,
                     blaze::unpadded,
                     blaze::rowMajor>
a2(mdata.data(), 3UL, 3UL);
```

We can use initialized matrix objects in mathematical operations. Notice that we used two parameters: `blaze::unaligned` and `blaze::unpadded`. The `unpadded` parameter may be used in some functions or methods to control the behavior of padding or truncation of arrays. This parameter can be important in certain scenarios where you want to avoid unnecessary padding or truncating of data during operations such as reshaping, slicing, or concatenating arrays. The `blaze::unaligned` parameter allows users to perform operations on unaligned data, which can be useful in certain scenarios where the data is not aligned to specific memory boundaries.

Matrix and vector arithmetic operations in the Blaze library are offered either through overloads of standard C++ arithmetic operators such as `+`, `-`, or `*`, or through methods such as `dot()` and `cross()`. The following code sample shows how to express general math operations in Blaze:

```
blaze::StaticMatrix<float, 2UL, 2UL> a = {{1, 2}, {3, 4}};
auto b = a;

// element wise operations
blaze::StaticMatrix<float, 2UL, 2UL> result = a % b;
a = b * 4;

// matrix operations
result = a + b;
a += b;
result = a * b;
```

Notice that, in Blaze, arithmetic operators such as `+` do not perform any computation by themselves. These operators return an *expression object*, which describes what computation to perform. The actual computation happens later when the whole expression is evaluated, typically in the `=` arithmetic operator or in a constructor of a concrete object. It can lead to some non-obvious behaviors, primarily if a developer uses the `auto` keyword too frequently. The library provides two functions, `eval()` and `evaluate()`, to evaluate a given expression. The `evaluate()` function assists in deducing the exact result type of the operation via the `auto` keyword, and the `eval()` function should be used to explicitly evaluate a sub-expression within a larger expression.

Sometimes, we need to perform operations only on a part of the matrix. For this purpose, Blaze provides the `blaze::submatrix` and `blaze::subvector` classes, which can be parameterized with template parameters. These parameters are the top-left starting point and the width and height of a region. Also, there are functions with the same names that take the same arguments and can be used in a runtime. The following code shows how to use this class:

```
blaze::StaticMatrix<float, 4UL, 4UL> m = {{1, 2, 3, 4},
                                             {5, 6, 7, 8},
                                             {9, 10, 11, 12},
                                             {13, 14, 15, 16}};

// make a view of the middle part of matrix
auto b = blaze::submatrix<1UL, 1UL, 2UL, 2UL>(m);

// change values in original matrix
blaze::submatrix<1UL, 1UL, 2UL, 2UL>(m) *= 0;
```

There are two more functions to access rows and columns by index, which are also a type of `block` operation. The following snippet shows how to use the `col` and `row` functions:

```
blaze::row<1UL>(m) += 3;
blaze::column<2UL>(m) /= 4;
```

In contrast to Eigen, Blaze doesn't support implicit broadcasting. But there is the `blaze::expand()` function that can virtually expand a matrix or vector without actual memory allocation. The following code shows how to use it:

```
blaze::DynamicMatrix<float, blaze::rowVector> mat =
    blaze::uniform(4UL, 4UL, 2);
blaze::DynamicVector<float, blaze::rowVector> vec = {1, 2, 3, 4};
auto ex_vec = blaze::expand(vec, 4UL);
mat += ex_vec;
```

The result of this operation will be the following:

```
( 3 4 5 6 )  
( 3 4 5 6 )  
( 3 4 5 6 )  
( 3 4 5 6 )
```

Using ArrayFire

ArrayFire is a general-purpose high-performance C++ library for parallel computing.

Parallel computing is a method of solving complex problems by dividing them into smaller tasks and executing them simultaneously across multiple processors or cores. This approach can significantly speed up the processing time compared to sequential computing, making it an essential tool for data-intensive applications such as ML.

It provides a single `array` type to represent matrices, volumes, and vectors. This array-based notation expresses computational algorithms in readable mathematical notation so users don't need to express parallel computations explicitly. It has extensive vectorization and parallel batched operations. This library supports accelerated execution on CUDA and OpenCL devices. Another interesting feature of the ArrayFire library is that it optimizes memory usage and arithmetic calculations by runtime analysis of the executed code. This becomes possible by avoiding many temporary allocations.

We can define the type for a matrix with known dimensions and a floating-point data type like this:

```
af::array a(3, 3, af::dtype::f32);
```

We can define a 64-bit floating vector in the following way:

```
af::array v(3, af::dtype::f64);
```

To put some values into these objects, we can use several approaches. We can use special predefined initialization functions, as follows:

```
a = af::constant(0, 3, 3); // Zero matrix  
a = af::identity(3, 3); // Identity matrix  
v = af::randu(3); // Random generated vector  
  
// Matrix filled with a single value  
a = af::constant(3, 3, 3);  
  
// Matrix filled from the initializer list  
a = af::array(  
    af::dim4(3, 3),  
    {1.f, 2.f, 3.f, 4.f, 5.f, 6.f, 7.f, 8.f, 9.f});
```

This code construction initializes the matrix values in the following way:

```
1.0 4.0 7.0  
2.0 5.0 8.0  
3.0 6.0 9.0
```

Notice that the matrix was initialized in the column-major format. The `ArrayFire` library doesn't support row-major initialization.

We can use direct element access to set or change matrix coefficients. The following code sample shows how to use the `()` operator for such an operation:

```
a(0,0) = 3;
```

One important difference from other libraries that we discussed is that you can't map existent C/C++ array data to the `ArrayFire` `array` object, as it will be copied. The following snippet shows this situation:

```
std::vector<float> mdata = {1, 2, 3, 4, 5, 6, 7, 8, 9};  
a = af::array(3, 3, mdata.data());
```

Only memory allocated in CUDA or OpenCL devices will not be copied, but `ArrayFire` will take ownership of a pointer.

We can use initialized array objects in mathematical operations. Arithmetic operations in the `ArrayFire` library are offered either through overloads of standard C++ arithmetic operators such as `+`, `-`, or `*`, or through methods such as `af::matmul`. The following code sample shows how to express general math operations in `ArrayFire`:

```
auto a = af::array(af::dim4(2, 2), {1, 2, 3, 4});  
a = a.as(af::dtype::f32);  
auto b = a.copy();  
  
// element wise operations  
auto result = a * b;  
a = b * 4;  
// matrix operations  
result = a + b;  
a += b;  
result = af::matmul(a, b);
```

In contrast to other libraries we already discussed, `ArrayFire` doesn't extensively use template expression for joining arithmetical operations. This library uses a **just-in-time (JIT)** compilation engine that converts mathematical expressions into computational kernels for CUDA, OpenCL, or CPU devices. Also, this engine merges different operations together to provide the best performance. This operation fusion technology decreases the number of kernel calls and reduces global memory operations.

Sometimes, we need to perform operations only on a part of the array. For this purpose, ArrayFire provides a special indexing technique. There are particular classes that can be used to express index sub-ranges for given dimensions. They are as follows:

```
seq - representing a linear sequence  
end - representing the last element of a dimension  
span - representing the entire dimension
```

The following code shows an example of how to access only the central part of a matrix:

```
auto m = af::iota(af::dim4(4, 4));  
auto center = m(af::seq(1, 2), af::seq(1, 2));  
// modify a part of the matrix  
center *= 2;
```

To access and update a particular row or column in the array, there are `row(i)` and `col(i)` methods specifying a single row or column. They can be used in the following way:

```
m.row(1) += 3;  
m.col(2) /= 4;
```

Also, to work with several rows or columns, there are the `rows(first, last)` and `cols(first, last)` methods specifying a span of rows or columns.

ArrayFire doesn't support implicit broadcasting but there is the `af::batchFunc` function that can be used to simulate and parallelize such functionality. In general, this function finds a batch dimension of data and applies the given function to multiple data chunks in parallel. The following code shows how to use it:

```
auto mat = af::constant(2, 4, 4);  
auto vec = af::array(4, {1, 2, 3, 4});  
mat = af::batchFunc(  
    vec,  
    mat,  
    [] (const auto& a, const auto& b) { return a + b; });
```

The result of this operation will be the following:

```
3.0 3.0 3.0 3.0  
4.0 4.0 4.0 4.0  
5.0 5.0 5.0 5.0  
6.0 6.0 6.0 6.0
```

Notice that the vector was a column-major one.

Using Dlib

Dlib is a modern C++ toolkit containing ML algorithms and tools for creating computer vision software in C++. Most of the linear algebra tools in Dlib deal with dense matrices. However, there is also limited support for working with sparse matrices and vectors. In particular, the Dlib tools represent sparse vectors using the containers from the C++ **standard template library (STL)**.

There are two main container types in Dlib to work with linear algebra: the `matrix` and `vector` classes. Matrix operations in Dlib are implemented using the expression templates technique, which allows them to eliminate the temporary matrix objects that would usually be returned from expressions such as $M = A+B+C+D$.

We can create a matrix sized at compile time in the following way, by specifying dimensions as template arguments:

```
Dlib::matrix<double, 3, 1> y;
```

Alternatively, we can create dynamically sized matrix objects. In such a case, we pass the matrix dimensions to the constructor, as shown in the following code snippet:

```
Dlib::matrix<double> m(3, 3);
```

Later, we can change the size of this matrix with the following method:

```
m.set_size(6, 6);
```

We can initialize matrix values with a comma operator, as shown in the following code snippet:

```
m = 54.2, 7.4, 12.1,  
     1, 2, 3,  
     5.9, 0.05, 1;
```

As in the previous libraries, we can wrap an existing C++ array to the matrix object, as shown in the following code snippet:

```
double data[] = {1, 2, 3, 4, 5, 6};  
auto a = Dlib::mat(data, 2, 3); // create matrix with size 2x3
```

Also, we can access matrix elements with the `()` operator to modify or get a particular value, as shown in the following code snippet:

```
m(1, 2) = 3;
```

The Dlib library has a set of predefined functions to initialize a matrix with values such as the identity matrix, ones, or random values, as illustrated in the following code snippet:

```
auto a = Dlib::identity_matrix<double>(3);
auto b = Dlib::ones_matrix<double>(3,4);
auto c = Dlib::randm(3,4); // matrix with random values
                           //with size 3x3
```

Most linear algebra arithmetic operations in the Dlib library are implemented through overloads of standard C++ arithmetic operators such as +, -, or *. Other complex operations are provided by the library as standalone functions.

The following example shows the use of arithmetic operations in the Dlib library:

```
auto c = a + b;
auto e = a * b; // real matrix multiplication
auto d = Dlib::pointwise_multiply(a, b); // element wise
                                         //multiplication
a += 5;
auto t = Dlib::trans(a); // transpose matrix
```

To work with partial access to matrices, Dlib provides a set of special functions. The following code sample shows how to use some of them:

```
a = Dlib::rowm(b,0); // takes first row of matrix
a = Dlib::rowm(b,Dlib::range(0,1)); //takes first two rows
a = Dlib::colm(b,0); // takes first column
// takes a rectangular part from center:
a = Dlib::subm(b, range(1,2), range(1,2));
// initialize part of the matrix:
Dlib::set_subm(b,range(0,1), range(0,1)) = 7;
// add a value to the part of the matrix:
Dlib::set_subm(b,range(0,1), range(0,1)) += 7;
```

Broadcasting in the Dlib library can be modeled with the `set_rowm()`, `set_colm()`, and `set_subm()` functions that give modifier objects for a particular matrix row, column, or rectangular part of the original matrix. Objects returned from these functions support all set or arithmetic operations. The following code snippet shows how to add a vector to the columns:

```
Dlib::matrix<float, 2,1> x;
Dlib::matrix<float, 2,3> m;
Dlib::set_colm(b,Dlib::range(0,1)) += x;
```

In this section, we learned about the main concepts of linear algebra and their implementation in different C++ libraries. We saw how to create matrices and tensors, and how to perform different mathematical operations with them. In the following section, we will see our first complete ML example—solving the regression problem with the linear regression approach.

An overview of linear regression

Consider an example of the real-world supervised ML algorithm called linear regression. In general, **linear regression** is an approach for modeling a target value (dependent value) based on an explanatory value (independent value). This method is used for forecasting and finding relationships between values. We can classify regression methods by the number of inputs (independent variables) and the type of relationship between the inputs and outputs (dependent variables).

Simple linear regression is the case where the number of independent variables is *1*, and there is a linear relationship between the independent (*x*) and dependent (*y*) variables.

Linear regression is widely used in different areas such as scientific research, where it can describe relationships between variables, as well as in applications within industry, such as revenue prediction. For example, it can estimate a trend line that represents the long-term movement in the stock price time-series data. It tells whether the interest value of a specific dataset has increased or decreased over the given period, as illustrated in the following screenshot:

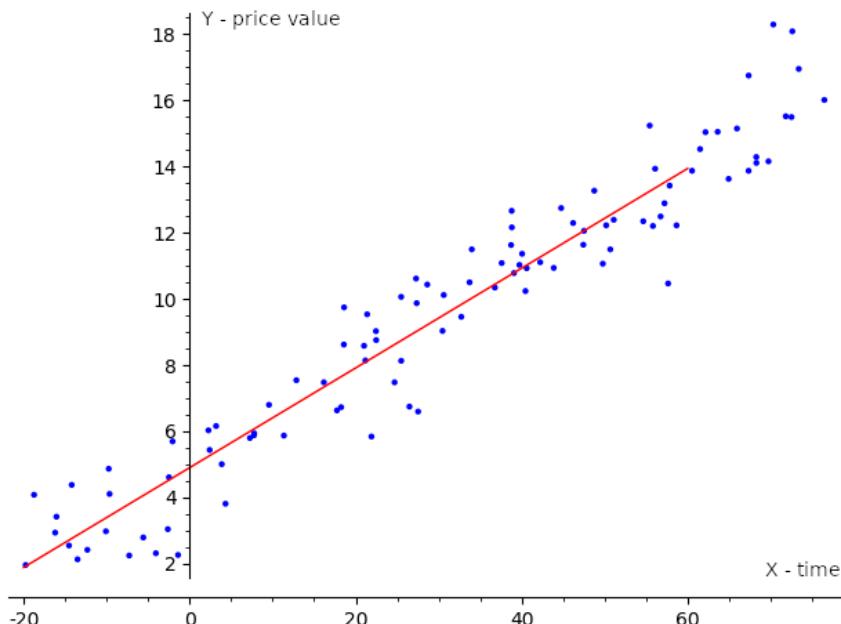


Figure 1.1 – Linear regression visualization

If we have one input variable (independent variable) and one output variable (dependent variable), the regression is called simple, and we use the term **simple linear regression** for it. With multiple independent variables, we call this **multiple linear regression** or **multivariable linear regression**. Usually, when we are dealing with real-world problems, we have a lot of independent variables, so we model such problems with multiple regression models. Multiple regression models have a universal definition that covers other types, so even simple linear regression is often defined using the multiple regression definition.

Solving linear regression tasks with different libraries

Assume that we have a dataset, $\{y_i, x_{i1}, \dots, x_{ip}\}_{i=1}^n$, so that we can express the linear relation between y and x with mathematical formula in the following way:

$$y_i = \beta_0 1 + \beta_1 * x_{i1} + \dots + \beta_{p+1} x_{ip} + \epsilon_i = x_i^T \beta + \epsilon_i, i = 1, \dots, n$$

Here, p is the dimension of the independent variable, and T denotes the transpose, so that $x_i^T \beta$ is the inner product between vectors x_i and β . Also, we can rewrite the previous expression in matrix notation, as follows:

$$y = X\beta + \epsilon$$

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

$$X = \begin{pmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_n^T \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1p} \\ 1 & x_{21} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \dots & x_{np} \end{pmatrix}$$

$$\beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix}$$

$$\epsilon = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

The preceding matrix notation can be explained as follows:

- y : This is a vector of observed target values.
- x : This is a matrix of row-vectors, x_i , which are known as explanatory or independent values.
- β : This is a $(p+1)$ dimensional parameters vector.
- ϵ : This is called an error term or noise. This variable captures all other factors that influence the y -dependent variable other than the regressors.

When we are considering simple linear regression, p is equal to 1, and the equation will look like this:

$$y_i = \beta_0 1 + \beta_1 x_i + \epsilon_i$$

The goal of the linear regression task is to find parameter vectors that satisfy the previous equation. Usually, there is no exact solution to such a system of linear equations, so the task is to estimate parameters that satisfy these equations with some assumptions. One of the most popular estimation approaches is one based on the principle of least squares: minimizing the sum of the squares of the differences between the observed dependent variable in the given dataset and those predicted by the linear function. This is called the **ordinary least squares (OLS)** estimator. So, the task can be formulated with the following formula:

$$\hat{\beta} = \operatorname{argmin}_{\beta} S(\beta)$$

In the preceding formula, the objective function, S , is given by the following matrix notation:

$$S(\beta) = \sum_{i=1}^n \left| y_i - \sum_{j=1}^p X_{ij} \beta_j \right|^2 = \|y - X\beta\|^2$$

This minimization problem has a unique solution, in the case that the p columns of the x matrix are linearly independent. We can get this solution by solving the *normal equation*, as follows:

$$\beta = (X^T X)^{-1} X^T y$$

Linear algebra libraries can solve such equations directly with an analytical approach, but it has one significant disadvantage—computational cost. In the case of large dimensions of y and x , requirements for computer memory amount and computational time are too big to solve real-world tasks.

So, usually, this minimization task is solved with iterative approaches. **gradient descent (GD)** is an example of such an algorithm. GD is a technique based on the observation that if the function $S(\beta)$ is defined and is differentiable in a neighborhood of a point β , then $S(\beta)$ decreases fastest when it goes in the direction of the negative gradient of S at point β .

We can change our $S(\beta)$ objective function to a form more suitable for an iterative approach. We can use the **mean squared error (MSE)** function, which measures the difference between the estimator and the estimated value, as illustrated here:

$$S(\beta) = \frac{1}{n} \sum_{i=1}^n (y_i - X_i \beta)^2$$

In the case of multiple regression, we take partial derivatives for this function for each x components, as follows:

$$\frac{\partial S}{\partial \beta_j}$$

So, in the case of linear regression, we take the following derivatives:

$$\begin{aligned}\frac{\partial S}{\partial \beta_0} &= \frac{2}{n} \sum_{i=1}^n (X_i \beta - y_i) \\ \frac{\partial S}{\partial \beta_1} &= \frac{2}{n} \sum_{i=1}^n (X_i \beta - y_i) X_i\end{aligned}$$

The whole algorithm has the following description:

1. Initialize β with zeros.
2. Define a value for the learning rate parameter that controls how much we are adjusting parameters during the learning procedure.
3. Calculate the following values of β :

$$\begin{aligned}\beta_0 &= \beta_0 - \gamma \frac{2}{n} \sum_{i=1}^n (X_i \beta - y_i) \\ \beta_1 &= \beta_1 - \gamma \frac{2}{n} \sum_{i=1}^n (X_i \beta - y_i) X_i\end{aligned}$$

4. Repeat steps 1–3 a number of times or until the MSE value reaches a reasonable amount.

The previously described algorithm is one of the simplest supervised ML algorithms. We described it with the linear algebra concepts we introduced earlier in the chapter. Later, it became more evident that almost all ML algorithms use linear algebra under the hood. Linear regression is widely used in various industries for predictive analysis, forecasting, and decision-making. Here are some real-world examples of linear regression applications in finance, marketing, and healthcare. Linear regression can be used to predict stock prices based on historical data such as company earnings, interest rates, and economic indicators. This helps investors make informed decisions about when to buy or sell stocks. Linear regression models can be built to predict customer behavior based on demographic information, purchase history, and other relevant data. This allows marketers to target their campaigns more effectively and optimize their marketing spend. Linear regression is used to analyze medical data to identify patterns and relationships that can help improve patient outcomes. For example, it can be used to study the impact of certain treatments on patient health.

The following samples show the higher-level API in different linear algebra libraries for solving the *linear regression* task, and we provide them to show how libraries can simplify the complicated math used underneath. We will give the details of the APIs used in these samples in the following chapters.

Solving linear regression tasks with Eigen

There are several iterative methods for solving problems of the $Ax = b$ form in the `Eigen` library. The `LeastSquaresConjugateGradient` class is one of them, which allows us to solve linear regression problems with the conjugate gradient algorithm. The `ConjugateGradient` algorithm can converge more quickly to the function's minimum than regular GD but requires that matrix A is positively defined to guarantee numerical stability. The `LeastSquaresConjugateGradient` class has two main settings: the maximum number of iterations and a tolerance threshold value that is used as a stopping criterion as an upper bound to the relative residual error, as illustrated in the following code block:

```
typedef float DType;
using Matrix = Eigen::Matrix<DType, Eigen::Dynamic, Eigen::Dynamic>;
int n = 10000;
Matrix x(n,1);
Matrix y(n,1);
Eigen::LeastSquaresConjugateGradient<Matrix> gd;
gd.setMaxIterations(1000);
gd.setTolerance(0.001) ;
gd.compute(x);
auto b = gddg.solve(y);
```

For new x inputs, we can predict new y values with matrix operations, as follows:

```
Eigen::Matrixxf new_x(5, 2);
new_x << 1, 1, 1, 2, 1, 3, 1, 4, 1, 5;
auto new_y = new_x.array().rowwise() * b.transpose().array();
```

Also, we can calculate the parameter's b vector (the linear regression task solution) by solving the *normal equation* directly, as follows:

```
auto b = (x.transpose() * x).ldlt().solve(x.transpose() * y);
```

Solving linear regression tasks with Blaze

Since Blaze is just a mathematical library, there are no special classes or functions to solve linear regression tasks. However, the normal equation approach can be easily implemented. Let's see how to define and solve linear regression tasks with Blaze:

Assume we have our training data:

```
typedef blaze::DynamicMatrix<float,blaze::columnMajor> Matrix;
typedef blaze::DynamicVector<float,blaze::columnVector> Vector;

// the first column of X is just 1 for the bias term
Matrix x(n, 2UL);
Matrix y(n, 1UL);
```

So we can find linear regression coefficients in the following way:

```
// calculate X^T*X
auto xtx = blaze::trans(x) * x;

// calculate the inverse of X^T*X
auto inv_xtx = blaze::inv(xtx);

// calculate X^T*y
auto xty = blaze::trans(x) * y;

// calculate the coefficients of the linear regression
Matrix beta = inv_xtx * xty;
```

Then, we can use estimated coefficients for making predictions on new data. The following code snippet shows how to do it:

```
auto line_coeffs = blaze::expand(
    blaze::row<0UL>(blaze::trans(beta)), new_x.rows());
auto new_y = new_x % line_coeffs;
```

Notice that we virtually expand the coefficients vector to perform element-wise multiplication with the x data.

Solving linear regression tasks with ArrayFire

ArrayFire also doesn't have special functions and classes to solve this type of problem. However, as it has all the necessary mathematical abstraction, the normal equation approach can be applied. Another approach is an iterative one that uses GD. This algorithm was described in the first section of this chapter. Such a technique eliminates the necessity of calculating inverse matrices, making it possible to apply it to a larger amount of training data. Calculating an inverse for a large matrix is a very performance-expensive operation.

Let's define a lambda function to calculate prediction values from data and coefficients:

```
auto predict = [] (auto& v, auto& w) {
    return af::batchFunc(v, w, [] (const auto& a, const auto& b) {
        return af::sum(a * b, /*dim*/ 1);
    });
};
```

Assume that we have training data defined in the x and y variables. We define the `train_weights` variable to hold and update coefficients that want to learn from the training data:

```
// the first column is for the bias term
af::dim4 weights_dim(1, 2);
auto train_weights = af::constant(0.f, weights_dim, af::dtype::f32);
```

Then, we can define the GD loop where we will iteratively update coefficients. The following code snippet shows how to implement it:

```
af::array j, dj; // cost value and its gradient
float lr = 0.1f; // learning rate
int n_iter = 300;
for (int i = 0; i < n_iter; ++i) {
    std::cout << "Iteration " << i << ":\n";
    // get the cost
    auto h = predict(x, train_weights);
    auto diff = (y - h);
```

```
auto j = af::sum(diff * diff) / n;
af_print(j);

// find the gradient of cost
auto dm = (-2.f / n) * af::sum(x.col(1) * diff);
auto dc = (-2.f / n) * af::sum(diff);
auto dj = af::join(1, dc, dm);

// update the parameters via gradient descent
train_weights = train_weights - lr * dj;
}
```

The most important part of this loop is calculating the prediction error:

```
auto h = predict(x, train_weights);
auto diff = (y - h);
```

Another important part is calculating the gradient values based on partial derivatives related to each of the coefficients:

```
auto dm = (-2.f / n) * af::sum(x.col(1) * diff);
auto dc = (-2.f / n) * af::sum(diff);
```

We join these values into one vector to make a single expression for updating the training parameters:

```
auto dj = af::join(1, dc, dm);
train_weights = train_weights - lr * dj;
```

We can stop this training loop after a number of iterations or when the cost function value reaches some appropriate convergence value. The cost function value can be calculated in the following way:

```
auto j = af::sum(diff * diff) / n;
```

You can see that this is just a sum of squared errors for all training samples.

Linear regression with Dlib

The Dlib library provides the `krr_trainer` class, which can get the template argument of the `linear_kernel` type to solve linear regression tasks. This class implements direct analytical solving for this type of problem with the kernel ridge regression algorithm, as illustrated in the following code block:

```
std::vector<matrix<double>> x;
std::vector<float> y;
krr_trainer<KernelType> trainer;
trainer.set_kernel(KernelType());
decision_function<KernelType> df = trainer.train(x, y);
```

For new `x` inputs, we can predict new `y` values in the following way:

```
std::vector<matrix<double>> new_x;
for (auto& v : x) {
    auto prediction = df(v);
    std::cout << prediction << std::endl;
}
```

In this section, we learned how to solve the linear regression problem with different C++ libraries. We saw that some of them contain the complete algorithm implementation that can be easily applied, and we saw how to implement this approach from scratch using just basic linear algebra primitives.

Summary

In this chapter, we learned what ML is, how it differs from other computer algorithms, and how it became so popular. We also became familiar with the necessary mathematical background required to begin working with ML algorithms. We looked at software libraries that provide APIs for linear algebra and also implemented our first ML algorithm—linear regression.

There are other linear algebra libraries for C++. Moreover, the popular deep learning frameworks use their own implementations of linear algebra libraries. For example, the MXNet framework is based on the mshadow library, and the PyTorch framework is based on the ATen library. Some of these libraries can use GPU or special CPU instructions to speed up calculations. Such features do not usually change the API but require some additional library initialization settings or explicit object conversion to different backends such as CPUs or GPUs.

Real ML projects can be challenging and complex. Common pitfalls include data quality issues, overfitting and underfitting, choosing the wrong model, and not having enough computing resources. Poor data quality can also affect model performance. It's essential to clean and preprocess data to remove outliers, handle missing values, and transform features for better representation. The model should be chosen based on the nature of the problem and the available data. Overfitting occurs when the model memorizes the training data instead of learning general patterns, while underfitting happens when the model cannot capture the underlying structure of the data. To avoid these pitfalls, it is important to have a clear understanding of the problem, use appropriate preprocessing techniques for data, choose the right model, and evaluate the performance using metrics that are relevant to the task. Best practices in ML also include monitoring the model's performance in production and making adjustments as needed. We will discuss the details of these techniques throughout the book. In the next two chapters, we will learn more about available software tools that are necessary to implement more complicated algorithms, and we will also learn more theoretical background on how to manage ML algorithms.

Further reading

- *Basic Linear Algebra for Deep Learning*: <https://towardsdatascience.com/linear-algebra-for-deep-learning-f21d7e7d7f23>
- *Deep Learning*, an MIT Press book: https://www.deeplearningbook.org/contents/linear_algebra.html
- *What is Machine Learning?*: <https://www.mathworks.com/discovery/machine-learning.html>
- The Eigen library documentation: <https://gitlab.com/libeigen/eigen>
- The xtensor library documentation: <https://xtensor.readthedocs.io/en/latest/>
- The Dlib library documentation: <http://dlib.net/>
- The blaze library documentation: <https://bitbucket.org/blaze-lib/blaze/wiki/Home>
- The ArrayFire library documentation: <https://arrayfire.org/docs/index.htm>

2

Data Processing

One of the essential things in **machine learning (ML)** is the data that we use for training. We can gather training data from the processes we work with, or we can take already prepared training data from third-party sources. In any case, we have to store training data in a file format that should satisfy our development requirements. These requirements depend on the task we solve, as well as the data-gathering process. Sometimes, we need to transform data stored in one format to another to satisfy our needs. Examples of such needs are as follows:

- Increasing human readability to ease communication with engineers
- The existence of compression possibility to allow data to occupy less space on secondary storage
- The use of data in the binary form to speed up the parsing process
- Supporting complex relations between different parts of data to make precise mirroring of a specific domain
- Platform independence to be able to use the dataset in different development and production environments

Today, there exists a variety of file formats that are used for storing different kinds of information. Some of these are very specific, and some of them are general-purpose. There are software libraries that allow us to manipulate these file formats. There is rarely a need to develop a new format and parser from scratch. Using existing software for reading a format can significantly reduce development and testing time, which allows us to focus on particular tasks.

This chapter discusses how to process popular file formats that we use for storing data. It shows what libraries exist for working with **JavaScript Object Notation (JSON)**, **Comma-Separated Values (CSV)**, and **Hierarchical Data Format v5 (HDF5)** formats. This chapter also introduces the basic operations required to load and process image data with the OpenCV and Dlib libraries and how to convert the data format used in these libraries to data types used in linear algebra libraries. It also describes data normalization techniques such as feature scaling and standardization procedures to deal with heterogeneous data.

This chapter will cover the following topics:

- Parsing data formats to C++ data structures
- Initializing matrix and tensor objects from C++ data structures
- Manipulating images with the OpenCV and Dlib libraries
- Transforming images into matrix and tensor objects of various libraries
- Normalizing data

Technical requirements

The required technologies and installations for this chapter are as follows:

- Modern C++ compiler with C++17/C++20 support
- CMake build system version >= 3.22
- Dlib library installation
- mlpack library installation
- Flashlight library installation
- Eigen library installation
- hdf5lib library installation
- HighFive library installation
- nlohmann-json library installation
- Fast-CPP-CSV-Parser library installation

The code for this chapter can be found at the following GitHub repo: <https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition>

Parsing data formats to C++ data structures

The most popular format for representing structured data is called **CSV**. This format is just a text file with a two-dimensional table in it whereby values in a row are separated with commas, and rows are placed on every new line. It looks like this:

```
1, 2, 3, 4
5, 6, 7, 8
9, 10, 11, 12
```

The advantages of this file format are that it has a straightforward structure, many software tools can process it, it is human-readable, and it is supported on a variety of computer platforms. Disadvantages are a lack of support for multidimensional data and data with complex structuring, as well as slow parsing speed in comparison with binary formats.

Another widely used format is **JSON**. Although the format contains JavaScript in its abbreviation, we can use it with almost all programming languages. This is a file format with name-value pairs and arrays of such pairs. It has rules on how to group such pairs into distinct objects and array declarations, and there are rules on how to define values of different types. The following code sample shows a file in JSON format:

```
{  
    "name": "Bill",  
    "age": 25,  
    "phones": [  
        {  
            "type": "home",  
            "number": 43534590  
        },  
        {  
            "type": "work",  
            "number": 56985468  
        }  
    ]  
}
```

The advantages of this format are human readability, software support on many computer platforms, and the possibility to store hierarchical and nested data structures. Disadvantages are its slow parsing speed in comparison with binary formats and the fact it is not very useful for representing numerical matrices. In terms of character reading, binary formats offer more direct access to the underlying data structure, allowing for faster and more precise character extraction. With text formats, additional steps may be required to convert the characters into their numerical representations, potentially introducing additional processing overhead.

Often, we use a combination of file formats to represent a complex dataset. For example, we can describe object relations with JSON, and data/numerical data in the binary form can be stored in a folder structure on the filesystem with references to it in JSON files.

HDF5 is a specialized file format for storing scientific data. This file format was developed to store heterogeneous multidimensional data with a complex structure. It provides fast access to single elements because it has optimized data structures for using secondary storage. Furthermore, HDF5 supports data compression. In general, this file format consists of named groups that contain multidimensional arrays of multitype data. Each element of this file format can contain metadata, as illustrated in the following diagram:

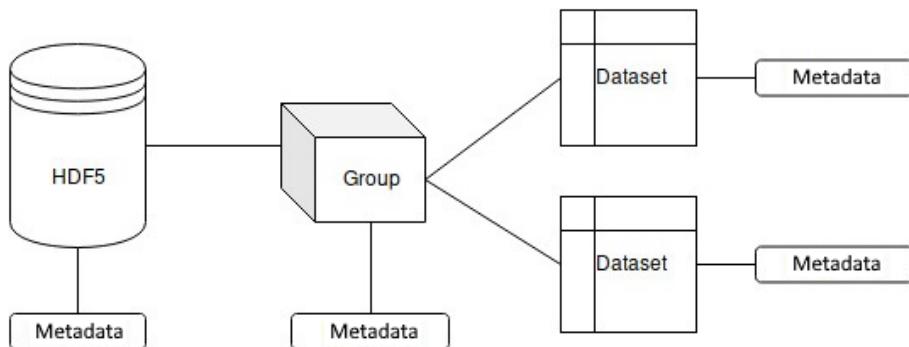


Figure 2.1 – HDF5 format structure

The advantages of this format are its high read-and-write speed, fast access to distinct elements, and its ability to support data with a complex structure and various types of data. Disadvantages are the requirement of specialized tools for editing and viewing by users, the limited support of type conversions among different platforms, and using a single file for the whole dataset. The last issue makes data restoration almost impossible in the event of file corruption. So, it makes sense to regularly back up your data to prevent data loss in case of hardware failure or accidental deletion.

There are a lot of other formats for representing datasets for ML, but we found the ones mentioned to be the most useful.

Reading CSV files with the Fast-CPP-CSV-Parser library

Consider how to deal with CSV format in C++. There are many different libraries for parsing CSV format with C++. They have different sets of functions and different ways to integrate them into applications. The easiest way to use C++ libraries is to use header-only libraries because this eliminates the need to build and link them. We propose to use the `Fast - CPP - CSV - Parser` library because it is a small single-file header-only library with the minimal required functionality, which can be easily integrated into a development code base. It also provides a fast and efficient way to read and write CSV data.

As an example of a CSV file format, we use the `Iris` dataset, which describes three different types of iris plants (*Iris setosa*, *Iris versicolor*, and *Iris virginica*) and was conceived by R.A. Fisher. Each row in the file contains the following fields: sepal length, sepal width, petal length, petal width, and a string with a class name. This dataset is used for examples of how to classify an unknown iris flower based on these four features.

Note

The reference to the Iris dataset is the following: *Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [https://archive.ics.uci.edu/static/public/53/iris.zip]. Irvine, CA: University of California, School of Information and Computer Science.*

To read this dataset with the `Fast-CPP-CSV-Parser` library, we need to include a single header file, as follows:

```
#include <csv.h>
```

Then, we define an object of the type `io::CSVReader`. We must define the number of columns as a template parameter. This parameter is one of the library limitations because we need to be aware of the CSV file structure. The code for this is illustrated in the following snippet:

```
const uint32_t columns_num = 5;
io::CSVReader<columns_num> csv_reader(file_path);
```

Next, we define containers for storing the values we read, as follows:

```
std::vector<std::string> categorical_column;
std::vector<double> values;
```

Then, to make our code more generic and gather all information about column types in one place, we introduce the following helper types and functions. We define a tuple object that describes values for a row, like this:

```
using RowType =
    std::tuple<double, double, double, double, std::string>;
RowType row;
```

The reason for using a tuple is that we can easily iterate it with metaprogramming techniques. Then, we define two helper functions. One is for reading a row from a file, and it uses the `read_row()` method of the `io::CSVReader` class. The `read_row()` method takes a variable number of parameters of different types. Our `RowType` type describes these values. We do automatic parameter filling by using the `std::index_sequence` type with the `std::get` function, as illustrated in the following code snippet:

```
template <std::size_t... Idx, typename T, typename R>
bool read_row_help(std::index_sequence<Idx...>, T& row, R& r) {
    return r.read_row(std::get<Idx>(row)...);
}
```

The second helper function uses a similar technique for transforming a row tuple object to our value vectors, as follows:

```
template <std::size_t... Idx, typename T>
void fill_values(std::index_sequence<Idx...>,
                 T& row,
                 std::vector<double>& data) {
    data.insert(data.end(), {std::get<Idx>(row)...});
}
```

Now, we can put all the parts together. We define a loop where we continuously read row values and move them to our containers. After we read a row, we check the return value of the `read_row()` method, which tells us if the read was successful or not. A `false` return value means that we have reached the end of the file. In the case of a parsing error, we catch an exception from the `io::error` namespace. There are exception types for different parsing failures. In the following example, we handle number parsing errors:

```
try {
    bool done = false;
    while (!done) {
        done = !read_row_help(
            std::make_index_sequence<
                std::tuple_size<RowType>::value>{},
            row, csv_reader);

        if (!done) {
            categorical_column.push_back(std::get<4>(row));
            fill_values(
                std::make_index_sequence<columns_num - 1>{},
                row, values);
        }
    }
} catch (const std::exception& err) {
    // Ignore badly formatted samples
    std::cerr << err.what() << std::endl;
}
```

Also, notice that we moved only four values to our vector of doubles because the last column contains string objects that we put into another vector of categorical values.

In this code sample, we saw how to parse the particular dataset with string and numerical values into two containers: `std::vector<std::string> categorical_column` and `std::vector<double> values`.

Preprocessing CSV files

Sometimes, the data we have comes in a format that's incompatible with the libraries we want to use. For example, the Iris dataset file contains a column that contains strings. Many ML libraries cannot read such values because they assume that CSV files contain only numerical values that can be directly loaded into an internal matrix representation.

So, before using such datasets, we need to preprocess them. In the case of the Iris dataset, we need to replace the `categorical` column containing string labels with numeric encoding. In the following code sample, we replace strings with distinct numbers, but in general, such an approach is a bad idea, especially for classification tasks. ML algorithms usually learn only numerical relations, so a more suitable approach would be to use specialized encoding—for example, one-hot encoding. One-hot encoding is a method used in ML to represent categorical data as numerical values. It involves creating a binary vector for each unique value in the categorical feature, where only one element in the vector is set to 1 and all others are set to 0. The code can be seen in the following block:

```
#include <fstream>
#include <regex>
...
std::ifstream data_stream("iris.data");
std::string data_string(
    (std::istreambuf_iterator<char>(data_stream)),
    std::istreambuf_iterator<char>())
);
data_string = std::regex_replace(data_string,
                                std::regex("Irissetosa"),
                                "1");
data_string = std::regex_replace(data_string,
                                std::regex("Irisversicolor"),
                                "2");
data_string = std::regex_replace(data_string,
                                std::regex("Irisvirginica"),
                                "3");
std::ofstream out_stream("iris_fix.csv");
out_stream << data_string;
```

We read the CSV file content to the `std::string` object with the `std::ifstream` object. Also, we use `std::regex` routines to replace string class names with numbers. Using `regex` functions allows us to reduce code size and make it more expressive in comparison with the loop approach, which typically uses the `std::string::find()` and `std::string::replace()` methods. After replacing all categorical class names in the file, we create a new file with the `std::ofstream` object.

Reading CSV files with the `mlpack` library

Many ML frameworks already have routines for reading the CSV file format to their internal representations. In the following code sample, we show how to load a CSV file with the `mlpack` library into the `matrix` object. The CSV parser in this library can automatically create numerical mappings for non-numeric values, so we easily load the Iris dataset without additional preprocessing.

To read a CSV file with the `mlpack` library, we have to include the corresponding headers, as follows:

```
#include <mlpack/core.hpp>
using namespace mlpack;
```

We can use the `data::Load` function to load the CSV data from a file, as illustrated in the following code snippet:

```
arma::mat dataset;
data::DatasetInfo info;
data::Load(file_name,
           dataset,
           info,
           /*fail with error*/ true);
```

Notice that the `data::Load` function takes the `dataset` matrix object to load data and the `info` object of the `DatasetInfo` type that can be used to get additional information about the loaded file. Also, the last Boolean `true` parameter was used to make the function throw an exception in the loading error case. For example, we can get the number of columns and an available mapping for non-numeric values, as follows:

```
std::cout << "Number of dimensions: " << info.Dimensionality()
             << std::endl;
std::cout << "Number of classes: " << info.NumMappings(4)
             << std::endl;
```

Due to the fact that data is loaded as is, there are no automatic assumptions about dataset structure. So, to extract labels, we need to manually divide the loaded matrix object as follows:

```
arma::Row<size_t> labels;
labels = arma::conv_to<arma::Row<size_t>>::from(
    dataset.row(dataset.n_rows - 1));
dataset.shed_row(dataset.n_rows - 1);
```

We used the `arma::conv_to` function to create the standalone `arma::Row` object from the dataset row. Then, we deleted the last row from the dataset with the `shed_row` method.

Reading CSV files with the Dlib library

The `Dlib` library can load CSV files directly to its matrix type, as the `mlpack` library does. For this operation, we can use a simple C++ streaming operator and a standard `std::ifstream` object.

As a first step, we make the necessary `include` statements, as follows:

```
include <Dlib/matrix.h>
using namespace Dlib;
```

Then, we define a `matrix` object and load data from the file, like this:

```
matrix<double> data;
std::ifstream file("iris_fix.csv");
file >> data;
std::cout << data << std::endl;
```

In the `Dlib` library, `matrix` objects are used for training ML algorithms directly without the need to transform them into intermediate dataset types.

Reading JSON files with the `nlohmann-json` library

Some datasets come with structured annotations and can contain multiple files and folders. An example of such a complex dataset is the **Common Objects in Context (COCO)** dataset. This dataset contains a text file with annotations for describing relations between objects and their structural parts. This widely known dataset is used to train models for segmentation, object detection, and classification tasks. Annotations in this dataset are defined in the JSON file format. JSON is a widely used file format for objects' (entities') representations.

It is just a text file with special notations for describing relations between objects and their parts. In the following code samples, we show how to work with this file format using the `nlohmann-json` library. This library provides a simple and intuitive interface for working with JSON, making it easy to convert between JSON strings and C++ data structures such as maps, vectors, and custom classes. It also supports various features such as automatic type conversion, pretty printing, and error handling. However, we are going to use a more straightforward dataset that defines paper reviews. The authors of this dataset are Keith, B., Fuentes, E., and Meneses, C., and they made this dataset for their work titled *A Hybrid Approach for Sentiment Analysis Applied to Paper Reviews* (2017).

The following sample shows a reduced part of this JSON-based dataset:

```
{
  "paper": [
    {
      "id": 1,
      "preliminary_decision": "accept",
      "review": [
        {
          "confidence": "4",
          "evaluation": "1",
          "id": 1,
          "lan": "es",
          "orientation": "0",
          "remarks": "",
          "text" : "- El artículo aborda un problema contingente\n"
        }
      ]
    }
  ]
}
```

```
        y muy relevante, e incluye tanto un diagnóstico\n
        nacional de uso de buenas prácticas como una solución\n
        (buenas prácticas concretas)... ",\n
        "timespan": "2010-07-05"\n    },\n    {\n        "confidence": "4",\n        "evaluation": "1",\n        "id": 2,\n        "lan": "es",\n        "orientation": "1",\n        "remarks": "",\n        "text" : "El artículo presenta recomendaciones\n
prácticas para el desarrollo de software seguro... ",\n        "timespan": "2010-07-05"\n    },\n    {\n        "confidence": "5",\n        "evaluation": "1",\n        "id": 3,\n        "lan": "es",\n        "orientation": "1",\n        "remarks": "",\n        "text" : "- El tema es muy interesante y puede ser de\n
muchía ayuda una guía para incorporar prácticas de\nseguridad... ",\n        "timespan": "2010-07-05"\n    }\n]\n}\n]
```

There are two main approaches to parsing and processing JSON files, which are listed as follows:

- The first approach assumes the parsing of whole files at once and creating a **Document Object Model (DOM)**. The DOM is a hierarchical structure of objects that represents entities stored in files. It is usually stored in computer memory, and, in the case of large files, it can occupy a significant amount of memory.

- Another approach is to parse the file continuously and provide an **application program interface (API)** for a user to handle and process each event related to the file-parsing process. This second approach is usually called **Simple API for XML (SAX)**. Despite its name, it's a general approach that is used with non-XML data too. SAX is faster than DOM for parsing large XML files because it doesn't build a complete tree representation of the entire document in memory. However, it can be more difficult to use for complex operations that require accessing specific parts of the document.

Using a DOM for working with training datasets usually requires a lot of memory for structures that are useless for ML algorithms. So, in many cases, it is preferable to use the SAX interface. It allows us to filter irrelevant data and initialize structures that we can use directly in our algorithms. In the following code sample, we use this approach.

As a preliminary step, we define types for paper/review entities, as follows:

```
...
struct Paper {
    uint32_t id{0};
    std::string preliminary_decision;
    std::vector<Review> reviews;
};

using Papers = std::vector<Paper>;
...

struct Review {
    std::string confidence;
    std::string evaluation;
    uint32_t id{0};
    std::string language;
    std::string orientation;
    std::string remarks;
    std::string text;
    std::string timespan;
};
```

Then, we declare a type for the object, which will be used by the parser to handle parsing events. This type should be inherited from the `nlohmann::json::json_sax_t` base class, and we need to override virtual handler functions that the parser will call when a particular parsing event occurs, as illustrated in the following code block:

```
#include <nlohmann/json.hpp>
using json = nlohmann::json;
...
struct ReviewsHandler
```

```

: public json::json_sax_t {
    ReviewsHandler(Papers* papers) : papers_(papers) {}
    bool null() override;
    bool boolean(bool) override;
    bool number_integer(number_integer_t) override;
    bool number_float(number_float_t, const string_t&) override;
    bool binary(json::binary_t&) override;
    bool parse_error(std::size_t, const std::string&,
                     const json::exception& ex) override;
    bool number_unsigned(number_unsigned_t u) override;
    bool string(string_t& str) override ;
    bool key(string_t& str) override;
    bool start_object(std::size_t) override;
    bool end_object() override;
    bool start_array(std::size_t)override;
    bool end_array() override;

    Paper paper_;
    Review review_;
    std::string key_;
    Papers* papers_{nullptr};
    HandlerState state_{HandlerState::None};
};


```

We have to override all methods, but we can provide real handler implementations only for objects, arrays parsing events, and events for parsing unsigned int/string values. Other methods can have trivial implementations as follows:

```

bool number_float(number_float_t, const string_t&) override {
    return true;
}

```

Now, we can use the `nlohmann::json::sax_parse` method to load a JSON file; this method takes the `std::istream` object and a `handler` object as the second argument. The following code block shows how to use it:

```

std::ifstream file(filename);
if (file) {
    // define papers data container to be filled
    Papers papers;
    // define object with SAX handlers
    ReviewsHandler handler(&papers);
    // parse file
    bool result = json::sax_parse(file, &handler);
}

```

```
// check parsing result
if (!result) {
    throw std::runtime_error(handler.error_);
}
return papers;
} else {
    throw std::invalid_argument("File can't be opened " + filename);
}
```

When there are no parsing errors, we will have an initialized array of `Paper` type objects. Consider, more precisely, the event handler's implementation details. Our event handler works as a state machine. In one state, we populate it with the `Review` objects, and in another one, with the `Paper` objects, and there are states for other events, as shown in the following code snippet:

```
enum class HandlerState {
    None,
    Global,
    PapersArray,
    Paper,
    ReviewArray,
    Review
};
```

We parse the unsigned `unit` values only for the `Id` attributes of the `Paper` and the `Review` objects, and we update these values according to the current state and the previously parsed key, as follows:

```
bool number_unsigned(number_unsigned_t u) override {
    bool res{true};
    try {
        if (state_ == HandlerState::Paper && key_ == "id") {
            paper_.id = u;
        } else if (state_ == HandlerState::Review && key_ == "id") {
            review_.id = u;
        } else {
            res = false;
        }
    } catch (...) {
        res = false;
    }
    key_.clear();
    return res;
}
```

String values also exist in both types of objects, so we do the same checks to update corresponding values, as follows:

```
bool string(string_t& str) override {
    bool res{true};
    try {
        if (state_ == HandlerState::Paper &&
            key_ == "preliminary_decision") {
            paper_.preliminary_decision = str;
        } else if (state_ == HandlerState::Review &&
                   key_ == "confidence") {
            review_.confidence = str;
        } else if (state_ == HandlerState::Review &&
                   key_ == "evaluation") {
            review_.evaluation = str;
        } else if (state_ == HandlerState::Review &&
                   key_ == "lan") {
            review_.language = str;
        } else if (state_ == HandlerState::Review &&
                   key_ == "orientation") {
            review_.orientation = str;
        } else if (state_ == HandlerState::Review &&
                   key_ == "remarks") {
            review_.remarks = str;
        } else if (state_ == HandlerState::Review &&
                   key_ == "text") {
            review_.text = str;
        } else if (state_ == HandlerState::Review &&
                   key_ == "timespan") {
            review_.timespan = str;
        } else {
            res = false;
        }
    } catch (...) {
        res = false;
    }

    key_.clear();
    return res;
}
```

The event handler for the JSON key attribute stores the key value to the appropriate variable, which we use to identify a current object in the parsing process, as follows:

```
bool key(string_t& str) override {
    key_ = str;
    return true;
}
```

The `start_object` event handler switches states according to the current key value and the previous state value. We base the current implementation on the knowledge of the structure of the current JSON file: there is no array of `Paper` objects, and each `Paper` object includes an array of reviews. It is one of the limitations of the SAX interface—we need to know the structure of the document to implement all event handlers correctly. The code can be seen in the following block:

```
bool start_object(std::size_t) override {
    if (state_ == HandlerState::None && key_.empty()) {
        state_ = HandlerState::Global;
    } else if (state_ == HandlerState::PapersArray && key_.empty()) {
        state_ = HandlerState::Paper;
    } else if (state_ == HandlerState::ReviewArray && key_.empty()) {
        state_ = HandlerState::Review;
    } else {
        return false;
    }
    return true;
}
```

In the `end_object` event handler, we populate arrays of `Paper` and `Review` objects according to the current state. Also, we switch the current state back to the previous one by running the following code:

```
bool end_object() override {
    if (state_ == HandlerState::Global) {
        state_ = HandlerState::None;
    } else if (state_ == HandlerState::Paper) {
        state_ = HandlerState::PapersArray;
        papers_->push_back(paper_);
        paper_ = Paper();
    } else if (state_ == HandlerState::Review) {
        state_ = HandlerState::ReviewArray;
        paper_.reviews.push_back(review_);
    } else {
        return false;
    }
}
```

```
    }
    return true;
}
```

In the `start_array` event handler, we switch the current state to a new one according to the `current_state` value by running the following code:

```
bool start_array(std::size_t) override {
    if (state_ == HandlerState::Global && key_ == "paper") {
        state_ = HandlerState::PapersArray;
        key_.clear();
    } else if (state_ == HandlerState::Paper && key_ == "review") {
        state_ = HandlerState::ReviewArray;
        key_.clear();
    } else {
        return false;
    }
    return true;
}
```

In the `end_array` event handler, we switch the current state to the previous one based on our knowledge of the document structure by running the following code:

```
bool end_array() override {
    if (state_ == HandlerState::ReviewArray) {
        state_ = HandlerState::Paper;
    } else if (state_ == HandlerState::PapersArray) {
        state_ = HandlerState::Global;
    } else {
        return false;
    }
    return true;
}
```

The vital thing in this approach is to clear the current `key` value after object processing. This helps us to debug parsing errors, and we always have actual information about the currently processed entity.

For small files, using the DOM approach can be preferable because it leads to less code and cleaner algorithms.

Writing and reading HDF5 files with the HighFive library

HDF5 is a highly efficient file format for storing datasets and scientific values. The HighFive library provides a higher-level C++ interface for the C library provided by the HDF Group. In this example, we propose to look at its interface by transforming the dataset used in the previous section to HDF5 format.

The main concepts of the HDF5 format are groups and datasets. Each group can contain other groups and have attributes of different types. Also, each group can contain a set of dataset entries. Each dataset is a multidimensional array of values of the same type, which also can have attributes of different types.

Let's start with including the required headers, as follows:

```
#include <highfive/H5DataSet.hpp>
#include <highfive/H5DataSpace.hpp>
#include <highfive/H5File.hpp>
```

Then, we have to create a `file` object where we will write our dataset, as follows:

```
HighFive::File file(file_name,
    HighFive::File::ReadWrite | HighFive::File::Create |
    HighFive::File::Truncate);
```

After we have a `file` object, we can start creating groups. We define a group of papers that should hold all `paper` objects, as follows:

```
auto papers_group = file.createGroup("papers");
```

Then, we iterate through an array of papers (as shown in the previous section) and create a group for each `paper` object with two attributes: the numerical `id` attribute and the `preliminary_decision` attribute of the `string` type, as illustrated in the following code block:

```
for (const auto& paper : papers) {
    auto paper_group = papers_group.createGroup(
        "paper_" + std::to_string(paper.id));

    std::vector<uint32_t> id = {paper.id};
    auto id_attr = paper_group.createAttribute<uint32_t>(
        "id", HighFive::DataSpace::From(id));
    id_attr.write(id);

    auto dec_attr = paper_group.createAttribute<std::string>(
        "preliminary_decision",
        HighFive::DataSpace::From(paper.preliminary_decision));
    dec_attr.write(paper.preliminary_decision);
}
```

After we have created an attribute, we have to put in its value with the `write()` method. Notice that the `HighFive::DataSpace::From` function automatically detects the size of the attribute value. The size is the amount of memory required to hold the attribute's value. Then, for each `paper_group` object, we create a corresponding group of reviews, as follows:

```
auto reviews_group = paper_group.createGroup("reviews");
```

We insert into each `reviews_group` object a dataset of numerical values of `confidence`, `evaluation`, and `orientation` fields. For the dataset, we define `DataSpace` (the number of elements in the dataset) of size 3 and define a storage type as a 32-bit integer, as follows:

```
std::vector<size_t> dims = {3};
std::vector<int32_t> values(3);
for (const auto& r : paper.reviews) {
    auto dataset = reviews_group.createDataSet<int32_t>(
        std::to_string(r.id), HighFive::DataSpace(dims));
    values[0] = std::stoi(r.confidence);
    values[1] = std::stoi(r.evaluation);
    values[2] = std::stoi(r.orientation);
    dataset.write(values);
}
```

After we have created and initialized all objects, the Papers/Reviews dataset in HDF5 format is ready. When the `file` object leaves the scope, its destructor saves everything to the secondary storage.

Having the file in the HDF5 format, we can consider the `HighFive` library interface for file reading.

As the first step, we again create a `HighFive::File` object, but with attributes for reading, as follows:

```
HighFive::File file(file_name, HighFive::File::ReadOnly);
```

Then, we use the `getGroup()` method to get the top-level `papers_group` object, as follows:

```
auto papers_group = file.getGroup("papers");
```

The `getGroup` method allows us to get a specific group by its name, so it's a type of navigation through the HDF5 file structure.

We need to get a list of all nested objects in this group because we can access objects only by their names. We can do this by running the following code:

```
auto papers_names = papers_group.listObjectNames();
```

Using a loop, we iterate over all `papers_group` objects in the `papers_group` container, like this:

```
for (const auto& pname : papers_names) {
    auto paper_group = papers_group.getGroup(pname);
    ...
}
```

For each `paper` object, we read its attributes and the memory space required for the attribute value. Also, because each attribute can be multidimensional, we should take care of it and allocate an appropriate container, as follows:

```
std::vector<uint32_t> id;
paper_group.getAttribute("id").read(id);
std::cout << id[0];

std::string decision;
paper_group.getAttribute("preliminary_decision").read(decision);
std::cout << " " << decision << std::endl;
```

For reading datasets, we can use the same approach: get the `reviews` group, then get a list of dataset names, and, finally, read each dataset in a loop, as follows:

```
auto reviews_group = paper_group.getGroup("reviews");
auto reviews_names = reviews_group.listObjectNames();
std::vector<int32_t> values(2);

for (const auto& rname : reviews_names) {
    std::cout << "\t review: " << rname << std::endl;

    auto dataset = reviews_group.getDataSet(rname);
    auto selection = dataset.select({1}, {2});
    // or use just dataset.read method to get whole data

    selection.read(values);

    std::cout << "\t\t evaluation: " << values[0] << std::endl;
    std::cout << "\t\t orientation: " << values[1] << std::endl;
}
```

Notice that we use the `select()` method for the dataset, which allows us to read only a part of the dataset. We define this part with ranges given as arguments. There is the `read()` method in the `dataset` type to read a whole dataset at once.

Using these techniques, we can read and transform any HDF5 dataset. This file format allows us to work only with part of the required data and not to load the whole file to the memory. Also, because this is a binary format, its reading is more efficient than reading large text files. Other useful features of HDF5 are the following:

- Compression options for datasets and attributes, reducing storage space and transfer time.
- Parallelization of I/O operations, enabling multiple threads or processes to access the file simultaneously. This can greatly increase throughput and reduce processing time.

In this section, we saw how to load different file formats with data into C++ data structures provided by various C++ libraries. Especially we learned how to fill matrix and tensor objects that will be used in different ML algorithms. In the following section, we will see how to initialize the same data structures with values from regular C++ containers, which can be important when you implement your own data loader.

Initializing matrix and tensor objects from C++ data structures

There are a variety of file formats used for datasets, and not all of them might be supported by libraries. For using data from unsupported formats, we might need to write custom parsers. After we read values to regular C++ containers, we usually need to convert them into object types used in the ML framework we use. As an example, let's consider the case of reading matrix data from files into C++ objects.

Working with the Eigen library

Using the Eigen library, we can wrap a C++ array into an `Eigen::Matrix` object with the `Eigen::Map` type. The wrapped object will behave as a standard Eigen matrix. We have to parametrize the `Eigen::Map` type with the type of matrix that has the required behavior. Also, when we create the `Eigen::Map` object, it takes as arguments a pointer to the C++ array and matrix dimensions, as illustrated in the following code snippet:

```
std::vector<double> values;
...
auto x_data = Eigen::Map<Eigen::Matrix<double,
                                         Eigen::Dynamic,
                                         Eigen::Dynamic,
                                         Eigen::RowMajor>>(
    values.data(),
    rows_num,
    columns_num);
```

Working with the Blaze library

The `Blaze` library has special classes that can be used to create wrappers for C++ arrays. To wrap a C++ container with objects of these classes, we have to pass a pointer to the data and corresponding dimensions as arguments, as illustrated in the following code snippet:

```
std::array<int, 4> data = {1, 2, 3, 4};
blaze::CustomVector<int,
    blaze::unaligned,
    blaze::unpadded,
    blaze::rowMajor>
v2(data.data(), data.size());

std::vector<float> mdata = {1, 2, 3, 4, 5, 6, 7, 8, 9};
blaze::CustomMatrix<float,
    blaze::unaligned,
    blaze::unpadded,
    blaze::rowMajor>
a2(mdata.data(), 3UL, 3UL);
```

Notice that additional template parameters were used to specify memory layout, alignment, and padding.

Working with the Dlib library

The `Dlib` library has the `Dlib::mat()` function for wrapping C++ containers into the `Dlib` matrix object. It also takes a pointer to the data and matrix dimensions as arguments, as illustrated in the following code snippet:

```
double data[] = {1, 2, 3, 4, 5, 6};
auto m2 = Dlib::mat(data, 2, 3); // create matrix with size 2x3
```

The `Dlib::mat` function has other overloads that can take other types of containers to create a matrix.

Working with the ArrayFire library

The `ArrayFire` library has a single technique to initialize the array object with an external memory pointer. It can be used as follows:

```
float host_data[] = {0, 1, 2, 3, 4, 5};
array A(2, 3, host_data);
```

In this example, we initialize the 2×3 matrix with the data from the C array object. We used the `array` type constructor. The first two arguments are matrix row and column numbers, and the last one is the pointer to the data. We can initialize the `array` type object with the CUDA pointer in the same, but the fourth argument should be the `afDevice` specification.

Working with the mlpack library

The `mlpack` framework uses the `Armadillo` library for linear algebra objects. So, to wrap a C++ container into the `arma::mat` object, we can use the corresponding constructor that takes a pointer to the data and matrix dimensions, as illustrated in the following code snippet:

```
std::vector<double> values;
...
arma::mat(values.data(), n_rows, n_cols, /*copy_aux_mem*/ false);
```

If the fourth parameter named `copy_aux_mem` is set to `false`, the data will be not copied into the matrix's internal buffer.

Notice that all of these functions only make a wrapper for the original C++ array where the data is stored and don't copy the values into a new location. If we want to copy values from a C++ array to a `matrix` object, we usually need to call a `clone()` method or an analog of it for the wrapper object.

After we have a matrix object for an ML framework we use, we can initialize other specialized objects for training ML algorithms. Examples of such abstractions are the `f1::TensorDataset` class in the `Flashlight` library or the `torch::data::Dataset` class in the `libtorch` library.

In this section, we learned how to initialize matrix and tensor objects with regular C++ containers and pointers. The following section will move to another important topic: manipulation images.

Manipulating images with the OpenCV and Dlib libraries

Many ML algorithms are related to **computer vision (CV)** problems. Examples of such tasks are object detection in images, segmentation, image classification, and others. To be able to deal with such tasks, we need instruments for working with images. We usually need routines to load images to computer memory, as well as routines for image processing. For example, the standard operation is image scaling, because many ML algorithms are trained only on images of a specific size. This limitation follows from the algorithm structure or is a hardware requirement. For example, we cannot load large images to the **graphics processing unit (GPU)** memory because of its limited size.

Also, hardware requirements can lead to a limited range of numeric types that our hardware supports, so we will need to change the initial image representation to one that our hardware can efficiently process. Also, ML algorithms usually assume a predefined layout of image channels, which can be different from the layout in the original image file.

Another type of image-processing task is the creation of training datasets. In many cases, we have a limited number of available images for a specific task. However, to make a machine algorithm train well, we usually need more training images. So, the typical approach is to augment existing images. Augmentation can be done with operations such as random scaling, cropping parts of images, rotations, and other operations that can be used to make different images from the existing set.

In this section, we show how to use two of the most popular libraries for image processing for C++. OpenCV is a framework for solving CV problems that includes many ready-to-use implementations of CV algorithms. Also, it has many functions for image processing. Dlib is a CV and ML framework with a large number of implemented algorithms, as well as a rich set of image-processing routines.

Using OpenCV

In the OpenCV library, an image is treated as a multidimensional matrix of values. There is a special `cv::Mat` type for this purpose. There are two base functions: the `cv::imread()` function loads the image, and the `cv::imwrite()` function writes the image to a file, as illustrated in the following code snippet:

```
#include <opencv2/opencv.hpp>
...
cv::Mat img = cv::imread(file_name);
cv::imwrite(new_file_name, img);
```

Also, there are functions to manage images located in a memory buffer. The `cv::imdecode()` function loads an image from the memory buffer, and the `cv::imencode()` function writes an image to the memory buffer.

Scaling operations in the OpenCV library can be done with the `cv::resize()` function. This function takes an input image, an output image, the output image size or scale factors, and an interpolation type as arguments. The interpolation type governs how the output image will look after the scaling. General recommendations are as follows:

- Use `cv::INTER_AREA` for shrinking
- Use `cv::INTER_CUBIC` (slow) or `cv::INTER_LINEAR` for zooming
- Use `cv::INTER_LINEAR` for all resizing purposes because it is fast

The main difference between linear and cubic scaling lies in their approach to scaling pixels. Linear scaling preserves the aspect ratio and is simpler, while cubic scaling attempts to maintain details and transitions. The choice between the two depends on the specific requirements of your project and the desired outcome.

The following code sample shows how to scale an image:

```
cv::resize(img,
           img,
           {img.cols / 2, img.rows / 2},
           0,
           0,
           cv::INTER_AREA);
cv::resize(img, img, {}, 1.5, 1.5, cv::INTER_CUBIC);
```

There is no special function for image cropping in the OpenCV library, but the `cv::Mat` type overrides the `operator()` method, which takes a cropping rectangle as an argument and returns a new `cv::Mat` object with part of the image surrounded by the specified rectangle. Also, note that this object will share the same memory with the original image, so its modification will change the original image too. To make a deep copy of the `cv::Mat` object, we need to use the `clone()` method, as follows:

```
img = img(cv::Rect(0, 0, img.cols / 2, img.rows / 2));
```

Sometimes, we need to move or rotate an image. The OpenCV library supports translation and rotation operations for images through affine transformations. We have to manually—or with helper functions—create a matrix of 2D affine transformations and then apply it to our image. For the move (the translation), we can create such a matrix manually and then apply it to an image with the `cv::wrapAffine()` function, as follows:

```
cv::Mat trm = (cv::Mat<double>(2, 3) << 1, 0, -50, 0, 1, -50);
cv::wrapAffine(img, img, trm, {img.cols, img.rows});
```

We can create a rotation matrix with the `cv::getRotationMatrix2D()` function. This takes a point of origin and the rotation angle in degrees, as illustrated in the following code snippet:

```
auto rotm = cv::getRotationMatrix2D({img.cols / 2, img.rows / 2},
                                     45,
                                     1);
cv::wrapAffine(img, img, rotm, {img.cols, img.rows});
```

Another useful operation is extending an image size without scaling but with added borders. There is the `cv::copyMakeBorder()` function in the OpenCV library for this purpose. This function has different options on how to create borders. It takes an input image, an output image, border sizes for the top, the bottom, the left, and the right sides, the type of the border, and the border color. Border types can be one of the following:

- `BORDER_CONSTANT`: Make function fill borders with a single color
- `BORDER_REPLICATE`: Make function fill borders with copies of the last pixel values on each side (for example, `aaaaaa|abcdefg|hhhhhh`)

- **BORDER_REFLECT**: Make function fill borders with copies of opposite pixel values on each side (for example, `fedcba|abcdefgh|hgfedcb`)
- **BORDER_WRAP**: Make function fill borders by simulating the image duplication (for example, `cdefgh|abcdefgh|abcdefg`)

The following example shows how to use this function:

```
int top = 50; // px
int bottom = 20; // px
int left = 150; // px
int right = 5; // px
cv::copyMakeBorder(img, img, top, bottom, left, right,
                  cv::BORDER_CONSTANT | cv::BORDER_ISOLATED,
                  cv::Scalar(255, 0, 0));
```

When we are using this function, we should take care of the origin of the source image. The OpenCV documentation says: “*If the source image is a part of a bigger image, the function will try to use the pixels outside of the ROI (short for region of interest) to form a border. To disable this feature and always do extrapolation, as if the source image was not a part of another image, use border type BORDER_ISOLATED.*”

The function described previously is very helpful when we need to adapt training images of different sizes to the one standard image size used in some ML algorithms because, with this function, we do not distort target image content.

There is the `cv::cvtColor()` function to convert different color spaces in the OpenCV library. The function takes an input image, an output image, and a conversion scheme type. For example, in the following code sample, we convert the **red, green, and blue (RGB)** color space to a grayscale one:

```
cv::cvtColor(img, img, cv::COLOR_RGB2GRAY);
// now pixels values are in range 0-1
```

This can be very handy in certain scenarios.

Using Dlib

Dlib is another popular library for image processing. This library has different functions and classes for math routines and image processing. The library documentation recommends using the `Dlib::array2d` type for images. The `Dlib::array2d` type is a template type that has to be parametrized with a pixel type. Pixel types in the Dlib library are defined with pixel-type traits. There are the following predefined pixel types: `rgb_pixel`, `bgr_pixel`, `rgb_alpha_pixel`, `hspace_pixel`, `lab_pixel`, and any scalar type can be used for grayscaled pixels' representation.

We can use the `load_image()` function to load an image from disk, as follows:

```
#include <Dlib/image_io.h>
#include <Dlib/image_transforms.h>
using namespace Dlib;
...
array2d<rgb_pixel> img;
load_image(img, file_path);
```

For a scaling operation, there is the `Dlib::resize_image()` function. This function has two different overloads. One takes a single scale factor and a reference to an image object. The second one takes an input image, an output image, the desired size, and an interpolation type. To specify the interpolation type in the `Dlib` library, we should call special functions: the `interpolate_nearest_neighbor()`, the `interpolate_quadratic()`, and the `interpolate_bilinear()` functions. The criteria for choosing one of them are the same as for the ones that we discussed in the *Using OpenCV* section. Notice that the output image for the `resize_image()` function should be already preallocated, as illustrated in the following code snippet:

```
array2d<rgb_pixel> img2(img.nr() / 2, img.nc() / 2);
resize_image(img, img2, interpolate_nearest_neighbor());
resize_image(1.5, img); // default interpolate_bilinear
```

To crop an image with `Dlib`, we can use the `Dlib::extract_image_chips()` function. This function takes an original image, rectangle-defined bounds, and an output image. Also, there are overloads of this function that take an array of rectangle bounds and an array of output images, as follows:

```
extract_image_chip(
    img,
    rectangle(0, 0, img.nc() / 2, img.nr() / 2),
    img2);
```

The `Dlib` library supports image transformation operations through affine transformations. There is the `Dlib::transform_image()` function, which takes an input image, an output image, and an affine transformation object. An example of the transformation object could be an instance of the `Dlib::point_transform_affine` class, which defines the affine transformation with a rotation matrix and a translation vector. Also, the `Dlib::transform_image()` function can take an interpolation type as the last parameter, as illustrated in the following code snippet:

```
transform_image(img, img2, interpolate_bilinear(),
               point_transform_affine(
                   identity_matrix<double>(2),
                   Dlib::vector<double, 2>(-50, -50)));
```

In case we only need to do a rotation, Dlib has the `Dlib::rotate_image()` function. The `Dlib::rotate_image()` function takes an input image, an output image, a rotation angle in degrees, and an interpolation type, as follows:

```
rotate_image(img, img2, -45, interpolate_bilinear());
```

There is no complete analog of a function for adding borders to images in the Dlib library. There are two functions: `Dlib::assign_border_pixels()` and `Dlib::zero_border_pixels()` for filling image borders with specified values. Before using these routines, we should resize the image and place the content in the right position. The new image size should include the borders' widths. We can use the `Dlib::transform_image()` function to move the image content into the right place. The following code sample shows how to add borders to an image:

```
int top = 50; // px
int bottom = 20; // px
int left = 150; // px
int right = 5; // px
img2.set_size(img.nr() + top + bottom, img.nc() + left + right);
transform_image(
    img, img2, interpolate_bilinear(),
    point_transform_affine(
        identity_matrix<double>(2),
        Dlib::vector<double, 2>(-left/2, -top/2)
    )
);
```

For color-space conversions, there exists the `Dlib::assign_image()` function in the Dlib library. This function uses color-type information from pixel-type traits we used for the image definition. So, to convert an image to another color space, we should define a new image with the desired type of pixels and pass it to this function. The following example shows how to convert the RGB image to a **blue, green, red (BGR)** one:

```
array2d<bgr_pixel> img_bgr;
assign_image(img_bgr, img);
```

To make a grayscale image, we can define an image with the `unsigned char` pixel type, as follows:

```
array2d<unsigned char> img_gray;
assign_image(img_gray, img);
```

In this section, we learned how to load and preprocess images with the OpenCV and Dlib libraries. The next important step is to convert images into matrix or tensor structures to be able to use them in ML algorithms; it will be described in the following section.

Transforming images into matrix or tensor objects of various libraries

In most cases, images are represented in computer memory in an interleaved format, which means that pixel values are placed one by one in linear order. Each pixel value consists of several numbers representing a color. For example, for the RGB format, there will be three values placed together. So, in the memory, we will see the following layout for a 4x4 image:

```
rgb rgb rgb rgb
rgb rgb rgb rgb
rgb rgb rgb rgb
rgb rgb rgb rgb
```

For image-processing libraries, such a value layout is not a problem, but many ML algorithms require different ordering. For example, it's a common approach for **neural networks (NNs)** to take image channels separately ordered, one by one. The following example shows how such a layout is usually placed in memory:

```
r r r r r g g g g b b b b
r r r r r g g g g b b b b
r r r r r g g g g b b b b
r r r r r g g g g b b b b
```

So, often, we need to deinterleave image representation before passing it to some ML algorithm. It means that we need to extract color channels into separate vectors.

Moreover, we usually need to convert a color's value data type too. For example, OpenCV library users often use floating-point formats, which allows them to preserve more color information in image transformations and processing routines. The opposite case is when we use a 256-bit type for color-channel information, but then we need to convert it to a floating-point type. So, in many cases, we need to convert the underlying data type to another one more suitable for our needs.

Deinterleaving in OpenCV

By default, when we load an image with the OpenCV library, it loads the image in the BGR format and with `char` as the underlying data type. So, we need to convert it to the RGB format, like this:

```
cv::cvtColor(img, img, cv::COLOR_BGR2RGB);
```

Then, we can convert the underlying data type to the `float` type, like this:

```
img.convertTo(img, CV_32FC3, 1/255.0);
```

Next, to deinterleave channels, we need to split them with the `cv::split()` function, like this:

```
cv::Mat bgr[3];
cv::split(img, bgr);
```

Then, we can place channels back to the `cv::Mat` object in the order we need with the `cv::vconcat()` function, which concatenates matrices vertically, as follows:

```
cv::Mat ordered_channels;
cv::vconcat(bgr[2], bgr[1], ordered_channels);
cv::vconcat(ordered_channels, bgr[0], ordered_channels);
```

There is a useful method in the `cv::Mat` type named `isContinuous` that allows us to check if the matrix's data is placed in memory with a single contiguous block. If that is `true`, we can copy this block of memory or pass it to routines that work with plain C arrays.

Deinterleaving in Dlib

The `Dlib` library uses the `unsigned char` type for pixel color representation, and we can use floating-point types only for grayscaled images. The `Dlib` library stores pixels in row-major order with interleaved channels, and data is placed in memory continuously with a single block. There are no special functions in the `Dlib` library to manage image channels, so we cannot deinterleave them or mix them. However, we can use raw pixel data to manage color values manually. Two functions in the `Dlib` library can help us: the `image_data()` function to access raw pixel data, and the `width_step()` function to get the padding value.

The most straightforward approach to deinterleave the `Dlib` image object is using a loop over all pixels. In such a loop, we can split each pixel value into separate colors.

As a first step, we define containers for each of the channels, as follows:

```
auto channel_size = static_cast<size_t>(img.nc() * img.nr());
std::vector<unsigned char> ch1(channel_size);
std::vector<unsigned char> ch2(channel_size);
std::vector<unsigned char> ch3(channel_size);
```

Then, we read color values for each pixel with two nested loops over image rows and columns, like this:

```
size_t i{0};
for (long r = 0; r < img.nr(); ++r) {
    for (long c = 0; c < img.nc(); ++c) {
        ch1[i] = img[r][c].red;
        ch2[i] = img[r][c].green;
        ch3[i] = img[r][c].blue;
        ++i;
    }
}
```

```
    }  
}
```

The result is three containers with color-channel values, which we can use separately. They are suitable to initialize grayscaled images for use in image-processing routines. Alternatively, we can use them to initialize a matrix-type object that we can process with linear algebra routines.

We saw how to load and prepare images for use in linear algebra abstractions and ML algorithms. In the next section, we will learn general methods to prepare data for use in ML algorithms. Such methods will help us make learning procedures more stable and converge faster.

Normalizing data

Data normalization is a crucial preprocessing step in ML. In general, data normalization is a process that transforms multiscale data to the same scale. Feature values in a dataset can have very different scales—for example, the height can be given in centimeters with small values, but the income can have large-value amounts. This fact has a significant impact on many ML algorithms.

For example, if some feature values differ from values of other features several times, then this feature will dominate over others in classification algorithms based on the Euclidean distance. Some algorithms have a strong requirement for normalization of input data; an example of such an algorithm is the **Support Vector Machine (SVM)** algorithm. NNs also usually require normalized input data. Also, data normalization has an impact on optimization algorithms. For example, optimizers based on the **gradient descent (GD)** approach can converge much quicker if data has the same scale.

There are several methods of normalization, but from our point of view, the most popular are the standardization, the min-max, and the mean normalization methods.

Standardization is a process of making data have a zero mean and a standard deviation equal to 1. The formula for standardized vector is $x' = \frac{x - \bar{x}}{\sigma}$, where x is an original vector, \bar{x} is an average value of x calculated with the formula $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$, and σ is the standard deviation of x calculated with the formula $\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}}$.

Min-max normalization or rescaling is a process of making data fit the range of $[0, 1]$. We can do rescaling with the following formula:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Min-max scaling is useful when there are significant differences in the scale of different features in your dataset. It helps to make the features comparable, which is important for many ML models.

Mean normalization is used to fit data into the range $[-1, 1]$ so that its mean becomes zero. We can use the following formula to do mean normalization:

$$x' = \frac{x - \bar{x}}{\max(x) - \min(x)}$$

This transformation helps to make the data more easily interpretable and improves the performance of some ML algorithms by reducing the impact of outliers and ensuring that all features are on a similar scale. Consider how we can implement these normalization techniques and which ML framework functions can be used to calculate them.

We assume that each row of this matrix $x = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}$ is one training sample, and the value in each column is the value of one feature of the current sample.

Normalizing with Eigen

There are no functions for data normalization in the `Eigen` library. However, we can implement them according to the provided formulas.

For standardization, we first have to calculate the standard deviation, as follows:

```
Eigen::Array<double, 1, Eigen::Dynamic> std_dev =
  ((x.rowwise() - x.colwise().mean())
   .array()
   .square()
   .colwise()
   .sum() /
  (x_data.rows() - 1))
  .sqrt();
```

Notice that some reduction functions in the `Eigen` library work only with array representation; examples are the `sum()` and the `sqrt()` functions. We have also calculated the mean for each feature—we used the `x.colwise().mean()` function combination, which returns a vector of mean. We can use the same approach for other feature statistics' calculations.

Having the standard deviation value, the rest of the formula for standardization will look like this:

```
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> x_std =
  (x.rowwise() - x.colwise().mean()).array().rowwise() / std_dev;
```

Implementation of min-max normalization is very straightforward and does not require intermediate values, as illustrated in the following code snippet:

```
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> x_min_max =
  (x.rowwise() - x.colwise().minCoeff()).array().rowwise() /
  (x.colwise().maxCoeff() - x.colwise().minCoeff()).array();
```

We implement the mean normalization in the same way, like this:

```
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> x_avg =
(x.rowwise() - x.colwise().mean()).array().rowwise() /
(x.colwise().maxCoeff() - x.colwise().minCoeff()).array();
```

Notice that we implement formulas in a vectorized way without loops; this approach is more computationally efficient because it can be compiled for execution on a GPU or the **central processing unit's (CPU's) Single Instruction Multiple Data (SIMD)** instructions.

Normalizing with mlpack

There are different classes for feature scaling in the mlpack library. The most interesting for us are `data::data::MinMaxScaler`, which implements min-max normalization (or rescaling), and `mlpack::data::StandardScaler`, which implements data standardization. We can reuse objects of those classes for scaling different data with the same learned statistics. It can be useful in cases when we train an ML algorithm on one data format with applied rescaling, and then we use the algorithm for predictions on new data. To make this algorithm work as we want, we have to rescale new data in the same way as we did in the training process, as follows:

```
#include <mlpack/core.hpp>
...
arma::mat features;
arma::Row<size_t> labels;
data::MinMaxScaler min_max_scaler;
min_max_scaler.Fit(features); // learn statistics

arma::mat scaled_dataset;
min_max_scaler.Transform(features, scaled_dataset);
```

To learn statistics values, we use the `Fit()` method, and for feature modification, we use the `Transform()` method of the `MinMaxScaler` class.

The `StandardScaler` class can be used in the same manner, as follows:

```
data::StandardScaler standard_scaler;
standard_scaler.Fit(features);
standard_scaler.Transform(features, scaled_dataset);
```

To print the matrix object in the mlpack library, the standard streaming operators can be used as follows:

```
std::cout << scaled_dataset << std::endl;
```

Also, to revert the applied scaling, these classes have the `InverseTransform` method.

Normalizing with Dlib

The `Dlib` library provides functionality for feature standardization with the `Dlib::vector_normalizer` class. There is one limitation to using this class—we cannot use it with one big matrix containing all training samples. Alternatively, we should represent each sample with a separate vector object and put them into the C++ `std::vector` container, as follows:

```
std::vector<matrix<double>> samples;
...
vector_normalizer<matrix<double>> normalizer;
samples normalizer.train(samples);
samples = normalizer(samples);
```

We see that the object of this class can be reused, but it should be trained first. The `train` method implementation can look like this:

```
matrix<double> m(mean(mat(samples)));
matrix<double> sd(reciprocal(stddev(mat(samples))));
for (size_t i = 0; i < samples.size(); ++i)
    samples[i] = pointwise_multiply(samples[i] - m, sd);
```

Notice that the `Dlib::mat()` function has different overloads for matrix creation from different sources. Also, we use the `reciprocal()` function that makes the $m' = \frac{1}{m}$ matrix if m is the input matrix.

Printing matrices for debugging purposes in the `Dlib` library can be done with the simple streaming operator, as illustrated in the following code snippet:

```
std::cout << mat(samples) << std::endl;
```

We can see that the `Dlib` library provides a rich interface for data preprocessing that can be easily used.

Normalizing with Flashlight

The `Flashlight` library doesn't have particular classes to perform feature scaling. But it has functions to calculate basic statistics easily, so we can implement feature scaling algorithms as follows:

```
fl::Tensor x;
...
// min-max scaling
auto x_min = fl::amin(x, {1});
auto x_max = fl::amax(x, {1});
auto x_min_max = (x - x_min) / (x_max - x_min);

// normalization(z-score)
```

```
auto x_mean = fl::mean(x, {1});  
auto x_std = fl::std(x, {1});  
auto x_norm = (x - x_mean) / x_std;
```

The `fl::amin` and `fl::amax` functions find the minimum and maximum values. The `fl::mean` and `fl::std` functions calculate the mean and standard deviation correspondingly. All these functions do their calculation along a specified dimension that comes as the second parameter. It means that we scale each `x` feature in the `dataset` separately.

We can print a `fl::Tensor` object with the standard C++ streaming operator, as follows:

```
std::cout << dataset << std::endl;
```

We saw that despite the FlashLight library not providing special classes for data preprocessing, we can build them with linear algebra routines.

Summary

In this chapter, we considered how to load data from CSV, JSON, and HDF5 formats. CSV is easy to read and write, making it suitable for small to medium-sized datasets. CSV files are often used for tabular data, such as customer information, sales records, or financial transactions. JSON is a lightweight data interchange format that is human-readable and easy to parse. It is commonly used for representing structured data, including objects, arrays, and key-value pairs. In ML, JSON can be used to store data for training models, such as feature vectors, labels, and metadata. HDF5 is a high-performance file format designed for scientific data storage and analysis. It supports large datasets with complex structures, allowing for efficient storage of multidimensional arrays and tables. HDF5 files are commonly used in applications where large amounts of data need to be stored and accessed efficiently.

We saw how to convert the loaded data into objects suitable for use in different ML frameworks. We used the libraries' APIs to convert raw C++ arrays into matrices and higher-level dataset objects for ML algorithms.

We looked at how to load and process images with the OpenCV and Dlib libraries. These libraries offer a wide range of functions and algorithms that can be used in various applications for CV. The libraries can be used for basic image preprocessing, as well as for more complicated systems that use ML for solving industry-important tasks such as face detection and recognition, which can be used to build security systems, and for access control or facial authentication. Object detection can be used for tasks such as counting objects in an image, detecting defects in products, identifying specific objects, or tracking their movement. This is useful in industrial automation, surveillance systems to identify suspicious activities, and autonomous vehicles. Image segmentation allows users to extract specific parts of an image for further analysis. This is essential for diagnosing diseases in medical imaging analysis. Motion tracking of objects over time is also used for sports analytics, traffic monitoring, and surveillance.

We became familiar with the data normalization process, which is very important for many ML algorithms. Also, we saw which normalization techniques are available in ML libraries, and we implemented some normalization approaches with linear algebra functions from the Eigen library.

In the following chapter, we will see how to measure a model's performance on different types of data. We will look at special techniques that help us to understand how the model describes the training dataset well and how it performs on new data. Also, we will learn the different types of parameters ML models depend on and see how to select the best combination of them to improve the model's performance.

Further reading

- The HDF5® library and file format: <https://www.hdfgroup.org/solutions/hdf5/>
- GitHub link for Fast-CPPCSV Parser: <https://github.com/ben-strasser/Fast-CPP-CSV-Parser>
- OpenCV: <https://opencv.org/>
- Dlib C++ library: <http://Dlib.net/>
- Flashlight documentation: <https://fl.readthedocs.io/en/latest/index.html>
- nlohmann-json documentation: <https://json.nlohmann.me/>
- mlpack documentation: <https://mlpack.org/doc/index.html>
- *A Hybrid Approach for Sentiment Analysis Applied to Paper Reviews* dataset: <https://archive.ics.uci.edu/static/public/410/paper+reviews.zip>

3

Measuring Performance and Selecting Models

This chapter describes bias and variance effects and their pathological cases, which usually appear when training **machine learning (ML)** models.

In this chapter, we will learn how to deal with overfitting by using regularization and discuss different techniques we can use. We will also consider different model performance estimation metrics and how they can be used to detect training problems. Toward the end of this chapter, we will look at how to find the best hyperparameters for a model by introducing the grid search technique and its implementation in C++.

The following topics will be covered in this chapter:

- Performance metrics for ML models
- Understanding bias and variance characteristics
- Model selection with the grid search technique

Technical requirements

For this chapter, you will need the following:

- A modern C++ compiler with C++20 support
- CMake build system version ≥ 3.10
- `Dlib` library
- `mlpack` library
- `Flashlight` library
- `Plotcpp` library

The code files for this chapter can be found in the following GitHub repo: <https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition/tree/main/Chapter03>

Performance metrics for ML models

When we develop or implement a particular ML algorithm, we need to estimate how well it works. In other words, we need to estimate how well it solves our task. Usually, we use some numeric metrics for algorithm performance estimation. An example of such a metric could be a value of **mean squared error (MSE)** that's been calculated for target and predicted values. We can use this value to estimate how distant our predictions are from the target values we used for training. Another use case for performance metrics is their use as objective functions in optimization processes. Some performance metrics are used for manual observations, though others can be used for optimization purposes too.

Performance metrics are different for each of the ML algorithm types. In *Chapter 1, Introduction to Machine Learning with C++*, we discussed that two main categories of ML algorithms exist: **regression algorithms** and **classification algorithms**. There are other types of algorithms in the ML discipline, but these two are the most common ones. This section will go over the most popular performance metrics for regression and classification algorithms.

Regression metrics

Regression task metrics are used to measure how close predicted values are to ground truth ones. Such measurements can help us estimate the prediction quality of the algorithm. Under regression metrics, there are four main metrics, which we will dive into in the following subsections.

MSE and RMSE

MSE is a widely used metric for regression algorithms to estimate their quality. It is an average squared difference between the predictions and ground truth values. This is given by the following equation:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Here, n is the number of predictions and ground truth items, y_i is the ground truth value for the i^{th} item, and \hat{y}_i is the prediction value for the i^{th} item.

MSE is often used as a target loss function for optimization algorithms because it is smoothly differentiable and is a convex function.

The **root mean squared error (RMSE)** metric is usually used to estimate performance, such as when we need to give bigger weights to higher errors (to penalize them). We can interpret this as the standard deviation of the differences between predictions and ground truth values. This is given by the following equation:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

To calculate MSE with the `Dlib` library, there exists the `mean_squared_error` function, which takes two floating-point vectors and returns the MSE.

The `mlpack` library provides the `MeanSquaredError` class with the static `Evaluate` function that runs an algorithm prediction and calculates the MSE.

The `Flashlight` library also has the `MeanSquaredError` class; objects to this class can be used as a loss function so that it has forward and backward functions. Also, this library has the `MSEMeter` class that measures the MSE between targets and predictions and can be used for performance tracking.

Mean absolute error

Mean absolute error (MAE) is another popular metric that's used for quality estimation for regression algorithms. The MAE metric is a linear function with equally weighted prediction errors. It means that it does not take into account the direction of errors, which can be problematic in some cases. For example, if a model consistently underestimates or overestimates the true value, the MAE will still give a low score, even though the model may not be performing well. But this metric is more robust for outliers than RMSE. It is given by the following equation:

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

We can use the `MeanSquaredError` class in the `Flashlight` library to calculate this type of error. The `MeanSquaredError` class implements the loss functionality so that it has the forward/backward functions. Unfortunately, there is no specific functionality for the MAE calculation in the `Dlib` and `mlpack` libraries, but it can be easily implemented with their linear algebra backends.

R-squared

The R-squared metric is also known as a **coefficient of determination**. It is used to measure how well our independent variables (features from the training set) describe the problem and explain the variability of dependent variables (prediction values). Higher values tell us that the model explains our data well enough, while lower values tell us that the model makes many errors. This is given by the following equations:

$$\begin{aligned}SS_{\text{tot}} &= \sum_i (y_i - \bar{y})^2 \\SS_{\text{res}} &= \sum_i (y_i - \hat{y}_i)^2 \\R^2 &= 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} = 1 - \frac{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}{\frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2}\end{aligned}$$

Here, n is the number of predictions and ground truth items, y_i is the ground truth value for the i^{th} item, and \hat{y}_i is the prediction value for the i^{th} item.

The only problem with this metric is that adding new independent variables may increase R-squared in some cases, so it's crucial to consider the quality and relevance of these variables and to avoid overfitting the model. It may seem that the model begins to explain data better, but this isn't true—this value only increases if there are more training items.

There are no out-of-the-box functions for calculating this metric in the `Flashlight` library; however, it is simple to implement it with linear algebra functions.

There is the `r_squared` function in the `Dlib` library for computing the R-squared coefficient between matching elements of two `std::vector` instances.

The `mlpack` library has the `R2Score` class, which has the static `Evaluate` function that runs prediction for the specified algorithm and calculates the R-squared error.

Adjusted R-squared

The adjusted R-squared metric was designed to solve the previously described problem of the R-squared metric. It is the same as the R-squared metric but with a penalty for a large number of independent variables. The main idea is that if new independent variables improve the model's quality, the values of this metric increase; otherwise, they decrease. This can be given by the following equation:

$$R^2_{\text{adj}} = 1 - \frac{SS_{\text{res}}/(n-k)}{SS_{\text{tot}}/(n-1)} = 1 - (1-R^2) \frac{k-1}{k-n-1}$$

Here, k is the number of parameters and n is the number of samples.

Classification metrics

Before we start discussing classification metrics, we have to introduce an important concept called the **confusion matrix**. Let's assume that we have two classes and an algorithm that assigns them to an object. Here, the confusion matrix will look like this:

	$y = 1$	$y = 0$
$\hat{y} = 1$	True positive (TP)	False positive (FP)
$\hat{y} = 0$	False negative (FN)	True negative (TN)

Here, \hat{y} is the predicted class of the object and y is the ground truth label. The confusion matrix is an abstraction that we use to calculate different classification metrics. It gives us the number of items that were classified correctly and misclassified. It also provides us with information about the misclassification type. The false negatives are items that our algorithm incorrectly classified as negative ones, while the false positives are items that our algorithm incorrectly classified as positive ones. In this section, we'll learn how to use this matrix and calculate different classification performance metrics.

A confusion matrix can be created in the `mlpack` library with the `ConfusionMatrix` function; this function works only for discrete data/categorical data.

The `Dlib` library also has instruments to get a confusion matrix. There is the `test_multiclass_decision` function that tests a multi-class decision function and returns a confusion matrix describing the results. It also has the `test_sequence_labeler` class that tests the `labeler` object against the given samples and labels and returns a confusion matrix summarizing the results.

Accuracy

One of the most obvious classification metrics is accuracy:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

This provides us with a ratio of all positive predictions to all others. In general, this metric is not very useful because it doesn't show us the real picture in terms of cases with an odd number of classes. Let's consider a spam classification task and assume we have 10 spam letters and 100 non-spam letters. Our algorithm predicted 90 of them correctly as non-spam and classified only 5 spam letters correctly. In this case, accuracy will have the following value:

$$\text{accuracy} = \frac{5 + 90}{5 + 90 + 10 + 5} = 86.4$$

However, if the algorithm predicts all letters as non-spam, then its accuracy should be as follows:

$$\text{accuracy} = \frac{0 + 100}{0 + 100 + 0 + 10} = 90.9$$

This example shows that our model still doesn't work because it is unable to predict all spam letters, but the accuracy value is good enough.

To calculate accuracy in the `Flashlight` library, we use the `FrameErrorMeter` class that can estimate the accuracy or the error rate depending on user settings.

The `mlpack` library has the `Accuracy` class with the static `Evaluate` function that runs classification for the specified algorithm and calculates the accuracy value.

Unfortunately, the `Dlib` library doesn't have functions to calculate accuracy values, so if needed, the function should be implemented with the linear algebra backend.

Precision and recall

To estimate algorithm quality for each classification class, we will introduce two metrics: **precision** and **recall**. The following diagram shows all objects that are used in classification and how they have been marked according to the algorithm's results:

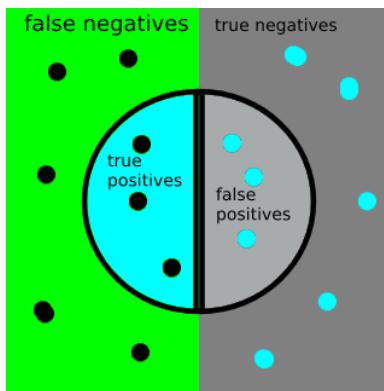


Figure 3.1 – Precision and recall

The circle in the center contains *selected elements*—the elements our algorithm predicted as positive ones.

Precision is proportional to the number of correctly classified items within selected ones that are defined as follows:

$$\text{precision} = \frac{TP}{TP + FP}$$

Recall is proportional to the number of correctly classified items within all ground truth positive items that are defined as follows:

$$\text{recall} = \frac{TP}{TP + FN}$$

Another name for recall is **sensitivity**. Let's assume that we are interested in the detection of positive items—let's call them relevant ones. So, we use the recall value as a measure of an algorithm's ability to detect relevant items and the precision value as a measure of an algorithm's ability to see the differences between classes. These measures do not depend on the number of objects in each of the classes, and we can use them for imbalanced dataset classification.

There are several approaches to calculate these metrics in the Dlib library. There is the `average_precision` function that can be used to calculate the precision value directly. Also, there is the `test_ranking_function` function that tests the given ranking function on the provided data and can return the mean average precision. Another way is to use the `test_sequence_segmenter` class that tests a `segmenter` object against the given samples and returns the precision, recall, and

F1-score, where `sequence_segmenter` is an object for segmenting a sequence of objects into a set of non-overlapping chunks.

The `mlpack` library has two classes—`Precision` and `Recall`—with static `Evaluate` functions that run classification for the specified algorithm and calculate precision and recall correspondingly.

The `Flashlight` library doesn't have the functionality to calculate these values.

F-score

In many cases, it is useful to have only one metric that shows the classification's quality. For example, it makes sense to use some algorithms to search for the best hyperparameters, such as the grid search algorithm, which will be discussed later in this chapter. Such algorithms usually use one metric to compare different classification results after applying various parameter values during the search process. One of the most popular metrics for this case is the F-measure (or the F-score), which can be given as follows:

$$F_{\beta} = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}$$

Here, β is the precision metric weight. Usually, the β value is equal to 1. In such a case, we have the multiplier value equal to 2, which gives us $F_1 = 1$ if the `precision` = 1 and the `recall` = 1. In other cases, when the precision value or the recall value tends to be zero, the F-measure value will also decrease.

The `Dlib` library provides the F1-score calculation only in the scope of the `test_sequence_segmenter` class functionality, which tests a `segmenter` object against the given samples and returns the precision, recall, and F1-score.

There is the `F1` class in the `mlpack` library that has the `Evaluate` function, which can be used to calculate the F1-score value by running a classification with the specified algorithm and data.

The `Flashlight` library doesn't have the functionality to calculate F-score values.

AUC-ROC

Usually, a classification algorithm will not return a concrete class identifier but a probability of an object belonging to some class. So, we usually use a threshold to decide whether an object belongs to a class or not. The most apparent threshold is 0.5, but it can work incorrectly in the case of imbalanced data (when we have a lot of values for one class and significantly fewer for another class).

One of the methods we can use to estimate a model without the actual threshold is the value of the **Area Under the Receiver Operating Characteristic Curve (AUC-ROC)**. This curve is a line from (0,0) to (1,1) in coordinates of the **True Positive Rate (TPR)** and the **False Positive Rate (FPR)**:

$$TPR = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{FP + TN}$$

The TPR value is equal to the recall, while the FPR value is proportional to the number of objects of the negative class that were classified incorrectly (they should be positive). In an ideal case, when there are no classification errors, we have $FPR = 0$, $TPR = 1$, and the area under the **receiver operating characteristic (ROC)** curve will be equal to 1. In the case of random predictions, the area under the ROC curve will be equal to 0.5 because we will have an equal number of TP and FP classifications:

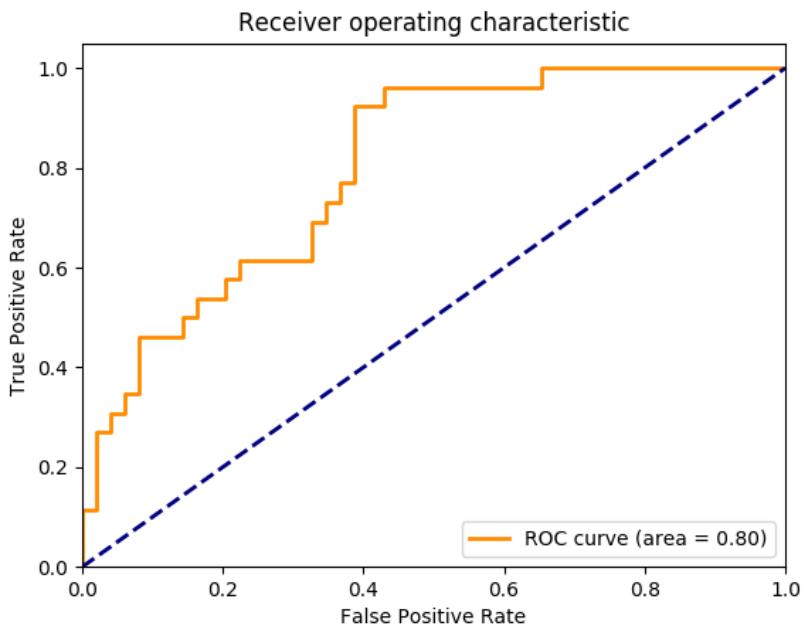


Figure 3.2 – ROC

Each point on the curve corresponds to some threshold value. Notice that the curve's steepness is an essential characteristic because we want to minimize the FPR, so we usually want this curve to tend to point (0,1). We can also successfully use the AUC-ROC metric with imbalanced datasets.

There is the `compute_roc_curve` function in the `Dlib` library that computes the ROC curve of the given data.

Unfortunately, the `Flashlight` and `mlpack` libraries don't have the functionality to compute the AUC-ROC metric.

Log-Loss

The logistic loss function value (the Log-Loss) is used as a target loss function for optimization purposes. It is given by the following equation:

$$\text{logloss} = -\frac{1}{n} \cdot \sum_{i=1}^l (y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i))$$

We can understand the Log-Loss value as the accuracy being corrected but with penalties for incorrect predictions. This function gives significant penalties, even for single misclassified objects, so all outlier objects in the data should be processed separately or removed from the dataset.

There is the `loss_binary_log` class in the `Dlib` library that implements the Log-Loss, which is appropriate for binary classification problems. This class is designed to be used as a **neural network (NN)** module.

The `Flashlight` library has the `BinaryCrossEntropy` class that computes the binary cross-entropy loss between an input tensor `x` and a target tensor `y`. Also, the main aim of this class is the loss function implementation for NN training.

The `CrossEntropyError` class in the `mlpack` library also represents a loss function for NN construction; it has forward and backward functions. So, it is used to measure the network's performance according to the cross-entropy between the input and target distributions.

In the current section, we learned about performance estimation metrics that can give you a clearer picture of your model accuracy, precision, and other performance characteristics. In the following section, we will learn about bias and variance and how to estimate and fix model prediction characteristics.

Understanding bias and variance characteristics

Bias and variance characteristics are used to predict model behavior. For example, the high variance effect, also known as **overfitting**, is a phenomenon in ML where the constructed model explains the examples from the training set but works relatively poorly on the examples that did not participate in the training process. This occurs because while training a model, random patterns will start appearing that are normally absent from the general population. The opposite of overfitting is known as **underfitting**, which corresponds to the high bias effect. This happens when the trained model becomes unable to predict patterns in new data or even in the training data. Such an effect can be the result of a limited training dataset or weak model design.

Before we go any further and describe what they mean, we should consider **validation**. Validation is a technique that's used to test model performance. It estimates how well the model makes predictions on new data. New data is data that we did not use for the training process. To perform validation, we usually divide our initial dataset into two or three parts. One part should contain most of the data and will be used for training, while the other ones will be used to validate and test the model. Usually,

validation is performed for iterative algorithms after one training cycle (often called an **epoch**). Alternatively, we perform testing after the overall training process.

Validation and testing operations evaluate the model on the data we have excluded from the training process, which results in the values of the performance metrics that we chose for this particular model. For example, the original dataset can be divided into the following parts: 80% for training, 10% for validation, and 10% for testing. The values of these validation metrics can be used to estimate model and prediction error trends. The most crucial issue for validation and testing is that the data for them should always be from the same distribution as the training data.

Throughout the rest of this chapter, we will use the polynomial regression model to show different prediction behaviors. The polynomial degree will be used as a hyperparameter.

Bias

Bias is a prediction characteristic that tells us about the distance between model predictions and ground truth values. Usually, we use the term *high bias* or *underfitting* to say that model prediction is too far from the ground truth values, which means that the model generalization ability is weak. Consider the following graph:

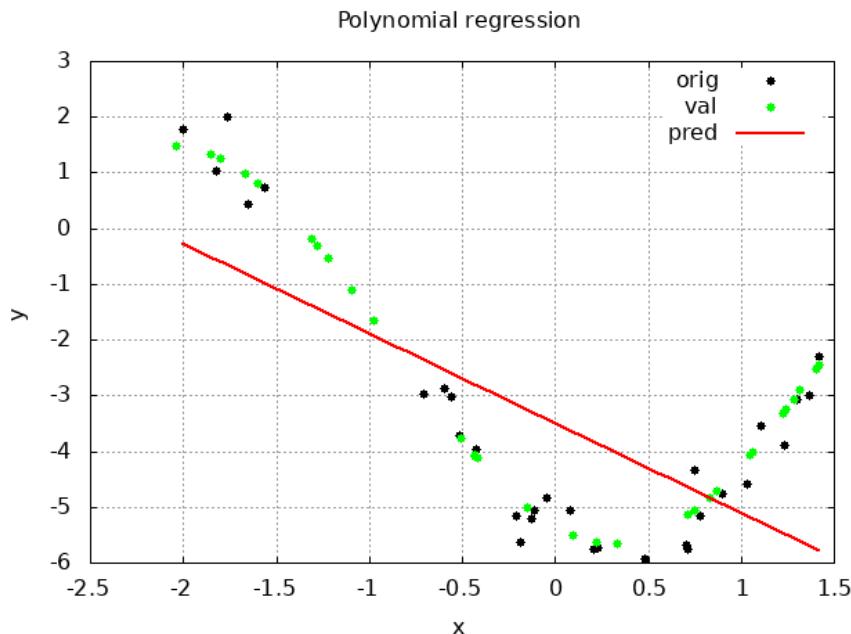


Figure 3.3 – Regression model predictions with the polynomial degree equal to 1

This graph shows the original values, the values used for validation, and a line that represents the polynomial regression model output. In this case, the polynomial degree is equal to 1. We can see that the predicted values do not describe the original data at all, so we can say that this model has a high bias. Also, we can plot validation metrics for each training cycle to get more information about the training process and the model's behavior.

The following graph shows the MAE metric values for the training process of the polynomial regression model, where the polynomial degree is equal to 1:

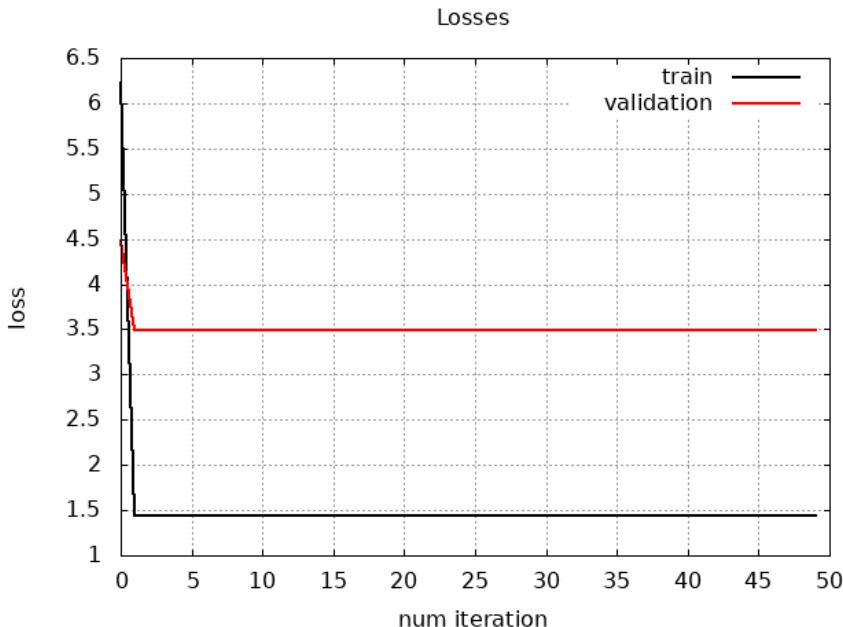


Figure 3.4 – Train and validation loss values

We can see that the lines for the metric values for the train and validation data are parallel and distant enough. Moreover, these lines do not change their direction after numerous training iterations. These facts also tell us that the model has a high bias because, for a regular training process, validation metric values should be close to the training values.

To deal with high bias, we can add more features to the training samples. For example, increasing the polynomial degree for the polynomial regression model adds more features; these all-new features describe the original training sample because each additional polynomial term is based on the original sample value.

Variance

Variance is a prediction characteristic that tells us about the variability of model predictions; in other words, how big the range of output values can be. Usually, we use the term *high variance* or *overfitting* in the case when a model tries to incorporate many training samples very precisely. In such a case, the model cannot provide a good approximation for new data but has excellent performance on the training data.

The following graph shows the behavior of the polynomial regression model, with the polynomial degree equal to 15:

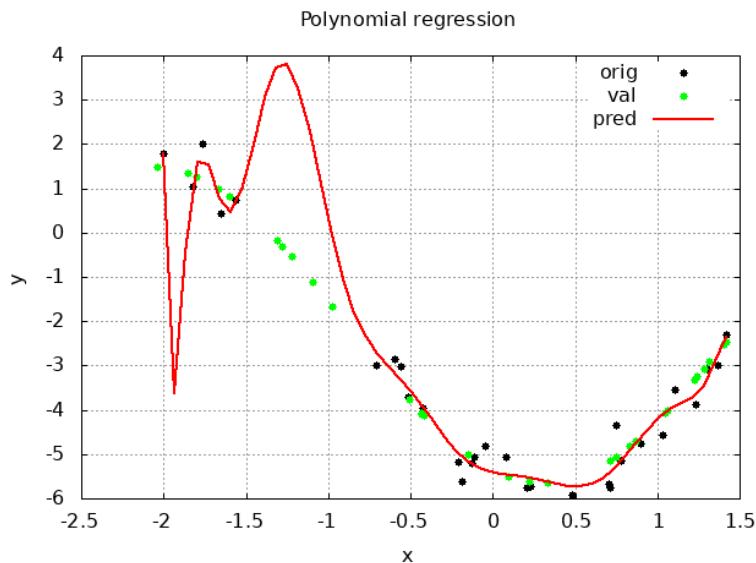


Figure 3.5 – Regression model predictions with the polynomial degree equal to 15

We can see that the model incorporates almost all the training data. Notice that the training data is indicated as `orig` in the plot's legend, while the data used for validation is indicated as `val` in the plot's legend. We can see that these two sets of data—training data and validation data—are somehow distant from each other and that our model misses the validation data because of a lack of approximation. The following graph shows the MAE values for the learning process:

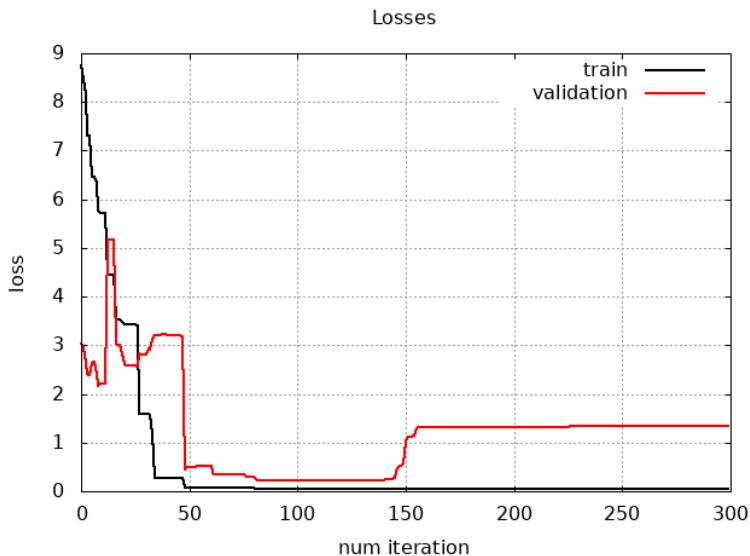


Figure 3.6 – Validation error

We can see that after approximately 75 learning iterations, the model began to predict training data much better, and the error value became lower. However, for the validation data, the MAE values began to increase. To deal with high variance, we can use special regularization techniques, which we will discuss in the following sections. We can also increase the number of training samples and decrease the number of features in one sample to reduce high variance.

The performance metrics plots we discussed in the preceding paragraphs can be drawn at the runtime of the training process. We can use them to monitor the training process to see high bias or high variance problems. Notice that for the polynomial regression model, MAE is a better performance characteristic than MSE or RMSE because squared functions average errors too much. Moreover, even a straight-line model can have low MSE values for such data because errors from both sides of the line compensate for each other. The choice between MAE and MSE depends on the specific task and goals of the project. If the overall accuracy of predictions is important, then MAE may be preferable. But MAE gives equal weight to all errors, regardless of their magnitude. This means that it is not sensitive to outliers, which can significantly affect the overall error. So, if it is necessary to minimize large errors, then MSE can give more accurate results. In some cases, you can use both metrics to analyze the performance of the model in more depth.

Normal training

Consider the case of a training process where the model has balanced bias and variance:

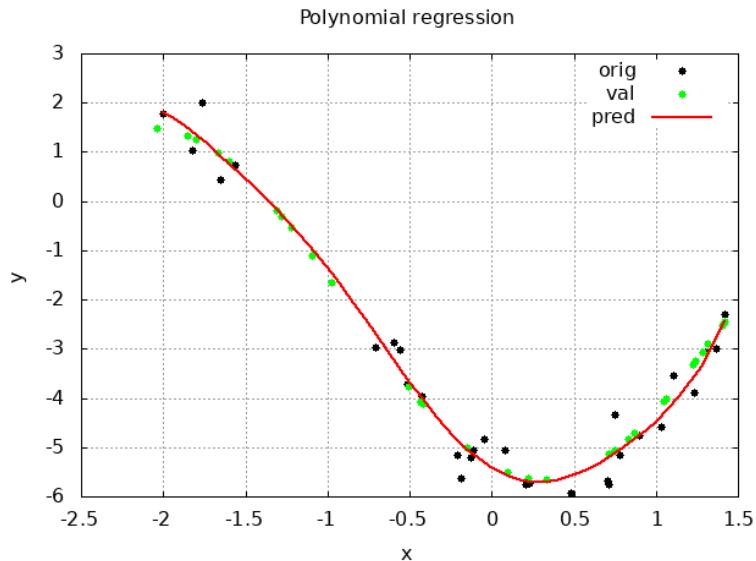


Figure 3.7 – Predictions when a model was trained ideally

In this graph, we can see that the polynomial regression model's output for the polynomial degree is equal to eight. The output values are close to both the training data and validation data. The following graph shows the MAE values during the training process:

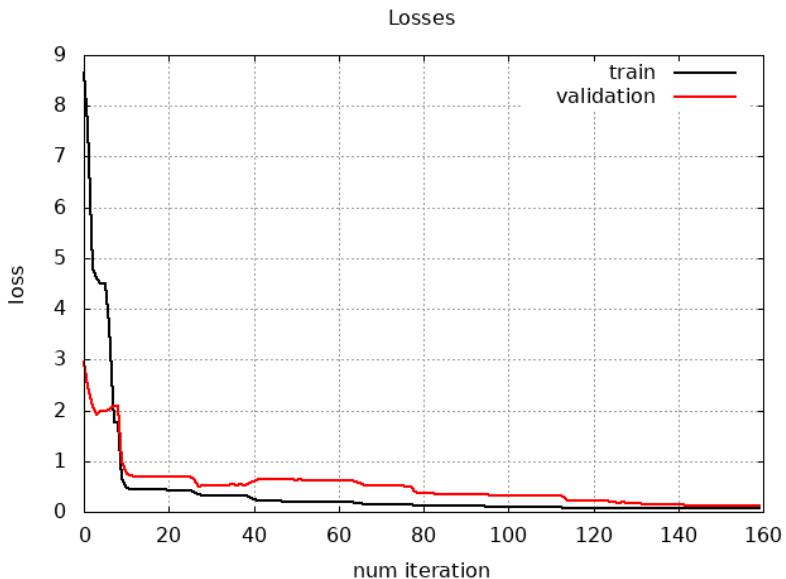


Figure 3.8 – Loss values when a model was trained ideally

We can see that the MAE value decreases consistently and that the predicted values for the training and validation data become close to the ground truth values. This means that the model's hyperparameters were good enough to balance bias and variance.

Regularization

Regularization is a technique that's used to reduce model overfitting. There are two main approaches to regularization. The first one is known as training data preprocessing. The second one is loss function modification. The main idea of the loss function modification technique is to add terms to the loss function that penalize algorithm results, thereby leading to significant variance. The idea of training data preprocessing techniques is to add more distinct training samples. Usually, in such an approach, new training samples are generated by augmenting existing ones. In general, both approaches add some prior knowledge about the task domain to the model. This additional information helps us with variance regularization. Therefore, we can conclude that regularization is any technique that leads to minimizing generalization errors.

L1 regularization – Lasso

L1 regularization is an additional term to the loss function:

$$\text{loss} + \lambda \sum_{j=1}^p |\beta_j|$$

This additional term adds the absolute value of the magnitude of parameters as a penalty. Here, λ is a regularization coefficient. Higher values of this coefficient lead to stronger regularization and can lead to underfitting. Sometimes, this type of regularization is called **Least Absolute Shrinkage and Selection Operator (Lasso)** regularization. The general idea behind L1 regularization is to penalize less important features. We can think of it as a feature selection process because as the optimization proceeds, some of the coefficients (for example, in linear regression) become zero, indicating that those features are not contributing to the model's performance. We end up with a sparse model with fewer features.

L2 regularization – Ridge

L2 regularization is also an additional term to the loss function:

$$\text{loss} + \lambda \sum_{j=1}^p \beta_j^2$$

This additional term adds a squared value of the magnitude of parameters as a penalty. This penalty shrinks the magnitude of the parameters toward zero. λ is also a coefficient of regularization. Its higher values lead to stronger regularization and can lead to underfitting because the model becomes too constrained and unable to learn complex relationships in the data. Another name for this regularization type is **ridge regularization**. Unlike L1 regularization, this type does not have a feature selection characteristic. Instead, we can interpret it as a model smoothness configurator. In addition, L2 regularization is computationally more efficient for gradient descent-based optimizers because its differentiation has an analytical solution.

In the `Dlib` library, regularization mechanisms are usually integrated into algorithm implementations—for example, the `rr_trainer` class that represents a tool for performing linear ridge regression, which is the regularized **least squares support vector machine (LSSVM)**.

There is the `LRegularizer` class in the `mlpack` library that implements a generalized L-regularizer, allowing both L1 and L2 regularization methods for NNs. Some algorithm implementations, such as the **least angle regression**, known as **LARS**, and **linear regression**, also have integrated regularization. An object of the `LARS` class can train a LARS/LASSO/Elastic Net model and has L1 and L2 regularization parameters. The `LinearRegression` class has a regularization parameter for ridge regression.

There is no standalone functionality in the `Flashlight` library for regularization. All regularizations are integrated into the NN optimization algorithm implementations.

Data augmentation

The data augmentation process can be treated as regularization because it adds some prior knowledge about the problem to the model. This approach is common in **computer vision (CV)** tasks such as image classification or object detection. In such cases, when we can see that the model begins to overfit and does not have enough training data, we can augment the images we already have to increase the size of our dataset and provide more distinct training samples. Image augmentations are random image rotations, cropping and translations, mirroring flips, scaling, and proportion changes. But data augmentation should be carefully designed for the following reasons:

- If the generated data is too similar to the original data, it can lead to overfitting.
- It can introduce noise or artifacts into the dataset, which can degrade the quality of the resulting models.
- The augmented data may not accurately reflect the real-world distribution of data, leading to a domain shift between the training and test sets. This can result in poor generalization performance.

Early stopping

Stopping the training process early can also be interpreted as a form of regularization. This means that if we detected that the model started to overfit, we can stop the training process. In this case, the model will have parameters once the training has stopped.

Regularization for NNs

L1 and L2 regularizations are widely used to train NNs and are usually called **weight decay**. Data augmentation also plays an essential role in the training processes for NNs. Other regularization methods can be used in NNs. For example, Dropout is a particular type of regularization that was developed especially for NNs. This algorithm randomly drops some NN nodes; it makes other nodes more insensitive to the weights of other nodes, which means the model becomes more robust and stops overfitting.

In the following sections, we will see how to select model hyperparameters with automated algorithms versus manual tuning.

Model selection with the grid search technique

It is necessary to have a set of proper hyperparameter values to create a good ML model. The reason for this is that having random values leads to controversial results and behaviors that are not expected by the practitioner. There are several approaches we can follow to choose the best set of hyperparameter values. We can try to use hyperparameters from algorithms we have already trained that are similar to our task. We can also try to find some heuristics and tune them manually. However, this task can be automated. The grid search technique is an automated approach for searching for the best hyperparameter values. It uses the cross-validation technique for model performance estimation.

Cross-validation

We have already discussed what the validation process is. It is used to estimate the model's performance data that we haven't used for training. If we have a limited or small training dataset, randomly sampling the validation data from the original dataset leads to the following problems:

- The size of the original dataset is reduced
- There is the probability of leaving data that's important for validation in the training part

To solve these problems, we can use the cross-validation approach. The main idea behind it is to split the original dataset in such a way that all the data will be used for training and validation. Then, the training and validation processes are performed for all partitions, and the resulting values are averaged.

The most well-known method of cross-validation is K -fold cross-validation, where K refers to the number of folds or partitions used to split the dataset. The idea is to divide the dataset into K blocks of the same size. Then, we use one of the blocks for validation and the others for training. We repeat this process K times, each time choosing a different block for validation, and in the end, we average all the results. The data splitting scheme during the whole cross-validation cycle looks like this:

1. Divide the dataset into K blocks of the same size.
2. Select one of the blocks for validation and the remaining $K-1$ blocks for training.

3. Repeat this process, making sure that each block is used for validation and the rest are used for training.
4. Average the results of the performance metrics that were calculated for the validation sets on each iteration.

The following diagram shows the cross-validation cycle:

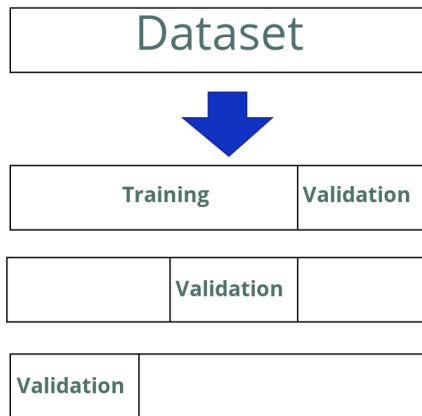


Figure 3.9 – K-fold validation scheme

Grid search

The main idea behind the grid search approach is to create a grid of the most reasonable hyperparameter values. The grid is used to generate a reasonable number of distinct parameter sets quickly. We should have some prior knowledge about the task domain to initialize the minimum and maximum values for grid generation, or we can initialize the grid with some reasonable broad ranges. However, if the chosen ranges are too broad, the process of searching for parameters can take a long time and will require a significant amount of computational resources.

At each step, the grid search algorithm chooses a set of hyperparameter values and trains a model. After that, the training step algorithm uses the K-fold cross-validation technique to estimate model performance. We should also define a single model performance estimation metric for model comparison that the algorithm will calculate at each training step for every model. After completing the model training process with each set of parameters from every grid cell, the algorithm chooses the best set of hyperparameter values by comparing the metric's values and selecting the best one. Usually, the set with the smallest value is the best one.

Consider an implementation of this algorithm in different libraries. Our task is to select the best set of hyperparameters for the polynomial regression model, which gives us the best curve that fits the given data. The data in this example is some cosine function values with some random noise.

mlpack example

The mlpack library contains a special `HyperParameterTuner` class to do hyperparameter searches with different algorithms in both discrete and continuous spaces. The default search algorithm is grid search. This class is a template and should be specialized for a concrete task. The general definition is the following:

```
template<typename MLAlgorithm, typename Metric,
         typename CV,...>
class HyperParameterTuner{...};
```

We can see that the main template parameters are the algorithm that we want to find, hyperparameters for the performance metric, and the cross-validation algorithm. Let's define a `HyperParameterTuner` object to search for the best regularization value for the linear ridge regression algorithm. The definition will be the following:

```
double validation_size = 0.2;
HyperParameterTuner<LinearRegression,
    MSE,
    SimpleCV> parameters_tuner(
        validation_size, samples, labels);
```

Here, `LinearRegression` is the target algorithm class, `MSE` is the performance metric class that calculates the MSE, and `SimpleCV` is the class that implements cross-validation. It splits data into two sets, training and validation, and then runs training on the training set and evaluates performance on the validation set. Also, we see that we pass the `validation_size` parameter into the constructor. It has a value of 0.2, which means usage of 80% of data for training and the remaining 20% for evaluation with MSE. The two following constructor parameters are our training dataset; `samples` are just samples, and `labels` are corresponding labels.

Let's see how we can generate a training dataset for these examples. It will take the two following steps:

1. Generating data that follows some predefined pattern—for example, 2D normally distributed points, plus some noise
2. Data normalization

The following sample shows how to generate data using the Armadillo library, which is the mlpack mathematical backend:

```
std::pair<arma::mat, arma::rowvec> GenerateData(
    size_t num_samples) {
    arma::mat samples = arma::randn<arma::mat>(1, num_samples);
    arma::rowvec labels = samples + arma::randn<arma::rowvec>
        (num_samples, arma::distr_param(1.0, 1.5));
    return {samples, labels};
```

```
}

...
size_t num_samples = 1000;
auto [raw_samples, raw_labels] = GenerateData(num_samples);
```

Notice that for samples, we used the `arma::mat` type, and for labels, the `arma::rowvec` type. So, samples are placed into the matrix entity and labels into a one-dimensional vector correspondingly. Also, we used the `arma::randn` function to generate normally distributed data and noise.

Now, we can normalize data in the following way:

```
data::StandardScaler sample_scaler;
sample_scaler.Fit(raw_samples);
arma::mat samples(1, num_samples);
sample_scaler.Transform(raw_samples, samples);

data::StandardScaler label_scaler;
label_scaler.Fit(raw_labels);
arma::rowvec labels(num_samples);
label_scaler.Transform(raw_labels, labels);
```

We used the object of the `StandardScaler` class from the `mlpack` library to perform normalization. This object should be first trained on some data with the `Fit` method to learn mean and variance, and then it can be applied to other data with the `Transform` method.

Now, let's discuss how to prepare the data and how to define the hyperparameter tuner object. So, we are ready to launch the grid search for the best regularization values; the following sample shows how to do it:

```
arma::vec lambdas{0.0, 0.001, 0.01, 0.1, 1.0};
double best_lambda = 0;
std::tie(best_lambda) = parameters_tuner.Optimize(lambdas);
```

We defined a `lambdas` vector with our search space and then called the `Optimize` method of the hyperparameter tuner object. You can see that the return value is a tuple, and we use the `std::tie` function to extract the specific value. The `Optimize` method takes a variable number of arguments depending on the ML algorithm we use in the search, and each argument will define a search space for each hyperparameter used in the algorithm. The constructor of the `LinearRegression` class has only one `lambda` parameter. After the search is finished, we use the best-searched parameter directly, or we can get the best-optimized model object, as shown in the following code snippet:

```
LinearRegression& linear_regression = parameters_tuner.BestModel();
```

We can now try the new model on new data to see how it works. At first, we will generate data and normalize it with a pre-trained scaler object, as shown in the following sample:

```
size_t num_new_samples = 50;
arma::dvec new_samples_values = arma::linspace<
    arma::dvec>(x_minmax.first, x_minmax.second, num_new_samples);
arma::mat new_samples(1, num_new_samples);
new_samples.row(0) = arma::trans(new_samples_values);
arma::mat norm_new_samples(1, num_new_samples);
sample_scaler.Transform(new_samples, norm_new_samples);
```

Here, we used the `arma::linspace` function to get a linearly distributed range of data. This function produces a vector; we wrote additional code that transforms this vector into a matrix object. Then, we used the already trained `sample_scaler` object to normalize the data.

The following sample shows how to use the model with the best parameter we found with the grid search:

```
arma::rowvec predictions(num_new_samples);
linear_regression.Predict(norm_new_samples, predictions);
```

The one important thing you have to notice is that the ML algorithm used for a grid search should be supported by `SimpleCV` or other validation classes. If it doesn't have a default implementation, you will need to provide it yourself.

Optuna with Flashlight example

There is no support for any hyperparameter tuning algorithms in the Flashlight library, but we can use an external tool named Optuna to deal with ML programs that we want to search the best hyperparameters for. The main idea is to use some **inter-process communication (IPC)** approach to run training with different parameters and get some performance metric values after training.

Optuna is a hyperparameter tuning framework designed to be used with different ML libraries and programs. It is implemented with the Python programming language, so the main area of its application is tools that have some Python APIs. But Optuna also has a **command-line interface (CLI)** that can be used with tools that don't support Python. Another way to use such tools is to call them from Python, passing their command-line parameters and reading their standard output. This book will show an example of such type of Optuna usage, because writing a Python program is more useful than creating Bash scripts for the same automatization tasks, from the author's point of view.

To use Optuna for hyperparameter tuning, we need to complete the three following stages:

1. Define an objective function for optimization.
2. Create a study object.
3. Run optimization process.

Let's see how we can write a simple Optuna program in Python to search for the best parameter for a polynomial regression algorithm written in C++ with the Flashlight library.

First, we need to import the required Python libraries:

```
import optuna
import subprocess
```

Then, we need to define an objective function; it's a function that is called by the Optuna tuning algorithm. It takes the `Trial` class object that contains a set of hyperparameter distributions and should return a performance metric value. The exact hyperparameter values should be sampled from the passed distributions. The following sample shows how we implement such a function:

```
def objective(trial: optuna.trial.Trial):
    lr = trial.suggest_float("learning_rate", low=0.01, high=0.05)
    d = trial.suggest_int("polynomial_degree", low=8, high=16)
    bs = trial.suggest_int("batch_size", low=16, high=64)
    result = subprocess.run(
        [binary_path, str(d), str(lr), str(bs)],
        stdout=subprocess.PIPE
    )
    mse = float(result.stdout)
    return mse
```

In this code, we used a family of functions in the `trial` object to sample concrete hyperparameter values. We sampled the `lr` learning rate with the call of the `suggest_float` method, and the `d` polynomial degree and the `bs` batch size with the `suggest_int` method. You can see that the signatures of these methods are pretty much the same. They take the name of hyperparameter, the low and the high bounds of a value range, and they can take a step value, which we didn't use. These methods can sample values from discrete space and from continuous space too. The `suggest_float` method samples from a continuous space, and the `suggest_int` method samples from a discrete space.

Then, we called the `run` method from the `subprocess` module; it launches another process in the system. This method takes the array of strings of command-line parameters and some other parameters—in our case, the `stdout` redirection. This redirection is needed because we want to get the process's output in a return value of the `run` method call; you can see this in the last lines as `result.stdout`, which is converted from string to floating-point values and is interpreted as the MSE.

Having the objective function, we can define a `study` object. This object tells Optuna how to tune hyperparameters. The two main characteristics of this object are the optimization direction and a search space for the hyperparameter sampler algorithm. The following sample shows how to define a discrete search space for our task:

```
search_space = {
    "learning_rate": [0.01, 0.025, 0.045], ],
    "polynomial_degree": [8, 14, 16],
    "batch_size": [16, 32, 64],
}
```

In this Python code, we defined a `search_space` dictionary with three items. Each item has the key string and the array value. The keys are `learning_rate`, `polynomial_degree`, and `batch_size`. After we define the search space, we can create a `study` object; the following sample shows this:

```
study = optuna.create_study(
    study_name="PolyFit",
    direction="minimize",
    sampler=optuna.samplers.GridSampler(search_space),
)
```

We used the `create_study` function from the `optuna` module and passed three parameters: `study_name`, `direction`, and `sampler`. The optimization direction we specified will be minimization, as we want to minimize MSE. For the `sampler` object, we used `GridSampler`, because we want to implement the grid search approach, and we initialized it with our search space.

The last step is to apply an optimization process and get the best hyperparameters. We can do this in the following way:

```
study.optimize(objective)
print(f"Best value: {study.best_value}
      (params: {study.best_params})\n")
```

We used the `optimize` method of our `study` object. You can see that it took a single parameter —our `objective` function that calls the external process to try sampled hyperparameters. The result of optimization was stored in the `study` object in the `best_value` and the `best_params` fields. The `best_value` field contains the best MSE value, and the `best_params` field contains the dictionary with the best hyperparameters.

This is the minimal sample of how to use Optuna. This framework has a wide variety of tuning and sampling algorithms, and a real application can be much more complicated. Also, the use of Python saves us from writing a lot of boilerplate code for the CLI approach.

Let's take a short look at a polynomial regression implementation with the Flashlight library. I will show only the most important parts; a full example can be found here: https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition/blob/main/Chapter03/flashlight/grid_fl.cc.

The first important part is that our program should take all hyperparameters from the command-line argument. It can be implemented in the following way:

```
int main(int argc, char** argv) {
    if (argc < 3) {
        std::cout << "Usage: " << argv[0] <<
        " polynomial_degree learning_rate batch_size" << std::endl;
        return 0;
    } else {
        // Hyper parameters
        int polynomial_degree = std::atoi(argv[1]);
        double learning_rate = std::atof(argv[2]);
        int batch_size = std::atoi(argv[3]);
        // Other code...
    }
}
```

First, we checked that we had enough command-line arguments by comparing `argc` parameter with the required number. In the fail case, we print a help message. But in the successful case, we read all hyperparameters from the `argv` parameter and convert them from strings to the appropriate types.

Then, our program generates 2D cosine function points and mixes them with noise. To approximate a nonlinear function with linear regression, we can convert a simple $Ax+b$ approach to a more complex polynomial like the following one:

$$a_1*x + a_2*x^2 + a_3*x^3 + \dots + a_n*x^n + b$$

It means that we have to choose some polynomial degree and convert our single-dimensional `x` value to a multidimensional one by raising `x` elements to the corresponding powers. So, the polynomial degree is the most important hyperparameter in this algorithm.

The Flashlight library doesn't have any special implementation for regression algorithms because this library is oriented toward NN algorithms. But regression can be easily implemented with the gradient descent approach; the following code sample shows how it can be done:

```
// define learnable variables
auto weight = fl::Variable(fl::rand({polynomial_degree, 1}),
                           /*calcGrad*/ true);
auto bias = fl::Variable(fl::full({1}, 0.0),
                        /*calcGrad*/ true);
```

```
float mse = 0;
fl::MeanSquaredError mse_func;
for (int e = 1; e <= num_epochs; ++e) {
    fl::Tensor error = fl::fromScalar(0);
    for (auto& batch : *batch_dataset) {
        auto input = fl::Variable(batch[0],
                                 /*calcGrad*/ false);
        auto local_batch_size = batch[0].shape().dim(1);
        auto predictions = fl::matmul(fl::transpose(
            weight), input) + fl::tile(
            bias, {1, local_batch_size});
        auto targets = fl::Variable(
            fl::reshape(batch[1], {1, local_batch_size}),
            /*calcGrad*/ false);
        // Mean Squared Error Loss
        auto loss = mse_func.forward(predictions, targets);
        // Compute gradients using backprop
        loss.backward();
        // Update the weight and bias
        weight.tensor() -= learning_rate * weight.grad().tensor();
        bias.tensor() -= learning_rate * bias.grad().tensor();
        // Clear the gradients for next iteration
        weight.zeroGrad();
        bias.zeroGrad();
        mse_func.zeroGrad();
        error += loss.tensor();
    }
    // Mean Squared Error for the epoch
    error /= batch_dataset->size();
    mse = error.scalar<float>();
```

First, we defined learnable variables for the Flashlight `autograd` system; they are weights for each power of X and bias term. Then, we ran a loop for a specified number of epochs and the second loop over data batches to make the calculation vectorized; it makes computations more effective and makes the learning process less noise-dependent. For each batch of training data, we calculated predictions by getting a polynomial value; see the line with a call of the `matmul` function. The objective of the `MeanSquareError` class was used to get the loss function value. To calculate the corresponding gradients, see the `mse_func.forward` and `mse_func.backward` calls. Then, we updated our polynomial weights and biases with the learning rate and corresponding gradients.

All these concepts will be described in detail in the following chapters. The next important part is the `error` and `mse` value calculations. The `error` value is the Flashlight tensor object that contains the average MSE for the whole epoch, and the `mse` value is the floating-point value of this single-value tensor. This `mse` variable is printed to the standard output stream of the program at the end of the training, as follows:

```
std::cout << mse;
```

We read this value in our Python program and return the result of our Optuna objective function for the given set of hyperparameters.

Dlib example

The Dlib library also contains all the necessary functionality for the grid search algorithm. However, we should use functions instead of classes. The following code snippet shows the `CrossValidationScore` function's definition. This function performs cross-validation and returns the value of the performance metric:

```
auto CrossValidationScore = [&] (const double gamma,
    const double c,
    const double degree_in) {
    auto degree = std::floor(degree_in);
    using KernelType = Dlib::polynomial_kernel<SampleType>;
    Dlib::svr_trainer<KernelType> trainer;
    trainer.set_kernel(KernelType(gamma, c, degree));
    Dlib::matrix<double> result = Dlib::
        cross_validate_regression_trainer(
            trainer, samples, raw_labels, 10);
    return result(0, 0);
};
```

The `CrossValidationScore` function takes the hyperparameters that were set as arguments. Inside this function, we defined a trainer for a model with the `svr_trainer` class, which implements kernel ridge regression based on the **support vector machine (SVM)** algorithm. We used the polynomial kernel, just like we did for the Shogun library example.

After we defined the model, we used the `cross_validate_regression_trainer()` function to train the model with the cross-validation approach. This function automatically splits our data into folds, with its last argument being the number of folds. The `cross_validate_regression_trainer()` function returns the matrix, along with the values of different performance metrics. Notice that we do not need to define them because they are predefined in the library's implementation.

The first value in this matrix is the average MSE value. We used this value as a function result. However, there is no strong requirement for what value this function should return; the requirement is that the return value should be numeric and comparable. Also, notice that we defined the `CrossValidationScore` function as a lambda to simplify access to the training data container defined in the outer scope.

Next, we can search for the best parameters that were set with the `find_min_global` function:

```
auto result = find_min_global(
    CrossValidationScore,
    {0.01, 1e-8, 5}, // minimum values for gamma, c, and degree
    {0.1, 1, 15}, // maximum values for gamma, c, and degree
    max_function_calls(50));
```

This function takes the cross-validation function, the container with minimum values for parameter ranges, the container with maximum values for parameter ranges, and the number of cross-validation repeats. Notice that the initialization values for parameter ranges should go in the same order as the arguments that were defined in the `CrossValidationScore` function. Then, we can extract the best hyperparameters and train our model with them:

```
double gamma = result.x(0);
double c = result.x(1);
double degree = result.x(2);
using KernelType = Dlib::polynomial_kernel<SampleType>;
Dlib::svr_trainer<KernelType> trainer;
trainer.set_kernel(KernelType(gamma, c, degree));
auto descision_func = trainer.train(samples, raw_labels)
```

We used the same model definition as in the `CrossValidationScore` function. For the training process, we used all of our training data. The `train` method of the `trainer` object was used to complete the training process. The training result is a function that takes a single sample as an argument and returns a prediction value.

Summary

In this chapter, we discussed how to estimate an ML model's performance and what metrics can be used for such estimation. We considered different metrics for regression and classification tasks and what characteristics they have. We also saw how performance metrics can be used to determine the model's behavior and looked at bias and variance characteristics. We looked at some high bias (underfitting) and high variance (overfitting) problems and considered how to solve them. We also learned about regularization approaches, which are often used to deal with overfitting. We then studied what validation is and how it is used in the cross-validation technique. We saw that the cross-validation technique allows us to estimate model performance while training limited data. In the last section, we combined an evaluation metric and cross-validation in the grid search algorithm, which we can use to select the best set of hyperparameters for our model.

In the next chapter, we'll learn about ML algorithms we can use to solve concrete problems. The next topic we will discuss in depth is clustering—the procedure of splitting the original set of objects into groups classified by properties. We will look at different clustering approaches and their characteristics.

Further reading

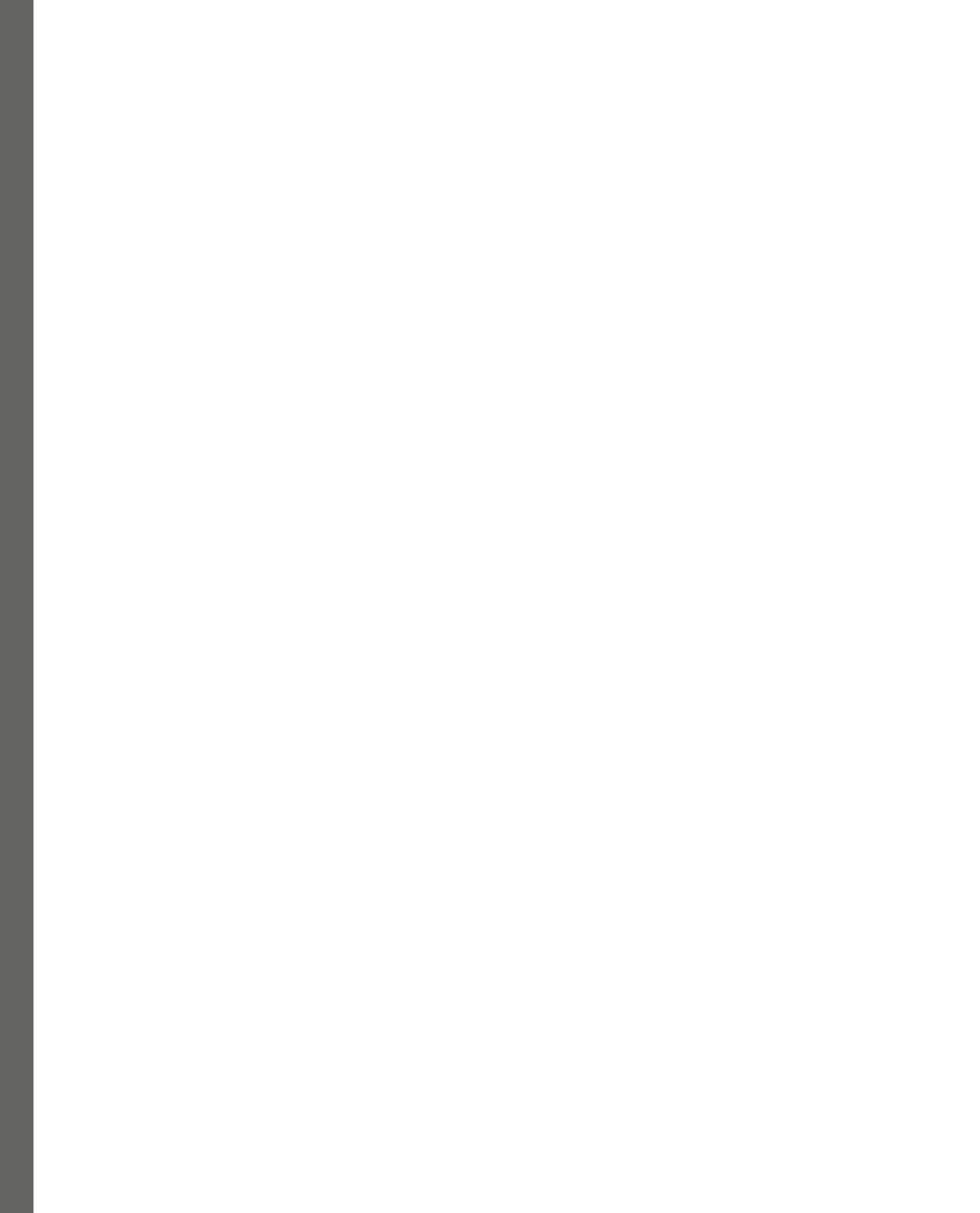
- *Choosing the Right Metric for Evaluating Machine Learning Models—Part 1:* <https://medium.com/usf-msds/choosing-the-right-metric-for-machine-learning-models-part-1-a99d7d7414e4>
- *Understand Regression Performance Metrics:* <https://becominghuman.ai/understanding-regression-performance-metrics-bdb0e7fcc1b3>
- *Classification Performance Metrics:* <https://nlpforhackers.io/classification-performance-metrics/>
- *REGULARIZATION: An important concept in Machine Learning:* <https://towardsdatascience.com/regularization-for-machine-learning-67c37b132d61>
- An overview of regularization techniques in **deep learning (DL)** (with Python code): <https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques>
- *Understanding the Bias-Variance Tradeoff:* <https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229>
- DL – Overfitting: <https://towardsdatascience.com/combatting-overfitting-in-deep-learning-efb0fdabfccc>
- *A Gentle Introduction to k-fold Cross-Validation:* <https://machinelearningmastery.com/k-fold-cross-validation/>

Part 2: Machine Learning Algorithms

In this part, we'll show you how to implement different well-known machine learning models (algorithms) using a variety of C++ frameworks.

This part comprises the following chapters:

- *Chapter 4, Clustering*
- *Chapter 5, Anomaly Detection*
- *Chapter 6, Dimensionality Reduction*
- *Chapter 7, Classification*
- *Chapter 8, Recommender Systems*
- *Chapter 9, Ensemble Learning*



4

Clustering

Clustering is an unsupervised machine learning method that's used for splitting the original dataset of objects into groups classified by properties. In **machine learning**, an object is typically represented as a point in a multidimensional metric space. Every space dimension corresponds to an object property (feature), and the metric is a function of the values of these properties. Depending on the types of dimensions in this space, which can be both numerical and categorical, we choose a type of clustering algorithm and specific metric function. This choice depends on the nature of different object properties' types.

At the present stage, clustering is often used as the first step in data analysis. The task of clustering was formulated in scientific areas such as statistics, pattern recognition, optimization, and machine learning. At the time of writing, the number of methods for partitioning groups of objects into clusters is quite large—several dozen algorithms, and even more when you take into account their various modifications.

The following topics will be covered in this chapter:

- Measuring distance in clustering
- Types of clustering algorithms
- Examples of using the `mlpack` library for dealing with the clustering task samples
- Examples of using the `Dlib` library for dealing with the clustering task samples
- Plotting data with C++

Technical requirements

You'll require the following technologies and installations to complete this chapter:

- A modern C++ compiler with C++17 support
- CMake build system version ≥ 3.8
- The `Dlib` library

- The `mlpack` library
- The `plotcpp` library

The code files for this chapter can be found in this book's GitHub repository: <https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition/tree/main/Chapter04>.

Measuring distance in clustering

A metric or distance measure is essential in clustering because it determines the similarity between objects. However, before applying a distance measure to objects, we must make a vector of object characteristics; usually, this is a set of numerical values, such as human height or weight. Also, some algorithms can work with categorical object features (or characteristics). The standard practice is to normalize feature values. Normalization ensures that each feature has the same impact in a distance measure calculation. Many distance measure functions can be used in the scope of the clustering task. The most popular ones that are used for numerical properties are **Euclidean distance**, **squared Euclidean distance**, **Manhattan distance**, and **Chebyshev distance**. The following subsections describe them in detail.

Euclidean distance

Euclidean distance is the most widely used distance measure. In general, this is a geometric distance in the multidimensional space. The formula for Euclidean distance is as follows:

$$\delta(x, \bar{x}) = \sqrt{\sum_i^n (x_i - \bar{x}_i)^2}$$

Squared Euclidean distance

Squared Euclidean distance has the same properties as Euclidean distance but assigns greater significance (weight) to the distant values than to closer ones. Here's the formula for squared Euclidean distance:

$$\delta(x, \bar{x}) = \sum_i^n (x_i - \bar{x}_i)^2$$

Manhattan distance

Manhattan distance is an average difference by coordinates. In most cases, its value gives the same clustering results as Euclidean distance. However, it reduces the significance (weight) of the distant values (outliers). Here's the formula for Manhattan distance:

$$\delta(x, \bar{x}) = \sum_i^n |x_i - \bar{x}_i|$$

Chebyshev distance

Chebyshev distance can be useful when we need to classify two objects as different when they differ only by one of the coordinates. Here's the formula for Chebyshev distance:

$$\delta(x, \bar{x}) = \max(|x_i - \bar{x}_i|)$$

The following diagram shows the differences between the various distances:

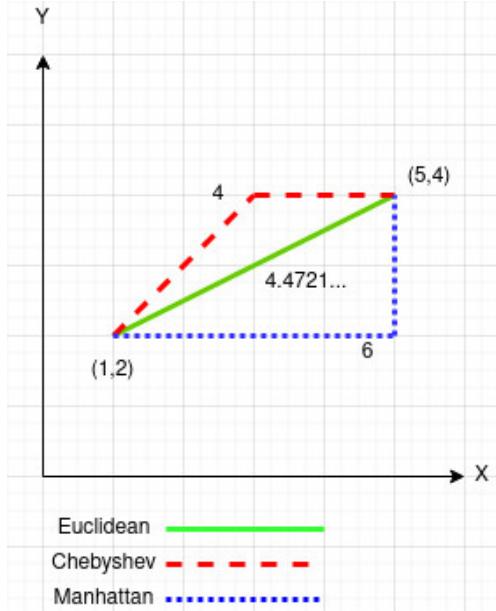


Figure 4.1 – The difference between different distance measures

Here, we can see that *Manhattan* distance is the sum of the distances in both dimensions, like walking along city blocks. *Euclidean* distance is just the length of a straight line. *Chebyshev* distance is a more flexible alternative to *Manhattan* distance because diagonal moves are also taken into account.

In this section, we became familiar with the main clustering concept, which is a distance measure. In the following section, we'll discuss various types of clustering algorithms.

Types of clustering algorithms

There are different types of clustering that we can classify into the following groups: **partition-based**, **spectral**, **hierarchical**, **density-based**, and **model-based**. The partition-based group of clustering algorithms can be logically divided into distance-based methods and ones based on graph theory.

Before we cover different types of clustering algorithms, let's understand the main difference between clustering and classification. The main difference between the two is an undefined set of target groups, which is determined by the clustering algorithm. The set of target groups (clusters) is the algorithm's result.

We can split cluster analysis into the following phases:

- Selecting objects for clustering
- Determining the set of object properties that we'll use for the metric
- Normalizing property values
- Calculating the metric
- Identifying distinct groups of objects based on metric values

After analyzing clustering results, some correction may be required for the selected metric of the chosen algorithm.

We can use clustering for various real-world tasks, including the following:

- Splitting news into several categories for advertisers
- Identifying customer groups by their preferences for market analysis
- Identifying plant and animal groups for biological studies
- Identifying and categorizing properties for city planning and management
- Detecting earthquake epicenter clusters to identify danger zones
- Categorizing groups of insurance policyholders for risk management
- Categorizing books in libraries
- Searching for hidden structural similarities in the data

With that, let's dive into the different types of clustering algorithms.

Partition-based clustering algorithms

The partition-based methods use a similarity measure to combine objects into groups. A practitioner usually selects the similarity measure for such kinds of algorithms, using prior knowledge about a problem or heuristics to select the measure properly. Sometimes, several measures need to be tried with the same algorithm so that the best one can be chosen. Also, partition-based methods usually require either the number of desired clusters or a threshold that regulates the number of output clusters to be specified explicitly. The choice of a similarity measure can significantly affect the quality and accuracy of the clusters produced, potentially leading to misinterpretations of data patterns and insights.

Distance-based clustering algorithms

The most known representatives of this family of methods are the k-means and k-medoids algorithms. They take the k input parameter and divide the data space into k clusters so that the similarity between objects in one cluster is maximal. Also, they minimize the similarity between objects of different clusters. The similarity value is calculated as the distance from the object to the cluster center. The main difference between these methods lies in the way the cluster center is defined.

With the k-means algorithm, the similarity is proportional to the distance to the cluster center of mass. The cluster center of mass is the average value of cluster objects' coordinates in the data space. The k-means algorithm can be briefly described with a few steps. First, we select k random objects and define each of them as a cluster prototype that represents the cluster's center of mass. Then, the remaining objects are attached to the cluster with greater similarity. After that, the center of mass of each cluster is recalculated. For each obtained partition, a particular evaluation function is calculated, the values of which at each step form a converging series. This process continues until the specified series converges to its limit value.

In other words, moving objects from one cluster to another ends when the clusters remain unchanged. Minimizing the evaluation function allows the resulting clusters to be as compact and separate as possible. The k-means method works well when clusters are compact *clouds* that are significantly separated from each other. It's useful for processing large amounts of data but isn't applicable for detecting clusters of non-convex shapes or clusters with very different sizes. Moreover, the method is susceptible to noise and isolated points since even a small number of such points can significantly affect how the center mass of the cluster is calculated.

To reduce the influence of noise and isolated points on the clustering result, the k-medoids algorithm, in contrast to the k-means algorithm, uses one of the cluster objects (known as the representative object) as the center of the cluster. As in the k-means method, k representative objects are selected at random. Each of the remaining objects is combined into a cluster with the nearest representative object. Then, each representative object is replaced iteratively with an arbitrary unrepresentative object from the data space. The replacement process continues until the quality of the resulting clusters improves. The clustering quality is determined by the sum of deviations between objects and the representative object of the corresponding cluster, which the method tries to minimize. Thus, the iterations continue until the representative object in each of the clusters becomes the medoid.

The **medoid** is the object closest to the center of the cluster. The algorithm is poorly scalable for processing large amounts of data, but this problem is solved by the **Clustering Large Applications based on RAndomized Search (CLARANS)** algorithm, which complements the k-medoids method. CLARANS attempts to address scalability issues by using a randomized search technique to find good solutions more efficiently. Such an approach makes it possible to quickly converge on a good solution without exhaustively searching all possible combinations of medoids. For multidimensional clustering, the **Projected Clustering (PROCLUS)** algorithm can be used.

Graph theory-based clustering algorithms

The essence of algorithms based on graph theory is to represent target objects in graph form. Graph vertices correspond to objects, and the edge weights are equal to the distance between vertices. The advantages of graph clustering algorithms are their excellent visibility, relative ease of implementation, and their ability to make various improvements based on geometrical considerations. The main graph theory concepts used for clustering are selecting connected components, constructing a minimum spanning tree, and multilayer graph clustering.

The algorithm for selecting connected components is based on the R input parameter, and the algorithm removes all edges in the graph with distances greater than R . Only the closest pairs of objects remain connected. The algorithm's goal is to find the R value at which the graph collapses into several connected components. The resulting components are clusters. To select the R parameter, a histogram of the distribution of pairwise distances is usually constructed. For problems with a well-defined cluster data structure, there will be two peaks in the histogram—one corresponds to in-cluster distances and the second to inter-cluster distances. The R parameter is selected from the minimum zone between these peaks. Managing the number of clusters using the distance threshold can be difficult.

The minimum spanning tree algorithm builds a minimal spanning tree on the graph, and then successively removes the edges with the highest weight. The following diagram shows the minimum spanning tree that's been obtained for nine objects:

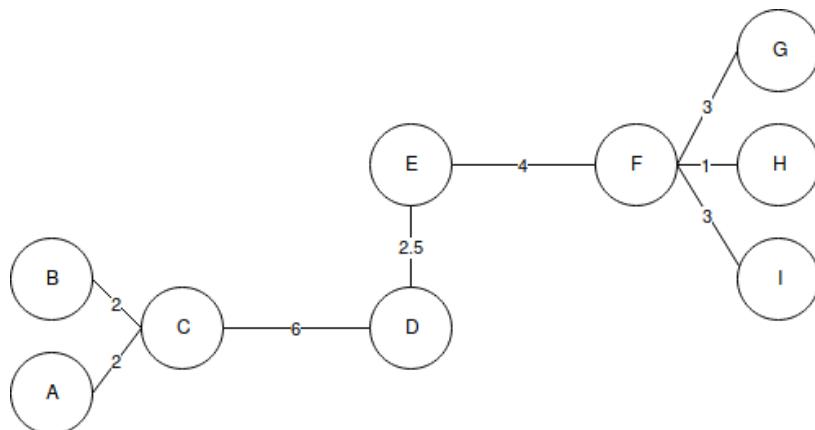


Figure 4.2 – Spanning tree example

By removing the link between C and D , with a length of 6 units (the edge with the maximum distance), we obtain two clusters: $\{A, B, C\}$ and $\{D, E, F, G, H, I\}$. We can divide the second cluster into two more clusters by removing the EF edge, which has a length of 4 units.

The multilayer clustering algorithm is based on identifying connected components of a graph at some level of distance between objects (vertices). The threshold, C , defines the distance level—for example, if the distance between objects is $0 \leq \delta(x, \bar{x}) \leq 1$, then $0 \leq c \leq 1$.

The layer clustering algorithm generates a sequence of sub-graphs of the graph, G , that reflect the hierarchical relationships between clusters, $G^0 \subseteq G^1 \subseteq \dots \subseteq G^m$, where the following applies:

- $G^t = (V, E^t)$: A sub-graph on the c^t level
- $E^t = e_{ij} \in E : \delta_{ij} \leq c^t$
- c^t : The t^{th} threshold of distance
- m : The number of hierarchy levels
- $G^0 = (V, o)$, o : An empty set of graph edges, when $t^0 = 1$
- $G^m = G$: A graph of objects without thresholds on distance, when $t^m = 1$

By changing the c^0, \dots, c^m distance thresholds, where $0 = c^0 < c^1 < \dots < c^m = 1$, it's possible to control the hierarchy depth of the resulting clusters. Thus, a multilayer clustering algorithm can create both flat and hierarchical data partitioning.

Spectral clustering algorithms

Spectral clustering refers to all methods that divide a set of data into clusters using the eigenvectors of the adjacency matrix of a graph or other matrices derived from it. An adjacency matrix describes a complete graph with vertices in objects and edges between each pair of objects with a weight corresponding to the degree of similarity between these vertices. Spectral clustering involves transforming the initial set of objects into a set of points in space whose coordinates are elements of eigenvectors. The formal name for such a task is the **normalized cuts problem**.

The resulting set of points is then clustered using standard methods—for example, with the k-means algorithm. Changing the representation created by eigenvectors allows us to set the properties of the original set of clusters more clearly. Thus, spectral clustering can separate points that can't be separated by applying k-means—for example, when the k-means method gets a convex set of points. The main disadvantage of spectral clustering is its cubic computational complexity and quadratic memory requirements.

Hierarchical clustering algorithms

Among the algorithms of hierarchical clustering, there are two main types: **bottom-up** and **top-down-based algorithms**. Top-down algorithms work on the principle that at the beginning, all objects are placed in one cluster, which is then divided into smaller and smaller clusters. Bottom-up algorithms are more common than top-down ones. They place each object in a separate cluster at the beginning of the work and then merge clusters into larger ones until all the objects in the dataset are contained in one cluster, building a system of nested partitions. The results of such algorithms are usually presented in tree form, called a **dendrogram**. A classic example of such a tree is the *Tree of Life*, which describes the classification of animals and plants.

The main problem with hierarchical methods is the difficulty of determining the stop condition in such a way as to isolate natural clusters and, at the same time, prevent their excessive splitting. Another problem with hierarchical clustering methods is choosing the point of separation or merging of clusters. This choice is critical because after splitting or merging clusters at each subsequent step, the method will operate only on newly formed clusters. Therefore, the wrong choice of a merge or split point at any step can lead to poor-quality clustering. Also, hierarchical methods can't be applied to large datasets because deciding whether to divide or merge clusters requires a large number of objects and clusters to be analyzed, which leads to a significant computational complexity of the method.

There are several metrics or linkage criteria for cluster union that are used in hierarchical clustering methods:

- **Single linkage (nearest neighbor distance):** In this method, the distance between the two clusters is determined by the distance between the two closest objects (nearest neighbors) in different clusters. The resulting clusters tend to chain together.
- **Complete linkage (distance between the most distant neighbors):** In this method, the distances between clusters are determined by the largest distance between any two objects in different clusters (that is, the most distant neighbors). This method usually works very well when objects come from separate groups. If the clusters are elongated or their natural type is *chained*, then this method is unsuitable.
- **Unweighted pairwise mean linkage:** In this method, the distance between two different clusters is calculated as the average distance between all pairs of objects in them. This method is useful when objects form different groups, but it works equally well in the case of elongated (chained-type) clusters.
- **Weighted pairwise mean linkage:** This method is identical to the unweighted pairwise mean method, except that the size of the corresponding clusters (the number of objects contained in them) is used as a weighting factor in the calculations. Therefore, this method should be used when we assume unequal cluster sizes.
- **Weighted centroid linkage:** In this method, the distance between two clusters is defined as the distance between their centers of mass.

- **Weighted centroid linkage (median):** This method is identical to the previous one, except that the calculations use weights for the distance measured between cluster sizes. Therefore, if there are significant differences in-cluster sizes, this method is preferable to the previous one.

The following diagram displays a hierarchical clustering dendrogram:

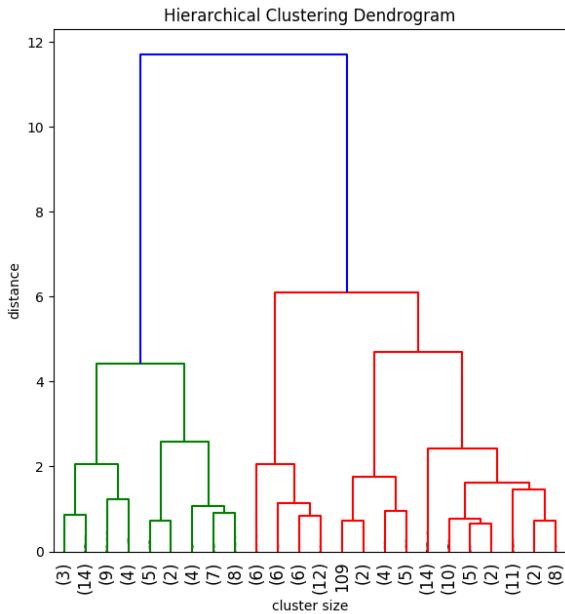


Figure 4.3 – Hierarchical clustering example

The preceding diagram shows an example of a dendrogram for hierarchical clustering, where you can see how the number of clusters depends on the distance between objects. Larger distances lead to a smaller number of clusters.

Density-based clustering algorithms

In density-based methods, clusters are considered as regions where the multiple objects' density is high. This is separated by regions with a low density of objects.

The **Density-Based Spatial Clustering of Applications with Noise (DBSCAN)** algorithm is one of the first density clustering algorithms to be created. The basis of this algorithm is several statements, detailed as follows:

- The ϵ - neighborhood property of an object is the ϵ radius neighborhood area around the object.
- The root object is an object whose ϵ - neighborhood contains a minimum non-zero number of objects. Assume that this minimum number equals a predefined value named *MinPts*.

- The p object is directly densely accessible from the q object if p is in the ϵ -neighborhood property of q and q is the root object.
- The p object is densely accessible from the q object for the given ϵ and $MinPts$ if there's a sequence of p_1, \dots, p_n objects, where $p_1 = q$ and $p_n = p$, such that p_{i+1} is directly densely accessible from p_i , $1 \leq i \leq n$.
- The p object is densely connected to the q object for the given ϵ and $MinPts$ if there's an o object such that p and q are densely accessible from o .

The DBSCAN algorithm checks the neighborhood of each object to search for clusters. If the ϵ -neighborhood property of the p object contains more points than $MinPts$, then a new cluster is created with the p object as a root object. DBSCAN then iteratively collects objects directly densely accessible from root objects, which can lead to the union of several densely accessible clusters. The process ends when no new objects can be added to any cluster.

Unlike the partition-based methods, DBSCAN doesn't require the number of clusters to be specified in advance; it only requires the ϵ and $MinPts$ values as these parameters directly affect the result of clustering. The optimal values of these parameters are difficult to determine, especially for multidimensional data spaces. Also, the distributed data in such spaces is often asymmetrical, which makes it impossible to use global density parameters for their clustering. For clustering multidimensional data spaces, there's the **Subspace Clustering (SUBCLU)** algorithm, which is based on the DBSCAN algorithm.

The **MeanShift** approach also falls into the category of density-based clustering algorithms. It's a non-parametric algorithm that shifts dataset points toward the center of the highest-density region within a certain radius. The algorithm makes such shifts iteratively until points converge to a local maximum of the density function. Such local maxima are also called the mode, so the algorithm is sometimes called mode-seeking. These local maximums represent the cluster centroids in the dataset.

Model-based clustering algorithms

Model-based algorithms assume that there's a particular mathematical model of the cluster in the data space and try to maximize the likelihood of this model and the data available. Often, this uses the apparatus of mathematical statistics.

The **Expectation-Maximization (EM)** algorithm assumes that the dataset can be modeled using a linear combination of multidimensional normal distributions. Its purpose is to estimate distribution parameters that maximize the likelihood function that's used as a measure of model quality. In other words, it assumes that the data in each cluster obeys a particular distribution law—namely, the normal distribution. With this assumption, it's possible to determine the optimal parameters of the distribution law—the mean and variance at which the likelihood function is maximal. Thus, we can assume that any object belongs to all clusters, but with a different probability. In this instance, the task will be to fit the set of distributions to the data and determine the probabilities of the object belonging to each cluster. The object should be assigned to the cluster for which this probability is higher than the others.

The EM algorithm is simple and easy to implement. It isn't sensitive to isolated objects and quickly converges in the case of successful initialization. However, it requires us to specify k number of clusters, which implies a *priori* knowledge about the data. Also, if the initialization fails, the algorithm may be slow to converge, or we might obtain a poor-quality result. Such algorithms don't apply to high-dimensionality spaces since, in this case, it's complicated to assume a mathematical model for distributing data in this space.

Now that we understand the various types of clustering algorithms, let's look at their uses in many industries to group similar data points into clusters. Here are some examples of how clustering algorithms can be applied:

- **Customer segmentation:** Clustering algorithms can be used to segment customers based on their purchase history, demographics, and other attributes. This information can then be used for targeted marketing campaigns, personalized product recommendations, and customer service.
- **Image recognition:** In the field of computer vision, clustering algorithms are used to group images based on visual features such as color, texture, and shape. This can be useful for image classification, object detection, and scene understanding.
- **Fraud detection:** In finance, clustering algorithms can detect suspicious transactions by grouping them based on similarities in transaction patterns. This helps to identify potential fraud and prevent financial losses.
- **Recommender systems:** In e-commerce, clustering algorithms group products based on customer preferences. This allows recommender systems to suggest relevant products to customers, increasing sales and customer satisfaction.
- **Social network analysis:** In social media, clustering algorithms identify groups of users with similar interests or behaviors. This enables targeted advertising, content creation, and community building.
- **Genomics:** In biology, clustering algorithms analyze gene expression data to identify groups of genes that are co-expressed under specific conditions. This aids in understanding gene function and disease mechanisms.
- **Text mining:** In natural language processing, clustering algorithms categorize documents based on their content. This is useful for topic modeling, document classification, and information retrieval.

These are just a few examples of the wide range of applications of clustering algorithms. The specific use case will depend on the industry, dataset, and business objectives.

In this section, we discussed various clustering algorithms and their uses. In the following sections, we'll learn how to use them in real-world examples with various C++ libraries.

Examples of using the mlpack library for dealing with the clustering task samples

The `mlpack` library contains implementations of the model-based, density-based, and partition-based clustering approaches. The model-based algorithm is called **Gaussian Mixture Models (GMM)** and is based on EM, while the partition-based algorithm is the k-means algorithm. There are two density-based algorithms we can use: DBSCAN and MeanShift clustering.

GMM and EM with mlpack

The GMM algorithm assumes that clusters can be fit to some Gaussian (normal) distributions; it uses the EM approach for training. There are the `GMM` and `EMFit` classes in the `mlpack` library that implement this approach, as illustrated in the following code snippet:

```
GMM gmm(num_clusters, /*dimensionality*/ 2);
KMeans<> kmeans;
size_t max_iterations = 250;
double tolerance = 1e-10;
EMFit<KMeans<>, NoConstraint> em(max_iterations, tolerance, kmeans);
gmm.Train(inputs, /*trials*/ 3, /*use_existing_model*/ false, em);
```

Notice that the constructor of the `GMM` class takes the desired number of clusters and feature dimensionality as an argument. After `GMM` object initialization, the object of the `EMFit` class was initialized with maximum iterations, tolerance, and a clustering object. The tolerance parameter in `EMFit` controls how similar two points must be to be considered as part of the same cluster. A higher tolerance value means that the algorithm will group more points, resulting in fewer clusters. Conversely, a lower tolerance value leads to more clusters with fewer points in each one. The clustering object—in our case, `kmeans`—will be used by the algorithm to find initial centroids for Gaussian fitting. Then, we passed the training features and the `EM` object into the training method. Now, we have the trained `GMM` model. In the `mlpack` library, the trained `gmm` object should be used to classify new feature points, but we can use it to show cluster assignments for the original data that we used for training. The following piece of code shows these steps and also plots the results of clustering:

```
arma::Row<size_t> assignments;
gmm.Classify(inputs, assignments);

Clusters plot_clusters;
for (size_t i = 0; i != inputs.n_cols; ++i) {
    auto cluser_idx = assignments[i];
    plot_clusters[cluser_idx].first.push_back(inputs.at(0, i));
    plot_clusters[cluser_idx].second.push_back(inputs.at(1, i));
```

```

    }

PlotClusters(plot_clusters, "GMM", name + "-gmm.png");

```

Here, we used the `GMM::Classify()` method to identify which cluster our objects belong to. This method filled a row vector of cluster identifiers per element that corresponds to the input data. The resulting cluster indices were used for filling the `plot_clusters` container. This container maps cluster indices with input data coordinates for plotting. It was used as an argument for the `PlotClusters()` function, which visualized the clustering result, as illustrated in the following figure:

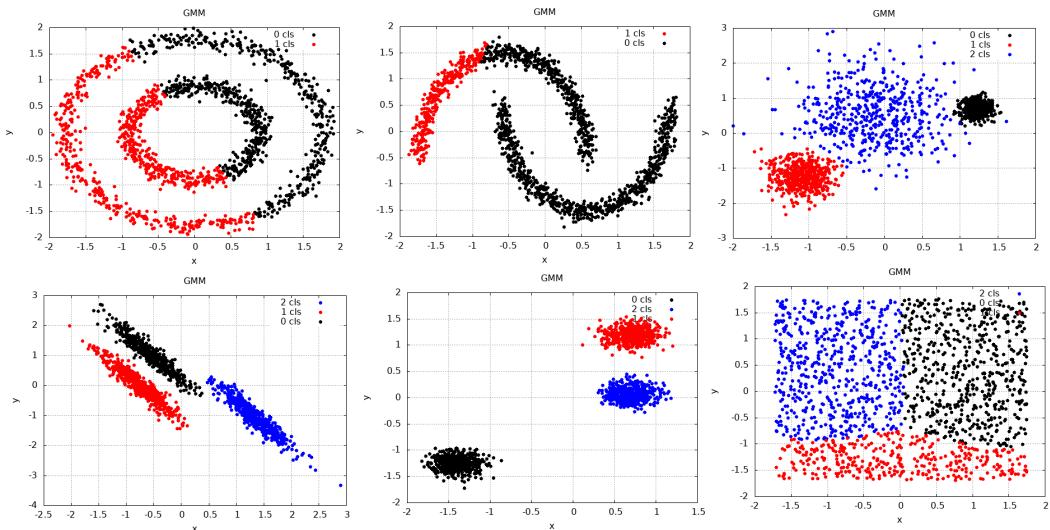


Figure 4.4 – mlpack GMM clustering visualization

In the preceding picture, we can see how the GMM and EM algorithms work on different artificial datasets.

K-means clustering with mlpack

The k-means algorithm in the `mlpack` library is implemented in the `KMeans` class. The constructor of this class takes several parameters, with the most important ones being the number of iterations and the object for distance metric calculation. In the following example, we'll use the default values so that the constructor will be called without parameters. Once we've constructed an object of the `KMeans` type, we'll use the `KMeans::Cluster()` method to run the algorithm and assign a cluster label to each of the input elements, as follows:

```

arma::Row<size_t> assignments;
KMeans<> kmeans;
kmeans.Cluster(inputs, num_clusters, assignments);

```

The result of clusterization is the `assignments` container object with labels. Notice that the desired number of clusters was passed as an argument for the `Cluster` method. The following code sample shows how to plot the results of clustering:

```
Clusters plot_clusters;
for (size_t i = 0; i != inputs.n_cols; ++i) {
    auto cluser_idx = assignments[i];
    plot_clusters[cluser_idx].first.push_back(inputs.at(0, i));
    plot_clusters[cluser_idx].second.push_back(inputs.at(1, i));
}
PlotClusters(plot_clusters, "K-Means", name + "-kmeans.png");
```

As we can see, the code for visualization is the same as it was for the previous example—it's the result of the uniform clustering API in the `mlpack` library. We received the same `assignments` container that we converted in the data structure, which is suitable for the visualization library we're using, and called the `PlotClusters` function. The visualization result is illustrated in the following figure:

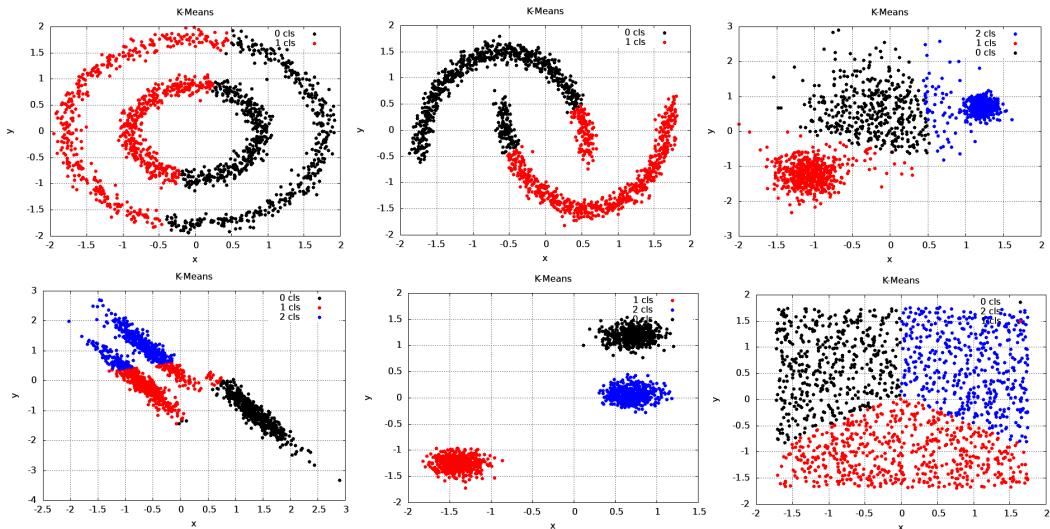


Figure 4.5 – `mlpack` K-means clustering visualization

In the preceding figure, we can see how the k-means algorithm works on different artificial datasets.

DBSCAN with mlpack

The DBSCAN class implements the corresponding algorithm in the mlpack library. The constructor of this class takes several parameters, but the two most important are the epsilon and the minimum points number. In the following code snippet, we're creating the object of this class:

```
DBSCAN<> dbscan(/*epsilon*/ 0.1, /*min_points*/ 15);
```

Here, `epsilon` is the radius of a range search, while `min_points` is the minimum number of points required to form a cluster. After constructing an object of the DBSCAN type, we can use the `Cluster()` method to run the algorithm and assign a cluster label to each of the input elements, as follows:

```
dbscan.Cluster(inputs, assignments);
```

The result of clusterization is an `assignments` container object with labels. Notice that for this algorithm, we didn't specify the desired number of clusters because the algorithm determined them by itself. The code for the visualization is the same as for the previous examples—we convert the `assignments` container into a data structure that's suitable for the visualization library we're using and call the `PlotClusters` function. The following figure shows the DBSCAN clustering visualization result:

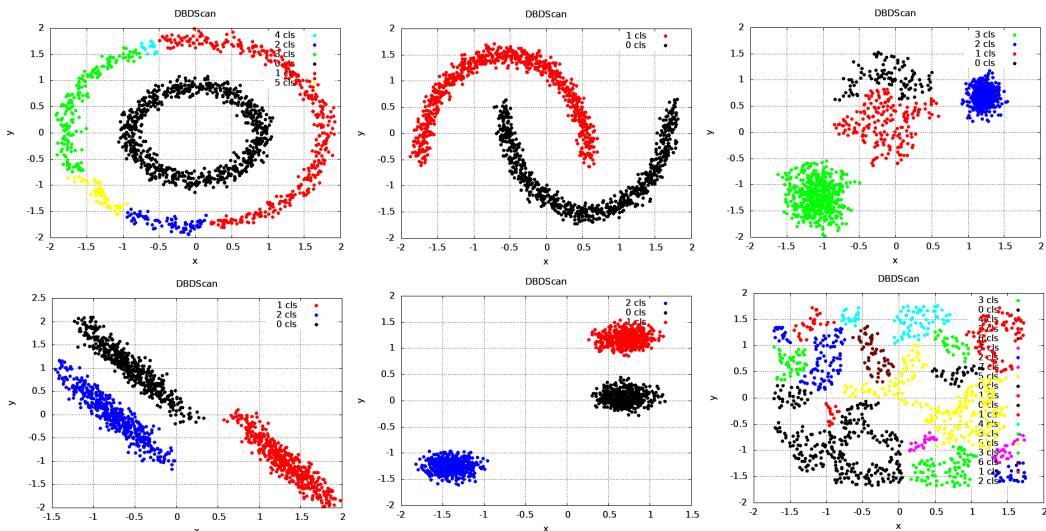


Figure 4.6 – mlpack DBSCAN clustering visualization

In the preceding figure, we can see how the DBSCAN algorithm works on different artificial datasets. The main difference from previous algorithms is the bigger number of clusters that the algorithm found. From this, we can see that their centroids are near some local density maximums.

MeanShift clustering with mlpack

The `MeanShift` class implements the corresponding algorithm in the `mlpack` library. The constructor of this class takes several parameters, with the most important one being the density region search radius. It's quite tricky to manually find the appropriate value for this parameter. However, the library gives us a very useful method to determine it automatically. In the following code snippet, we're creating an object of the `MeanShift` class without specifying the radius parameter explicitly:

```
MeanShift<> mean_shift;
auto radius = mean_shift.EstimateRadius(inputs);
mean_shift.Radius(radius);
```

Here, we used the `EstimateRadius` method to get the automatic radius estimation, which is based on the **k-nearest neighbor (KNN)** search. Once we've initialized the `MeanShift` object with the appropriate search radius value, we can use the `Cluster()` method to run the algorithm and assign a cluster label to each of the input elements, as follows:

```
arma::Row<size_t> assignments;
arma::mat centroids;
mean_shift.Cluster(inputs, assignments, centroids);
```

The result of clusterization is the `assignments` container object with labels and an additional matrix that contains the cluster centroids coordinates. For this algorithm, we also didn't specify the number of clusters because the algorithm determined them by itself. The code for visualization is the same as for the previous examples—we convert the `assignments` container into a data structure suitable for the visualization library we're using and call the `PlotClusters` function. The following figure shows the `MeanShift` clustering visualization result:

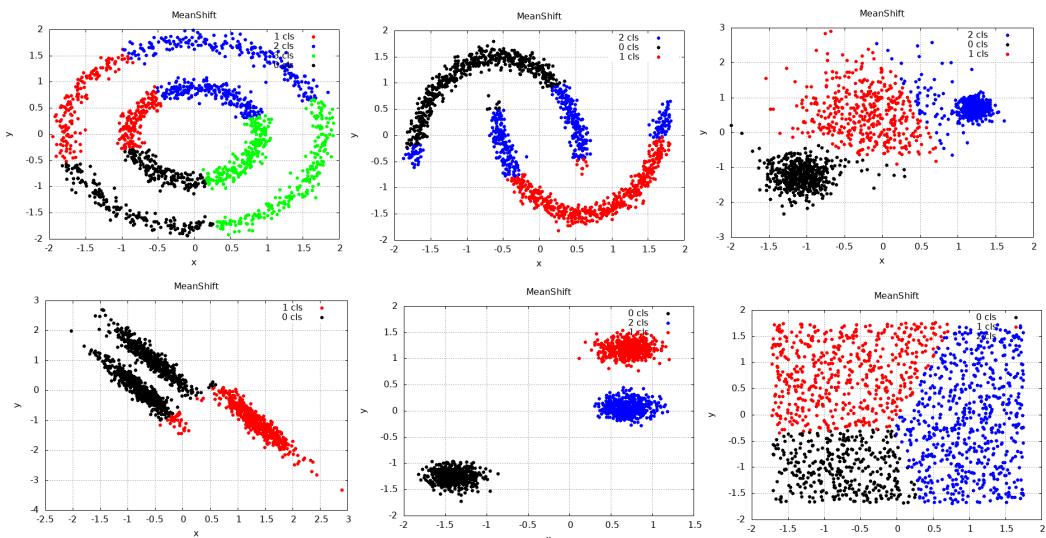


Figure 4.7 – `mlpack` `MeanShift` clustering visualization

The preceding figure shows how the MeanShift algorithm works on different artificial datasets. We can see that the results are somehow similar to those for K-means clustering but the number of clusters was determined automatically. We can also see that in the one of datasets, the algorithm failed to get the correct number of clusters, so we must perform experiments with search radius values to get more precise clustering results.

Examples of using the Dlib library for dealing with the clustering task samples

The Dlib library provides k-means, spectral, hierarchical, and two more graph clustering algorithms—**Newman** and **Chinese Whispers**—as clustering methods. Let's take a look.

K-means clustering with Dlib

The Dlib library uses kernel functions as the distance functions for the k-means algorithm. An example of such a function is the radial basis function. As an initial step, we define the required types, as follows:

```
typedef matrix<double, 2, 1> sample_type;
typedef radial_basis_kernel<sample_type> kernel_type;
```

Then, we initialize an object of the `kkmeans` type. Its constructor takes an object that will define cluster centroids as input parameters. We can use an object of the `kcentroid` type for this purpose. Its constructor takes three parameters: the first one is the object that defines the kernel (distance function), the second is the numerical accuracy for the centroid estimation, and the third is the upper limit on the runtime complexity (actually, the maximum number of dictionary vectors the `kcentroid` object is allowed to use), as illustrated in the following code snippet:

```
kcentroid<kernel_type> kc(kernel_type(0.1), 0.01, 8);
kkmeans<kernel_type> kmeans(kc);
```

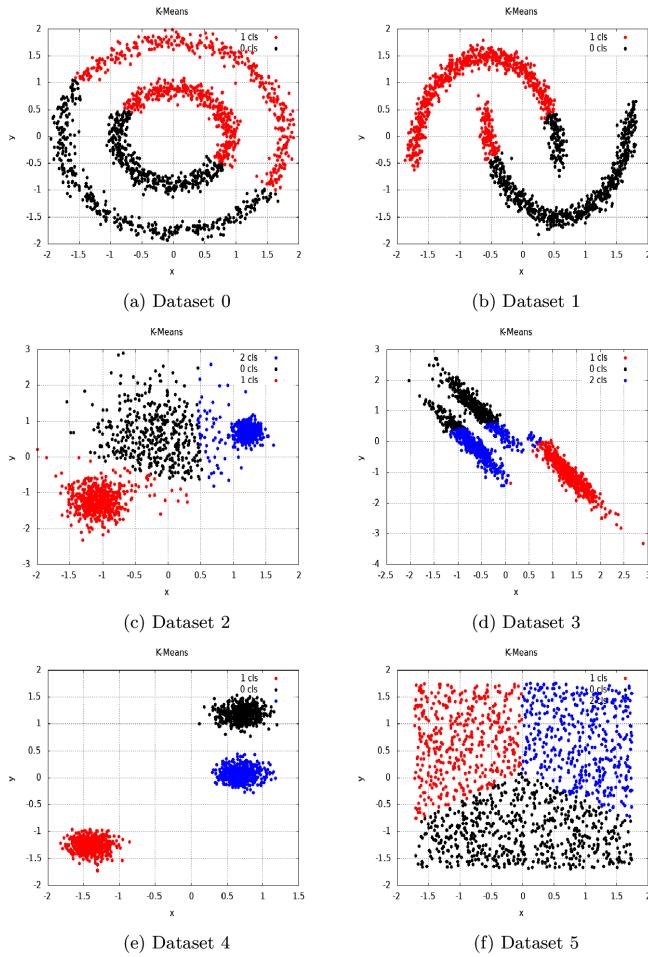
As a next step, we initialize cluster centers with the `pick_initial_centers()` function. This function takes the number of clusters, the output container for center objects, the training data, and the distance function object as parameters, as follows:

```
std::vector<sample_type> samples; //training dataset
...
size_t num_clusters = 2;
std::vector<sample_type> initial_centers;
pick_initial_centers(num_clusters,
                     initial_centers,
                     samples,
                     kmeans.get_kernel());
```

When initial centers are selected, we can use them for the `kkmeans::train()` method to determine exact clusters, as follows:

```
kmeans.set_number_of_centers(num_clusters);
kmeans.train(samples, initial_centers);
for (size_t i = 0; i != samples.size(); i++) {
    auto cluster_idx = kmeans(samples[i]);
    ...
}
```

We used the `kmeans` object as a functor to perform clustering on a single data item. The clustering result will be the cluster's index for the item. Then, we used cluster indices to visualize the final clustering result, as illustrated in the following figure:



K-Means clustering algorithm on different datasets

Figure 4.8 – Dlib K-means clustering visualization

In the preceding figure, we can see how the k-means clustering algorithm that's implemented in the Dlib library works on different artificial datasets.

Spectral clustering with Dlib

The spectral clustering algorithm in the Dlib library is implemented in the `spectral_cluster` function. It takes the distance function object, the training dataset, and the number of clusters as parameters. As a result, it returns a container with cluster indices, which have the same ordering as the input data. In the following sample, an object of the `knn_kernel` type is used as a distance function. You'll find its implementation in the samples provided in this book. This `knn_kernel` distance function object estimates the first KNN objects to the given one. These objects are determined with the KNN algorithm, which uses the Euclidean distance for the distance measure, as follows:

```
typedef matrix<double, 2, 1> sample_type;
typedef knn_kernel<sample_type> kernel_type;
...
std::vector<sample_type> samples;
...
std::vector<unsigned long> clusters =
spectral_cluster(kernel_type(samples, 15),
                  samples,
                  num_clusters);
```

The `spectral_cluster()` function call filled the `clusters` object with cluster index values, which we can use to visualize the clustering result, as illustrated in the following figure:

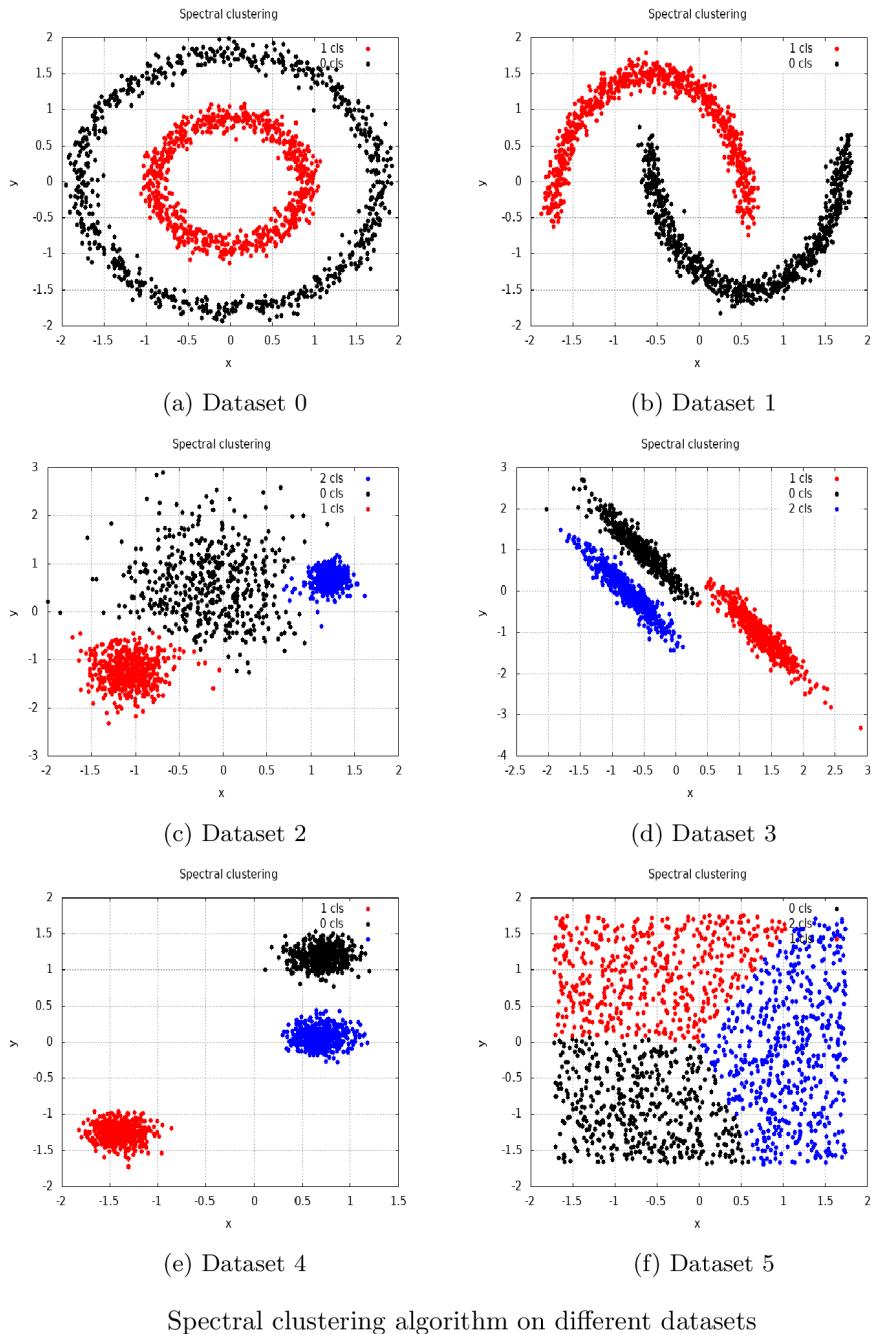


Figure 4.9 – Dlib spectral clustering visualization

In the preceding figure, we can see how the spectral clustering algorithm that's implemented in the Dlib library works on different artificial datasets.

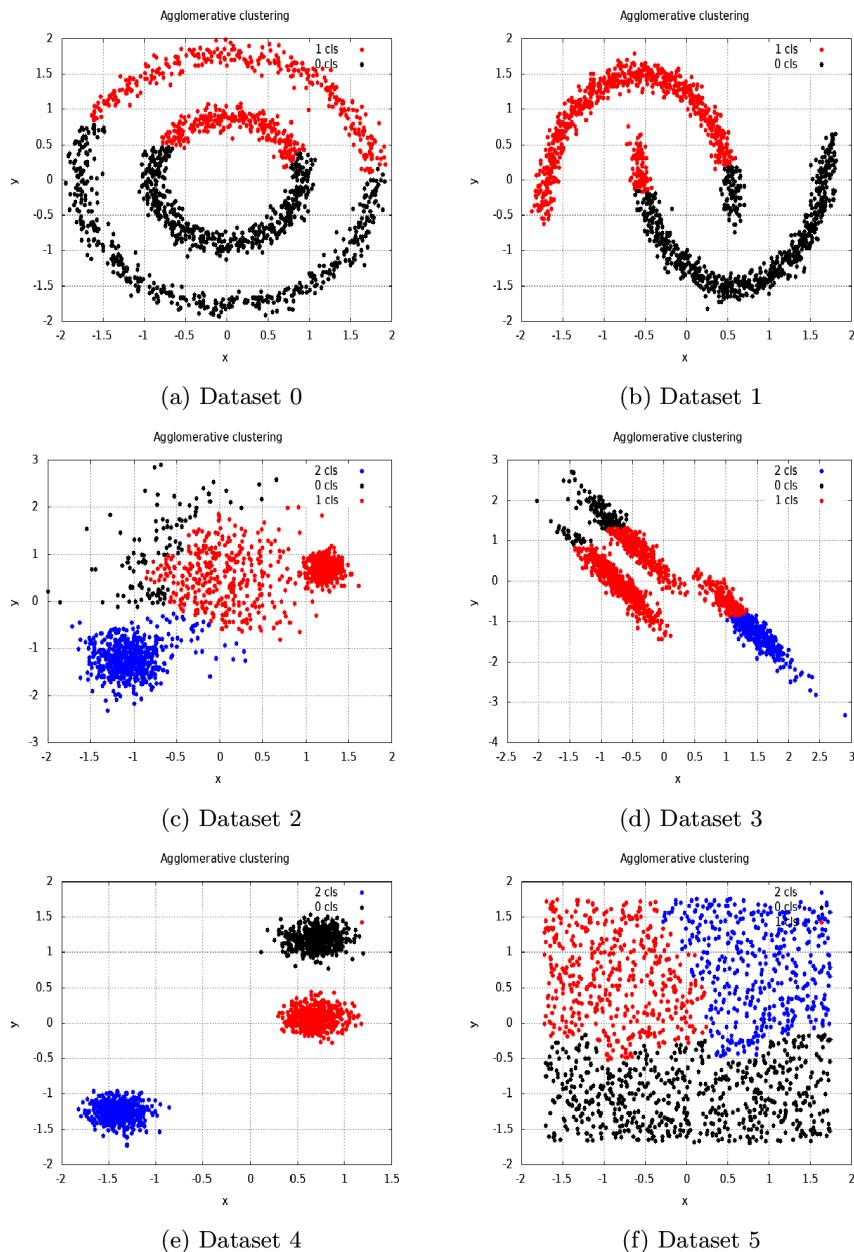
Hierarchical clustering with Dlib

The Dlib library implements the agglomerative hierarchical (bottom-up) clustering algorithm. The `bottom_up_cluster()` function implements this algorithm. This function takes the matrix of distances between dataset objects, the cluster indices container (as the output parameter), and the number of clusters as input parameters. Note that it returns the container with cluster indices in the order of distances provided in the matrix.

In the following code sample, we've filled the distance matrix with pairwise Euclidean distances between each pair of elements in the input dataset:

```
matrix<double> dists(inputs.nr(), inputs.nr());
for (long r = 0; r < dists.nr(); ++r) {
    for (long c = 0; c < dists.nc(); ++c) {
        dists(r, c) = length(subm(inputs, r, 0, 1, 2) -
                            subm(inputs, c, 0, 1, 2));
    }
}
std::vector<unsigned long> clusters;
bottom_up_cluster(dists, clusters, num_clusters);
```

The `bottom_up_cluster()` function call filled the `clusters` object with cluster index values, which we can use to visualize the clustering result, as illustrated in the following figure:



Hierarchical clustering algorithm on different datasets

Figure 4.10 – Dlib hierarchical clustering visualization

In the preceding figure, we can see how the hierarchical clustering algorithm that's implemented in the `Dlib` library works on different artificial datasets.

Newman modularity-based graph clustering algorithm with Dlib

The implementation of this algorithm is based on the work *Modularity and community structure in networks*, by M. E. J. Newman. This algorithm is based on the modularity matrix for a network or a graph and it isn't based on particular graph theory. However, it does have some similarities with spectral clustering because it also uses eigenvectors.

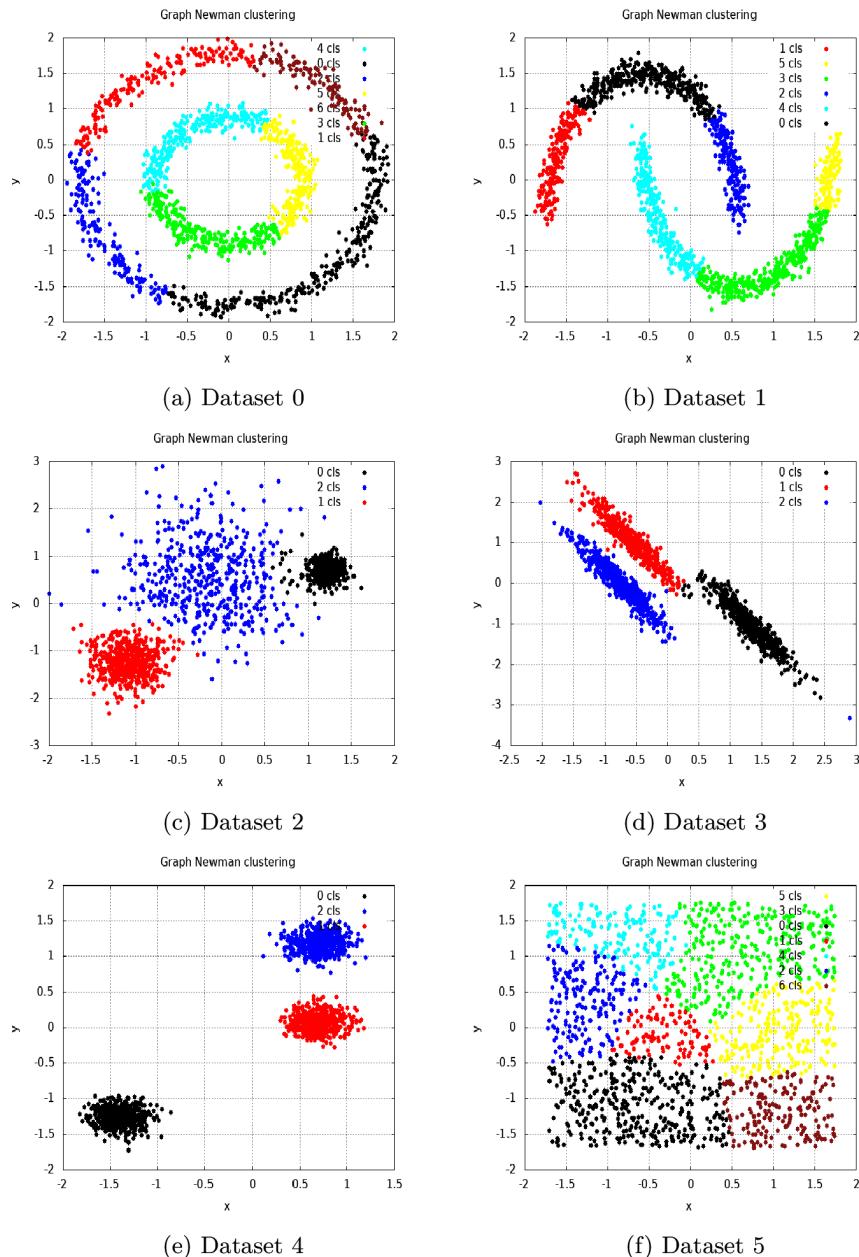
The `Dlib` library implements this algorithm in the `newman_cluster()` function, which takes a vector of weighted graph edges and outputs the container with cluster indices for each vertex. The vector of weighted graph edges represents the connections between nodes in the network, with each edge having a weight that indicates its strength. The weights are used to determine the similarity between nodes and thus influence the clustering process. The initial step for using this algorithm is to define graph edges. In the following code sample, we're making edges between almost every pair of dataset objects. Notice that we only use pairs with a distance greater than a threshold (this was done for performance considerations). The threshold distance can be adjusted to achieve different levels of granularity in the clustering results.

Also, this algorithm doesn't require prior knowledge of the number of clusters as it can determine the number of clusters by itself. Here's the code:

```
for (long i = 0; i < inputs.nr(); ++i) {
    for (long j = 0; j < inputs.nr(); ++j) {
        auto dist = length(subm(inputs, i, 0, 1, 2) -
                           subm(inputs, j, 0, 1, 2));
        if (dist < 0.5)
            edges.push_back(sample_pair(i, j, dist));
    }
}
remove_duplicate_edges(edges);
std::vector<unsigned long> clusters;
const auto num_clusters = newman_cluster(edges, clusters);
```

The `newman_cluster()` function call filled the `clusters` object with cluster index values, which we can use to visualize the clustering result. Notice that another approach for edge weight calculation can lead to another clustering result. Also, edge weight values should be initialized according to a certain task. The edge length was chosen only for demonstration purposes.

The result can be seen in the following figure:



Newman graph clustering algorithm on different datasets

Figure 4.11 – Dlib Newman clustering visualization

In the preceding figure, we can see how the Newman clustering algorithm that's implemented in the `Dlib` library works on different artificial datasets.

Chinese Whispers – graph clustering algorithm with Dlib

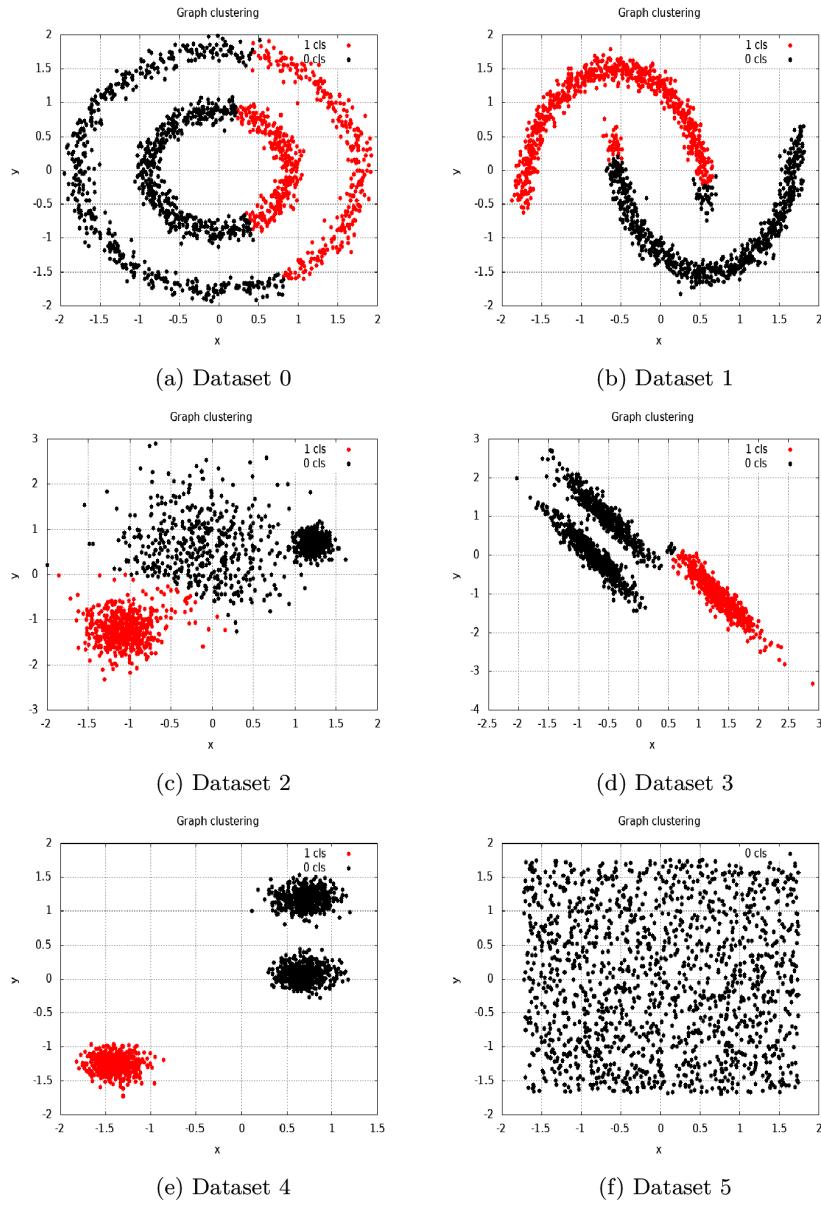
The Chinese Whispers algorithm is an algorithm that's used to partition the nodes of weighted, undirected graphs. It was described in the paper *Chinese Whispers – an Efficient Graph Clustering Algorithm and its Application to Natural Language Processing Problems*, by Chris Biemann. This algorithm also doesn't use any unique graph theory methods; instead, it uses the idea of using local contexts for clustering, so it can be classified as a density-based method.

In the `Dlib` library, this algorithm is implemented in the `chinese_whispers()` function, which takes the vector of weighted graph edges and outputs the container with cluster indices for each of the vertices. For performance considerations, we limit the number of edges between dataset objects with a threshold on distance. The meaning of weighted graph edges and threshold parameters are the same as for the Newman algorithm. Moreover, as with the Newman algorithm, this one also determines the number of resulting clusters by itself. The code can be seen in the following snippet:

```
std::vector<sample_pair> edges;
for (long i = 0; i < inputs.nr(); ++i) {
    for (long j = 0; j < inputs.nr(); ++j) {
        auto dist = length(subm(inputs, i, 0, 1, 2) -
                           subm(inputs, j, 0, 1, 2));
        if (dist < 1)
            edges.push_back(sample_pair(i, j, dist));
    }
}
std::vector<unsigned long> clusters;
const auto num_clusters = chinese_whispers(edges, clusters);
```

The `chinese_whispers()` function call filled the `clusters` object with cluster index values, which we can use to visualize the clustering result. Notice that we used 1 as the threshold for edge weights; another threshold value can lead to another clustering result. Also, edge weight values should be initialized according to a certain task. The edge length was chosen only for demonstration purposes.

The result can be seen in the following figure:



Chinese-Whispers graph clustering algorithm on different datasets

Figure 4.12 – Dlib Chinese Whispers clustering visualization

In the preceding figure, we can see how the Chinese Whispers clustering algorithm that's implemented in the `Dlib` library works on different artificial datasets.

In this and previous sections, we saw a lot of examples of images that show clustering results. The following section will explain how to use the `plotcpp` library, which we used to plot these images.

Plotting data with C++

After clustering, we plot the results with the `plotcpp` library, which is a thin wrapper around the `gnuplot` command-line utility. With this library, we can draw points on a scatter plot or draw lines. The initial step to start plotting with this library is creating an object of the `Plot` class. Then, we must specify the output destination of the drawing. We can set the destination with the `Plot::SetTerminal()` method, which takes a string with a destination point abbreviation. For example, we can use the `qt` string value to show the **operating system (OS)** window with our drawing or we can use a string with a picture file extension to save a drawing to a file, as in the code sample that follows. We can also configure the title of the drawing, the axis labels, and some other parameters with the `Plot` class methods. However, it does not cover all possible configurations available for `gnuplot`. In cases where we need some unique options, we can use the `Plot::gnuplotCommand()` method to make a direct `gnuplot` configuration.

There are two drawing approaches we can follow to draw a set of different graphics on one plot:

1. We can use the `Draw2D()` method with objects of the `Points` or `Lines` classes, but in this case, we should specify all graphics configurations before compilation.
2. We can use the `Plot::StartDraw2D()` method to get an intermediate drawing state object. Then, we can use the `Plot::AddDrawing()` method to add different drawings to one plot. The `Plot::EndDraw2D()` method should be called after we've drawn the last graphics.

We can use the `Points` type to draw points. An object of this type should be initialized with start and end forward iterators for the integral numeric data types, which represent coordinates. We should specify three iterators as points coordinates, two iterators for the `x` coordinates, which is where they start and end, and one iterator for the `y` coordinates' start. The number of coordinates in the containers should be the same. The last parameter is the `gnuplot` visual style configuration. Objects of the `Lines` class can be configured in the same way.

Once we've completed all drawing operations, we should call the `Plot::Flush()` method to render all commands to the window or the file, as shown in the following code block:

```
// Define helper data types for point clusters coordinates

// Container type for single coordinate values
using Coords = std::vector<DataType>;
// Paired x, y coordinate containers
using PointCoords = std::pair<Coords, Coords>;
```

```
// Clusters mapping container
using Clusters = std::unordered_map<index_t, PointCoords>;
```

```
// define color values container
const std::vector<std::string> colors{
    "black", "red",     "blue",   "green",
    "cyan",   "yellow", "brown", "magenta"};
...
```

```
// Function for clusters visualization
void
PlotClusters(const Clusters& clusters,
             const std::string& name,
             const std::string& file_name) {
// Instantiate plotting object
plotcpp::Plot plt;
// Configure plotting object
plt.SetTerminal("png");
plt.SetOutput(file_name);
pltSetTitle(name);
plt.SetXLabel("x");
plt.SetYLabel("y");
plt.SetAutoscale();
plt.gnuplotCommand("set grid");
// Start 2D scatter plot drawing
auto draw_state =
    plt.StartDraw2D<Coords::const_iterator>();
for (auto& cluster : clusters) {
    std::stringstream params;
    // Configure cluster visualization color string
    params << "lc rgb '" << colors[cluster.first]
        << "' pt 7";
    // Create cluster name string
    auto cluster_name =
        std::to_string(cluster.first) + " cls";
    // Create points visualization object using "cluster"
    // points
    plotcpp::Points points(cluster.second.first.begin(),
                           cluster.second.first.end(),
                           cluster.second.second.begin(),
                           cluster_name, params.str());
```

```
    // Add current cluster visualization to the 2D scatter
    // plot
```

```
    plt.AddDrawing(draw_state, points);
}
// Finalize 2D scatter plot
plt.EndDraw2D(draw_state);
// Render the plot
plt.Flush();
}
```

In this example, we learned how to plot our clustering results with the `plotcpp` library. We must be able to configure different visualization parameters, such as the type of the plot, point colors, and axes names as these parameters make our plot more informative. We also learned how to save this plot in a file so that we can use it later or insert it into another document. This library will be used throughout this book for visualizing results.

Summary

In this chapter, we considered what clustering is and how it differs from classification. We looked at different types of clustering methods, such as partition-based, spectral, hierarchical, density-based, and model-based methods. We also observed that partition-based methods can be divided into more categories, such as distance-based methods and graph theory-based methods.

Then, we used implementations of these algorithms, including the k-means algorithm (the distance-based method), the GMM algorithm (the model-based method), the Newman modularity-based algorithm, and the Chinese Whispers algorithm, for graph clustering. We also learned how to use the hierarchical and spectral clustering algorithm implementations in programs. We saw that the crucial issues for successful clustering include the choice of the distance measure function, the initialization step, the splitting or merging strategy, and prior knowledge of the number of clusters.

A combination of these issues is unique for each specific algorithm. We also saw that a clustering algorithm's results depend a lot on dataset characteristics and that we should choose the algorithm according to these.

At the end of this chapter, we studied how we can visualize clustering results with the `plotcpp` library.

In the next chapter, we'll learn what a data anomaly is and what machine learning algorithms exist for anomaly detection. We'll also see how anomaly detection algorithms can be used to solve real-life problems, and which properties of such algorithms play a more significant role in different tasks.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- *The 5 Clustering Algorithms Data Scientists Need to Know*: <https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68>
- *Clustering*: <https://scikit-learn.org/stable/modules/clustering.html>
- *Different Types of Clustering Algorithm*: <https://www.geeksforgeeks.org/different-types-clustering-algorithm/>
- *An introduction to clustering and different methods of clustering*: <https://www.analyticsvidhya.com/blog/2016/11/an-introduction-to-clustering-and-different-methods-of-clustering/>
- Graph theory introductory book: *Graph Theory (Graduate Texts in Mathematics)*, by Adrian Bondy and U.S.R. Murty
- *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, by Trevor Hastie, Robert Tibshirani, and Jerome Friedman, covers a lot of aspects of machine learning theory and algorithms

5

Anomaly Detection

Anomaly detection is where we search for unexpected values in a given dataset. An anomaly is a deviation in system behavior or data value from the standard or expected value. Anomalies are also known as outliers, errors, deviations, and exceptions. They can occur in data that's of a diverse nature and structure as a result of technical failures, accidents, deliberate hacks, and more.

There are many methods and algorithms we can use to search for anomalies in various types of data. These methods use different approaches to solve the same problem. There are unsupervised, supervised, and semi-supervised algorithms. However, in practice, unsupervised methods are the most popular. The **unsupervised anomaly detection** technique detects anomalies in unlabeled test datasets, under the assumption that most of the dataset is normal. It does this by searching for data points that are unlikely to fit the rest of the dataset. Unsupervised algorithms are more popular because of the nature of anomaly events, which are significantly rare compared to normal or expected data, so it is usually very difficult to get a suitably labeled dataset for anomaly detection.

Broadly speaking, anomaly detection applies to a wide range of areas, such as intrusion detection, fraud detection, fault detection, health monitoring, event detection (in sensor networks), and the detection of environmental disruptions. Often, anomaly detection is used as the preprocessing step for data preparation, before the data is passed on to other algorithms.

So, in this chapter, we'll discuss the most popular unsupervised algorithms for anomaly detection and its applications.

The following topics will be covered in this chapter:

- Exploring the applications of anomaly detection
- Learning approaches for anomaly detection
- Examples of using different C++ libraries for anomaly detection

Technical requirements

The list of software that you'll need to complete the examples in this chapter is as follows:

- Shogun-toolbox library
- Shark-ML library
- Dlib library
- PlotCpp library
- Modern C++ compiler with C++17 support
- CMake build system version ≥ 3.8

The code files for this chapter can be found at the following GitHub repo: <https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition/tree/main/Chapter05>

Exploring the applications of anomaly detection

Two areas in data analysis look for anomalies: **outlier detection** and **novelty detection**.

A *new object* or *novelty* is an object that differs in its properties from objects in the training dataset. Unlike an outlier, the new object is not in the dataset itself, but it can appear at any point after a system has started working. Its task is to detect when it appears. For example, if we were to analyze existing temperature measurements and identify abnormally high or low values, then we would be detecting outliers. On the other hand, if we were to create an algorithm that, for every new measurement, evaluates the temperature's similarity to past values and identifies significantly unusual ones, then we would be detecting novelties.

The reasons for outliers appearing include data errors, the presence of noise, misclassified objects, and foreign objects from other datasets or distributions. Let's explain two of the most obscure types of outliers: data errors and data from different distributions. Data errors can broadly refer to inaccuracies in measurements, rounding errors, and incorrect entries. An example of an object belonging to a different distribution is measurements that have come from a broken sensor. This is because these values will belong to a range that may be different from what was expected.

Novelties usually appear as a result of fundamentally new object behavior. For example, if our objects are computer system behavior descriptions, then after a virus has penetrated the computer and deleted some information from these descriptions, they will be rendered as novelties. Another example of a novelty could be a new group of customers that behave differently from others but have some similarities to other customers. The main feature of novelty objects is that they are new, in that it's impossible to have information about all possible virus infections or breakdowns in the training set. Creating such a training dataset is a complicated process and often does not make sense. However, fortunately, we can obtain a large enough dataset by focusing on the ordinary (regular) operations of the system or mechanism.

Often, the task of anomaly detection is similar to the task of **classification**, but there is an essential difference: **class imbalances**. For example, equipment failures (anomalies) are significantly rarer than having the equipment functioning normally.

We can observe anomalies in different kinds of data. In the following graph, we can see an example of anomalies in a numeric series:

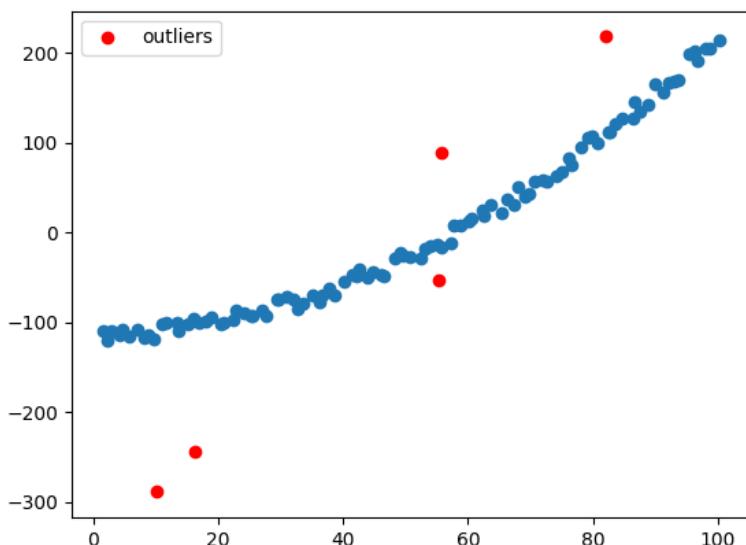


Figure 5.1 – Example of anomalies in a numeric series

In the following diagram, we can see anomalies in graphs; these anomalies can be as edges as well as vertices (see elements marked with a lighter color):

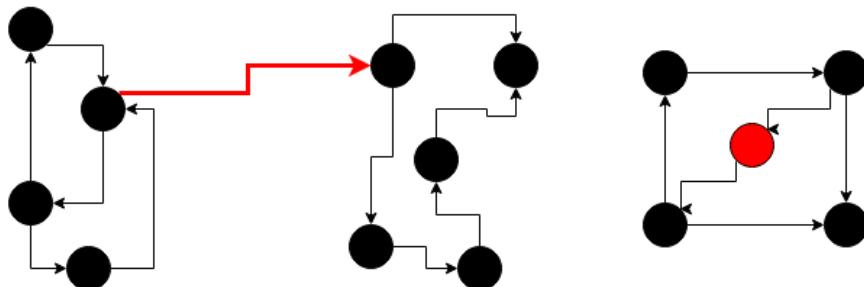


Figure 5.2 – Anomalies in graphs

The following text shows anomalies in a sequence of characters:

AABBCCCAABBCCCAACABBBCCAABB

The quality or performance of anomaly detection tasks can be estimated, just like classification tasks can, by using, for example, **Area Under the Receiver Operating Characteristic Curve (AUC-ROC)**.

We have discussed what anomalies are, so let's see what approaches there are to detect them.

Learning approaches for anomaly detection

In this section, we'll look at the most popular and straightforward methods we can use for anomaly detection.

Detecting anomalies with statistical tests

Statistical tests are usually used to catch extreme values for individual features. The general name for this type of test is **extreme-value analysis**. An example of such a test is the use of the Z-score measure:

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

Here, x_i is a sample from the dataset, μ is the mean of all samples from the dataset, and δ is the standard deviation of samples in the dataset. A Z-score value tells us how many standard deviations a data point is distant from the mean. So, by choosing the appropriate threshold value, we can filter some values as anomalies. Any data points with a Z-score greater than the threshold will be considered anomalies or unusual values in the dataset. Typically, values above 3 or below -3 are considered anomalies, but you can adjust this threshold based on your specific project requirements. The following graph shows which values from some type of normally distributed data can be treated as anomalies or outliers by using the Z-score test:

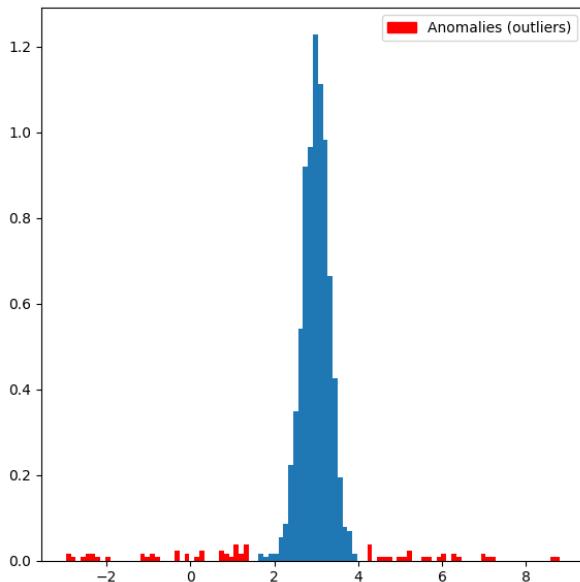


Figure 5.3 – Z-score anomaly detection

One important concept that we should mention is extreme values—the maximum and minimum values from the given dataset. It is important to understand that extreme values and anomalies are different concepts. The following is a small data sample:

```
[1, 39, 2, 1, 101, 2, 1, 100, 1, 3, 101, 1, 3, 100, 101, 100, 100]
```

We can consider the value 39 as an anomaly, but not because it is a maximal or minimal value. It is crucial to understand that an anomaly needn't be an extreme value.

Although extreme values are not anomalies in general, in some cases, we can adapt methods of extreme-value analysis to the needs of anomaly detection. However, this depends on the task at hand and should be carefully analyzed by **machine learning** (ML) practitioners.

Detecting anomalies with the Local Outlier Factor method

The distance measurement-based methods are widely used for solving different ML problems, as well as for anomaly detection. These methods assume that there is a specific metric in the object space that helps us find anomalies. The general assumption when we use distance-based methods for anomaly detection is that the anomaly only has a few neighbors, while a normal point has many. Therefore, for example, the distance to the k^{th} neighbor can serve as a good measure of anomalies, as reflected in the **Local Outlier Factor (LOF)** method. This method is based on estimating the density of objects that have been checked for anomalies. Objects lying in the areas of lowest density are considered anomalies or outliers.

The advantage of the LOF method over other methods is that it works in conjunction with the local density of objects. Therefore, the LOF effectively identifies outliers even when there are objects of different classes in the dataset that may not be considered anomalies during training. For example, let's assume that there is a distance, k -distance (A), from the object (A) to the k^{th} nearest neighbor. Note that the set of k nearest neighbors includes all objects within this distance. We denote the set of k nearest neighbors as $N_k(A)$. This distance is used to determine the reachability distance:

$$\text{reachability-distance}_k(A, B) = \max(\text{k-distance}(B), \text{dist}(A, B))$$

If point A lies among k neighbors of point B , then *reachability-distance* will be equal to the k -*distance* of point B . Otherwise, it will be equal to the exact distance between points A and B , which is given by the `dist` function. The local reachability density of an object A is defined as follows:

$$\text{ldr}_k(A) = 1 / \left(\frac{\sum_{B \in N_k(A)} \text{reachability-distance}_k(A, B)}{|N_k(A)|} \right)$$

Local reachability density is the inverse of the average reachability distance of the object, A , from its neighbors. Note that this is not the average reachability distance of neighbors from A (which, by definition, should have been k -distance(A)), but is the distance at which A can be reached from its neighbors. The local reachability densities are then compared with the local reachability densities of the neighbors:

$$\text{LOF}_k(A) = \left(\frac{\sum_{B \in N_k(A)} \text{ldr}_k(B)}{|N_k(A)|} \right) / \text{ldr}_k(A)$$

The provided formula gives the average local reachability density of the neighbors, divided by the local reachability density of the object itself:

- A value of approximately 1 means that the object can be compared with its neighbors (and therefore it is not an outlier)
- A value less than 1 indicates a dense area (objects have many neighbors)
- A value significantly larger than 1 indicates anomalies

The disadvantage of this method is the fact that the resulting values are difficult to interpret. A value of 1 or less indicates that a point is purely internal, but there is no clear rule by which a point will be an outlier. In one dataset, the value 1.1 may indicate an outlier. However, in another dataset with a different set of parameters (for example, if there is data with sharp local fluctuations), the value 2 may also indicate internal objects. These differences can also occur within a single dataset due to the locality of the method.

Detecting anomalies with isolation forest

The idea of an isolation forest is based on the **Monte Carlo principle**: a random partitioning of the feature space is carried out so that, on average, isolated points are cut off from normal ones. The final result is averaged over several runs of the stochastic algorithm, and the result will form an isolation forest of corresponding trees. The isolation tree algorithm then builds a random binary decision tree. The root of the tree is the whole feature space. In the next node, a random feature and a random partitioning threshold are selected, and they are sampled from a uniform distribution on the range of the minimum and maximum values of the selected feature. The isolation forest construction process ends when all objects in the node coincide identically. This is the stopping criterion. The mark of the leaves is the `anomaly_score` value of the algorithm, which is the depth of the leaves in the constructed tree. The following formula shows how the anomaly score can be calculated:

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}}$$

Here, $h(x)$ is the path length of the observation, \mathcal{X} , $E(h(x))$ is an average of $h(x)$ from a collection of isolation trees, $c(n)$ is the average path length of the unsuccessful search in a binary search tree, and n is the number of external nodes.

We're assuming that it is common for anomalies to appear in leaves with a low depth, which is close to the root, but for regular objects, the tree will build several more levels. The number of such levels is proportional to the size of the cluster. Consequently, `anomaly_score` is proportional to the points lying in it.

This assumption means that objects from clusters of small sizes (which are potentially anomalies) will have a lower `anomaly_score` value than those from clusters of regular data:

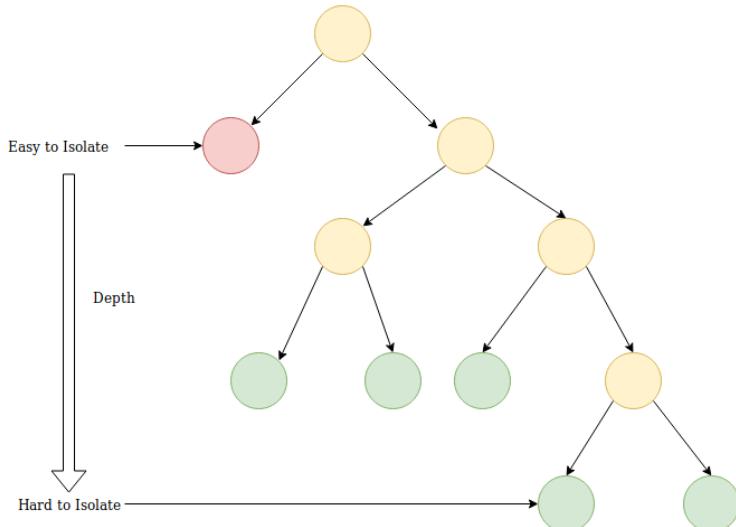


Figure 5.4 – Isolation forest visualization

The isolation forest method is widely used and implemented in various libraries.

Detecting anomalies with One-Class Support Vector Machine

The support vector method is a binary classification method based on using a **hyperplane** to divide objects into classes. The dimensions of the hyperplane are always chosen so that they're less than the dimensions of the original space. In \Re^3 , for example, a hyperplane is an ordinary two-dimensional plane. The distance from the hyperplane to each class should be as short as possible. The vectors that are closest to the separating hyperplane are called support vectors. In practice, cases where the data can be divided by a hyperplane—in other words, linear cases—are quite rare. In this case, all the elements of the training dataset are embedded in the higher dimension space, X , using a special mapping. In this case, the mapping is chosen so that in the new space, X , the dataset is linearly separable. Such mapping is based on kernel functions and is usually named the Kernel trick; it will be discussed more precisely in *Chapter 6*.

One-Class Support Vector Machine (OCSVM) is an adaptation of the support vector method that focuses on anomaly detection. OCSVM differs from the standard version of **Support Vector Machine (SVM)** in a way that the resulting optimization problem includes an improvement for determining a small percentage of predetermined anomalous values, which allows this method to be used to detect anomalies. These anomalous values lie between the starting point and the optimal separating hyperplane. All other data belonging to the same class falls on the opposite side of the optimal separating hyperplane.

There's also another type of OCSVM method that uses a spherical, instead of a planar (or linear), approach. The algorithm obtains a spherical boundary, in the feature space, around the data. The volume of this hypersphere is minimized to reduce the effect of incorporating outliers in the solution.

A **spherical mapping** is appropriate when the data has a spherical shape, such as when the data points are distributed evenly around the origin. A **planar (or linear) mapping** is more appropriate when the data has a planar shape, such as when the data points lie on a line or plane. Also, other kernel functions can be used to map the data into a higher-dimensional space where it is linearly separable. The choice of kernel function depends on the nature of the data.

OCSVM assigns a label, which is the distance from the test data point to the optimal hyperplane. Positive values in the OCSVM output represent normal behavior (with higher values representing greater normality), while negative values represent anomalous behavior (the lower the value, the more significant the anomaly).

In order to assign a label, the OCSVM first trains on a dataset that consists of only normal or expected behavior. This dataset is called the **positive class**. The OCSVM then tries to find a hyperplane that maximizes the distance between the positive class and the origin. This hyperplane is called the **decision boundary**.

Once the decision boundary has been found, any new data point that falls outside of this boundary is considered an anomaly or outlier. The OCSVM assigns a label of **anomaly** to these data points.

Density estimation approach

One of the most popular approaches to anomaly detection is density estimation, which involves estimating the probability distribution of normal data and then flagging observations as anomalies if they fall outside the expected range. The basic idea behind density estimation is to fit a model to the data that represents the underlying distribution of normal behavior. This model can be a simple parametric distribution such as **Gaussian** or a more complex non-parametric model such as **kernel density estimation (KDE)**. Once the model is trained on normal data, it can be used to estimate the density of new observations. Observations with low density are considered anomalies.

There are several advantages to using a density estimation approach:

- It is flexible and can handle a wide range of data types
- It does not require labeled data for training, making it suitable for **unsupervised learning (UL)**
- It can detect both point anomalies (observations that are significantly different from the rest) and contextual anomalies (observations that do not follow the normal pattern within a specific context)

However, there are also some challenges with this approach:

- The choice of model and parameters can affect the performance of the algorithm
- Outliers can influence the estimated density, leading to false positives
- The algorithm may not be able to detect anomalies that are not well represented in the training data

Overall, density estimation is a powerful tool for anomaly detection that can be customized to suit the specific needs of an application. By carefully selecting the model and tuning the parameters, it is possible to achieve high accuracy and precision in detecting anomalies.

Using multivariate Gaussian distribution for anomaly detection

Let's assume we have some samples $x^{(i)}$ in a dataset and that they are labeled and normally distributed (Gaussian distribution). In such a case, we can use distribution properties to detect anomalies. Let's assume that the function $p(x)$ gives us the probability of a sample being normal. A high probability corresponds to a regular sample, while a low probability corresponds to an anomaly. We can, therefore, choose thresholds to distinguish between regular values and anomalies with the following **anomaly model formula**:

$$\begin{aligned} p(x_{test}) < \epsilon, & \text{ flag as an outlier or an anomaly} \\ p(x_{test}) \geq \epsilon, & \text{ flag as a normal or a non-anomalous} \end{aligned}$$

If $[x \in \mathbb{R}]$ and x follows the Gaussian distribution with the mean, μ , and the variance, σ^2 , it is denoted as follows:

$$x \sim \mathcal{N}(\mu, \sigma^2)$$

The following formula gives the probability of \mathbf{x} in a Gaussian distribution:

$$p(\mathbf{x}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\mathbf{x} - \mu)^2}{2\sigma^2}\right)$$

Here, $\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}$ is the mean and $\sigma^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)^2$ is the variance (σ is the standard deviation). This formula is known as parametrized **probability density function (PDF)**, and it describes the relative likelihood of different outcomes in a continuous random variable. It is used to model the distribution of a random variable and provides information about the probability of observing a specific value or range of values for that variable.

Next, we'll introduce an example of the general approach we follow for anomaly detection with Gaussian distribution density estimation:

1. Let's say we're given a new example, $\mathbf{x} = \{x_1, x_2 \dots x_n\}$
2. Select the features, x_i , that are regular, meaning they determine anomalous behavior.
3. Fit the μ_i and σ_i^2 parameters.
4. Compute $p(\mathbf{x}) = \prod_{j=1}^n p(x_j; \mu_j, \sigma_j^2)$ using an equation to calculate the probability of \mathbf{x} in a Gaussian distribution.
5. Determine if \mathbf{x} is an anomaly by comparing it with the threshold, ϵ ; see the anomaly model formula.

The following graph shows an example of Gaussian distribution density estimation for normally distributed data:

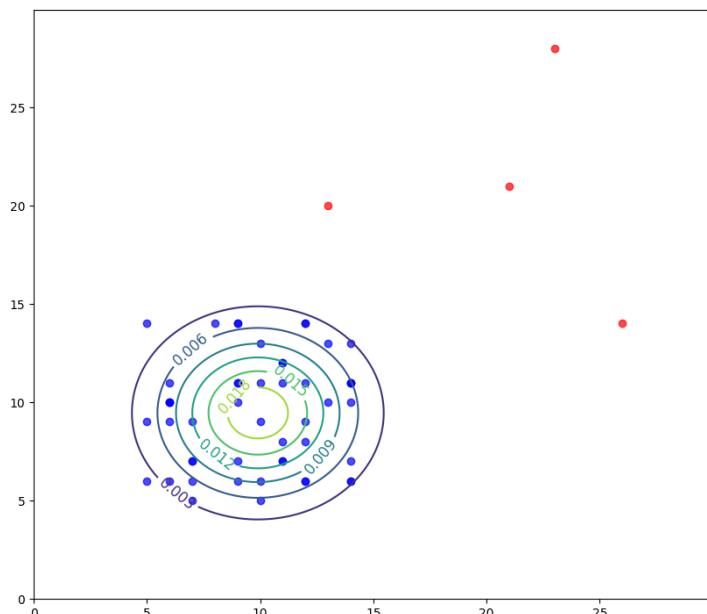


Figure 5.5 – Gaussian density estimation visualization

In this approach, we assume that selected features are independent, but usually, in real data, there are some correlations between them. In such a case, we should use a multivariate Gaussian distribution model instead of a univariate one.

The following formula gives the probability of x in a multivariate Gaussian distribution:

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} \exp \left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu) \right)$$

Here, μ is the mean, Σ is the correlation matrix, and $|\Sigma|$ is the determinant of the matrix, Σ :

$$\Sigma = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)(x^{(i)} - \mu)^T$$

The following graphs shows the difference between the univariate and the multivariate Gaussian distribution estimation models for a dataset with correlated data. Notice how distribution boundaries cover the regular data with a darker color, while anomalies are marked with a lighter color:

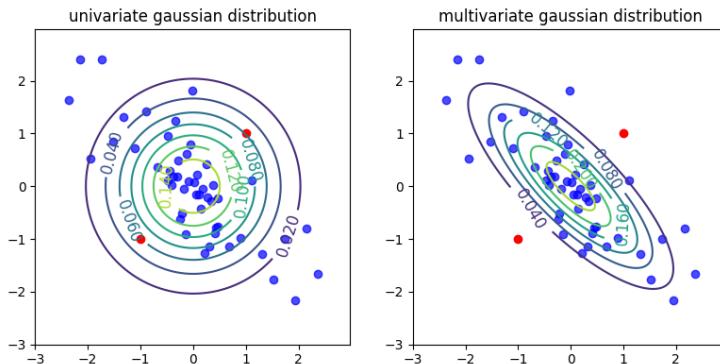


Figure 5.6 – Univariate and multivariate Gaussian distributions

We can see that the multivariate Gaussian distribution can take into account correlations in the data and adapt its shape to them. This characteristic allows us to detect anomalies correctly for types of data whose distribution follows a Gaussian (normal) distribution shape. Also, we can see one of the advantages of this approach: results can be easily visualized in two or three dimensions, providing a clear understanding of the data.

In the next section, we will look at another approach for anomaly detection.

KDE

In the KDE approach, our goal is to approximate a complex mixture of random distributions around point samples into a single function. So, the main idea is to center a probability distribution function at each data point and then take their average. It means that each discrete point in our dataset is replaced by an extended probability distribution, called a **kernel**. The probability density at any given point is

then estimated as the sum of the kernel functions centered at each discrete point. If the point is close to many other points its estimated probability density will be larger than if it is far away from any sample point. This approach can be used to find anomalies as points where the estimated density is minimal.

Mathematically, we can formulate the KDE function for univariate kernels as follows:

$$KDE(x) = \frac{1}{N} \sum_{i=1}^N K_h(x - x_i)$$

Here, K_h is a smoothed kernel defined with the following formula:

$$K_h(u) = \frac{1}{h} K\left(\frac{u}{h}\right)$$

Here, h is the bandwidth parameter that defines the width of a kernel. The bandwidth parameter controls the degree of smoothness or roughness of the kernel function. A larger bandwidth results in a smoother function, while a smaller bandwidth leads to a more rugged one. Choosing the right bandwidth is crucial for achieving good generalization performance.

K can be, for example, a Gaussian kernel function, which is the most common one:

$$K(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$$

The following graph shows a plot of KDE of Gaussian distribution made from individual kernels:

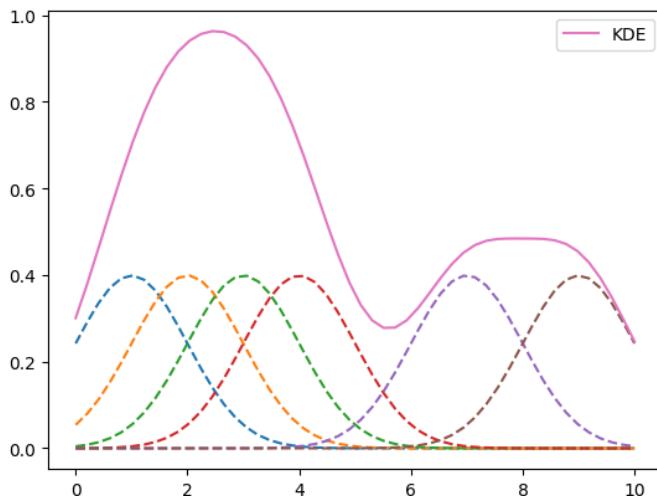


Figure 5.7 – KDE of Gaussian distribution

You can see in this graph that the density value will be the maximum for points around the value 3 because most sample points are located in this region. Another cluster of points is located around point 8, but their density is quite a bit smaller.

Before training the KDE model, it is important to preprocess the data to ensure that it is suitable for the KDE algorithm. This may include scaling the data so that all features have similar ranges, removing outliers or anomalies, and transforming the data if necessary.

Handling high-dimensional data efficiently in KDE can be challenging due to the **curse of dimensionality (CoD)**. One approach is to use dimensionality reduction techniques such as **principal component analysis (PCA)** or **t-distributed stochastic neighbor embedding (t-SNE)** to reduce the number of dimensions before applying KDE. Another approach is to use sparse KDE algorithms that only consider a subset of data points when calculating the density estimate for a given location.

In the following section, we will look at another approach for anomaly detection.

Density estimation trees

The **density estimation tree (DET)** algorithm also can be used to detect anomalies by thresholding the density value for certain sample points. This is a non-parametric technique based on decision tree construction. The main advantage of this algorithm is the fast analytical complexity of density estimation at any given point, its $O(\log n)$ time, where n is the number of points in the tree. The tree is constructed iteratively in a top-to-bottom approach. Each leaf, t , is divided into two sub-leaves t_l and t_r by maximizing residual gain s that is defined as follows:

$$s = R(t) - R(t_l) - R(t_r)$$

Here, $R(t)$ is the tree loss function:

$$R(t) = \frac{N_t^2}{N_{total}^2 V_t}$$

N is the number of candidates the t leaf contains, and V is the leaf's volume. Then, the actual density for a leaf t can be calculated as follows:

$$D(t) = \frac{N_t}{N_{total} V_t}$$

So, to estimate a density value for a given point, we have to determine to which leaf it belongs and then get the leaf's density.

In the current section, we discussed various anomaly detection approaches, and in the following sections, we will see how to use various C++ libraries to deal with the anomaly detection task.

Examples of using different C++ libraries for anomaly detection

In this section, we'll look at some examples of how to implement the algorithms we described previously for anomaly detection.

C++ implementation of the isolation forest algorithm for anomaly detection

Isolation forest algorithms can be easily implemented in pure C++ because their logic is pretty straightforward. Also, there are no implementations of this algorithm in popular C++ libraries. Let's assume that our implementation will only be used with two-dimensional data. We are going to detect anomalies in a range of samples where each sample contains the same number of features.

Because our dataset is large enough, we can define a wrapper for the actual data container. This allows us to reduce the number of copy operations we perform on the actual data:

```
using DataType = double;
template <size_t Cols>
using Sample = std::array<DataType, Cols>;
template <size_t Cols>
using Dataset = std::vector<Sample<Cols>>;
...
template <size_t Cols>
struct DatasetRange {
    DatasetRange(std::vector<size_t>&& indices,
                 const Dataset<Cols>* dataset)
        : indices(std::move(indices)), dataset(dataset) {}
    size_t size() const { return indices.size(); }
    DataType at(size_t row, size_t col) const {
        return (*dataset)[indices[row]][col];
    }
    std::vector<size_t> indices;
    const Dataset<Cols>* dataset;
};
```

The `DatasetRange` type holds a reference to the vector of `Sample` type objects and to the container of indices that point to the samples in the dataset. These indices define the exact dataset objects that this `DatasetRange` object points to.

Next, we define the elements of the isolation tree, with the first one being the `Node` type:

```
struct Node {
    Node() {}
    Node(const Node&) = delete;
    Node& operator=(const Node&) = delete;
    Node(std::unique_ptr<Node> left,
         std::unique_ptr<Node> right, size_t split_col,
         DataType split_value)
        : left(std::move(left)),
          right(std::move(right)),
```

```
    split_col(split_col),
    split_value(split_value) {}
Node(size_t size) : size(size), is_external(true) {}

    std::unique_ptr<Node> left;
    std::unique_ptr<Node> right;
    size_t split_col{0};
    DataType split_value{0};
    size_t size{0};
    bool is_external{false};
};
```

This type is a regular tree-node structure. The following members are specific to the isolation tree algorithm:

- `split_col`: This is the index of the feature column where the algorithm caused a split
- `split_value`: This is the value of the feature where the algorithm caused a split
- `size`: This is the number of underlying items for the node
- `is_external`: This is the flag that indicates whether the node is a leaf

Taking the `Node` type as a basis, we can define the procedure of building an isolation tree. We aggregate this procedure with the auxiliary `IsolationTree` type. Because the current algorithm is based on random splits, the auxiliary data is the random engine object.

We only need to initialize this object once, and then it will be shared among all tree-type objects. This approach allows us to make the results of the algorithm reproducible in the case of constant seeding. Furthermore, it makes debugging the randomized algorithm much simpler:

```
template <size_t Cols>
class IsolationTree {
public:
    using Data = DatasetRange<Cols>;
    IsolationTree(const IsolationTree&) = delete;
    IsolationTree& operator=(const IsolationTree&) = delete;
    IsolationTree(std::mt19937* rand_engine, Data data, size_t hlim)
        : rand_engine(rand_engine) {
        root = MakeIsolationTree(data, 0, hlim);
    }
    IsolationTree(IsolationTree&& tree) {
        rand_engine = std::move(tree.rand_engine);
        root = std::move(tree.root);
    }
    double PathLength(const Sample<Cols>& sample) {
```

```

        return PathLength(sample, root.get(), 0);
    }

private:
    std::unique_ptr<Node> MakeIsolationTree(const Data& data,
                                              size_t height,
                                              size_t hlim);
    double PathLength(const Sample<Cols>& sample,
                      const Node* node, double height);

private:
    std::mt19937* rand_engine;
    std::unique_ptr<Node> root;
} ;

```

Next, we'll do the most critical work in the `MakeIsolationTree()` method, which is used in the constructor to initialize the root data member:

```

std::unique_ptr<Node> MakeIsolationTree(const Data& data,
                                         size_t height, size_t hlim) {
    auto len = data.size();
    if (height >= hlim || len <= 1) {
        return std::make_unique<Node>(len);
    } else {
        std::uniform_int_distribution<size_t> cols_dist(0, Cols - 1);
        auto rand_col = cols_dist(*rand_engine);
        std::unordered_set<DataType> values;
        for (size_t i = 0; i < len; ++i) {
            auto value = data.at(i, rand_col);
            values.insert(value);
        }
        auto min_max = std::minmax_element(values.begin(), values.end());
        std::uniform_real_distribution<DataType> value_dist(
            *min_max.first, *min_max.second);
        auto split_value = value_dist(*rand_engine);

        std::vector<size_t> indices_left;
        std::vector<size_t> indices_right;
        for (size_t i = 0; i < len; ++i) {
            auto value = data.at(i, rand_col);
            if (value < split_value) {
                indices_left.push_back(data.indices[i]);
            } else {
                indices_right.push_back(data.indices[i]);
            }
        }
    }
}

```

```

        }
    }

    return std::make_unique<Node>(
        MakeIsolationTree(
            Data{std::move(indices_left), data.dataset},
            height + 1, hlim),
        MakeIsolationTree(
            Data{std::move(indices_right), data.dataset},
            height + 1, hlim),
        rand_col, split_value);
    }
}

```

Initially, we checked the termination conditions to stop the splitting process. If we meet them, we return a new node marked as an external leaf. Otherwise, we start splitting the passed data range. For splitting, we randomly select the `feature` column and determine the unique values of the selected feature. Then, we randomly select a value from an interval between the `max` and `min` values among the feature values from all the samples. After we make these random selections, we compare the values of the selected splitting feature to all the samples from the input data range and put their indices into two lists. One list is for values higher than the splitting values, while another list is for values that are lower than them. Then, we return a new tree node initialized with references to the left and right nodes, which are initialized with recursive calls to the `MakeIsolationTree()` method.

Another vital method of the `IsolationTree` type is the `PathLength()` method. We use it for anomaly score calculations. It takes the sample as an input parameter and returns the amortized path length to the corresponding tree leaf from the root node:

```

double PathLength(const Sample<Cols>& sample, const Node* node,
                  double height) {
    assert(node != nullptr);
    if (node->is_external) {
        return height + CalcC(node->size);
    } else {
        auto col = node->split_col;
        if (sample[col] < node->split_value) {
            return PathLength(sample, node->left.get(), height + 1);
        } else {
            return PathLength(sample, node->right.get(), height + 1);
        }
    }
}

```

The `PathLength()` method finds the leaf node during tree traversal based on sample feature values. These values are used to select a tree traversal direction based on the current node-splitting values. During each step, this method also increases the resulting height. The result of this method is a sum of the actual tree traversal height and the value returned from the call to the `CalcC()` function, which then returns the average path's length of unsuccessful searches in a binary search tree of equal height to the leaf node. The `CalcC()` function can be implemented in the following way, according to the formula from the original paper, which describes the isolation forest algorithm (you can find a reference to this in the *Further reading* section):

```
double CalcC(size_t n) {
    double c = 0;
    if (n > 1)
        c = 2 * (log(n - 1) + 0.5772156649) - (
            2 * (n - 1) / n);
    return c;
}
```

The final part of the algorithm's implementation is the creation of the forest. The forest is an array of trees built from a limited number of samples, randomly chosen from the original dataset. The number of samples used to build the tree is a hyperparameter of this algorithm. Furthermore, this implementation uses heuristics as the stopping criteria, in that it is a maximum tree height `hlim` value.

Let's see how it is used in the tree-building procedure. The `hlim` value is calculated only once, and the following code shows this. Moreover, it is based on the number of samples that are used to build a single tree:

```
template <size_t Cols>
class IsolationForest {
public:
    using Data = DatasetRange<Cols>;
    IsolationForest(const IsolationForest&) = delete;
    IsolationForest& operator=(const IsolationForest&) = delete;
    IsolationForest(const Dataset<Cols>& dataset,
                    size_t num_trees, size_t sample_size)
        : rand_engine(2325) {
        std::vector<size_t> indices(dataset.size());
        std::iota(indices.begin(), indices.end(), 0);

        size_t hlim = static_cast<size_t>(ceil(log2(sample_size)));
        for (size_t i = 0; i < num_trees; ++i) {
            std::vector<size_t> sample_indices;
            std::sample(indices.begin(), indices.end(),
                        std::back_insert_iterator(sample_indices),
                        sample_size, rand_engine);
            trees.emplace_back(
                std::move(sample_indices), hlim));
    }
};
```

```

        &rand_engine,
        Data(std::move(sample_indices), &dataset), hlim);
    }
    double n = dataset.size();
    c = CalcC(n);
}

double AnomalyScore(const Sample<Cols>& sample) {
    double avg_path_length = 0;
    for (auto& tree : trees) {
        avg_path_length += tree.PathLength(sample);
    }
    avg_path_length /= trees.size();
    double anomaly_score = pow(2, -avg_path_length / c);
    return anomaly_score;
}

private:
    std::mt19937 rand_engine;
    std::vector<IsolationTree<Cols>> trees;
    double c{0};
};

}

```

The tree forest is built in the constructor of the `IsolationForest` type. We also calculated the value of the average path length of the unsuccessful search in a binary search tree for all of the samples in the constructor. We use this forest in the `AnomalyScore()` method for the actual process of anomaly detection. It implements the formula for the anomaly score value for a given sample. It returns a value that can be interpreted in the following way: if the returned value is close to 1, then the sample has anomalous features, while if the value is less than 0.5, then we can assume that the sample is a normal one.

The following code shows how we can use this algorithm. Furthermore, it uses Dlib primitives for the dataset's representation:

```

void IsolationForest(const Matrix& normal, const Matrix& test) {
    iforest::Dataset<2> dataset;
    auto put_to_dataset = [&](const Matrix& samples) {
        for (long r = 0; r < samples.nr(); ++r) {
            auto row = dlib::rowm(samples, r);
            double x = row(0, 0);
            double y = row(0, 1);
            dataset.push_back({x, y});
        }
    };
}

```

```

put_to_dataset(normal);
put_to_dataset(test);
iforest::IsolationForest iforest(dataset, 300, 50);
double threshold = 0.6; // change this value to see isolation
                       //boundary
for (auto& s : dataset) {
    auto anomaly_score = iforest.AnomalyScore(s);
    // std::cout << anomaly_score << " " << s[0] << " " << s[1]
    // << std::endl;
    if (anomaly_score < threshold) {
        // Do something with normal
    } else {
        // Do something with anomalies
    }
}
}

```

In the preceding example, we converted and merged the given datasets for the container that's suitable for our algorithm. Then, we initialized the object of the `IsolationForest` type, which immediately builds the isolation forest with the following hyperparameters: the number of trees is 100, and the number of samples used for one tree is 50.

Finally, we called the `AnomalyScore()` method for each sample from the dataset in order to detect anomalies with thresholds and return their values. In the following graph, we can see the result of anomaly detection after using the isolation forest algorithm. The points labeled as `1 cls` are the anomalies:

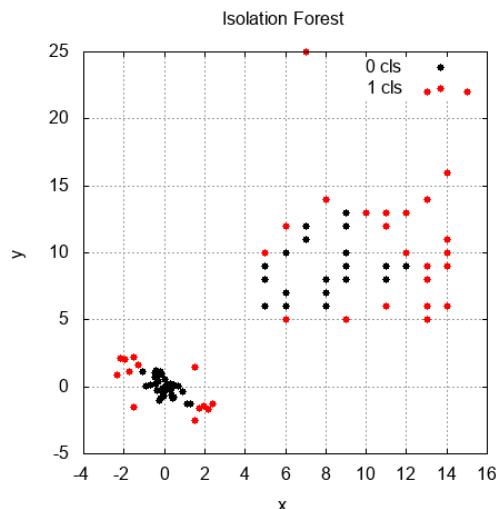


Figure 5.8 – Anomaly detection with isolation forest algorithm

In this section, we learned how to implement the isolation forest algorithm from scratch. The following section will show you how to use the `Dlib` library for anomaly detection.

Using the `Dlib` library for anomaly detection

The `Dlib` library provides a couple of implemented algorithms that we can use for anomaly detection: the OCSVM model and the multivariate Gaussian model.

OCSVM with `Dlib`

There is only one algorithm that's implemented in the `Dlib` library straight out of the box: OCSVM. There is a `svm_one_class_trainer` class in this library that can be used to train the corresponding algorithm, which should be configured with a kernel object, and the `nu` parameter, which controls the smoothness (in other words, the degree to which it controls the ratio between generalization and overfitting) of the solution.

The most widely used kernel is based on the Gaussian distribution and is known as the **Radial Basis Function (RBF)** kernel. It is implemented in the `radial_basis_kernel` class. Typically, we represent datasets in the `Dlib` library as a C++ vector of separate samples. Therefore before using this `trainer` object, we have to convert a matrix dataset into a vector:

```
void OneClassSvm(const Matrix& normal, const Matrix& test) {
    typedef matrix<double, 0, 1> sample_type;
    typedef radial_basis_kernel<sample_type> kernel_type;
    svm_one_class_trainer<kernel_type> trainer;
    trainer.set_nu(0.5); // control smoothness of the solution
    trainer.set_kernel(kernel_type(0.5)); // kernel bandwidth
    std::vector<sample_type> samples;
    for (long r = 0; r < normal.nr(); ++r) {
        auto row = rowm(normal, r);
        samples.push_back(row);
    }
    decision_function<kernel_type> df = trainer.train(samples);
    Clusters clusters;
    double dist_threshold = -2.0;
    auto detect = [&](auto samples) {
        for (long r = 0; r < samples.nr(); ++r) {
            auto row = dlib::rowm(samples, r);
            auto dist = df(row);
            if (p > dist_threshold) {
                // Do something with anomalies
            } else {
                // Do something with normal
            }
        }
    }
}
```

```

    }
};

detect(normal);
detect(test);
}

```

The result of the training process is a decision function object of the `decision_function<kernel_type>` class that we can use for single sample classification. Objects of this type can be used as a regular function. The result of a decision function is the distance from the normal class boundary, so the most distant samples can be classified as anomalies. The following graph shows an example of how the OCSVM algorithm from the `Dlib` library works. Note that the dots labeled as `1 cls` correspond to anomalies:

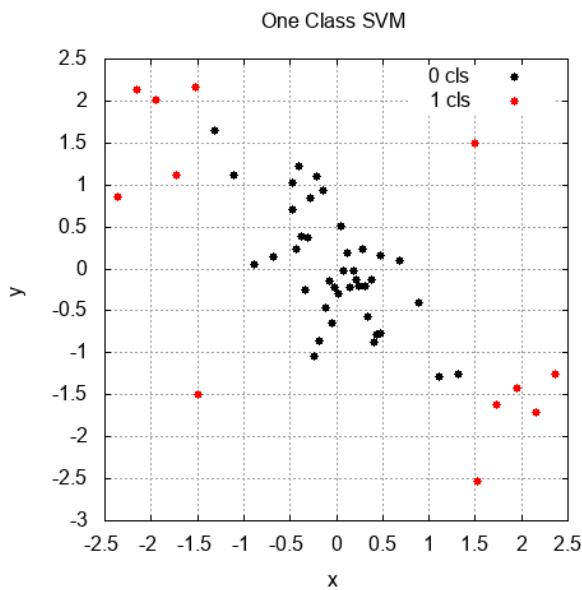


Figure 5.9 – Anomaly detection with Dlib OCSVM implementation

We can see that OCSVM solved the task well and detected anomalies that are very interpretable. In the next section, we will see how to use a multivariate Gaussian model to detect anomalies.

Multivariate Gaussian model with Dlib

Using the linear algebra facilities of the Dlib library (or any other library, for that matter), we can implement anomaly detection with the multivariate Gaussian distribution approach. The following example shows how to implement this approach with Dlib linear algebra routines:

```
void multivariateGaussianDist(const Matrix& normal,
                               const Matrix& test) {
    // assume that rows are samples and columns are features
    // calculate per feature mean
    dlib::matrix<double> mu(1, normal.nc());
    dlib::set_all_elements(mu, 0);

    for (long c = 0; c < normal.nc(); ++c) {
        auto col_mean = dlib::mean(dlib::colm(normal, c));
        dlib::set_colm(mu, c) = col_mean;
    }

    // calculate covariance matrix
    dlib::matrix<double> cov(normal.nc(), normal.nc());
    dlib::set_all_elements(cov, 0);
    for (long r = 0; r < normal.nr(); ++r) {
        auto row = dlib::rowm(normal, r);
        cov += dlib::trans(row - mu) * (row - mu);
    }
    cov *= 1.0 / normal.nr();
    double cov_det = dlib::det(cov); // matrix determinant
    dlib::matrix<double> cov_inv = dlib::inv(cov); // inverse matrix

    // define probability function
    auto first_part = 1. / std::pow(2. * M_PI, normal.nc() / 2.) /
                      std::sqrt(cov_det);
    auto prob = [&](const dlib::matrix<double>& sample) {
        dlib::matrix<double> s = sample - mu;
        dlib::matrix<double> exp_val_m =
            s * (cov_inv * dlib::trans(s));
        double exp_val = -0.5 * exp_val_m(0, 0);
        double p = first_part * std::exp(exp_val);
        return p;
    };
    // change this parameter to see the decision boundary
    double prob_threshold = 0.001;
    auto detect = [&](auto samples) {
        for (long r = 0; r < samples.nr(); ++r) {
```

```
    auto row = dlib::rowm(samples, r);
    auto p = prob(row);
    if (p >= prob_threshold) {
        // Do something with anomalies
    } else {
        // Do something with normal
    }
}
};

detect(normal);
detect(test);
}
```

The idea of this approach is to define a function that returns the probability of appearing, given a sample in a dataset. To implement such a function, we calculate the statistical characteristics of the training dataset. In the first step, we calculate the mean values of each feature and store them in the one-dimensional matrix. Then, we calculate the covariance matrix for the training samples using the formula for the correlation matrix that was given in the prior theoretical section named *Density estimation approach*. Next, we determine the correlation matrix determinant and inverse version. We define a lambda function named `prob` to calculate the probability of a single sample using the formula provided in the *Using multivariate Gaussian distribution* section.

For large datasets, the computational complexity of calculating the covariance matrix can become a significant factor in the overall runtime of an ML model. Furthermore, optimizing the calculation of the covariance matrix for large datasets requires a combination of techniques, including sparsity, parallelization, approximation, and efficient algorithms.

We also define a probability threshold to separate anomalies. It determines the boundary between normal and anomalous behavior, and it plays a crucial role in the classification of samples as anomalies or normal. Engineers must carefully consider their application's requirements and adjust the threshold accordingly to achieve the desired level of sensitivity. For example, in security applications where false alarms are costly, a higher threshold might be preferred to minimize false positives. In contrast, in medical diagnoses where missing a potential anomaly could have serious consequences, a lower threshold might be more appropriate to ensure that no true anomalies go undetected.

Then, we iterate over all the examples (including the training and testing datasets) to find out how the algorithm separates regular samples from anomalies. In the following graph, we can see the result of this separation. The dots labeled as `1 cls` are anomalies:

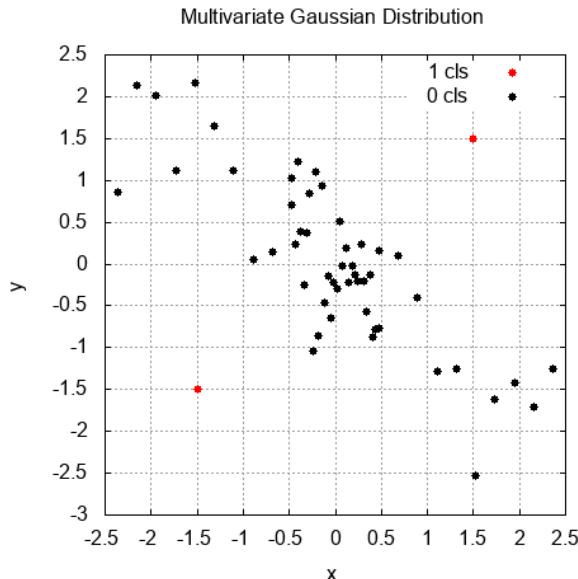


Figure 5.10 – Anomaly detection with Dlib multivariate Gaussian distribution

We see that this approach found fewer anomalies than the previous one, so you should be aware that some methods will not work well with your data and it will make sense to try different methods. In the following section, we will see how to use a multivariate Gaussian model from the `mlpack` library for the same task.

Multivariate Gaussian model with `mlpack`

We already discussed in the previous chapter the `GMM` and the `EMFit` classes that exist in the `mlpack` library. The **Expectation-Maximization with Fit (EMFit)** algorithm is an ML technique used to estimate the parameters of a **gaussian mixture model (GMM)**. It works by iteratively optimizing the parameters to fit the data. We can use them not only for solving clustering tasks but also for anomaly detection. There will only be one difference: we have to specify only one cluster for the training. So, `GMM` class initialization will look like the following:

```
GMM gmm(/*gaussians*/ 1, /*dimensionality*/ 2);
```

The initializations for the `KMeans` and the `EMFit` algorithms will be the same as in the previous example:

```
KMeans<> kmeans;
size_t max_iterations = 250;
double tolerance = 1e-10;
EMFit<KMeans<>, NoConstraint> em(max_iterations,
                                         tolerance,
                                         kmeans);
```

```

gmm.Train(normal,
/*trials*/ 3,
/*use_existing_model*/ false,
em);

```

The values for `max_iterations` and the convergence tolerance variables influence the training process by determining how long the algorithm runs and when it stops. A higher number of trials may lead to more accurate results but also increase computational time. The convergence tolerance determines how close the parameters must be to their previous values before the algorithm stops. If the tolerance is too low, the algorithm may never converge, while if it is too high, it may converge to a suboptimal solution.

Then, we can use the `Probability` method of the `gmm` object to test some new data. The only new action we have to take is to define a probability threshold that will be used to check if a new sample belongs to the original data distribution or if it's an anomaly. This can be done as follows:

```
double prob_threshold = 0.001;
```

Having this threshold, the usage of the `Probability` method will be the following:

```

auto detect = [&](const arma::mat& samples) {
    for (size_t c = 0; c < samples.n_cols; ++c) {
        auto sample = samples.col(c);
        double x = sample.at(0, 0);
        double y = sample.at(1, 0);
        auto p = gmm.Probability(sample);
        if (p >= prob_threshold) {
            plot_clusters[0].first.push_back(x);
            plot_clusters[0].second.push_back(y);
        } else {
            plot_clusters[1].first.push_back(x);
            plot_clusters[1].second.push_back(y);
        }
    }
};

```

Here, we defined a lambda function that can be applied to any dataset defined as a matrix. In this function, we have used a simple loop over all samples in which we applied the `Probability` method for a single sample and compared the returned value with the threshold. If the probability value is too low, we mark the sample as an anomaly by adding its coordinates to the `plotting_cluster[1]` object. To plot the anomaly detection result, we used the same approach as was described in the previous chapter. The following code shows how to use the function we defined:

```

arma::mat normal;
arma::mat test;

```

```
Clusters plot_clusters;
detect(normal);
detect(test);
PlotClusters(plot_clusters, "Density Estimation Tree", file_name);
```

We applied the `detect` function to both sets of data: the `normal` one that was used for the training and the new one, `test`. You can see the anomaly detection result in the following graph:

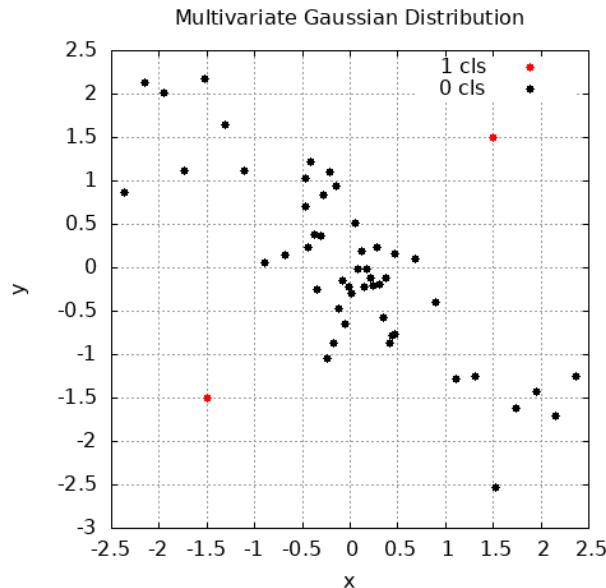


Figure 5.11 – Anomaly detection with mlpack multivariate Gaussian distribution

The two outliers were detected. You can try to change the probability threshold to see how the decision boundary will be changed and what objects will be classified as anomalies.

In the following section, we will see how to use the KDE algorithm implementation from the `mlpack` library.

KDE with mlpack

The KDE algorithm in the `mlpack` library is implemented in the `KDE` class. This class can be specialized with several template parameters; the most important ones are `KernelType`, `MetricType`, and `TreeType`. Let's use the Gaussian kernel, the Euclidean distance as the metric, and KD-Tree for the tree type. A tree data structure is used to optimize the algorithm's computational complexity. For each query point, the algorithm will apply a kernel function to each reference point, so the computational complexity can be $O(N^2)$ in the naive implementation for N query points and N reference points. The tree optimization avoids many similar calculations, because kernel function values decrease with

distance, but it also introduces some level of approximation. The following code snippet shows how to define a KDE object for our sample:

```
using namespace mlpack;
...
KDE<GaussianKernel,
    EuclideanDistance,
    arma::mat,
    KDTree>
kde(/*rel error*/ 0.0, /*abs error*/ 0.01, GaussianKernel());
```

Due to the approximations used in the algorithm, the API allows us to define relative and absolute error tolerances. Relative and absolute error tolerances control the level of approximation in the KDE estimate. A higher tolerance allows for more approximation, which can reduce computational complexity but also decrease accuracy. Conversely, a lower tolerance requires more computation but can result in a more accurate estimate.

The **relative error tolerance** parameter specifies the maximum relative error allowed between the true density and the estimated density at any point. It is used to determine the optimal bandwidth for the kernel.

The **absolute error tolerance** parameter sets the maximum absolute error allowed between the true density and the estimated density over the entire domain. It can be used to ensure that the estimated density is within a certain range of the true density.

For our sample, we defined only the absolute one. The next step is to train our algorithm object with the normal data without anomalies; it can be done as follows:

```
arma::mat normal;
...
kde.Train(normal);
```

You can see that different algorithms in the `mlpack` library use mostly the same API. Then, we can define a function to classify the given data as normal or as an anomaly. The following code shows its definition:

```
double density_threshold = 0.1;
Clusters plot_clusters;
auto detect = [&](const arma::mat& samples) {
    arma::vec estimations;
    kde.Evaluate(samples, estimations);

    for (size_t c = 0; c < samples.n_cols; ++c) {
```

```
auto sample = samples.col(c);
double x = sample.at(0, 0);
double y = sample.at(1, 0);
auto p = estimations.at(c);
if (p >= density_threshold) {
    plot_clusters[0].first.push_back(x);
    plot_clusters[0].second.push_back(y);
} else {
    plot_clusters[1].first.push_back(x);
    plot_clusters[1].second.push_back(y);
}
}
};
```

We defined a lambda function that takes the data matrix and passes it to the `Evaluate` method of the `kde` object. This method evaluated and assigned density value estimation for every sample in the given data matrix. Then, we just compared those estimations with the `density_threshold` value to decide if the sample was normal or an anomaly. Samples with low-density values were classified as anomalies.

To select an optimal threshold, you need to balance the trade-off between sensitivity and specificity based on your specific use case. If you prioritize detecting all anomalies, you may want to set a lower threshold to increase the number of true positives, even if it means accepting more false positives. Conversely, if you prioritize minimizing false alarms, you might choose a higher threshold, which could miss some anomalies but reduce the number of false positives. In practice, selecting an optimal density threshold often involves experimenting with different values and evaluating the results using metrics such as precision, recall, and F1 score. Additionally, domain knowledge and expert input can help guide the selection process.

This function also prepares data for plotting in the same manner as we did earlier. The following code shows how to plot two datasets with normal and outlier data:

```
arma::mat normal;
arma::mat test;
detect(normal);
detect(test);
PlotClusters(plot_clusters, "Density Estimation Tree", file_name);
```

You can see the anomaly detection result in the following graph:

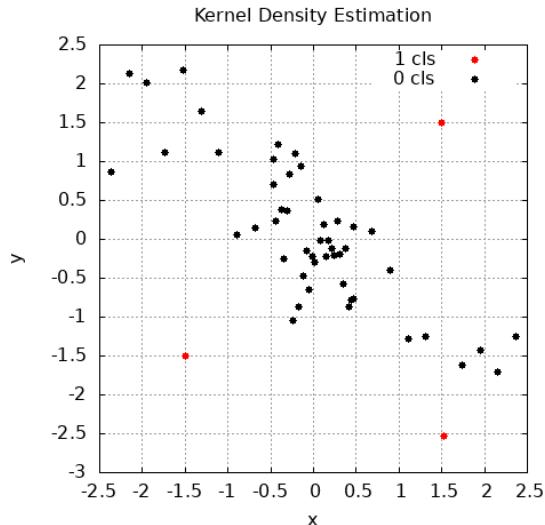


Figure 5.12 – Anomaly detection with KDE from mlpack

We can see that using the KDE method, we can find the two outliers, as we detected in the previous sections. The points labeled with `1 cls` are outliers. By changing the density threshold, you can see how the decision boundary will be changed.

In the following section, we will see how to use the DET algorithm implementation from the `mlpack` library.

DET with mlpack

The DET method from the `mlpack` library is implemented in the `DTree` class. To start working with it, we have to make a copy of the training normal data because an object of the `DTree` class will change the input data order. The data ordering is changed because `mlpack` creates a tree data structure directly on the given data object. The following code snippet shows how to define such an object:

```
arma::mat data_copy = normal;
DTree<> det(data_copy);
```

Due to the input data reordering the algorithms, the API is required to provide a mapping of the indices that will show the relation of new indices to the old ones. Such mapping can be initialized as follows:

```
arma::Col<size_t> data_indices(data_copy.n_cols);
for (size_t i = 0; i < data_copy.n_cols; i++) {
    data_indices[i] = i;
}
```

Here, we just stored the original relation of the data indices to the input data. Later, this mapping will be updated by the algorithm. Now, can build the DET by calling the `Grow` method, as follows:

```
size_t max_leaf_size = 5;
size_t min_leaf_size = 1;
det.Grow(data_copy, data_indices, false, max_leaf_size,
          min_leaf_size);
```

The two main parameters of the `Grow` method are `max_leaf_size` and `min_leaf_size`. They should be turned manually after a series of experiments or with some prior knowledge of the dataset characteristics. It can be tricky to estimate those parameters with automated techniques such as cross-validation because, for anomaly detection tasks, we don't usually have enough data marked as anomalies. So, the values for this example were chosen manually.

Having an initialized DET, we can use the `ComputeValue` method to estimate a density for some given data sample. If we choose a density threshold value, we can detect anomalies just by comparison with this value. We used the same approach in other algorithms too. The following code snippet shows how to use a threshold to distinguish between normal and anomalous data and build a data structure for result plotting:

```
double density_threshold = 0.01;
Clusters plot_clusters;
auto detect = [&] (const arma::mat& samples) {
    for (size_t c = 0; c < samples.n_cols; ++c) {
        auto sample = samples.col(c);
        double x = sample.at(0, 0);
        double y = sample.at(1, 0);
        auto p = det.ComputeValue(sample);
        if (p >= density_threshold) {
            plot_clusters[0].first.push_back(x);
            plot_clusters[0].second.push_back(y);
        } else {
            plot_clusters[1].first.push_back(x);
            plot_clusters[1].second.push_back(y);
        }
    }
};
```

We defined a `detect` function that simply iterates over the input data matrix columns and applies the `ComputeValue` method for every given sample to get the density estimate. Then, this function compares a value with `density_threshold`, and if the density is big enough, puts the sample in the first plotting cluster. Otherwise, the sample comes to the second plotting cluster. We can apply this function as follows:

```
detect(normal);
detect(test);
PlotClusters(plot_clusters, "Density Estimation Tree", file_name);
```

Here, `normal` and `test` are matrices with the normal and anomalous data samples, correspondingly. The following graph shows the detection result plotting:

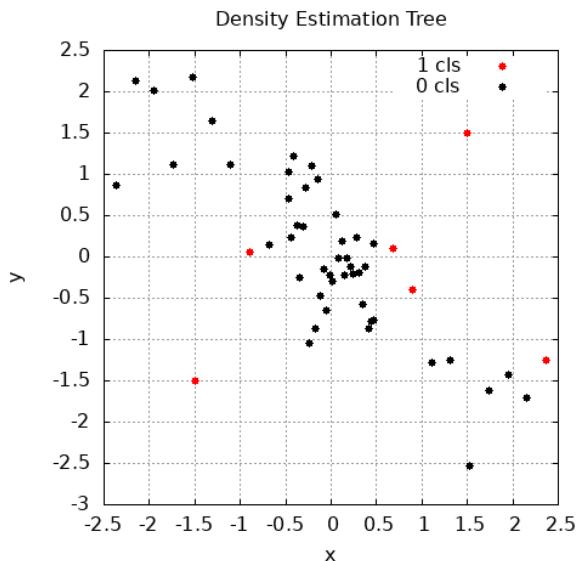


Figure 5.13 – Anomaly detection with DET algorithm

You may notice that this method classified more data points as anomalies compared to previous methods. To change this detection result, you can play with three parameters: the density threshold and leaf `min` and `max` sizes. This method with more complex turning may be useful for datasets where you don't know the data distribution rule (the kernel form) or it's hard to write code for it. The same goes for when you have normal data with several clusters with different distributions.

Summary

In this chapter, we examined anomalies in data. We discussed several approaches to anomaly detection and looked at two kinds of anomalies: outliers and novelties. We considered the fact that anomaly detection is primarily a UL problem, but despite this, some algorithms require labeled data, while others are semi-supervised. The reason for this is that, generally, there is a tiny number of positive examples (that is, anomalous samples) and a large number of negative examples (that is, standard samples) in anomaly detection tasks.

In other words, we don't usually have enough positive samples to train algorithms. That is why some solutions use labeled data to improve algorithm generalization and precision. On the contrary, **supervised learning (SL)** usually requires a large number of positive and negative examples, and their distribution needs to be balanced.

Also, notice that the task of detecting anomalies does not have a single formulation and that it is often interpreted differently, depending on the nature of the data and the goal of the concrete task. Moreover, choosing the correct anomaly detection method depends primarily on the task, data, and available *a priori* information. We also learned that different libraries can give slightly different results, even for the same algorithms.

In the following chapter, we will discuss dimension reduction methods. Such methods help us to reduce the dimensionality of data with high dimensionality into a new representation of data with lower dimensionality while preserving essential information from the original data.

Further reading

- Anomaly detection learning resources: <https://github.com/yzhao062/anomaly-detection-resources>
- *Outlier Detection with One-Class SVMs: An Application to Melanoma Prognosis*: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3041295/>
- Isolation forest: <https://feitonyliu.files.wordpress.com/2009/07/liu-iforest.pdf>
- Ram, Parikshit & Gray, Alexander. (2011). *Density estimation trees*. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 627-635. 10.1145/2020408.2020507: https://www.researchgate.net/publication/221654618_Density_estimation_trees
- Tutorial on KDE: https://faculty.washington.edu/yen chic/18W_425/Lec6_hist_KDE.pdf

6

Dimensionality Reduction

In this chapter, we'll go through a number of dimension-reduction tasks. We'll look at the conditions in which dimension-reduction is required and learn how to use dimension-reduction algorithms efficiently in C++ with various libraries. Dimensionality reduction involves transforming high-dimensional data into a new representation with fewer dimensions while preserving the most crucial information from the original data. Such a transformation can help us visualize multidimensional space, which can be useful in the data exploration stage or when identifying the most relevant features in dataset samples. Some **machine learning** (ML) techniques can perform better or faster if our data has a smaller number of features since it can consume fewer computational resources. The main purpose of this kind of transformation is to save the essential features—those features that hold the most critical information present in the original data.

The following topics will be covered in this chapter:

- An overview of dimension-reduction methods
- Exploring linear methods for dimension-reduction
- Exploring non-linear methods for dimension-reduction
- Understanding dimension-reduction algorithms with various C++ libraries

Technical requirements

The technologies you'll need for this chapter are as follows:

- The Tapkee library
- The Dlib library
- The plotcpp library
- A modern C++ compiler with C++20 support
- The CMake build system, version ≥ 3.24

The code files for this chapter can be found at the following GitHub repository: <https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition/tree/main/Chapter06>.

An overview of dimension-reduction methods

The main goal of dimension-reduction methods is to make the dimension of the transformed representation correspond with the internal dimension of the data. In other words, it should be similar to the minimum number of variables necessary to express all the possible properties of the data. Reducing the dimension helps mitigate the impact of the curse of dimensionality and other undesirable properties that occur in high-dimensional spaces. As a result, reducing dimensionality can effectively solve problems regarding classification, visualization, and compressing high-dimensional data. It makes sense to apply dimensionality reduction only when particular data is redundant; otherwise, we can lose important information. In other words, if we are able to solve the problem using data of smaller dimensions with the same level of efficiency and accuracy, then some of our data is redundant. Dimensionality reduction allows us to reduce the time and computational costs of solving a problem. It also makes data and the results of data analysis easier to interpret.

It makes sense to reduce the number of features when the information that can be used to solve the problem at hand qualitatively is contained in a specific subset of features. Non-informative features are a source of additional noise and affect the accuracy of the model parameter's estimation. In addition, datasets with a large number of features can contain groups of correlated variables. The presence of such feature groups leads to the duplication of information, which may distort the model's results and affect how well it estimates the values of its parameters.

The methods surrounding dimensionality reduction are mainly unsupervised because we don't know which features or variables can be excluded from the original dataset without losing the most crucial information.

Some real-life examples of dimensionality reduction include the following:

- In recommender systems, dimensionality reduction can be used to represent users and items as vectors in a lower-dimensional space, making it easier to find similar users or items
- In image recognition, dimensionality reduction techniques, such as **principal component analysis (PCA)**, can be applied to reduce the size of images while preserving their important features
- In text analysis, dimensionality reduction can be employed to transform large collections of documents into lower-dimensional representations that capture the main topics discussed in the documents

Dimensionality reduction methods can be classified into two groups: feature selection and the creation of new low-dimensional features. These methods can then be subdivided into **linear** and **non-linear** approaches, depending on the nature of the data and the mathematical apparatus being used.

Feature selection methods

Feature selection methods don't change the initial values of the variables or features; instead, they remove the irrelevant features from the source dataset. Some of the feature selection methods we can use are as follows:

- **Missing value ratio:** This method is based on the idea that a feature that misses many values should be eliminated from a dataset because it doesn't contain valuable information and can distort the model's performance results. So, if we have some criteria for identifying missing values, we can calculate their ratio to typical values and set a threshold that we can use to eliminate features with a high missing value ratio.
- **Low variance filter:** This method is used to remove features with low variance because such features don't contain enough information to improve model performance. To apply this method, we need to calculate the variance for each feature, sort them in ascending order by this value, and leave only those with the highest variance values.
- **High correlation filter:** This method is based on the idea that if two features have a high correlation, then they carry similar information. Also, highly correlated features can significantly reduce the performance of some ML models, such as linear and logistic regression. Therefore, the primary goal of this method is to leave only the features that have a high correlation with target values and don't have much correlation with each other.
- **Random forest:** This method can be used for feature selection effectively (although it wasn't initially designed for this kind of task). After we've built the forest, we can estimate what features are most important by estimating the impurity factor in the tree's nodes. This factor shows the measure of split distinctness in the tree's nodes, and it demonstrates how well the current feature (a random tree only uses one feature in a node to split input data) splits data into two distinct buckets. Then, this estimation can be averaged across all the trees in the forest. Features that split data better than others can be selected as the most important ones.
- **Backward feature elimination and forward feature selection:** These are iterative methods that are used for feature selection. In backward feature elimination, after we've trained the model with a full feature set and estimated its performance, we remove its features one by one and train the model with a reduced feature set. Then, we compare the model's performances and decide how much performance is improved by removing feature changes—in other words, we're deciding how important each feature is. In forward feature selection, the training process goes in the opposite direction. We start with one feature and then add more of them. These methods are very computationally expensive and can only be used on small datasets.

Dimensionality reduction methods

Dimensionality reduction methods transform an original feature set into a new feature set that usually contains new features that weren't present in the initial dataset. These methods can also be divided into two subclasses—linear and non-linear. The non-linear methods are usually more computationally expensive, so if we have a prior assumption about our feature's data linearity, we can choose the more suitable class of methods at the initial stage.

The following sections will describe the various linear and non-linear methods we can use for dimension-reduction.

Exploring linear methods for dimension-reduction

In this section, we will describe the most popular linear methods that are used for dimension-reduction, such as the following:

- PCA
- **Singular value decomposition (SVD)**
- **Independent component analysis (ICA)**
- **Linear discriminant analysis (LDA)**
- Factor analysis
- **Multidimensional scaling (MDS)**

Principal component analysis

PCA is one of the most intuitively simple and frequently used methods for applying dimension-reduction to data and projecting it onto an orthogonal subspace of features. In a very general form, it can be represented as the assumption that all our observations look like some ellipsoid in the subspace of our original space. Our new basis in this space coincides with the axes of this ellipsoid. This assumption allows us to get rid of strongly correlated features simultaneously since the basis vectors of the space we project them onto are orthogonal.

The dimension of this ellipsoid is equal to the dimension of the original space, but our assumption that the data lies in a subspace of a smaller dimension allows us to discard the other subspaces in the new projection; namely, the subspace with the least extension of the ellipsoid. We can do this greedily, choosing a new element one by one on the basis of our new subspace, and then taking the axis of the ellipsoid with maximum dispersion successively from the remaining dimensions.

To reduce the dimension of our data from n to $k, k \leq n$, we need to choose the top k axes of such an ellipsoid, sorted in descending order by dispersion along the axes. To begin with, we calculate the variances and covariances of the original features. This is done by using a **covariance matrix**. By the definition of covariance, for two signs, X_i and X_j , their covariance should be as follows:

$$\text{cov}(X_i, X_j) = E[(X_i - \mu_i)(X_j - \mu_j)] = E[X_i X_j] - \mu_i \mu_j$$

Here, μ_i is the mean of the i^{th} feature.

In this case, we note that the covariance is symmetric and that the covariance of the vector itself is equal to its dispersion. Thus, the covariance matrix is a symmetric matrix where the dispersions of the corresponding features lie on the diagonal and the covariances of the corresponding pairs of features lie outside the diagonal. In the matrix view, where X is the observation matrix, our covariance matrix looks like this:

$$\Sigma = E[(\mathbf{X} - E[\mathbf{X}])(\mathbf{X} - E[\mathbf{X}])^T]$$

The covariance matrix is a generalization of variance in the case of multidimensional random variables—it also describes the shape (spread) of a random variable, as does the variance. Matrices, such as linear operators, have eigenvalues and eigenvectors. They are interesting because when we act on the corresponding linear space or transform it with our matrix, the eigenvectors remain in place, and they are only multiplied by the corresponding eigenvalues. This means they define a subspace that remains in place or *goes into itself* when we apply a linear operator matrix to it. Formally, an eigenvector, w_i , with an eigenvalue for a matrix is defined simply as $Mw_i = \lambda_i w_i$.

The covariance matrix for our sample, \mathbf{x} , can be represented as a product, $\mathbf{x}^T \mathbf{x}$. From the Rayleigh relation, it follows that the maximum variation of our dataset can be achieved along the eigenvector of this matrix, which corresponds to the maximum eigenvalue. This is also true for projections on a higher number of dimensions—the variance (covariance matrix) of the projection onto the m -dimensional space is maximum in the direction of m eigenvectors with maximum eigenvalues. Thus, the principal components that we would like to project our data for are simply the eigenvectors of the corresponding top k pieces of the eigenvalues of this matrix.

The largest vector has a direction similar to the regression line, and by projecting our sample onto it, we lose information, similar to the sum of the residual members of the regression. It is necessary to make the operation, $v^T x$ (the vector length (magnitude) should be equal to one), perform the projection. If we don't have a single vector and have a hyperplane instead, then instead of the vector, v^T , we take the matrix of basis vectors, V^T . The resulting vector (or matrix) is an array of projections of our observations; that is, we need to multiply our data matrix on the basis vectors matrix, and we get the projection of our data orthogonally. Now, if we multiply the transpose of our data matrix and the matrix of the principal component vectors, we restore the original sample in the space where we projected it onto the basis of the principal components. If the number of components is less than the dimension of the original space, we lose some information.

Singular value decomposition

SVD is an important method that's used to analyze data. The resulting matrix decomposition has a meaningful interpretation from an ML point of view. It can also be used to calculate PCA. SVD is rather slow. Therefore, when the matrices are too large, randomized algorithms are used. However, the SVD calculation is computationally more efficient than the calculation for the covariance matrix and its eigenvalues in the original PCA approach. Therefore, PCA is often implemented in terms of SVD. Let's take a look.

The essence of SVD is straightforward—any matrix (real or complex) is represented as a product of three matrices:

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^*$$

Here, \mathbf{U} is a unitary matrix of order m and Σ is a matrix of size $m \times n$ on the main diagonal, which is where there are non-negative numbers called singular values (elements outside the main diagonal are zero—such matrices are sometimes called rectangular diagonal matrices). \mathbf{V}^* is a Hermitian-conjugate \mathbf{v} matrix of order n . The m columns of the matrices \mathbf{U} and n columns of the matrix \mathbf{V} are called the left and right singular vectors of matrix \mathbf{x} , respectively. To reduce the number of dimensions, matrix Σ is important, the elements of which, when raised to the second power, can be interpreted as a variance that each component puts into a joint distribution, and they are in descending order: $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{noise}$. Therefore, when we choose the number of components in SVD (as in PCA), we should take the sum of their variances into account.

The relation between SVD and PCA can be described in the following way: \mathbf{C} is the covariance matrix given by $\mathbf{C} = \mathbf{X}^\top \mathbf{X} / (n - 1)$. It is a symmetric matrix, so it can be diagonalized as $\mathbf{C} = \mathbf{V}\mathbf{L}\mathbf{V}^\top$, where \mathbf{v} is a matrix of eigenvectors (each column is an eigenvector) and \mathbf{L} is a diagonal matrix of eigenvalues, λ_i , in decreasing order on the diagonal. The eigenvectors are called principal axes or principal directions of the data. Projections of the data on the principal axes are called **principal components**, also known as **principal component scores**. They are newly transformed variables. The j^{th} principal component is given by the j^{th} column of $\mathbf{x}\mathbf{v}$. The coordinates of the i^{th} data point in the new principal component's space are given by the i^{th} row of $\mathbf{x}\mathbf{v}$.

By performing SVD on \mathbf{x} , we get $\mathbf{x} = \mathbf{u}\mathbf{s}\mathbf{v}^\top$, where \mathbf{u} is a unitary matrix and \mathbf{S} is the diagonal matrix of singular values, s_i . We can observe that $\mathbf{c} = \mathbf{v}\mathbf{s}\mathbf{u}^\top \mathbf{u}\mathbf{s}\mathbf{v}^\top / (n - 1) = \mathbf{v} \frac{\mathbf{s}^2}{n - 1} \mathbf{v}^\top$, which means that the right singular vectors, \mathbf{v} , are principal directions and that singular values are related to the eigenvalues of the covariance matrix via $\lambda_i = s_i^2 / (n - 1)$. Principal components are given by $\mathbf{x}\mathbf{v} = \mathbf{u}\mathbf{s}\mathbf{v}^\top \mathbf{v} = \mathbf{u}\mathbf{s}$.

Independent component analysis

The **ICA** method was proposed as a way to solve the problem of **blind signal separation (BSS)**; that is, selecting independent signals from mixed data. Let's look at an example of the task of BSS. Suppose we have two people in the same room who are talking and generating acoustic waves. We have two microphones in different parts of the room, recording sound. The analysis system receives two signals

from the two microphones, each of which is a digitized mixture of two acoustic waves—one from people speaking and one from some other noise (for example, playing music). Our goal is to select our initial signals from the incoming mixtures. Mathematically, the problem can be described as follows. We represent the incoming mixture in the form of a linear combination, where \mathbf{a} represents the displacement coefficients and \mathbf{s} represents the values of the vector of independent components:

$$\mathbf{x}_i(t) = \sum_{j=1}^n a_{i,j} s_j(t), (i = 1, \dots, m)$$

In matrix form, this can be expressed as follows:

$$\mathbf{x} = \mathbf{As}$$

Here, we have to find the following:

$$\mathbf{y} = \mathbf{A}^{-1}\mathbf{x} = \mathbf{Wx}.$$

In this equation, X is a matrix of input signal values, A is a matrix of displacement coefficients or mixing matrix, and S is a matrix of independent components. Thus, the problem is divided into two. The first part is to get an estimate, $\mathbf{y} = \mathbf{Wx}$, of the variables, $s_j(t)$, of the original independent components. The second part is to find the matrix, A . How this method works is based on two principles:

- Independent components must be statistically independent (S matrix values). Roughly speaking, the values of one vector of an independent component do not affect the values of another component.
- Independent components must have a non-Gaussian distribution.

The theoretical basis of ICA is the central limit theorem, which states that the distribution of the sum (average or linear combination) of N independent random variables approaches Gaussian for $N \rightarrow \infty$. In particular, if x_i are random variables independent of each other, taken from an arbitrary distribution with an average, μ , and a variance of σ^2 , then if we denote the mean of these variables as $\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$, we can say that $\frac{\bar{x} - \mu}{\sigma/\sqrt{N}}$ approaches the Gaussian with a mean of 0 and a variance of 1. To solve the BSS problem, we need to find the matrix, w , so that $\mathbf{y} = \mathbf{Wx} = \mathbf{WAs}$. Here, the \mathbf{S} should be as close as possible to the original independent sources. We can consider this approach as the inverse process of the central limit theorem. All ICA methods are based on the same fundamental approach—finding a matrix, W , that maximizes non-Gaussianity, thereby minimizing the independence of \mathbf{S} .

The Fast ICA algorithm aims to maximize the function, $\sum_i E\{G(y_i)\}$, where $y_i = w_i^T x$ are components of $y = \mathbf{W}x$. Therefore, we can rewrite the function's equation in the following form:

$$\sum_i E\{G(y_i)\} = \sum_i E\{G(w_i^T x)\}$$

Here, the w_i^T vector is the i^{th} row of the matrix, W .

The ICA algorithm performs the following steps:

1. It chooses the initial value of w .
2. It calculates $w \Rightarrow E\{xg'(w^T x)\} - E\{g'(w^T x)\} w$, where $g(z) = \frac{dG(z)}{dz}$ is the derivative of the function, $G(z)$.
3. It normalizes $w \Rightarrow \frac{w}{\|w\|}$.
4. It repeats the previous two steps until w stops changing.

To measure non-Gaussianity, Fast ICA relies on a nonquadratic non-linear function, $G(z)$, that can take the following forms:

$$\begin{aligned} G(z) &= \logcosh(u), g(z) = \tanh(z), g'(z) = 1 - \tanh^2(z) \\ G(z) &= -\exp^{-\frac{z^2}{2}}, g(z) = z \exp^{-\frac{z^2}{2}}, g'(z) = (1 - z^2) \exp^{-\frac{z^2}{2}} \end{aligned}$$

Linear discriminant analysis

LDA is a type of multivariate analysis that allows us to estimate differences between two or more groups of objects at the same time. The basis of discriminant analysis is the assumption that the descriptions of the objects of each k^{th} class are instances of a multidimensional random variable that's distributed according to the normal (Gaussian) law, $N_m(\mu_k; \Sigma_k)$, with an average, μ_k , and the following covariance matrix:

$$\mathbf{C}_k = \frac{1}{n_k - 1} \sum_{i=1}^{n_k} (\mathbf{x}_{ik} - \mu_k)^T (\mathbf{x}_{ik} - \mu_k)$$

The index, m , indicates the dimension of the feature space. Consider a simplified geometric interpretation of the LDA algorithm for the case of two classes. Let the discriminant variables, x , be the axes of the m -dimensional Euclidean space. Each object (sample) is a point of this space with coordinates representing the fixed values of each variable. If both classes differ from each other in observable variables (features), they can be represented as clusters of points in different regions of the considered space that may partially overlap. To determine the position of each class, we can calculate its **centroid**, which is an imaginary point whose coordinates are the average values of the variables (features) in the class. The task of discriminant analysis is to create an additional z axis that passes through a cloud of points in such a way that the projections on it provide the best separability into two classes (in other words, it maximizes the distance between classes). Its position is given by a **linear discriminant (LD)** function with weights, β_j , that determine the contribution of each initial variable, x_j :

$$z(\mathbf{x}) = \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m$$

If we assume that the covariance matrices of the objects of classes 1 and 2 are equal, that is, $\mathbf{C} = \mathbf{C}_1 = \mathbf{C}_2$, then the vector of coefficients, β_1, \dots, β_m , of the LD, $z(\mathbf{x})$, can be calculated using the formula $\beta = \mathbf{C}^{-1}(\mu_1 - \mu_2)$, where \mathbf{C}^{-1} is the inverse of the covariance matrix and μ_k is the mean of the k^{th} class. The resulting axis coincides with the equation of a line passing through the centroids of two groups of class objects. The generalized Mahalanobis distance, which is equal to the distance between them in the multidimensional feature space, is estimated as $D^2 = \beta(\mu_1 - \mu_2)$.

Thus, in addition to the assumption regarding the normal (Gaussian) distribution of class data, which in practice occurs quite rarely, the LDA has a stronger assumption about the statistical equality of intragroup dispersions and correlation matrices. If there are no significant differences between them, they are combined into a calculated covariance matrix, as follows:

$$\mathbf{C} = (\mathbf{C}_1(\mathbf{n}_1 - 1) + \mathbf{C}_2(\mathbf{n}_2 - 1)) / (\mathbf{n}_1 + \mathbf{n}_2 - 2)$$

This principle can be generalized to a larger number of classes. The final algorithm may look like this:

$$S_W = \sum_{i=1}^n \Sigma_i$$

The interclass scattering matrix is calculated like this:

$$S_B = \sum_{i=1}^n N_i(\mu_i - \mu)(\mu_i - \mu)^T$$

Here, $\mu = \frac{1}{n} \sum_{i=1}^n x_i$ is the mean of all objects (samples), n is the number of classes, N_i is the number of objects in the i^{th} class, $\mu_i = \frac{1}{n_i} \sum \{x | x \in c_i\}$ is the intraclass' mean, $\Sigma_i = X_i C_n X_i^T$ is the scattering matrix for the i^{th} class, and $C_n = I_n - \frac{1}{n} J$ is a centering matrix, where J is the $n \times n$ matrix of all 1s.

Based on these matrices, the $S_W^{-1} S_B$ matrix is calculated, for which the eigenvalues and the corresponding eigenvectors are determined. In the diagonal elements of the matrix, we must select the s of the largest eigenvalues and transform the matrix, leaving only the corresponding s rows in it. The resulting matrix can be used to convert all objects into lower-dimensional space.

This method requires labeled data, meaning it is a supervised method.

Factor analysis

Factor analysis is used to reduce the number of variables that are used to describe data and determine the relationships between them. During the analysis, variables that correlate with each other are combined into one factor. As a result, the dispersion between components is redistributed, and the structure of factors becomes more understandable. After combining the variables, the correlation of components within each factor becomes higher than their correlation with components from other

factors. It is assumed that known variables depend on a smaller number of unknown variables and that we have a random error that can be expressed as follows:

$$X_{ij} = \sum_{k=1}^K \alpha_{i,k} W_{j,k} + \varepsilon_{ij}$$

Here, α is the load and W is the factor.

The concept of **factor load** is essential. It is used to describe the role of the factor (variable) when we wish to form a specific vector from a new basis. The essence of factor analysis is the procedure of rotating factors, that is, redistributing the dispersion according to a specific method. The purpose of rotations is to define a simple structure of factor loadings. Rotation can be orthogonal and oblique. In the first form of rotation, each successive factor is determined to maximize the variability that remains from the previous factors. Therefore, the factors are independent and uncorrelated with each other. The second type is a transformation in which factors correlate with each other. There are about 13 methods of rotation that are used in both forms. The factors that have a similar effect on the elements of the new basis are combined into one group. Then, from each group, it is recommended to leave one representative. Some algorithms, instead of choosing a representative, calculate a new factor with some heuristics that become central to the group.

Dimensionality reduction occurs while transitioning to a system of factors that are representatives of groups, and the other factors are discarded. There are several commonly used criteria for determining the number of factors. Some of these criteria can be used together to complement each other. An example of a criterion that's used to determine the number of factors is the Kaiser criterion or the eigenvalue criterion: only factors with eigenvalues equal to or greater than *one* are selected. This means that if a factor does not select a variance equivalent to at least one variance of one variable, then it is omitted. The general factor analysis algorithm follows these steps:

1. It calculates the correlation matrix.
2. It selects the number of factors for inclusion, for example, with the Kaiser criterion.
3. It extracts the initial set of factors. There are several different extraction methods, including maximum likelihood, PCA, and principal axis extraction.
4. It rotates the factors to a final solution that is equal to the one that was obtained in the initial extraction but that has the most straightforward interpretation.

Multidimensional scaling

MDS can be considered as an alternative to factor analysis when, in addition to the correlation matrices, an arbitrary type of object similarity matrix can be used as input data. MDS is not so much a formal mathematical procedure but rather a method of efficiently placing objects, thus keeping an appropriate distance between them in a new feature space. The dimension of the new space in MDS is always substantially less than the original space. The data that's used for analysis by MDS is often obtained from the matrix of pairwise comparisons of objects.

The main MDS algorithm's goal is to restore the unknown dimension, p , of the analyzed feature space and assign coordinates to each object in such a way that the calculated pairwise Euclidean distances between the objects coincide as much as possible with the specified pairwise comparison matrix. We are talking about restoring the coordinates of the new reduced feature space with the accuracy of orthogonal transformation, ensuring the pairwise distances between the objects do not change.

Thus, the aim of MDS methods can also be formulated in order to display the configuration information of the original multidimensional data that's given by the pairwise comparison matrix. This is provided as a configuration of points in the corresponding space of lower dimension.

Classical MDS assumes that the unknown coordinate matrix, X , can be expressed by eigenvalue decomposition, $B = XX'$. B can be computed from the proximity matrix D (a matrix with distances between samples) by using double centering. The general MDS algorithm follows these steps:

1. It computes the squared proximity matrix, $D^{(2)} = [d_{ij}^2]$.
2. It applies double centering, $B = -\frac{1}{2}JD^{(2)}J'$, using the centering matrix, $J = I - \frac{1}{n}11'$, where n is the number of objects.
3. It determines the m largest eigenvalues, $\lambda_1, \lambda_2, \dots, \lambda_m$, and the corresponding eigenvectors, e_1, e_2, \dots, e_m , of B (where m is the number of dimensions desired for the output).
4. It computes $X = E_m \Lambda_m^{1/2}$, where E_m is the matrix of m eigenvectors and Λ_m is the diagonal matrix of m eigenvalues of B .

The disadvantage of the MDS method is that it does not take into account the distribution of nearby points since it uses Euclidean distances in calculations. If you ever find multidimensional data lying on a curved manifold, the distance between data points can be much more than Euclidean.

Now that we've discussed the linear methods we can use for dimension-reduction, let's look at what non-linear methods exist.

Exploring non-linear methods for dimension-reduction

In this section, we'll discuss the widespread non-linear methods and algorithms that are used for dimension-reduction, such as the following:

- Kernel PCA
- Isomap
- Sammon mapping
- Distributed **stochastic neighbor embedding (SNE)**
- Autoencoders

Kernel PCA

Classic PCA is a linear projection method that works well if the data is linearly separable. However, in the case of linearly non-separable data, a non-linear approach is required. The basic idea of working with linearly inseparable data is to project it into a space with a larger number of dimensions, where it becomes linearly separable. We can choose a non-linear mapping function, ϕ , so that the sample mapping, x , can be written as $\mathbf{x} \rightarrow \phi(\mathbf{x})$. This is called the **kernel function**. The term *kernel* describes a function that calculates the scalar product of mapping (in a higher-order space) samples x with ϕ , $k(x_i, x_j) = \phi(\mathbf{x}_i)\phi(\mathbf{x}_j)^T$. This scalar product can be interpreted as the distance measured in the new space. In other words, the ϕ function maps the original d -dimensional elements into the k -dimensional feature space of a higher dimension by creating non-linear combinations of the original objects. For example, a function that displays 2D samples, $x = \{x_1, x_2\}$, in 3D space can look like $\phi(x) \rightarrow x_1^2, x_2^2, \sqrt{2}x_1x_2$.

In a linear PCA approach, we are interested in the principal components that maximize the variance in the dataset. We can maximize variance by calculating the eigenvectors (principal components) that correspond to the largest eigenvalues based on the covariance matrix of our data and project our data onto these eigenvectors. This approach can be generalized to data that is mapped into a higher dimension space using the kernel function. However, in practice, the covariance matrix in a multidimensional space is not explicitly calculated since we can use a method called the **kernel trick**. The kernel trick allows us to project data onto the principal components without explicitly calculating the projections, which is much more efficient. The general approach is as follows:

1. Compute the **kernel matrix** equal to $K_{i,j} = k(x_i, x_j)$.
2. Make it so that it has a zero mean value, $K' = K - \mathbf{1}_N K - K \mathbf{1}_N + \mathbf{1}_N K \mathbf{1}_N$, where $\mathbf{1}_N$ is a matrix of $N \times N$ size with $1/N$ elements.
3. Calculate the eigenvalues and eigenvectors of K' .
4. Sort the eigenvectors in descending order, according to their eigenvalues.
5. Take n eigenvectors that correspond to the largest eigenvalues, where n is the number of dimensions of a new feature space.

These eigenvectors are projections of our data onto the corresponding main components. The main difficulty of this process is selecting the correct kernel and configuring its hyperparameters. Two frequently used kernels are the polynomial kernel $k(x, y) = (x^T y + c)^d$ and the Gaussian (Radial Basis Function (RBF)) $k(x, y) = \exp(-\gamma \|x - y\|_2^2)$ ones.

Isomap

The **Isomap** algorithm is based on the manifold projection technique. In mathematics, the **manifold** is a topological space (which is, in general, a set of points with their neighbors) that locally resembles the Euclidian space near each point. For example, one-dimensional manifolds include lines and circles but not figures with self-intersections. 2D manifolds are called **surfaces**; for example, they can be a

sphere, a plane, or a torus, but these surfaces can't have self-intersection. For example, a circle is a one-dimensional manifold embedded into a 2D space. Here, each arc of the circle locally resembles a straight-line segment. A 3D curve can also be a manifold if it can be divided into straight-line segments that can be embedded in 3D space without self-intersections. A 3D shape can be a manifold if its surface can be divided into flat plane patches without self-intersections.

The basics of applying manifold projection techniques are to search for a manifold that is close to the data, project the data onto the manifold, and then unfold it. The most popular technique that's used to find the manifold is to build a graph based on information about data points. Usually, these data points are placed into the graph nodes, and the edges simulate the relationships between the data points.

The Isomap algorithm depends on two parameters:

- The number of neighbors, k , used to search for geodetic distances
- The dimension of the final space, m

In brief, the Isomap algorithm follows these steps:

1. First, it constructs a graph representing geodesic distances. For each point, we search the k nearest neighbors and construct a weighted, undirected graph from the distances to these nearest neighbors. The edge weight is the Euclidean distance to the neighbor.
2. Using an algorithm to find the shortest distance in the graph, for example, Dijkstra's algorithm, we need to find the shortest distance between each pair of vertices. We can consider this distance as a geodesic distance on a manifold.
3. Based on the matrix of pairwise geodesic distances we obtained in the previous step, train the MDS algorithm.
4. The MDS algorithm associates a set of points in the m -dimensional space with the initial set of distances.

Sammon mapping

Sammon mapping is one of the first non-linear dimensionality reduction algorithms. In contrast to traditional dimensionality reduction methods, such as PCA, Sammon mapping does not define a data conversion function directly. On the contrary, it only determines the measure of how well the conversion results (a specific dataset of a smaller dimension) correspond to the structure of the original dataset. In other words, it does not try to find the optimal transformation of the original data; instead, it searches for another dataset of lower dimensions with a structure that's as close to the original one as possible. The algorithm can be described as follows. Let's say we have nN -dimensional vectors, x_i ($i = 1, 2, \dots, n$). Here, n vectors are defined in the M -dimensional space, ($M = 2, 3$), which is denoted by y_i . The distances between the vectors in the N -dimensional space will be denoted by $d_{ij}^* = (y_i, y_j)$ and in the M -dimensional space, $d_{ij} = (x_i, x_j)$. To determine the distance between the vectors, we can use any metric; in particular, the Euclidean distance. The goal of non-linear Sammon

mapping is to search a selection of vectors, \mathbf{y} , in order to minimize the error function, E , which is defined by the following formula:

$$E = \frac{1}{c} \sum_{i < j}^n \frac{\left[d_{ij}^* - d_{ij} \right]^2}{d_{ij}^*}$$

$$c = \sum_{i < j}^n d_{ij}^*$$

$$d_{ij} = \sqrt{\sum_{k=1}^M [y_{ik} - y_{jk}]^2}$$

To minimize the error function, E , Sammon used Newton's minimization method, which can be simplified as follows:

$$y_{pq}^{k+1} = y_{pq} - \eta \Delta_{pq}(k)$$

$$\Delta_{pq} = \frac{\frac{\partial E}{\partial y_{pq}}}{\frac{\partial^2 E}{\partial y_{pq}^2}}$$

Here, η is the learning rate.

Distributed stochastic neighbor embedding

The SNE problem is formulated as follows: we have a dataset with points described by a multidimensional variable with a dimension of space substantially higher than three. It is necessary to obtain a new variable that exists in a 2D or 3D space that would maximally preserve the structure and patterns in the original data. The difference between t-SNE and the classic SNE lies in the modifications that simplify the process of finding the global minima. The main modification is replacing the normal distribution with the Student's t-distribution for low-dimensional data. SNE begins by converting the multidimensional Euclidean distance between points into conditional probabilities that reflect the similarity of points. Mathematically, it looks like this:

$$p_{j|i} = \frac{\exp(-\|x_i - x_j\|^2 / 2\delta_i^2)}{\sum_{k \neq i} \exp(-\|x_i - x_k\|^2 / 2\delta_i^2)}$$

This formula shows how close the point x_j lies to the point x_i with a Gaussian distribution around x_i , with a given deviation of δ . δ is different for each point. It is chosen so that the points in areas with higher density have less variance than others.

Let's denote 2D or 3D mappings of the (x_i, x_j) pair as the (y_i, y_j) pair. It is necessary to estimate the conditional probability using the same formula. The standard deviation is $\frac{1}{\sqrt{2}}$:

$$q_{j|i} = \frac{\exp(-||y_i - y_j||^2)}{\sum_{k \neq i} \exp(-||y_i - y_k||^2)}$$

If the mapping points, y_i and y_j , correctly simulate the similarity between the original points of the higher dimension, x_i and x_j , then the corresponding conditional probabilities, $p_{j|i}$ and $q_{j|i}$, will be equivalent. As an obvious assessment of the quality of how $q_{j|i}$ reflects $p_{j|i}$, divergence, or the Kullback-Leibler distance is used. SNE minimizes the sum of such distances for all mapping points using gradient descent. The following formula determines the loss function for this method:

$$Cost = \sum_i \sum_j p_{j|i} \log\left(\frac{p_{j|i}}{q_{j|i}}\right)$$

It has the following gradient:

$$\frac{\partial Cost}{\partial y} = 2 \sum_j (p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j})(y_i - y_j)$$

The authors of this problem proposed the following physical analogy for the optimization process. Let's imagine that springs connect all the mapping points. The stiffness of the spring connecting points i and j depends on the difference between the similarity of two points in a multidimensional space and two points in a mapping space. In this analogy, the gradient is the resultant force that acts on a point in the mapping space. If we let the system go, after some time, it results in balance, and this is the desired distribution. Algorithmically, it searches for balance while taking the following moments into account:

$$Y^{(t)} = Y^{(t+1)} + \eta \frac{\partial Cost}{\partial y} + \alpha(t)(Y^{(t-1)} - Y^{(t-2)})$$

Here, η is the learning rate and α is the coefficient of inertia. Classic SNE also allows us to get good results but can be associated with difficulties when optimizing the loss function and the crowding problem. t-SNE doesn't solve these problems in general, but it makes them much more manageable.

The loss function in t-SNE has two principal differences from the loss function of classic SNE. The first one is that it has a symmetric form of similarity in a multidimensional space and a simpler gradient version. Secondly, instead of using a Gaussian distribution for points from the mapping space, the t-distribution (Student) is used.

Autoencoders

Autoencoders represent a particular class of neural networks that are configured so that the output of the autoencoder is as close as possible to the input signal. In its most straightforward representation, the autoencoder can be modeled as a multilayer perceptron in which the number of neurons in the output layer is equal to the number of inputs. The following diagram shows that by choosing an intermediate hidden layer of a smaller dimension, we compress the source data into the lower dimension. Usually, values from this intermediate layer are a result of an autoencoder:

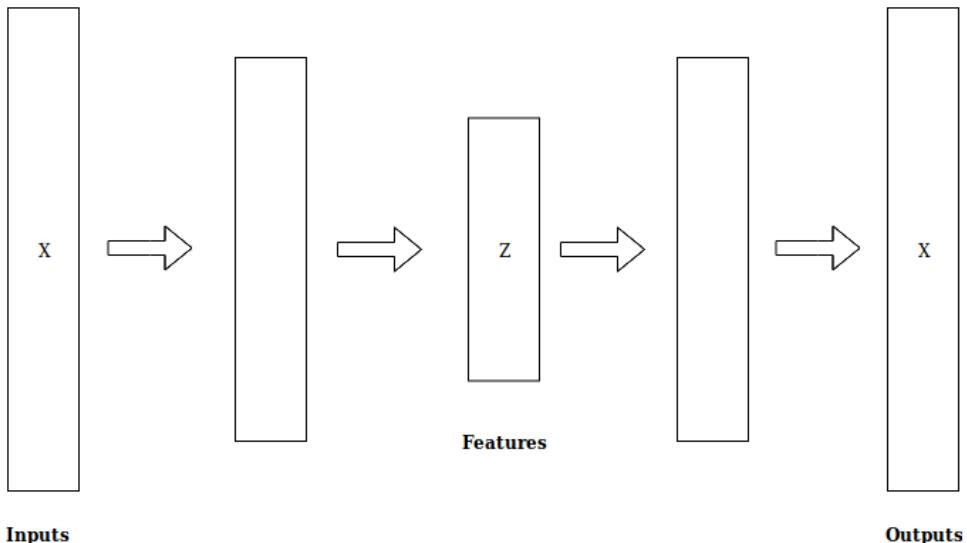


Figure 6.1 – Autoencoder architecture

Now that we have learned about the linear and non-linear methods that can be used for dimension-reduction and explored the components of each of the methods in detail, we can enhance our implementation of dimension-reduction with the help of some practical examples.

Understanding dimension-reduction algorithms with various C++ libraries

Let's look at how to use dimensionality reduction algorithms in practice. All of these examples use the same dataset, which contains four normally distributed 2D point sets that have been transformed with Swiss roll mapping, $f(x, y) \rightarrow x \cos(x), y, x \sin(x)$, into a 3D space. You can find the dataset and related details in the book's GitHub repository here: <https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition>. The following graph shows the result of this mapping.

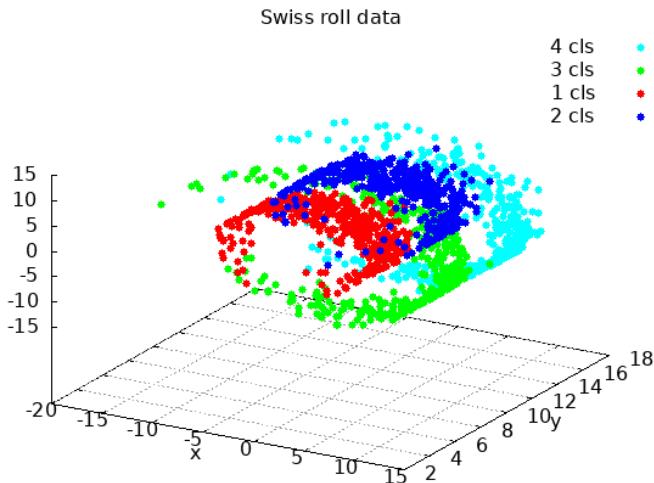


Figure 6.2 – Swiss roll dataset

This dataset is labeled. Each of the normally distributed parts has its own labels, and we can see these labels as a certain color on the result. We use these colors to show transformation results for each of the algorithms we'll be using in the following samples. This gives us an idea of how the algorithm works. The following sections provide concrete examples of how to use the `Dlib`, `Tapkee`, and `libraries`.

Using the `Dlib` library

There are three dimensionality reduction methods in the `Dlib` library—two linear ones, known as PCA and LDA, and one non-linear one, known as Sammon mapping.

PCA

PCA is one of the most popular dimensionality reduction algorithms and it has a couple of implementations in the `Dlib` library. There is the `Dlib::vector_normalizer_pca` type, for which objects can be used to perform PCA on user data. This implementation also normalizes the data. In some cases, this automatic normalization is useful because we always have to perform PCA on normalized data. An object of this type should be parameterized with the input data sample type. After we've instantiated an object of this type, we use the `train()` method to fit the model to our data. The `train()` method takes `std::vector` as samples and the `eps` value as parameters. The `eps` value controls how many dimensions should be preserved after the PCA has been transformed. This can be seen in the following code:

```
void PCAReduction(const std::vector<Matrix> &data, double target_dim)
{
    // instantiate the PCA algorithm object.
    Dlib::vector_normalizer_pca<Matrix> pca;
```

```

// train the PCA algorithm
pca.train(data, target_dim / data[0].nr());
// apply trained algorithm to the new data
std::vector<Matrix> new_data;
new_data.reserve(data.size());
for (size_t i = 0; i < data.size(); ++i) {
    new_data.emplace_back(pca(data[i]));
}
// example how to get transformed values
for (size_t r = 0; r < new_data.size(); ++r) {
    Matrix vec = new_data[r];
    double x = vec(0, 0);
    double y = vec(1, 0);
}

```

After the algorithm has been trained, we use the object to transform individual samples. Take a look at the first loop in the code and notice how the `pca ([data [i]])` call performs this transformation.

The following graph shows the result of the PCA transformation:

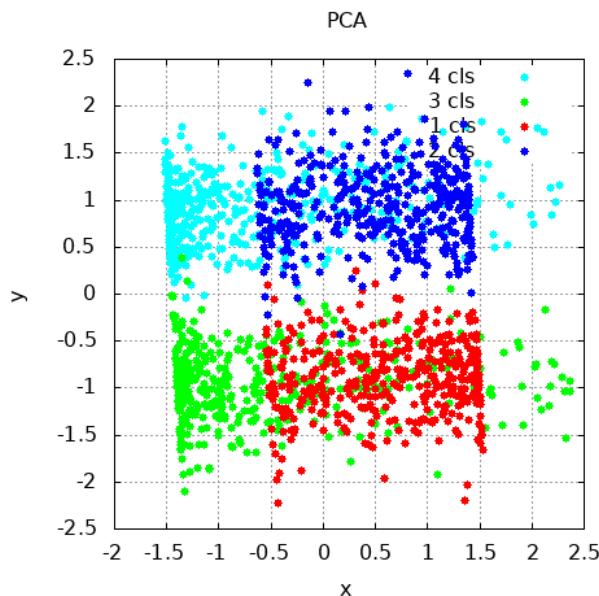


Figure 6.3 – Dlib PCA transformation visualization

Data compression with PCA

We can use dimensionality reduction algorithms for a slightly different task—data compression with information loss. This can be easily demonstrated when applying the PCA algorithm to images. Let's implement PCA from scratch with the `Dlib` library using SVD decomposition. We can't use an existing implementation because it performs normalization in a way we can't fully control.

First, we need to load an image and transform it into matrix form:

```
void PCACompression(const std::string& image_file, long target_dim) {
    array2d<Dlib::rgb_pixel> img;
    load_image(img, image_file);

    array2d<unsigned char> img_gray;
    assign_image(img_gray, img);
    save_png(img_gray, "original.png");

    array2d<DataType> tmp;
    assign_image(tmp, img_gray);
    Matrix img_mat = Dlib::mat(tmp);
    img_mat /= 255.; // scale

    std::cout << "Original data size " << img_mat.size() <<
    std::endl;
```

After we've loaded the RGB image, we convert it into grayscale and transform its values into floating points. The next step is to transform the image matrix into samples that we can use for PCA training. This can be done by splitting the image into rectangular patches that are 8 x 8 in size with the `Dlib::subm()` function and then flattening them with the `Dlib::reshape_to_column_vector()` function:

```
std::vector<Matrix> data;
int patch_size = 8;

for (long r = 0; r < img_mat.nr(); r += patch_size) {
    for (long c = 0; c < img_mat.nc(); c += patch_size) {
        auto sm =
            Dlib::subm(img_mat, r, c, patch_size, patch_size);
        data.emplace_back(Dlib::reshape_to_column_vector(sm));
    }
}
```

When we have our samples, we can normalize them by subtracting the mean and dividing them by their standard deviation. We can make these operations vectorized by converting our vector of samples into the matrix type. We do this with the `Dlib::mat()` function:

```
// normalize data
auto data_mat = mat(data);
Matrix m = mean(data_mat);
Matrix sd = reciprocal(sqrt(variance(data_mat)));

matrix<decltype(data_mat)::type, 0, 1,
        decltype(data_mat)::mem_manager_type>
x(data_mat);
for (long r = 0; r < x.size(); ++r)
    x(r) = pointwise_multiply(x(r) - m, sd);
```

After we've prepared the data samples, we calculate the covariance matrix with the `Dlib::covariance()` function and perform SVD with the `Dlib::svd()` function. The SVD results are the eigenvalues matrix and the eigenvectors matrix. We sorted the eigenvectors according to the eigenvalues and left only a small number (in our case, 10 of them) of eigenvectors corresponding to the biggest eigenvalues. The number of eigenvectors we left is the number of dimensions in the new feature space:

```
Matrix temp, eigen, pca;
// Compute the svd of the covariance matrix
Dlib::svd(covariance(x), temp, eigen, pca);
Matrix eigenvalues = diag(eigen);

rsort_columns(pca, eigenvalues);

// leave only required number of principal components
pca = trans(colm(pca, range(0, target_dim)));
```

Our PCA transformation matrix is called `pca`. We used it to reduce the dimensions of each of our samples with simple matrix multiplication. Look at the following cycle and notice the `pca * data[i]` operation:

```
// dimensionality reduction
std::vector<Matrix> new_data;
size_t new_size = 0;
new_data.reserve(data.size());
for (size_t i = 0; i < data.size(); ++i) {
    new_data.emplace_back(pca * data[i]);
    new_size += static_cast<size_t>(new_data.back().size());
}
```

```
std::cout << "New data size "
    << new_size + static_cast<size_t>(pca.size())
    << std::endl;
```

Our data has been compressed and we can see its new size in the console output. Now, we can restore the original dimension of the data to be able to see the image. To do this, we need to use the transposed PCA matrix to multiply the reduced samples. Also, we need to denormalize the restored sample to get actual pixel values. This can be done by multiplying the standard deviation and adding the mean we got from the previous steps:

```
auto pca_matrix_t = Dlib::trans(pca);
Matrix isd = Dlib::reciprocal(sd);
for (size_t i = 0; i < new_data.size(); ++i) {
    Matrix sample = pca_matrix_t * new_data[i];
    new_data[i] = Dlib::pointwise_multiply(sample, isd) + m;
}
```

After we've restored the pixel values, we reshape them and place them in their original location in the image:

```
size_t i = 0;
for (long r = 0; r < img_mat.nr(); r += patch_size) {
    for (long c = 0; c < img_mat.nc(); c += patch_size)
    {
        auto sm = Dlib::reshape(new_data[i],
                               patch_size, patch_size);
        Dlib::set_subm(img_mat, r, c, patch_size,
                      patch_size) = sm;
        ++i;
    }
}

img_mat *= 255.0;
assign_image(img_gray, img_mat);
equalize_histogram(img_gray);
save_png(img_gray, "compressed.png");
}
```

Let's look at the result of compressing a standard test image that is widely used in image processing. The following is the Lena 512 x 512 px image:



Figure 6.4 – Original image before compression

Its original grayscale size is 262,144 bytes. After we perform PCA compression with only 10 principal components, its size becomes 45,760 bytes. We can see the result in the following figure:



Figure 6.5 – Image after compression

Here, we can see that most of the essential visual information was preserved, despite the high compression rate.

LDA

The Dlib library also has an implementation of the LDA algorithm, which can be used for dimensionality reduction. It's a supervised algorithm, so it needs labeled data. This algorithm is implemented with the `Dlib::compute_lda_transform()` function, which takes four parameters. The first one is the input/output parameter—as input, it is used to pass input training data (in matrix form) and as output, it receives the LDA transformation matrix. The second parameter is the output for the mean values. The third parameter is the labels for the input data, while the fourth one is the desired number of target dimensions. The following code shows an example of how to use LDA for dimension-reduction with the Dlib library:

```
void LDAReduction(const Matrix &data,
                   const std::vector<unsigned long> &labels,
                   unsigned long target_dim) {
    Dlib::matrix<DataType, 0, 1> mean;
    Matrix transform = data;
    // Apply LDA on input data,
    // result will be in the "transform object"
    Dlib::compute_lda_transform(transform, mean, labels,
                               target_dim);
    // Apply LDA "transform" to the input "data"
    for (long r = 0; r < data.nr(); ++r) {
        Matrix row =
            transform * Dlib::trans(Dlib::rowm(data, r)) - mean;
        double x = row(0, 0);
        double y = row(1, 0);
    }
}
```

To perform an actual LDA transform after the algorithm has been trained, we multiply our samples with the LDA matrix. In our case, we also transposed them. The following code shows the essential part of this example:

```
transform * Dlib::trans(Dlib::rowm(data, r))
```

The following graph shows the result of using LDA reduction on two components:

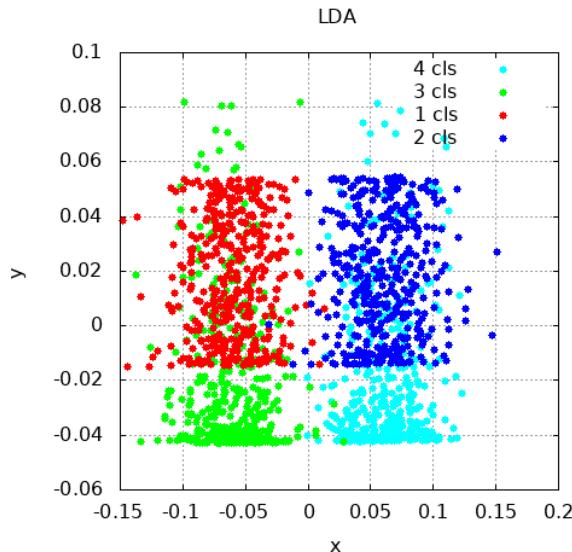


Figure 6.6 – The Dlib LDA transformation visualization

In the following block, we will see how to use Sammon mapping dimensionality reduction algorithm implementation from the `Dlib` library.

Sammon mapping

In the `Dlib` library, Sammon mapping is implemented with the `Dlib::sammon_projection` type. We need to create an instance of this type and then use it as a functional object. Functional object call arguments are the data that we need to transform and the number of dimensions of the new feature space. The input data should be in the form of the `std::vector` of the single samples of the `Dlib::matrix` type. All samples should have the same number of dimensions. The result of using this functional object is a new vector of samples with a reduced number of dimensions:

```
void SammonReduction(const std::vector<Matrix> &data, long target_dim)
{
    Dlib::sammon_projection sp;
    auto new_data = sp(data, target_dim);
    for (size_t r = 0; r < new_data.size(); ++r) {
        Matrix vec = new_data[r];
        double x = vec(0, 0);
        double y = vec(1, 0);
    }
}
```

The following graph shows the result of using this dimensionality reduction algorithm:

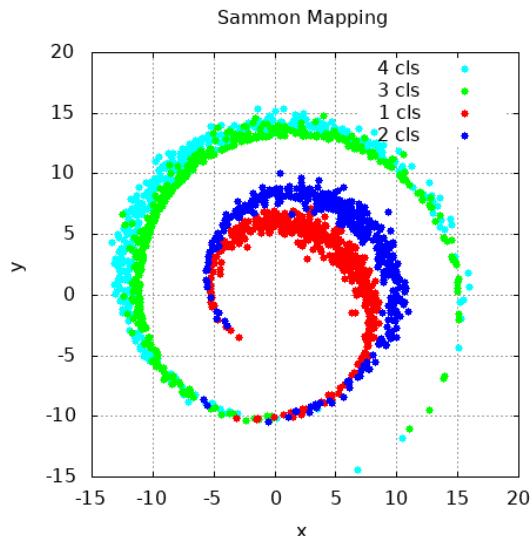


Figure 6.7 – The Dlib Sammon mapping transformation visualization

In the next section, we will learn how to use the Tapkee library for solving dimensionality reduction tasks.

Using the Tapkee library

The Tapkee library contains numerous dimensionality reduction algorithms, both linear and non-linear ones. This is the headers-only C++ template library so it doesn't require any compilation, and can be easily integrated into your application. It has a couple of dependencies: the `fmt` library for the formatted output and the Eigen3 as the math backend.

There are no special classes for algorithms in this library because it provides a uniform API based on parameters to build a dimensionality reduction object. So, using this approach, we can define a single function that will take a set of parameters and high-dimensional input data and perform dimensionality reduction. The result will be a 2D plot. The following code sample shows its implementation:

```
void Reduction(tapkee::ParametersSet parameters,
               bool with_kernel,
               const tapkee::DenseMatrix& features,
               const tapkee::DenseMatrix& labels,
               const std::string& img_file) {
    using namespace tapkee;
    // define the kernel callback object,
    // that will be applied to the input "features"
    gaussian_kernel_callback kcb(features, 2.0);
    // define distance callback object,
```

```
// that will be applied to the input "features"
eigen_distance_callback dcb(features);
// define the feature access callback object
eigen_features_callback fcb(features);
// save the initial features indices order
auto n = features.cols();
std::vector<int> indices(n);
for (int i = 0; i < n; ++i) indices[i] = i;
TapkeeOutput result;
if (with_kernel) {
    // apply feature transformation with kernel function
    result =
        initialize()
            .withParameters(parameters)
            .withKernel(kcb)
            .withFeatures(fcb)
            .withDistance(dcb)
            .embedRange(indices.begin(), indices.end());
} else {
    // apply features transformation without kernel
    // //function
    result =
        initialize()
            .withParameters(parameters)
            .withFeatures(fcb)
            .withDistance(dcb)
            .embedRange(indices.begin(), indices.end());
}
// create helper object for transformed data
// //visualization result
Clusters clusters;
for (index_t i = 0; i < result.embedding.rows(); ++i) {
    // get a transformed feature
    auto new_vector = result.embedding.row(i);
    // populate visualization helper structure
    auto label = static_cast<int>(lables(i));
    clusters[label].first.push_back(new_vector[0]);
    clusters[label].second.push_back(new_vector[1]);
}
// Visualize dimensionality reduction result
PlotClusters(clusters,
             get_method_name(parameters[method]),
             img_file);
}
```

This is the single function that we will use to see several algorithms from the Tapkee library. The main parameter it takes is `tapkee::ParametersSet`; an object of this type can be initialized with the dimensionality reduction method type, the number of target dimensions, and some special parameters that configure the selected method. The `with_kernel` parameter specifies if this function will attach the kernel transform to the algorithm pipeline or not. The library API allows you to attach the kernel transform for any algorithm but it will be used only if the algorithm implementation uses it. In our case, we will use it only for the Kernel PCA method. The last parameters for the Reduction function are the input data, labels for plotting, and the output file name.

Let's look into the implementation. At first, we define the callback functors for a method pipeline. There are the Gaussian kernel callback, the linear distance callback, and the feature callback. Notice that all of these callback objects were initialized with the input data. It was done to reduce the data coping in the pipeline. The library API requires callback objects to be able to produce some result for two data indices. These callback objects should implement functionality to get access to the original data. So, all our callbacks were constructed from library-defined classes and store references to the original data containers. `tapkee::DenseMatrix` is just a typedef for the Eigen3 dense matrix. Another important thing is the index map usage for the data access, for example, the `indices` variable; it allows you to use your data more flexibly. The following snippet shows the dimensionality reduction object creation and application:

```
initialize() .  
withParameters(parameters) .  
withKernel(kcb) .  
withFeatures(fcb) .  
withDistance(dcb) .  
embedRange(indices.begin(), indices.end());
```

You can see that the API is uniform and the builder functions that perform configurations start with the word `with`. At first, we passed the parameters to the build, then three callback functors, and finally called the `embedRange` method that does the actual dimensionality reduction. `eigen_features_callback` is needed to access particular data values with indices, and `eigen_distance_callback` is used in some algorithms to measure the distance between item vectors; in our case, it's just the Euclidean distance, but you can define any you need.

In the last part of the function, the `Clusters` object is populated with new 2D coordinates and labels. Then, this object is used to plot the dimensionality reduction result. The plotting approach from the previous chapters was applied.

In the following subsections, we will learn how to use different dimensionality reduction methods with our general function. These methods will be as follows:

- PCA
- Kernel PCA

- MDS
- Isomap
- Factor analysis
- t-distributed SNEs

PCA

Having the general function for the dimensionality reduction, we can use it for applying different methods. The following code shows how to create a parameter set to configure the PCA method:

```
bool with_kernel = false;
Reduction((method = PCA, target_dimension = target_dim),
           with_kernel, input_data, labels_data,
           "pca-tapkee.png");
```

The first argument is the initialization of the `tapkee::ParametersSet` type object. We used the `tapkee::PCA` enumeration value and specified the number of target dimensions. Also, we didn't use a kernel. `input_data` and `labels_data` are input data loaded from the file, and these variables have `tapkee::DenseMatrix` type, which actually is the Eigen3 dense matrix. The last parameter is the name of the output file for the plot image.

The following graph shows the result of applying the Tapkee PCA implementation to our data:

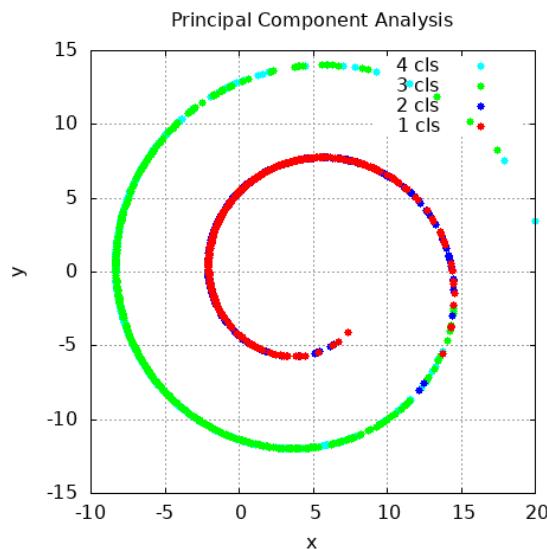


Figure 6.8 – The Tapkee PCA transformation visualization

You can see that this method was not able to spatially separate our 3D data in the 2D space.

Kernel PCA

The non-linear version of PCA is also implemented in the Tapkee library. To use this method, we define the kernel method and pass it as a callback to the `withKernel` builder method of library API. We already did it in our general function, so the only thing we have to do is pass the `with_kernel` parameter as `true`. We used the Gaussian kernel, which is defined as follows:

$$k(\mathbf{x}, \mathbf{x}') = \exp(-||\mathbf{x} - \mathbf{x}'||^2 \gamma)$$

Here, γ is the scale coefficient that can be estimated as a median of the item difference, or just configured manually. The kernel callback function is defined in the following way:

```
struct gaussian_kernel_callback {
    gaussian_kernel_callback(
        const tapkee::DenseMatrix& matrix,
        tapkee::ScalarType gamma)
        : feature_matrix(matrix), gamma(gamma) {};
    inline tapkee::ScalarType kernel(
        tapkee::IndexType a, tapkee::IndexType b) const {
        auto distance =
            (feature_matrix.col(a) - feature_matrix.col(b))
                .norm();
        return exp(-(distance * distance) * gamma);
    }
    inline tapkee::ScalarType operator()(

        tapkee::IndexType a, tapkee::IndexType b) const {
        return kernel(a, b);
    }
    const tapkee::DenseMatrix& feature_matrix;
    tapkee::ScalarType gamma{1};
}
```

Tapkee requires that you define the method named `kernel` and the function operator. Our implementation is very simple; the main feature here is that the reference to the particular data is stored as a member. It's done in a such way because the library will use only indices to call the kernel functor. The actual call to our general function looks like this:

```
bool with_kernel = true;
Reduction(method = KernelPCA, target_dimension = target_dim),
          with_kernel, input_data, labels_data,
          "kernel-pca-tapkee.png");
```

The following graph shows the result of applying the Tapkee kernel PCA implementation to our data:

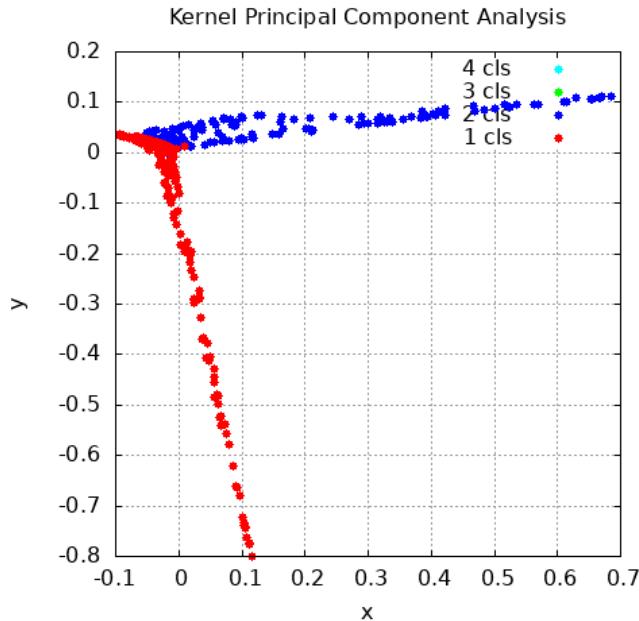


Figure 6.9 – The Tapkee kernel PCA transformation visualization

We can see that this type of kernel makes some parts of the data separated, but that other ones were reduced too much.

MDS

To use the MDS algorithm, we should just pass the method name to our general dimensionality reduction function. There are no other configurable parameters, especially for this algorithm. The following example shows how to use this method:

```
Reduction( (method = MultidimensionalScaling,
            target_dimension = target_dim),
            false, input_data, labels_data, "mds-tapkee.png";
```

The following graph shows the result of applying the Tapkee MDS algorithm to our data:

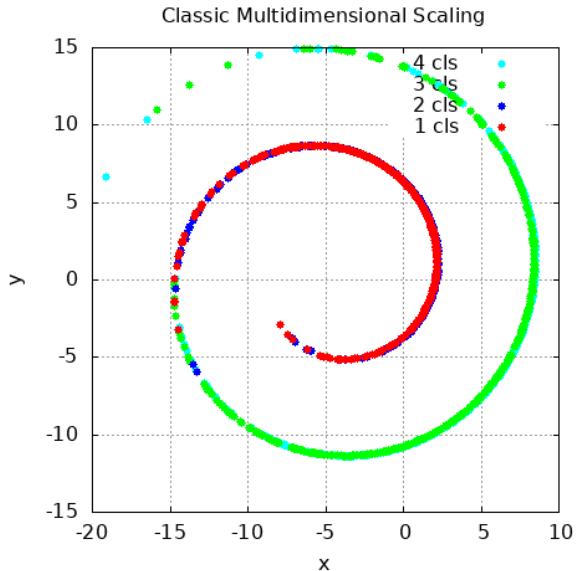


Figure 6.10 – The Tapkee MDS transformation visualization

You can see that the result is very similar to the PCA algorithm, and there is the same problem that the method was not able to spatially separate our data in the 2D space.

Isomap

The Isomap method can be applied to our data as follows:

```
Reduction(method = Isomap, target_dimension = target_dim,
          num_neighbors = 100),
          false, input_data, labels_data, "isomap-tapkee.png");
```

Apart from the method name and target dimensions, the `num_neighbors` parameter was passed. This is the number of nearest neighbor values that will be used by the Isomap algorithm.

The following graph shows the result of applying the Tapkee Isomap implementation to our data:

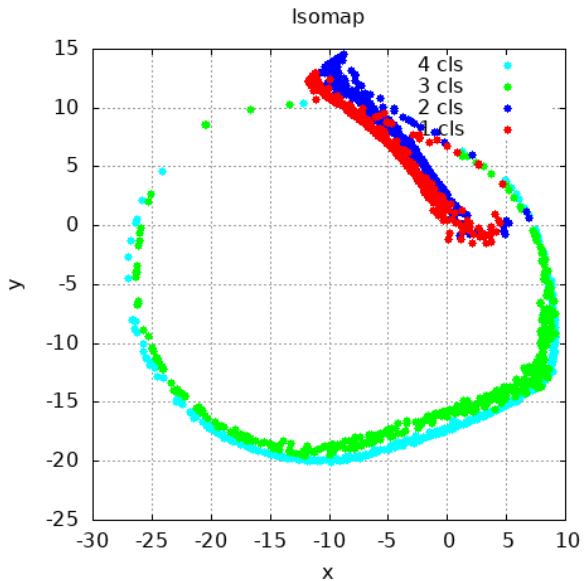


Figure 6.11 – The Tapkee Isomap transformation visualization

You can see that this method can spatially separate our data in the 2D space, but the clusters are too close to each other. Also, you can play with the number of neighbor parameters to get another separation.

Factor analysis

The factor analysis method can be applied to our data as follows:

```
Reduction( (method = FactorAnalysis,
            target_dimension = target_dim,
            fa_epsilon = 10e-5, max_iteration = 100),
            false, input_data, labels_data,
            "isomap-tapkee.png");
```

Apart from the method name and target dimensions, the `fa_epsilon` and `max_iteration` parameters can be passed. `fa_epsilon` is used to check the algorithm's convergence.

The following graph shows the result of applying the Tapkee factor analysis implementation to our data:

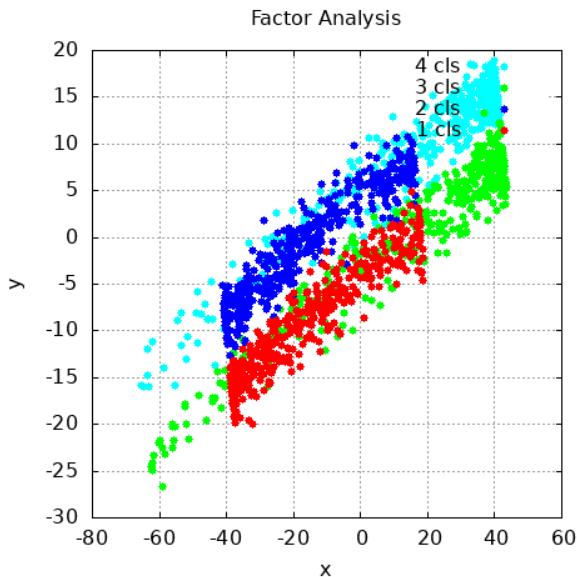


Figure 6.12 – The Tapkee factor analysis transformation visualization

This method also fails to clearly separate our data in the 2D space.

t-SNE

The t-SNE method can be applied to our data as follows:

```
Reduction( (method = tDistributedStochasticNeighborEmbedding,
            target_dimension = target_dim,
            sne_perplexity = 30),
            false, input_data, labels_data,
            "tsne-tapkee.png");
```

Apart from the method name and target dimensions, the `sne_perplexity` parameter was specified. This parameter regulates the algorithm convergence. Also, you can change the `sne_theta` value, which is the learning rate.

The following graph shows the result of applying the Tapkee t-SNE implementation to our data:

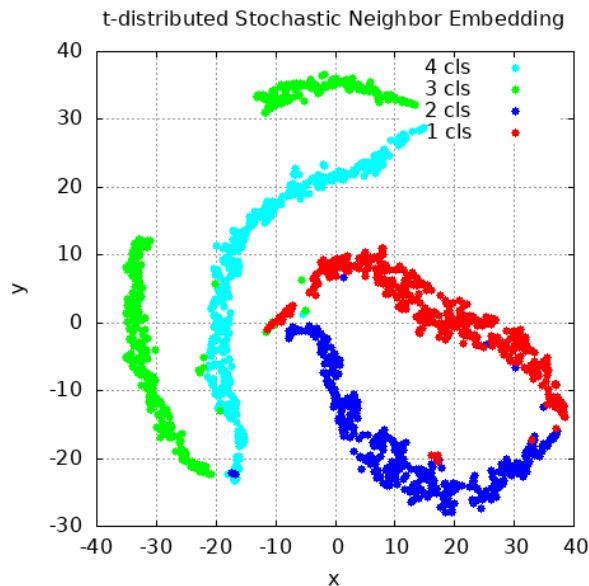


Figure 6.13 – The Tapkee t-SNE transformation visualization

You can see that this method gave the most reasonable separation of our data in the 2D space; there are distinct bounds between all clusters.

Summary

In this chapter, we learned that dimensionality reduction is the process of transferring data that has a higher dimension into a new representation of data with a lower dimension. It is used to reduce the number of correlated features in a dataset and extract the most informative features. Such a transformation can help increase the performance of other algorithms, reduce computational complexity, and make human-readable visualizations.

We learned that there are two different approaches to solving this task. One is feature selection, which doesn't create new features, while the second one is dimensionality reduction algorithms, which make new feature sets. We also learned that dimensionality reduction algorithms are linear and non-linear and that we should select either type, depending on our data. We saw that there are a lot of different algorithms with different properties and computational complexity and that it makes sense to try different ones to see which are the best solutions for particular tasks. Note that different libraries have different implementations for identical algorithms, so their results can differ, even for the same data.

The area of dimensionality reduction algorithms is a field that's in continual development. There is, for example, a new algorithm called **Uniform Manifold Approximation and Projection (UMAP)** that's based on Riemannian geometry and algebraic topology. It competes with the t-SNE algorithm in terms of visualization quality but also preserves more of the original data's global structure after the transformation is complete. It is also much more computationally effective, which makes it suitable for large-scale datasets. However, at the moment, there is no C++ implementation of it.

In the next chapter, we will discuss classification tasks and how to solve them. Usually, when we have to solve a classification task, we have to divide a group of objects into several subgroups. Objects in such subgroups share some common properties that are distinct from the properties in other subgroups.

Further reading

- A survey of dimensionality reduction techniques: <https://arxiv.org/pdf/1403.2877.pdf>
- A short tutorial for dimensionality reduction: https://www.math.uwaterloo.ca/~aghodsib/courses/f06stat890/readings/tutorial_stat890.pdf
- Guide to 12 dimensionality reduction techniques (with Python code): <https://www.analyticsvidhya.com/blog/2018/08/dimensionality-reduction-techniques-python/>
- A geometric and intuitive explanation of the covariance matrix and its relationship with linear transformation, an essential building block for understanding and using PCA and SVD: <https://datascienceplus.com/understanding-the-covariance-matrix>
- The kernel trick: <https://dscm.quora.com/The-Kernel-Trick>

7

Classification

In machine learning, the task of classification is that of dividing a set of observations (objects) into groups called **classes**, based on an analysis of their formal description. In **classification**, each observation (object) is assigned to a group or nominal category based on specific qualitative properties. Classification is a supervised task because it requires known classes for training samples. The labeling of a training set is usually done manually, with the involvement of specialists in the given field of study. It's also notable that if classes are not initially defined, then there will be a problem with clustering. Furthermore, in the classification task, there may be more than two classes (multi-class), and each of the objects may belong to more than one class (intersecting).

In this chapter, we will discuss various approaches to solving a classification task with machine learning. We are going to look at some of the most well-known and widespread algorithms, which are logistic regression, **support vector machine (SVM)**, and **k-nearest neighbors (kNN)**. Logistic regression is one of the most straightforward algorithms based on linear regression and a special loss function. SVM is based on a concept of support vectors that helps to build a decision boundary to separate data. This approach can be effectively used with high-dimensional data. kNN has a simple implementation algorithm that uses the idea of data compactness. Also, we will show how the multi-class classification problem can be solved with the algorithms mentioned previously. We will implement program examples to see how to use these algorithms to solve the classification task with different C++ libraries.

The following topics are covered in this chapter:

- An overview of classification methods
- Exploring various classification methods
- Examples of using C++ libraries for dealing with the classification task

Technical requirements

The required technologies and installations for this chapter include the following:

- The `mlpack` library
- The `Dlib` library
- The `Flashlight` library
- A modern C++ compiler with C++20 support
- CMake build system version ≥ 3.10

The code files for this chapter can be found at the following GitHub repo: <https://github.com/PacktPublishing/Hands-On-Machine-Learning-with-C-Second-Edition/tree/main/Chapter07>.

An overview of classification methods

Classification is a fundamental task in **applied statistics**, **machine learning**, and **artificial intelligence (AI)**. This is because classification is one of the most understandable and easy-to-interpret data analysis technologies, and classification rules can be formulated in a natural language. In machine learning, a classification task is solved using supervised algorithms because the classes are defined in advance, and the objects in the training set have class labels. Analytical models that solve a classification task are called **classifiers**.

Classification is the process of moving an object to a predetermined class based on its formalized features. Each object in this problem is usually represented as a vector in N -dimensional space. Each dimension in that space is a description of one of the features of the object.

We can formulate the classification task with mathematical notation. Let X denote the set of descriptions of objects, and Y be a finite set of names or class labels. There is an unknown objective function—namely, the mapping $y^* : X \rightarrow Y$, whose values are known only on the objects of the final training sample, $X^m = (x_1, y_1), \dots, (x_m, y_m)$. So, we have to construct an $a : X \rightarrow Y$ algorithm, capable of classifying an $x \in X$ arbitrary object. In mathematical statistics, classification problems are also called discriminant analysis problems.

The classification task is applicable to many areas, including the following:

- **Trade:** The classification of customers and products allows a business to optimize marketing strategies, stimulate sales, and reduce costs
- **Telecommunications:** The classification of subscribers allows a business to appraise customer loyalty, and therefore develop loyalty programs

- **Medicine and health care:** Assisting the diagnosis of disease by classifying the population into risk groups
- **Banking:** The classification of customers is used for credit-scoring procedures

Classification can be solved by using the following methods:

- Logistic regression
- The kNN method
- SVM
- Discriminant analysis
- Decision trees
- Neural networks

We looked into discriminant analysis in *Chapter 6, Dimensionality Reduction*, as an algorithm for dimensionality reduction, but most libraries provide an **application programming interface (API)** for working with the discriminant analysis algorithm as a classifier, too. We will discuss decision trees in *Chapter 9, Ensemble Learning*, focusing on algorithm ensembles. We will also discuss neural networks in the chapter that follows this: *Chapter 10, Neural Networks for Image Classification*.

Now we've discussed what the classification task is, let's look at various classification methods.

Exploring various classification methods

Nowadays, **deep learning** has become increasingly popular for classification tasks too, especially when dealing with complex and high-dimensional data such as images, audio, and text. Deep neural networks can learn hierarchical representations of data that allow them to perform accurate classification. In this chapter, we concentrate on more classical classification approaches because they are still applicable and usually require fewer computational resources. Specifically, we will discuss some of the classification methods such as logistic regression, **kernel ridge regression (KRR)**, the kNN method, and SVM approaches.

Logistic regression

Logistic regression determines the degree of dependence between the categorical dependent and one or more independent variables by using the logistic function. It aims to find the values of the coefficients for the input variables, as with linear regression. The difference, in the case of logistic regression, is that the output value is converted by using a non-linear (logistic) function. The logistic function has an S-shaped curve and converts any value to a number between 0 and 1. This property is useful because we can apply the rule to the output of the logistic function to bind 0 and 1 to a class prediction. The following screenshot shows a logistic function graph:

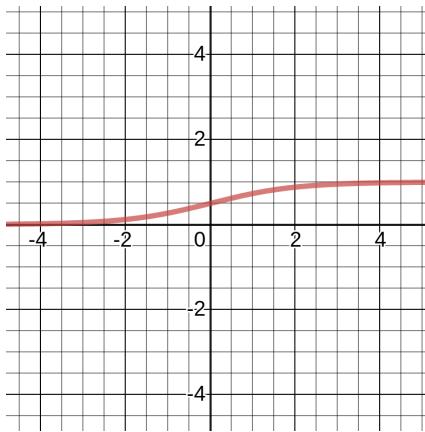


Figure 7.1 – Logistic function

For example, if the result of the function is less than 0.5, then the output is 0. Prediction is not just a simple answer (+1 or -1) either, and we can interpret it as a probability of being classified as +1.

In many tasks, this interpretation is an essential business requirement. For example, in the task of credit-scoring, where logistic regression is traditionally used, the probability of a loan being defaulted on is a common prediction. As with the case of linear regression, logistic regression performs the task better if outliers and correlating variables are removed. The logistic regression model can be quickly trained and is well-suited for binary classification problems.

The basic idea of a linear classifier is that the feature space can be divided by a hyperplane into two half-spaces, in each of which one of the two values of the target class is predicted. If we can divide a feature space without errors, then the training set is called **linearly separable**. Logistic regression is a unique type of linear classifier, but it is able to predict the probability of p_+ , attributing the example of \vec{x}_i to the class +, as illustrated here:

$$p_+ = P(y_i = 1 \mid \vec{x}_i, \vec{w})$$

Consider the task of binary classification, with labels of the target class denoted by +1 (positive examples) and -1 (negative examples). We want to predict the probability of $p_+ \in [0, 1]$; so, for now, we can build a linear forecast using the following optimization technique: $b(\vec{x}) = \vec{w}^T \vec{x} \in \mathbb{R}$. So, how do we convert the resulting value into a probability whose limits are [0, 1]? This approach requires a specific function. In the logistic regression model, the specific function $\sigma(z) = \frac{1}{1 + \exp^{-z}}$ is used for this.

Let's denote $P(X)$ by the probability of the occurring event X . The probability odds ratio $OR(X)$ is determined from $\frac{P(X)}{1 - P(X)}$. This is the ratio of the probabilities of whether the event will occur or not. We can see that the probability and the odds ratio both contain the same information. However, while $P(X)$ is in the range of 0 to 1, $OR(X)$ is in the range of 0 to ∞ . If you calculate the logarithm of $OR(X)$ (known as the **logarithm of the odds**, or the **logarithm of the probability ratio**), it is easy to see that the following applies: $\log OR(X) \in \mathbb{R}$.

Using the logistic function to predict the probability of p_+ , it can be obtained from the probability ratio (for the time being, let's assume we have the weights too) as follows:

$$p_+ = \frac{OR_+}{1 + OR_+} = \frac{\exp^{\vec{w}^T \vec{x}}}{1 + \exp^{\vec{w}^T \vec{x}}} = \frac{1}{1 + \exp^{-\vec{w}^T \vec{x}}} = \sigma(\vec{w}^T \vec{x})$$

So, the logistic regression predicts the probability of classifying a sample to the "+" class as a sigmoid transformation of a linear combination of the model weights vector, as well as the sample's features vector, as follows:

$$p_+(x_i) = P(y_i = 1 | \vec{x}_i, \vec{w}) = \sigma(\vec{w}^T \vec{x}_i)$$

From the maximum likelihood principle, we can obtain an optimization problem that the logistic regression solves—namely, the minimization of the logistic loss function. For the "-" class, the probability is determined by a similar formula, as illustrated here:

$$p_-(\vec{x}_i) = P(y_i = -1 | \vec{x}_i, \vec{w}) = 1 - \sigma(\vec{w}^T \vec{x}_i) = \sigma(-\vec{w}^T \vec{x}_i)$$

The expressions for both classes can be combined into one, as illustrated here:

$$P(y = y_i | \vec{x}_i, \vec{w}) = \sigma(y_i \vec{w}^T \vec{x}_i)$$

Here, the expression $M(\vec{x}_i) = y_i \vec{w}^T \vec{x}_i$ is called the margin of classification of the \vec{x}_i object. The classification margin can be understood as a model's *confidence* in the object's classification. An interpretation of this margin is as follows:

- If the margin vector's absolute value is large and positive, the class label is set correctly, and the object is far from the separating hyperplane. Such an object is therefore classified confidently.
- If the margin is large (by modulo) but negative, then the class label is set incorrectly. The object is far from the separating hyperplane. Such an object is most likely an anomaly.
- If the margin is small (by modulo), then the object is close to the separating hyperplane. In this case, the margin sign determines whether the object is correctly classified.

In the discrete case, the likelihood function $f(x_1, \dots, x_n, \theta)$ can be interpreted as the probability that the sample X_1, \dots, X_n is equal to x_1, \dots, x_n in the given set of experiments. Furthermore, this probability depends on θ , as illustrated here:

$$f(\mathbf{x}, \theta) = \prod_{i=1}^n f_\theta(x_i). \dots. ta(X_n = x_n) = P_\theta(X_1 = x_1, \dots, X_n = x_n).$$

The maximum likelihood estimate $\hat{\theta}$ for the unknown parameter θ is called the value of θ , for which the function $f(\mathbf{x}, \theta)$ reaches its maximum (as a function of θ , with fixed x_1, \dots, x_n), as illustrated here:

$$\hat{\theta} = \arg \max_{\theta} f(\mathbf{X}, \theta).$$

Now, we can write out the likelihood of the sample—namely, the probability of observing the given vector \vec{y} in the sample X . We make one assumption—objects arise independently from a single distribution, as illustrated here:

$$P(\vec{y} | X, \vec{w}) = \prod_{i=1}^{\ell} P(y = y_i | \vec{x}_i, \vec{w})$$

Let's take the logarithm of this expression since the sum is much easier to optimize than the product, as follows:

$$\begin{aligned} \log P(\vec{y} | X, \vec{w}) &= \log \prod_{i=1}^{\ell} P(y = y_i | \vec{x}_i, \vec{w}) \\ &= - \sum_{i=1}^{\ell} \log(1 + \exp^{-y_i \vec{w}^T \vec{x}_i}) \end{aligned}$$

In this case, the principle of maximizing the likelihood leads to a minimization of the expression, as illustrated here:

$$\mathcal{L}_{\text{log}}(X, \vec{y}, \vec{w}) = \sum_{i=1}^{\ell} \log(1 + \exp^{-y_i \vec{w}^T \vec{x}_i}).$$

This formula is a logistic loss function, summed over all objects of the training sample. Usually, it is a good idea to add some regularization to a model to deal with overfitting. **L2 regularization** of logistic regression is arranged in much the same way as for the ridge regression (**linear regression** with regularization). However, it is common to use the controlled variable decay parameter C that is used in SVM models, where it denotes soft margin parameter denotation. So, for logistic regression, C is equal to the inverse regularization coefficient $c = \frac{1}{\lambda}$. The relationship between C and λ would be the following: lowering C would strengthen the regularization effect. Therefore, instead of the functional $\mathcal{L}_{\text{log}}(X, \vec{y}, \vec{w})$, the following function should be minimized:

$$J(X, \vec{y}, \vec{w}) = \arg \min_{\vec{w}} (C \sum_{i=1}^{\ell} \log(1 + \exp^{-y_i \vec{w}^T \vec{x}_i}) + |\vec{w}|^2)$$

For this function minimization, we can apply different methods—for example, the method of least squares, or the **gradient descent** method. The vital issue with logistic regression is that it is generally a linear classifier, in order to deal with non-linear decision boundaries, which typically use polynomial features with original features as a basis for them. This approach was discussed in *Chapter 3* when we discussed polynomial regression.

KRR

KRR combines linear ridge regression (linear regression and L2 norm regularization) with the kernel trick and can be used for classification problems. It learns a linear function in the higher-dimensional space produced by the chosen kernel and training data. For non-linear kernels, it learns a non-linear function in the original space.

The model learned by KRR is identical to the SVM model, but these approaches have the following differences:

- The KRR method uses squared error loss, while the SVM model uses insensitive loss or hinge loss for classification
- In contrast to the SVM method, the KRR training can be completed in closed form so that it can be trained faster for medium-sized datasets
- The learned KRR model is non-sparse and can be slower than the SVM model when it comes to prediction times

Despite these differences, both approaches usually use L2 regularization.

SVM

The SVM method is a set of algorithms used for classification and regression analysis tasks. Considering that in an N -dimensional space, each object belongs to one of two classes, SVM generates an $(N-1)$ -dimensional hyperplane to divide these points into two groups. It's similar to an on-paper depiction of points of two different types that can be linearly divided. Furthermore, the SVM selects the hyperplane, which is characterized by the maximum distance from the nearest group elements.

The input data can be separated using various hyperplanes. The best hyperplane is a hyperplane with the maximum resulting separation and the maximum resulting difference between the two classes.

Imagine the data points on the plane. In the following case, the separator is just a straight line:

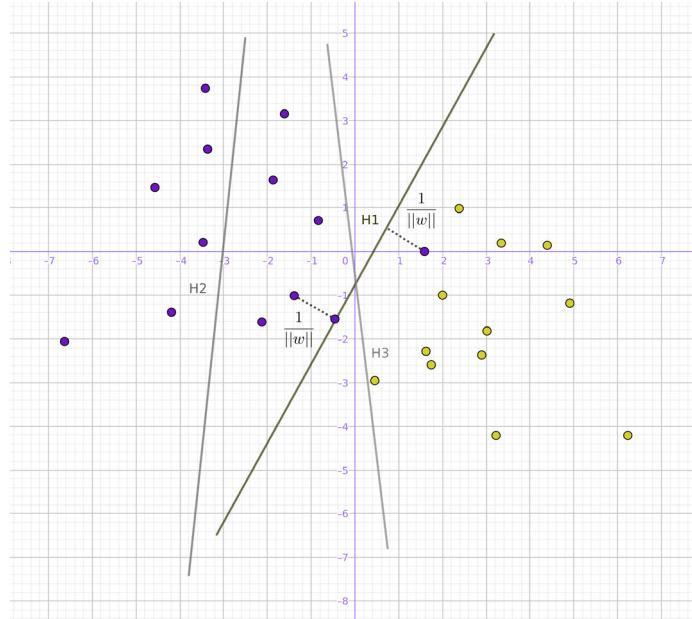


Figure 7.2 – Point separation line

Let's draw distinct straight lines that divide the points into two sets. Then, choose a straight line as far as possible from the points, maximizing the distance from it to the nearest point on each side. If such a line exists, then it is called the maximum margin hyperplane. Intuitively, a good separation is achieved due to the hyperplane itself (which has the longest distance to the nearest point of the training sample of any class), since, in general, the bigger the distance, the smaller the classifier error.

Consider learning samples given with the set $D = \{(x_i, y_i)\}$ consisting of n objects with p parameters, where y takes the values -1 or 1, thus defining the point's classes. Each point x is a vector of the dimension p . Our task is to find the maximum margin hyperplane that separates the observations. We can use analytic geometry to define any hyperplane as a set of points x that satisfy the condition, as illustrated here:

$$g(x) = w^t x + w_0$$

Here, $g(x) > 0 \Rightarrow x \in \text{class}[1]$ and $g(x) < 0 \Rightarrow x \in \text{class}[2]$.

Thus, the linear separating (discriminant) function is described by the equation $g(x)=0$. The distance from the point to the separating function $g(x)=0$ (the distance from the point to the plane) is equal to the following:

$$\frac{|w^t x + w_0|}{\|w\|}$$

x_i lies in the closure of the boundary that is $|w^t x_i + w_0| = 1$. The border, which is the width of the dividing strip, needs to be as large as possible. Considering that the closure of the boundary satisfies the condition $|w^t x_i + w_0| = 1$, then the distance from x_i to $g(x) = 0$ is as follows:

$$\frac{|w^t x + w_0|}{\|w\|} = \frac{1}{\|w\|}$$

Thus, the width of the dividing strip is $\frac{2}{\|w\|}$. To exclude points from the dividing strip, we can write out the following conditions:

$$\begin{cases} w^t x_i + w_0 \geq 1, & \text{if } x_i \text{ belongs to the first class,} \\ w^t x_i + w_0 \leq -1, & \text{if } x_i \text{ belongs to the second class.} \end{cases}$$

Let's also introduce the index function u_i that shows to which class x_i belongs, as follows:

$$\begin{cases} u_i = 1, & \text{if } x_i \text{ belongs to the first class,} \\ u_i = -1, & \text{if } x_i \text{ belongs to the second class.} \end{cases}$$

Thus, the task of choosing a separating function that generates a corridor of the greatest width can be written as follows:

$$J(w) = \frac{1}{2} \|w\|^2 \rightarrow \min$$

The $J(w)$ function was introduced with the assumption that $u_i (w^t x_i + w_0) \geq 1$ for all i . Since the objective function is quadratic, this problem has a unique solution.

According to the Kuhn-Tucker theorem, this condition is equivalent to the following problem:

$$L(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j u_i u_j x_i^T x_j \rightarrow \max$$

This is provided that $\alpha > 0 \forall i$ and $\sum_{i=1}^n \alpha_i u_i = 0$, where $\alpha = \{\alpha_1, \dots, \alpha_n\}$, are new variables. We can rewrite $L(\alpha)$ in the matrix form, as follows:

$$L(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix}^T H \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix},$$

The H coefficients of the matrix can be calculated as follows:

$$H_{i,j} = u_i u_j x_i^T x_j$$

Quadratic programming methods can solve the task $L(\alpha) \rightarrow \max$.

After finding the optimal $\alpha = \{\alpha_1, \dots, \alpha_n\}$ for every i , one of the two following conditions is fulfilled:

- $\alpha_i = 0$ (i corresponds to a non-support vector)
- $\alpha_i \neq 0$ and $u_i (w^T x_i + w_0 - 1) = 0$ (i corresponds to a support vector)

Then, w can be found from the relation $w = \sum_{i=1}^n \alpha_i u_i x_i$, and the value of w_0 can be determined, considering that for any $\alpha_i > 0$ and $\alpha_i [u_i (w^T x_i + w_0) - 1] = 0$, as follows:

$$w_0 = \frac{1}{u_i} - w^T x_i$$

So, we can calculate w_0 with the provided formula considering the provided conditions.

Finally, we can obtain the discriminant function, illustrated here:

$$g(x) = \left(\sum \{\alpha_i u_i x_i | x_i \in S\} \right)^T x + w_0$$

Note that the summation is not carried out over all vectors, but only over the set S , which is the set of support vectors $S = \{x_i | \alpha_i \neq 0\}$.

Unfortunately, the described algorithm is realized only for linearly separable datasets, which, in itself, occurs rather infrequently. There are two approaches for working with linearly non-separable data.

One of them is called a soft margin, which chooses a hyperplane that divides the training sample as purely (with minimal error) as possible, while at the same time maximizing the distance to the nearest point on the training dataset. For this, we have to introduce additional variables, ξ_i , which characterize the magnitude of the error on each object x_i . Furthermore, we can introduce the penalty for the total error into the goal functional, like this:

$$\begin{cases} \frac{1}{2} \|w\|^2 + \lambda \sum_{i=1}^n \xi_i \rightarrow \min, \\ u_i (w^t x_i + w_0) \geq 1 - \xi_i, i = 1, \dots, n, \\ \xi_i \geq 0, i = 1, \dots, n, \end{cases}$$

Here, λ is a method tuning parameter that allows you to adjust the relationship between maximizing the width of the dividing strip and minimizing the total error. The value of the penalty ξ_i for the corresponding object x_i depends on the location of the object x_i relative to the dividing line. So, if x_i lies on the opposite side of the discriminant function, then we can assume the value of the penalty $\xi_i > 1$, if x_i lies in the dividing strip, and comes from its class. The corresponding weight is, therefore, $0 < \xi_i < 1$. In the ideal case, we assume that $\xi_i < 0$. The resulting problem can then be rewritten as follows:

$$J(w, \xi_1, \dots, \xi_n) = \frac{1}{2} \|w\|^2 + \beta \sum_{i=1}^n I(\xi_i > 0) \rightarrow \min$$

Notice that elements that are not an ideal case are involved in the minimization process too, as illustrated here:

$$I(\xi_i > 0) = \begin{cases} 1, & \xi_i > 0, \\ 0, & \xi_i \leq 0, \end{cases}$$

Here, the constant β is the weight that takes into account the width of the strip. If β is small, then we can allow the algorithm to locate a relatively high number of elements in a non-ideal position (in the dividing strip). If β is vast, then we require the presence of a small number of elements in a non-ideal position (in the dividing strip). Unfortunately, the minimization problem is rather complicated due to the $I(\xi_i)$ discontinuity. Instead, we can use the minimization of the following:

$$J(w, \xi_1, \dots, \xi_n) = \frac{1}{2} \|w\|^2 + \beta \sum_{i=1}^n \xi_i$$

This occurs under restrictions $\forall i$, as illustrated here:

$$\begin{cases} u_i (w^t x_i + w_0) \geq 1 - \xi_i, \\ \xi_i \geq 0. \end{cases}$$

Another idea of the SVM method in the case of the impossibility of a linear separation of classes is the transition to a space of higher dimension, in which such a separation is possible. While the original problem can be formulated in a finite-dimensional space, it often happens that the samples for discrimination are not linearly separable in this space. Therefore, it is suggested to map the original finite-dimensional space into a larger dimension space, which makes the separation much easier. To keep the computational load reasonable, the mappings used in support vector algorithms provide ease of calculating points in terms of variables in the original space, specifically in terms of the kernel function.

First, the function of the mapping $\varphi(x)$ is selected to map the data of x into a space of a higher dimension. Then, a non-linear discriminant function can be written in the form $g(x) = w^t \varphi(x) + w_0$. The idea of the method is to find the kernel function $K(x_i, x_j) = \varphi(x_i)^T \varphi(x_j)$ and maximize the objective function, as illustrated here:

$$L(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j u_i u_j K(x_i, x_j) \rightarrow \max$$

Here, to minimize computations, the direct mapping of data into a space of a higher dimension is not used. Instead, an approach called the kernel trick is used—that is, $K(x, y)$, which is a kernel matrix. The kernel trick is a method used in machine learning algorithms to transform non-linear data into a higher-dimensional space where it becomes linearly separable. This allows to use of linear algorithms to solve non-linear problems because they are often simpler and more computationally efficient (see *Chapter 6* for the detailed explanation of the kernel trick).

In general, the more support vectors the method chooses, the better it generalizes. Any training example that does not constitute a support vector is correctly classified if it appears in the test set because the border between positive and negative examples is still in the same place. Therefore, the expected error rate of the support vector method is, as a rule, equal to the proportion of examples that are support vectors. As the number of measurements grows, this proportion also grows, so the method is not immune from the curse of dimensionality but it is more resistant to it than most algorithms.

It is also worth noting that the support vector method is sensitive to noise and data standardization.

Also, the method of SVMs is not only limited to the classification task but can also be adapted for solving regression tasks. So, you can usually use the same SVM software implementation for solving classification and regression tasks.

kNN method

The kNN is a popular classification method that is sometimes used in regression problems. It is one of the most natural approaches to classification. The essence of the method is to classify the current item by the most prevailing class of its neighbors. Formally, the basis of the method is the hypothesis of compactness: if the metric of the distance between the examples is clarified successfully, then similar examples are more likely to be in the same class. For example, if you don't know what type of product to specify in the ad for a Bluetooth headset, you can find five similar headset ads. If four of them are categorized as *Accessories* and only one as *Hardware*, common sense will tell you that your ad should probably be in the *Accessories* category.

In general, to classify an object, you must perform the following operations sequentially:

1. Calculate the distance from the object to other objects in the training dataset.
2. Select the k of training objects, with the minimal distance to the object that is classified.
3. Set the classifying object class to the class most often found among the nearest k neighbors.

If we take the number of nearest neighbors $k = 1$, then the algorithm loses the ability to generalize (that is, to produce a correct result for data not previously encountered in the algorithm) because the new item is assigned to the closest class. If we set too high a value, then the algorithm may not reveal many local features.

The function for calculating the distance must meet the following rules:

- $d(x, y) \geq 0$
- $d(x, y) = 0$ only when $x = y$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq d(x, y) + d(y, z)$ in the case when the x , y , and z points don't lie on one straight line

In this case, x , y , and z are feature vectors of compared objects. For ordered attribute values, Euclidean distance can be applied, as illustrated here:

$$D_E = \sqrt{\sum_i^n (x_i - y_i)^2}$$

In this case, n is the number of attributes.

For string variables that cannot be ordered, the difference function can be applied, which is set as follows:

$$dd(x, y) = \begin{cases} 0, & x = y \\ 1, & x \neq y \end{cases}$$

When finding the distance, the importance of the attributes is sometimes taken into account. Usually, attribute relevance can be determined subjectively by an expert or analyst, and is based on their own experience, expertise, and problem interpretation. In this case, each i^{th} square of the difference in the sum is multiplied by the coefficient Z_i . For example, if the attribute A is three times more important than the attribute B ($Z_A = 3$, $Z_B = 1$), then the distance is calculated as follows:

$$D_E = \sqrt{3(x_A - y_A)^2 + (x_B - y_B)^2}$$

This technique is called **stretching the axes**, which reduces the classification error.

The choice of class for the object of classification can also be different, and there are two main approaches to making this choice: *unweighted voting* and *weighted voting*.

For unweighted voting, we determine how many objects have the right to vote in the classification task by specifying the k number. We identify such objects by their minimal distance to the new object. The individual distance to each object is no longer critical for voting. All have equal rights in a class definition. Each existing object votes for the class to which it belongs. We assign a class with the most votes to a new object. However, there may be a problem if several classes score an equal number of votes. Weighted voting removes this problem.

During the weighted vote, we also take into account the distance to the new object. The smaller the distance, the more significant the contribution of the vote. The votes for the class formula is as follows:

$$\text{votes (class)} = \sum_{i=1}^n \frac{1}{d^2(X, Y_i)}$$

In this case, $d^2(X, Y_i)$ is the square of the distance from the known object Y_i to the new object X , while n is the number of known objects of the class for which votes are calculated. `class` is the name of the class. The new object corresponds to the class with the most votes. In this case, the probability that several classes gain the same number of votes is much lower. When $k = 1$, the new object is assigned to the class of the nearest neighbor.

A notable feature of the kNN approach is its laziness. Laziness means that the calculations begin only at the moment of the classification. When using training samples with the kNN method, we don't simply build the model but also do sample classification simultaneously. Note that the method of nearest neighbors is a well-studied approach (in machine learning, econometrics, and statistics, only linear regression is more well-known). For the method of nearest neighbors, there are quite a few crucial theorems that state that on *infinite* samples, kNN is the optimal classification method. The authors of the classic book *The Elements of Statistical Learning* consider kNN to be a theoretically ideal algorithm, the applicability of which is limited only by computational capabilities and the curse of dimensionality.

kNN is one of the simplest classification algorithms, so it is often ineffective in real-world tasks. The KNN algorithm has several disadvantages. Besides a low classification accuracy when we don't have enough samples, the kNN classifier's problem is the speed of classification: if there are N objects in the training set and the dimension of the space is K , then the number of operations for classifying a test sample can be estimated as $O(K * M * N)$. The dataset used for the algorithm must be representative. The model cannot be *separated* from the data: to classify a new example, you need to use all the examples.

The positive features include the fact that the algorithm is resistant to abnormal outliers since the probability of such a record falling into the number of kNN is small. If this happens, then the impact on the vote (uniquely weighted) with $k > 2$ is also likely to be insignificant, and therefore, the impact on the classification result is also small. The program implementation of the algorithm is relatively simple, and the algorithm result is easily interpreted. Experts in applicable fields, therefore, understand the logic of the algorithm, based on finding similar objects. The ability to modify the algorithm by using the most appropriate combination of functions and metrics allows you to adjust the algorithm for a specific task.

Multi-class classification

Most of the existing methods of multi-class classification are either based on binary classifiers or are reduced to them. The general idea of such an approach is to use a set of binary classifiers trained to separate different groups of objects from each other. With such a multi-class classification, various voting schemes for a set of binary classifiers are used.

In the **one-against-all** strategy for N classes, N classifiers are trained, each of which separates its class from all other classes. At the recognition stage, the unknown vector X is fed to all N classifiers. The membership of the vector X is determined by the classifier that gave the highest estimate. This approach can meet the problem of class imbalances when they arise. Even if the task of a multi-class classification is initially balanced (that is, it has the same number of training samples in each class), when training a binary classifier, the ratio of the number of samples in each binary problem increases

with an increase in the number of classes, which, therefore significantly affects tasks with a notable number of classes.

The **each-against-each** strategy allocates $\frac{N(N - 1)}{2}$ classifiers. These classifiers are trained to distinguish all possible pairs of classes of each other. For the input vector, each classifier gives an estimate of $f_{ij}(X)$, reflecting membership in the classes i and j . The result is a class with a maximum sum $\sum_{i \neq j} (g(f_{ij}(X)))$, where g is a monotonically non-decreasing function—for example, identical or logistic.

The **shooting tournament** strategy also involves training $\frac{N(N - 1)}{2}$ classifiers that distinguish all possible pairs of classes. Unlike the previous strategy, at the stage of classification of the vector X , we arrange a tournament between classes. We create a tournament tree, where each class has one opponent and only a winner can go to the next tournament stage. So, at each step, only one classifier determines the vector X class, then the *winning* class is used to determine the next classifier with the next pair of classes. The process is carried out until there is only one winning class left, which should be considered the result.

Some methods can produce multi-class classification immediately, without additional configuration and combinations. The kNN algorithms or neural networks can be considered examples of such methods.

Also, the logistic regression can be generalized for the multi-class case by using the softmax function. The softmax function is used to determine the probability that a sample belongs to a particular class, it looks as follows:

$$P(x^{(i)}) = \frac{\exp(\theta^{(k)\top} x^{(i)})}{\sum_{j=1}^K \exp(\theta^{(j)\top} x^{(i)})}$$

Here, K is the number of possible classes and the theta is a vector of learnable parameters. For the case when $K=2$, this expression reduces to the logistic regression:

$$P(x^{(i)}) = \frac{1}{\exp(\theta^{(1)\top} x^{(i)}) + \exp(\theta^{(2)\top} x^{(i)})} \begin{bmatrix} \exp(\theta^{(1)\top} x^{(i)}) \\ \exp(\theta^{(2)\top} x^{(i)}) \end{bmatrix} = \begin{bmatrix} \frac{1}{1+\exp((\theta^{(1)}-\theta^{(2)})^\top x^{(i)})} \\ 1 - \frac{1}{1+\exp((\theta^{(1)}-\theta^{(2)})^\top x^{(i)})} \end{bmatrix}$$

Replacing the vector difference:

$$\theta^{(2)} - \theta^{(1)}$$

with a single parameter vector, we can see that the probability for the one class will be predicted as follows:

$$\frac{1}{1 + \exp(-(\theta')^\top x^{(i)})}$$

For the second class, it will be the following:

$$1 - \frac{1}{1 + \exp(-(\theta')^\top x^{(i)})}$$

As you can see, these expressions are equal to the logistic regression we already saw.

Now we have become familiar with some of the most widespread classification algorithms, let's look at how to use them in different C++ libraries.

Examples of using C++ libraries for dealing with the classification task

Let's now see how to use the methods we've described for solving a classification task on artificial datasets, which we can see in the following screenshot:

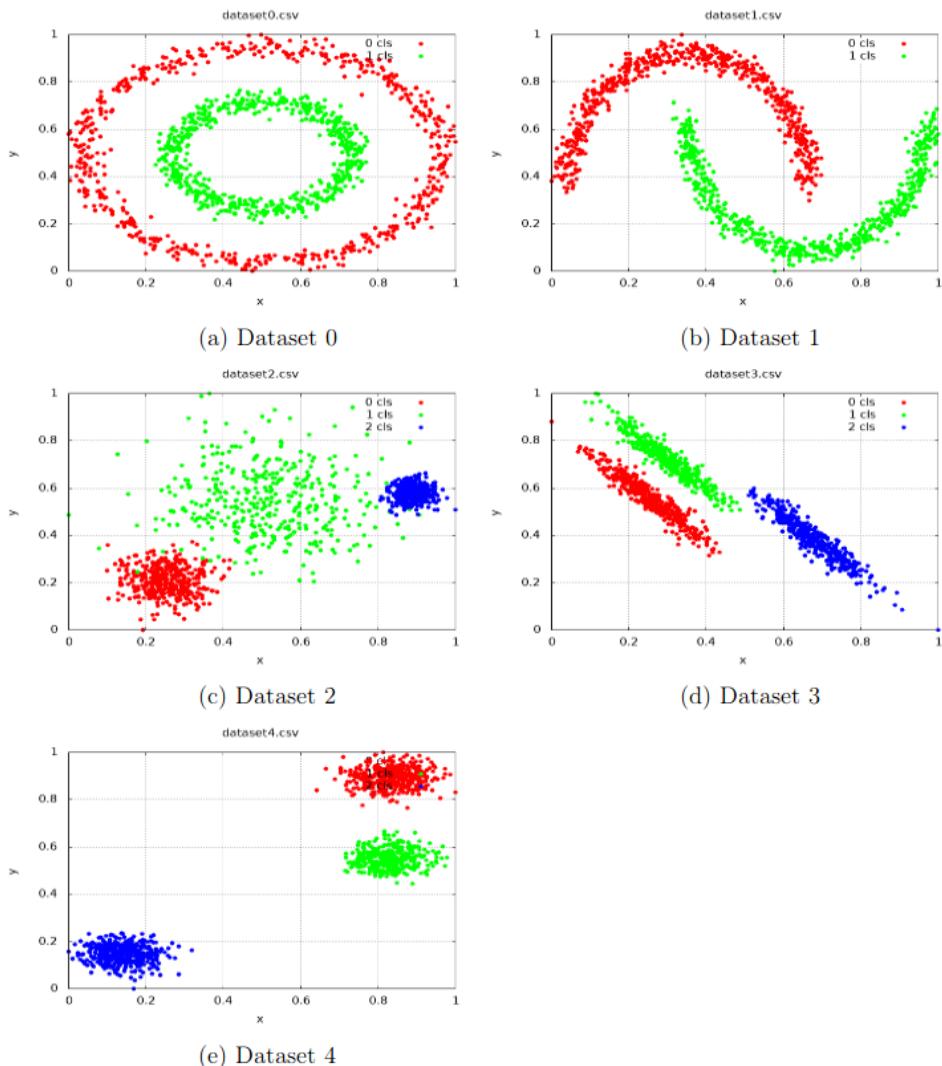


Figure 7.3 – Artificial datasets

As we can see, these datasets contain two and three different classes of objects, so it makes sense to use methods for multi-class classification because such tasks appear more often in real life; they can be easily reduced to binary classification.

Classification is a supervised technique, so we usually have a training dataset, as well as new data for classification. To model this situation, we will use two datasets in our examples, one for training and one for testing. They come from the same distribution in one large dataset. However, the test set won't be used for training; therefore, we can evaluate the accuracy metric and see how well models perform and generalize.

Using the `mlpack` library

In this section, we show how to use the `mlpack` library for solving the classification task. This library provides the implementation for the three main types of classification algorithms: logistic regression, softmax regression, and SVM.

With softmax regression

The `mlpack` library implements multi-class logistic regression in the `SoftmaxRegression` class. Using this class is very simple. We have to initialize an object with the number of samples that will be used for training and the number of classes. Let's say we have the following objects as training data and labels:

```
using namespace mlpack;
size_t num_classes;
arma::mat train_input;
arma::Row<size_t> train_labels;
```

Then we can initialize an object of `SoftmaxRegression` as follows:

```
SoftmaxRegression smr(train_input.n_cols, num_classes);
```

After we have the classifier object, we can train it and apply the classification function for some new data. The following code snippet shows how it can be done:

```
smr.Train(train_input, train_labels, num_classes);
arma::Row<size_t> predictions;
smr.Classify(test_input, predictions);
```

Having the prediction vector we can visualize it with the technique we are using in this book that is based on the `plotcpp` library. Notice that the `Train` and `Classify` method requires the `size_t` type for the class labels. The following screenshot shows the results of applying the `mlpack` implementation of the softmax regression algorithm to our datasets:

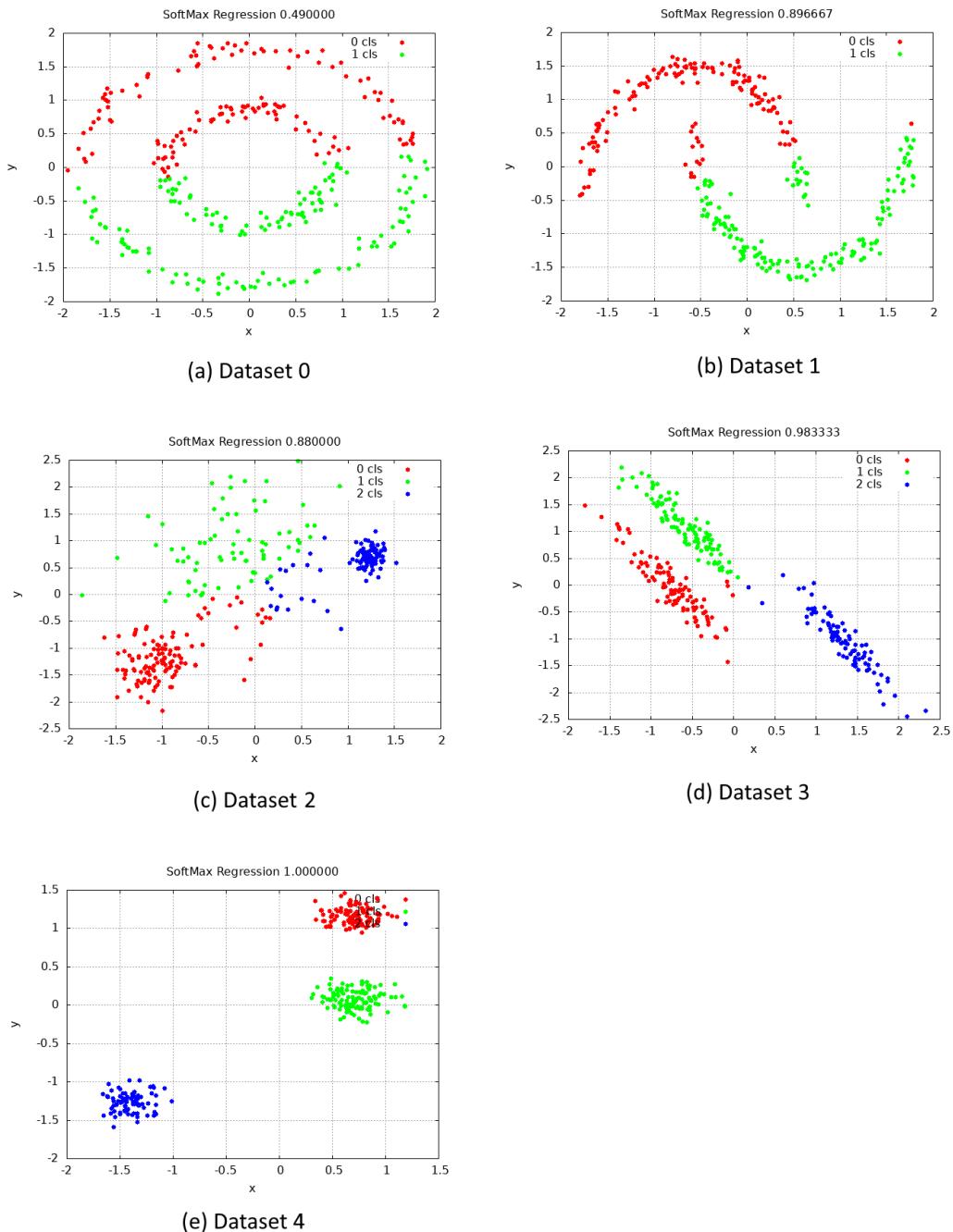


Figure 7.4 – Softmax classification with mlpack

Notice that we have classification errors in the **Dataset 0**, **Dataset 1**, and **Dataset 2** datasets, and other datasets were classified almost correctly.

With SVMs

The `mlpack` library also has an implementation of the multi-class SVM algorithm in the `LinearSVM` class. The library provides mostly the same API for all classification algorithms so the initialization of the classifier object is mostly the same as in the previous example. The main difference is that you can use the constructor without parameters. So, the object initialization will be the following:

```
mlpack::LinearSVM<> lsvm;
```

Then, we train the classifier with the `Train` method and apply the `Classify` method for new data samples, as follows:

```
lsvm.Train(train_input, train_labels, num_classes);
arma::Row<size_t> predictions;
lsvm.Classify(test_input, predictions);
```

The following screenshot shows the results of applying the `mlpack` implementation of the SVM algorithm to our datasets:

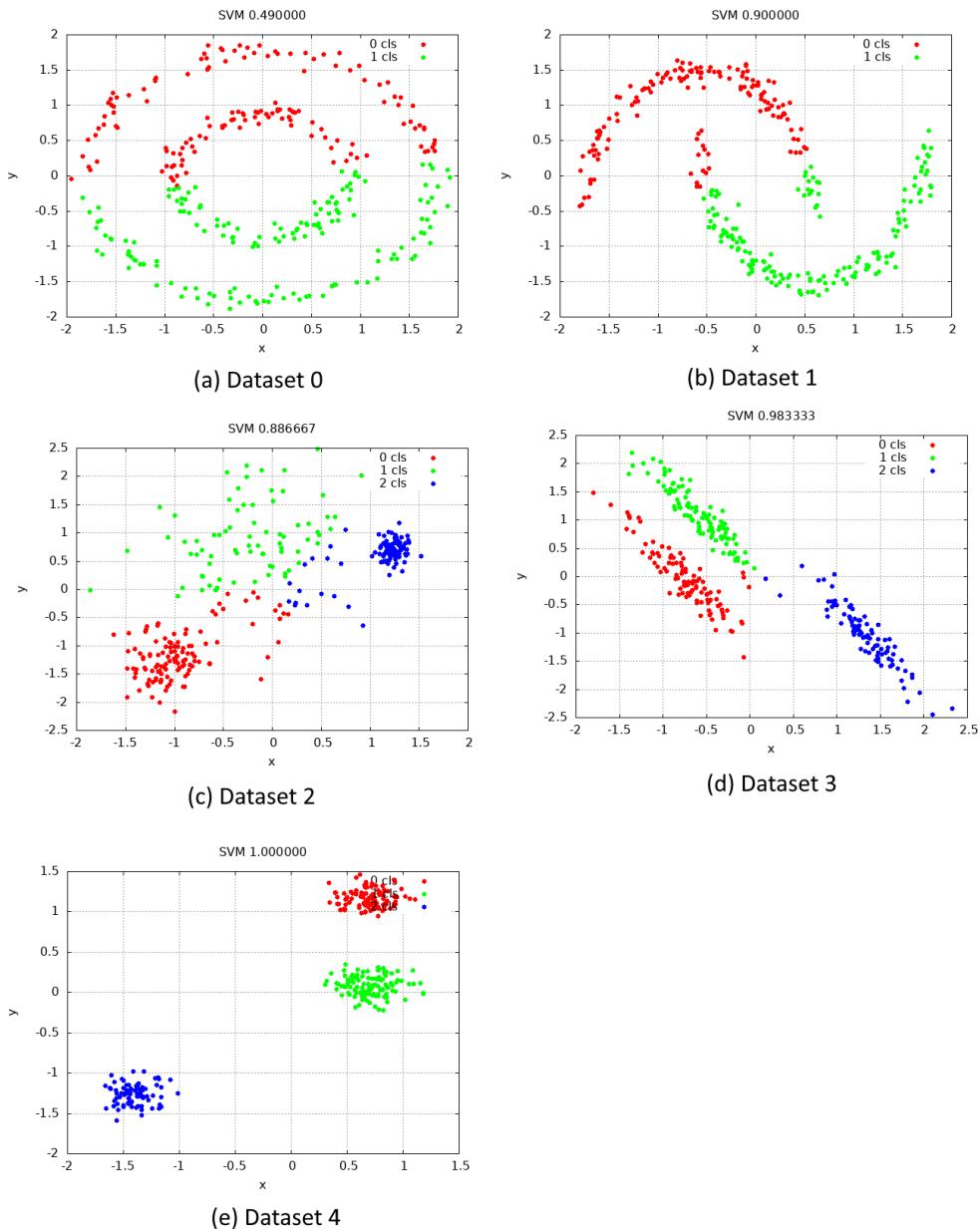


Figure 7.5 – SVM classification with mlpack

You can see that the results we got from the SVM method are pretty much the same as we got with the softmax regression.

With the linear regression algorithm

The `mlpack` library also implements the classic logistic regression algorithm in the `LogisticRegression` class. An object of this class can be applied to classify samples only into two classes. The usage API is the same as in the previous examples for the `mlpack` library. The typical application of this class will be the following:

```
using namespace mlpack;
LogisticRegression<> lr;
lr.Train(train_input, train_labels);

arma::Row<size_t> predictions;
lr.Classify(test_input, predictions);
```

The following screenshot shows the results of applying the two-class logistic regression to our datasets:

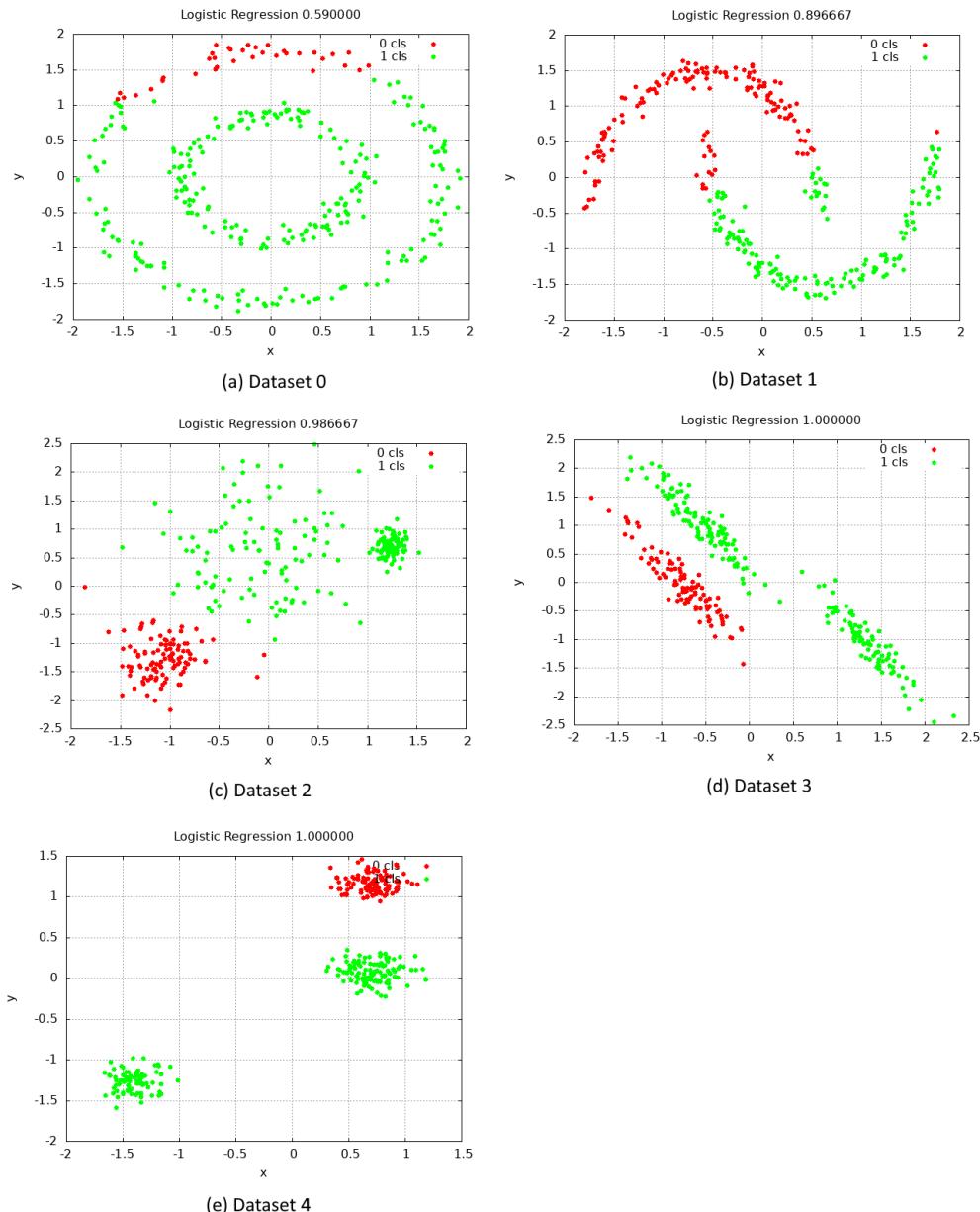


Figure 7.6 – Logistic regression classification with mlpack

You can see that we only got reasonable classifications for **Dataset 3** and **Dataset 4** as they can be separated with a straight line. However, due to the two-class limitation, we were not able to get the correct results.

Using the Dlib library

The `Dlib` library doesn't have many classification algorithms. There are two that are most applicable: *KRR* and *SVM*. These methods are implemented as binary classifiers, but for multi-class classification, this library provides the `one_vs_one_trainer` class, which implements the voting strategy. Note that this class can use classifiers of different types so that you can combine the KRR and the SVM for one classification task. We can also specify which classifiers should be used for which distinct classes.

With KRR

The following code sample shows how to use the `Dlib` KRR algorithm implementation for the multi-class classification:

```
void KRRClassification(const Samples& samples,
                      const Labels& labels,
                      const Samples& test_samples,
                      const Labels& test_labels) {
    using OVOtrainer = one_vs_one_trainer<
        any_trainer<SampleType>>;
    using KernelType = radial_basis_kernel<SampleType>;
    krr_trainer<KernelType> krr_trainer;
    krr_trainer.set_kernel(KernelType(0.1));
    OVOtrainer trainer;
    trainer.set_trainer(krr_trainer);
    one_vs_one_decision_function<OVOtrainer> df =
        trainer.train(samples, labels);
    // process results and estimate accuracy
    DataType accuracy = 0;
    for (size_t i = 0; i != test_samples.size(); i++) {
        auto vec = test_samples[i];
        auto class_idx = static_cast<size_t>(df(vec));
        if (static_cast<size_t>(test_labels[i]) == class_idx)
```

```
    ++accuracy;
    // ...
}
accuracy /= test_samples.size();
}
```

Firstly, we initialized the object of the `krr_trainer` class, and then we configured it with the instance of a kernel object. In this example, we used the `radial_basis_kernel` type for the kernel object, in order to deal with samples that can't be linearly separated. After we obtained the binary classifier object, we initialized the instance of the `one_vs_one_trainer` class and added this classifier to its stack with the `set_trainer()` method. Then, we used the `train()` method for training our multi-class classifier. As with most of the algorithms in the `Dlib` library, this one assumes that the training samples and labels have the `std::vector` type, whereby each element has a `matrix` type. The `train()` method returns a decision function—namely, the object that behaves as a functor, which then takes a single sample and returns a classification label for it. This decision function is an object of the `one_vs_one_decision_function` type. The following piece of code demonstrates how we can use it:

```
auto vec = test_samples[i];
auto class_idx = static_cast<size_t>(df(vec));
```

There is no explicit implementation for the accuracy metric in the `Dlib` library; so, in this example, accuracy is calculated directly as a ratio of correctly classified test samples against the total number of test samples.

The following screenshot shows the results of applying the `Dlib` implementation of the KRR algorithm to our datasets:

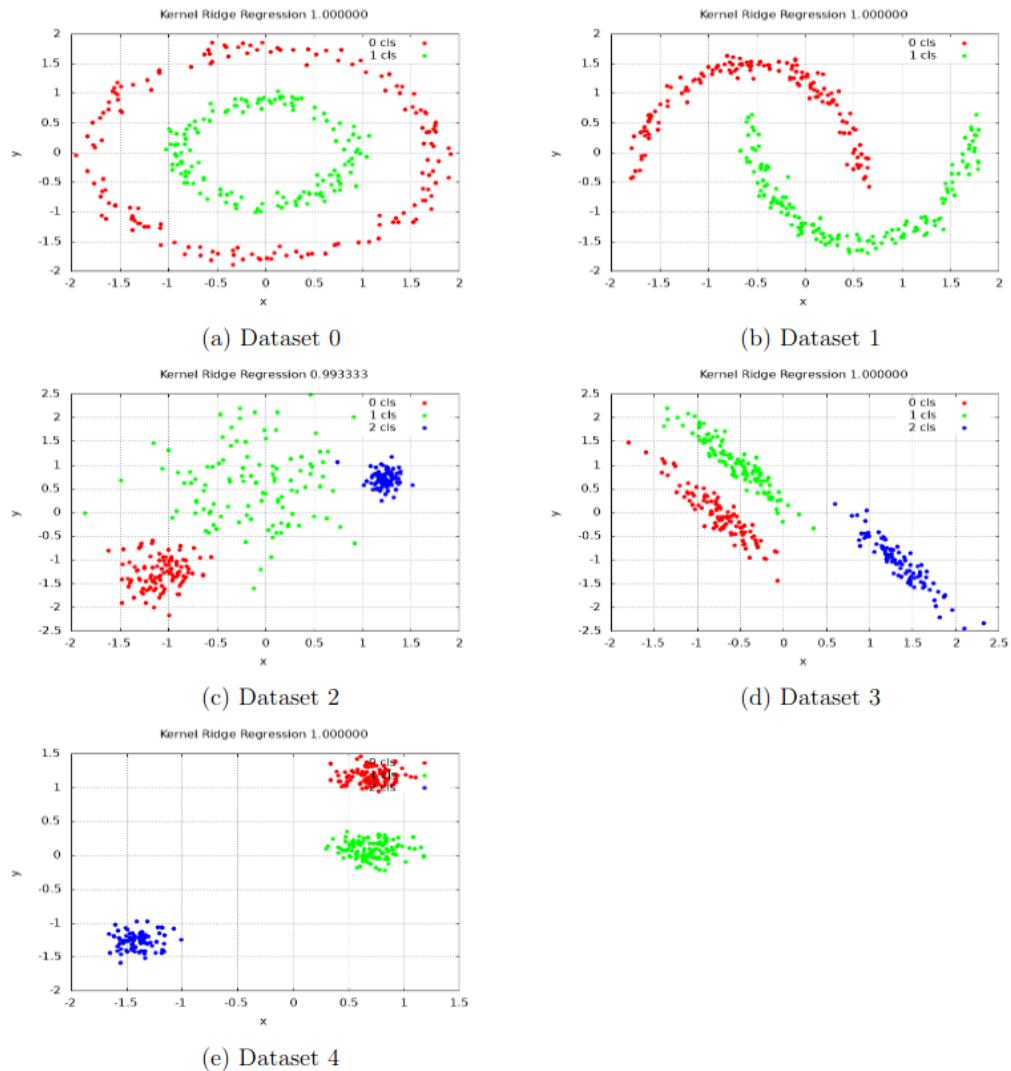


Figure 7.7 – KRR classification with Dlib

Notice that the KRR algorithm performed a correct classification on all datasets.

With SVM

The following code sample shows how to use the Dlib SVM algorithm implementation for multi-class classification:

```
void SVMClassification(const Samples& samples,
                      const Labels& labels,
                      const Samples& test_samples,
                      const Labels& test_labels) {
    using OVOtrainer = one_vs_one_trainer<
        any_trainer<SampleType>>;
    using KernelType = radial_basis_kernel<SampleType>;

    svm_nu_trainer<KernelType> svm_trainer;
    svm_trainer.set_kernel(KernelType(0.1));

    OVOtrainer trainer;
    trainer.set_trainer(svm_trainer);

    one_vs_one_decision_function<OVOtrainer> df =
        trainer.train(samples, labels);

    // process results and estimate accuracy
    DataType accuracy = 0;
    for (size_t i = 0; i != test_samples.size(); i++) {
        auto vec = test_samples[i];
        auto class_idx = static_cast<size_t>(df(vec));
        if (static_cast<size_t>(test_labels[i]) == class_idx)
            ++accuracy;
        // ...
    }
    accuracy /= test_samples.size();
}
```

This sample shows that the Dlib library also has a unified API for using different algorithms, and the main difference from the previous example is the object of the binary classifier. For the SVM classification, we used an object of the `svm_nu_trainer` type, which was also configured with the `kernel` object of the `radial_basis_kernel` type.

The following screenshot shows the results of applying the Dlib implementation of the SVM algorithm to our datasets:

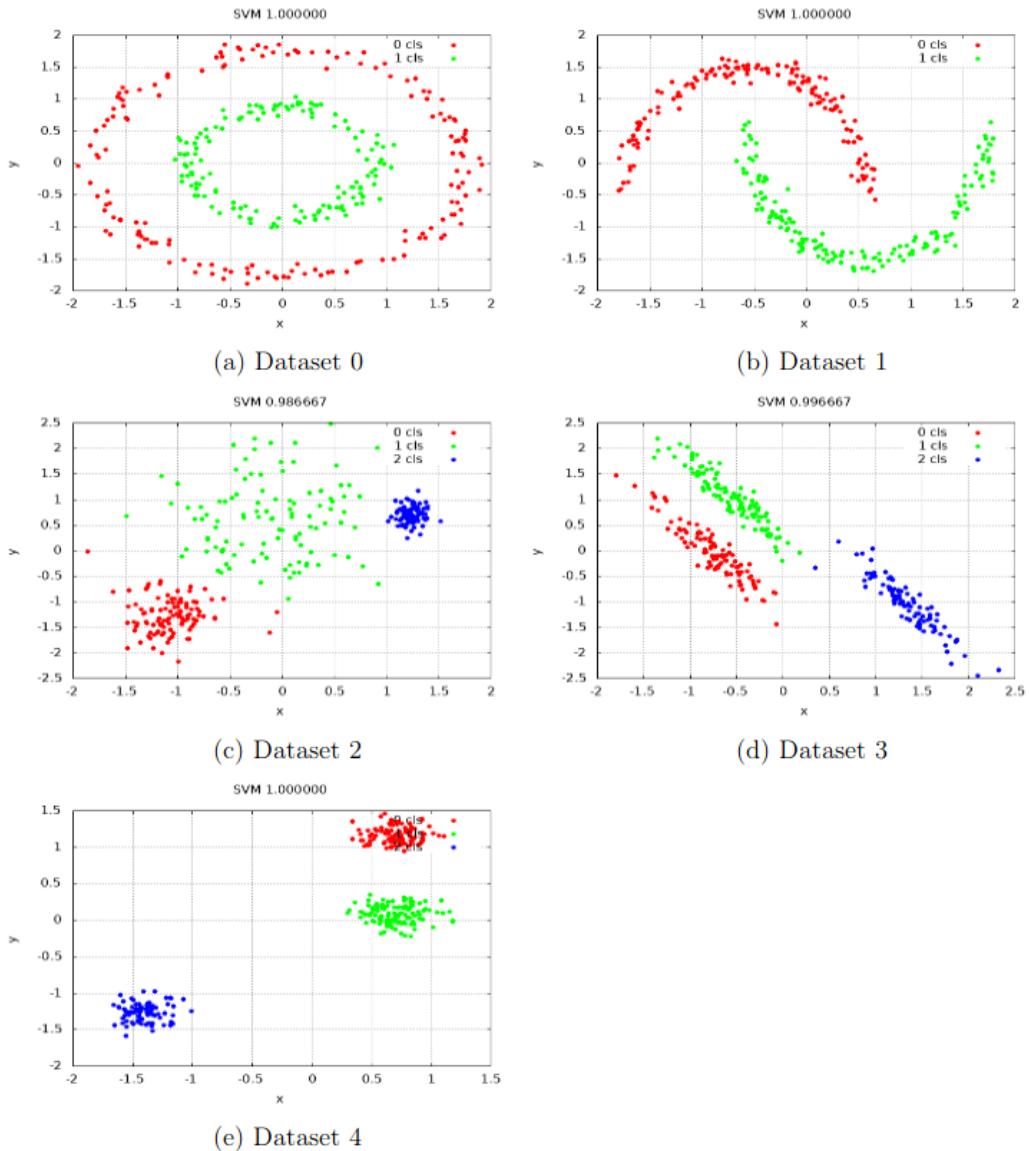


Figure 7.8 – SVM classification with Dlib

You can see the Dlib implementation of the SVM algorithm also did correct classification on all datasets because of the **radial basis function (RBF)** kernel trick usage. Remember that the mlpack implementation of the same algorithm made incorrect classification in some cases due to its linearity.

Using the Flashlight library

The Flashlight library doesn't have any special classes for the classification algorithms. But using the linear algebra primitives and auto-gradient facilities of the library, we can implement the logistic regression algorithm from scratch. Also, to handle the datasets that are non-linearly separable, the kernel trick approach will be implemented.

With logistic regression

The following example shows how to implement the two-class classification with the Flashlight library. Let's define a function for training a linear classifier; it will have the following signature:

```
f1::Tensor train_linear_classifier(  
    const f1::Tensor& train_x,  
    const f1::Tensor& train_y, float learning_rate);
```

`train_x` and `train_y` are the training samples and their labels correspondingly. The function's result is the learned parameters vector—in our case, defined with the `f1::Tensor` class. We are going to use the batched gradient descent algorithm to learn the parameters vector. So, we can use the Flashlight dataset types to simplify work with training data batches. The following code snippet shows how we can make a dataset object that will allow us to iterate over batches:

```
std::vector<f1::Tensor> fields{train_x, train_y};  
auto dataset = std::make_shared<f1::TensorDataset>(fields);  
int batch_size = 8;  
auto batch_dataset = std::make_shared<f1::BatchDataset>(  
    dataset, batch_size);
```

At first, we defined the regular dataset object with the `f1::TensorDataset` type and then we used it to create the object of the `f1::BatchDataset` type that was initialized with the batch size value also.

Next, we need to initialize the parameters vector that we will learn with the gradient descent, as follows:

```
auto weights = f1::Variable(f1::rand({  
    train_x.shape().dim(0), 1}), /*calcGrad=*/true);
```

Notice that we explicitly passed the `true` value as the last argument to enable the gradient calculation by the Flashlight autograd mechanism. Now, we are ready to define the training cycle with the predefined number of epochs. In each of the epochs, we will iterate over all batches in the dataset. So, such a cycle can be defined as follows:

```
int num_epochs = 100;
for (int e = 1; e <= num_epochs; ++e) {
    fl::Tensor epoch_error = fl::fromScalar(0);
    for (auto& batch : *batch_dataset) {
        auto x = fl::Variable(batch[0], /*calcGrad=*/false);
        auto y = fl::Variable(
            fl::reshape(batch[1], {1, batch[1].shape().dim(0)}),
            /*calcGrad=*/false);
    }
}
```

You can see two nested loops: the outer one for epochs and the inner one for batches. The `batch_dataset` object used in the `for` loop is compatible with C++ range based for loop construction, so it's easily used to access the batches. Also, notice that we defined two variables, `x` and `y`, with the `fl::Variable` type as we did for weights. Usage of this type makes it possible to pass tensor values into the autograd mechanism. And for these variables, we didn't configure the gradients calculation because they are not trainable parameters. Another important issue is that we used `fl::reshape` to make all tensor shapes compatible with the matrix multiplication that will be applied in the loss function calculation. The logistic regression loss function looks as follows:

$$\text{loss} = \sum \log(1 + \exp(-yw^T x))$$

In the code, we can implement it with the following lines:

```
auto z = fl::matmul(fl::transpose(weights), x);
auto loss = fl::sum(fl::log(1 + fl::exp(-1 * y * z)), /*axes=*/{1});
```

After we get the loss value, we can apply the gradient descent algorithm to correct the weights (parameters vector) according to the influence of the current training samples batch. The following code snippet shows how to do it:

```
loss.backward();
weights.tensor() -= learning_rate * weights.grad().tensor();
weights.zeroGrad();
```

Notice that the last step in the gradient zeroing was done to make it possible to learn something new from the next training sample and not mix gradients. At the end of the training cycle, the resulting parameters vector can be returned from the function as follows:

```
return weights.tensor();
```

And the following sample shows how our training function can be used :

```
fl::Tensor train_x;
fl::Tensor train_y;
auto weights = train_linear_classifier(
    train_x, train_y, /*learning_rate=*/0.1f);
```

Having the learned parameter vector, we can use it to classify a new data sample as follows:

```
fl::Tensor sample;
constexpr float threshold = 0.5;
auto p = fl::sigmoid(fl::matmul(fl::transpose(weights), sample));
if (p.scalar<float>() > threshold)
    return 1;
else
    return 0;
```

You can see that we implemented the logistic function call that returns a result into the `p` variable. The value of this variable can be interpreted as the probability of the event that the sample belongs to a particular class. We introduced the `threshold` variable to check the probability. If it is greater than this threshold, then we classify the sample as it has a class of 1; otherwise, it has a class of 0.

The following screenshot shows the results of applying Flashlight implementation of the logistic regression algorithm to our datasets with two classes:

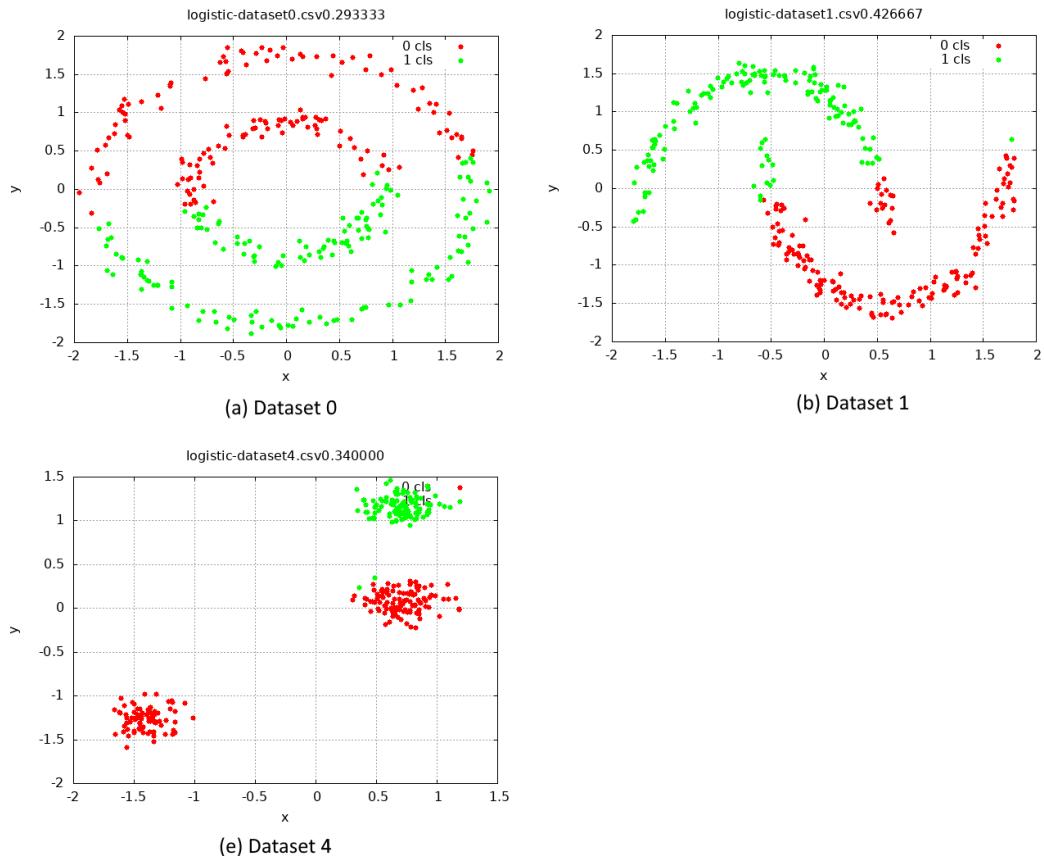


Figure 7.9 – Logistic regression classification with Flashlight

You can see that it fails to correctly classify **Dataset 0** and **Dataset 1** with a non-linear class boundary but successfully classified **Dataset 4** as it is linearly separable.

With logistic regression and kernel trick

To address the problem with non-linear class boundaries, we can apply the kernel trick. Let's see how we can implement it in the Flashlight library with the Gaussian kernel. The idea is to move our data samples into higher dimension space where they can be linearly separable. The Gaussian kernel function looks like as follows:

$$K(x, y) = \exp(\gamma \|x - y\|^2)$$

To make our calculations more computationally effective, we can rewrite them in the following way:

$$K(x, y) = \exp(\gamma(\|x\|^2 + \|y\|^2 - 2\|x\|\|y\|))$$

The following code sample shows this formula implementation:

```
fl::Tensor make_kernel_matrix(const fl::Tensor& x,
                             const fl::Tensor& y, float gamma) {
    auto x_norm = fl::sum(fl::power(x, 2), /*axes=*/{-1});
    x_norm = fl::reshape(x_norm, {x_norm.dim(0), 1});
    auto y_norm = fl::sum(fl::power(y, 2), /*axes=*/{-1});
    y_norm = fl::reshape(y_norm, {1, y_norm.dim(0)});
    auto k = fl::exp(-gamma * (x_norm + y_norm -
        2 * fl::matmul(fl::transpose(x), y)));
    return k;
}
```

The `make_kernel_matrix` function takes two matrices and applies the Gaussian kernel returning the single matrix. Let's see how we can apply it to our problem. At first, we apply it to our training dataset as follows:

```
constexpr float rbf_gamma = 100.f;
auto kx = make_kernel_matrix(train_x, train_x, rbf_gamma);
```

Notice that the function was called with the same `train_x` value for the two arguments. So, we moved our training dataset into the higher dimension space based on this training dataset. The gamma is a scaling hyperparameter that was configured manually in this example. Having this transformed dataset, we can train a classifier with the function that we created in the previous example as follows:

```
auto kweights = train_linear_classifier(kx, train_y, learning_rate);
```

Then, to use these weights (parameters vector), we should apply the kernel to the new data samples in the following way:

```
fl::Tensor sample;
auto k_sample = make_kernel_matrix(fl::reshape(sample, {
    sample.dim(0), 1}), train_x, rbf_gamma);
```

You can see that we used the reshaped new sample as the first argument and the training set tensor as the second argument. So, we transformed the new sample into the higher dimension space based on the original training data to preserve the same space properties. Then, we can apply the same classification procedure with a threshold as in the previous example, as follows:

```
constexpr float threshold = 0.5;
auto p = fl::sigmoid(fl::matmul(fl::transpose(kweights),
    fl::transpose(k_sample)));
if (p.scalar<float>() > threshold)
    return 1;
else
    return 0;
```

You can see that we just used the transformed weights tensor and transformed sample.

The following screenshot shows the results of applying the logistic regression with the kernel trick implementation to our two-class datasets:

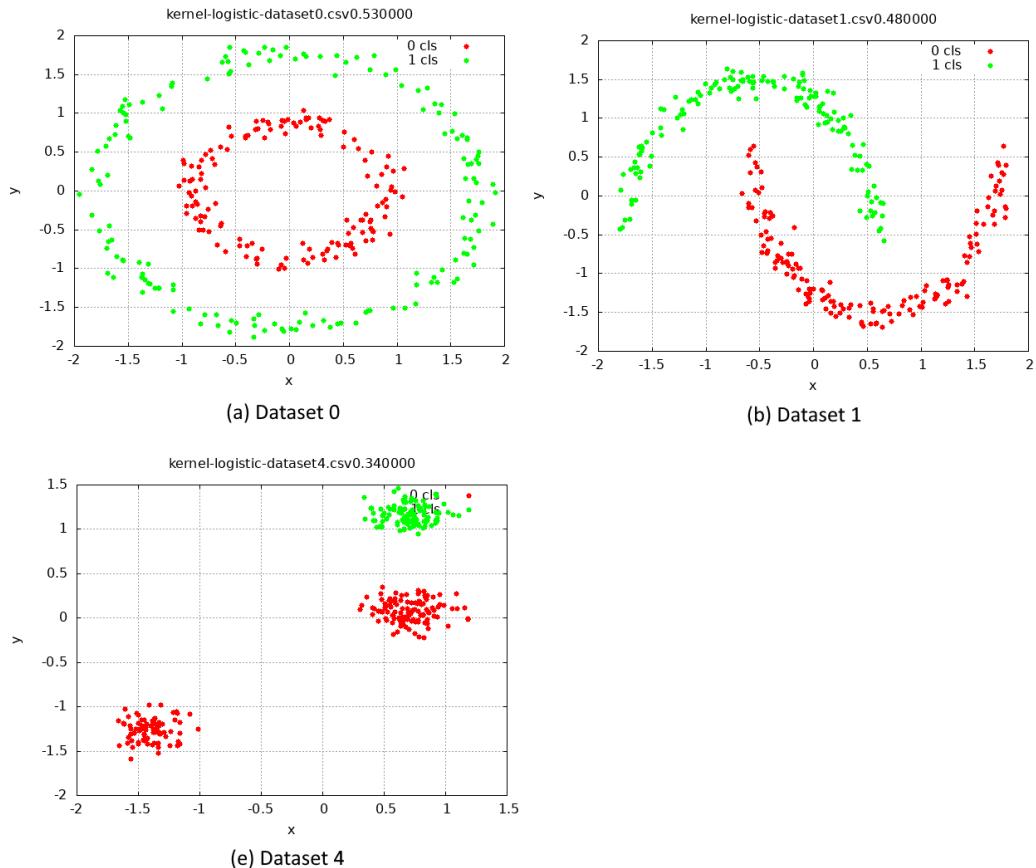


Figure 7.10 – Logistic regression with kernel trick classification with Flashlight

You can see that with the kernel trick logistic regression successfully classified data with non-linear class boundaries.

Summary

In this chapter, we discussed supervised machine learning approaches to solving classification tasks. These approaches use trained models to determine the class of an object according to its characteristics. We considered two methods of binary classification: logistic regression and SVMs. We looked at the approaches for the implementation of multi-class classification.

We saw that working with non-linear data requires additional improvements in the algorithms and their tuning. Implementations of classification algorithms differ in terms of performance, as well as the amount of required memory and the amount of time required for learning. Therefore, the classification algorithm's choice should be guided by a specific task and business requirements. Furthermore, their implementations in different libraries can produce different results, even for the same algorithm. Therefore, it makes sense to have several libraries for your software.

In the next chapter, we will discuss recommender systems. We will see how they work, which algorithms exist for their implementation, and how to train and evaluate them. In the simplest sense, recommender systems are used to predict which objects (goods or services) are of interest to a user. Examples of such systems can be seen in many online stores such as Amazon or on streaming sites such as Netflix, which recommend new content based on your previous consumption.

Further reading

- *Logistic Regression—Detailed Overview*: <https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc>
- Understanding SVM algorithm from examples (along with code): <https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/>
- Understanding Support Vector Machines: A Primer: <https://appliedmachinelearning.wordpress.com/2017/03/09/understanding-support-vector-machines-a-primer/>
- *Support Vector Machine: Kernel Trick; Mercer's Theorem*: <https://towardsdatascience.com/understanding-support-vector-machine-part-2-kernel-trick-mercers-theorem-e1e6848c6c4d>
- SVMs with Kernel Trick (lecture): https://ocw.mit.edu/courses/sloan-school-of-management/15-097-prediction-machine-learning-and-statistics-spring-2012/lecture-notes/MIT15_097S12_lec13.pdf
- *Support Vector Machines—Kernels and the Kernel Trick*: https://cogsys.uni-bamberg.de/teaching/ss06/hs_svm/slides/SVM_Seminarbericht_Hofmann.pdf
- *A Complete Guide to K-Nearest-Neighbors with Applications in Python and R*: <https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor>

8

Recommender Systems

Recommender systems are algorithms, programs, and services that are designed to use data to predict which objects (goods or services) are of interest to a user. There are two main types of recommender systems: *content-based* and *collaborative filtering*. **Content-based recommender systems** are based on data that's been collected from specific products. They recommend objects to a user that are similar to ones the user has previously acquired or shown interest in. **Collaborative filtering recommender systems** filter out objects that a user might like based on the reaction history of other, similar users of these systems. They also usually consider the user's previous reactions.

In this chapter, we'll learn how to implement recommender system algorithms based on both content and collaborative filtering. We're going to discuss different approaches for implementing collaborative filtering algorithms, implement systems using only the linear algebra library, and learn how to use the `mlpack` library to solve collaborative filtering problems. We'll be using the MovieLens dataset provided by GroupLens from a research lab in the Department of Computer Science and Engineering at the University of Minnesota: <https://grouplens.org/datasets/movielens/>.

The following topics will be covered in this chapter:

- An overview of recommender system algorithms
- Understanding the collaborative filtering method
- Examples of item-based collaborative filtering with C++

Technical requirements

To complete this chapter, you'll need the following:

- The `Eigen` library
- The `Armadillo` library
- The `mlpack` library

- A modern C++ compiler with C++20 support
- CMake build system version ≥ 3.10

The code files for this chapter can be found in this book's GitHub repository: <https://github.com/Packt Publishing/Hands-On-Machine-Learning-with-C-Second-Edition/tree/main/Chapter08>.

An overview of recommender system algorithms

A recommender system's task is to inform a user about an object that could be the most interesting to them at a given time. Often, such an object is a product or service, but it may be information—for example, in the form of a recommended news article.

Before we dive into the technicalities of the recommender system, let's look at some real-world scenarios where recommender systems are used to improve user experience and increase sales. The following are the most common applications:

- Recommender systems help online retailers suggest products that might interest a customer based on their past purchases, browsing history, and other data. This helps customers find relevant products more easily and increases the likelihood of conversion.
- Music and video streaming services use recommender systems to suggest music or videos based on a user's listening or viewing history. The goal is to provide personalized content recommendations that keep users engaged with the platform.
- Social media platforms such as Meta and Instagram use recommender systems to show users content from friends and pages they might be interested in. This keeps users engaged and spending time on the platform.
- Advertisers use recommender systems to target adverts at specific audiences based on their interests, demographics, and behavior. This improves the effectiveness of advertising campaigns.
- News websites, blogs, and search engines use recommender systems to recommend articles, stories, or search results based on user preferences and search history.
- Recommender systems can be used to suggest treatments, medications, or medical procedures based on patient data and medical research.
- Travel websites and hotel booking platforms use recommender systems to suggest travel destinations, accommodations, and activities based on a traveler's preferences, budget, and travel history.
- Educational platforms and online courses use recommender systems to personalize learning experiences by suggesting courses, materials, and learning paths based on student performance, interests, and goals.

- Video game platforms use recommender systems to suggest games based on player preferences, play style, and gaming history.

These are just a few examples of how recommender systems are applied in real-life scenarios. They've become an essential tool for businesses looking to improve customer engagement, increase sales, and provide personalized experiences.

Despite the many existing algorithms, we can divide recommender systems into several basic approaches. The most common are as follows:

- **Summary-based:** Non-personal models based on the average product rating
- **Content-based:** Models based on the intersection of product descriptions and user interests
- **Collaborative filtering:** Models based on interests of similar user groups
- **Matrix factorization:** Methods based on the preferences matrix's decomposition

The basis of any recommender system is the preferences matrix. It has all users of the service laid on one of the axes and recommendation objects on the other. The recommendation objects are usually called **items**. At the intersection of rows and columns (user, item), this matrix is filled with ratings that indicate user interest in a product, expressed on a given scale (for example, from 1 to 5), as illustrated in the following table:

	item1	item 2	item3
user1	1		
user2		2	4
user3	1	1	1
user4			5
user5	3	1	
user6		4	

Table 8.1 – User interest count

Users usually evaluate only a small number of the items in the catalog; the task of the recommender system is to summarize this information and predict the attitude the user might have toward other items. In other words, you need to fill in all the blank cells in the preceding table.

People's consumption patterns are different, and new products don't have to be recommended all the time. You can show repeated items—for example, when a user has bought something they'll need again. According to this principle, there are two groups of items:

- **Repeatable:** For example, shampoos or razors, which are always needed
- **Unrepeatable:** For example, books or films, which are rarely purchased repeatedly

If the product can't be attributed to one of these groups, it makes sense to determine the group type of repetitive purchases individually (someone usually buys only a specific brand, but someone else might try everything in the catalog).

Determining what product *interests* a user is also subjective. Some users need things only from their favorite category (conservative recommendations), while others respond more to non-standard goods (risky recommendations). For example, a video-hosting service may only recommend new series from their favorite TV series (conservative) but may periodically recommend new shows or new genres. Ideally, you should choose a strategy for displaying recommendations for each client separately by using generalized information about the client's preferences.

The essential part of datasets that are used to build recommendation models is user reactions to different objects or items. These reactions are typically called user ratings of objects. We can obtain user ratings in the following ways:

- **Explicit ratings:** The user gives their rating for the product, leaves a review, or *likes* the page.
- **Implicit ratings:** The user doesn't express their attitude, but an indirect conclusion can be made from their actions. For example, if they bought a product, it means they like it; if they read the description for a long time, it means they have serious interest.

Of course, explicit preferences are better. However, in practice, not all services allow users to express their interests clearly, and not all users have the desire to do so. Both types of assessments are often used in tandem and complement each other well.

It's also essential to distinguish between the terms *prediction* (the prediction of the degree of interest) and the *recommendation* itself (showing the recommendation). How to show something is a separate task from the task of *what to show*. *How to show* is a task that uses the estimates obtained in the prediction step, and can be implemented in different ways.

In this section, we discussed the basics of recommender systems. In the following sections, we'll look at the essential building blocks of recommender systems. Let's begin by looking at the main principles of content-based filtering, user and item-based collaborative filtering, and collaborative filtering based on matrix factorization.

Non-personalized recommendations

For non-personalized recommendations, the potential interest of the user is determined by the average rating of the product: *if everyone likes it, you'll like it too*. According to this principle, most services work when the user isn't authorized on the system.

Content-based recommendations

Personal recommendations use the maximum information available about the user—primarily, information about their previous purchases. Content-based filtering was one of the first approaches to be developed for personalized recommendations. In this approach, the product's description (content) is compared with the interests of the user, which are obtained from their previous assessments. The more the product meets these interests, the higher the potential interest of the user. The obvious requirement here is that all products in the catalog should have a description.

Historically, the subject of content-based recommendations was products with unstructured descriptions: films, books, or articles. Their features may be, for example, text descriptions, reviews, or casts. However, nothing prevents the use of usual numerical or categorical features.

Unstructured features are described in a text-typical way—vectors in the space of words (vector-space model). Each element of a vector is a feature that potentially characterizes the interest of the user. Similarly, an item (product) is a vector in the same space.

As users interact with the system (say, they buy films), the vector descriptions of the goods they've purchased merge (sum up and normalize) into a single vector and, thus, form the vector of a user's interests. Using this vector of interests, we can find the product and the description of which is closest to it—that is, we can solve the problem of finding the nearest neighbors.

When forming the vector space of a product presentation, instead of individual words, you can use shingles or n-grams (successive pairs of words, triples of words, or other numbers of words). This approach makes the model more detailed, but more data is required for training.

In different places of the description of the product, the weight of keywords may differ (for example, the description of the film may consist of a title, a brief description, and a detailed description). Product descriptions from different users can be weighed differently. For example, we can give more weight to active users who have many ratings. Similarly, you can weigh them by item. The higher the average rating of an object, the greater its weight (similar to PageRank). If the product description allows links to external sources, then you can also analyze all third-party information related to the product.

The cosine distance is often used to compare product representation vectors. This distance measures the value of proximity between two vectors.

When adding a new assessment, the vector of interests is updated incrementally (only for those elements that have changed). During the update, it makes sense to give a bit more weight to new estimates since the user's preferences may change. You'll notice that content-based filtering almost wholly repeats the query-document matching mechanism used in search engines such as Google. The only difference lies in the form of a search query—content filtering systems use a vector that describes the interests of the user, whereas search engines use keywords of the requested document. When search engines began to add personalization, this distinction was erased even more.

User-based collaborative filtering

This class of system began to develop in the 90s. Under this approach, recommendations are generated based on the interests of other, similar users. Such recommendations are the result of the **collaboration** of many users, hence the name of the method.

The classical implementation of the algorithm is based on the principle of **k-nearest neighbors (kNN)**. For every user, we look for the **k** most similar to them (in terms of preferences). Then, we supplement the information about the user with known data from their neighbors. So, for example, if it's known that your neighbors are delighted with a movie, and you haven't watched it for some reason, this is a great reason to recommend this movie.

The similarity is, in this case, a synonym for a *correlation* of interests and can be considered in many ways—Pearson's correlation, cosine distance, Jaccard distance, Hamming distance, and other types of distances.

The classical implementation of the algorithm has one distinct disadvantage—it's poorly applicable in practice due to the quadratic complexity of the calculations. As with any nearest neighbor method, it requires all pairwise distances between users to be calculated (and there may be millions of users). It's easy to calculate that the complexity of calculating the distance matrix is $O(n^2m)$, where n is the number of users, and m is the number of items (goods).

This problem can be partly mitigated by purchasing high-performance hardware. But if you approach it wisely, then it's better to introduce some corrections to the algorithm in the following way:

- Update distances not with every purchase but with batches (for example, once a day)
- Don't recalculate the distance matrix completely, but update it incrementally
- Choose some iterative and approximate algorithms (for example, **Alternating Least Squares (ALS)**)

Fulfill the following assumptions to make the algorithm more practical:

- The tastes of people don't change over time (or they do change, but they're the same for everyone)
- If people's tastes are the same, then they're the same in everything

For example, if two clients prefer the same films, then they also like the same book. This assumption is often the case when the recommended products are homogeneous (for example, films only). If this isn't the case, then a couple of clients may well have the same eating habits but their political views might be the opposite; here, the algorithm is less efficient.

The neighborhood of the user in the space of preferences (the user's neighbors), which we analyze to generate new recommendations, can be chosen in different ways. We can work with all users of the system; we can set a certain proximity threshold; we can choose several neighbors at random; or we can take the **k** most similar neighbors (this is the most popular approach). If we take too many neighbors, we get a higher chance of random noise, and vice versa. If we take too little, we get more accurate recommendations, but fewer goods can be recommended.

An interesting development in the collaborative approach is trust-based recommendations, which take into account not only the proximity of people according to their interests but also their *social* proximity and the degree of trust between them. If, for example, we see that on Facebook, a girl occasionally visits a page that has her friend's audio recordings, then she trusts her musical taste. Therefore, when making recommendations to the girl, you can add new songs from her friend's playlist.

Item-based collaborative filtering

The item-based approach is a natural alternative to the classic user-based approach described previously and almost repeats it, except for one thing—it applies to the transposed preference matrix, which looks for similar products instead of users.

For each client, a user-based collaborative filtering system searches a group of customers who are similar to this user in terms of previous purchases, and then the system averages their preferences. These average preferences serve as recommendations for the user. In the case of item-based collaborative filtering, the nearest neighbors are searched for on a variety of products (items) using the columns of a preference matrix, and the averaging occurs precisely according to them.

If some products are meaningfully similar to each other, then users' reactions to these products will be the same. Therefore, when we see that some products have a strong correlation between their estimates, this may indicate that these products are equivalent to each other.

The main advantage of the item-based approach over the user-based approach is lower computation complexity. When there are many users (almost always), the task of finding the nearest neighbor becomes poorly computable. For example, for 1 million users, you need to calculate and store ~500 billion distances. If the distance is encoded in 8 bytes, this results in 4 **terabytes (TB)** for the distance matrix alone. If we take an item-based approach, then the computational complexity decreases from $O(N^2n)$ to $O(n^2N)$, and the distance matrix has a dimension no longer than 1 million per 1 million but 100 by 100, as per the number of items (goods).

Estimating the proximity of products is much more accurate than assessing the proximity of users. This assumption is a direct consequence of the fact that there are usually many more users than items, and therefore the standard error in calculating the correlation of items is significantly less because we have more information to work from.

In the user-based version, the description of users usually has a very sparse distribution (there are many goods, but only a few evaluations). On the one hand, this helps to optimize the calculation—we multiply only those elements where an intersection exists. But, on the other hand, the list of items that a system can recommend to a user is minimal due to the limited number of user neighbors (users who have similar preferences). Also, user preferences may change over time, but the descriptions of the goods are much more stable.

The rest of the algorithm almost wholly repeats the user-based version: it uses the same cosine distance as the primary measure of proximity and has the same need for data normalization. Since the correlation of items is considered on a higher number of observations, it isn't as critical to recalculate it after each new assessment, and this can be done periodically in a batch mode.

Now, let's look at another approach to generalizing user interests based on matrix factorization methods.

Factorization algorithms

It would be nice to describe the interests of the user with more extensive features—not in the format of *they love movies X, Y, and Z*, but in the format of *they love romantic comedies*. Besides the fact that it increases the generalizability of the model, it also solves the problem of having a large data dimension—after all, the interests are described not by the items vector, but by a significantly smaller preference vector.

Such approaches are also called **spectral decomposition** or **high-frequency filtering** (since we remove the noise and leave the useful signal). There are many different types of matrix decomposition in algebra, and one of the most commonly used is called **singular value decomposition (SVD)**.

Initially, the SVD method was used to select pages that are similar in meaning but not in content. More recently, it has started being used in recommendations. The method is based on decomposing the original R rating matrix into a product of three matrices, $R = U * D * S$, where the sizes of the matrices are $(k, m) = (k, r) * (r, r) * (r, m)$ and r is the rank of the decomposition, which is the parameter characterizing the degree of detail decomposition.

Applying this decomposition to our matrix of preferences, we can get the following two matrices of factors (abbreviated descriptions):

- **U:** A compact description of user preferences
- **S:** A compact description of the characteristics of the product

When using this approach, we can't know which particular characteristics correspond to the factors in the reduced descriptions; for us, they're encoded with some numbers. Therefore, SVD is an uninterpreted model. It's sufficient to multiply the matrix of factors to obtain an approximation of the matrix of preferences. By doing this, we get a rating for all customer-product pairs.

A typical family of such algorithms is called **non-negative matrix factorization (NMF)**. As a rule, the calculation of such expansions is very computationally expensive. Therefore, in practice, they often resort to their approximate iterative variants. ALS is a popular iterative algorithm for decomposing a matrix of preferences into a product of two matrices: **user factors (U)** and **product factors (I)**. It works on the principle of minimizing the **root mean square error (RMSE)** on the affixed ratings. Optimization takes place alternately—first by user factors, then by product factors. Also, to avoid retraining, the regularization coefficients are added to the RMSE.

If we supplement the matrix of preferences with a new dimension containing information about the user or product, then we can work not with the matrix of preferences, but with the tensor. Thus, we use more available information and possibly get a more accurate model.

In this section, we considered different approaches to solving recommender systems' tasks. Now, we're going to discuss methods for estimating the similarity of user preferences.

Similarity or preferences correlation

We can consider the similarity or correlation of two user preferences in different ways, but in general, we need to compare two vectors. Let's look at some of the most popular vector comparison measures.

Pearson's correlation coefficient

This measure is a classic coefficient that can be applied when comparing vectors. Its primary disadvantage is that when the intersection is estimated as low, then the correlation can be high by accident. To combat accidental high correlation, you can multiply by a factor of 50/min (50, rating intersection) or any other damping factor, the effect of which decreases with an increasing number of estimates. An example is shown here:

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2(y_i - \bar{y})^2}}$$

Spearman's correlation

The main difference compared to Pearson's correlation is the rank factor—that is, it doesn't work with absolute values of ratings, but with their sequence numbers. In general, the result is very close to Pearson's correlation. An example is shown here:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

Cosine distance

Cosine distance is another classic measuring factor. If you look closely, the cosine of the angle between the standardized vectors is Pearson's correlation, the same formula. This distance uses cosine properties: if the two vectors are co-directed (that is, the angle between them is 0), then the cosine of the angle between them is 1. Conversely, the cosine of the angle between perpendicular vectors is 0. An example is shown here:

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|}$$

With that, we've discussed methods we can use to estimate the similarity of user preferences. The next important issue we'll discuss is preparing data so that it can be used in recommender system algorithms.

Data scaling and standardization

All users evaluate (rate) items differently. If someone puts 5s in a row, instead of waiting for 4s from someone else, it's better to normalize the data before calculating it—that is, convert the data into a single scale so that the algorithm can compare the results correctly. After, the predicted estimate needs to be converted into the original scale via inverse transformation (and, if necessary, rounded to the nearest whole number).

There are several ways to normalize data:

- **Centering (mean-centering):** From the user's ratings, subtract their average rating. This type of normalization is only relevant for non-binary matrices.
- **Standardization (z-score):** In addition to centering, this divides the user's rating by the standard deviation of the user. But in this case, after the inverse transformation, the rating can go beyond the scale (for example, six on a five-point scale), but such situations are quite rare and can be solved by rounding to the nearest acceptable estimate.
- **Double standardization:** The first time data is normalized by user ratings; the second time, by item ratings.

The details of these normalization techniques were provided in *Chapter 2, Data Processing*. The following section will describe a problem with recommender systems known as the **cold start problem**, which appears in the early stages of system work when the system doesn't have enough data to make predictions.

Cold start problem

A cold start is a typical situation when a sufficient amount of data hasn't been accumulated for the correct operation of the recommender system yet (for example, when a product is new or is just rarely bought). If the ratings of only three users estimate the average rating, such an assessment isn't reliable, and users understand this. In such situations, ratings are often artificially adjusted.

The first way to do this is to show not the average value, but the smoothed average (damped mean). With a small number of ratings, the displayed rating leans more toward a specific safe *average* indicator, and as soon as a sufficient number of new ratings are typed, the *averaging* adjustment stops operating.

Another approach is to calculate confidence intervals for each rating. Mathematically, the more estimates we have, the smaller the variation of the average will be and, therefore, the more confidence we have in its accuracy.

For example, we can display the lower limit of the interval (low **confidence interval (CI)** bound) as a rating. At the same time, it's clear that such a system is quite conservative, with a tendency to underestimate ratings for new items.

Since the estimates are limited to a specific scale (for example, from 0 to 1), the usual methods for calculating the confidence interval are poorly applicable here due to the distribution tails that go to infinity, and the symmetry of the interval itself. There's a more accurate way to calculate it—the Wilson CI.

The cold start problem is also relevant for non-personalized recommendations. The general approach here is to replace what currently can't be calculated by different heuristics—for example, replace it with an average rating, use a simpler algorithm, or not use the product at all until the data has been collected.

Another issue that should be considered when we develop a recommender system is the relevance of recommendations, which considers factors other than the user's interests—for example, it can be the freshness of a publication or a user's rating.

Relevance of recommendations

In some cases, it's also essential to consider the *freshness* of the recommendation. This consideration is especially important for articles or posts on forums. Fresh entries should often get to the top. The correction factors (damping factors) are usually used to make such updates. The following formulas are used for calculating the rating of articles on media sites.

Here's an example of a rating calculation in the *Hacker news* magazine:

$$\text{rank} = \frac{(U - D - 1)^{0.8} * P}{T^{1.8}}$$

Here, U denotes upvotes, D denotes downvotes, P denotes penalty (additional adjustment for the implementation of other business rules), and T denotes recording time.

The following equation shows a *Reddit* rating calculation:

$$\text{rank} = \log_{10} (\max(1, U - D)) - \frac{|U - D| T}{\text{const}}$$

Here, U denotes the number of upvotes, D denotes the number of downvotes, and T denotes the recording time. The first term evaluates the *quality of the record*, and the second corrects for the time.

There's no universal formula, and each service invents the formula that best solves its problem; it can only be tested empirically.

The following section will discuss the existing approaches to testing recommender systems. This isn't a straightforward task because it's usually hard to estimate the quality of a recommendation without having exact target values in a training dataset.

Assessing system quality

Testing a recommender system is a complicated process that always poses many questions, mainly due to the ambiguity of the concept of *quality*.

In general, in machine learning problems, there are two main approaches to testing:

- Offline model testing on historical data using retro tests
- Testing the model using A/B testing (we run several options and see which one gives the best result)

Both of these approaches are actively used in developing recommender systems. The main limitation that we have to face is that we can only evaluate the accuracy of the forecast on those products that the user has already evaluated or rated. The standard approach is to use cross-validation alongside the **leave-one-out** and **leave-p-out** methods. Repeating the test and averaging the results provides a more stable assessment of quality.

The *leave-one-out* approach uses the model that's been trained on all items except one and is evaluated by the user. This excluded item is used for model testing. This procedure is done for all n items, and an average is calculated among the obtained n quality estimates.

The *leave-p-out* approach is the same, but at each step, p points are excluded.

We can divide all quality metrics into the following three categories:

- **Prediction accuracy:** Estimates the accuracy of the predicted rating
- **Decision support:** Evaluates the relevance of the recommendations
- **Rank accuracy metrics:** Evaluates the quality of the ranking of recommendations issued

Unfortunately, there's no single recommended metric for all occasions, and everyone who's involved in testing a recommender system selects it to fit their goals.

In the following section, we'll formalize the collaborative filtering method and show the math behind it.

Understanding the collaborative filtering method

In this section, we'll formalize the recommender system problem. We have a set of users, $u \in U$, a set of items, $i \in I$ (movies, tracks, products, and so on), and a set of estimates, $(r_{ui}, u, i, \dots) \in D$. Each estimate is given by a user u , an object i , its result r_{ui} , and, possibly, some other characteristics.

We're required to predict preference as follows:

$$\hat{r}_{ui} = \text{Predict}(u, i, \dots) \approx r_{ui}$$

We're required to predict personal recommendations as follows:

$$u \mapsto (i_1, \dots, i_K) = \text{Recommend}_K(u, \dots)$$

We're required to predict similar objects as follows:

$$u \mapsto (i_1, \dots, i_M) = \text{Similar}_M(i)$$

Remember that the main idea behind collaborative filtering is that similar users usually like similar objects. Let's start with the simplest method:

1. Select some conditional measures of similarity of users according to their history of $\text{sim}(u, v)$ ratings.
2. Unite users into groups (clusters) so that similar users will end up in the same cluster: $u \mapsto F(u)$
3. Predict the item's user rating as the cluster's average rating for this object:

$$\hat{r}_{ui} = \frac{1}{|F(u)|} \sum_{v \in F(u)} r_v i$$

This algorithm has several problems:

- There's nothing to recommend to new or atypical users. For such users, there's no suitable cluster with similar users.
- It ignores the specificity of each user. In a sense, we divide all users into classes (templates).
- If no one in the cluster has rated the item, the prediction won't work.

We can improve this method and replace hard clustering with the following formula:

$$\hat{r}_{ui} = \bar{r}_u + \frac{\sum_{v \in U_i} \text{sim}(u, v)(r_{u,i} - \bar{r}_v)}{\sum_{v \in U_i} \text{sim}(u, v)}$$

For an item-based version, the formula will be symmetrical, as follows:

$$\hat{r}_{ui} = \bar{r}_i + \frac{\sum_{j \in I_u} \text{sim}(i, j)(r_{u,j} - \bar{r}_j)}{\sum_{j \in I_u} \text{sim}(i, j)}$$

These approaches have the following disadvantages:

- Cold start problem
- Bad predictions for new and atypical users or items
- Trivial recommendations
- Resource intensity calculations

To overcome these problems, you can use SVD. The preference (ratings) matrix can be decomposed into the product of three matrices, $A = U * \Sigma * V^T$. Let's denote the product of the first two matrices for one matrix, $R \approx U * V$, where R is the matrix of preferences, U is the matrix of parameters of users, and V is the matrix of parameters of items.

To predict the user rating, U , for an item, I , we take a vector, p_u (parameter set), for a given user and a vector for a given item, q_i . Their scalar product is the prediction we need: $\hat{r}_{ui} = \langle p_u, q_i \rangle$. Using this approach, we can identify the hidden features of items and user interests by user history. For example, it may happen that at the first coordinate of the vector, each user has a number indicating whether the user is more likely to be a boy or a girl, and the second coordinate is a number reflecting the approximate age of the user. In the item, the first coordinate shows whether it's more interesting to boys or girls, and the second one shows the age group of users this item appeals to.

However, there are also several problems. The first one is the preferences matrix, R , which isn't entirely known to us, so we can't merely take its SVD decomposition. Secondly, the SVD decomposition isn't the only one we have, so even if we find at least some decomposition, it's unlikely that it's optimal for our task.

Here, we need machine learning. We can't find the SVD decomposition of the matrix since we don't know the matrix itself. However, we can take advantage of this idea and come up with a prediction model that works like SVD. Our model depends on many parameters—vectors of users and items. For the given parameters, to predict the estimate, we must take the user vector, the vector of the item, and get their scalar product, $\hat{r}_{ui}(\theta) = p_u^T q_i$. However, since we don't know vectors, they still need to be obtained. The idea is that we have user ratings with which we can find optimal parameters so that our model can predict these estimates as accurately as possible using the following equation: $E_{ui}(\hat{r}_{ui}(\theta) - r_{ui})^2 \rightarrow \min(\theta)$. We want to find such parameters' θ values so that the square error is as small as possible. We also want to make fewer mistakes in the future, but we don't know what estimates we need. Accordingly, we can't optimize parameters' θ values. We already know the ratings given by users, so we can try to choose parameters based on the estimates we already have to minimize the error. We can also add another term, the *regularizer*, as shown here:

$$\sum_{(u,i) \in D} (\hat{r}_{ui}(\theta) - r_{ui})^2 + \lambda \sum_{\theta \in \Theta} \theta^2 \rightarrow \min(\theta)$$

Regularization is needed to combat overfitting. To find the optimal parameters, you need to optimize the following function:

$$J(\Theta) = \sum_{(u,i) \in D} (p_u^T q_i - r_{ui})^2 + \lambda \left(\sum_u ||p_u||^2 + \sum_i ||q_i||^2 \right)$$

There are many parameters: for each user and item, we have the vector that we want to optimize. The most well-known method for optimizing functions is **gradient descent (GD)**. Suppose we have a function of many variables, and we want to optimize it. We take an initial value, and then we look at where we can move to minimize this value. The GD method is an iterative algorithm—it takes the parameters of a certain point repeatedly, looks at the gradient, and steps against its direction, as shown here:

$$\Theta_{t+1} = \Theta_t - \eta \nabla J(\Theta)$$

There are various problems with this method: it works very slowly and it finds local, rather than global, minima. The second problem isn't so bad for us because in our case, the value of the function in local minima is close to the global optimum.

However, the GD method isn't always necessary. For example, if we need to calculate the minimum for a parabola, there's no need to act by this method as we know precisely where its minimum is. It turns out that the functionality that we're trying to optimize—the sum of the squares of errors plus the sum of the squares of all the parameters—is also a quadratic function, which is very similar to a parabola. For each specific parameter, if we fix all the others, it's just a parabola. For those, we can accurately determine at least one coordinate. The ALS method is based on this assumption. We alternate between accurately finding minima in one coordinate or another, as shown here:

$$\begin{aligned}\hat{p}_u(\Theta) &= \arg \min_{p_u} J(\Theta) = (Q_u^T Q_u + \lambda I)^{-1} Q_u^T r_u \\ \hat{q}_i(\Theta) &= \arg \min_{q_i} J(\Theta) = (P_i^T P_i + \lambda I)^{-1} P_i^T r_i\end{aligned}$$

We fix all the parameters of the items, optimize the parameters of users, fix the parameters of users, and then optimize the parameters of items. We act iteratively, as shown here:

$$\begin{aligned}\forall u \in U p_u^{2t+1} &= \hat{p}_u(\Theta_{2t}) \\ \forall i \in I q_i^{2t+2} &= \hat{q}_i(\Theta_{2t+1})\end{aligned}$$

This method works reasonably quickly, and you can parallelize each step. However, there's still a problem with implicit data because we have neither full user data nor full item data. So, we can penalize the items that don't have ratings in the update rule. By doing so, we depend only on the items that have ratings from the users and don't make any assumptions about the items that aren't rated. So, let's define a weight matrix, w_{ui} , as follows:

$$w_{ui} = \begin{cases} 0 & \text{if } q_{ui} = 0 \\ 1 & \text{else} \end{cases}$$

The cost functions that we're trying to minimize look like this:

$$\begin{aligned}J(x_u) &= (q_u - x_u Y) W_u (q_u - x_u Y)^T + \lambda x_u x_u^T \\ J(y_i) &= (q_i - X y_i) W_i (q_i - X y_i)^T + \lambda y_i y_i^T\end{aligned}$$

Note that we need regularization terms to avoid overfitting the data. We can use the following solutions for factor vectors:

$$\begin{aligned} x_u &= (Y W_u Y^T + \lambda I)^{-1} Y W_u q_u \\ y_i &= (X^T W_i X + \lambda I)^{-1} X^T W_i q_i \end{aligned}$$

Here, $W_u \in \mathbb{R}^{nn}$ and $W_i \in \mathbb{R}^{mm}$ are diagonal matrices.

Another approach for dealing with implicit data is to introduce confidence levels. Let's define a set of binary observation variables:

$$p_{u,i} = \begin{cases} 1 & \text{if } q_{u,i} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Now, we can define confidence levels for each $p_{u,i}$ value. When $q_{u,i} = 0$, we have low confidence. This can be because the user has never been exposed to that item or it may be unavailable at the time. For example, it could be explained by the user buying a gift for someone else. Hence, we would have *low confidence*. When $q_{u,i}$ is larger, we should have much more confidence. For example, we can define confidence as follows:

$$c_{u,i} = 1 + \alpha q_{u,i}$$

Here, α is a hyperparameter that should be tuned for a given dataset. The updated optimization function is as follows:

$$\min_{x,y} \sum_{r_{u,i} \text{ is known}} c_{u,i} (p_{u,i} - x_u^\top y_i)^2 + \lambda (\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2)$$

Here, C^i is a diagonal matrix with $C_{u,u}^i = c_{u,i}$ values. The following solutions are for user and item ratings:

$$\begin{aligned} y_u &= (X^\top C^i X + \lambda I)^{-1} X^\top C^i p_i \\ x_u &= (Y^\top C^u Y + \lambda I)^{-1} Y^\top C^u p_u \end{aligned}$$

However, it's an expensive computational problem to calculate the $X^\top C^i X$ expression. However, it can be optimized in the following way:

$$X^\top C^i X = X^\top X + X^\top (C^i - I) X$$

This means that $X^\top X$ can be precomputed at each of the steps, and $(C^i - I)$ only contains the non-zero entries where $r_{u,i}$ was non-zero. Now that we've learned about collaborative filtering in detail, let's understand it further practically by considering a few examples of how to implement a collaborative filtering recommender system.

In the following sections, we'll learn how to use different C++ libraries for developing recommender systems.

Examples of item-based collaborative filtering with C++

Let's look at how we can implement a collaborative filtering recommender system. We'll be using the MovieLens dataset provided by GroupLens from the research lab in the Department of Computer Science and Engineering at the University of Minnesota: <https://grouplens.org/datasets/movielens/>. They've provided a full dataset containing 20 million movie ratings and a smaller one for education that contains 100,000 ratings. We recommend starting with the smaller one because it allows us to see results earlier and detect implementation errors faster.

This dataset consists of several files, but we're only interested in two of them: `ratings.csv` and `movies.csv`. The rating file contains lines with the following format: the user ID, the movie ID, the rating, and the timestamp. In this dataset, users made ratings on a 5-star scale, with half-star increments (0.5 stars to 5.0 stars). The movie's file contains lines with the following format: the movie ID, the title, and the genre. The movie ID is the same in both files so that we can see which movies users are rating.

Using the Eigen library

First, let's learn how to implement a collaborative filtering recommender system based on matrix factorization with ALS and with a pure linear algebra library as a backend. In the following sample, we're using the Eigen library. The steps to implement a collaborative filtering recommender system are as follows:

1. First, we must make base type definitions, as follows:

```
using DataType = float;
// using Eigen::ColMajor is Eigen restriction - todense method
// always returns
// matrices in ColMajor order
using Matrix = Eigen::Matrix<DataType,
                           Eigen::Dynamic,
                           Eigen::Dynamic,
                           Eigen::ColMajor>;
using SparseMatrix =
Eigen::SparseMatrix<DataType, Eigen::ColMajor>;
using DiagonalMatrix =
Eigen::DiagonalMatrix<DataType,
                           Eigen::Dynamic,
                           Eigen::Dynamic>;
```

2. These definitions allow us to write less source code for matrices' types and to quickly change floating-point precision. Next, we must define and initialize the ratings (preferences) matrix, list of movie titles, and binary rating flags matrix, as follows:

```
SparseMatrix ratings_matrix; // user-item ratings
SparseMatrix p; // binary variables
std::vector<std::string> movie_titles;
```

We have a particular helper function, `LoadMovies`, which loads files into the map container, as shown in the following code snippet:

```
auto movies_file = root_path / "movies.csv";
auto movies = LoadMovies(movies_file);

auto ratings_file = root_path / "ratings.csv";
auto ratings = LoadRatings(ratings_file);
```

3. Once the data has been loaded, we can initialize matrix objects so that they're the right size:

```
ratings_matrix.resize(
    static_cast<Eigen::Index>(ratings.size()),
    static_cast<Eigen::Index>(movies.size()));
ratings_matrix.setZero();
p.resize(ratings_matrix.rows(), ratings_matrix.cols());
p.setZero();
movie_titles.resize(movies.size());
```

However, because we've loaded data into the map, we need to move the required rating values to the matrix object.

4. Now, we must initialize the movie titles list, convert user IDs into our zero-based sequential order, and initialize the binary rating matrix (this is used in the algorithm to deal with implicit data), as follows:

```
Eigen::Index user_idx = 0;
for (auto& r : ratings) {
    for (auto& m : r.second) {
        auto mi = movies.find(m.first);
        Eigen::Index movie_idx =
            std::distance(movies.begin(), mi);
        movie_titles[static_cast<size_t>(movie_idx)] =
            mi->second;
        ratings_matrix.insert(user_idx, movie_idx) =
            static_cast<DataType>(m.second);
        p.insert(user_idx, movie_idx) = 1.0;
    }
    ++user_idx;
}
ratings_matrix.makeCompressed();
```

5. Once the rating matrix has been initialized, we must define and initialize our training variables:

```
auto m = ratings_matrix.rows();
auto n = ratings_matrix.cols();

Eigen::Index n_factors = 100;
auto y = InitializeMatrix(n, n_factors);
auto x = InitializeMatrix(m, n_factors);
```

In the preceding code snippet, the **y** matrix corresponds to user preferences, while the **x** matrix corresponds to the item parameters. We've also defined the number of factors we'll be interested in after decomposition. These matrices are initialized with random values and normalized. Such an approach is used to speed up algorithm convergence. This can be seen in the following code snippet:

```
Matrix InitializeMatrix(Eigen::Index rows,
                        Eigen::Index cols) {
    Matrix mat = Matrix::Random(rows, cols).array().abs();
    auto row_sums = mat.rowwise().sum();
    mat.array().colwise() /= row_sums.array();
    return mat;
}
```

6. Then, we must define and initialize the regularization matrix and identity matrices, which are constant during all learning cycles:

```
DataType reg_lambda = 0.1f;
SparseMatrix reg = (reg_lambda * Matrix::Identity(
    n_factors, n_factors)).sparseView();
// Define diagonal identity terms
SparseMatrix user_diag = -1 * Matrix::Identity(
    n, n).sparseView();
SparseMatrix item_diag = -1 * Matrix::Identity(
    m, m).sparseView();
```

7. Additionally, because we're implementing an algorithm version that can handle implicit data, we need to convert our rating matrix into another format to decrease computational complexity. Our version of the algorithm needs user ratings in the form of $c_{u,i} = 1 + \alpha q_{u,i}$ and diagonal matrices for every user and item so that we can make two containers with corresponding matrix objects. The code for this can be seen in the following block:

```
std::vector<DiagonalMatrix> user_weights(
    static_cast<size_t>(m));
std::vector<DiagonalMatrix> item_weights(
```

```

        static_cast<size_t>(n));
{
    Matrix weights(ratings_matrix);
    weights.array() *= alpha;
    weights.array() += 1;
    for (Eigen::Index i = 0; i < m; ++i) {
        user_weights[static_cast<size_t>(i)] =
            weights.row(i).asDiagonal();
    }
    for (Eigen::Index i = 0; i < n; ++i) {
        item_weights[static_cast<size_t>(i)] =
            weights.col(i).asDiagonal();
    }
}
}

```

Now, we're ready to implement the main learning loop. As discussed previously, the ALS algorithm can be easily parallelized, so we use the OpenMP compiler extension to calculate user and item parameters in parallel.

- Let's define the main learning cycle, which runs for a specified number of iterations:

```

size_t n_iterations = 5;
for (size_t k = 0; k < n_iterations; ++k) {
    auto yt = y.transpose();
    auto yty = yt * y;
    ...
    // update item parameters
    ... auto xt = x.transpose();
    auto xtx = xt * x;
    ...
    // update users preferences
    ... auto w_mse = CalculateWeightedMse(
        x, y, p, ratings_matrix, alpha);
}

```

- The following code shows how to update item parameters:

```

#pragma omp parallel
{
    Matrix diff;
    Matrix ytcuy;
    Matrix a, b, update_y;
    #pragma omp for private(diff, ytcuy, a, b, update_y)
    for (size_t i = 0; i < static_cast<size_t>(m); ++i) {

```

```
diff = user_diag;
diff += user_weights[i];
ytcuy = yty + yt * diff * y;
auto p_val =
    p.row(static_cast<Eigen::Index>(i)).transpose();
a = ytcuy + reg;
b = yt * user_weights[i] * p_val;
update_y = a.colPivHouseholderQr().solve(b);
x.row(static_cast<Eigen::Index>(i)) =
    update_y.transpose();
}
}
```

10. The following code shows how to update users' preferences:

```
#pragma omp parallel
{
    Matrix diff;
    Matrix xtcux;
    Matrix a, b, update_x;
#pragma omp for private(diff, xtcux, a, b, update_x)
    for (size_t i = 0; i < static_cast<size_t>(n); ++i) {
        diff = item_diag;
        diff += item_weights[i];
        xtcux = xtx + xt * diff * x;
        auto p_val = p.col(static_cast<Eigen::Index>(i));
        a = xtcux + reg;
        b = xt * item_weights[i] * p_val;
        update_x = a.colPivHouseholderQr().solve(b);
        y.row(static_cast<Eigen::Index>(i)) =
            update_x.transpose();
    }
}
```

Here, we have two parts of the loop body that are pretty much the same. First, we updated item parameters with frizzed user options, and then we updated user preferences with frizzed item parameters. Notice that all matrix objects were moved outside of the internal loop body to reduce memory allocations and significantly improve program performance. Also, notice that we parallelized the user and item parameters' calculations separately because one of them should always be frizzed when the other is being calculated. To calculate exact values for user preferences and item parameters, we must use the following formula:

$$\begin{aligned}y_u &= (X^\top X + X^\top (C^i - I)X + \lambda I)^{-1} X^\top C^i p_i \\x_u &= (Y^\top Y + Y^\top (C^i - I)Y + \lambda I)^{-1} Y^\top C^u p_u\end{aligned}$$

$X^\top X$ and $Y^\top Y$ are precomputed at each step. Also, notice that these formulas are expressed in the form of the linear equation system, $X = AB$. We use the `colPivHouseholderQr` function from the `Eigen` library to solve it and get exact values for the user and item parameters. This linear equation system can also be solved with other methods. The `colPivHouseholderQr` function was chosen because it shows a better ratio between computational speed and accuracy in the `Eigen` library implementation.

11. To estimate the progress of the learning process of our system, we can calculate the **mean squared error (MSE)** between the original rating matrix and a predicted one. To calculate the predicted rating matrix, we must define another function:

```
Matrix RatingsPredictions(const Matrix& x, const Matrix& y) {
    return x * y.transpose();
}
```

12. To calculate the MSE, we can use the $c_{u,i}(p_{u,i} - x_u^\top y_i)^2$ expression from our optimization function:

```
DataType CalculateWeightedMse(const Matrix& x,
                               const Matrix& y,
                               const SparseMatrix& p,
                               const SparseMatrix& ratings_matrix,
                               DataType alpha) {

    Matrix c(ratings_matrix);
    c.array() *= alpha;
    c.array() += 1.0;
    Matrix diff(p - RatingsPredictions(x, y));
    diff = diff.array().pow(2.f);
    Matrix weighted_diff = c.array() * diff.array();
    return weighted_diff.array().mean();
}
```

Please note that we have to use weights and binary ratings to get a meaningful value for the error because a similar approach was used during the learning process. Direct error calculation gives the wrong result because the predicted matrix has non-zero predictions, whereas the original rating matrix has zeros. It's essential to understand that this algorithm doesn't learn the original scale of ratings (from 0 to 5); instead, it learns prediction values in a range from 0 to 1. It follows on from the function we optimize, as shown here:

$$\min_{x,y} \sum_{\substack{r_{u,i} \text{ is known}}} c_{u,i} (p_{u,i} - x_u^\top y_i)^2 + \lambda (\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2)$$

13. We can use the previously defined movies list to show movie recommendations. The following function shows user preferences and system recommendations. To identify what a user likes, we'll show movie titles that the user has rated with a rating value of more than 3. We'll also show movies that the system rates as equal to or higher than a 0.8 rating coefficient to identify which movie the system recommends to the user by running the following code:

```
void PrintRecommendations(
    const Matrix& ratings_matrix,
    const Matrix& ratings_matrix_pred,
    const std::vector<std::string>& movie_titles) {
    // collect recommendations
    auto n = ratings_matrix.cols();
    std::vector<std::string> liked;
    std::vector<std::string> recommended;
    for (Eigen::Index u = 0; u < 5; ++u) {
        for (Eigen::Index i = 0; i < n; ++i) {
            DataType orig_value = ratings_matrix(u, i);
            if (orig_value >= 3.f) {
                liked.push_back(
                    movie_titles[static_cast<size_t>(i)]);
            }
            DataType pred_value = ratings_matrix_pred(u, i);
            if (pred_value >= 0.8f && orig_value < 1.f) {
                recommended.push_back(
                    movie_titles[static_cast<size_t>(i)]);
            }
        }
    }
    // print recommendations
    std::cout << "\nUser " << u << " liked :";
    for (auto& l : liked) {
        std::cout << l << " ";
    }
    std::cout << "\nUser " << u << " recommended :";
```

```

        for (auto& r : recommended) {
            std::cout << r << " ; ";
        }
        std::cout << std::endl;
        liked.clear();
        recommended.clear();
    }
}

```

This function can be used as follows:

```

PrintRecommendations(ratings_matrix,
                      RatingsPredictions(x, y),
                      movie_titles);

```

Using the mlpack library

The `mlpack` library is a general-purpose machine learning library that provides a lot of different algorithms and command-line tools to process the data and learn these algorithms without explicit programming. As a basis, this library uses the `Armadillo` linear algebra library for math calculations. Other libraries we've used in previous chapters don't have collaborative filtering algorithm implementations.

To load the MovieLens dataset, use the same loading helper function that you did in the previous section. Once the data has been loaded, convert it into a format suitable for an object of the `mlpack::cf::CFType` type. This type implements a collaborative filtering algorithm and can be configured with different types of matrix factorization approaches. An object of this type can use dense as well as sparse rating matrices. In the case of a dense matrix, it should have three rows. The first row corresponds to users, the second row corresponds to items, and the third row corresponds to the rating. This structure is called a **coordinate list format**. In the case of the sparse matrix, it should be a regular (user, item) table, as in the previous example. So, let's define the sparse matrix for ratings. It should have the `arma::SpMat<DataType>` type from the `Armadillo` library, as illustrated in the following code block:

```

arma::SpMat<DataType> ratings_matrix(ratings.size(),
                                       movies.size());
std::vector<std::string> movie_titles;
{
    // fill matrix with data
    movie_titles.resize(movies.size());
    size_t user_idx = 0;
    for (auto& r : ratings) {
        for (auto& m : r.second) {
            auto mi = movies.find(m.first);

```

```
    auto movie_idx = std::distance(movies.begin(), mi);
    movie_titles[static_cast<size_t>(movie_idx)] =
        mi->second;
    ratings_matrix(user_idx, movie_idx) =
        static_cast<DataType>(m.second);
}
++user_idx;
}
}
```

Now, we can initialize the `mlpack::cf::CFType` class object. It takes the next parameters in the constructor: the rating matrix, the matrix decomposition policy, the number of neighbors, the number of target factors, the number of iterations, and the minimum value of learning error, after which the algorithm can stop.

For this object, only perform the nearest neighbor search on the **H** matrix. This means you avoid calculating the full rating matrix, using the observation that if the rating matrix is $\mathbf{X} = \mathbf{W} \mathbf{H}$, then the following applies:

```
distance(X.col(i), X.col(j)) = distance(W.H.col(i), W.H.col(j))
```

This expression can be seen as the nearest neighbor search on the **H** matrix with the Mahalanobis distance, as illustrated in the following code block:

```
// factorization rank
size_t n_factors = 100;
size_t neighborhood = 50;

mlpack::NMFPolicy decomposition_policy;

// stopping criterions
size_t max_iterations = 20;
double min_residue = 1e-3;

mlpack::CFType cf(ratings_matrix,
                  decomposition_policy,
                  neighborhood,
                  n_factors,
                  max_iterations,
                  min_residue);
```

Notice that as a decomposition policy, the object of the `mlpack::NMFPolicy` type was used. This shows how to implement the non-negative matrix factorization algorithm with the ALS approach. There are several decomposition algorithms in the `mlpack` library. For example, batch SVD decomposition is implemented in the `mlpack::BatchSVDPolicy` type. The constructor of this object also does the complete training, so after its call has finished, we can use this object to get recommendations. Recommendations can be retrieved with the `GetRecommendations` method. This method gets the number of recommendations you want to get, the output matrix for recommendations, and the list of user IDs for users you want to get recommendations from, as shown in the following code block:

```
arma::Mat<size_t> recommendations;
// Get 5 recommendations for specified users.
arma::Col<size_t> users;
users << 1 << 2 << 3;
cf.GetRecommendations(5, recommendations, users);
for (size_t u = 0; u < recommendations.n_cols; ++u) {
    std::cout << "User " << users(u) << " recommendations are: ";

    for (size_t i = 0; i < recommendations.n_rows; ++i) {
        std::cout << movie_titles[recommendations(i, u)] << ";";
    }
    std::cout << std::endl;
}
```

Notice that the `GetRecommendations` method returns the item IDs as its output. So, we can see that using this library for implementing a recommender system is much easier than writing it from scratch. Also, there are many more configuration options in the `mlpack` library for building such systems—for example, we can configure the neighbor detection policy and which distance measure to use. These configurations can significantly improve the quality of the system you build because you can make them according to your particular task.

Summary

In this chapter, we discussed what recommender systems are and the types that exist today. We studied two main approaches to building recommender systems: content-based recommendations and collaborative filtering. We identified two types of collaborative filtering: user-based and item-based. Then, we looked at how to implement these approaches, as well as their pros and cons. We found out that an important issue we must rectify when implementing recommender systems is the amount of data and the associated large computational complexity of algorithms. We considered approaches to overcome computational complexity problems, such as partial data updates and approximate iterative algorithms such as ALS. We found out how matrix factorization can help to solve the problem with incomplete data, improve the generalizability of the model, and speed up the calculations. We also implemented a system of collaborative filtering based on the linear algebra library and used the `mlpack` general-purpose machine learning library.

It makes sense to look at new methods that can be applied to recommender system tasks, such as autoencoders, variational autoencoders, or deep collaborative approaches. In recent research papers, these approaches show more impressive results than classical methods such as ALS. All these new methods are non-linear models, so they can potentially beat the limited modeling capacity of linear factor models.

In the next chapter, we'll discuss ensemble learning techniques. The main idea of these techniques is to combine either different types of machine learning algorithms or use a set of the same kind of algorithms to obtain better predictive performance. Combining several algorithms into one ensemble allows us to get the best characteristics of each so that we can cover the disadvantages in a single algorithm.

Further reading

- *Collaborative Filtering for Implicit Feedback Datasets*: <http://yifanhu.net/PUB/cf.pdf>
- *ALS Implicit Collaborative Filtering*: <https://medium.com/radon-dev/als-implicit-collaborative-filtering-5ed653ba39fe>
- *Collaborative Filtering*: <https://datasciencemakesimpler.wordpress.com/tag/alternating-least-squares/>
- The `mlpack` library's official site: <https://www.mlpack.org/>
- The `Armadillo` library's official site: <http://arma.sourceforge.net/>
- *Variational Autoencoders for Collaborative Filtering*, by Dawen Liang, Rahul G. Krishnan, Matthew D. Hoffman, and Tony Jebara: <https://arxiv.org/abs/1802.05814>
- *Deep Learning-Based Recommender System: A Survey and New Perspectives*, by Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay: <https://arxiv.org/abs/1707.07435>
- *Training Deep AutoEncoders for Collaborative Filtering*, by Oleksii Kuchaiev, and Boris Ginsburg: <https://arxiv.org/abs/1708.01715>

9

Ensemble Learning

Anyone who works with data analysis and machine learning will understand that no method is ideal or universal. This is why there are so many methods. Researchers and enthusiasts have been searching for years for a compromise between the accuracy, simplicity, and interpretability of various models. Moreover, how can we increase the accuracy of the model, preferably without changing its essence? One way to improve the accuracy of models is to create and train model **ensembles**—that is, sets of models used to solve the same problem. The ensemble training methodology is the training of a final set of simple classifiers, with a subsequent merging of the results of their predictions into a single forecast of the aggregated algorithm.

This chapter describes what ensemble learning is, what types of ensembles exist, and how they can help to obtain better predictive performance. In this chapter, we will also implement examples of these approaches with different C++ libraries.

The following topics will be covered in this chapter:

- An overview of ensemble learning
- Learning about decision trees and random forests
- Examples of using C++ libraries for creating ensembles

Technical requirements

The technologies and installations required in the chapter are as follows:

- The `Dlib` library
- The `mlpack` library
- A modern C++ compiler with C++20 support
- A CMake build system version ≥ 3.22

The code files for this chapter can be found at the following GitHub repo: <https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition/tree/main/Chapter09>

An overview of ensemble learning

The training of an ensemble of models is understood to be the procedure of training a final set of elementary algorithms whose results are then combined to form the forecast of an aggregated classifier. The model ensemble's purpose is to improve the accuracy of the prediction of the aggregated classifier, particularly when compared with the accuracy of every single elementary classifier. It is intuitively clear that combining simple classifiers can give a more accurate result than each simple classifier separately. Despite that, simple classifiers can be sufficiently accurate on particular datasets, but at the same time, they can make mistakes on different datasets.

An example of ensembles is **Condorcet's jury theorem** (1784). A jury must come to a correct or incorrect consensus, and each juror has an independent opinion. If the probability of the correct decision of each juror is more than 0.5, then the probability of a correct decision from the jury as a whole (tending toward 1) increases with the size of the jury. If the probability of making the correct decision is less than 0.5 for each juror, then the probability of making the right decision monotonically decreases (tending toward zero) as the jury size increases.

The theorem is as follows:

- N : The number of jury members
- p : The probability of the jury member making the right decision
- μ : The probability of the entire jury making the correct decision
- m : The minimum majority of jury members:

$$m = \text{floor}(N/2) + 1$$

- C_N^i : The number of combinations of N by i :

$$\mu = \sum_{i=m}^N C_N^i p^i (1-p)^{N-i}$$

If $p > 0.5$ then $\mu > p$

If $N \rightarrow \infty$ then $\mu \rightarrow 1$

Therefore, based on general reasoning, three reasons why ensembles of classifiers can be successful can be distinguished, as follows:

- **Statistical:** The classification algorithm can be viewed as a search procedure in the space of the **H hypothesis**, concerned with the distribution of data in order to find the best hypothesis. By learning from the final dataset, the algorithm can find many different hypotheses that describe the training sample equally well. By building an ensemble of models, we *average out* the error of each hypothesis and reduce the influence of instabilities and randomness in the formation of a new hypothesis.
- **Computational:** Most learning algorithms use methods for finding the extremum of a specific objective function. For example, neural networks use **gradient descent (GD)** methods to minimize prediction errors. Decision trees use greedy algorithms that minimize data entropy. These optimization algorithms can become stuck at a local extremum point, which is a problem because their goal is to find a global optimum. The ensembles of models combining the results of the prediction of simple classifiers, trained on different subsets of the source data, have a higher chance of finding a global optimum since they start a search for the optimum from different points in the initial set of hypotheses.
- **Representative:** A combined hypothesis may not be in the set of possible hypotheses for simple classifiers. Therefore, by building a combined hypothesis, we expand the set of possible hypotheses.

Condorcet's jury theorem and the reasons provided previously are not entirely suitable for real, practical situations because the algorithms are not independent (they solve one problem, they learn on one target vector, and can only use one model, or a small number of models).

Therefore, the majority of techniques in applied ensemble development are aimed at ensuring that the ensemble is diverse. This allows the errors of individual algorithms in individual objects to be compensated for by the correct operations of other algorithms. Overall, building the ensemble results in an improvement in both the quality and variety of simple algorithms. The goal is to create a diverse set of predictions that complement each other and reduce the overall variance and bias of the ensemble's predictions.

The simplest type of ensemble is model averaging, whereby each member of the ensemble makes an equal contribution to the final forecast. The fact that each model has an equal contribution to the final ensemble's forecast is a limitation of this approach. The problem is in unbalanced contributions. Despite that, there is a requirement that all members of the ensemble have prediction skills higher than random chance.

However, it is known that some models work much better or much worse than other models. Some improvements can be made to solve this problem, using a weighted ensemble in which the contribution of each member to the final forecast is weighted by the performance of the model. When the weight of the model is a small positive value and the sum of all weights equals 1, the weights can indicate the percentage of confidence in (or expected performance from) each model.

At this time, the most common approaches to ensemble construction are as follows:

- **Bagging:** This is an ensemble of models studying in parallel on different random samples from the same training set. The final result is determined by the voting of the algorithms of the ensemble. For example, in classification, the class that is predicted by the most classifiers is chosen.
- **Boosting:** This is an ensemble of models trained sequentially, with each successive algorithm being trained on samples in which the previous algorithm made a mistake.
- **Stacking:** This is an approach whereby a training set is divided into N blocks, and a set of simple models is trained on $N-1$ of them. An N -th model is then trained on the remaining block, but the outputs of the underlying algorithms (forming the so-called **meta-attribute**) are used as the target variable.
- **Random forest:** This is a set of decision trees built independently, and whose answers are averaged and decided by a majority vote.

The following sections discuss the previously described approaches in detail.

Using a bagging approach for creating ensembles

Bagging (from the bootstrap aggregation) is one of the earliest and most straightforward types of ensembles. Bagging is based on the statistical bootstrap method, which aims to obtain the most accurate sample estimates and to extend the results to the entire population. The bootstrap method is as follows.

Suppose there is an X dataset of size M . Evenly select from the dataset N objects and return each object back to the dataset after selection. Before selecting the next one, we can generate N sub-datasets. This procedure means that N times, we select an arbitrary sample object (we assume that each object is *picked up* with the same probability $\frac{1}{M}$), and each time, we choose from all the original M objects. Also, this procedure is called sampling with replacement, and it means that each element in the dataset has an equal chance of being selected multiple times.

We can imagine this as a bag from which balls are taken. The ball selected at a given step is returned to the bag following its selection, and the next choice is again made with equal probability from the same number of balls. Note that due to the ball being returned each time, there are repetitions.

Each new selection is denoted as X_1 . Repeating the procedure k times, we generate k sub-datasets. Now, we have a reasonably large number of samples, and we can evaluate various statistics of the original distribution.

The main descriptive statistics are the sample mean, median, and standard deviation. Summary statistics—for example, the sample mean, median, and correlation—can vary from sample to sample. The bootstrap idea is to use sampling results as a fictitious population to determine the sample distribution of statistics. The bootstrap method analyzes a large number of phantom samples, called

bootstrap samples. For each sample, an estimate of the target statistics is calculated, then the estimates are averaged. The bootstrap method can be viewed as a modification of the **Monte Carlo method**.

Suppose there is the X training dataset. With the help of the bootstrap method, we can generate x_1, \dots, x_n sub-datasets. Now, on each sub-dataset, we can train our $b_i(x)$ classifier. The final classifier averages these classifier responses (in the case of classification, this corresponds to a vote), as follows:
 $a(x) = \frac{1}{M} \sum_{i=1}^M b_i(x)$. The following diagram shows this scheme:

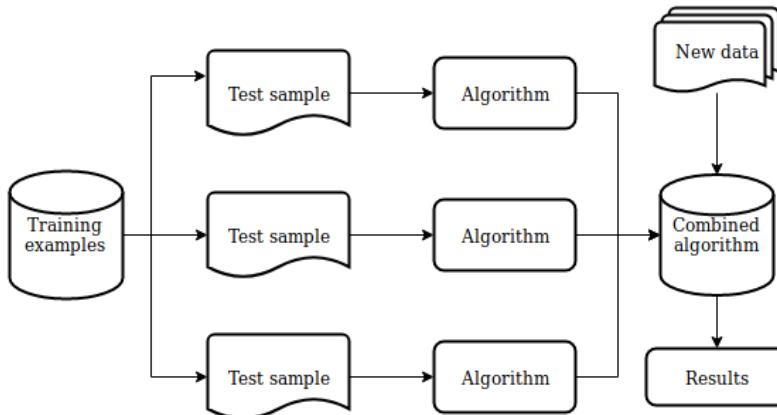


Figure 9.1 – Bagging approach scheme

Consider the regression problem by using simple algorithms $b_1(x), \dots, b_n(x)$. Suppose that there is a true answer function for all $y(x)$ objects, and there is also a distribution on $p(x)$ objects. In this case, we can write the error of each regression function as follows:

$$\varepsilon_i(x) = b_i(x) - y(x), i = 1, \dots, n$$

We can also write the expectation of the **mean squared error (MSE)** as follows:

$$E_x (b_i(x) - y(x))^2 = E_x \varepsilon_i^2(x)$$

The average error of the constructed regression functions is as follows:

$$E_1 = \frac{1}{n} E_x \sum_{i=1}^n \varepsilon_i^2(x)$$

Now, suppose the errors are unbiased and uncorrelated, as shown here:

$$\begin{aligned} E_x \varepsilon_i(x) &= 0, \\ E_x \varepsilon_i(x) \varepsilon_j(x) &= 0, i \neq j \end{aligned}$$

Now we can write a new regression function that averages the responses of the functions we have constructed, as follows:

$$a(x) = \frac{1}{n} \sum_{i=1}^n b_i(x)$$

Let's find its **root MSE (RMSE)** to see the effect of averaging, as follows:

$$\begin{aligned} E_n &= E_x \left(a(x) - y(x) \right)^2 \\ &= E_x \left(\frac{1}{n} \sum_{i=1}^n b_i(x) - y(x) \right)^2 \\ &= E_x \left(\frac{1}{n} \sum_{i=1}^n \varepsilon_i \right)^2 \\ &= \frac{1}{n^2} E_x \left(\sum_{i=1}^n \varepsilon_i^2(x) + \sum_{i \neq j} \varepsilon_i(x) \varepsilon_j(x) \right) \\ &= \frac{1}{n} E_1 \end{aligned}$$

Thus, averaging the answers has allowed us to reduce the average square of the error by n times.

Bagging also allows us to reduce the variance of the trained algorithm and prevent overfitting. The effectiveness of bagging is based on the underlying algorithms, which are trained on various sub-datasets that are quite different, and their errors are mutually compensated during voting. Also, outlying objects may not fall into some of the training sub-datasets, which also increases the effectiveness of the bagging approach.

Bagging is useful with small datasets when the exclusion of even a small number of training objects leads to the construction of substantially different simple algorithms. In the case of large datasets, sub-datasets that are significantly smaller than the original ones are usually generated.

Notice that the assumption about uncorrelated errors is rarely satisfied. If this assumption is incorrect, then the error reduction is not as significant as we might have assumed.

In practice, bagging provides a good improvement to the accuracy of results when compared to simple individual algorithms, particularly if a simple algorithm is sufficiently accurate but unstable. Improving the accuracy of the forecast occurs by reducing the spread of the error-prone forecasts of individual algorithms. The advantage of the bagging algorithm is its ease of implementation, as well as the possibility of parallelizing the calculations for training each elementary algorithm on different computational nodes.

Using a gradient boosting method for creating ensembles

The main idea of boosting is that the elementary algorithms are not built independently. We build every sequential algorithm so that it corrects the mistakes of the previous ones and therefore improves the quality of the whole ensemble. The first successful version of boosting was **Adaptive Boosting (AdaBoost)**. It is now rarely used since gradient boosting has supplanted it.

Suppose that we have a set of pairs, where each pair consists of attribute x and target variable y , $\{(x_i, y_i)\}_{i=1, \dots, n}$. On this set, we restore the dependence of the form $y = f(x)$. We restore it by the approximation $\hat{f}(x)$. To select the best approximation solution, we use a specific loss function of the form $L(y, f)$, which we should optimize as follows:

$$\begin{aligned} y &\approx \hat{f}(x), \\ \hat{f}(x) &= \arg \min_{f(x)} L(y, f(x)) \end{aligned}$$

We also can rewrite the expression in terms of mathematical expectations, since the amount of data available for learning is limited, as follows:

$$\hat{f}(x) = \arg \min_{f(x)} \mathbb{E}_{x,y}[L(y, f(x))]$$

Our approximation is inaccurate. However, the idea behind boosting is that such an approximation can be improved by adding to the model with the result of another model that corrects its errors, as illustrated here:

$$\hat{f}_{m+1}(x) = \hat{f}_m(x) + h(x)$$

The following equation shows the ideal error correction model:

$$\hat{f}_{m+1}(x) = \hat{f}_m(x) + h(x) = y$$

We can rewrite this formula in the following form, which is more suitable for the corrective model:

$$h(x) = y - \hat{f}_m(x)$$

Based on the preceding assumptions listed, the goal of boosting is to approximate $h(x)$ to make its results correspond as closely as possible to the *residuals* $y - F_m(x)$. Such an operation is performed sequentially—that is, \hat{f}_{m+1} improves the results of the previous \hat{f}_m function.

A further generalization of this approach allows us to consider the residuals as a negative gradient of the loss function, specifically of the form $\frac{1}{2}(y - \hat{f}(x))^2$. In other words, gradient boosting is a method of GD with the loss function and its gradient replacement.

Now, knowing the expression of the loss function gradient, we can calculate its values on our data. Therefore, we can train models so that our predictions are better correlated with this gradient (with a minus sign). Hence, we will solve the regression problem, trying to correct the predictions for these residuals. For classification, regression, and ranking, we always minimize the squared difference between the residuals and our predictions.

In the gradient boosting method, an approximation of the function of the following form is used:

$$\hat{f}(x) = \sum_{i=1}^M \gamma_i h_i(x) + \text{const}$$

This is the sum of $h_i(x)$ functions of the \mathcal{H} class; they are collectively called **weak models** (algorithms). Such an approximation is carried out sequentially, starting from the initial approximation, which is a certain constant, as follows:

$$F_0(x) = \arg \min_{\gamma} \sum_{i=1}^n L(y_i, \gamma)$$

$$F_m(x) = F_{m-1}(x) + \arg \min_{h_m \in \mathcal{H}} \left[\sum_{i=1}^n L(y_i, F_{m-1}(x_i) + h_m(x_i)) \right]$$

Unfortunately, the choice of the h optimal function at each step for an arbitrary loss function is extremely difficult, so a more straightforward approach is used. The idea is to use the GD method by using differentiable $h \in \mathfrak{R}$ functions and a differentiable loss function, as illustrated here:

$$F_m(x) = F_{m-1}(x) - \gamma_m \sum_{i=1}^n \nabla_{F_{m-1}} L(y_i, F_{m-1}(x_i))$$

$$\gamma_m = \arg \min_{\gamma} \sum_{i=1}^n L(y_i, F_{m-1}(x_i) - \gamma \nabla_{F_{m-1}} L(y_i, F_{m-1}(x_i)))$$

The boosting algorithm is then formed as follows:

1. Initialize the model with constant values, like this:

$$F_0(x) = \arg \min_{\gamma} \sum_{i=1}^n L(y_i, \gamma)$$

2. Repeat the specified number of iterations and do the following:

I. Calculate the pseudo-residuals, as follows:

$$r_{im} = - \left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \dots, n$$

Here, n is the number of training samples, m is the iteration number, and L is the loss function.

II. Train the elementary algorithm (regression model) $h_m(x)$ on pseudo-residuals with data of the form $\{(x_i, r_{im})\}_{i=1}^n$.

III. Calculate the γ_m coefficient by solving a one-dimensional optimization problem as follows:

$$\gamma_m = \arg \min_{\gamma} \sum_{i=1}^n L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i))$$

IV. Update the model, as follows:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

The inputs to this algorithm are as follows:

- The $\{(x_i, y_i)\}_{i=1, \dots, n}$ dataset
- The number of M iterations
- The $L(y, f)$ loss function with an analytically written gradient (such a form of gradient allows us to reduce the number of numerical calculations)
- The choice of the family of functions of the $h(x)$ elementary algorithms, with the procedure of their training and hyperparameters

The constant for the initial approximation, as well as the γ -optimal coefficient, can be found by a binary search, or by another line search algorithm relative to the initial loss function (rather than the gradient).

Examples of loss functions for regression are as follows:

- $L(y, f) = (y - f)^2$: An L_2 loss, also called **Gaussian loss**. This formula is the classic conditional mean and the most common and simple option. If there are no additional information or model sustainability requirements, it should be used.
- $L(y, f) = |y - f|$: An L_1 loss, also called **Laplacian loss**. This formula, at first glance, is not very differentiable and determines the conditional median. The median, as we know, is more resistant to outliers. Therefore, in some problems, this loss function is preferable since it does not penalize large deviations as much as a quadratic function.

- $L(y, f) = \begin{cases} (1 - \alpha) \cdot |y - f|, & \text{if } y - f \leq 0 \\ \alpha \cdot |y - f|, & \text{if } y - f > 0 \end{cases}$: An L_q loss, also called **Quantile loss**. If we don't want a conditional median but do want a conditional 75% quantile, we would use this option with $\alpha = 0.75$. This function is asymmetric and penalizes more observations that turn out to be on the side of the quantile we need.

Examples of loss functions for classification are as follows:

- $L(y, f) = \log(1 + \exp(-2yf))$ Logistic loss, also known as **Bernoulli loss**. An interesting property of this loss function is that we penalize even correctly predicted class labels. By optimizing this loss function, we can continue to distance classes and improve the classifier even if all observations are correctly predicted. This function is the most standard and frequently used loss function in a binary classification task.
- $L(y, f) = \exp(-yf)$: **AdaBoost loss**. It so happens that the classic AdaBoost algorithm that uses this loss function (different loss functions can also be used in the AdaBoost algorithm) is equivalent to gradient boosting. Conceptually, this loss function is very similar to logistic loss, but it has a stronger exponential penalty for classification errors and is used less frequently.

The idea of bagging is that it can be used with a gradient boosting approach too, which is known as **stochastic gradient boosting**. In this way, a new algorithm is trained on a sub-sample of the training set. This approach can help us to improve the quality of the ensemble and reduce the time it takes to build elementary algorithms (whereby each is trained on a reduced number of training samples).

Although boosting itself is an ensemble, other ensemble schemes can be applied to it—for example, by averaging several boosting methods. Even if we average boosts with the same parameters, they will differ due to the stochastic nature of the implementation. This randomness comes from the choice of random sub-datasets at each step or selecting different features when we are building decision trees (if they are chosen as elementary algorithms).

Currently, the base **gradient boosting machine (GBM)** has many extensions for different statistical tasks. These are as follows:

- GLMBoost and GAMBoost as an enhancement of the existing **generalized additive model (GAM)**
- CoxBoost for survival curves
- RankBoost and LambdaMART for ranking

Additionally, there are many implementations of the same GBM under different names and different platforms, such as these:

- **Stochastic GBM**
- **Gradient boosted decision trees (GBDT)**
- **Gradient boosted regression trees (GBRT)**

- **Multiple additive regression trees (MART)**
- **Generalized boosting machine (GBM)**

Furthermore, boosting can be applied and used over a long period of time in the ranking tasks undertaken by search engines. The task is written based on a loss function, which is penalized for errors in the order of search results; therefore, it becomes convenient to insert it into a GBM.

Using a stacking approach for creating ensembles

The purpose of stacking is to use different algorithms trained on the same data as elementary models. A meta-classifier is then trained on the results of the elementary algorithms or source data, also supplemented by the results of the elementary algorithms themselves. Sometimes, a meta-classifier uses the estimates of distribution parameters that it receives (for example, estimates of the probabilities of each class for classification) for its training, rather than the results of elementary algorithms.

The most straightforward stacking scheme is blending. For this scheme, we divide the training set into two parts. The first part is used to teach a set of elementary algorithms. Their results can be considered new features (meta-features). We then use them as complementary features with the second part of the dataset and train the new meta-algorithm. The problem with such a blending scheme is that neither the elementary algorithms nor the meta-algorithm use the entire set of data for training. To improve the quality of blending, you can average the results of several blends trained at different partitions in the data.

A second way to implement stacking is to use the entire training set. In some sources, this is known as *generalization*. The entire set is divided into parts (folds), then the algorithm sequentially goes through the folds and teaches elementary algorithms on all the folds except the one randomly chosen fold. The remaining fold is used for the inference of the elementary algorithms. The output values of elementary algorithms are interpreted as the new meta-attributes (or new features) calculated from the folds. In this approach, it is also desirable to implement several different partitions into folds, and then average the corresponding meta-attributes. For a meta-algorithm, it makes sense to apply regularization or add some normal noise to the meta-attributes. The coefficient with which this addition occurs is analogous to the regularization coefficient. We can summarize that the basic idea behind the described approach is to use a set of base algorithms; then, using another meta-algorithm, we combine their predictions with the aim of reducing the generalization error.

Unlike boosting and traditional bagging, you can use algorithms of a different nature (for example, a ridge regression in combination with a random forest) in stacking. However, it is essential to remember that for different algorithms, different feature spaces are needed. For example, if categorical features are used as target variables, then the random forest algorithm can be used as-is, but for the regression algorithms, you must first run one-hot encoding.

Since meta-features are the results of already trained algorithms, they strongly correlate. This fact is *a priori* one of the disadvantages of this approach; the elementary algorithms are often under-optimized during training to combat correlation. Sometimes, to combat this drawback, the training of elementary algorithms is used not on the target feature, but on the differences between a feature and the target.

Using the random forest method for creating ensembles

Before we move to the random forest method, we need to familiarize ourselves with the decision tree algorithm, which is the basis for the random forest ensemble algorithm.

Decision tree algorithm overview

A decision tree is a supervised machine learning algorithm based on how a human solves the task of forecasting or classification. Generally, this is a k -dimensional tree with decision rules in the nodes and a prediction of the objective function at the leaf nodes. The decision rule is a function that allows you to determine which of the child nodes should be used as a parent for the considered object. There can be different types of objects in the decision tree leaf—namely, the class label assigned to the object (in the classification tasks), the probability of the class (in the classification tasks), and the value of the objective function (in the regression task).

In practice, binary decision trees are used more often than trees with an arbitrary number of child nodes.

The algorithm for constructing a decision tree in its general form is formed as follows:

1. Firstly, check the criterion for stopping the algorithm. If this criterion is executed, select the prediction issued for the node. Otherwise, we have to split the training set into several non-intersecting smaller sets.
2. In the general case, a $Q_t(x)$ decision rule is defined at the t node, which takes into account a certain range of values. This range is divided into R_t disjoint sets of objects: S_1, S_2, \dots, S_{R_t} , where R_t is the number of descendants of the node, and each S_i is a set of objects that fall into the i^{th} descendant.
3. Divide the set in the node according to the selected rule, and repeat the algorithm recursively for each node.

Most often, the $Q_t(x)$ decision rule is simply the feature—that is, $x^{i(t)}$. For partitioning, we can use the following rules:

- $S_t(j) = \{x \in \mathbb{X} : h_j \leq xi(t) \leq h_{j+1}\}$ for chosen boundary values h_1, \dots, h_{j+1} .
- $S_t(1) = \{x \in \mathbb{X} : \langle x, v \rangle \leq 0\}; S_t(2) = \{x \in \mathbb{X} : \langle x, v \rangle > 0\}$, where $\langle x, v \rangle$ is a vector's scalar product. In fact, it is a corner value check.
- $S_t(1) = \{x \in \mathbb{X} : \rho(x, x_0) \leq h\}; S_t(2) = \{x \in \mathbb{X} : \rho(x, x_0) > h\}$, where the distance ρ is defined in some metric space (for example, $\rho(x, y) = |x - y|$).
- $S_t(1) = \{x \in \mathbb{X} : xi(t) \leq h\}; S_t(2) = \{x \in \mathbb{X} : xi(t) > h\}$, where h is a predicate.

In general, you can use any decision rules, but those that are easiest to interpret are better since they are easier to configure. There is no particular point in taking something more complicated than predicates since you can create a tree with 100% accuracy on the training set with the help of the predicates.

Usually, a set of decision rules is chosen to build a tree. To find the optimal one among them for each particular node, we need to introduce a criterion for measuring optimality. The $I(x)$ measure is introduced for this and is used to measure how objects are scattered (regression) or how the classes are mixed (classification) in a specific t node. This measure is called the **impurity function**. It is required for finding a maximum of $\Delta I(X_t, t)$ according to all features and parameters from a set of decision rules, in order to select a decision rule. With this choice, we can generate the optimal partition for the set of objects in the current node.

Information gain, $\Delta I(X_t, t)$, is how much information we can get for the selected split, and is calculated as follows:

$$\Delta I(X_t, t) = I(X_t, t) - \sum_{i=1}^R I(X_{t_i}, t_i) \frac{N(t_i)}{N(t)}$$

In the preceding equation, the following applies:

- R is the number of sub-nodes the current node is broken into
- t is the current node
- t_1, \dots, t_R are the descendant nodes that are obtained with the selected partition
- $N(t_i)$ is the number of objects in the training sample that fall into the child i
- $N(t)$ is the number of objects trapped in the current node
- X_{t_i} are the objects trapped in the t_i^{th} vertex

We can use the MSE or the **mean absolute error (MAE)** as the $I(t)$ impurity function for regression tasks. For classification tasks, we can use the following functions:

- Gini criterion $I(p_1 \dots p_C) = \sum_{k \neq k'} p_k p_{k'} = \sum_i p_i (1 - p_i)$ as the probability of misclassification, specifically if we predict classes with probabilities of their occurrence in a given node
- Entropy $I(p_1 \dots p_C) = - \sum_i p_i \ln p_i$ as a measure of the uncertainty of a random variable
- Classification error $I(p_1 \dots p_C) = 1 - \max_i p_i$ as the error rate in the classification of the most potent class

In the functions described previously, p_i is an *a priori* probability of encountering an object of class i in a node t —that is, the number of objects in the training sample with labels of class i falling into t divided by the total number of objects in t ($p_i = \frac{N(t_i)}{N(t)}$).

The following rules can be applied as stopping criteria for building a decision tree:

- Limiting the maximum depth of the tree
- Limiting the minimum number of objects in the sheet
- Limiting the maximum number of leaves in a tree
- Stopping if all objects in the node belong to the same class
- Requiring that information gain is improved by at least 8% during splitting

There is an error-free tree for any training set, which leads to the problem of overfitting. Finding the right stopping criterion to solve this problem is challenging. One solution is **pruning**—after the whole tree is constructed, we can cut some nodes. Such an operation can be performed using a test or validation set. Pruning can reduce the complexity of the final classifier and improve predictive accuracy by reducing overfitting.

The pruning algorithm is formed as follows:

1. We build a tree for the training set.
2. Then, we pass a validation set through the constructed tree and consider any internal node t and its left and right sub-nodes L_t, R_t .
3. If no one object from the validation sample has reached t , then we can say that this node (and all its subtrees) is insignificant, and make t the leaf (set the predicate's value for this node equal to the set of the majority class using the training set).
4. If objects from the validation set have reached t , then we have to consider the following three values:
 - The number of classification errors from a subtree of t
 - The number of classification errors from the L_t subtree
 - The number of classification errors from the R_t subtree

If the value for the first case is zero, then we make node t as a leaf node with the corresponding prediction for the class. Otherwise, we choose the minimum of these values. Depending on which of them is minimal, we do the following, respectively:

- If the first is minimal, do nothing
- If the second is minimal, replace the tree from node t with a subtree from node L_t
- If the third is minimal, replace the tree from node t with a subtree from node R_t

Such a procedure regularizes the algorithm to beat overfitting and increase the ability to generalize. In the case of a k -dimensional tree, different approaches can be used to select the forecast in the leaf. We can take the most common class among the objects of the training that fall under this leaf for classification. Alternatively, we can calculate the average of the objective functions of these objects for regression.

We apply a decision rule to a new object starting from the tree root to predict or classify new data. Thus, it is determined which subtree the object should go into. We recursively repeat this process until we reach some leaf node and, finally, we return the value of the leaf node we found as the result of classification or regression.

Random forest method overview

Decision trees are a suitable family of elementary algorithms for bagging since they are quite complicated and can ultimately achieve zero errors on any training set. We can use a method that uses random subspaces (such as bagging) to reduce the correlation between trees and avoid overfitting. The elementary algorithms are trained on different subsets of the feature space, which are also randomly selected. An ensemble of decision tree models using the random subspace method can be constructed using the following algorithm.

Where the number of objects for training is N and the number of features is D , proceed as follows:

1. Select L as the number of individual trees in the ensemble.
2. For each individual l tree, select $dl < D$ as the number of features for l . Typically, only one value is used for all trees.
3. For each tree, create an X_n training subset using the bootstrap method.

Now, build decision trees from X_n samples as follows:

1. Select dl random features from the source, then the optimal division of the training set will limit its search to them.
2. According to a given criterion, we choose the best attribute and make a split in the tree according to it.
3. The tree is built until no more than n_{\min} objects remain in each leaf, until we reach a certain height of the tree, or until the training set is exhausted.

Now, to apply the ensemble model to a new object, it is necessary to combine the results of individual models by majority voting or by combining *a posteriori* probabilities. An example of a final classifier is as follows:

$$a(x) = \frac{1}{L} \sum_{i=1}^L l_i(x)$$

Consider the following fundamental parameters of the algorithm and their properties:

- **The number of trees:** The more trees, the better the quality, but the training time and the algorithm's workload also increase proportionally. Often, with an increasing number of trees, the quality of the training set rises (it can even go up to 100% accuracy), but the quality of the test set is asymptotic (so you can estimate the minimum required number of trees).
- **The number of features for the splitting selection:** With an increasing number of features, the forest's construction time increases too, and the trees become more uniform than before. Often, in classification problems, the number of attributes is chosen equal to \sqrt{D} and $D/3$ for regression problems.
- **Maximum tree depth:** The smaller the depth, the faster the algorithm is built and will work. As the depth increases, the quality during training increases dramatically. The quality may also increase on the test set. It is recommended to use the maximum depth (except when there are too many training objects and we obtain very deep trees, the construction of which takes considerable time). When using shallow trees, changing the parameters associated with limiting the number of objects in the leaf and for splitting does not lead to a significant effect (the leaves are already large). Using shallow trees is recommended in tasks with a large number of noisy objects (outliers).
- **The impurity function:** This is a criterion for choosing a feature (decision rule) for branching. It is usually MSE/MAE for regression problems. For classification problems, it is the Gini criterion, the entropy, or the classification error. The balance and depth of trees may vary depending on the specific impurity function we choose.

We can consider a random forest as bagging decision trees, and during these trees' training, we use features from a random subset of features for each partition. This approach is a universal algorithm since random forests exist for solving problems of classification, regression, clustering, anomaly search, and feature selection, among other tasks.

In the following section, we will see how to use different C++ libraries for developing machine learning model ensembles.

Examples of using C++ libraries for creating ensembles

The following sections will show how to use ensembles within the `Dlib` and `mlpack` libraries. There are out-of-the-box implementations of random forest and gradient boosting algorithms in these libraries; we will show how to use their **application programming interfaces (APIs)** to work with these algorithms. Also, we will implement a stacking ensemble technique from scratch, using primitives from the `mlpack` library.

Ensembles with Dlib

There is only the random forest algorithm implementation in the Dlib library, and in this section, we will show the specific API for using it in practice.

To show the random forest algorithm application, we need to have some dataset for this task. Let's create an artificial dataset that models the cosine function. First, we define datatypes to represent samples and label items. The following code sample shows how it is done:

```
using DataType = double;
using SampleType = dlib::matrix<DataType, 0, 1>;
using Samples = std::vector<SampleType>;
using Labels = std::vector<DataType>;
```

Then we define the `GenerateData` function:

```
std::pair<Samples, Labels> GenerateData(DataType start,
                                         DataType end,
                                         size_t n) {
    Samples x;
    x.resize(n);
    Labels y;
    y.resize(n);
    auto step = (end - start) / (n - 1);
    auto x_val = start;
    size_t i = 0;
    for (auto& x_item : x) {
        x_item = SampleType({x_val});
        auto y_val = std::cos(M_PI * x_val);
        y[i] = y_val;
        x_val += step;
        ++i;
    }
    return {x, y};
}
```

The `GenerateData` function takes three parameters: the `start` and `end` values of the generation range and the `n` numbers of points to generate. The implementation simply calculates cosine values in the loop. The function returns a pair of `std::vector` type objects containing the `double` values. The result of this function will be used for testing.

To show that the random forest algorithm really can approximate values, we will add some noise to the original data. The following code snippet shows the `GenerateNoiseData` function implementation:

```
std::pair<Samples, Labels> GenerateNoiseData(DataType start,
                                              DataType end,
                                              size_t n) {
    Samples x;
    x.resize(n);
    Labels y;
    y.resize(n);
    std::mt19937 re(3467);
    std::uniform_real_distribution<DataType> dist(start, end);
    std::normal_distribution<DataType> noise_dist;
    for (size_t i = 0; i < n; ++i) {
        auto x_val = dist(re);
        auto y_val =
            std::cos(M_PI * x_val) + (noise_dist(re) * 0.3);
        x[i] = SampleType({x_val});
        y[i] = y_val;
    }
    return {x, y};
}
```

The `GenerateNoiseData` function also calculates cosine values in the simple loop. It takes the same input parameters as the `GenerateData` function. However, instead of the sequential value generation, this function samples a random value from the specified range on each iteration. For each sample, it calculates the cosine value and also adds the noise samples. The noise is generated by random distribution. The function also returns two `std::vector` type objects containing the double values, the first one for training inputs and the second one for target values.

Using these data generation functions, we can create the training and the test datasets as follows:

```
auto [train_samples, train_labels] =
    GenerateNoiseData(start, end, num_samples);
auto [test_samples, test_labels] =
    GenerateData(start, end, num_samples);
```

Now, the usage of the Dlib random forest implementation is very simple. The following code snippet shows it:

```
#include <dlib/random_forest.h>
...
dlib::random_forest_regression_trainer<
    dlib::dense_feature_extractor> trainer;
constexpr size_t num_trees = 1000;
```

```

trainer.set_num_trees(num_trees);
auto random_forest = trainer.train(train_samples, train_labels);
for (const auto& sample : test_samples) {
    auto prediction = random_forest(sample);
    ...
}

```

Here, we used the instance of the `random_forest_regression_trainer` class named `trainer` to create the `random_forest` object with the `train` method. The `trainer` object was configured with the number of trees to use. The `random_forest_regression_trainer` class was parametrized with the `dense_feature_extractor` class—this is the only feature extractor class provided now by the Dlib library, but you can create a custom one. The `train` method simply takes two `std::vector` type objects, the first one for the input data values and the second one for the target data values.

After the training, the `random_forest` object was created, and it was used as a functional object to make a prediction for a single value.

The following figure shows the result of applying the random forest algorithm from the Dlib library. The original data is shown as stars and the predicted data is shown as lines:

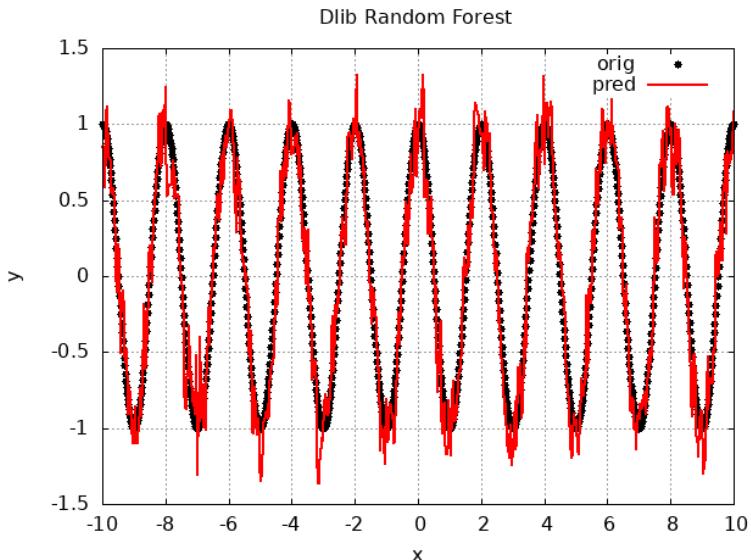


Figure 9.2 – Regression with random forest and Dlib

Note that this method is not very applicable to the regression task on this dataset. You can see that global trends were learned successfully but there are a lot of errors in small details.

Ensembles with `mlpack`

There are two ensemble learning algorithms in the `mlpack` library: the random forest and AdaBoost algorithms. For this set of samples, we will use the *Breast Cancer Wisconsin (Diagnostic)* dataset located at <https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic>. It is taken from *D. Dua, and C. Graff (2019), UCI Machine Learning Repository* (<http://archive.ics.uci.edu/ml>).

There are 569 instances in this dataset, and each instance has 32 attributes: the ID, the diagnosis, and 30 real-value input features. The diagnosis can have two values: M = malignant, and B = benign. Other attributes have 10 real-value features computed for each cell nucleus, as follows:

- Radius (mean distance from the center to the perimeter)
- Texture (standard deviation of grayscale values)
- Perimeter
- Area
- Smoothness (local variation in radius lengths)
- Compactness
- Concavity (severity of concave portions of the contour)
- Concave points (number of concave portions of the contour)
- Symmetry
- Fractal dimension (*coastline approximation*—1)

This dataset can be used for a binary classification task.

Data preparation for `mlpack`

There is the `DatasetInfo` class in `mlpack` to describe a dataset. An instance of this class can be used with different algorithms. Also, there is the `data::Load` function in `mlpack` that can automatically load datasets from the `.csv`, `.tsv`, and `.txt` files. However, this function assumes that there is only numerical data in such files that can be interpreted as a matrix. In our case, the data is in the `.csv` format but the `Diagnosis` column contains the string values `M` and `B`. So, we simply convert them into 0 and 1.

When we have the correct `.csv` file, the data can be loaded in the following way:

```
arma::mat data;
mlpack::data::DatasetInfo info;
data::Load(dataset_name, data, info, /*fail with error*/ true);
```

We passed into the `Load` function the dataset filename and references to the `data` matrix object and the `info` object. Also, notice that we asked the function to generate an exception in the fail case by passing the last parameter as `true`.

Then we split the data into the input and target parts as follows:

```
// extract the labels row
arma::Row<size_t> labels;
labels = arma::conv_to<arma::Row<size_t>>::from( data.row(0));

// remove the labels row
data.shed_row(0);
```

Here, we used the `row` method of the matrix object to get the particular row. Then we converted its values into the `size_t` type with the `arma::conv_to` function because the first row in our dataset consists of labels. Finally, we removed the first row from the `data` object to make it usable as input data.

Having input data and labels matrices, we can split them into the training and testing parts as follows:

```
// split dataset into the train and test parts - make views
size_t train_num = 500;
arma::Row<size_t> train_labels = labels.head_cols( train_num );
arma::mat test_input = data.tail_cols( num_samples - train_num );
arma::Row<size_t> test_labels =
    labels.tail_cols( num_samples - train_num );
```

We used the `head_cols` method of the matrix object to take the first `train_num` columns from the input data and label them as the train values, and we used the `tail_cols` method of the matrix object to take the last columns as the test values.

Using random forest with mlpack

The random forest algorithm in the `mlpack` library is located in the `RandomForest` class. This class has two main methods: `Train` and `Classify`. The `Train` method can be used as follows:

```
using namespace mlpack;
RandomForest<> rf;
rf.Train(train_input,
          train_labels,
          num_classes,
          /*numTrees=*/100,
          /*minimumLeafSize=*/10,
          /*minimumGainSplit=*/1e-7,
          /*maximumDepth=*/10);
```

The first four parameters are self-descriptive. The last ones are more algorithm-specific. The `minimumLeafSize` parameter is the minimum number of points in each tree's leaf nodes. The `minimumGainSplit` parameter is the minimum gain for splitting a decision tree node. The `maximumDepth` parameter is the maximum allowed tree depth.

After the use of the `Train` method with the training data, the `rf` object can be used for the classification with the `Classify` method. This method takes as its first parameter the single value or the row matrix as input, and the second parameter is the reference to the single prediction value or the vector of predictions that will be filled by this method.

There is the `Accuracy` class in `mlpack` that can be used to estimate an algorithm's accuracy. It can work with different algorithm objects and have a unified interface. We can use it as follows:

```
Accuracy acc;
auto acc_value = acc.Evaluate(rf, test_input, test_labels);
std::cout << "Random Forest accuracy = " << acc_value << std::endl;
```

We used the `Evaluate` method to get the accuracy value for the random forest algorithm trained with our data. The printed value is `Random Forest accuracy = 0.971014`.

Using AdaBoost with mlpack

Another ensemble-based algorithm in the `mlpack` library is AdaBoost. It is based on an ensemble of weak learners that is used to produce a strong learner. Let's define an AdaBoost algorithm object based on simple perceptrons as weak learners, as follows:

```
using namespace mlpack;
Perceptron<> p;
AdaBoost<Perceptron<>> ab;
```

We parametrized the `AdaBoost` class with the `Perceptron` class as the template parameter. After the `AdaBoost` object is instantiated, we can use the `Train` method to train it with our dataset. The following code snippet shows how to use the `Train` method:

```
ab.Train(train_input,
          train_labels,
          num_classes,
          p,
          /*iterations*/ 1000,
          /*tolerance*/ 1e-10);
```

The first three input parameters are pretty obvious—the input data, labels, and number of classes for classification. Then we passed the `p` object; it's the instance of the `Perceptron` class, our weak learner. After the weak learner object, we passed into the `Train` method the number of iterations to learn and the accuracy tolerance to stop learning early.

After the strong learner `ab` is trained, we can use the `Classify` method to get classification for a new data value. Also, we can use an object of the `Accuracy` class to estimate the accuracy of the trained algorithm. We already saw how to use `Accuracy` in the previous chapter. Its API is the same for all algorithms in `mlpack`. For AdaBoost, we can use it as follows:

```
Accuracy acc;
auto acc_value = acc.Evaluate(ab, test_input, test_labels);
std::cout << "AdaBoost accuracy = " << acc_value << std::endl;
```

For the AdaBoost algorithm with the same dataset, we got the following output: `AdaBoost accuracy = 0.985507`. The accuracy is slightly better than we got with the random forest algorithm.

Using a stacking ensemble with mlpack

To show the implementation of more ensemble learning techniques, we can develop the stacking approach manually. This is not hard with the `mlpack` library, or indeed any other library.

The stacking approach is based on learning a set of weak learners. Usually, the k-fold technique is used to implement this. It means that we learn a weak model on $k - 1$ folds and use the last fold for predictions. Let's see how we can create k-fold-splitting using `mlpack`. We will use the same dataset as we did for the previous subsections. The main idea is to repeat the dataset to be able to get different folds just by using indices. The following code snippet defines the `KfoldDataSet` structure with just one method and constructor:

```
struct KFoldDataSet {
    KFoldDataSet(const arma::mat& train_input,
                 const arma::Row<size_t>& train_labels,
                 size_t k);
    std::tuple<arma::mat, arma::Row<size_t>, arma::mat,
               arma::Row<size_t>>
    get_fold(const size_t i);
    size_t k{0};
    size_t bin_size{0};
    size_t last_bin_size{0};
    arma::mat inputs;
    arma::Row<size_t> labels;
};
```

The constructor takes the input data, labels, and number of folds for splitting. The `get_fold` method takes an index of a fold and returns four values:

- The matrix contains $k - 1$ folds of the input data
- The row matrix contains $k - 1$ folds of labels
- The matrix contains the last fold of the input data
- The row matrix contains the last fold of labels

Constructor implementation can be as follows:

```
KFoldDataSet(const arma::mat& train_input,
             const arma::Row<size_t>& train_labels, size_t k)
: k(k) {
    fold_size = train_input.n_cols / k;
    last_fold_size = train_input.n_cols - ((k - 1) * fold_size);
    inputs = arma::join_rows(
        train_input, train_input.cols(0, train_input.n_cols -
                                      last_fold_size - 1));
    labels = arma::join_rows(
        train_labels,
        train_labels.cols(
            0, train_labels.n_cols - last_fold_size - 1));
}
```

Here, we calculated the `fold_size` values by dividing the total number of samples in the input data by the number of folds. The total number of samples can be unaligned with the number of folds, so the last fold size can be different. That is why we additionally calculated the `last_fold_size` value to be able to make a correct splitting later. Having the fold size values, we used the `arma::join_rows` function to repeat training samples. This function joins two matrices; for the first parameter, we used the original sample matrices, and for the second, we used reduced ones. We took just the $k-1$ columns for the second parameter using the `cols` method of the matrix object.

When we have the repeated data samples, the `get_fold` method implementation can be as follows:

```
std::tuple<arma::mat, arma::Row<size_t>, arma::mat,
           arma::Row<size_t>>
get_fold(const size_t i) {
    const size_t subset_size =
        (i != 0) ? last_fold_size + (k - 2) * fold_size
                  : (k - 1) * fold_size;
    const size_t last_subset_size =
        (i == k - 1) ? last_fold_size : fold_size;
    // take k-1
    auto input_fold =
        arma::mat(inputs.colptr(fold_size * i), inputs.n_rows,
                  subset_size, false, true);
    auto labels_fold = arma::Row<size_t>(
        labels.colptr(fold_size * i), subset_size, false, true);
    // take last k-th
    auto last_input_fold =
        arma::mat(inputs.colptr(fold_size * (i + k - 1)),
                  inputs.n_rows, last_subset_size, false, true);
```

```

auto last_labels_fold =
    arma::Row<size_t>(labels.colptr(fold_size * (i + k - 1)),
                         last_subset_size, false, true);
return {input_fold, labels_fold, last_input_fold,
        last_labels_fold};
}

```

The most important part of the `get_fold` method is to get the correct number of samples that belong to the $k-1$ subset and the last fold subset. So, at first, we checked whether the required fold to be split is last or not because the last fold may contain a different size of samples. Having this information, we just multiplied the $k-1$ or $k-2$ numbers by the fold size, and conditionally added the last fold size to have the sample subsets. For the last fold, we also conditionally got the fold size depending on the required fold-splitting index.

Having the correct subset sizes, we used the `colptr` method to get the pointer to the starting column sample from the repeated data. We used such a pointer and the subset size to initialize the `arma::mat` object pointing to the existing data without copying by setting the `copy_aux_mem` constructor parameter to `false`. Using such an approach, we initialized matrices for the $k-1$ fold samples and the last fold samples and returned them as a tuple.

Having the `KfoldDataSet` class, we can move further and implement the `StackingClassification` function. Its declaration can be as follows:

```

void StackingClassification(
    size_t num_classes,
    const arma::mat& raw_train_input,
    const arma::Row<size_t>& raw_train_labels,
    const arma::mat& test_input,
    const arma::Row<size_t>& test_labels);

```

It will take the number of classification classes, the train input and label data, and the test input and label data to estimate the accuracy of the algorithm.

The `StackingClassification` function implementation can be split into the following steps:

1. Prepare training dataset.
2. Create meta-datasets.
3. Train meta-model.
4. Train weak models.
5. Evaluate the test data.

Let's take a look at each of them, one by one. The dataset preparation can be implemented as follows:

```
using namespace mlpack;

// Shuffle data
arma::mat train_input;
arma::Row<size_t> train_labels;
ShuffleData(raw_train_input, raw_train_labels, train_input,
            train_labels);

// Normalize data
data::StandardScaler sample_scaler;
sample_scaler.Fit(train_input);
arma::mat scaled_train_input(train_input.n_rows,
                             train_input.n_cols);
sample_scaler.Transform(train_input, scaled_train_input);
```

We used the `ShuffleData` function to randomize the training and the testing data, and we used the `sample_scaler` object of the `StandardScaler` class to scale our train and test data to the zero mean and unit variance. Notice that we fitted a scaler object on the train data and then used it with the `Transform` method. We will use this scaler object later for the test data too.

Having the prepared dataset, we can create a meta-dataset using weak (or elementary) algorithms that will be used for the stacking. It will be three weak algorithms' models:

- Softmax regression
- Decision tree
- Linear SVM

The meta-dataset generation is based on the training and the evaluation of the weak models on the k-fold splits prepared from the original dataset. We already created the `KFoldDataSet` class for this purpose and its usage will be as follows:

```
size_t k = 30;
KFoldDataSet meta_train(scaled_train_input, train_labels, k);
```

Here, we instantiated the `meta_train` dataset object for the 30-fold split. The following code snippet shows how the meta dataset can be generated using three weak models:

```
for (size_t i = 0; i < k; ++i) {
    auto [fold_train_inputs, fold_train_labels, fold_valid_inputs,
          fold_valid_labels] = meta_train.get_fold(i);
    arma::Row<size_t> predictions;
    auto [fold_train_inputs, fold_train_labels, fold_valid_inputs,
```

```
    fold_valid_labels] = meta_train.get_fold(i);
arma::Row<size_t> predictions;
arma::mat meta_feature;
LinearSVM<> local_weak0;
local_weak0.Train(fold_train_inputs, fold_train_labels,
                  num_classes);
local_weak0.Classify(fold_valid_inputs, predictions);
meta_feature = arma::join_cols(
    meta_feature,
    arma::conv_to<arma::mat>::from(predictions));
SoftmaxRegression local_weak1(fold_train_inputs.n_cols,
                             num_classes);
local_weak1.Train(fold_train_inputs, fold_train_labels,
                  num_classes);
local_weak1.Classify(fold_valid_inputs, predictions);
meta_feature = arma::join_cols(
    meta_feature,
    arma::conv_to<arma::mat>::from(predictions));
DecisionTree<> local_weak2;
local_weak2.Train(fold_train_inputs, fold_train_labels,
                  num_classes);
local_weak2.Classify(fold_valid_inputs, predictions);
meta_feature = arma::join_cols(
    meta_feature,
    arma::conv_to<arma::mat>::from(predictions));
meta_train_inputs =
    arma::join_rows(meta_train_inputs, meta_feature);
meta_train_labels =
    arma::join_rows(meta_train_labels, fold_valid_labels);
}
```

The meta-dataset was stored in the `meta_train_inputs` and `meta_train_labels` matrix objects. Using the loop, we iterated the 30-fold indices and for each index, we called the `get_fold` method of the `meta_train` object. This call gave us the k^{th} fold split, which contained the four matrices for the training and the evaluation. Then we trained the local weak objects, which, in our case, were instances of the `LinearSVM`, `SoftmaxRegression`, and `DecisionTree` class objects.

For their training, we used the fold's training inputs and labels. Having - trained weak models(the `LinearSVM`, `SoftmaxRegression`, and `DecisionTree` models), we evaluated them using the `Classify` method on the fold's test input located in the `fold_valid_input` object. The classification result was placed in the `predictions` object. All three classification results were stacked into the new `meta_feature` matrix object using the `join_cols` function. So, we replaced the original dataset features with new meta-features. This `meta_feature` object was added to the `meta_train_inputs` object using the `join_rows` method. The fold's test labels located

in `fold_valid_labels` were added to the meta-dataset using the `join_rows` function on the `meta_train_labels` object.

After the meta-dataset was created, we used the `DecisionTree` instance to train the meta-model as follows:

```
DecisionTree<> meta_model;
meta_model.Train(meta_train_inputs,
                  meta_train_labels,
                  num_classes);
```

To be able to use this meta-model, we have to create a weak model again. It will be used to generate meta-input features for this meta-model. It can be done as follows:

```
LinearSVM<> weak0;
weak0.Train(scaled_train_input, train_labels, num_classes);

SoftmaxRegression weak1(scaled_train_input.n_cols, num_classes);
weak1.Train(scaled_train_input, train_labels, num_classes);

DecisionTree<> weak2;
weak2.Train(scaled_train_input, train_labels, num_classes);
```

Here, we used the whole training dataset for training each of the weak models.

Having trained the ensemble, we can evaluate it on the test dataset. Since we used data preprocessing, we should also transform our test data in the same way as we transformed our training data. We scale the testing data as follows:

```
arma::mat scaled_test_input(test_input.n_rows,
                           test_input.n_cols);
sample_scaler.Transform(test_input, scaled_test_input);
```

The ensemble evaluation starts by predicting meta-features, using the weak models we trained before. We will store predictions from every weak model in the following objects:

```
arma::mat meta_eval_inputs;
arma::Row<size_t> meta_eval_labels;
The next code snippet shows how we get the meta-features from the weak models:

weak0.Classify(scaled_test_input, predictions);
meta_eval_inputs = arma::join_cols(meta_eval_inputs,
                                   arma::conv_to<arma::mat>::from(predictions));
weak1.Classify(scaled_test_input, predictions);
meta_eval_inputs = arma::join_cols(meta_eval_inputs,
```

```
arma::conv_to<arma::mat>::from(predictions)) ;  
  
weak2.Classify(scaled_test_input, predictions);  
meta_eval_inputs = arma::join_cols(meta_eval_inputs,  
        arma::conv_to<arma::mat>::from(predictions));
```

Here, we stacked predictions from each of the weak models into the `meta_eval_inputs` object as we did for the meta-dataset creation.

After we have created the meta-features, we can pass them as input to the `Classify` method of the `meta_model` object to generate the real predictions. We can also calculate the accuracy, like this:

```
Accuracy acc;  
auto acc_value =  
    acc.Evaluate(meta_model, meta_eval_inputs, test_labels);  
std::cout << "Stacking ensemble accuracy = " << acc_value <<  
std::endl;
```

The output of this code is `Stacking ensemble accuracy = 0.985507`. You can see that this ensemble performs better than the random forest implementation, even with default settings. In the case of some additional tuning, it could give even better results.

Summary

In this chapter, we examined various methods for constructing ensembles of machine learning algorithms. The main purposes of creating ensembles are to reduce the error of the elementary algorithms, expand the set of possible hypotheses, and increase the probability of reaching the global optimum during optimization.

We saw that there are three main approaches to building ensembles: training elementary algorithms on various datasets and averaging the errors (bagging), consistently improving the results of the previous, weaker algorithms (boosting), and learning the meta-algorithm from the results of elementary algorithms (stacking). Note that the methods of building ensembles that we've covered, except stacking, require that the elementary algorithms belong to the same class, and this is one of the main requirements for ensembles. It is also believed that boosting gives more accurate results than bagging but, at the same time, is more prone to overfitting. The main disadvantage of stacking is that it begins to significantly improve the results of elementary algorithms only with a relatively large number of training samples.

In the next chapter, we will discuss the fundamentals of **artificial neural networks (ANNs)**. We'll look at the historical aspect of their creation, go through the basic mathematical concepts used in ANNs, implement a **multilayer perceptron (MLP)** network and a simple **convolutional neural network (CNN)**, and discuss what deep learning is and why it is so trendy.

Further reading

- *Ensemble methods: Bagging & Boosting:* <https://medium.com/@sainikhilesh/difference-between-bagging-and-boosting-f996253acd22>
- *How to explain gradient boosting:* <https://explained.ai/gradient-boosting/>
- Original article by Jerome Friedman called *Greedy Function Approximation: A Gradient Boosting Machine:* <https://jerryfriedman.su.domains/ftp/trebst.pdf>
- Ensemble Learning to Improve Machine Learning Results: <https://www.kdnuggets.com/2017/09/ensemble-learning-improve-machine-learning-results.html>
- Introduction to decision trees: <https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/>
- How to visualize decision trees: <https://explained.ai/decision-tree-viz/>
- *Understanding Random Forest:* <https://towardsdatascience.com/understanding-random-forest-58381e0602d2>

Part 3: Advanced Examples

In this part, we'll describe what neural networks are and how they can be applied to solving image classification tasks. We'll also describe what modern **large language models (LLMs)** are and how they assist in solving neural processing tasks such as sentiment analysis.

This part comprises the following chapters:

- *Chapter 10, Neural Networks for Image Classification*
- *Chapter 11, Sentiment Analysis with BERT and Transfer Learning*



10

Neural Networks for Image Classification

In recent years, we have seen huge interest in **neural networks**, which are successfully used in various areas, such as business, medicine, technology, geology, and physics. Neural networks have come into play wherever it is necessary to solve problems of forecasting, classification, or control. Neural networks are intuitive as they are based on a simplified biological model of the human nervous system. They arose from research in the field of artificial intelligence, namely, from attempts to reproduce the ability of biological nervous systems to learn and correct mistakes by modeling the low-level structure of the brain. Neural networks are compelling modeling methods that allow us to reproduce extremely complex dependencies because they are non-linear. Neural networks also cope better with the *curse of dimensionality* than other methods that don't allow modeling dependencies for a large number of variables.

In this chapter, we'll look at the basic concepts of artificial neural networks and show you how to implement neural networks with different C++ libraries. We'll also go through the implementation of the multilayer perceptron and simple convolutional networks and find out what deep learning is and what its applications are.

The following topics will be covered in this chapter:

- An overview of neural networks
- Delving into convolutional networks
- What is deep learning?
- Examples of using C++ libraries to create neural networks
- Understanding image classification using the LeNet architecture

Technical requirements

- You will need the following to complete this chapter:
- The `Dlib` library
- The `mlpack` library
- The `Flashlight` library
- The `PyTorch` library
- Modern C++ compiler with C++20 support
- CMake build system version ≥ 3.22

The code files for this chapter can be found in the following GitHub repository: <https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition/tree/main/Chapter10>.

An overview of neural networks

In this section, we will discuss what artificial neural networks are and their building blocks. We will learn how artificial neurons work and how they relate to their biological analogs. We will also discuss how to train neural networks with the backpropagation method, as well as how to deal with the overfitting problem.

A neural network is a sequence of neurons interconnected by synapses. The structure of the neural network came into the world of programming directly from biology. Thanks to this structure, computers can analyze and even remember information. Neural networks are based on the human brain, which contains millions of neurons that transmit information in the form of electrical impulses.

Artificial neural networks are inspired by biology because they are composed of elements with similar functionalities to those of biological neurons. These elements can be organized in a way that corresponds to the anatomy of the brain, and they demonstrate a large number of properties that are inherent to the brain. For example, they can learn from experience, generalize previous precedents to new cases, and identify significant features from input data that contains redundant information.

Now, let's understand the process of a single neuron.

Neurons

The **biological neuron** consists of a body and processes that connect it to the outside world. The processes along which a neuron receives excitation are called **dendrites**. The process through which a neuron transmits excitation is called an **axon**. Each neuron has only one axon. Dendrites and axons have a rather complex branching structure. The junction of the axon and a dendrite is called a **synapse**. The following figure shows the biological neuron scheme:

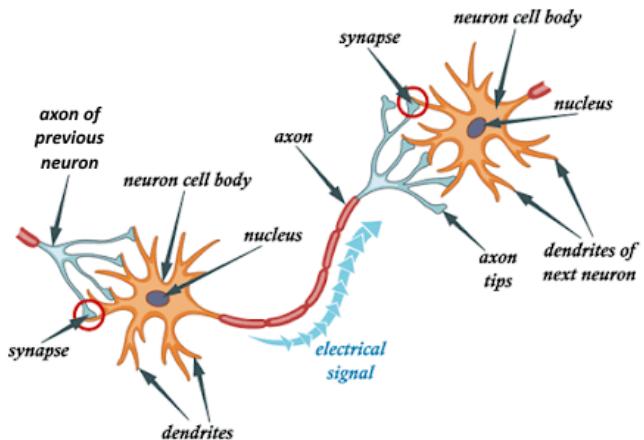


Figure 10.1 – Biological neuron scheme

The main functionality of a neuron is to transfer excitation from dendrites to an axon. However, signals that come from different dendrites can affect the signal in the axon. A neuron gives off a signal if the total excitation exceeds a certain limit value, which varies within certain limits. If the signal is not sent to the axon, the neuron does not respond to excitation. The intensity of the signal that the neuron receives (and therefore the activation possibility) strongly depends on synapse activity. A synapse is a contact for transmitting this information. Each synapse has a length, and special chemicals transmit a signal along it. This basic circuit has many simplifications and exceptions compared to a biological system, but most neural networks model themselves on these simple properties.

The artificial neuron receives a specific set of signals as input, each of which is the output of another neuron. Each input is multiplied by the corresponding weight, which is equivalent to its synaptic power. Then, all the products are summed up, and the result of this summation is used to determine the level of neuron activation. The following diagram shows a model that demonstrates this idea:

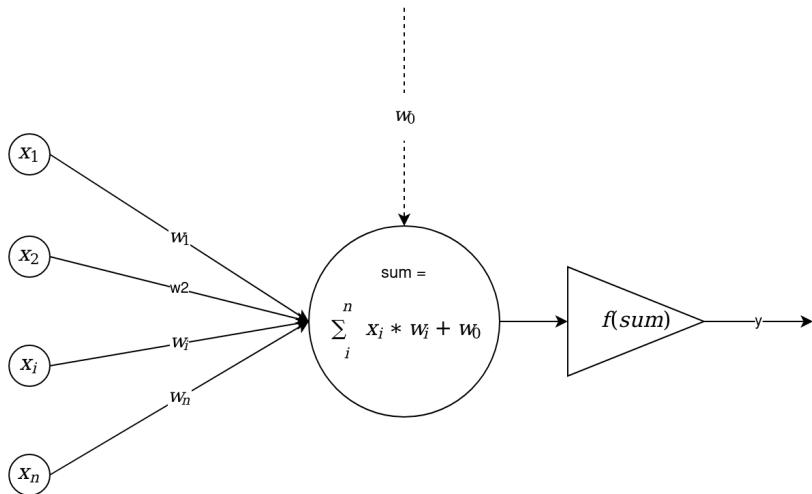


Figure 10.2 – Mathematical neuron scheme

Here, a set of input signals, denoted by x_1, x_2, \dots, x_n , go to an artificial neuron. These input signals correspond to the signals that arrive at the synapses of a biological neuron. Each signal is multiplied by the corresponding weight, w_1, w_2, \dots, w_n , and passed to the summing block. Each weight corresponds to the strength of one biological synaptic connection. The summing block, which corresponds to the body of the biological neuron, algebraically combines the weighted inputs.

The w_0 signal, which is called bias, displays the function of the limit value, known as the **shift**. This signal allows us to shift the origin of the activation function, which subsequently leads to an increase in the neuron's learning speed. The bias signal is added to each neuron. It learns like all the other weights, except it connects to the signal, $+1$, instead of to the output of the previous neuron. The received signal is processed by the activation function, f , and gives a neural signal, y , as output. The activation function is a way to normalize the input data. It narrows the range of sum so that the values of $f(\text{sum})$ belong to a specific interval. That is, if we have a large input number, passing it through the activation function gets us output in the required range. There are many activation functions, and we'll go through them later in this chapter. To learn more about neural networks, we'll have a look at a few more of their components.

The perceptron and neural networks

The first appearance of artificial neural networks can be traced to the article *A logical calculus of the ideas immanent in nervous activity*, which was published in 1943 where an early model of an artificial neuron was proposed. Later, American neurophysiologist Frank Rosenblatt invented the perceptron concept in 1957 as a mathematical model of the human brain's information perception. Currently, terms such as **single-layer perceptron (SLP)**, or just perceptron, and **multilayer perceptron (MLP)** are used. Usually, under the layers in the perceptron is a sequence of neurons, located at the same level and not connected. The following diagram shows this model:

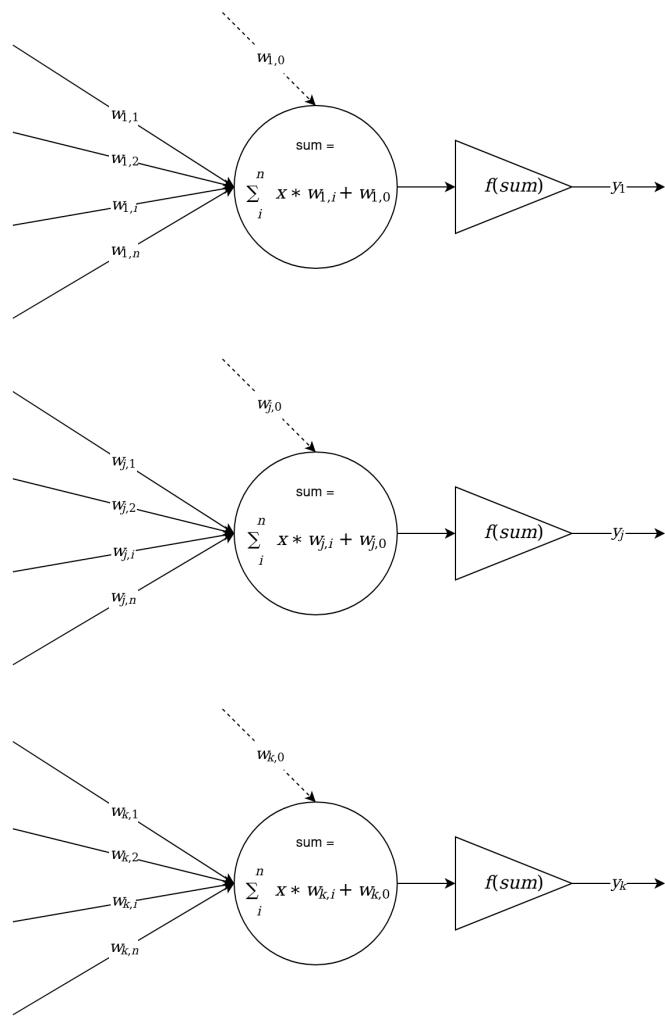


Figure 10.3 – One layer of perceptrons

Typically, we can distinguish between the following types of neural network layers:

- **Input:** This is just the source data or signals arriving as the input of the system (model). For example, these can be individual components of a specific vector from the training set, $x = \{x_1, x_2, \dots, x_n\}$.
- **Hidden:** This is a layer of neurons located between the input and output layers. There can be more than one hidden layer.

- **Output:** This is the last layer of neurons that aggregates the model's work, and its outputs are used as the result of the model's work.

The term **single-layer perceptron** is often understood to describe a model that consists of an input layer and an artificial neuron that aggregates the input data. This term is sometimes used in conjunction with the term **Rosenblatt's perceptron**, but this is not entirely correct since Rosenblatt used a randomized procedure to set up connections between input data and neurons to transfer data to a different dimension, which made it possible to solve the problems that arose when classifying linearly non-separable data. In Rosenblatt's work, a perceptron consists of S and A neuron types, and an R adder. S neurons are the input layers, A neurons are the hidden layers, and the R neuron generates the model's result. The terminology's ambiguity arose because the weights were used only for the R neuron, while constant weights were used between the S and A neuron types. However, note that connections between these types of neurons were established according to a particular randomized procedure:

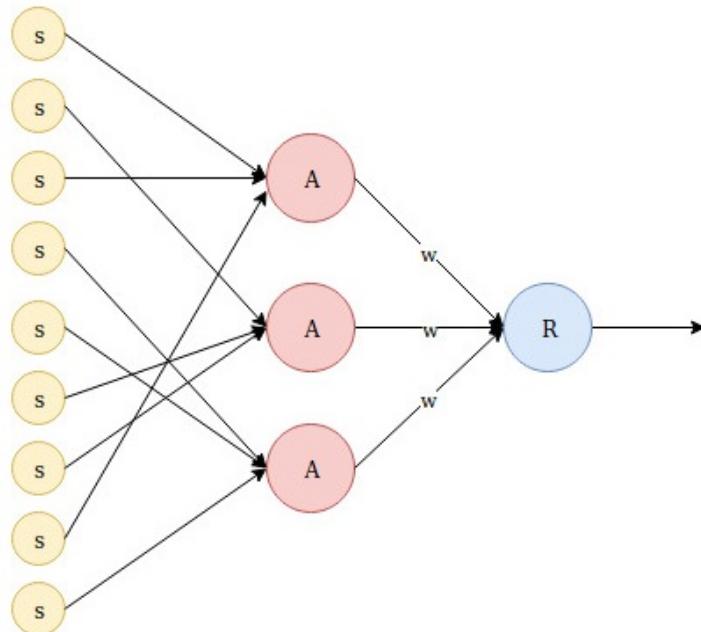


Figure 10.4 – Rosenblatt's perceptron scheme

The term **MLP** refers to a model that consists of an input layer, a certain number of hidden layers, and an output layer. This can be seen in the following diagram:

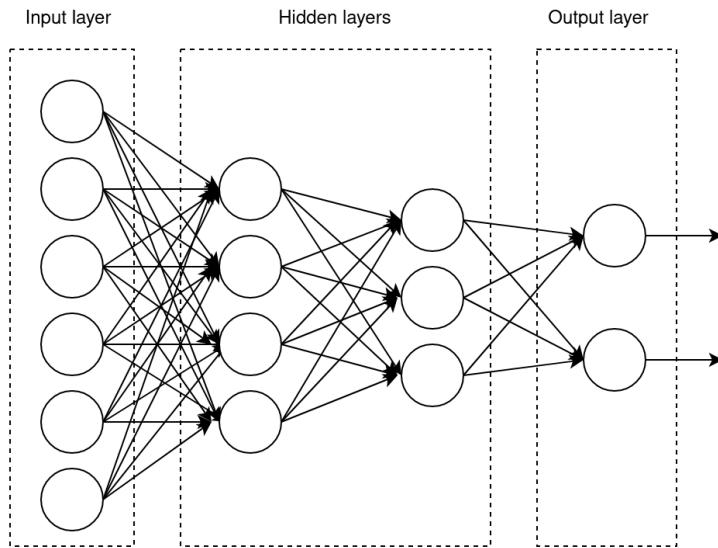


Figure 10.5 – MLP

It should be noted that the architecture of a perceptron (or neural network) includes the direction of signal propagation. In the preceding examples, all communications are directed from the input neurons to the output ones—this is called a feedforward network. Other network architectures may also include feedback between neurons.

The second point that we need to pay attention to in the architecture of a perceptron is the number of connections between neurons. In the preceding diagram, we can see that each neuron in one layer connects to all the neurons in the next layer—this is called a **fully connected layer**. This is not a requirement, but we can see an example of a layer with different types of connections in the *Rosenblatt's perceptron* scheme in *Figure 10.3*.

Now, let's learn about one of the ways in which artificial neural networks can be trained.

Training with the backpropagation method

Let's consider the most common method that's used to train a feedforward neural network: the **error backpropagation method**. It is related to supervised methods. Therefore, it requires target values in the training examples.

The algorithm uses the output error of a neural network. At each iteration of the algorithm, there are two network passes—forward and backward. On a forward pass, an input vector is propagated from the network inputs to its outputs and forms a specific output vector corresponding to the current (actual) state of the weights. Then, the neural network error is calculated. On the backward pass, this error propagates from the network output to its inputs, and the neuron weights are corrected.

The function that's used to calculate the network error is called the **loss function**. An example of such a function is the square of the difference between the actual and target values:

$$E = \frac{1}{2} \sum_{i=1}^k (y'_i - y_i)^2$$

Here, k is the number of output neurons in the network, y' is the target value, and y is the actual output value. The algorithm is iterative and uses the principle of *step-by-step* training; the weights of the neurons of the network are adjusted after one training example is submitted as input. On the backward pass, this error propagates from the network output to its inputs, and the following rule corrects the neuron's weights:

$$\Delta w_{i,j}(n) = -\eta \frac{\partial E_{av}}{\partial w_{ij}} \\ w_{i,j}(n) = w_{i,j}(n-1) + \Delta w_{i,j}(n)$$

Here, $w_{i,j}$ is the weight of the j^{th} connection of the i^{th} neuron, and η is the learning rate parameter, which allows us to control the value of the correction step, $\Delta w_{j,i}$. To accurately adjust to a minimum of errors, this is selected experimentally in the learning process (it varies in the range from 0 to 1).

Choosing an appropriate learning rate can significantly impact the performance and convergence of machine learning models. If the learning rate is too small, the model may converge slowly or not converge at all. On the other hand, if the learning rate is too large, the model can overshoot the optimal solution and diverge, resulting in poor accuracy and overfitting. To avoid such issues, it is important to carefully choose the learning rate based on the specific problem and dataset. Adaptive learning rates (such as Adam) can help automatically adjust the learning rate during training, making it easier to achieve good results. n is the number of the hierarchy of the algorithm (that is, the step number). Let's say that the output sum of the i^{th} neuron is as follows:

$$S_i = \sum_{i=1}^n w_{ij} x_i$$

From this, we can show the following:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial S_i} \frac{\partial S_i}{\partial w_{ij}} = x_i \frac{\partial E}{\partial S_i}$$

Here, we can see that the differential, ∂S_i , of the activation function of the neurons of the network, $f(s)$, must exist and not be equal to zero at any point; that is, the activation function must be differentiable on the entire numerical axis. Therefore, to apply the backpropagation method, sigmoidal activation functions, such as logistic or hyperbolic tangents, are often used.

In practice, training is continued not until the network is precisely tuned to the minimum of the error function, but until a sufficiently accurate approximation is achieved. This process allows us to reduce the number of learning iterations and prevent the network from overfitting.

Currently, many modifications of the backpropagation algorithm have been developed. Let's look at some of them.

Backpropagation method modes

There are three main modes of the backpropagation method:

- Stochastic
- Batch
- Mini-batch

Let's see what these modes are and how they differ from each other.

Stochastic mode

In stochastic mode, the backpropagation method introduces corrections to the weight coefficients immediately after calculating the network output on one training sample.

The stochastic method is slower than the batch method. Given that it does not carry out an accurate gradient descent, instead introducing some *noise* using an undeveloped gradient, it can get out of local minima and produce better results. It is also easier to apply when working with large amounts of training data.

Batch mode

For the batch mode of gradient descent, the loss function is calculated immediately for all available training samples, and then corrections of the weight coefficients of the neuron are introduced by the error backpropagation method.

The batch method is faster and more stable than stochastic mode, but it tends to stop and get stuck at local minima. Also, when it needs to train large amounts of data, it requires substantial computational resources.

Mini-batch mode

In practice, mini-batches are often used as a compromise. The weights are adjusted after processing several training samples (mini-batches). This is done less often than with stochastic descent, but more often than with batch mode.

Now that we've looked at the main backpropagation training modes, let's discuss the problems of the backpropagation method.

Backpropagation method problems

Despite the mini-batch method not being universal, it is widespread at the moment because it provides a compromise between computational scalability and learning effectiveness. It also has individual flaws. Most of its problems come from the indefinitely long learning process. In complex tasks, it may take days or even weeks to train the network. Also, while training the network, the values of the weights can become enormous due to correction. This problem can lead to all or most of the neurons beginning to function at enormous values, in the region where the derivative of the loss function is very small. Since the error that's sent back during the learning process is proportional to this derivative, the learning process can practically freeze.

The gradient descent method can get stuck in a local minimum without hitting a global minimum. The error backpropagation method uses a kind of gradient descent; that is, it descends along the error surface, continuously adjusting the weights until they reach a minimum. The surface of the error of a complex network is rugged and consists of hills, valleys, folds, and ravines in a high-dimensional space. A network can fall into a local minimum when there is a much deeper minimum nearby. At the local minimum point, all directions lead upward, and the network is unable to get out of it. The main difficulty in training neural networks comes down to the methods that are used to exit the local minima: each time we leave a local minimum, the next local minimum is searched by the same method, thereby backpropagating the error until it is no longer possible to find a way out of it.

A careful analysis of the proof of convergence shows that weight corrections are assumed to be infinitesimal. This assumption is not feasible in practice since it leads to an infinite learning time. The step size should be taken as the final size. If the step size is fixed and very small, then the convergence will be too slow, while if it is fixed and too large, then paralysis or permanent instability can occur. Today, many optimization methods have been developed that use a variable correction step size. They adapt the step size depending on the learning process (examples of such algorithms include Adam, Adagrad, RMSProp, Adadelta, and Nesterov Accelerated Gradient).

Notice that there is the possibility of the network overfitting. With too many neurons, the ability of the network to generalize information can be lost. The network can learn an entire set of samples provided for training, but any other images, even very similar ones, may be classified incorrectly. To prevent this problem, we need to use regularization and pay attention to this when designing our network architecture.

The backpropagation method – an example

To understand how the backpropagation method works, let's look at an example.

We'll introduce the following indexing for all expression elements: l is the index of the layer, i is the index of the neuron in the layer, and j is the index of the current element or connection (for example, weight). We use these indexes as follows:

- $q_{i,j}^l$

This expression should be read as the j^{th} element of the i^{th} neuron in the l^{th} layer.

Let's say we have a network that consists of three layers, each of which contains two neurons:

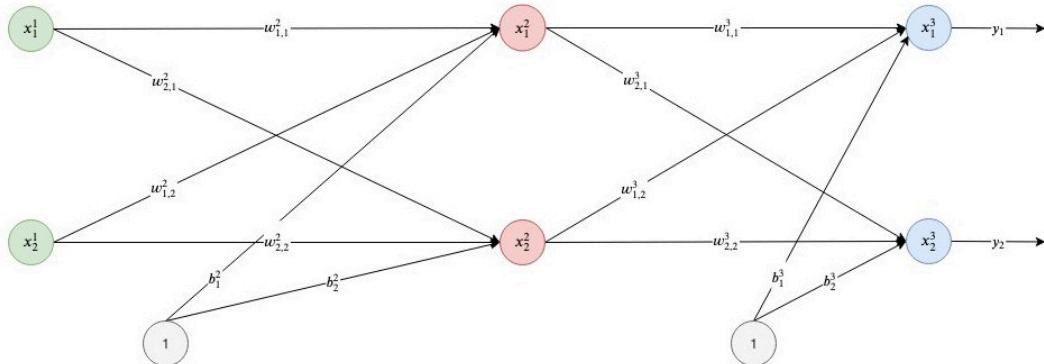


Figure 10.6 – Three-layer neural network

- As the loss function, we choose the square of the difference between the actual and target values:

$$E = \frac{1}{2} \sum (\text{target} - \text{output})^2$$

$$E_{\text{total}} = \frac{1}{2} \sum_{i=1}^{N^l} (y'_i - y_i)^2$$

- Here, y' is the target value of the network output, y is the actual result of the output layer of the network, and N^l is the number of neurons in the output layer.
- This formula calculates the output sum of the neuron, i , in the layer, l :

$$S = \sum wx + b$$

$$s_i^l = \sum_{j=1}^{K_i^l} w_{ij}^l x_{i=j}^{l-1} + b_i^l$$

- Here, K_i^l is the number of inputs of a specific neuron and b_i^l is the bias value for a specific neuron.
- For example, for the first neuron from the second layer, it is equal to the following:

$$s_1^2 = w_{1,1}^2 x_1^1 + w_{1,2}^2 x_2^1 + b_1^2$$

- Don't forget that no weights for the first layer exist because this layer only represents the input values.
- The activation function that determines the output of a neuron should be a sigmoid, as follows:

$$F(x) = \frac{1}{1 + e^{-x}}$$

- Its properties, as well as other activation functions, will be discussed later in this chapter. Accordingly, the output of the i th neuron in the l th layer ($l \neq 1$) is equal to the following:

$$x_i^l = F(s_i^l)$$

- Now, we implement stochastic gradient descent; that is, we correct the weights after each training example and move in a multidimensional space of weights. To get to the minimum of the error, we need to move in the direction opposite to the gradient. We have to add error correction to each weight, $w_{i,j}^l$, based on the corresponding output. The following formula shows how we calculate the error correction value, $\Delta w_{i,j}^l$, with respect to the E_{total} output:

$$\Delta w_{i,j}^l = -\eta \frac{\partial E_{total}}{\partial w_{i,j}^l}$$

- Now that we have the formula for the error correction value, we can write a formula for the weight update:

$$w_{ij}^l = w_{ij}^l + \Delta w_{i,j}^l$$

- Here, η is a learning rate value.
- The partial derivative of the error with respect to the weights, w_{ij}^l , is calculated using the chain rule, which is applied twice. Note that w_{ij}^l affects the error only in the sum, S_i^l :

$$E_{total} = \sum E(F(S(w)))$$

$$\frac{\partial E_{total}}{\partial w_{ij}^l} = \frac{\partial E_{total}}{\partial S_i^l} \frac{\partial S_i^l}{\partial w_{ij}^l} = \frac{\partial E_{total}}{\partial F_i^l} \frac{\partial F_i^l}{\partial S_j^l} \frac{\partial S_j^l}{\partial w_{ij}^l}$$

- We start with the output layer and derive an expression that's used to calculate the correction for the weight, $w_{1,1}^3$. To do this, we must sequentially calculate the components. Consider how the error is calculated for our network:

$$E_1 = \frac{1}{2}(y'_1 - y_1)^2 = \frac{1}{2}(y'_1 - F_1^3(\sum_{j=1}^2 w_{1,j}^3 x_{i=j}^2 + b_1^2))^2$$

$$E_2 = \frac{1}{2}(y'_2 - y_2)^2 = \frac{1}{2}(y'_2 - F_2^3(\sum_{j=1}^2 w_{2,j}^3 x_{i=j}^2 + b_2^2))^2$$

$$E_{total} = E_1 + E_2$$

- Here, we can see that E_2 does not depend on the weight of $w_{1,1}^3$. Its partial derivative with respect to this variable is equal to 0:

$$\frac{\partial E_{total}}{\partial F_i^l} = \frac{\partial E_1}{\partial F_i^l} + \frac{\partial E_2}{\partial F_i^l} = \frac{\partial E_1}{\partial F_i^l} + 0$$

- Then, the general expression changes to follow the next formula:

$$\frac{\partial E_{total}}{\partial w_{1,1}^3} = \frac{\partial E_1}{\partial F_1^3} \frac{\partial F_1^3}{\partial S_1^3} \frac{\partial S_1^3}{\partial w_{1,1}^3}$$

- The first part of the expression is calculated as follows:

$$\frac{\partial E_1}{\partial F_1^3} = 2 \frac{1}{2} (y'_1 - y_1), \text{ where } y_1 = x_1^3$$

- The sigmoid derivative is $F' = F(x)(1 - F(x))$, respectively. For the second part of the expression, we get the following:

$$\frac{\partial F_1^3}{\partial S_1^3} = y_1(1 - y_1) = x_1^3(1 - x_1^3)$$

- The third part is the partial derivative of the sum, which is calculated as follows:

$$\begin{aligned} s_1^3 &= w_{1,1}^3 x_1^2 + w_{1,2}^3 x_2^2 + b_1^3 \\ \frac{\partial S_1^3}{\partial w_{1,1}^3} &= 1(w_{1,1}^3)^{(1-1)} x_1^2 + 0 + 0 = x_1^2 \end{aligned}$$

- Now, we can combine everything into one formula:

$$\frac{\partial E_{total}}{\partial w_{1,1}^3} = (y'_1 - x_1^3) * x_1^3(1 - x_1^3) * x_1^2$$

- We can also derive a general formula in order to calculate the error correction for all the weights of the output layer:

$$\frac{\partial E_{total}}{\partial w_{i,j}^{l_{out}}} = (y'_i - x_i^{l_{out}}) x_i^{l_{out}} (1 - x_i^{l_{out}}) x_{i=j}^{l_{out}-1}$$

- Here, l_{out} is the index of the output layer of the network.
- Now, we can consider how the corresponding calculations are carried out for the inner (hidden) layers of the network. Let's take, for example, the weight, $w_{1,1}^2$. Here, the approach is the same, but with one significant difference—the output of the neuron of the hidden layer is passed to the input of all (or several) of the neurons of the output layer, and this must be taken into account:

$$\frac{\partial E_{total}}{\partial w_{1,1}^2} = \left(\frac{\partial E_1}{\partial F_1^2} + \frac{\partial E_2}{\partial F_1^2} \right) \frac{\partial F_1^2}{\partial S_1^2} \frac{\partial S_1^2}{\partial w_{1,1}^2}$$

$$\begin{aligned} E_1 &= F_1^3(S_1^3(F_1^2)) \\ \frac{\partial E_1}{\partial F_1^2} &= \frac{\partial E_1}{\partial S_1^3} \frac{\partial S_1^3}{\partial F_1^2} = \frac{\partial E_1}{\partial F_1^3} \frac{\partial F_1^3}{\partial S_1^3} \frac{\partial S_1^3}{\partial F_1^2} \end{aligned}$$

- Here, we can see that $\frac{\partial E_1}{\partial F_1^3}$ and $\frac{\partial F_1^3}{\partial S_1^3}$ have already been calculated in the previous step and that we can use their values to perform calculations:

$$\begin{aligned} F_1^2 &= x_1^2 \\ S_1^3 &= w_{1,1}^3 x_1^2 + w_{1,2}^3 x_2^2 + b_1^3 \\ \frac{\partial S_1^3}{\partial F_1^2} &= \frac{\partial S_1^3}{\partial x_1^2} = w_{1,1}^3 \end{aligned}$$

By combining the obtained results, we receive the following output:

$$\frac{\partial E_1}{\partial F_1^2} = \frac{\partial E_1}{\partial F_1^3} \frac{\partial F_1^3}{\partial S_1^3} w_{1,1}^3$$

Similarly, we can calculate the second component of the sum using the values that were calculated in the previous steps— $\frac{\partial E_2}{\partial F_2^3}$ and $\frac{\partial F_2^3}{\partial S_2^3}$:

$$\frac{\partial E_2}{\partial F_1^2} = \frac{\partial E_2}{\partial F_2^3} \frac{\partial F_2^3}{\partial S_2^3} w_{2,1}^3$$

The remaining parts of the expression for weight correction, $w_{1,1}^3$, are obtained as follows, similar to how the expressions were obtained for the weights of the output layer:

$$\begin{aligned} \frac{\partial F_1^2}{\partial S_1^2} &= x_1^2(1 - x_1^2) \\ \frac{\partial S_1^2}{\partial w_{1,1}^2} &= \frac{\partial(w_{1,1}^2 x_1^2 + w_{2,1}^2 x_2^2 + b_1^2)}{\partial w_{1,1}^2} = x_1^2 \end{aligned}$$

By combining the obtained results, we obtain a general formula that we can use to calculate the magnitude of the adjustment of the weights of the hidden layers:

$$\frac{\partial E_{total}}{\partial w_{i,j}^{l_h}} = \left(\sum_{q=1}^{L_{next}} \frac{\partial E}{\partial F_q^{l_{h+1}}} \frac{\partial F_q^{l_{h+1}}}{\partial S_q^{l_{h+1}}} w_{q,j=i}^{l_{h+1}} \right) x_i^{l_h} (1 - x_i^{l_h}) x_{i=j}^{l_h-1}$$

Here, l_h is the index of the hidden layer and L_{next} is the number of neurons in the layer, l_{h+1} .

Now, we have all the necessary formulas to describe the main steps of the error backpropagation algorithm:

1. Initialize all weights, w_{ij}^l , with small random values (the initialization process will be discussed later).
2. Repeat this several times, sequentially, for all the training samples, or a mini-batch of samples.
3. Pass a training sample (or a mini-batch of samples) to the network input and calculate and remember all the outputs of the neurons. These calculate all the sums and values of our activation functions.
4. Calculate the errors for all the neurons of the output layer:

$$\delta_i^{l_{out}} = (y'_i - x_i^{l_{out}}) x_i^{l_{out}} (1 - x_i^{l_{out}})$$

5. For each neuron on all l layers, starting from the penultimate one, calculate the error:

$$\delta_i^l = \left(\sum_{q=1}^{L_{\text{next}}} \delta_q^{l+1} w_{i=q, j=i}^{l+1} \right) x_i^l (1 - x_i^l)$$

Here, L_{next} is the number of neurons in the $l + 1$ layer.

6. Update the network weights:

$$\begin{aligned}\Delta w_{ij}^l &= -\eta \delta_i^l x_i^l \\ w_{ij}^l &= w_{ij}^l + \Delta w_{ij}^l\end{aligned}$$

Here, η is the learning rate value.

There are many versions of the backpropagation algorithm that improve the stability and convergence rate of the algorithm. One of the very first proposed improvements was the use of momentum. At each step, the value Δw is memorized, and at the next step, we use a linear combination of the current gradient value and the previous one:

$$\begin{aligned}\text{step n: } q_{ij}^l &= \Delta w_{ij}^l \\ \text{step n+1: } w_{ij}^l &= w_{ij}^l + \Delta w_{ij}^l + \alpha \Delta q_{ij}^l\end{aligned}$$

α is the hyperparameter that's used for additional algorithm tuning. This algorithm is more common now than the original version because it allows us to achieve better results during training.

The next important element that's used to train the neural network is the loss function.

Loss functions

With the loss function, neural network training is reduced to the process of optimally selecting the coefficients of the matrix of weights in order to minimize the error. This function should correspond to the task, for example, categorical cross-entropy for the classification problem or the square of the difference for regression. Differentiability is also an essential property of the loss function if the backpropagation method is used to train the network. Let's look at some of the popular loss functions that are used in neural networks:

- The **mean squared error (MSE)** loss function is widely used for regression and classification tasks. Classifiers can predict continuous scores, which are intermediate results that are only converted into class labels (usually by a threshold) as the very last step of the classification process. The MSE can be calculated using these continuous scores rather than the class labels. The advantage of this is that we avoid losing information due to dichotomization. The standard form of the MSE loss function is defined as follows:

$$L = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

- The **mean squared logarithmic error (MSLE)** loss function is a variant of the MSE and is defined as follows:

$$L = \frac{1}{n} \sum_{i=1}^n (\log(y_i + 1) - \log(\hat{y}_i + 1))^2$$

By taking the log of the predictions and target values, the variance that we are measuring has changed. It is often used when we do not want to penalize considerable differences in the predicted and target values when both the predicted and actual values are big numbers. Also, the MSLE penalizes underestimates more than overestimates.

- The **L2** loss function is the square of the L2 norm of the difference between the actual value and the target value. It is defined as follows:

$$L = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

- The **mean absolute error (MAE)** loss function is used to measure how close forecasts or predictions are to the eventual outcomes:

$$L = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

The MAE requires complicated tools such as linear programming to compute the gradient. The MAE is more robust to outliers than the MSE since it does not make use of the square.

- The **L1** loss function is the sum of absolute errors of the difference between the actual value and target value. Similar to the relationship between the MSE and L2, L1 is mathematically similar to the MAE except it does not have division by n . It is defined as follows:

$$L = \sum_{i=1}^n |y_i - \hat{y}_i|$$

- The **cross-entropy** loss function is commonly used for binary classification tasks where labels are assumed to take values of 0 or 1. It is defined as follows:

$$L = -\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Cross-entropy measures the divergence between two probability distributions. If the cross-entropy is large, this means that the difference between the two distributions is significant, while if the cross-entropy is small, this means that the two distributions are similar to each other. The cross-entropy loss function has the advantage of faster convergence, and it is more likely to reach global optimization than the quadratic loss function.

- The **negative log-likelihood** loss function is used in neural networks for classification tasks. It is used when the model outputs a probability for each class rather than the class label. It is defined as follows:

$$L = \frac{1}{n} \sum_{i=1}^n \log(\hat{y}_i)$$

- The **cosine proximity** loss function computes the cosine proximity between the predicted value and the target value. It is defined as follows:

$$L = -\frac{\sum_{i=1}^n y_i \cdot \hat{y}_i}{\sqrt{\sum_{i=1}^n (y_i)^2} \cdot \sqrt{\sum_{i=1}^n (\hat{y}_i)^2}}$$

This function is the same as the cosine similarity, which is a measure of similarity between two non-zero vectors. This is expressed as the cosine of the angle between them. Unit vectors are maximally similar if they are parallel and maximally dissimilar if they are orthogonal.

- The **hinge loss** function is used for training classifiers. The hinge loss is also known as the max-margin objective and is used for *maximum-margin* classification. It uses the raw output of the classifier's decision function, not the predicted class label. It is defined as follows:

$$L = \frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i \cdot \hat{y}_i)$$

There are many other loss functions. Complex network architectures often use several loss functions to train different parts of a network. For example, the *Mask RCNN* architecture, which is used for predicting object classes and boundaries on images, uses different loss functions: one for regression and another for classifiers. In the next section, we will discuss the neuron's activation functions.

Activation functions

What does an artificial neuron do? Simply put, it calculates the weighted sum of inputs, adds the bias, and decides whether to exclude this value or use it further. The artificial neuron doesn't know of a threshold that can be used to figure out whether the output value switches neurons to the activated state. For this purpose, we add an activation function. It checks the value that's produced by the neuron for whether external connections should recognize that this neuron is activated or whether it can be ignored. It determines the output value of a neuron, depending on the result of a weighted sum of inputs and a threshold value.

Let's consider some examples of activation functions and their properties.

The stepwise activation function

The stepwise activation function works like this—if the sum value is higher than a particular threshold value, we consider the neuron activated. Otherwise, we say that the neuron is inactive.

A graph of this function can be seen in the following figure:

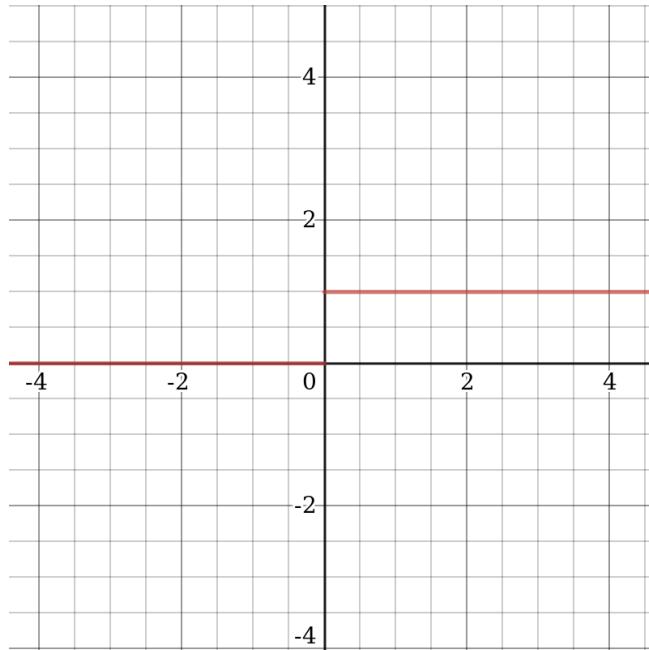


Figure 10.7 – Stepwise activation function

The function returns 1 (the neuron has been activated) when the argument is > 0 (the zero value is a threshold), and the function returns 0 (the neuron hasn't been activated) otherwise. This approach is easy, but it has flaws. Imagine that we are creating a binary classifier—a model that should say *yes* or *no* (activated or not). A stepwise function can do this for us—it prints 1 or 0. Now, imagine a case when more neurons are required to classify many classes: *class1*, *class2*, *class3*, or even more. What happens if more than one neuron is activated? All the neurons from the activation function derive 1.

In this case, questions arise about what class should ultimately be obtained for a given object. We only want one neuron to be activated, and the activation functions of other neurons should be zero (except in this case, we can be sure that the network correctly defines the class). Such a network is more challenging to train and achieve convergence. If the activation function is not binary, then the possible values are activated at 50%, activated at 20%, and so on. If several neurons are activated, we can find the neuron with the highest value of the activation function. Since there are intermediate values at the output of the neuron, the learning process runs smoother and faster.

In the stepwise activation function, the likelihood of several fully activated neurons appearing during training decreases (although this depends on what we are training and on what data). Also, the stepwise activation function is not differentiable at point 0 and its derivative is equal to 0 at all other points. This leads to difficulties when we use gradient descent methods for training.

The linear activation function

The linear activation function, $y = c x$, is a straight line and is proportional to the input (that is, the weighted sum on this neuron). Such a choice of activation function allows us to get a range of values, not just a binary answer. We can connect several neurons and if more than one neuron is activated, the decision is made based on the choice of, for example, the maximum value.

The following diagram shows what the linear activation function looks like:

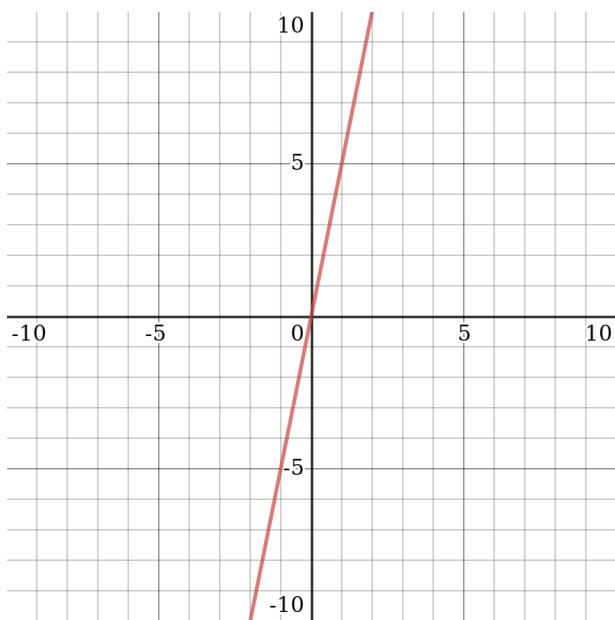


Figure 10.8 – Linear activation function

The derivative of $y = c x$ with respect to x is c . This conclusion means that the gradient has nothing to do with the argument of the function. The gradient is a constant vector, while the descent is made according to a constant gradient. If an erroneous prediction is made, then the backpropagation error's update changes are also constant and do not depend on the change that's made regarding the input.

There is another problem: related layers. A linear function activates each layer. The value from this function goes to the next layer as input while the second layer considers the weighted sum at its inputs and, in turn, includes neurons, depending on another linear activation function. It doesn't matter how many layers we have. If they are all linear, then the final activation function in the last layer is just a

linear function of the inputs on the first layer. This means that two layers (or N layers) can be replaced with one layer. Due to this, we lose the ability to make sets of layers. The entire neural network is still similar to the one layer with a linear activation function because it's the linear combination of linear functions.

The sigmoid activation function

The sigmoid activation function, $y = \frac{1}{1 + e^{-x}}$, is a smooth function, similar to a stepwise function:

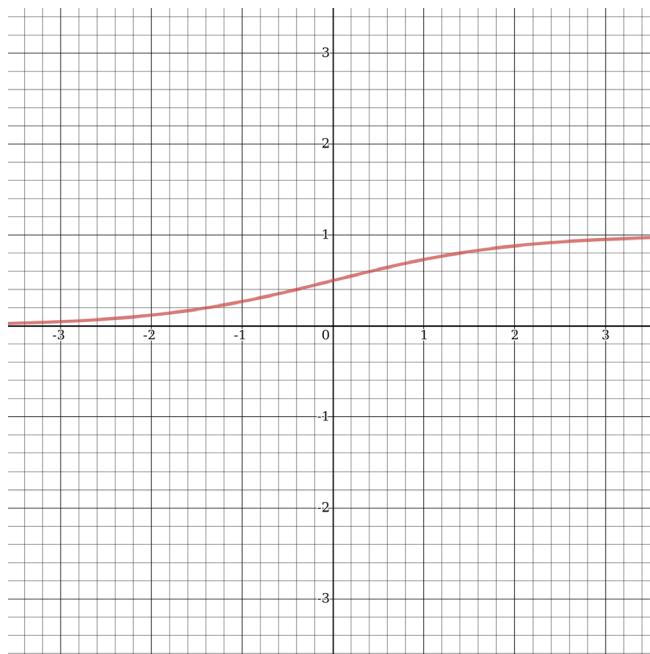


Figure 10.9 – Sigmoid activation function

A sigmoid is a non-linear function, and a combination of sigmoids also produces a non-linear function. This allows us to combine neuron layers. A sigmoid activation function is not binary, which makes an activation with a set of values from the range $[0,1]$, in contrast to a stepwise function. A smooth gradient also characterizes a sigmoid. In the range of values of x from -2 to 2, the values, y , change very quickly. This gradient property means that any small change in the value of x in this area entails a significant change in the value of y . This behavior of the function indicates that y tends to cling to one of the edges of the curve.

The sigmoid looks like a suitable function for classification tasks. It tries to bring the values to one of the sides of the curve (for example, to the upper edge at $x = 2$ and the lower edge at $x = -2$). This behavior allows us to find clear boundaries in the prediction.

Another advantage of a sigmoid over a linear function is as follows: in the first case, we have a fixed range of function values, $[0, 1]$, while a linear function varies within $(-\infty, \infty)$. This is advantageous because it does not lead to errors in numerical calculations when dealing with large values on the activation function.

Today, the sigmoid is one of the most popular activation functions in neural networks. But it also has flaws that we have to take into account. When the sigmoid function approaches its maximum or minimum, the output value of y tends to weakly reflect changes in x . This means that the gradient in such areas takes small values, and the small values cause the gradient to vanish. The **vanishing gradient** problem is a situation where a gradient value becomes too small or disappears and the neural network refuses to learn further or learns very slowly.

The hyperbolic tangent

The hyperbolic tangent is another commonly used activation function. It can be represented graphically as follows:

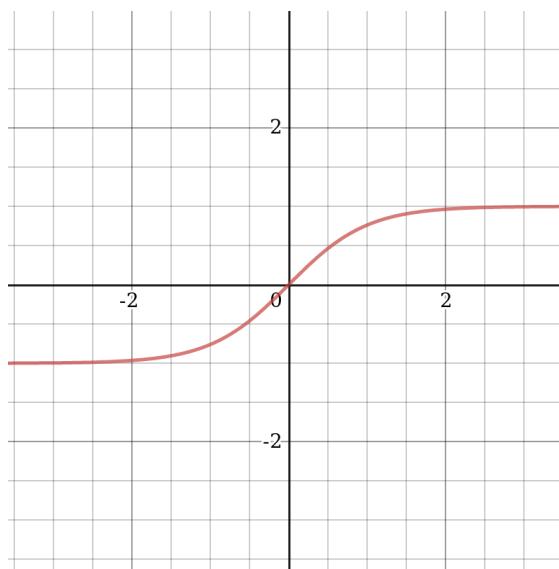


Figure 10.10 – Hyperbolic tangent activation function

The hyperbolic tangent is very similar to the sigmoid. This is the correct sigmoid function, $y = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$. Therefore, such a function has the same characteristics as the sigmoid we looked at earlier. Its nature is non-linear, it is well suited for a combination of layers, and the range of values of the function is $(-1, 1)$. Therefore, it makes no sense to worry about the values of the activation function leading to computational problems. However, it is worth noting that the gradient of the tangential function has higher values than that of the sigmoid (the derivative is steeper than it is for the sigmoid). Whether we choose a sigmoid or a tangent function depends on the requirements of the gradient's amplitude. As well as the sigmoid, the hyperbolic tangent has the inherent vanishing gradient problem.

The **rectified linear unit (ReLU)**, $y = \max(0, x)$, returns x if x is positive, and 0 otherwise:

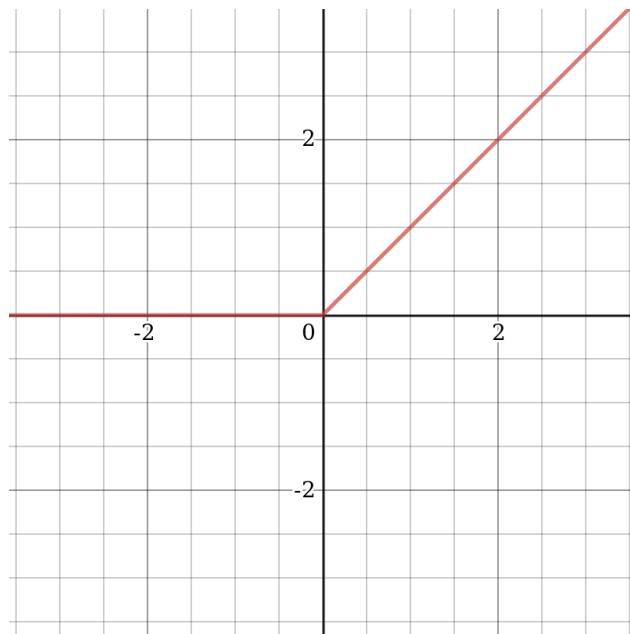


Figure 10.11 – ReLU activation function

At first glance, it seems that ReLU has all the same problems as a linear function since ReLU is linear in the first quadrant. But in fact, ReLU is non-linear, and a combination of ReLU is also non-linear. A combination of ReLU can approximate any function. This property means that we can use layers and they won't degenerate into a linear combination. The range of permissible values of ReLU is $[0, \infty]$, which means that its values can be quite high, thus leading to computational problems. However, this same property removes the problem of a vanishing gradient. It is recommended to use regularization and normalize the input data to solve the problem with large function values (for example, to the range of values $[0, 1]$).

Let's look at activation sparseness as a property of neural networks. Imagine a large neural network with many neurons. The use of a sigmoid or hyperbolic tangent entails the activation of all neurons. This action means that almost all activations must be processed to calculate the network output. In other words, activation is dense and costly.

Ideally, we want some neurons not to be activated, and this would make activations sparse and efficient. ReLU allows us to do this. Imagine a network with randomly initialized weights (or normalized) in which approximately 50% of activations are 0 because of the ReLU property, returning 0 for negative values of x . In such a network, fewer neurons are included (sparse activation), and the network itself becomes lightweight.

Since part of the ReLU is a horizontal line (for negative values of x), the gradient on this part is 0. This property leads to the fact that weights cannot be adjusted during training. This phenomenon is called the **dying ReLU problem**. Because of this problem, some neurons are turned off and do not respond, making a significant part of the neural network passive. However, there are variations of ReLU that help solve this problem. For example, it makes sense to replace the horizontal part of the function (the region where $x < 0$) with the linear one using the expression $y = 0.01x$. There are other ways to avoid a zero gradient, but the main idea is to make the gradient non-zero and gradually restore it during training.

Also, ReLU is significantly less demanding on computational resources than hyperbolic tangent or sigmoid because it performs simpler mathematical operations than the aforementioned functions.

The critical properties of ReLU are its small computational complexity, nonlinearity, and unsusceptibility to the vanishing gradient problem. This makes it one of the most frequently used activation functions for creating deep neural networks.

Now that we've looked at a number of activation functions, we can highlight their main properties.

Activation function properties

The following is a list of activation function properties that are worth considering when deciding which activation function to choose:

- **Nonlinearity:** If the activation function is non-linear, it can be proved that even a two-level neural network can be a universal approximator of the function.
- **Continuous differentiability:** This property is desirable for providing gradient descent optimization methods.
- **Value range:** If the set of values for the activation function is limited, gradient-based learning methods are more stable and less prone to calculation errors since there are no large values. If the range of values is infinite, training is usually more effective, but care must be taken to avoid exploding the gradient (it means that gradient values can get extremal values and the learning ability will be lost).
- **Monotonicity:** If the activation function is monotonic, the error surface associated with the single-level model is guaranteed to be convex. This allows us to learn more effectively.
- **Smooth functions with monotone derivatives:** In some cases, these provide a higher degree of generality.

Now that we've discussed the main components used to train neural networks, it's time to learn how to deal with the overfitting problem, which regularly appears during the training process.

Regularization in neural networks

Overfitting is one of the problems of machine learning models and neural networks in particular. The problem is that the model only explains the samples from the training set, thus adapting to the training samples instead of learning to classify samples that were not involved in the training process (losing the ability to generalize). Usually, the primary cause of overfitting is the model's complexity (in terms of the number of parameters it has). The complexity can be too high for the training set available and, ultimately, for the problem to be solved. The task of the regularizer is to reduce the model's complexity, preserving the number of parameters. Let's consider the most common regularization methods that are used in neural networks.

The most popular regularization methods are L2 regularization, dropout, and batch normalization. Let's take a look at each.

L2 regularization

L2 regularization (weight decay) is performed by penalizing the weights with the highest values. Penalizing is performed by minimizing their L_2 -norm using the λ parameter—a regularization coefficient that expresses the preference for minimizing the norm when we need to minimize losses on the training set. That is, for each weight, w , we add the term, $\frac{\lambda}{2} \|\vec{w}\|^2 = \frac{\lambda}{2} \sum_{i=1}^W w_i^2$, to the loss function, $L(\hat{y}, y)$ (the $\frac{1}{2}$ factor is used so that the gradient of this term with respect to the w parameter is equal to λw and not $2\lambda w$ for the convenience of applying the error backpropagation method). We must select λ correctly. If the coefficient is too small, then the effect of regularization is negligible. If it is too large, the model can reset all the weights.

Dropout regularization

Dropout regularization consists of changing the structure of the network. Each neuron can be excluded from a network structure with some probability, p . The exclusion of a neuron means that with any input data or parameters, it returns 0.

Excluded neurons do not contribute to the learning process at any stage of the backpropagation algorithm. Therefore, the exclusion of at least one of the neurons is equal to learning a new neural network. This *thinning* network is used to train the remaining weights. A gradient step is taken, after which all ejected neurons are returned to the neural network. Thus, at each step of the training, we set up one of the possible $2N$ network architectures. By architecture, we mean the structure of connections between neurons, and by N , we're denoting the total number of neurons. When we are evaluating a neural network, neurons are no longer thrown out. Each neuron output is multiplied by $(1 - p)$. This means that in the neuron's output, we receive its response expectation for all $2N$ architectures. Thus, a neural network trained using dropout regularization can be considered a result of averaging responses from an ensemble of $2N$ networks.

Batch normalization

Batch normalization makes sure that the effective learning process of neural networks isn't impeded. The input signal can be significantly distorted by the mean and variance as the signal propagates through the inner layers of a network, even if we initially normalized the signal at the network input. This phenomenon is called the internal covariance shift and is fraught with severe discrepancies between the gradients at different levels or layers. Therefore, we have to use stronger regularizers, which slows down the pace of learning.

Batch normalization offers a straightforward solution to this problem: normalize the input data in such a way as to obtain zero mean and unit variance. Normalization is performed before entering each layer. During the training process, we normalize the batch samples, and during use, we normalize the statistics obtained based on the entire training set since we cannot see the test data in advance. We calculate the mean and variance for a specific batch, $B = x_1, \dots, x_m$, as follows:

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

Using these statistical characteristics, we transform the activation function in such a way that it has zero mean and unit variance throughout the whole batch:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Here, $\epsilon > 0$ is a parameter that protects us from dividing by 0 in cases where the standard deviation of the batch is very small or even equal to zero. Finally, to get the final activation function, y , we need to make sure that, during normalization, we don't lose the ability to generalize. Since we applied scaling and shifting operations to the original data, we can allow arbitrary scaling and shifting of normalized values, thereby obtaining the final activation function:

$$y_i = \gamma \hat{x}_i + \beta$$

Here, γ and β are the parameters of batch normalization that the system can be trained with (they can be optimized by the gradient descent method on the training data). This generalization also means that batch normalization can be useful when applying the input of a neural network directly.

This method, when applied to multilayer networks, almost always successfully reaches its goal—it accelerates learning. Moreover, it's an excellent regularizer, allowing us to choose the learning rate, the power of the L_2 regularizer, and the dropout. The regularization here is a consequence of the fact that the result of the network for a specific sample is no longer deterministic (it depends on the whole batch that this result was obtained from), which simplifies the generalization process.

The next important topic we'll look at is neural network initialization. This affects the convergence of the training process, training speed, and overall network performance.

Neural network initialization

The principle of choosing the initial values of weights for the layers that make up the model is very important. Setting all the weights to 0 is a severe obstacle to learning because none of the weights can be active initially. Assigning weights to the random values from the interval, [0, 1], is also usually not the best option. Actually, model performance and learning process convergence can strongly rely on correct weight initialization; however, the initial task and model complexity can also play an important role. Even if the task's solution does not assume a strong dependency on the values of the initial weights, a well-chosen method of initializing weights can significantly affect the model's ability to learn. This is because it presets the model parameters while taking the loss function into account. Let's look at two popular methods that are used to initialize weights.

Xavier initialization method

The **Xavier initialization method** is used to simplify the signal flow through the layer during both the forward pass and the backward pass of the error for the linear activation function. This method also works well for the sigmoid function, since the region where it is unsaturated also has a linear character. When calculating weights, this method relies on probability distribution (such as the uniform or the normal ones) with a variance of $\text{Var}(W) = \frac{2}{n_{in} + n_{out}}$, where n_{in} and n_{out} are the number of neurons in the previous and subsequent layers, respectively.

He initialization method

The **He initialization method** is a variation of the Xavier method that's more suitable for ReLU activation functions because it compensates for the fact that this function returns zero for half of the definition domain. This method of weight calculation relies on a probability distribution with the following variance:

$$\text{Var}(W) = \frac{2}{n_{in}}$$

There are also other methods of weight initialization. Which one you choose is usually determined by the problem being solved, the network topology, the activation functions being used, and the loss function. For example, for recursive networks, the orthogonal initialization method can be used. We'll provide a concrete programming example of neural network initialization in *Chapter 12, Exporting and Importing Models*.

In the previous sections, we looked at the basic components of artificial neural networks, which are common to almost all types of networks. In the next section, we will discuss the features of convolutional neural networks that are often used for image processing.

Delving into convolutional networks

The MLP is the most powerful feedforward neural network. It consists of several layers, where each neuron receives its copy of all the output from the previous layer of neurons. This model is ideal for certain types of tasks, for example, training on a limited number of more or less unstructured parameters.

Nevertheless, let's see what happens to the number of parameters (weights) in such a model when raw data is used as input. For example, the CIFAR-10 dataset contains $32 \times 32 \times 32$ color images, and if we consider each channel of each pixel as an independent input parameter for the MLP, each neuron in the first hidden layer adds about 3,000 new parameters to the model! With the increase in image size, the situation quickly gets out of hand, producing images that users can't use for real applications.

One popular solution is to lower the resolution of the images so that an MLP becomes applicable. Nevertheless, when we lower the resolution, we risk losing a large amount of information. It would be great if it were possible to process the information before applying a decrease in quality so that we don't cause an explosive increase in the number of model parameters. There is a very effective way to solve this problem, which is based on the convolution operation.

Convolution operator

This approach was first used for neural networks that worked with images, but it has been successfully used to solve problems from other subject areas. Let's consider using this method for image classification.

Let's assume that the image pixels that are close to each other interact more closely when forming a feature of interest for us (the feature of an object in the image) than pixels located at a considerable distance. Also, if a small trait is considered very important in the process of image classification, it does not matter in which part of the image this trait is found.

Let's have a look at the concept of a convolution operator. We have a two-dimensional image of I and a small K matrix that has dimensions of $h \times w$ (the so-called convolution kernel) constructed in such a way that it graphically encodes a feature. We compute a minimized image of $I * K$, superimposing the core to the image in all possible ways and recording the sum of the elements of the original image and the kernel:

$$(I * K)_{xy} = \sum_{i=1}^h \sum_{j=1}^w K_{ij} \times I_{x+i-1, y+j-1}$$

An exact definition assumes that the kernel matrix is transposed, but for machine learning tasks, it doesn't matter whether this operation was performed or not. The convolution operator is the basis of the convolutional layer in a CNN. The layer consists of a certain number of kernels, \vec{K} (with additive displacement components, \vec{b} , for each kernel), and calculates the convolution of the output image of the previous layer using each of the kernels, each time adding a displacement component. In the end, the activation function, σ , can be applied to the entire output image. Usually, the input stream for a convolutional layer consists of d channels—for example, red/green/blue for the input layer, in which

case the kernels are also expanded so that they also consist of d channels. The following formula is obtained for one channel of the output image of the convolutional layer, where K is the kernel and b is the stride (shift) component:

$$\text{conv}(I, K)_{x,y} = \sigma(b + \sum_{i=1}^h \sum_{j=1}^w \sum_{k=1}^d K_{ijk} \times I_{x+i-1, y+j-1, k})$$

The following diagram schematically depicts the preceding formula:

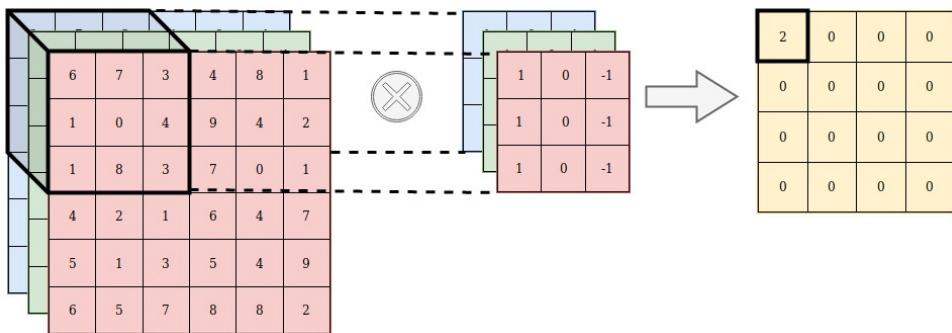


Figure 10.12 – Convolution operation scheme

If the additive (stride) component is not equal to 1, then this can be schematically depicted as follows:

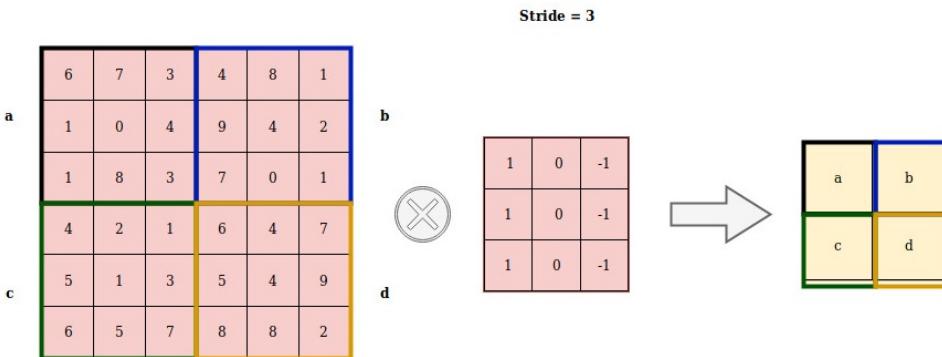


Figure 10.13 – Convolution with the stride equals one

Please note that since all we are doing here is adding and scaling the input pixels, the kernels can be obtained from the existing training sample using the gradient descent method, similar to calculating weights in an MLP. An MLP could perfectly cope with the functions of the convolutional layer, but it requires a much longer training time, as well as a more significant amount of training data.

Notice that the convolution operator is not limited to two-dimensional data: most deep learning frameworks provide layers for one-dimensional or N -dimensional convolutions directly out of the box. It is also worth noting that although the convolutional layer reduces the number of parameters compared to a fully connected layer, it uses more hyperparameters—parameters that are selected before training.

In particular, the following hyperparameters are selected:

- **Depth:** How many kernels and bias coefficients will be involved in one layer.
- The **height** and **width** of each kernel.
- **Step (stride):** How much the kernel is shifted at each step when calculating the next pixel of the resulting image. Usually, the step value that's taken is equal to 1, and the larger the value is, the smaller the size of the output image that's produced.
- **Padding:** Note that convoluting any kernel of a dimension greater than 1×1 reduces the size of the output image. Since it is generally desirable to keep the size of the original image, the pattern is supplemented with zeros along the edges.

One pass of the convolutional layer affects the image by reducing the length and width of a particular channel but increasing its value (depth).

Another way to reduce the image dimension and save its general properties is to downsample the image. Network layers that perform such operations are called **pooling layers**.

Pooling operation

A pooling layer receives small, separate fragments of the image and combines each fragment into one value. There are several possible methods of aggregation. The most straightforward one is to take the maximum from a set of pixels. This method is shown schematically in the following diagram:

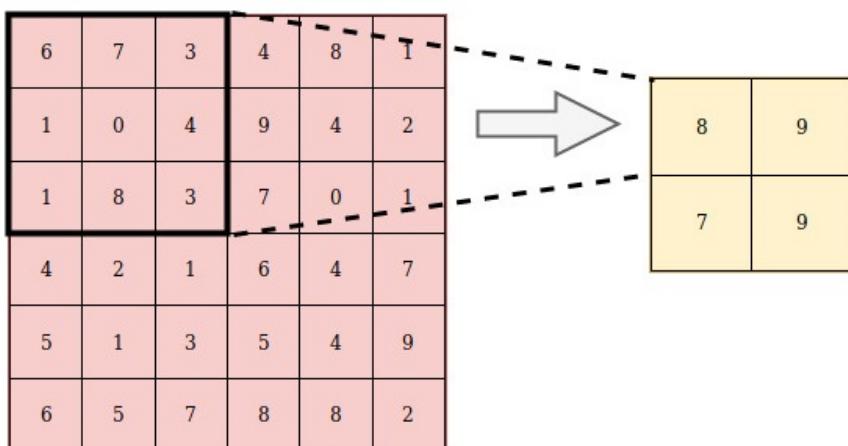


Figure 10.14 – Pooling operation

Let's consider how maximum pooling works. In the preceding diagram, we have a matrix of numbers that's 6×6 in size. The pooling window's size equals 3, so we can divide this matrix into the four smaller submatrices of size 3×3 . Then, we can choose the maximum number from each submatrix and make a smaller matrix of size 2×2 from these numbers.

The most important characteristic of a convolutional or pooling layer is its receptive field value, which allows us to understand how much information is used for processing. Let's discuss it in detail.

Receptive field

An essential component of the convolutional neural network architecture is a reduction in the amount of data from the input to the output of the model while still increasing the channel depth. As mentioned earlier, this is usually done by choosing a convolution step (stride) or pooling layers. The receptive field determines how much of the original input from the source is processed at the output. The expansion of the receptive field allows convolutional layers to combine low-level features (lines, edges) to create higher-level features (curves, textures):

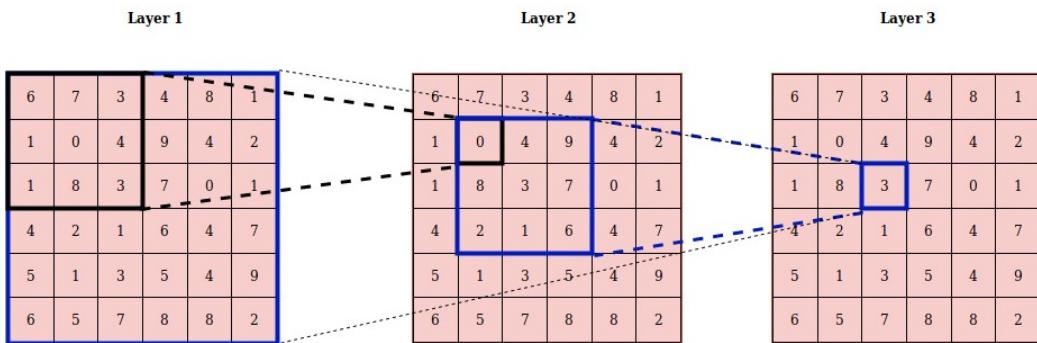


Figure 10.15 – Receptive field concept

The receptive field, l_k , of layer k can be given by the following formula:

$$l_k = l_{k-1} + ((f_k - 1) \prod_{i=1}^k s_i)$$

Here, l_{k-1} is the receptive field of the layer, $k - 1$, f_k is the filter size, and s_i is the stride of layer i . So, for the preceding example, the input layer has $RF = 1$, the hidden layer has $RF = 3$, and the last layer has $RF = 5$.

Now that we're acquainted with the basic concepts of CNNs, let's look at how we can combine them to create a concrete network architecture for image classification.

Convolution network architecture

The network is developed from a small number of low-level filters in the initial stages to a vast number of filters, each of which finds a specific high-level attribute. The transition from level to level provides a hierarchy of pattern recognition.

One of the first convolutional network architectures that was successfully applied to the pattern recognition task was the LeNet-5, which was developed by Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. It was used to recognize handwritten and printed numbers in the 1990s. The following diagram shows this architecture:

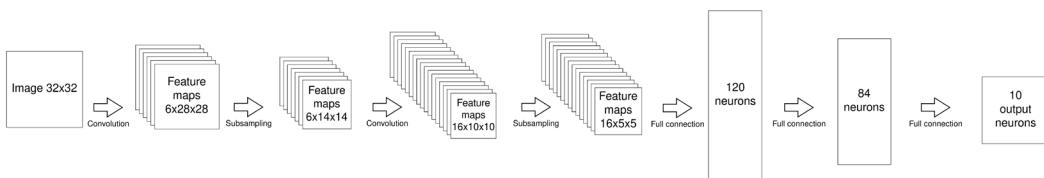


Figure 10.16 – LeNet-5 network architecture

The network layers of this architecture are explained in the following table:

Number	Layer	Feature map (depth)	Size	Kernel size	Stride	Activation
Input	Image	1	32 x 32	-	-	-
1	Convolution	6	28 x 28	5 x 5	1	tanh
2	Average pool	6	14 x 14	2 x 2	2	tanh
3	Convolution	16	10 x 10	5 x 5	1	tanh
4	Average pool	16	5 x 5	2 x 2	2	tanh
5	Convolution	120	1 x 1	5 x 5	1	tanh
6	FC		84	-	-	tanh
Output	FC		10	-	-	softmax

Table 10.1 – LeNet-5 layer properties

Notice how the depth and size of the layer are changing toward the final layer. We can see that the depth was increasing and that the size became smaller. This means that toward the final layer, the number of features the network can learn increased, but their size became smaller. Such behavior is very common among different convolutional network architectures.

In the next section, we will discuss deep learning, which is a subset of machine learning that uses artificial neural networks to learn and make decisions. It's called *deep* learning because the neural networks used have multiple layers, allowing them to model complex relationships and patterns in data.

What is deep learning?

Most often, the term deep learning is used to describe artificial neural networks that are designed to work with large amounts of data and use complex algorithms to train the model. Algorithms for deep learning can use both supervised and unsupervised algorithms (reinforcement learning). The learning process is *deep* because, over time, the neural network covers an increasing number of levels. The deeper the network is (that is, the more hidden layers, filters, and levels of feature abstraction it has), the higher the network's performance. On large datasets, deep learning shows better accuracy than traditional machine learning algorithms.

The real breakthrough that led to the current resurgence of interest in deep neural networks occurred in 2012, after the publication of the article *ImageNet classification with deep convolutional neural networks*, by *Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton* in the *Communications of the ACM* magazine. The authors have put together many different learning acceleration techniques. These techniques include convolutional neural networks, the intelligent use of GPUs, and some innovative math tricks: optimized linear neurons (ReLU) and dropout, showing that in a few weeks, they could train a complex neural network to a level that would surpass the result of traditional approaches used in computer vision.

Now, systems based on deep learning are applied in various fields and have successfully replaced the traditional approaches to machine learning. Some examples of areas where deep learning is used are as follows:

- **Speech recognition:** All major commercial speech recognition systems (such as Microsoft Cortana, Xbox, Skype Translator, Amazon Alexa, Google Now, Apple Siri, Baidu, and iFlytek) are based on deep learning.
- **Computer vision:** Today, deep learning image recognition systems are already able to give more accurate results than the human eye, for example, when analyzing medical research images (MRI, X-ray, and so on).
- **Discovery of new drugs:** For example, the AtomNet neural network was used to predict new biomolecules and was put forward for the treatment of diseases such as the Ebola virus and multiple sclerosis.
- **Recommender systems:** Today, deep learning is used to study user preferences.
- **Bioinformatics:** It is also used to study the prediction of genetic ontologies.

As we delve deeper into the realm of neural network development, we will explore how C++ libraries can be used for creating and training artificial neural network models.

Examples of using C++ libraries to create neural networks

Many machine learning libraries have an API for creating and working with neural networks. All the libraries we used in the previous chapters—`mlpack`, `Dlib`, and `Flashlight`—are supported by neural networks. But there are also specialized frameworks for neural networks; for example, one popular one is the PyTorch framework. The difference between a specialized library and a general-purpose library is that a specialized library supports more configurable options and different network types, layers, and loss functions. Also, specialized libraries usually have more modern instruments, and these instruments are introduced to their APIs more quickly.

In this section, we'll create a simple MLP for a regression task with the `mlpack`, `Dlib`, and `Flashlight` libraries. We'll also use the PyTorch C++ API to create a more advanced network—a convolutional deep neural network with the LeNet5 architecture, which we discussed earlier in the *Convolution network architecture* section. We'll use this network for image classification.

Let's learn how to use the `mlpack`, `Dlib`, and `Flashlight` libraries to create a simple MLP for a regression task. The task is the same for all series samples—MLP should learn cosine functions at limited intervals. In this book's code samples, we can find the full program for data generation and MLP training. Here, we'll discuss the essential parts of the programs that are used for the neural network's API view. Note that the activation functions we'll be using for these samples are the Tanh and ReLU functions. We've chosen them in order to achieve better convergence for this particular task.

Dlib

The `Dlib` library has an API for working with neural networks. It can also be built with Nvidia CUDA support for performance optimization. Using CUDA or OpenCL for GPUs is important if we are planning to work with a large amount of data and deep neural networks.

The approach used in the `Dlib` library for neural networks is the same as for other machine learning algorithms in this library. We should instantiate and configure an object of the required algorithm class and then use a particular trainer to train it on a dataset.

There is the `dnn_trainer` class for training neural networks in the `Dlib` library. Objects of this class should be initialized with an object of the concrete network and the object of the optimization algorithm. The most popular optimization algorithm is the stochastic gradient descent algorithm with momentum, which we discussed in the *Backpropagation method modes* section. This algorithm is implemented in the `sgd` class. Objects of the `sgd` class should be configured with the weight decay regularization and momentum parameter values. The `dnn_trainer` class has the following essential configuration methods: `set_learning_rate`, `set_mini_batch_size`, and `set_max_num_epochs`. These set the learning rate parameter value, the mini-batch size, and the maximum number of training epochs, respectively. Also, this trainer class supports dynamic learning rate change so that we can, for example, make a lower learning rate for later epochs. The learning rate shrink parameter can be configured with the `set_learning_rate_shrink_factor` method. But for the following example, we'll use the constant learning rate because, for this particular data, it gives better training results.

The next essential item for instantiating the trainer object is the neural network type object. The Dlib library uses a declarative style to define the network architecture, and for this purpose, it uses C++ templates. So, to define the neural network architecture, we should start with the network's input. In our case, this is of the `matrix<double>` type. We need to pass this as the template argument to the next layer type; in our case, this is the fully connected layer of the `fc` type. The fully connected layer type also takes the number of neurons as the template argument. To define the whole network, we should create the nested type definitions, until we reach the last layer and the loss function. In our case, this is the `loss_mean_squared` type, which implements the mean squared loss function, which is usually used for regression tasks.

The following code snippet shows the network definition with the Dlib library API:

```
using NetworkType = loss_mean_squared<fc<
    1,
    htan<fc<
        8,
        htan<fc<16,
            htan<fc<32, input<matrix<double>>>>>>>;
```

This definition can be read in the following order:

1. We started with the input layer:

```
input<matrix<double>
```

2. Then, we added the first hidden layer with 32 neurons:

```
fc<32, input<matrix<double>>
```

3. After, we added the hyperbolic tangent activation function to the first hidden layer:

```
htan<fc<32, input<matrix<double>>>
```

4. Next, we added the second hidden layer with 16 neurons and an activation function:

```
htan<fc<16, htan<fc<32, input<matrix<double>>>>>
```

5. Then, we added the third hidden layer with 8 neurons and an activation function:

```
htan<fc<8, htan<fc<16, htan<fc<32,
    input<matrix<double>>>>>>
```

6. Then, we added the last output layer with 1 neuron and without an activation function:

```
fc<1, htan<fc<8, htan<fc<16, htan<fc<32,
    input<matrix<double>>>>>>
```

7. Finally, we finished with the loss function:

```
loss_mean_squared<...>
```

The following snippet shows the complete source code example with a network definition:

```
size_t n = 10000;
...
std::vector<matrix<double>> x(n);
std::vector<float> y(n);
...
using NetworkType = loss_mean_squared<
    fc < 1, htan < fc < 8, htan < fc < 16, htan < fc < 32,
    input < matrix < double >>>>>>>;
NetworkType network;
float weight_decay = 0.0001f;
float momentum = 0.5f;
sgd solver(weight_decay, momentum);
dnn_trainer<NetworkType> trainer(network, solver);
trainer.set_learning_rate(0.01);
trainer.set_learning_rate_shrink_factor(1); // disable learning rate
                                            //changes
trainer.set_mini_batch_size(64);
trainer.set_max_num_epochs(500);
trainer.be_verbose();
trainer.train(x, y);
network.clean();

auto predictions = network(new_x);
```

Now that we've configured the trainer object, we can use the `train` method to start the actual training process. This method takes two C++ vectors as input parameters. The first one should contain training objects of the `matrix<double>` type and the second one should contain the target regression values that are `float` types. We can also call the `be_verbose` method to see the output log of the training process. After the network has been trained, we call the `clean` method to allow the network object to clear the memory from the intermediate training values and therefore reduce memory usage.

mlpack

To create the neural network with the mlpack library, we have to start by defining the architecture of the network. We use the FFN class in the mlpack library to do so, which is used for aggregating the network layers. **FFN** stands for **feedforward network**. The library has classes for creating layers:

- **Linear**: The fully connected layer with a value for the output size
- **Sigmoid**: The sigmoid activation function layer
- **Convolution**: The 2D convolutional layer
- **ReLU**: The ReLU activation function layer
- **MaxPooling**: The maximum pooling layer
- **Softmax**: The layer with the softmax activation function

There are other types of layers in the library. All of them can be added to the FFN type object to create a neural network. The first step of neural network creation is FFN object instantiation, and it can be done as follows:

```
MeanSquaredError loss;
ConstInitialization init(0.);
FFN<MeanSquaredError, ConstInitialization> model( loss, init );
```

You can see that FFN class constructor takes two arguments. The first one is a loss function object, which in our case is the `MeanSeqaredError` type object. The second one is an initialization object; we used the `ConstantInitialization` type with 0 value. There are other initialization types in the mlpack library; for example, you can use the `HeInitialization` or `GlorotInitialization` types.

Then, to add a new layer to the network, we can use the following code:

```
model.Add<Linear>( 8 );
model.Add<ReLU>();
...
```

The new object layers were added with the `Add` method, and the template parameter was used to specialize the layer type. This method takes as a variable the number of parameters, which depends on the layer type. In this example, we passed a single parameter—the output dimensions of the fully connected linear layer. The FNN object automatically configures the input dimension.

Before we will be able to train the network, we have to create an optimization method object. We can do so as follows:

```
size_t epochs = 100;
ens::MomentumSGD optimizer(
    /*stepSize=*/0.01,
```

```
/*batchSize=*/ 64,  
/*maxIterations=*/ epochs * x.n_cols,  
/*tolerance=*/1e-10,  
/*shuffle=*/false);
```

We created an object that implements stochastic gradient descent optimization with momentum. The optimizer types in the `mlpack` library take not only the optimization parameters, such as the learning rate value, but also parameters to configure the whole training cycle. We passed the learning rate, the batch size, the number of iterations, the loss value tolerance for early stopping, and the flag to shuffle the dataset as arguments. Notice that we didn't pass the training epochs number directly; instead, we calculated the `maxIteration` parameter value as a product of the epoch number and the training element number. The `MomentumSGD` class is just the template specialization of the `SGD` class with the `MomentumUpdate` policy class. So, to update the default momentum value, we have to access the particular field, as follows:

```
optimizer.UpdatePolicy().Momentum() = 0.5;
```

There are various other optimizers in the `mlpack` library that follow the same initialization scheme.

Having the network and the optimizer objects, we can train a model as follows:

```
model.Train(x, y, optimizer);
```

We passed the `x` and `y` matrices with training samples and the optimizer objects as arguments into the `Train` method of the `FFN` type object. There is no special type for a dataset in the `mlpack` library, so the raw `arma::mat` objects are used for this purpose. In the general case, the training is done silently, which is not useful during experiments. So, there are additional parameters in the `Train` method to add verbosity. Following the optimizer parameter, the `Train` method accepts a number of callbacks. For example, if we want to see the training process log with loss values in the console, we can add the `ProgressBar` object callback as follows:

```
model.Train(x, y, optimizer, ens::ProgressBar());
```

Also, we can add another type callback. In the following example, we add the early stopping callback and the callback to record the best parameter values:

```
ens::StoreBestCoordinates<arma::mat> best_params;  
model.Train(scaled_x,  
            scaled_y,  
            optimizer,  
            ens::ProgressBar(),  
            ens::EarlyStopAtMinLoss(20),  
            best_params);
```

We configured the training to stop if the loss value doesn't change for 20 batches, and to save parameters for the best loss value into the `best_params` object.

The complete source code for this example is as follows:

```

MeanSquaredError loss;
ConstInitialization init(0.);
FFN<MeanSquaredError, ConstInitialization> model( loss, init );
model.Add<Linear>(8);
model.Add<ReLU>();
model.Add<Linear>(16);
model.Add<ReLU>();
model.Add<Linear>(32);
model.Add<ReLU>();
model.Add<Linear>(1);

// Define optimizer
size_t epochs = 100;
ens::MomentumSGD optimizer(
    /*stepSize=*/0.01,
    /*batchSize=*/ 64,
    /*maxIterations=*/ epochs * x.n_cols,
    /*tolerance=*/1e-10,
    /*shuffle=*/false);

ens::StoreBestCoordinates<arma::mat> best_params;
model.Train(x, y, optimizer, ens::ProgressBar(),
            ens::EarlyStopAtMinLoss(20), best_params);

```

After we have a trained model, we can use it for prediction as follows:

```

arma::mat predictions;
model.Predict(x, predictions);

```

Here we created an output variable named `predictions` and passed it with the input variable, `x`, to the `Predict` method. The `model` object has all the latest trained weights, but we can replace them with the best weights that we saved in the `best_weights` callback as follows:

```

model.Parameters() = best_params.BestCoordinates();

```

We just replaced the current weights by using the `Parameters` method of the `model` object.

In the next sub-section, we will implement the same neural network but with the Flashlight framework.

Flashlight

To create a neural network with the Flashlight library, you have to follow the same steps as we did with `m1pack` library. The main difference is that you will need to implement the training loop yourself. This gives you more flexibility when you deal with complex architectures and training approaches. Let's start with a network definition. We create a feedforward model with fully connected linear layers as follows:

```
f1::Sequential model;
model.add(f1::View({1, 1, 1, -1}));
model.add(f1::Linear(1, 8));
model.add(f1::ReLU());
```

The `Sequential` class was used to create the network object, and then the `add` method was used to populate it with layers. We used the `Linear` and `ReLU` layers as we did in the previous example. The main difference is that the first layer we added was the `View` type object. It was needed to make the model correctly process a batch of input data. The Flashlight tensor data layout is **width, height, channels, batch (WHCN)**. So, the view shape `{1, 1, 1, -1}` means that our input data is single-channel, one-dimensional data, and the batch size should be detected automatically because we used `-1` for the last dimension.

Next, we have to define a loss function object as follows:

```
auto loss = f1::MeanSquaredError();
```

We used the MSE loss again because we are solving the same regression task. Creating an optimizer object also looks the same as for other frameworks:

```
float learning_rate = 0.01;
float momentum = 0.5;
auto sgd = f1::SGDOptimizer(model.params(), learning_rate, momentum);
```

We also used stochastic gradient descent with a momentum algorithm. Notice that the optimizer object constructor takes the model parameters as the first argument. It's a different approach from the `Dlib` and `m1pack` libraries, where the training process is mostly hidden in the top-level training API.

The approach where you pass model parameters to an optimizer is more common for frameworks where you configure the training process more precisely; you will see it in PyTorch.

Having all base blocks initialized, we can implement a training loop. Such a loop will contain the following important steps:

1. Prediction step—the forward propagation.
2. Loss value calculation.
3. Gradients calculation—the backpropagation.
4. Optimization step where the gradient values will be used.
5. Clearing gradients.

We can implement these steps as follows:

```
const int epochs = 500;
for (int epoch_i = 0; epoch_i < epochs; ++epoch_i) {
    for (auto& batch : batch_dataset) {
        // Forward propagation
        auto predicted = model(f1::input(batch[0]));
        // Calculate loss
        auto local_batch_size = batch[0].shape().dim(0);
        auto target =
            f1::reshape(batch[1], {1, 1, 1, local_batch_size});
        auto loss_value =
            loss(predicted,
                  f1::noGrad(target)); // Backward propagation
        loss_value.backward();
        // Optimization - updating parameters
        sgd.step();
        // clearing gradients
        sgd.zeroGrad();
    }
}
```

Notice that we made two loops, one over epochs and an internal one over batches, in our training dataset. In the inner loop, we used the `batch_dataset` variable; we assume that it has the `f1::BatchDataset` dataset type, so the `batch` loop variable is the `std::vector` of tensors. Usually, it will have only two tensors, for the input data and the target batch data.

We used the `f1::input` function to wrap our input batch tensor, `batch[0]`, into the Flashlight `Variable` type with disabled gradient calculations. The `Variable` type is used for the Flashlight auto-gradient mechanism. For the target batch data, `batch[1]`, we used the `f1::noGrad` function to disable gradient calculation.

Our `model` object returns a prediction tensor with a 4D shape in the WHCN format. If didn't reshape your dataset for a convenient shape, you will have to use the `f1::reshape` function for every batch as we did in this example; otherwise, you will get shape inconsistency errors in the loss value calculation.

After we calculated the loss value with the `loss` object using the predicted and target values, we calculated the gradient values. This was done with the `backward` method of the `loss_value` object, which has the `f1 : Variable` type.

Having the gradient values calculated, we used the `step` method of the `sgd` object to apply the optimization step for the network parameters. Remember that we initialized the optimization `sgd` object with the model parameters (weights). For the final step, we called the `zeroGrad` method for the optimizer to clear the network parameter gradients.

When the network (model) is trained, it's easy to use it for prediction, as follows:

```
auto predicted = model(f1::noGrad(x));
```

`x` should be your input data. Disabling the gradient calculation for the model evaluation (prediction) stage is very important because it can save a lot of computational resources and increase overall model throughput.

In the next section, we will implement a more complex neural network to solve an image classification task using the PyTorch library.

Understanding image classification using the LeNet architecture

In this section, we'll implement a CNN for image classification. We are going to use the famous dataset of handwritten digits called the **Modified National Institute of Standards and Technology (MNIST)** dataset, which can be found at <http://yann.lecun.com/exdb/mnist/>. The dataset is a standard that was proposed by the *US National Institute of Standards and Technology* to calibrate and compare image recognition methods using machine learning, primarily based on neural networks.

The creators of the dataset used a set of samples from the US Census Bureau, with some samples written by students of American universities added later. All the samples are normalized, anti-aliased grayscale images of 28 x 28 pixels. The MNIST dataset contains 60,000 images for training and 10,000 images for testing. There are four files:

- `train-images-idx3-ubyte`: Training set images
- `train-labels-idx1-ubyte`: Training set labels
- `t10k-images-idx3-ubyte`: Test set images
- `t10k-labels-idx1-ubyte`: Test set labels

The files that contain labels are in the following format:

Offset	Type	Value	Description
0	32-bit integer	0x00000801(2049)	Magic number (MSB first)
4	32-bit integer	60,000 or 10,000	Number of items
8	Unsigned char	??	Label
9	Unsigned char	??	Label
...

Table 10.2 – MNIST labels file format

The label values are from 0 to 9. The files that contain images are in the following format:

Offset	Type	Value	Description
0	32-bit integer	0x00000803(2051)	Magic number (MSB first)
0	32-bit integer	60,000 or 10,000	Number of images
0	32-bit integer	28	Number of rows
0	32-bit integer	28	Number of columns
0	Unsigned byte	??	Pixel
0	Unsigned byte	??	Pixel
...

Table 10.3 – MNIST image file format

Pixels are stored in a row-wise manner, with values in the range of [0, 255]. 0 means background (white), while 255 means foreground (black).

In this example, we are using the PyTorch deep learning framework. This framework is primarily used with the Python language. However, its core part is written in C++, and it has a well-documented and actively developed C++ client API called **LibPyTorch**. This framework is based on the linear algebra library called **ATen**, which heavily uses the Nvidia CUDA technology for performance improvement. The Python and C++ APIs are pretty much the same but have different language notations, so we can use the official Python documentation to learn how to use the framework. This documentation also contains a section stating the differences between C++ and Python APIs and specific articles about the usage of the C++ API.

The PyTorch framework is widely used for research in deep learning. As we discussed previously, the framework provides functionality for managing big datasets. It can automatically parallelize loading the data from a disk, manage pre-loaded buffers for the data to reduce memory usage, and limit expensive performance disk operations. It provides the `torch::data::Dataset` base class for the implementation of the user custom dataset. We only need to override two methods here: `get` and `size`. These methods are not virtual because we have to use the C++ template's polymorphism to inherit from this class.

Reading the training dataset

Consider the `MNISTDataset` class, which provides access to the MNIST dataset. The constructor of this class takes two parameters: one is the name of the file that contains the images, and the other is the name of the file that contains the labels. It loads whole files into its memory, which is not a best practice, but for this dataset, this approach works well because the dataset is small. For bigger datasets, we have to implement another scheme of reading data from the disk because usually, for real tasks, we are unable to load all the data into the computer's memory.

We use the OpenCV library to deal with images, so we store all the loaded images in the C++ vector of the `cv::Mat` type. Labels are stored in a vector of the `unsigned char` type. We write two additional helper functions to read images and labels from the disk: `ReadImages` and `ReadLabels`. The following snippet shows the header file for this class:

```
#include <torch/torch.h>
#include <opencv2/opencv.hpp>
#include <string>

class MNISTDataset
    : public torch::data::Dataset<MNISTDataset> {
public:
    MNISTDataset(const std::string& images_file_name,
                 const std::string& labels_file_name);
    // torch::data::Dataset implementation
    torch::data::Example<> get(size_t index) override;
    torch::optional<size_t> size() const override;

private:
    void ReadLabels(const std::string& labels_file_name);
    void ReadImages(const std::string& images_file_name);
    uint32_t rows_ = 0;
    uint32_t columns_ = 0;
    std::vector<unsigned char> labels_;
    std::vector<cv::Mat> images_;
}
```

The following snippet shows the implementation of the public interface of the class:

```
MNISTDataset::MNISTDataset(
    const std::string& images_file_name,
    const std::string& labels_file_name) {
    ReadLabels(labels_file_name);
    ReadImages(images_file_name);
}
```

We can see that the constructor passed the filenames to the corresponding loader functions. The `size` method returns the number of items that were loaded from the disk into the `labels` container:

```
torch::optional<size_t> MNISTDataset::size() const {
    return labels_.size();
}
```

The following snippet shows the `get` method's implementation:

```
torch::data::Example<> MNISTDataset::get(size_t index) {
    return {
        CvImageToTensor(images_[index]),
        torch::tensor(static_cast<int64_t>(labels_[index]),
                     torch::TensorOptions()
                         .dtype(torch::kLong)
                         .device(torch::DeviceType::CUDA)) };
}
```

The `get` method returns an object of the `torch::data::Example<>` class. In general, this type holds two values: the training sample represented with the `torch::Tensor` type and the target value, which is also represented with the `torch::Tensor` type. This method retrieves an image from the corresponding container using a given subscript, converts the image into the `torch::Tensor` type with the `CvImageToTensor` function, and uses the label value converted into the `torch::Tensor` type as a target value.

There is a set of `torch::tensor` functions that are used to convert a C++ variable into the `torch::Tensor` type. They automatically deduce the variable type and create a tensor with corresponding values. In our case, we explicitly convert the label into the `int64_t` type because the loss function we'll be using later assumes that the target values have a `torch::Long` type. Also, notice that we passed `torch::TensorOptions` as a second argument to the `torch::tensor` function. We specified the `torch` type of the tensor values and told the system to place this tensor in the GPU memory by setting the `device` argument to be equal to the `torch::DeviceType::CUDA` value and by using the `torch::TensorOptions` object. When we manually create the PyTorch tensors, we have to explicitly configure where to place them—in the CPU or in the GPU. Tensors that are placed in different types of memory can't be used together.

To convert the OpenCV image into a tensor, write the following function:

```
torch::Tensor CvImageToTensor(const cv::Mat& image) {
    assert(image.channels() == 1);
    std::vector<int64_t> dims{
        static_cast<int64_t>(1),
        static_cast<int64_t>(image.rows),
        static_cast<int64_t>(image.cols)};
    torch::Tensor tensor_image =
        torch::from_blob(image.data, torch::IntArrayRef(dims),
                        // clone is required to copy data
                        // from temporary object
                        torch::TensorOptions()
                            .dtype(torch::kFloat)
                            .requires_grad(false))
            .clone();
    return tensor_image.to(torch::DeviceType::CUDA);
}
```

The most important part of this function is the call to the `torch::from_blob` function. This function constructs the tensor from values located in memory that are referenced by the pointer that's passed as a first argument. A second argument should be a C++ vector with tensor dimension values; in our case, we specified a three-dimensional tensor with one channel and two image dimensions. The third argument is the `torch::TensorOptions` object. We specified that the data should be of the floating-point type and that it doesn't require a gradient calculation.

PyTorch uses the auto-gradient approach for model training, and it means that it doesn't construct a static network graph with pre-calculated gradient dependencies. Instead, it uses a dynamic network graph, which means that gradient flow paths for modules are connected and calculated dynamically during the backward pass of the training process. Such an architecture allows us to dynamically change the network's topology and characteristics while running the program. All the libraries we covered previously use a static network graph.

The third interesting PyTorch function that's used here is the `torch::Tensor::to` function, which allows us to move tensors from CPU memory to GPU memory and back.

Now, let's learn how to read dataset files.

Reading dataset files

We read the labels file with the `ReadLabels` function:

```
void MNISTDataset::ReadLabels(
    const std::string& labels_file_name) {
    std::ifstream labels_file(
```

```

        labels_file_name,
        std::ios::binary | std::ios::binary);
labels_file.exceptions(std::ifstream::failbit |
                      std::ifstream::badbit);
if (labels_file) {
    uint32_t magic_num = 0;
    uint32_t num_items = 0;
    if (read_header(&magic_num, labels_file) &&
        read_header(&num_items, labels_file)) {
        labels_.resize(static_cast<size_t>(num_items));
        labels_file.read(
            reinterpret_cast<char*>(labels_.data()),
            num_items);
    }
}
}

```

This function opens the file in binary mode and reads the header records, the magic number, and the number of items in the file. It also reads all the items directly to the C++ vector. The most important part is to correctly read the header records. To do this, we can use the `read_header` function:

```

template <class T>
bool read_header(T* out, std::istream& stream) {
    auto size = static_cast<std::streamsize>(sizeof(T));
    T value;
    if (!stream.read(reinterpret_cast<char*>(&value), size)) {
        return false;
    } else {
        // flip endianness
        *out = (value << 24) | ((value << 8) & 0x00FF0000) |
               ((value >> 8) & 0X0000FF00) | (value >> 24);
    }
}

```

This function reads the value from the input stream—in our case, the file stream—and flips the endianness. This function also assumes that header records are 32-bit integer values. In a different scenario, we would have to think of other ways to flip the endianness.

Reading the image file

Reading the images file is also pretty straightforward; we read the header records and sequentially read the images. From the header records, we get the total number of images in the file and the image size. Then, we define the OpenCV matrix object that has a corresponding size and type—the one-channel image with the underlying byte `CV_8UC1` type. We read images from the disk in a loop directly to the OpenCV matrix object by passing a pointer, which is returned by the `data` object variable, to the stream read function. The size of the data we need to read is determined by calling the `cv::Mat::size()` function, followed by the call to the `area` function. Then, we use the `convertTo` OpenCV function to convert an image from the `unsigned byte` type to the 32-bit floating-point type. This is important so that we have enough precision while performing math operations in the network layers. We also normalize all the data so that it's in the range [0, 1] by dividing it by 255.

We resize all the images so that they're 32 x 32 in size because the LeNet5 network architecture requires us to hold the original dimensions of the convolution filters:

```

void MNISTDataset::ReadImages(
    const std::string& images_file_name) {
    std::ifstream images_file(
        images_file_name,
        std::ios::binary | std::ios::binary);
    labels_file.exceptions(std::ifstream::failbit |
                           std::ifstream::badbit);

    if (labels_file) {
        uint32_t magic_num = 0;
        uint32_t num_items = 0;
        rows_ = 0;
        columns_ = 0;
        if (read_header(&magic_num, labels_file) &&
            read_header(&num_items, labels_file) &&
            read_header(&rows_, labels_file) &&
            read_header(&columns_, labels_file)) {
            assert(num_items == labels_.size());
            images_.resize(num_items);
            cv::Mat img(static_cast<int>(rows_),
                        static_cast<int>(columns_), CV_8UC1);
            for (uint32_t i = 0; i < num_items; ++i) {
                images_file.read(reinterpret_cast<char*>(img.data),
                                static_cast<std::streamsize>(
                                    img.size().area()));
                img.convertTo(images_[i], CV_32F);
                images_[i] /= 255; // normalize
                cv::resize(images_[i], images_[i],
                           cv::Size(32, 32)); // Resize to
            }
        }
    }
}

```

```

        // 32x32 size
    }
}
}
}
```

Now that we've loaded the training data, we have to define our neural network.

Neural network definition

In this example, we chose the LeNet5 architecture, which was developed by Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner (<http://yann.lecun.com/exdb/lenet/>). The architecture's details were discussed earlier in the *Convolution network architecture* section. Here, we'll show you how to implement it with the PyTorch framework.

All the structural parts of the neural networks in the PyTorch framework should be derived from the `torch::nn::Module` class. The following snippet shows the header file of the LeNet5 class:

```
#include <torch/torch.h>
class LeNet5Impl : public torch::nn::Module {
public:
    LeNet5Impl();
    torch::Tensor forward(torch::Tensor x);
private:
    torch::nn::Sequential conv_;
    torch::nn::Sequential full_;
};
TORCH_MODULE(LeNet5);
```

Notice that we defined the intermediate implementation class, which is called `LeNet5Impl`. This is because PyTorch uses a memory management model based on smart pointers, and all the modules should be wrapped in a special type. There is a special class called `torch::nn::ModuleHolder`, which is a wrapper around `std::shared_ptr` but also defines some additional methods for managing modules. So, if we want to follow all PyTorch conventions and use our module (network) with all PyTorch's functions without any problems, our module class definition should be as follows:

```
class Name : public torch::nn::ModuleHolder<Impl> {}
```

`Impl` is the implementation of our module, which is derived from the `torch::nn::Module` class. There is a special macro that can do this definition for us automatically; it is called `TORCH_MODULE`. We need to specify the name of our module in order to use it.

The most important function in this definition is the `forward` function. This function, in our case, takes the network's input and passes it through all the network layers until an output value is returned from this function. If we don't implement a whole network but rather *some* custom layers or *some* structural parts of a network, this function should assume we take the values from the previous layers or other parts of the network as input. Also, if we are implementing a custom module that isn't from the PyTorch standard modules, we should define the `backward` function, which should calculate gradients for our custom operations.

The next essential thing in our module definition is the usage of the `torch::nn::Sequential` class. This class is used to group sequential layers in the network and automate the process of forwarding values between them. We broke our network into two parts, one containing convolutional layers and another containing the final fully connected layers.

The PyTorch framework contains many functions for creating layers. For example, the `torch::nn::Conv2d` function created the two-dimensional convolutional layer. Another way to create a layer in PyTorch is to use the `torch::nn::Functional` function to wrap some simple function into the layer, which can then be connected with all the outputs of the previous layer. Notice that activation functions are not part of the neurons in PyTorch and should be connected as a separate layer. The following code snippet shows the definition of our network components:

```
static std::vector<int64_t> k_size = {2, 2};
static std::vector<int64_t> p_size = {0, 0};

LeNet5Impl::LeNet5Impl() {
    conv_ = torch::nn::Sequential(
        torch::nn::Conv2d(torch::nn::Conv2dOptions(1, 6, 5)),
        torch::nn::Functional(torch::tanh),
        torch::nn::Functional(
            torch::avg_pool2d,
            /*kernel_size*/
            /*kernel_size*/ torch::IntArrayRef(k_size),
            /*stride*/ torch::IntArrayRef(k_size),
            /*padding*/ torch::IntArrayRef(p_size),
            /*ceil_mode*/ false,
            /*count_include_pad*/ false),
        torch::nn::Conv2d(torch::nn::Conv2dOptions(6, 16, 5)),
        torch::nn::Functional(torch::tanh),
        torch::nn::Functional(
            torch::avg_pool2d,
            /*kernel_size*/ torch::IntArrayRef(k_size),
            /*stride*/ torch::IntArrayRef(k_size),
            /*padding*/ torch::IntArrayRef(p_size),
            /*ceil_mode*/ false,
            /*count_include_pad*/ false),
    );
}
```

```

        torch::nn::Conv2d(
            torch::nn::Conv2dOptions(16, 120, 5)),
            torch::nn::Functional(torch::tanh));
register_module("conv", conv_);
full_ = torch::nn::Sequential(
    torch::nn::Linear(torch::nn::LinearOptions(120, 84)),
    torch::nn::Functional(torch::tanh),
    torch::nn::Linear(torch::nn::LinearOptions(84, 10)));
register_module("full", full_);
}

```

Here, we initialized two `torch::nn::Sequential` modules. They take a variable number of other modules as arguments for constructors. Notice that for the initialization of the `torch::nn::Conv2d` module, we have to pass the instance of the `torch::nn::Conv2dOptions` class, which can be initialized with the number of input channels, the number of output channels, and the kernel size. We used `torch::tanh` as an activation function; notice that it is wrapped in the `torch::nn::Functional` class instance.

The average pooling function is also wrapped in the `torch::nn::Functional` class instance because it is not a layer in the PyTorch C++ API; it's a function. Also, the pooling function takes several arguments, so we bound their fixed values. When a function in PyTorch requires the values of the dimensions, it assumes that we provide an instance of the `torch::IntArrayRef` type. An object of this type behaves as a wrapper for an array with dimension values. We should be careful here because such an array should exist at the same time as the wrapper lifetime; notice that `torch::nn::Functional` stores `torch::IntArrayRef` objects internally. That is why we defined `k_size` and `p_size` as static global variables.

Also, pay attention to the `register_module` function. It associates the string name with the module and registers it in the internals of the parent module. If the module is registered in a certain way, we can use a string-based parameter search later (often used when we need to manually manage weight updates during training) and automatic module serialization.

The `torch::nn::Linear` module defines the fully connected layer and should be initialized with an instance of the `torch::nn::LinearOptions` type, which defines the number of inputs and the number of outputs, that is, a count of the layer's neurons. Notice that the last layer returns 10 values, not one label, despite us only having a single target label. This is the standard approach in classification tasks.

The following code shows the `forward` function's implementation, which performs model inference:

```

torch::Tensor LeNet5Impl::forward(at::Tensor x) {
    auto output = conv_->forward(x);
    output = output.view({x.size(0), -1});
    output = full_->forward(output);
    output = torch::log_softmax(output, -1);
}

```

```
    return output;  
}
```

This function is implemented as follows:

1. We passed the input tensor (image) to the `forward` function of the sequential convolutional group.
2. Then, we flattened its output with the `view` tensor method because fully connected layers assume that the input is flat. The `view` method takes the new dimensions for the tensor and returns a tensor view without exactly copying the data; `-1` means that we don't care about the dimension's value and that it can be flattened.
3. Then, the flattened output from the convolutional group is passed to the fully connected group.
4. Finally, we applied the softmax function to the final output. We're unable to wrap `torch::log_softmax` in the `torch::nn::Functional` class instance because of multiple overrides.

The softmax function converts a vector, \mathbf{z} , of dimension K into a vector, σ , of the same dimension, where each coordinate, σ_i , of the resulting vector is represented by a real number in the range $[0, 1]$ and the sum of the coordinates is 1.

The coordinates are calculated as follows:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{k=1}^K e^{z_k}}$$

The softmax function is used in machine learning for classification problems when the number of possible classes is more than two (for two classes, a logistic function is used). The coordinates, σ_i , of the resulting vector can be interpreted as the probabilities that the object belongs to the class, i . We chose this function because its results can be directly used for the cross-entropy loss function, which measures the difference between two probability distributions. The target distribution can be directly calculated from the target label value—we create the 10 value's vector of zeros and put one in the place indexed by the label value. Now, we have all the required components to train the neural network.

Network training

First, we should create PyTorch data loader objects for the train and test datasets. The data loader object is responsible for sampling objects from the dataset and making mini-batches from them. This object can be configured as follows:

1. First, we initialize the `MNISTDataset` type objects representing our datasets.
2. Then, we use the `torch::data::make_data_loader` function to create a data loader object. This function takes the `torch::data::DataLoaderOptions` type object with configuration settings for the data loader. We set the mini-batch size equal to 256 items and set 8 parallel data loading threads. We should also configure the sampler type, but in this case, we'll leave the default one—the random sampler.

The following snippet shows how to initialize the train and test data loaders:

```
auto train_images = root_path / "train-images-idx3-ubyte";
auto train_labels = root_path / "train-labels-idx1-ubyte";
auto test_images = root_path / "t10k-images-idx3-ubyte";
auto test_labels = root_path / "t10k-labels-idx1-ubyte";

// initialize train dataset
// -----
MNISTDataset train_dataset(train_images.native(),
                           train_labels.native());
auto train_loader = torch::data::make_data_loader(
    train_dataset.map(torch::data::transforms::Stack<>()),
    torch::data::DataLoaderOptions()
        .batch_size(256)
        .workers(8));
// initialize test dataset
// -----
MNISTDataset test_dataset(test_images.native(),
                           test_labels.native());
auto test_loader = torch::data::make_data_loader(
    test_dataset.map(torch::data::transforms::Stack<>()),
    torch::data::DataLoaderOptions()
        .batch_size(1024)
        .workers(8));
```

Notice that we didn't pass our dataset objects directly to the `torch::data::make_data_loader` function, but we applied the stacking transformation mapping to it. This transformation allows us to sample mini-batches in the form of the `torch::Tensor` object. If we skip this transformation, the mini-batches will be sampled as the C++ vector of tensors. Usually, this isn't very useful because we can't apply linear algebra operations to the whole batch in a vectorized manner.

The next step is to initialize the neural network object of the LeNet5 type, which we defined previously. We'll move it to the GPU to improve training and evaluation performance:

```
LeNet5 model;
model->to(torch::DeviceType::CUDA);
```

When the model of our neural network has been initialized, we can initialize an optimizer. We chose stochastic gradient descent with momentum optimization for this. It is implemented in the `torch::optim::SGD` class. The object of this class should be initialized with `model` (network) parameters and the `torch::optim::SGDOptions` type object. All `torch::nn::Module` type objects have the `parameters()` method, which returns the `std::vector<Tensor>` object containing all the parameters (weights) of the network. There is also the `named_parameters` method, which returns the dictionary of named parameters. Parameter names are created with the

names we used in the `register_module` function call. This method is handy if we want to filter parameters and exclude some of them from the training process.

The `torch::optim::SGDOptions` object can be configured with the values of the learning rate, the weight decay regularization factor, and the momentum value factor:

```
double learning_rate = 0.01;
double weight_decay = 0.0001; // regularization parameter
torch::optim::SGD optimizer(
    model->parameters(),
    torch::optim::SGDOptions(learning_rate)
        .weight_decay(weight_decay)
        .momentum(0.5));
```

Now that we have our initialized data loaders, the `network` object, and the `optimizer` object, we are ready to start the training cycle. The following snippet shows the training cycle's implementation:

```
int epochs = 100;
for (int epoch = 0; epoch < epochs; ++epoch) {
    model->train(); // switch to the training mode
    // Iterate the data loader to get batches from the dataset
    int batch_index = 0;
    for (auto& batch : (*train_loader)) {
        // Clear gradients
        optimizer.zero_grad();
        // Execute the model on the input data
        torch::Tensor prediction = model->forward(batch.data);
        // Compute a loss value to estimate error of our model
        // target should have size of [batch_size]
        torch::Tensor loss =
            torch::nll_loss(prediction, batch.target);
        // Compute gradients of the loss and parameters of our
        // model
        loss.backward();
        // Update the parameters based on the calculated
        // gradients.
        optimizer.step();
        // Output the loss every 10 batches.
        if (++batch_index % 10 == 0) {
            std::cout << "Epoch: " << epoch
                << " | Batch: " << batch_index
                << " | Loss: " << loss.item<float>()
                << std::endl;
        }
    }
}
```

We've made a loop that repeats the training cycle for 100 epochs. At the beginning of the training cycle, we switched our network object to training mode with `model->train()`. For one epoch, we iterate over all the mini-batches provided by the data loader object:

```
for (auto& batch : (*train_loader)) {
    ...
}
```

For every mini-batch, we did the next training steps, cleared the previous gradient values by calling the `zero_grad` method for the optimizer object, made a forward step over the network object, `model->forward(batch.data)`, and computed the loss value with the `nll_loss` function. This function computes the *negative log-likelihood* loss. It takes two parameters: the vector containing the probability that a training sample belongs to a class identified by position in the vector and the numeric class label (number). Then, we called the `backward` method of the loss tensor. It recursively computes the gradients for the overall network. Finally, we called the `step` method for the optimizer object, which updated all the parameters (weights) and their corresponding gradient values. The `step` method only updated the parameters that were used for initialization.

It's common practice to use test or validation data to check the training process after each epoch. We can do this in the following way:

```
model->eval(); // switch to the training mode
unsigned long total_correct = 0;
float avg_loss = 0.0;
for (auto& batch : (*test_loader)) {
    // Execute the model on the input data
    torch::Tensor prediction = model->forward(batch.data);
    // Compute a loss value to estimate error of our model
    torch::Tensor loss =
        torch::nll_loss(prediction, batch.target);
    avg_loss += loss.sum().item<float>();
    auto pred = std::get<1>(prediction.detach_().max(1));
    total_correct += static_cast<unsigned long>(
        pred.eq(batch.target.view_as(pred))
            .sum()
            .item<long>());
}
avg_loss /= test_dataset.size().value();
double accuracy = (static_cast<double>(total_correct) /
                    test_dataset.size().value());
std::cout << "Test Avg. Loss: " << avg_loss
        << " | "
        Accuracy : " << accuracy << std::endl;
```

First, we switched the model to evaluation mode by calling the `eval` method. Then we iterated over all the batches from the test data loader. For each of these batches, we performed a forward pass over the network, calculating the loss value in the same way that we did for our training process. To estimate the total loss (error) value for the model, we averaged the loss values for all the batches. To get the total loss for the batch, we used `loss.sum().item<float>()`. Here, we summarized the losses for each training sample in the batch and moved it to the CPU floating-point variable with the `item<float>()` method.

Next, we calculate the accuracy value. This is the ratio between correct answers and misclassified ones. Let's go through this calculation with the following approach. First, we determine the predicted class labels by using the `max` method of the tensor object:

```
auto pred = std::get<1>(prediction.detach_().max(1));
```

The `max` method returns a tuple, where the values are the maximum value of each row of the input tensor in the given dimension and the location indices of each maximum value the method found. Then, we compare the predicted labels with the target ones and calculate the number of correct answers:

```
total_correct += static_cast<unsigned long>(
    pred.eq(batch.target.view_as(pred)).sum().item<long>());
```

We used the `eq` tensor's method for our comparison. This method returns a boolean vector whose size is equal to the input vector, with values equal to 1 where the vector element components are equal and with values equal to 0 where they're not. To perform the comparison operation, we made a view for the target labels tensor with the same dimensions as the predictions tensor. The `view_as` method is used for this comparison. Then, we calculated the sum of 1 values and moved the value to the CPU variable with the `item<long>()` method.

By doing this, we can see that the specialized framework has more options we can configure and is more flexible for neural network development. It has more layer types and supports dynamic network graphs. It also has a powerful specialized linear algebra library that can be used to create new layers, as well as new loss and activation functions. It has powerful abstractions that enable us to work with big training data. One more important thing to note is that it has a C++ API very similar to the Python API, so we can easily port Python programs to C++ and vice versa.

Summary

In this chapter, we looked at what artificial neural networks are, looked at their history, and examined the reasons for their appearance, rise, and fall and why they have become one of the most actively developed machine learning approaches today. We looked at the difference between biological and artificial neurons before learning the basics of the perceptron concept, which was created by Frank Rosenblatt. Then, we discussed the internal features of artificial neurons and networks, such as activation functions and their characteristics, network topology, and convolution layer concepts. We

also learned how to train artificial neural networks with the error backpropagation method. We saw how to choose the right loss function for different types of tasks. Then, we discussed the regularization methods that are used to combat overfitting during training.

Finally, we implemented a simple MLP for a regression task with the `mlpack`, `Dlib`, and `Flashlight` C++ machine learning libraries. Then, we implemented a more advanced convolution network for an image classification task with `PyTorch`, a specialized neural network framework. This showed us the benefits of specialized frameworks over general-purpose libraries.

In the next chapter, we will discuss how to use pre-trained **large language models (LLMs)** and adapt them to our particular tasks. We will see how to use the transfer learning technique and the `BERT` network to perform sentiment analysis.

Further reading

- *Loss Functions for Deep Neural Networks in Classification*: <https://arxiv.org/pdf/1702.05659.pdf>
- *Neural Networks and Deep Learning*, by Michael Nielsen: <http://neuralnetworksanddeeplearning.com/>
- *Principles of Neurodynamics*, Rosenblatt, Frank (1962), Washington, DC: Spartan Books
- *Perceptrons*, Minsky M. L. and Papert S. A. 1969. Cambridge, MA: MIT Press
- *Neural Networks and Learning Machines*, Simon O. Haykin 2008
- *Deep Learning*, Ian Goodfellow, Yoshua Bengio, Aaron Courville 2016
- The PyTorch GitHub page: <https://github.com/pytorch/>
- The PyTorch documentation site: <https://pytorch.org/docs/>
- The LibPyTorch (C++) documentation site: <https://pytorch.org/cppdocs/>

11

Sentiment Analysis with BERT and Transfer Learning

The **Transformer architecture** is a **neural network model** that has gained significant popularity in **natural language processing (NLP)**. It was first introduced in a paper by Vaswani et al. in 2017. The main advantage of the Transformer is its ability to handle parallel processing, which makes it faster than RNNs. Another important advantage of the Transformer is its ability to handle long-range dependencies in sequences. This is achieved through the use of attention mechanisms, which allow the model to focus on specific parts of the input when generating the output.

In recent years, the Transformer has been applied to a wide range of NLP tasks, including machine translation, question-answering, and summarization. Its success can be attributed to its simplicity, scalability, and effectiveness in capturing long-term dependencies. However, like any model, the Transformer also has some limitations, such as its high computational cost and reliance on large amounts of data for training. Despite these limitations, the Transformer remains a powerful tool for NLP researchers and practitioners. One of the factors that makes it possible is its ability to use and adapt already pre-trained networks for particular tasks with a much lower amount of computational resources and training data.

There are two main approaches to transfer learning, known as **fine-tuning** and **transfer learning**. Fine-tuning is a process that further adjusts a pre-trained model to better suit a specific task or dataset. It involves unfreezing some or all of the layers in the pre-trained model and training them on the new data. The process of transfer learning typically involves taking a pre-trained model, removing the final layers that are specific to the original task, and adding new layers or modifying the existing ones to fit the new task. The parameters of the model's hidden layers are usually frozen during the training phase. This is one of the main differences from fine-tuning.

The following topics will be covered in this chapter:

- A general overview of the Transformer architecture
- A brief discussion of Transformer's main components and how they work together
- An example of how to apply the transfer learning technique to build a new model for sentiment analysis

Technical requirements

The following are the technical requirements for this chapter:

- The PyTorch library
- A modern C++ compiler with C++20 support
- The CMake build system version ≥ 3.22

The code files for this chapter can be found in the following GitHub repo: <https://github.com/PacktPublishing/Hands-On-Machine-Learning-with-C-second-edition/tree/master/Chapter11/pytorch>.

To configure the development environment, please follow the instructions described in the following document: https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition/blob/main/env_scripts/README.md.

Also, you can explore the scripts in that folder to see configuration details.

To build the example project for this chapter, you can use the following script: https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition/blob/main/build_scripts/build_ch11.sh.

An overview of the Transformer architecture

The Transformer is a type of neural network architecture that was first introduced in the paper *Attention Is All You Need* by Google researchers. It has become widely used in NLP and other domains due to its ability to process long-range dependencies and attention mechanisms. The following scheme shows the general Transformer architecture:

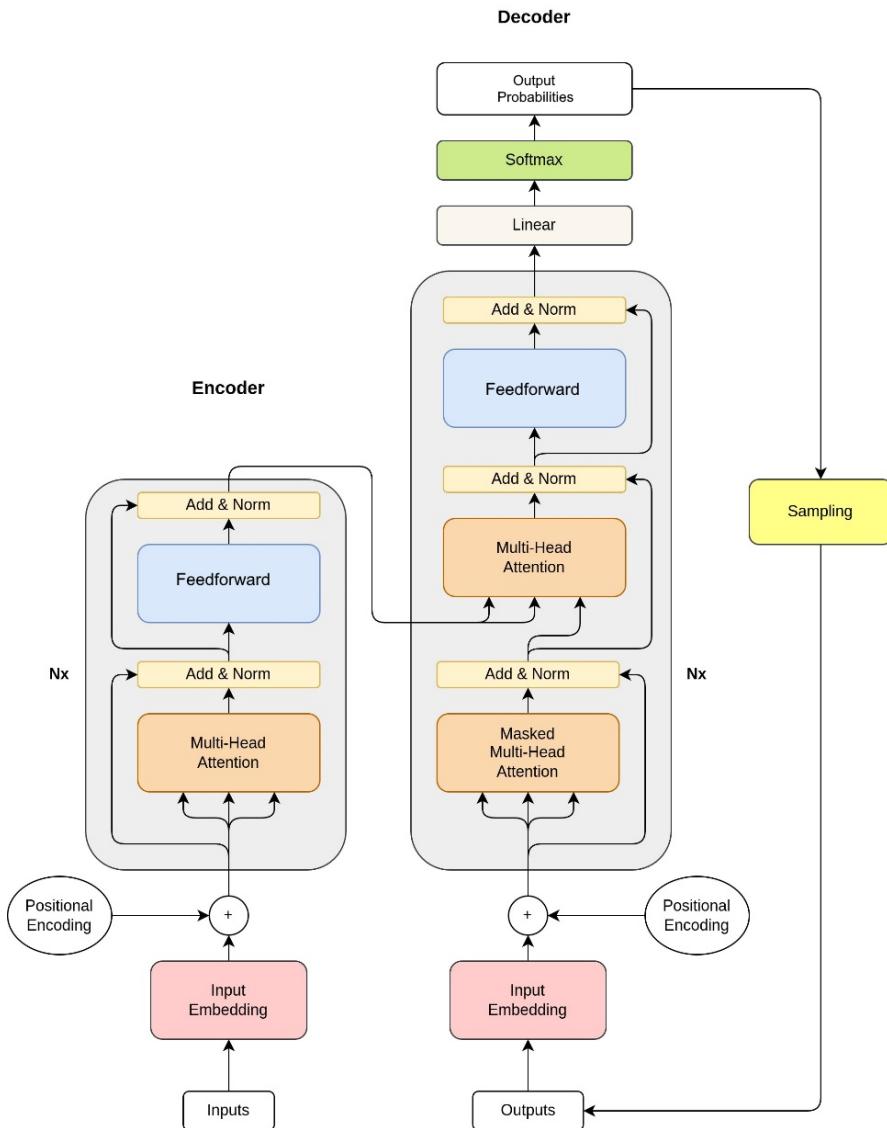


Figure 11.1 – The Transformer architecture

There are two main components of the Transformer architecture: the encoder and the decoder. The encoder processes the input sequence, while the decoder generates the output sequence. Common elements of these Transformer components are the following:

- **Self-attention:** The model uses self-attention mechanisms to learn the relationships between different parts of an input sequence. This allows it to capture long-distance dependencies, which are important for understanding context in natural languages.

- **Cross-attention:** This is a type of attention mechanism that is used when working with two or more sequences. In this case, the elements from one sequence attend to elements from another sequence, allowing the model to learn the relationships between the two inputs. It's used for communication between the encoder and decoder parts.
- **Multi-head attention:** Instead of using a single attention mechanism, the Transformer uses multiple attention heads, each with its own weights. This helps to model different aspects of the input sequence and improve the model's representational power.
- **Positional encoding:** Since the Transformer does not use recurrent or convolutional layers, it needs a way to preserve positional information about the input sequences. Positional encoding is used to add this information to the inputs. This is done by adding sine and cosine functions of different frequencies to the embedding vectors.
- **Feedforward neural networks:** Between the attention layers, there are fully connected feedforward neural networks. These networks help to transform the output of the attention layer into a more meaningful representation.
- **Residual connections and normalization:** Like many deep learning models, the Transformer includes residual connections and batch normalization to improve training stability and convergence.

Let's see in detail the main differences and tasks that are solved by the encoder and decoder parts.

Encoder

The **encoder** part of a Transformer is responsible for encoding the input data into a fixed-length vector representation, known as an embedding. This embedding captures the important features and information from the input and represents it in a more abstract form. The encoder typically consists of multiple layers of self-attention mechanisms and feedforward networks. Each layer of the encoder helps to refine and improve the representation of the input, capturing more complex patterns and dependencies.

In terms of internal representations, the encoder produces embeddings that can be either contextual or positional. Contextual embeddings focus on capturing semantic and syntactic information from the text, while positional embeddings encode information about the order and position of words in the sequence. Both types of embeddings play a role in capturing the context and relationships between words in a sentence:

- Contextual embeddings allow the model to understand the meaning of words based on their context, taking into account surrounding words and their relationships
- Positional embeddings, on the other hand, provide information about the position of each word in the sequence, helping the model understand the order and structure of the text

Together, contextual and positional embeddings in the encoder help the Transformer to better understand and represent the input data, enabling it to generate more accurate and meaningful outputs in downstream tasks.

Decoder

The **decoder** part of a Transformer takes the encoded representation as input and generates the final output. It consists of layers of attention and feedforward networks, similar to the encoder. However, it also includes additional layers that allow it to predict and generate outputs. The decoder predicts the next token in a sequence based on the encoded representation of the input sequence and its own previous predictions.

The output probabilities of the Transformer decoder represent the likelihood of each possible token being the next word in the sequence. These probabilities are calculated using a softmax function, which assigns a probability value to each token based on its relevance to the context.

During training, the decoder uses the encoded embedding to generate predictions and compare them with the true output. The difference between the predicted and true outputs is used to update the parameters of the decoder and improve its performance.

Output sampling is an essential part of the decoder process. It involves selecting the next token based on output probabilities. There are many methods for sampling the output. The following list shows some of the popular ones:

- **Greedy search:** This is the simplest method of sampling, where the most probable token at each step is selected based on the softmax probability distribution of the token values. While it is fast and easy to implement, it may not always find the optimal solution.
- **Top-k sampling:** Top-k sampling selects the top k tokens with the highest probabilities from the softmax distribution at each step, instead of selecting the most probable token. This method can help to diversify the samples and prevent the model from getting stuck in a local optimum.
- **Nucleus sampling:** Nucleus sampling, also known as top-p sampling, is a variant of top-k sampling that selects a subset of tokens from the top of the softmax distribution based on their probabilities. By selecting multiple tokens within a range of probabilities, nucleus sampling can improve the diversity and coverage of the output.

You now have an understanding of how the main Transformer components work and what elements are used within it. But this leaves the topic of how inputs are preprocessed before the decoder takes them to process uncovered. Let's see how input text can be converted into Transformer input.

Tokenization

Tokenization is the process of breaking down a sequence of text into smaller units, called tokens. These tokens can be individual words, subwords, or even characters, depending on the specific task and model architecture. For example, in the sentence *I love eating pizza*, the tokens would be *I*, *love*, *eating*, and *pizza*.

There are many tokenization methods used in Transformer models. The most used are the following:

- **Word tokenization:** This method splits the text into individual words. It is the most common approach and is suitable for tasks such as translation and text classification.
- **Subword tokenization:** In this method, the text is split into smaller units, called **subwords**. This can improve performance on tasks where words are often misspelled or truncated, such as machine translation.
- **Character tokenization:** This method breaks down the text into individual characters. It can be useful for tasks that require fine-grained analysis, such as sentiment analysis.

The choice of tokenization method depends on the characteristics of the dataset and the requirements of the task.

In the following subsection, we will use the BERT model, which uses **WordPiece tokenization**. WordPiece tokenization is designed to handle rare and out-of-vocabulary words effectively. Instead of treating each word as a separate token, it breaks down words into subword units called word pieces. This approach helps the model better handle unseen words while maintaining semantic information. In addition to word pieces, BERT also includes special tokens such as [CLS] (classify) and [SEP] (end). These tokens serve specific purposes in the model's architecture. Here's a step-by-step explanation of the WordPiece tokenization algorithm:

1. **Initial vocabulary:** Start with a small vocabulary that includes special tokens used by the model and the initial alphabet. The initial alphabet contains all the characters present at the beginning of a word and the characters inside a word preceded by the WordPiece prefix.
2. **Prefixing:** Add a prefix (such as ## for BERT) to each character within a word, resulting in a split such as w ##o ##r ##d. This creates subwords from each character in the original word.
3. **Learning merge rules:** WordPiece then learns merge rules based on frequency. Instead of simply merging the most frequent pair, it computes a score for each pair using the formula $\text{score} = (\text{freq_of_pair}) / (\text{freq_of_first_element} \times \text{freq_of_second_element})$. This prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary.
4. **Merging pairs:** The algorithm merges pairs with high scores, which means that the algorithm merges pairs that occur less frequently into individual elements.
5. **Iterative merging:** The process repeats until a desired number of merge operations has been performed or a predefined threshold is reached. At this point, the final vocabulary is created.

With the vocabulary, we can tokenize any word from the input in a similar way as we did previously, during the vocabulary construction. So, at first, we search for the whole word in the vocabulary. If we don't find it, we remove a character from the beginning preceding a subword with the ## prefix and search again. This process is continued until we find a subword. If we don't find any tokens for a word in a vocabulary, we usually skip this word.

The vocabulary is typically represented as a dictionary or lookup table that maps each word to a unique integer index. The vocabulary size is an important factor in the performance of a Transformer, as it determines the complexity of the model and its ability to handle different types of text.

Word embeddings

Even though we use tokenization to convert words into numbers, it doesn't provide any semantic meaning to the neural network. To be able to represent semantic proximity, we can use embedding. Embedding is where we map an arbitrary entity to a specific vector, for example, a node in a graph, an object in a picture, or the definition of a word. A set of embedding vectors can be treated as vector meaning space.

There are many approaches to creating embeddings for words. For example, the ones most known in classical NLP are Word2Vec and GloVe, which are based on statistical analysis and are actually standalone models.

For the Transformer, a different approach was proposed. With tokens as input, we pass them into the MLP layer, which gives embedding vectors for each of the tokens as output. This layer is trained along with the rest of the model and is an internal model component. In this way, we can have embeddings that are more fine-tuned for specific training data and particular tasks.

Using the encoder and decoder parts separately

The original Transformer architecture has both parts, the encoder and the decoder. But recently, it has been found that these parts can be used separately to solve different tasks. The two most well-known base architectures based on them are BERT and GPT.

BERT stands for **bidirectional encoder representations from transformers**. It's a state-of-the-art NLP model that uses a bidirectional approach to understand the context of words in a sentence. Unlike traditional models that only consider words in one direction, BERT can look at words both before and after a given word to better understand their meaning. It is based only on the encoder Transformer part. This makes it particularly useful for tasks that require understanding the context, such as semantic similarity and text classification. Also, it is designed to understand context in both directions, which means it can consider both the previous and following words in a sentence.

On the other hand, **GPT**, which stands for **generative pre-trained transformer**, is a generative model based only on the decoder Transformer part. It is designed to generate human-like text by predicting the next word in a sequence.

In the next section, we will develop a sentiment analysis model with the PyTorch library using BERT as the base model.

Sentiment analysis example with BERT

In this section, we are going to build a machine learning model that can detect review sentiment (detect whether a review is positive or negative) using PyTorch. As a training set, we are going to use the Large Movie Review Dataset, which contains a set of 25,000 movie reviews for training and 25,000 for testing, both of which are highly polarized.

As we said before, we will use an already pre-trained BERT model. BERT was chosen due to its ability to understand context and relationships between words, making it particularly effective for tasks such as question-answering, sentiment analysis, and text classification. Let's remember that transfer learning is a machine learning approach that involves transferring knowledge from a pre-trained model to a new or different problem domain. It is used when there is a lack of labeled data for the specific task at hand, or when training a model from scratch would be too computationally expensive.

Applying the transfer learning algorithm involves the following steps:

1. **Selection of a pre-trained model:** The model should be chosen based on relevance to the task.
2. **Adding a new task-specific head:** This could be, for example, a combination of fully connected linear layers with a final softmax for classification.
3. **Freezing the pre-trained parameters:** Freezing the parameters allows the model to retain its pre-learned knowledge.
4. **Training on the new dataset:** The model is trained using a combination of the pre-trained weights and the new data, allowing it to learn specific features and patterns relevant to the new domain in new layers.

We know that BERT-like models are used to extract some semantic knowledge from input data; in our case, it will be text. Also, BERT-like models usually represent extracted knowledge in the form of embedding vectors. These vectors can be used to train a new model head, for example, for a classification task. We will follow the previously described steps.

Exporting the model and vocabulary

The traditional method of using some pre-trained model in C++ using the PyTorch library is to load this model as a TorchScript. A common approach to getting this script is to trace a model available in Python and save it. There are a lot of pre-trained models on the <https://huggingface.co/>

site. Also, these models are available with the Python API. So, let's write a simple Python program to export the base BERT model:

1. The following code snippet shows how to import the required Python modules and load a pre-trained model for tracing:

```
import torch
from transformers import BertModel, BertTokenizer
model_name = "bert-base-cased"
tokenizer = BertTokenizer.from_pretrained(model_name,
                                           torchscript = True)
bert = BertModel.from_pretrained(model_name, torchscript=True)
```

We imported the `BertModel` and `BertTokenizer` classes from the `transformers` module, which is the library from Hugging Face that allows us to work with different Transformer-based models. We used the `bert-base-cased` model, which is the raw BERT model that was trained to understand general language semantics on large corpus of texts, and we also loaded the tokenizer module specialized for BERT models—`BertTokenizer`. Notice also that we used the `torchscript=True` parameter to be able to trace and save the model. This parameter tells the library to use operators and modules suitable for Torch JIT tracing.

2. Now that we have the loaded tokenizer object, we can tokenize some sample text for tracing as follows:

```
max_length = 128
tokenizer_out = tokenizer(text,
                           padding = "max_length",
                           max_length = max_length,
                           truncation = True,
                           return_tensors = "pt", )
attention_mask = tokenizer_out.attention_mask
input_ids = tokenizer_out.input_ids
```

Here, we defined the maximum number of tokens that can be generated, which is 128. The BERT model we loaded can process a maximum of 512 tokens at once, so you should configure this number for your task, for example, you can use a smaller number of tokens to satisfy performance restrictions on embedded devices. Also, we told the tokenizer to truncate longer sequences and pad shorter sequences to `max_length`. We also made the tokenizer return PyTorch tensors by specifying `return_tensors="pt"`.

We used two values returned from the tokenizer: `input_ids`, which is the token values, and `attention_mask`, which is a binary mask filled with 1 for real tokens and 0 for padded tokens that shouldn't be processed.

3. Now that we have tokens and masks, we can export the model as follows:

```
model.eval()
traced_script_module = torch.jit.trace(model,
                                         [ input_ids,
                                           attention_mask ])
traced_script_module.save("bert_model.pt")
```

We switched the model into evaluation mode because it will be used for tracing, not for training. Then, we used the `torch.jit.trace` function to trace the model on sample input that is the tuple of our generated tokens and attention mask. We used the `save` method of the traced module to save the model script into a file.

4. As well as the model, we also have to export the tokenizer vocabulary, as follows:

```
vocab_file = open("vocab.txt", "w")
for i, j in tokenizer.get_vocab().items():
    vocab_file.write(f"{i} {j}\n")
vocab_file.close()
```

Here, we just went through all the available tokens in the tokenizer object and saved them as [value - id] pairs listed in the text file.

Implementing the tokenizer

We can load the saved scripted model directly with the PyTorch C++ API, but we can't do the same with the tokenizer. Also, there is no tokenizer implementation in the PyTorch C++ API. So, we have to implement the tokenizer by ourselves:

1. The simplest tokenizer can actually be implemented easily. It can have the following definition for the header file:

```
#include <torch/torch.h>
#include <string>
#include <unordered_map>
class Tokenizer {
public:
    Tokenizer(const std::string& vocab_file_path,
              int max_len = 128);
    std::pair<torch::Tensor, torch::Tensor> tokenize(
        const std::string text);

private:
    std::unordered_map<std::string, int> vocab_;
    int max_len_{0};
}
```

We defined the `Tokenizer` class with a constructor that takes the name of the vocabulary file and the maximum length of the token sequence to produce. We also defined a single method, `tokenize`, that takes input text as an argument.

2. The constructor can be implemented as follows:

```
Tokenizer::Tokenizer(const std::string& vocab_file_path,
                     int max_len)
    : max_len_{max_len} {
    auto file = std::ifstream(vocab_file_path);
    std::string line;
    while (std::getline(file, line)) {
        auto sep_pos = line.find_first_of(' ');
        auto token = line.substr(0, sep_pos);
        auto id = std::stoi(line.substr(sep_pos + 1));
        vocab_.insert({token, id});
    }
}
```

We simply opened the given text file and read it line by line. We split each line into two components, which are the token string value and the corresponding ID. These components are delimited with the space character. Also, we converted `id` from a `string` to an `integer` value. All parsed token ID pairs were saved into the `std::unordered_map` container to be able to search for the token ID effectively.

3. The `tokenize` method implementation is slightly complex. We define it as follows:

```
std::pair<torch::Tensor, torch::Tensor>
Tokenizer::tokenize(const std::string text) {
    std::string pad_token = "[PAD]";
    std::string start_token = "[CLS]";
    std::string end_token = "[SEP]";
    auto pad_token_id = vocab_[pad_token];
    auto start_token_id = vocab_[start_token];
    auto end_token_id = vocab_[end_token];
```

Here, we got the special token ID values from the loaded vocabulary. These token IDs are needed by the BERT model to correctly process inputs.

4. The PAD token is used to mark empty tokens in the case when our input text is too short. It can be done as follows:

```
std::vector<int> input_ids(max_len_, pad_token_id);
std::vector<int> attention_mask(max_len_, 0);
input_ids[0] = start_token_id;
attention_mask[0] = 1;
```

As in a Python program, we created two vectors, one for token IDs and one for the attention mask. We used `pad_token_id` as the default value for token IDs and filled the attention mask with zeros. Then, we put `start_token_id` as the first element and put the corresponding value in the attention mask.

- Having defined output containers, we now define the intermediate objects for input text processing, as follows:

```
std::string word;
std::istringstream ss(text);
```

We moved our input text string into a `stringstream` object to be able to read it word by word. We also defined the corresponding word string object.

- The top-level processing cycle can be defined as follows:

```
int input_id = 1;
while (getline(ss, word, ' ')) {
    // search token in the vocabulary and increment input_id
    if (input_id == max_len_ - 1) {
        break;
    }
}
```

Here, we used the `getline` function to split an input string stream into words using the space character as a delimiter. This is a simple approach to splitting input text. Usually, tokenizers use more complex strategies for splitting. But it's enough for our task and our dataset. Also, we added the check that if we reach the maximum sequence length, we stop text processing, so we truncate it.

- Then, we have to identify the longest possible prefix in the first word that exists in the vocabulary; it can be a whole word. The implementation starts as follows:

```
size_t start = 0;
while (start < word.size()) {
    size_t end = word.size();
    std::string token;
    bool has_token = false;
    while (start < end) {
        // search the prefix in the vocabulary
        end--;
    }
    if (input_id == max_len_ - 1) {
        break;
    }
    if (!has_token) {
        break;
```

```

    }
    start = end;
}

```

Here, we defined the `start` variable to track the beginning of the prefix of a word, it's initialized with 0 which is the current word's beginning position. We defined the `end` variable to track the end of the prefix, it's initialized with the current word length as the word's last position. Initially, they pointed to the beginning and the end of the word. Then, in the internal loop, we continuously reduced the size of the prefix by decrementing the `end` variable. After each decrement, if we didn't find the prefix in the vocabulary, we repeated the process until the end of the word. Also, we made it so that after a successful token search, we swapped the `start` and `end` variables to split the word and continue prefix searches for the remaining part of the word. This was done because a word can consist of several tokens. Also, in this code, we made a check for the maximum token sequence length to stop the whole token search process.

8. The following step is where we carry out a prefix search in the vocabulary:

```

auto token = word.substr(start, end - start);
if (start > 0)
    token = "##" + token;
auto token_iter = vocab_.find(token);
if (token_iter != vocab_.end()) {
    attention_mask[input_id] = 1;
    input_ids[input_id] = token_iter->second;
    ++input_id;
    has_token = true;
    break;
}

```

Here, we extracted the prefix from the original word by using the `substr` function. If the `start` variable is not 0, we are working with the internal part of the word, so the addition of the `##` special prefix is carried out. We used the `find` method of the unordered map container to find a token (prefix). If the search was successful, we placed the token ID in the next position of the `input_ids` container, made a corresponding mark in `attention_mask`, increased the `input_id` index to move the current sequence position, and broke the loop to start working with the following word part.

9. After implementing the code for filling in the input IDs and attention mask, we put them into PyTorch tensor objects and return the output as follows:

```

attention_mask[input_id] = 1;
input_ids[input_id] = end_token_id;
auto input_ids_tensor = torch::tensor(
    input_ids).unsqueeze(0);
auto attention_masks_tensor = torch::tensor(
    attention_mask).unsqueeze(0);

```

```
    return std::make_pair(input_ids_tensor,
                          attention_masks_tensor);
```

Before converting input IDs and masks into tensors, we finalized the token ID sequence with the `end_token_id` value. Then, we used the `torch.tensor` function to create tensor objects. This function can take different inputs, and one of them is just `std::vector` with numeric values. Also, we used the `unsqueeze` function to add the batch dimension to tensors. We also returned the final tensors as a standard pair.

In the following subsection, we will implement a dataset loader class using the implemented tokenizer.

Implementing the dataset loader

We have to develop parser and data loader classes to move the dataset to memory in a format suitable for use with PyTorch:

1. Let's start with the parser. The dataset we have is organized as follows: there are two folders for the training and testing sets, and each of these folders contains two child folders, named `pos` and `neg`, which is where the positive and negative review files are placed, respectively. Each file in the dataset contains exactly one review, and its sentiment is determined by the folder it's placed in. In the following code sample, we will define the interface for the reader class:

```
#include <string>
#include <vector>
class ImdbReader {
public:
    ImdbReader(const std::string& root_path);
    size_t get_pos_size() const;
    size_t get_neg_size() const;
    const std::string& get_pos(size_t index) const;
    const std::string& get_neg(size_t index) const;
private:
    using Reviews = std::vector<std::string>;
    void read_directory(const std::string& path,
                        Reviews& reviews);
private:
    Reviews pos_samples_;
    Reviews neg_samples_;
    size_t max_size_{0};
};
```

We defined two vectors, `pos_samples_` and `neg_samples_`, which contain the reviews that were read from the corresponding folders.

2. We will assume that the object of this class should be initialized with the path to the root folder where one of the datasets is placed (the training set or testing set). We can initialize this in the following way:

```
int main(int argc, char** argv) {
    if (argc > 0) {
        auto root_path = fs::path(argv[1]);
        ... ImdbReader train_reader(root_path / "train");
        ImdbReader test_reader(root_path / "test");
    }
}
```

The most important parts of this class are the `constructor` and the `read_directory` methods.

3. The constructor is the main point wherein we fill the containers, `pos_samples_` and `neg_samples_`, with actual reviews from the pos and neg folders:

```
namespace fs = std::filesystem;
ImdbReader::ImdbReader(const std::string& root_path) {
    auto root = fs::path(root_path);
    auto neg_path = root / "neg";
    auto pos_path = root / "pos";
    if (fs::exists(neg_path) && fs::exists(pos_path)) {
        auto neg = std::async(std::launch::async, [&]() {
            read_directory(neg_path, neg_samples_);
        });
        auto pos = std::async(std::launch::async, [&]() {
            read_directory(pos_path, pos_samples_);
        });
        neg.get();
        pos.get();
    } else {
        throw std::invalid_argument("ImdbReader incorrect
                                    path");
    }
}
```

4. The `read_directory` method implements the logic for iterating files in the given directory and reads them as follows:

```
void ImdbReader::read_directory(const std::string& path,
                                Reviews& reviews) {
    for (auto& entry : fs::directory_iterator(path)) {
        if (fs::is_regular_file(entry)) {
            std::ifstream file(entry.path());
```

```

        if (file) {
            std::stringstream buffer;
            buffer << file.rdbuf();
            reviews.push_back(buffer.str());
        }
    }
}
}

```

We used the standard library directory iterator class, `fs::directory_iterator`, to get every file in the folder. The object of this class returns the object of the `fs::directory_entry` class, and this object can be used to determine whether this is a regular file with the `is_regular_file` method. We got the file path of this entry with the `path` method. We read the whole file to one string object using the `rdbuf` method of the `std::ifstream` type object.

Now that the `ImdbReader` class has been implemented, we can go further and start the dataset implementation. Our dataset class should return a pair of items: one representing the tokenized text and another the sentiment value. Also, we need to develop a custom function to convert the vector of tensors in a batch into one single tensor. This function is required if we want to make PyTorch compatible with our custom training data.

- Let's define the `ImdbSample` type for a custom training data sample. We will use this with the `torch::data::Dataset` type:

```

using ImdbData = std::pair<torch::Tensor, torch::Tensor>;
using ImdbExample = torch::data::Example<ImdbData,
                                         torch::Tensor>;

```

`ImdbData` represents the training data and has two tensors for a token sequence and an attention mask. `ImdbSample` represents the whole sample with a target value. A tensor contains 1 or 0 for positive or negative sentiment, respectively.

- The following code snippet shows the `ImdbDataset` class' declaration:

```

class ImdbDataset : public torch::data::Dataset<ImdbDataset,
                                                 ImdbExample> {
public:
    ImdbDataset(const std::string& dataset_path,
                std::shared_ptr<Tokenizer> tokenizer);
    // torch::data::Dataset implementation
    ImdbExample get(size_t index) override;
    torch::optional<size_t> size() const override;
private:
    ImdbReader reader_;
    std::shared_ptr<Tokenizer> tokenizer_;
};

```

We inherited our dataset class from the `torch::data::Dataset` class so that we can use it for data loader initialization. The PyTorch data loader object is responsible for sampling random training objects and making batches from them. The objects of our `ImdbDataset` class should be initialized with the root dataset path for the `ImdbReader` and `Tokenizer` objects. The constructor implementation is trivial; we just initialize the reader and store a pointer to the tokenizer. Notice that we used the pointer to the tokenizer to share it among the train and test datasets later. We overrode two methods from the `torch::data::Dataset` class: the `get` and `size` methods.

7. The following code shows how we implement the `size` method:

```
torch::optional<size_t> ImdbDataset::size() const {
    return reader_.get_pos_size() + reader_.get_neg_size();
}
```

The `size` method returns the number of reviews in the `ImdbReader` object.

8. The `get` method has a more complicated implementation than the previous method, as shown in the following code:

```
ImdbExample ImdbDataset::get(size_t index) {
    torch::Tensor target;
    const std::string* review=nullptr;
    if (index < reader_.get_pos_size()) {
        review = &reader_.get_pos(index);
        target = torch::tensor(1, torch::dtype(torch::kLong));
    } else {
        review =
            &reader_.get_neg(index - reader_.get_pos_size());
        target = torch::tensor(0, torch::dtype(torch::kLong));
    }
    // encode text
    auto tokenizer_out = tokenizer_->tokenize(*review);
    return {tokenizer_out, target.squeeze()};
}
```

First, we got the review text and sentiment value from the given index (the function argument value). In the `size` method, we returned the total number of positive and negative reviews, so if the input index is greater than the number of positive reviews, then this index points to a negative one. Then, we subtracted the number of positive reviews from it.

After we got the correct index, we also got the corresponding text review, assigned its address to the `review` pointer, and initialized the `target` tensor. The `torch::tensor` function was used to initialize the `target` tensor. This function takes an arbitrary numeric value and tensor options such as the required type.

With the review text, we just used the tokenizer object to create two tensors with token IDs and a sentiment mask. They are packed in the `tokenizer_out` pair object. We returned the pair of training tensors and one target tensor.

9. To be able to effectively use batched training, we create a special class so that PyTorch is able to convert our non-standard sample type into a batch tensor. In the simplest case, we will get the `std::vector` object of training samples instead of a single batch tensor. It was done as follows:

```
torch::data::transforms::Collation<ImdbExample> {
    ImdbExample apply_batch(std::vector<ImdbExample> examples)
        override {
            std::vector<torch::Tensor> input_ids;
            std::vector<torch::Tensor> attention_masks;
            std::vector<torch::Tensor> labels;
            input_ids.reserve(examples.size());
            attention_masks.reserve(examples.size());
            labels.reserve(examples.size());
            for (auto& example : examples) {
                input_ids.push_back(std::move(example.data.first));
                attention_masks.push_back(
                    std::move(example.data.second));
                labels.push_back(std::move(example.target));
            }
            return {{torch::stack(input_ids),
                     torch::stack(attention_masks)},
                     torch::stack(labels)};
        }
}
```

We inherited our class from the special PyTorch `torch::data::transforms::Collation` type and specialized it with the template parameter of our `ImdbExample` class. Having such a class, we overrode the virtual `apply_batch` function with an implementation that takes as input the `std::vector` object containing `ImdbExample` objects and returns the single `ImdbExample` object. It means that we merged all input IDs, attention masks, and target tensors in three separate tensors. This was done by creating three separate containers for input IDs, attention masks, and target tensors. They were filled in a simple loop over the input samples. Then, we just used the `torch::stack` function to merge (stack) these containers in single tensors. This class will be used in the `DataLoader` type object construction later.

Implementing the model

The next step is to create a `Model` class. We already have the exported model that we are going to use as a pre-trained part. We create a simple classification head with two linear fully connected layers and one dropout layer for regularization:

1. The header file for this class will look as follows:

```
#include <torch/script.h>
#include <torch/torch.h>
class ModelImpl : public torch::nn::Module {
public:
    ModelImpl() = delete;
    ModelImpl(const std::string& bert_model_path);
    torch::Tensor forward(at::Tensor input_ids,
                          at::Tensor attention_masks);
private:
    torch::jit::script::Module bert_;
    torch::nn::Dropout dropout_;
    torch::nn::Linear fc1_;
    torch::nn::Linear fc2_;
};
TORCH_MODULE(Model);
```

We included the `torch\script.h` header file to use the `torch::jit::script::Module` class. The instance of this class will be used as a representation of the previously exported BERT model. See the `bert_` member variable. Also, we defined member variables for the linear and dropout layers as instances of the `torch::jit::script::Module` class. We inherited our `ModelImpl` class from the `torch::nn::Module` class to integrate it into the PyTorch auto-gradient system.

2. The constructor implementation looks as follows:

```
ModelImpl::ModelImpl(const std::string& bert_model_path)
: dropout_(register_module(
    "dropout",
    torch::nn::Dropout(
        torch::nn::DropoutOptions().p(0.2)))),
fc1_(register_module(
    "fc1",
    torch::nn::Linear(
        torch::nn::LinearOptions(768, 512)))),
fc2_(register_module(
    "fc2",
    torch::nn::Linear(
```

```

        torch::nn::LinearOptions(512, 2))) {
bert_ = torch::jit::load(bert_model_path);
}

```

We used `torch::jit::load` to load the model that we exported from Python. This function takes a single argument—the model filename. Also, we initialized the dropout and linear layers and registered them in the parent `torch::nn::Module` object. The `fc1_linear` layer is the input one; it takes the 768-dimensional output of the BERT model. The `fc2_linear` layer is the output layer. It processes the internal 512-dimensional state into two classes.

3. The main model functionality is implemented in the `forward` function as follows:

```

torch::Tensor ModelImpl::forward(
    at::Tensor input_ids,
    at::Tensor attention_masks) {
std::vector<torch::jit::IValue> inputs = {
    input_ids, attention_masks};
auto bert_output = bert_.forward(inputs);
auto pooler_output =
    bert_output.toTuple()->elements()[1].toTensor();
auto x = fc1_(pooler_output);
x = torch::nn::functional::relu(x);
x = dropout_(x);
x = fc2_(x);
x = torch::softmax(x, /*dim=*/1);
return x;
}

```

This function takes the input token IDs plus the corresponding attention mask and returns a two-dimensional tensor with classification results. We interpret the sentiment analysis as a classification task. The `forward` implementation has two parts. One part is where we preprocess inputs with the loaded BERT model; this preprocessing is just an inference with pretrained BERT model backbone. The second part is where we pass the BERT output through our classification head. This is the trainable part. To use the BERT model, that is, the `torch::jit::script::Module` object, we packed inputs into the `std::vector` container of the `torch::jit::Ivalue` objects. The conversion from `torch::Tensor` was done automatically.

Then, we used standard for PyTorch `forward` function for inference. This function returns the `torch::jit` tuple object; the return type actually depends on the initial model that was traced. So, to get the PyTorch tensor from the `torch::jit` value object, we explicitly used the `toTuple` method to say how to interpret the output result. Then, we accessed the second tuple element by using the `elements` method, which provides the indexing operator for tuple elements. Finally, to get the tensor, we used the `toTensor` method of the `jit::Ivalue` object, which is a tuple element.

The BERT model we used returns two tensors. The first one represents the embedding values for input tokens, and the second one is the pooled output. Pooled output is the embeddings of the [CLS] token, for the input text. The linear layer weights that produced this output were trained from the next sentence prediction (classification) objective during BERT pre-training. So, these are the ideal values to use in the following text classification tasks. That is why we took the second element of the tuple returned by the BERT model.

The second part of the `forward` function is also simple. We passed the BERT output to the `fc1_` linear layer followed by the `relu` activation function. After this operation, we got the 512 internal hidden state. Then, this state was processed by the `dropout_` module to introduce some regularization into the model. The final stage was the use of the `fc2_` output linear module, which returns a two-dimensional vector followed by the `softmax` function. The `softmax` function converts logits into probability values with the range $[0, 1]$ because raw logits from a linear layer can have arbitrary values and need to convert them into target values.

Now, we have described all the components required for the training process. Let's see how the model training can be implemented.

Training the model

The first step in training is creating the dataset object, which can be done as follows:

```
auto tokenizer = std::make_shared<Tokenizer>(vocab_path);
ImdbDataset train_dataset(dataset_path / "train", tokenizer);
```

We created the `tokenizer` and `train_dataset` objects just by passing the paths to the corresponding files. The test dataset can be created in the same way with the same `tokenizer` object. Now that we have the dataset, we create a data loader as follows:

```
int batch_size = 8;
auto train_loader = torch::data::make_data_loader(
    train_dataset.map(Stack()),
    torch::data::DataLoaderOptions()
        .batch_size(batch_size)
        .workers(8));
```

We specified the size of the batch and used the `make_data_loader` function to create the data loader object. This object uses the dataset to effectively load and organize training samples in batches. We used the transformation `map` function of the `train_dataset` object with an instance of our collation `Stack` class to allow PyTorch to merge our training samples into tensor batches. Also, we specified some data loader options, that is, the batch size and the number of worker threads used to load and preprocess data.

We create the model object as follows:

```
torch::DeviceType device = torch::cuda::is_available()
    ? torch::DeviceType::CUDA
    : torch::DeviceType::CPU;
Model model(model_path);
model->to(device);
```

We used the `torch::cuda::is_available` function to determine whether the CUDA device is available in a system and initialized the `device` variable correspondingly. Using a CUDA device can significantly improve the training and inference of a model. The model was created with the constructor that takes the path to the exported BERT model. After model object initialization, we moved this object to the particular device.

The last component required for training is an optimizer, which we create as follows:

```
torch::optim::AdamW optimizer(model->parameters(),
    torch::optim::AdamWOptions(1e-5));
```

We used the `AdamW` optimizer, which is an improved version of the popular Adam optimizer. To construct the optimizer object, we passed the model parameters and the learning rate option as constructor arguments.

The training cycle can be defined as follows:

```
for (int epoch = 0; epoch < epochs; ++epoch) {
    model->train();
    for (auto& batch : (*train_loader)) {
        optimizer.zero_grad();
        auto batch_label = batch.target.to(device);
        auto batch_input_ids =
            batch.data.first.squeeze(1).to(device);
        auto batch_attention_mask =
            batch.data.second.squeeze(1).to(device);
        auto output =
            model(batch_input_ids, batch_attention_mask);
        torch::Tensor loss =
            torch::cross_entropy_loss(output, batch_label);
        loss.backward();
        torch::nn::utils::clip_grad_norm_(model->parameters(),
                                         1.0);
        optimizer.step();
    }
}
```

There are two nested loops. One is over the epochs and there is another, an internal one, that is over the batches. At the beginning of each internal loop is when a new epoch starts. We switched the model into training mode. This switch can be done once, but usually, you have some testing code that switches the model into evaluation mode, so this switch returns the model to the required state. Here, we omitted the test code for simplicity. It looks pretty similar to the training one, the only difference being disabling gradient calculations. In the internal loop, we used simple range-based `for` loop C++ syntax to iterate over batches. For every batch, at first, we cleared the gradient values by calling the `zero_grad` function for the optimizer object. Then, we decoupled the batch into separate tensor objects. Also, we moved these tensors to a GPU device if one is available. This was done with the `.to(device)` calls. We removed an additional dimension from the model input tensors with the `squeeze` method. This dimension appeared during an automatic batch creation.

Once all the tensors were prepared, we made a prediction with our model that gave us the `output` tensor. This output was used in the `torch::cross_entropy_loss` loss function, which is usually used for multi-class classification. It takes a tensor with probabilities for every class and the one-hot-encoded labels tensor. Then, we used the `backward` method of the `loss` tensor to calculate gradients. Also, we clipped gradients with the `clip_grad_norm_` function by setting a top value limit, to prevent them from exploding. Once the gradients were ready, we used the optimizer `step` function to update model weights according to the optimizer algorithm.

This architecture, with the settings we used, can result in more than 80% accuracy in the sentiment analysis of movie reviews in 500 training epochs.

Summary

This chapter introduced the Transformer architecture, a powerful model used in NLP and other fields of machine learning. We discussed the key components of the Transformer architecture, which include tokenization, embeddings, positional encoding, encoder, decoder, attention mechanisms, multi-head attention, cross-attention, residual connections, normalization layers, feedforward layers, and sampling techniques.

Finally, in the last part of this chapter, we developed an application so that we could perform a sentiment analysis of movie reviews. We applied the transfer learning technique to use the features learned by the pre-trained model in a new model designed for our specific task. We used the BERT model to produce a embedding representation of input texts and attached a linear layer classification head to classify review sentiments. We implemented a simple version of the tokenizer and dataset loader. We also developed the full training cycle of our classification head.

We used transfer learning instead of fine-tuning to utilize less computational resources because the fine-tuning technique usually involves re-training a full pre-trained model on a new dataset.

In the next chapter, we will discuss how to save and load model parameters. We will also look at the different APIs that exist in machine learning libraries for this purpose. Saving and loading model parameters can be quite an important part of the training process because it allows us to stop and restore training at an arbitrary moment. Also, saved model parameters can be used for evaluation purposes after the model has been trained.

Further reading

- PyTorch documentation: <https://pytorch.org/cppdocs/>
- Hugging Face BERT model documentation: https://huggingface.co/docs/transformers/model_doc/bert
- An illustrated Transformer explanation: <https://jalammar.github.io/illustrated-transformer>
- *Attention Is All You Need*, Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin: <https://arxiv.org/abs/1706.03762>
- A list of already pre-trained BERT-like models for sentiment analysis: <https://huggingface.co/models?other=sentiment-analysis>

Part 4: Production and Deployment Challenges

The crucial feature of C++ is the ability of the program to be compiled and run on a variety of hardware platforms. You can train your complex **machine learning (ML)** model on the fastest GPU in the data center and deploy it to tiny mobile devices with limited resources. This part will show you how to use C++ APIs of various ML frameworks to save and load trained models, and how to track and visualize a training process, which is crucial for ML practitioners to be able to control and check a model's training performance. Also, we will learn how to build programs that use ML models on Android devices; in particular, we will create an object detection system that uses a device's camera.

This part comprises the following chapters:

- *Chapter 12, Exporting and Importing Models*
- *Chapter 13, Tracking and Visualizing ML Experiments*
- *Chapter 14, Deploying Models on a Mobile Platform*



12

Exporting and Importing Models

In this chapter, we'll discuss how to save and load model parameters during and after training. This is important because model training can take days or even weeks. Saving intermediate results allows us to load them later for evaluation or production use.

Such regular save operations can be beneficial in the case of a random application crash. Another substantial feature of any **machine learning (ML)** framework is its ability to export the model architecture, which allows us to share models between frameworks and makes model deployment easier. The main topic of this chapter is to show how to export and import model parameters such as weights and bias values with different C++ libraries. The second part of this chapter is all about the **Open Neural Network Exchange (ONNX)** format, which is currently gaining popularity among different ML frameworks and can be used to share trained models. This format is suitable for sharing model architectures as well as model parameters.

The following topics will be covered in this chapter:

- ML model serialization APIs in C++ libraries
- Delving into the ONNX format

Technical requirements

The following are the technical requirements for this chapter:

- The `Dlib` library
- The `mlpack` library
- The `Flashlight` library
- The `pytorch` library

- The onnxruntime framework
- A modern C++ compiler with C++20 support
- CMake build system version ≥ 3.8

The code files for this chapter can be found in this book's GitHub repository: <https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition/tree/main/Chapter12>.

ML model serialization APIs in C++ libraries

In this section, we'll discuss the ML model sharing APIs that are available in the `Dlib`, `Flashlight`, `mlpack`, and `pytorch` libraries. There are three main types of sharing ML models among the different C++ libraries:

- Share model parameters (weights)
- Share the entire model's architecture
- Share both the model architecture and its trained parameters

In the following sections, we'll look at what API is available in each library and emphasize what type of sharing it supports.

Model serialization with Dlib

The `Dlib` library uses the serialization API for `decision_function` and neural network objects. Let's learn how to use it by implementing a real example.

First, we'll define the types for the neural network, regression kernel, and training sample:

```
using namespace Dlib;
using NetworkType = loss_mean_squared<fc<1, input<matrix<double>>>;
using SampleType = matrix<double, 1, 1>;
using KernelType = linear_kernel<SampleType>;
```

Then, we'll generate the training data with the following code:

```
size_t n = 1000;
std::vector<matrix<double>> x(n);
std::vector<float> y(n);
std::random_device rd;
std::mt19937 re(rd());
std::uniform_real_distribution<float> dist(-1.5, 1.5);

// generate data
```

```

for (size_t i = 0; i < n; ++i) {
    x[i](0, 0) = i;
    y[i] = func(i) + dist(re);
}

```

Here, `x` represents the predictor variable, while `y` represents the target variable. The target variable, `y`, is salted with uniform random noise to simulate real data. These variables have a linear dependency, which is defined with the following function:

```

double func(double x) {
    return 4. + 0.3 * x;
}

```

Once we've generated the data, we normalize it using the `vector_normalizer` type object. Objects of this type can be reused after training to normalize data with the learned mean and standard deviation. The following snippet shows how it's implemented:

```

vector_normalizer<matrix<double>> normalizer_x;
normalizer_x.train(x);
for (size_t i = 0; i < x.size(); ++i) {
    x[i] = normalizer_x(x[i]);
}

```

Finally, we train the `decision_function` object for kernel ridge regression with the `krr_trainer` type object:

```

void TrainAndSaveKRR(const std::vector<matrix<double>>& x,
                     const std::vector<float>& y) {
    krr_trainer<KernelType> trainer;
    trainer.set_kernel(KernelType());
    decision_function<KernelType> df = trainer.train(x, y);
    serialize("Dlib-krr.dat") << df;
}

```

Note that we initialized the trainer object with the instance of the `KernelType` object.

Now that we have the trained `decision_function` object, we can serialize it into a file with a stream object that's returned by the `serialize` function:

```

serialize("Dlib-krr.dat") << df;

```

This function takes the name of the file for storage as an input argument and returns an output stream object. We used the `<<` operator to put the learned weights of the regression model into the file. The serialization approach we used in the preceding code example only saves model parameters.

The same approach can be used to serialize almost all ML models in the Dlib library. The following code shows how to use it to serialize the parameters of a neural network:

```
void TrainAndSaveNetwork(
    const std::vector<matrix<double>>& x,
    const std::vector<float>& y) {
    NetworkType network;
    sgd solver;
    dnn_trainer<NetworkType> trainer(network, solver);
    trainer.set_learning_rate(0.0001);
    trainer.set_mini_batch_size(50);
    trainer.set_max_num_epochs(300);
    trainer.be_verbose();
    trainer.train(x, y);
    network.clean();
    serialize("Dlib-net.dat") << network;
    net_to_xml(network, "net.xml");
}
```

For neural networks, there's also the `net_to_xml` function, which saves the model structure. However, there's no function to load this saved structure into our program in the library API. It's the user's responsibility to implement a loading function.

The `net_to_xml` function exists if we wish to share the model between frameworks, as depicted in the Dlib documentation.

To check that parameter serialization works as expected, we can generate new test data to evaluate a loaded model on it:

```
std::cout << "Target values \n";
std::vector<matrix<double>> new_x(5);
for (size_t i = 0; i < 5; ++i) {
    new_x[i].set_size(1, 1);
    new_x[i](0, 0) = i;
    new_x[i] = normalizer_x(new_x[i]);
    std::cout << func(i) << std::endl;
}
```

Note that we've reused the `normalizer` object. In general, the parameters of the `normalizer` object should also be serialized and loaded because, during evaluation, we need to transform new data into the same statistical characteristics that we used for the training data.

To load a serialized object in the Dlib library, we can use the `deserialize` function. This function takes the filename and returns the input stream object:

```
void LoadAndPredictKRR(
    const std::vector<matrix<double>>& x) {
    decision_function<KernelType> df;
    deserialize("Dlib-krr.dat") >> df;
    // Predict
    std::cout << "KRR predictions \n";
    for (auto& v : x) {
        auto p = df(v);
        std::cout << static_cast<double>(p) << std::endl;
    }
}
```

As we discussed previously, in the Dlib library, serialization only stores model parameters. So, to load them, we need to use the model object with the same properties that it had before serialization was performed.

For a regression model, this means that we should instantiate a decision function object with the same kernel type.

For a neural network model, this means that we should instantiate a network object of the same type that we used for serialization, as can be seen in the following code block:

```
void LoadAndPredictNetwork(
    const std::vector<matrix<double>>& x) {
    NetworkType network;
    deserialize("Dlib-net.dat") >> network;
    // Predict
    auto predictions = network(x);
    std::cout << "Net predictions \n";
    for (auto p : predictions) {
        std::cout << static_cast<double>(p) << std::endl;
    }
}
```

In this section, we saw that the Dlib serialization API allows us to save and load ML model parameters but has limited options to serialize and load model architectures. In the next section, we'll look at the Shogun library model's serialization API.

Model serialization with Flashlight

The Flashlight library can save and load models and parameters into a binary format. It uses the Cereal C++ library internally for serialization. An example of this functionality is shown in the following example.

As in the previous example, we'll start by creating some sample training data:

```
int64_t n = 10000;
auto x = fl::randn({n});
auto y = x * 0.3f + 0.4f;

// Define dataset
std::vector<fl::Tensor> fields{x, y};
auto dataset = std::make_shared<fl::TensorDataset>(fields);
fl::BatchDataset batch_dataset(dataset, /*batch_size=*/64);
```

Here, we created a vector, `x`, with random data and used it to create our target variable, `y`, by applying a linear dependency formula. We wrapped our independent and target vectors into a `BatchDataset` object called `batch_dataset`, which we'll use to train a sample neural network.

The following code shows our neural network definition:

```
fl::Sequential model;
model.add(fl::View({1, 1, 1, -1}));
model.add(fl::Linear(1, 8));
model.add(fl::ReLU());
model.add(fl::Linear(8, 16));
model.add(fl::ReLU());
model.add(fl::Linear(16, 32));
model.add(fl::ReLU());
model.add(fl::Linear(32, 1));
```

As you can see, it's the same feedforward network that we used in the previous example, but this time for Flashlight.

The following code sample shows how to train the model:

```
auto loss = fl::MeanSquaredError();
float learning_rate = 0.01;
float momentum = 0.5;
auto sgd = fl::SGDOptimizer(model.params(),
                           learning_rate,
                           momentum);

const int epochs = 5;
for (int epoch_i = 0; epoch_i < epochs; ++epoch_i) {
```

```
for (auto& batch : batch_dataset) {  
    sgd.zeroGrad();  
    auto predicted = model(f1::input(batch[0]));  
    auto local_batch_size = batch[0].shape().dim(0);  
    auto target =  
        f1::reshape(batch[1], {1, 1, 1, local_batch_size});  
    auto loss_value = loss(predicted, f1::noGrad(target));  
    loss_value.backward();  
    sgd.step();  
}  
}
```

Here, we used the same training approach that we used previously. First, we defined the `loss` object and the `sgd` optimizer object. Then, we used the two loops over epochs and over batches to train the model. In the internal loop, we applied the model to get new predicted values from training batch data. Then, we used the `loss` object to calculate the MSE value with the batch target values. We also used the `backward` method of the loss value variable to calculate the gradients. Finally, we used the `sgd` optimizer object to update the model parameters with the `step` method.

Now that we have the trained model, we have two ways to save it in the `Flashlight` library:

1. Serialize the whole model with the architecture and weights.
2. Serialize only the model weights.

For the first option—that is, serialize the whole model with the architecture—we can do the following:

```
f1::save("model.dat", model);
```

Here, `model.dat` is the name of the file where we'll save the model. To load such a file, we can use the following code:

```
f1::Sequential model_loaded;  
f1::load("model.dat", model_loaded);
```

In this case, we created a new empty object called `model_loaded`. This new object is just the `f1::Sequential` container object without particular layers. All the layers and parameter values were loaded with the `f1::load` function. Once we've loaded the model, we can use it as follows:

```
auto predicted = model_loaded(f1::noGrad(new_x));
```

Here, `new_x` is some new data that we're using for evaluation purposes.

Such an approach when you store the whole model can be useful for applications that contain different models but have the same input and output interfaces as it can help you easily change or upgrade a model in production, for example.

The second option, which involves only saving the parameter (weight) values of a network, can be useful if we need to retrain a model regularly or if we share or reuse only some part of the model or its parameters. To do this, we can use the following code:

```
f1::save("model_params.dat", model.params());
```

Here, we used the `params` method of the `model` object to get all the model's parameters. This method returns the `std::vector` sequence of parameters for all model sub-modules. So, you can only manage some of them. To load saved parameters, we can use the following code:

```
std::vector<f1::Variable> params;
f1::load("model_params.dat", params);
for (int i = 0; i < static_cast<int>(params.size()); ++i) {
    model.setParams(params[i], i);
}
```

First, we created the empty `params` container. Then, with the `f1::load` function, we loaded parameter values into it. To be able to update particular sub-module parameter values, we used the `setParams` method. The '`setParams`' method takes a value and an integer position where we want to set this value. We saved all the model parameters so that we could take them back into the model sequentially.

Unfortunately, there's no way to load models and weights from other formats into the `Flashlight` library. So, if you need to load from another format, you have to write a converter and use the `setParams` method to set particular values. In the next section, we'll delve into the `mlpack` library's serialization API.

Model serialization with `mlpack`

The `mlpack` library only implements model parameter serialization. This serialization is based on functionality that exists in the `Armadillo` math library, which is used as the backend for `mlpack`. This means we can save parameter values in different file formats using the `mlpack` API. They are as follows:

- **CSV**: Denoted by `.csv`, or optionally `.txt`
- **ASCII**: Denoted by `.txt`
- **Armadillo ASCII**: Also denoted by `.txt`
- **PGM**: Denoted by `.pgm`
- **PPM**: Denoted by `.ppm`
- **Raw binary**: Denoted by `.bin`
- **Armadillo binary**: Also denoted by `.bin`
- **HDF5**: Denoted by `.hdf5`, `.hdf`, `.h5`, or `.he5`

Let's look at a minimal example of model creation and parameter management with mlpack. First, we need a model. The following code shows the function we can use to create one:

```
using ModelType = FFN<MeanSquaredError, ConstInitialization>;
ModelType make_model() {
    MeanSquaredError loss;
    ConstInitialization init(0.);
    ModelType model(loss, init);
    model.Add<Linear>(8);
    model.Add<ReLU>();
    model.Add<Linear>(16);
    model.Add<ReLU>();
    model.Add<Linear>(32);
    model.Add<ReLU>();
    model.Add<Linear>(1);
    return model;
}
```

The `create_model` function creates the feedforward network with several linear layers. Note that we made this model use MSE as the loss function and added the zero parameter initializer. Now that we have a model, we need some data to train it. The following code shows how to create linear dependent data:

```
size_t n = 10000;
arma::mat x = arma::randn(n).t();
arma::mat y = x * 0.3f + 0.4f;
```

Here, we created two single-dimensional vectors, similar to what we did for the `Flashlight` sample but using the Armadillo matrix API. Notice that we used the `t()` transpose method for the `x` vector since mlpack uses the column dimension for its training features.

Now, we can connect all the components and perform model training:

```
ens::Adam optimizer;
auto model = make_model();
model.Train(x, y, optimizer);
```

Here, we created the Adam algorithm optimizer object and used it in the model's `Train` method with the two data vectors we created previously. Now, we have the trained model and are ready to save its parameters. This can be done as follows:

```
data::Save("model.bin", model.Parameters(), true);
```

By default, the `data::Save` function automatically determines the file format to save with based on the provided filename extension. Here, we used the `Parameters` method of the model object to get the parameter values. This method returns a big matrix with all values. We also passed `true` as the third parameter to make the `save` function throw an exception in case of failure. By default, it will just return `false`; this is something you have to check manually.

We can use the `mlpack::data::Load` function to load parameter values, as shown here:

```
auto new_model = make_model();
data::Load("model.bin", new_model.Parameters());
```

Here, we created the `new_model` object; this is the same model but with parameters initialized as zero. Then, we used the `mlpack::data::Load` function to load parameter values from the file. Once again, we used the `Parameters` method to get the reference to the internal parameter values matrix and passed it to the `load` function. The third argument of the `load` function we set to `true` so that we can get an exception in case of errors.

Now that we've initialized the model, we can use it to make predictions:

```
arma::mat predictions;
new_model.Predict(new_x, predictions);
```

Here, we created an output matrix, `prediction`, and used the `Predict` method of the `new_model` object for model evaluation. Note that `new_x` is some new data that we wish to get predictions for.

Note that you can't load other frameworks' file formats into `mlpack`, so you have to create converters if you need them. In the next section, we'll look at the `pytorch` library's serialization API.

Model serialization with PyTorch

In this section, we'll discuss two approaches to network parameter serialization that are available in the `pytorch` C++ library:

- The `torch::save` function
- An object of the `torch::serialize::OutputArchive` type for writing parameters into the `OutputArchive` object

Let's start by preparing the neural network.

Initializing the neural network

Let's start by generating the training data. The following code snippet shows how we can do this:

```
torch::DeviceType device = torch::cuda::is_available()
? torch::DeviceType::CUDA
: torch::DeviceType::CPU;
```

Usually, we want to utilize as many hardware resources as possible. So, first, we checked whether a GPU with CUDA technology was available in the system by using the `torch::cuda::is_available()` call:

```
std::random_device rd;
std::mt19937 re(rd());
std::uniform_real_distribution<float> dist(-0.1f, 0.1f);
```

We defined the `dist` object so that we could generate the uniformly distributed real values in the -1 to 1 range:

```
size_t n = 1000;
torch::Tensor x;
torch::Tensor y;
{
    std::vector<float> values(n);
    std::iota(values.begin(), values.end(), 0);
    std::shuffle(values.begin(), values.end(), re);
    std::vector<torch::Tensor> x_vec(n);
    std::vector<torch::Tensor> y_vec(n);
    for (size_t i = 0; i < n; ++i) {
        x_vec[i] = torch::tensor(
            values[i],
            torch::dtype(torch::kFloat).device(
                device).requires_grad(false));
        y_vec[i] = torch::tensor(
            (func(values[i]) + dist(re)),
            torch::dtype(torch::kFloat).device(
                device).requires_grad(false));
    }
    x = torch::stack(x_vec);
    y = torch::stack(y_vec);
}
```

Then, we generated 1,000 predictor variable values and shuffled them. For each value, we calculated the target value with the linear function that we used in the previous examples—that is, `func`. Here's what this looks like:

```
float func(float x) {
    return 4.f + 0.3f * x;
}
```

Then, all the values were moved into the `torch::Tensor` objects with `torch::tensor` function calls. Notice that we used a previously detected device for tensor creation. Once we moved all the values to tensors, we used the `torch::stack` function to concatenate the predictor and target values in two distinct single tensors. This was required so that we could perform data normalization with the `pytorch` library's linear algebra routines:

```
auto x_mean = torch::mean(x, /*dim*/ 0);
auto x_std = torch::std(x, /*dim*/ 0);
x = (x - x_mean) / x_std;
```

Finally, we used the `torch::mean` and `torch::std` functions to calculate the mean and standard deviation of the predictor values and normalized them.

In the following code, we're defining the `NetImpl` class, which implements our neural network:

```
class NetImpl : public torch::nn::Module {
public:
    NetImpl() {
        l1_ = torch::nn::Linear(torch::nn::LinearOptions(
            1, 8).with_bias(true));
        register_module("l1", l1_);
        l2_ = torch::nn::Linear(torch::nn::LinearOptions(
            8, 4).with_bias(true));
        register_module("l2", l2_);
        l3_ = torch::nn::Linear(torch::nn::LinearOptions(
            4, 1).with_bias(true));
        register_module("l3", l3_);
        // initialize weights
        for (auto m : modules(false)) {
            if (m->name().find("Linear") != std::string::npos) {
                for (auto& p : m->named_parameters()) {
                    if (p.key().find("weight") != std::string::npos) {
                        torch::nn::init::normal_(p.value(), 0, 0.01);
                    }
                    if (p.key().find("bias") != std::string::npos) {
                        torch::nn::init::zeros_(p.value());
                    }
                }
            }
        }
    }

    torch::Tensor forward(torch::Tensor x) {
        auto y = l1_(x);
```

```
    y = 12_(y);
    y = 13_(y);
    return y;
}
private:
    torch::nn::Linear l1_{nullptr};
    torch::nn::Linear l2_{nullptr};
    torch::nn::Linear l3_{nullptr};
}
TORCH_MODULE(Net);
```

Here, we defined our neural network model as a network with three fully connected neuron layers with a linear activation function. Each layer is of the `torch::nn::Linear` type.

In the constructor of our model, we initialized all the network parameters with small random values. We did this by iterating over all the network modules (see the `modules` method call) and applying the `torch::nn::init::normal_` function to the parameters that were returned by the `named_parameters()` module's method. Biases were initialized to zeros with the `torch::nn::init::zeros_` function. The `named_parameters()` method returned objects consisting of a string name and a tensor value, so for initialization, we used its `value` method.

Now, we can train the model with our generated training data. The following code shows how we can train our model:

```
Net model;
model->to(device);
// initialize optimizer -----
double learning_rate = 0.01;
torch::optim::Adam optimizer(model->parameters(),
torch::optim::AdamOptions(learning_rate).weight_decay(0.00001));

// training
int64_t batch_size = 10;
int64_t batches_num = static_cast<int64_t>(n) / batch_size;
int epochs = 10;
for (int epoch = 0; epoch < epochs; ++epoch) {
    // train the model
    // -----
    model->train(); // switch to the training mode
    // Iterate the data
    double epoch_loss = 0;
    for (int64_t batch_index = 0; batch_index < batches_num;
        ++batch_index) {
        auto batch_x =
```

```

        x.narrow(0, batch_index * batch_size, batch_size)
            .unsqueeze(1);
    auto batch_y =
        y.narrow(0, batch_index * batch_size, batch_size)
            .unsqueeze(1);
    // Clear gradients
    optimizer.zero_grad();
    // Execute the model on the input data
    torch::Tensor prediction = model->forward(batch_x);
    torch::Tensor loss =
        torch::mse_loss(prediction, batch_y);
    // Compute gradients of the loss and parameters of
    // our model
    loss.backward();
    // Update the parameters based on the calculated
    // gradients.
    optimizer.step();
}
}

```

To utilize all our hardware resources, we moved the model to the selected computational device. Then, we initialized an optimizer. In our case, the optimizer used the Adam algorithm. After, we ran a standard training loop over the epochs where, for each epoch, we took the training batch, cleared the optimizer's gradients, performed a forward pass, computed the loss, performed a backward pass, and updated the model weights with the optimizer step.

To select a batch of training data from the dataset, we used the tensor's `narrow` method, which returned a new tensor with a reduced dimension. This function takes a new number of dimensions as the first parameter, the start position as the second parameter, and the number of elements to remain as the third parameter. We also used the `unsqueeze` method to add a batch dimension; this is required by the PyTorch API for the forward pass.

As we mentioned previously, there are two approaches we can use to serialize model parameters in pytorch in the C++ API (the Python API provides even more reach). Let's look at them.

Using the `torch::save` and `torch::load` functions

The first approach we can take to save model parameters is using the `torch:::save` function, which recursively saves parameters from the passed module:

```
torch:::save(model, "pytorch_net.pt");
```

To use it correctly with our custom modules, we need to register all the sub-modules in the parent one with the `register_module` module's method.

To load the saved parameters, we can use the `torch::load` function:

```
Net model_loaded;
torch::load(model_loaded, "pytorch_net.pt");
```

The function fills the passed module parameters with the values that are read from a file.

Using PyTorch archive objects

The second approach is to use an object of the `torch::serialize::OutputArchive` type and write the parameters we want to save into it. The following code shows how to implement the `SaveWeights` method for our model. This method writes all the parameters and buffers that exist in our module to the `archive` object, and then it uses the `save_to` method to write them in a file:

```
void NetImpl::SaveWeights(const std::string& file_name) {
    torch::serialize::OutputArchive archive;
    auto parameters = named_parameters(true /*recurse*/);
    auto buffers = named_buffers(true /*recurse*/);
    for (const auto& param : parameters) {
        if (param.value().defined()) {
            archive.write(param.key(), param.value());
        }
    }
    for (const auto& buffer : buffers) {
        if (buffer.value().defined()) {
            archive.write(buffer.key(), buffer.value(),
                         /*is_buffer*/ true);
        }
    }
    archive.save_to(file_name);
}
```

It's also important to save buffer tensors. Buffers can be retrieved from a module with the `named_buffers` module's method. These objects represent the intermediate values that are used to evaluate different modules. For example, we can be running mean and standard deviation values for the batch normalization module. In this case, we need them to continue being trained if we used serialization to save the intermediate steps and if our training process was stopped for some reason.

To load parameters that have been saved this way, we can use the `torch::serialize::InputArchive` object. The following code shows how to implement the `LoadWeights` method for our model:

```
void NetImpl::LoadWeights(const std::string& file_name) {
    torch::serialize::InputArchive archive;
    archive.load_from(file_name);
    torch::NoGradGuard no_grad;
```

```

auto parameters = named_parameters(true /*recurse*/);
auto buffers = named_buffers(true /*recurse*/);
for (auto& param : parameters) {
    archive.read(param.key(), param.value());
}
for (auto& buffer : buffers) {
    archive.read(buffer.key(), buffer.value(),
        /*is_buffer*/ true);
}
}

```

Here, the `LoadWeights` method uses the `load_from` method of the `archive` object to load parameters from the file. First, we took the parameters and buffers from our module with the `named_parameters` and `named_buffers` methods and filled in their values incrementally with the `read` method of the `archive` object.

Notice that we used an instance of the `torch::NoGradGuard` class to tell the `pytorch` library that we won't be performing any model calculation or graph-related operations. It's essential to do this because the `pytorch` library's construct calculation graph and any unrelated operations can lead to errors.

Now, we can use the new instance of our `model_loaded` model with `load` parameters to evaluate the model on some test data. Note that we need to switch the model to the evaluation model with the `eval` method. Generated test data values should also be converted into tensor objects with the `torch::tensor` function and moved to the same computational device that our model uses. The following code shows how we can implement this:

```

model_loaded->to(device);
model_loaded->eval();
std::cout << "Test:\n";
for (int i = 0; i < 5; ++i) {
    auto x_val = static_cast<float>(i) + 0.1f;
    auto tx = torch::tensor(
        x_val, torch::dtype(torch::kFloat).device(device));
    tx = (tx - x_mean) / x_std;
    auto ty = torch::tensor(
        func(x_val),
        torch::dtype(torch::kFloat).device(device));
    torch::Tensor prediction = model_loaded->forward(tx);
    std::cout << "Target:" << ty << std::endl;
    std::cout << "Prediction:" << prediction << std::endl;
}

```

In this section, we looked at two types of serialization in the `pytorch` library. The first approach involved using the `torch::save` and `torch::load` functions, which easily save and load all the model parameters, respectively. The second approach involved using objects of the `torch::serialize::InputArchive` and `torch::serialize::OutputArchive` types so that we can select what parameters we want to save and load.

In the next section, we'll discuss the ONNX file format, which allows us to share our ML model architecture and model parameters among different frameworks.

Delving into the ONNX format

The ONNX format is a special file format that's used to share neural network architectures and parameters between different frameworks. It's based on Google's Protobuf format and library. The reason why this format exists is to test and run the same neural network model in different environments and on different devices.

Usually, researchers use a programming framework that they know how to use to develop a model, and then run this model in a different environment for production purposes or if they want to share their model with other researchers or developers. This format is supported by all leading frameworks, including PyTorch, TensorFlow, MXNet, and others. However, there's a lack of support for this format from the C++ API of these frameworks and at the time of writing, they only have a Python interface for dealing with the ONNX format. Despite this, Microsoft provides the `onnxruntime` framework to run inference with this format directly with different backends, such as CUDA, CPUs, or even NVIDIA TensorRT.

Before we dive into the specifics of using the framework for our use case, it's important to consider certain limitations so that we can approach the problem statement in a well-rounded way. Sometimes, exporting to the ONNX format can be problematic due to the lack of certain operators or functions, which can limit the types of models that can be exported. Also, there can be limited support for dynamic dimensions for tensors and limited support for conditional operators, which limits our ability to use models with dynamic computational graphs and implement complex algorithms. These limitations depend on the target hardware. You'll find that embedded devices have the most restrictions and that some of these problems can only be found in the inference runtime. However, there is one big advantage of using ONNX—usually, it's possible to run such a model on a variety of different tensor math acceleration hardware.

TorchScript has fewer limitations for model operators and structure than ONNX. It's usually possible to export models with dynamic computational graphs that have been traced with all the required branches. However, there can be restrictions on hardware where you'll have to infer your model. For example, usually, it's not possible to use mobile GPUs or NPUs for inference with TorchScript. ExecuTorch should solve this problem in the future.

To utilize the available hardware as much as possible, we can use different inference engines from particular vendors. Usually, it's possible to convert a model in the ONNX format or that's using another method into its internal format to perform inference on a specific GPU or NPU. Examples of such engines include OpenVINO for Intel hardware, TensorRT from NVIDIA, ArmNN for ARM-based processors, and QNN for Qualcomm NPUs.

Now that we've understood the best way in which we can utilize the framework, let's understand how to use the ResNet neural network architecture for image classification.

Using the ResNet architecture for image classification

Generally, we, as developers, don't need to know how the ONNX format works internally because we're only interested in the files where the model has been saved. As mentioned previously, internally, the ONNX format is a Protobuf-formatted file. The following code shows the first part of the ONNX file, which describes how to use the ResNet neural network architecture for image classification:

```
ir_version: 3
graph {
    node {
        input: "data"
        input: "resnetv24_batchnorm0_gamma"
        input: "resnetv24_batchnorm0_beta"
        input: "resnetv24_batchnorm0_running_mean"
        input: "resnetv24_batchnorm0_running_var"
        output: "resnetv24_batchnorm0_fwd"
        name: "resnetv24_batchnorm0_fwd"
        op_type: "BatchNormalization"
        attribute {
            name: "epsilon"
            f: 1e-05
            type: FLOAT
        }
        attribute {
            name: "momentum"
            f: 0.9
            type: FLOAT
        }
        attribute {
            name: "spatial"
            i: 1
            type: INT
        }
    }
    node {
```

```
input: "resnetv24_batchnorm0_fwd"
input: "resnetv24_conv0_weight"
output: "resnetv24_conv0_fwd"
name: "resnetv24_conv0_fwd"
op_type: "Conv"
attribute {
    name: "dilations"
    ints: 1
    ints: 1
    type: INTS
}
attribute {
    name: "group"
    i: 1
    type: INT
}
attribute {
    name: "kernel_shape"
    ints: 7
    ints: 7
    type: INTS
}
attribute {
    name: "pads"
    ints: 3
    ints: 3
    ints: 3
    ints: 3
    type: INTS
}
attribute {
    name: "strides"
    ints: 2
    ints: 2
    type: INTS
}
...
}
```

Usually, ONNX files come in binary format to reduce file size and increase loading speed.

Now, let's learn how to use the `onnxruntime` API to load and run ONNX models. The ONNX community provides pre-trained models for the most popular neural network architectures in the publicly available Model Zoo (<https://github.com/onnx/models>).

There are a lot of ready-to-use models that can be used to solve different ML tasks. For example, we can use the ResNet-50 model for image classification tasks (<https://github.com/onnx/models/tree/main/validated/vision/classification/resnet/model/resnet50-v1-7.onnx>).

For this model, we have to download the corresponding `synset` file with image class descriptions to be able to return classification results in a human-readable manner. You can find the file at <https://github.com/onnx/models/blob/main/validated/vision/classification/synset.txt>.

To be able to use the `onnxruntime` C++ API, we have to use the following header:

```
#include <onnxruntime_cxx_api.h>
```

Then, we have to create the global shared `onnxruntime` environment and a model evaluation session, as follows:

```
Ort::Env env;
Ort::Session session(env,
                     "resnet50-v1-7.onnx",
                     Ort::SessionOptions{nullptr});
```

The `session` object takes a model's filename as its input argument and automatically loads it. Here, we passed the name of the downloaded model. The last parameter is the `SessionOptions` type object, which can be used to specify a particular device executor, such as CUDA. The `env` object holds some shared runtime state. The most valuable state is the logging data and the logging level, which can be configured with a constructor argument.

Once we've loaded a model, we can access its parameters, such as the number of model inputs, the number of model outputs, and parameter names. Such information will be very useful if you didn't know it beforehand because you need input parameter names to run inference. We can discover such model information as follows:

```
void show_model_info(const Ort::Session& session) {
    Ort::AllocatorWithDefaultOptions allocator;
```

Here, we created a function header and initialized the memory allocator for strings. Now, we can print the input parameter information:

```
    std::cout << "Input name " << i << " : " << input_name
                << std::endl;
    Ort::TypeInfo type_info = session.GetInputTypeInfo(i);
    auto tensor_info =
        type_info.GetTensorTypeAndShapeInfo();
    auto tensor_shape = tensor_info.GetShape();
    std::cout << "Input shape " << i << " : ";
    for (size_t j = 0; j < tensor_shape.size(); ++j)
        std::cout << tensor_shape[j] << " ";
    std::cout << std::endl;
}
```

Once we've discovered the input parameters, we can print the output parameter information, as follows:

```
auto num_outputs = session.GetOutputCount();
for (size_t i = 0; i < num_outputs; ++i) {
    auto output_name = session.GetOutputNameAllocated(i,
                                                       allocator);
    std::cout << "Output name " << i << " : " <<
                output_name << std::endl;
    Ort::TypeInfo type_info = session.GetOutputTypeInfo(i);
    auto tensor_info = type_info.GetTensorTypeAndShapeInfo();
    auto tensor_shape = tensor_info.GetShape();
    std::cout << "Output shape " << i << " : ";
    for (size_t j = 0; j < tensor_shape.size(); ++j)
        std::cout << tensor_shape[j] << " ";
    std::cout << std::endl;
}
}
```

Here, we used the `session` object to discover model properties. Using the `GetInputCount` and `GetOutputCount` methods, we got the number of corresponding input and output parameters. Then, we used the `GetInputNameAllocated` and `GetOutputNameAllocated` methods to get the parameter names by their indices. Notice that these methods require the `allocator` object. Here, we used the default one that was initialized at the top of the `show_model_info` function.

We can get the additional parameter type information with the `GetInputTypeInfo` and `GetOutputTypeInfo` methods by using their corresponding parameter indices. Then, by using these parameter type information objects, we can get the tensor information with the `GetTensorTypeAndShapeInfo` method. The most important piece of information here is the tensor shape that we got with the `GetShape` method of the `tensor_info` object. It's important because we need to use particular shapes for the model input and output tensors. The shape is represented as a vector of integers. Now, using the `show_model_info` function, we can get model input and output parameter information, create the corresponding tensors, and fill them with data.

In our case, the input is a tensor of size $1 \times 3 \times 224 \times 224$, which represents the RGB image for classification. The `onnxruntime` session object takes `Ort::Value` type objects as input and fills them as outputs.

The following snippet shows how to prepare the input tensor for the model:

```
constexpr const int width = 224;
constexpr const int height = 224;
std::array<int64_t, 4> input_shape{1, 3, width, height};
std::vector<float> input_image(3 * width * height);
read_image(argv[3], width, height, input_image);

auto memory_info = Ort::MemoryInfo::CreateCpu(OrtDeviceAllocator,
                                              OrtMemTypeCPU);

Ort::Value input_tensor =
    Ort::Value::CreateTensor<float>(memory_info,
                                    input_image.data(),
                                    input_image.size(),
                                    input_shape.data(),
                                    input_shape.size());
```

First, we defined constants that represent an input image's width and height. Then, we created the `input_shape` object, which defines the full shape of the tensor, including its batch dimension. With the shape, we created the `input_image` vector to hold the exact image data. This data container was filled with the `read_image` function, something we'll take a closer look at shortly. Finally, we created the `input_tensor` object with the `Ort::Value::CreateTensor` function, which takes the `memory_info` object and the references to the data and shape containers. The `memory_info` object was created with parameters to allocate the input tensor on the host CPU device. The output tensor can be created in the same way:

```
std::array<int64_t, 2> output_shape{1, 1000};
std::vector<float> result(1000);
Ort::Value output_tensor =
    Ort::Value::CreateTensor<float>(memory_info,
                                    result.data(),
                                    result.size(),
                                    output_shape.data(),
                                    output_shape.size());
```

Note that the `onnxruntime` API allows us to create an empty output tensor that will be initialized automatically. We can do this as follows:

```
Ort::Value output_tensor=nullptr;
```

Now, we can use the Run method for evaluation purposes:

```
const char* input_names[] = {"data"};
const char* output_names[] = {"resnetv17_dense0_fwd"};

Ort::RunOptions run_options;
session.Run(run_options,
            input_names,
            &input_tensor,
            1,
            output_names,
            &output_tensor,
            1);
```

Here, we defined the input and output parameters' names and constants and created the `run_options` object with default initialization. The `run_options` object can be used to log verbosity configuration, while the Run method can be used to evaluate the model. Notice that the input and output tensors were passed as pointers to arrays with corresponding element numbers. In our case, we specified single input and output elements.

The output of this model is image scores (probabilities) for each of the 1,000 classes of the ImageNet dataset, which was used to train the model. The following code shows how to decode the model's output:

```
std::map<size_t, std::string> classes = read_classes("synset.txt");
std::vector<std::pair<float, size_t>> pairs;
for (size_t i = 0; i < result.size(); i++) {
    if (result[i] > 0.01f) { // threshold check
        pairs.push_back(std::make_pair(
            output[i], i + 1)); // 0 --//background
    }
}
std::sort(pairs.begin(), pairs.end());
std::reverse(pairs.begin(), pairs.end());
pairs.resize(std::min(5UL, pairs.size()));
for (auto& p : pairs) {
    std::cout << "Class " << p.second << " Label "
        << classes.at(p.second) << " Prob " << p.first
        << std::endl;
}
```

Here, we iterated over each element of the result tensor data—that is, the `result` vector object we initialized earlier. This `result` object was filled with actual data values during model evaluation. Then, we placed the score values and class indices in the vector of corresponding pairs. This vector was sorted by score, in descending order. Then, we printed five classes with the maximum score.

In this section, we looked at an example of how to deal with the ONNX format with the `onnxruntime` framework. However, we still need to learn how to load input images into tensor objects, something we use for the model's input.

Loading images into `onnxruntime` tensors

Let's learn how to load image data according to the model's input requirements and memory layout. Previously, we initialized an `input_image` vector of the corresponding size. The model expects the input images to be normalized and three-channel RGB images whose shapes are $N \times 3 \times H \times W$, where N is the batch size and H and W are expected to be at least 224 pixels wide. Normalization assumes that the images are loaded into the $[0, 1]$ range and then normalized using means equal to $[0.485, 0.456, 0.406]$ and standard deviations equal to $[0.229, 0.224, 0.225]$.

Let's assume that we have the following function definition to load images:

```
void read_image(const std::string& file_name,
                int width,
                int height,
                std::vector<float>& image_data)
{
    ...
}
```

Let's write its implementation. To load images, we'll use the OpenCV library:

```
// load image
auto image = cv::imread(file_name, cv::IMREAD_COLOR);

if (!image.cols || !image.rows) {
    return {};
}

if (image.cols != width || image.rows != height) {
    // scale image to fit
    cv::Size scaled(
        std::max(height * image.cols / image.rows, width),
        std::max(height, width * image.rows / image.cols));
    cv::resize(image, image, scaled);
    // crop image to fit
    cv::Rect crop((image.cols - width) / 2,
                  (image.rows - height) / 2, width, height);
    image = image(crop);
}
```

Here, we read the image from a file with the `cv::imread` function. If the image dimensions aren't equal to the ones that have been specified, we need to resize the image with the `cv::resize` function and then crop the image if the image's dimensions exceed the ones that have been specified.

Then, we must convert the image into the floating-point type and RGB format:

```
image.convertTo(image, CV_32FC3);
cv::cvtColor(image, image, cv::COLOR_BGR2RGB);
```

Once formatting is complete, we can split the image into three separate channels with red, green, and blue colors. We should also normalize the color values. The following code shows how to do this:

```
std::vector<cv::Mat> channels(3);
cv::split(image, channels);

std::vector<double> mean = {0.485, 0.456, 0.406};
std::vector<double> stddev = {0.229, 0.224, 0.225};

size_t i = 0;
for (auto& c : channels) {
    c = ((c / 255) - mean[i]) / stddev[i];
    ++i;
}
```

Here, each channel was subtracted by the corresponding mean and divided by the corresponding standard deviation for the normalization process.

Then, we should concatenate the channels:

```
cv::vconcat(channels[0], channels[1], image);
cv::vconcat(image, channels[2], image);
assert(image.isContinuous());
```

In this case, the normalized channels were concatenated into one contiguous image with the `cv::vconcat` function.

The following code shows how to copy an OpenCV image into the `image_data` vector:

```
std::vector<int64_t> dims = {1, 3, height, width};
std::copy_n(reinterpret_cast<float*>(image.data),
image.size().area(),
image_data.begin());
```

Here, the image data was copied into a vector of floats, which was initialized with the specified dimensions. The OpenCV image data was accessed with the `cv::Mat::data` type member. We cast the image data into the floating-point type because this member variable is of the `unsigned char *` type. The pixel's data was copied with the standard `std::copy_n` function. This function was used to fill the `input_image` vector with actual image data. Then, the reference to the `input_image` vector data was used in the `CreateTensor` function to initialize the `Ort::Value` object.

Another important function that was used in the ONNX format example was a function that can read class definitions from a `synset` file. We'll take a look at this in the next section.

Reading the class definition file

In this example, we used the `read_classes` function to load the map of objects. Here, the key was an image class index and the value was a textual class description. This function is trivial and reads the `synset` file line by line. In such a file, each line contains a number and a class description string, separated by a space. The following code shows its definition:

```
using Classes = std::map<size_t, std::string>;
Classes read_classes(const std::string& file_name) {
    Classes classes;
    std::ifstream file(file_name);
    if (file) {
        std::string line;
        std::string id;
        std::string label;
        std::string token;
        size_t idx = 1;
        while (std::getline(file, line)) {
            std::istringstream line_stream(line);
            size_t i = 0;
            while (std::getline(line_stream, token, ' ')) {
                switch (i) {
                    case 0:
                        id = token;
                        break;
                    case 1:
                        label = token;
                        break;
                }
                token.clear();
                ++i;
            }
            classes.insert({idx, label});
            ++idx;
        }
    }
}
```

```
    }
}
return classes;
```

Notice that we used the `std::getline` function in the internal `while` loop to tokenize a single line string. We did this by specifying the third parameter that defines the delimiter character value.

In this section, we learned how to load the `synset` file, which represents the correspondence between class names and their IDs. We used this information to map a class ID that we got as a classification result to its string representation, which we showed to a user.

Summary

In this chapter, we learned how to save and load model parameters in different ML frameworks. We saw that all the frameworks we used in the `Flashlight`, `mlpack`, `Dlib`, and `pytorch` libraries have an API for model parameter serialization. Usually, these are quite simple functions that work with model objects and some input and output streams. We also discussed the serialization API, which can be used to save and load the overall model architecture. At the time of writing, some of the frameworks we used don't fully support such functionality. For example, the `Dlib` library can export neural networks in XML format but can't load them. The PyTorch C++ API lacks exporting functionality, but it can load and evaluate model architectures that have been exported from the Python API with its TorchScript functionality. However, the `pytorch` library does provide access to the library API, which allows us to load and evaluate models saved in the ONNX format from C++. However, note that you can export a model into the ONNX format from the PyTorch Python API that was previously exported into TorchScript and loaded, for example.

We also briefly looked at the ONNX format and realized that it's quite a popular format for sharing models among different ML frameworks. It supports almost all operations and objects that are used to serialize complex neural network models effectively. At the time of writing, it's supported by all popular ML frameworks, including TensorFlow, PyTorch, MXNet, and others. Also, Microsoft provides the ONNX runtime implementation, which allows us to run the ONNX model's inference without having to depend on any other frameworks.

At the end of this chapter, we developed a C++ application that can be used to run inference on the ResNet-50 model, which was trained and exported in ONNX format. This application was made with the `onnxruntime` C++ API so that we could load the model and evaluate it on the loaded image for classification.

In the next chapter, we'll discuss how to deploy ML models that have been developed with C++ libraries to mobile devices.

Further reading

- Dlib documentation: <http://Dlib.net/>
- PyTorch C++ API: <https://pytorch.org/cppdocs/>
- ONNX official page: <https://onnx.ai/>
- ONNX Model Zoo: <https://github.com/onnx/models>
- ONNX ResNet models for image classification: <https://github.com/onnx/models/blob/main/validated/vision/classification/resnet>
- onnxruntime C++ examples: https://github.com/microsoft/onnxruntime-inference-examples/tree/main/c_cxx
- Flashlight documentation: <https://fl.readthedocs.io/en/stable/index.html>
- mlpack documentation: <https://rcppmlpack.github.io/mlpack-doxygen/>

13

Tracking and Visualizing ML Experiments

In the world of **machine learning (ML)**, **visualization** and **experiment tracking systems** play a crucial role. These tools provide a way to understand complex data, track experiments, and make informed decisions about model development.

In ML, visualizing data is essential for understanding patterns, relationships, and trends. Data visualization tools allow engineers to create charts, graphs, and plots that help them explore and analyze their data. With the right visualization tool, engineers can quickly identify patterns and anomalies, which can be used to improve model performance.

Experiment tracking systems are designed to keep track of the progress of multiple experiments. They allow engineers to compare results, identify best practices, and avoid repeating mistakes. Experiment tracking tools also help with reproducibility, ensuring that experiments can be repeated accurately and efficiently.

Choosing the right tools for visualization and experiment tracking is critical. There are many open source and commercial options available, each with its strengths and weaknesses. It's important to consider factors such as ease of use, integration with other tools, and the specific needs of your project when selecting a tool.

In this chapter, we'll briefly discuss **TensorBoard**, one of the most widespread experiment visualization systems available. We'll also learn what type of visualizations it can provide and the challenges of using it with C++. As for the tracking system, we'll discuss the **MLflow framework** and provide a hands-on example of how to use it with C++. This example covers setting up a project, defining experiments, logging metrics, and visualizing the training process and showcases the power of experiment tracking tools in enhancing the ML development process.

By the end of this chapter, you should have a clear understanding of why these tools are essential for ML engineers and how they can help you achieve better results.

This chapter covers the following topics:

- Understanding visualization and experiment tracking systems
- Experiment tracking with MLflow's REST API

Technical requirements

The following are the technical requirements for this chapter:

- Flashlight library 0.4.0
- MLflow 2.5.0
- `cpp-httplib` v0.16.0
- `nlohmann json` v3.11.2
- A modern C++ compiler with C++20 support
- CMake build system version \geq 3.22

The code files for this chapter can be found in this book's GitHub repository: <https://github.com/PacktPublishing/Hands-On-Machine-Learning-with-C-second-edition/tree/master/Chapter13/flashlight>

Understanding visualization and tracking systems for experiments

Visualization and tracking systems for ML experiments are essential components of the ML development process. Together, these systems enable engineers to build more robust and effective ML models. They also help ensure reproducibility and transparency in the development process, which is crucial for scientific rigor and collaboration.

Visualization tools provide a graphical representation of data, allowing engineers to see patterns, trends, and relationships that might be difficult to detect in raw data. This can help engineers gain insights into the behavior of their models, identify areas for improvement, and make informed decisions about model design and hyperparameter tuning.

Experiment tracking systems allow engineers to log and organize experiments, including model architectures, hyperparameters, and training data. These systems provide an overview of the entire experimentation process, making it easier to compare different models and determine which ones perform best.

Next, we'll look at some of the key features of TensorBoard, a powerful visualization tool, and understand the essential components of MLflow, an effective experiment tracking system.

TensorBoard

TensorBoard is a visualization tool for ML models that provides insights into model performance and training progress. It also provides an interactive dashboard where users can explore graphs, histograms, scatter plots, and other visualizations related to their experiments.

Here are some key features of TensorBoard:

- **Visualization of metrics and scalars:** TensorBoard allows users to visualize various metrics related to their ML experiments. These metrics include `loss`, `accuracy`, `precision`, `recall`, and `F1 score`.
- **Histogram plots:** TensorBoard also provides histogram plots for a better understanding of model performance. These plots can help users understand the distribution of layer weight and gradient values.
- **Graphs:** Graphs in TensorBoard provide a visual representation of model architecture. Users can create graphs to analyze the correlation between inputs and outputs or to compare different models.
- **Images:** TensorBoard allows you to display image data and connect such a visualization to a training timeline. This can help users analyze input data, intermediate outputs, or convolutional filter result visualizations.
- **Embedding projector:** The embedding projector in TensorBoard allows users to explore high-dimensional data in lower dimensions using techniques such as **principal component analysis (PCA)**. This feature helps in visualizing complex datasets.
- **Comparison:** In TensorBoard, comparison enables users to compare the performance of multiple models side by side, making it easy to identify the best-performing model.

Unfortunately, TensorBoard doesn't integrate easily with C++ ML frameworks. The native C++ support only exists in the TensorFlow framework. Also, there's only one third-party open source library that allows us to use TensorBoard, and it isn't actively maintained.

TensorBoard can be integrated with various Python-based deep learning frameworks, including TensorFlow, PyTorch, and others. So, if you train your models in Python, it makes sense to consider it as an instrument that can help you understand how models are performing, identify potential issues, and make informed decisions about hyperparameters, data preprocessing, and model design.

Otherwise, to visualize training data, you can use `gnuplot`-based libraries such as `CppPlot`, as we did in the previous chapters. See *Chapter 3* for the 2D scatter and line plot visualization examples.

MLflow

MLflow is an open source framework that's designed for **machine learning operations (MLOps)** and helps teams manage, track, and scale their ML projects. It provides a set of tools and features for building, training, and deploying models, as well as for monitoring their performance and experimentation.

The main components of MLflow are as follows:

- **Experiment tracking:** MLflow allows users to track their experiments, including hyperparameters, code versions, and metrics. This helps in understanding the impact of different configurations on model performance.
- **Code reproducibility:** With MLflow, users can easily reproduce their experiments by tracking code versions and dependencies. This ensures consistency across experiments and makes it easier to identify issues.
- **Model Registry:** MLflow provides a Model Registry component where users can store, version, and manage their models. This allows for easy collaboration and model sharing within teams.
- **Integration with other tools:** MLflow integrates with popular data science and ML tools, such as Jupyter Notebook, TensorFlow, PyTorch, and more. This enables seamless integration with existing workflows. For non-Python environments, MLflow provides the REST API.
- **Deployment options:** MLflow offers various options for deploying models, including Docker containers, Kubernetes, and cloud platforms. This flexibility allows users to choose the best deployment strategy based on their needs.

Internally, MLflow uses a database to store metadata about experiments, models, and parameters. By default, it uses SQLite, but other databases such as PostgreSQL and MySQL are also supported. This allows for scalability and flexibility in terms of storage requirements. MLflow uses unique identifiers to track objects and operations within the platform. These identifiers are used to link different components of an experiment together, such as a run and its associated parameters. This makes it easy to reproduce experiments and understand the relationships between different parts of a workflow. It also provides a REST API for programmatic access to features such as model registration, tracking, and model life cycle management. It uses YAML configuration files for customizing and configuring MLflow behavior, and Python APIs for easy integration with MLflow components and workflows.

So, we can summarize that visualization and experiment tracking systems are essential tools for data scientists and engineers to understand, analyze, and optimize their ML models. These systems allow users to track the performance of different models, compare results, identify patterns, and make informed decisions about model development and deployment.

To illustrate how experiment tracking tools can be integrated into ML workflows, we'll provide a concrete example in the following section.

Experiment tracking with MLflow's REST API

Let's consider an example of an experiment involving a regression model. We'll use MLflow to log performance metrics and the parameters of a model for several experiments. While training the model, we'll visualize the results using a plot to show the accuracy and loss curves over time. Finally, we'll compare the results of different experiments using the tracking system so that we can select the best-performing model and optimize it further.

This example will demonstrate how experiment tracking can be seamlessly integrated into a C++ ML workflow, providing valuable insights and improving the overall quality of research.

Before you can use MLflow, you need to install it. You can install MLflow using pip:

```
pip install mlflow
```

Then, you'll need to start a server, like so:

```
mlflow server --backend-store-uri file:///samples/Chapter13/mlruns
```

This command starts the local tracking server at `http://localhost:5000`, which saves tracking data to the `/samples/Chapter13/mlruns` directory. If you need to access the MLflow server from remote machines, you can start the command with the `--host` and `--port` arguments.

Having started tracking the server, we can communicate with it using the REST API. The access point to this API is hosted at `http://localhost:5000/api/2.0/mlflow/`. MLflow uses JSON as its data representation for the REST API.

To implement a REST client for communicating with the tracking server, we'll use two additional libraries:

- `cpp-httplib`: For implementing HTTP communication
- `nlohmann json`: For implementing REST requests and responses

Note that the basic linear regression model will be implemented using the `Flashlight` library.

Next, we'll learn how to connect all these pieces. The first part we'll cover is implementing the REST client.

Implementing MLflow's REST C++ client

There are two main concepts in MLflow: **experiments** and **runs**. Together, they provide a structured approach to managing and tracking ML workflows. They help us organize our projects, ensure reproducibility, and facilitate collaboration among team members.

In MLflow, we can organize and track our ML experiments. An experiment can be thought of as a container for all the runs related to a specific project or goal. It allows you to keep track of different versions of your models, compare their performance, and identify the best one.

The following are the key features of an experiment:

- **Name:** Each experiment has a unique name that identifies it.
- **Tags:** You can add tags to an experiment to categorize it based on different criteria.
- **Artifacts location:** Artifacts are files that are generated throughout the experiment, such as images, logs, and more. MLflow allows you to store and version these artifacts.

A run represents a single execution of an experiment or a specific task within an experiment. Runs are used to record the details of each execution, such as its start time, end time, parameters, and metrics.

The key features of a run are as follows:

- **Start time:** The time when the run started
- **End time:** The time when the run finished
- **Parameters:** Model parameters such as batch size, learning rate, and more
- **Model:** The model code that was executed during the run
- **Output:** The results that were produced by the run, including metrics, artifacts, and more

Within a run, users can log parameters, metrics, and model representation.

Now that we understand the main concepts of MLflow's tracking structure, let's implement the MLflow REST client:

1. First, we're going to put all the implementation details for the REST client in a single `MLFlow` class. The header file should look as follows:

```
class MLFlow {  
public:  
    MLFlow(const std::string& host, size_t port);  
    void set_experiment(const std::string& name);  
    void start_run();  
    void end_run();  
    void log_metric(const std::string& name, float value,  
                    size_t epoch);  
    void log_param(const std::string& name,  
                  const std::string& value);  
    template <typename T>  
    void log_param(const std::string& name, T value) {  
        log_param(name, std::to_string(value));  
    }  
  
private:  
    httpplib::Client http_client_;
```

```
    std::string experiment_id_;
    std::string run_id_;
}
```

We made the constructor take a host and the port of the tracking server to communicate with. Then, we defined methods to start a named experiment and run inside it, as well as methods to log named metrics and parameters. After, we declared an instance of the `httpclient`:`Client` class, which will be used for HTTP communication with the tracking server. Finally, we provided member variables, which are the IDs of the current experiment and run.

2. Now, let's learn how to implement these methods. The constructor implementation is as follows:

```
MLFlow::MLFlow(const std::string& host, size_t port)
    : http_client_(host, port) {
}
```

Here, we initialized the `httpclient`:`Client` instance with the host and port values to initialize a connection with a tracking server.

3. The following code shows the `set_experiment` method's implementation:

```
void MLFlow::set_experiment(const std::string& name) {
    auto res = http_client_.Get(
        "/api/2.0/mlflow/experiments/"
        "get-by-name?experiment_name=" +
        name);
    if (check_result(res, 404)) {
        // Create a new experiment
        nlohmann::json request;
        request["name"] = name;
        res = http_client_.Post(
            "/api/2.0/mlflow/experiments/create",
            request.dump(), "application/json");
        handle_result(res);
        // Remember experiment ID
        auto json = nlohmann::json::parse(res->body);
        experiment_id_ =
            json["experiment_id"].get<std::string>();
    } else if (check_result(res, 200)) {
        // Remember experiment ID
        auto json = nlohmann::json::parse(res->body);
        experiment_id_ = json["experiment"]["experiment_id"]
            .get<std::string>();
    } else {
        handle_result(res);
    }
}
```

This method initializes an experiment for the following runs. There are two parts to this method—one for a new experiment and another for the existing one:

- I. First, we checked whether the experiment with the given name existed on the server using the following code:

```
auto res = http_client_.Get(
    "/api/2.0/mlflow/experiments/get-byname?experiment_name=" +
    name)
```

By comparing the results with the 404 and 202 codes, we identified that there's no such experiment or that it already exists.

- II. Because there's no existing experiment, we created a JSON-based request to create a new experiment, as follows:

```
nlohmann::json request;
request["name"] = name;
```

- III. Then, we passed it as the body of the HTTP request to the server, as follows:

```
res = http_client_.Post("/api/2.0/mlflow/experiments/create",
                       request.dump(), "application/json");
handle_result(res);
```

- IV. The dump method of the nlohmann::json object was used to convert the JSON into a string representation. After we got the result, we used the handle_result function to check for errors (this function will be discussed in more detail later). With the answer in the res variable, we took the experiment_id value, as follows:

```
auto json = nlohmann::json::parse(res->body);
experiment_id_ = json["experiment_id"].get<std::string>();
```

Here, we parsed the string that was returned by the server with the nlohmann::json::parse function and read the experiment_id value from the JSON object into our class member variable.

- V. In the second part of the method, which works when an experiment exists on the server, we parsed the response in a JSON object and took the experiment_id value.

There are two functions named handle_result that are used to check response codes and report errors if needed. The first one is used to check whether a response has some particular code and is implemented as follows:

```
bool check_result(const httpplib::Result& res, int code) {
    if (!res) {
        throw std::runtime_error(
```

```

        "REST error: " + httpplib::to_string(res.error())));
    }
    return res->status == code;
}

```

Here, we checked whether the `httpplib::Result` object has a valid response by using its Boolean cast operator. If there was a communication error, we threw the runtime exception. Otherwise, we returned the response code's comparison result.

VI. The second `handle_result` function is used to check that we got the successful answer from the server. The following code snippet shows how it's implemented:

```

void handle_result(const httpplib::Result& res) {
    if (check_result(res, 200))
        return;
    std::ostringstream oss;
    oss << "Request error status: " << res->status << " "
        << httpplib::detail::status_message(res->status);
    oss << ", message: " << std::endl
        << res->body;
    throw std::runtime_error(oss.str());
}

```

We used the previous `handle_result` function to check whether the response was valid and we got a 200 response code. If it's true, we're OK. However, in the case of a failure, we must make a detailed report and throw a runtime exception.

These functions help to simplify response error handling code and make it easier to debug communications.

4. The next two methods we're going to discuss are `start_run` and `end_run`. These methods mark a single run's bounds, within which we can log metrics, parameters, and artifacts. In production code, it makes sense to wrap such functionality into some RAII abstraction, but we made two methods for simplicity.

The `start_run` method can be implemented as follows:

```

void MLFlow::start_run() {
    nlohmann::json request;
    request["experiment_id"] = experiment_id_;
    request["start_time"] =
        std::chrono::duration_cast<std::chrono::milliseconds>(
            std::chrono::system_clock::now().time_since_epoch())
            .count();
    auto res =
        http_client_.Post("/api/2.0/mlflow/runs/create",
                         request.dump(), "application/json");
}

```

```

        handle_result(res);
        auto json = nlohmann::json::parse(res->body);
        run_id_ = json["run"]["info"]["run_id"];
    }
}

```

Here, we made a JSON-based request to create a run. This request was filled with the current `experiment_id` value and the run's start time. Then, we sent a request to the server and got a response that we checked with the `handle_result` function. If we receive an answer, we parse it in the `nlohmann::json` object and take the `run_id` value. The `run_id` value is stored in the object member and will be used in the following requests. After we call this method, the tracking server will write all metrics and parameters into this new run.

- To complete the run, we have to tell the server about it. The `end_run` method does just this:

```

void MLFlow::end_run() {
    nlohmann::json request;
    request["run_id"] = run_id_;
    request["status"] = "FINISHED";
    request["end_time"] =
        std::chrono::duration_cast<std::chrono::milliseconds>(
            std::chrono::system_clock::now()
                .time_since_epoch())
            .count();
    auto res = http_client_.Post(
        "/api/2.0/mlflow/runs/update", request.dump(),
        "application/json");
    handle_result(res);
}

```

Here, we made a JSON-based request that includes the `run_id` value, the finished status, and the end time. Then, we sent this request to the tracking server and checked the response. Notice that we sent the start and end times for a run, at which point the server used them to calculate the run duration. Due to this, you'll be able to see how the run duration time depends on its parameters.

Now that we have methods for setting an experiment and defining a run, we need methods so that we can log metrics values and run parameters.

Logging metric values and running parameters

The difference between metrics and parameters is that metrics are sequences of values within a run. You can log as many values of a single metric as you need. Usually, this number equals epochs or batches and MLflow will show live plots for these metrics. However, a single parameter can only be logged once per run, and it's typically a training characteristic such as the learning rate.

A metric is usually a numeric value, so we've made our `log_metric` method take a float and an argument for a value. Note that this method takes the metric name and the epoch index to make several distinct values for the same metric. The method's implementation is shown in the following code snippet:

```
void MLFlow::log_metric(const std::string& name,
                        float value, size_t epoch) {
    nlohmann::json request;
    request["run_id"] = run_id_;
    request["key"] = name;
    request["value"] = value;
    request["step"] = epoch;
    request["timestamp"] =
        std::chrono::duration_
        cast<std::chrono::milliseconds>(
            std::chrono::system_clock::now()
                .time_since_epoch())
        .count();
    auto res = http_client_.Post(
        "/api/2.0/mlflow/runs/log-metric", request.dump(),
        "application/json");
    handle_result(res);
}
```

Here, we made a JSON-based request that includes the `run_id` value, the metric's name as the `key` field, the metric's value, the epoch index as the `step` field, and the timestamp value. Then, we sent the request to the tracking server and checked the response.

A parameter value can have an arbitrary value type, so we used C++ templates to write a single method to process different value types. There are two `log_param` functions here—the first is a template function that converts any suitable parameter value into a string, whereas the second only takes a parameter name and a string value as arguments. The template can be implemented like so:

```
template <typename T>
void log_param(const std::string& name, T value) {
    log_param(name, std::to_string(value));
}
```

This template simply redirects a call to the second function after the value is converted into a string with the `std::to_string` function. So, if the value's type can't be converted into a string, a compilation error will occur.

The second `log_param` function's implementation can be seen in the following code snippet:

```
void MLFlow::log_param(const std::string& name,
                      const std::string& value) {
    nlohmann::json request;
    request["run_id"] = run_id_;
    request["key"] = name;
    request["value"] = value;

    auto res = http_client_.Post("/api/2.0/mlflow/runs/log-parameter",
                                request.dump(), "application/json");
    handle_result(res);
}
```

Here, we made a JSON-based request that includes the current `run_id` value, the parameter name as the `key` field, and the value. Then, we just sent the request and checked the response.

The REST API in MLflow is much richer than this; we only covered the basic functions here. For example, it's also capable of accepting model architectures in JSON format, logging input datasets, managing experiments and models, and much more.

Now that we understand the basic functionality for communicating with the MLflow server, let's learn how to implement an experiments tracking session for a regression task.

Integrating experiment tracking into linear regression training

In this section, we'll be using the `Flashlight` library to implement a linear regression model and train it. Our code starts with initializing Flashlight and connecting to an MLflow server, as follows:

```
f1::init();
MLFlow mlflow("127.0.0.1", "5000");
mlflow.set_experiment("Linear regression");
```

Here, we assumed that the tracking server has already been started on localhost. After, we set the experiment's name to `Linear regression`. Now, we can define the necessary parameters and start the run:

```
int batch_size = 64;
float learning_rate = 0.0001;
float momentum = 0.5;
int epochs = 100;
mlflow.start_run();
```

Having configured the run, we can load datasets for training and testing, define a model, and create an optimizer and loss function according to the parameters we defined previously:

```
// load datasets
auto train_dataset = make_dataset(/*n=*/10000, batch_size);
auto test_dataset = make_dataset(/*n=*/1000, batch_size);
// Define a model
fl::Sequential model;
model.add(fl::View({1, 1, 1, -1}));
model.add(fl::Linear(1, 1));
// define MSE loss
auto loss = fl::MeanSquaredError();
// Define optimizer
auto sgd = fl::SGDOptimizer(model.params(), learning_rate, momentum);
// Metrics meter
fl::AverageValueMeter meter;
```

Notice that we used all the previously defined parameters except for the epoch number. Now, we're ready to define the training cycle, like so:

```
for (int epoch_i = 0; epoch_i < epochs; ++epoch_i) {
    meter.reset();
    model.train();

    for (auto& batch : *train_dataset) {
        sgd.zeroGrad();
        // Forward propagation
        auto predicted = model(fl::input(batch[0]));
        // Calculate loss
        auto local_batch_size = batch[0].shape().dim(0);
        auto target =
            fl::reshape(batch[1], {1, 1, 1, local_batch_size});
        auto loss_value = loss(predicted, fl::noGrad(target));
        // Backward propagation
        loss_value.backward();
        // Update parameters
        sgd.step();
        meter.add(loss_value.scalar<float>());
    }
    // Train metrics logging
    // ...
    // Calculate and log test metrics
    // ...
}
```

The main part of the training cycle looks normal as we implemented this in the previous chapters. Note that we have two nested cycles—one for epochs and another for batches. At the beginning of the training epoch, we cleared the meter that's used for averaging the training loss metric and put the model into the training mode. Then, we cleared the gradients, made a forward pass, calculated the loss value, made a backward pass, updated the model weights with the optimizer step, and added the loss value to the averaging meter object. After the internal cycle, training was completed. At this point, we can log the average training loss metric value to the tracking server, as follows:

```
// train metrics logging
auto avr_loss_value = meter.value()[0];
mlflow.log_metric("train loss", avr_loss_value, epoch_i);
```

Here, we logged the train loss value for the epoch with the `epoch_i` index and used `train loss` as its name. For every epoch, this logging will add a new value for the metric and we'll be able to see the live plot of how the training loss changes over epochs in the MLflow UI. This plot will be shown in the following subsection.

After the training cycle, for every 10th epoch, we calculate the test loss metric, as shown here:

```
// Every 10th epoch calculate test metric
if (epoch_i % 10 == 0) {
    fl::AverageValueMeter test_meter;
    model.eval();
    for (auto& batch : *test_dataset) {
        // Forward propagation
        auto predicted = model(fl::input(batch[0]));
        // Calculate loss
        auto local_batch_size = batch[0].shape().dim(0);
        auto target =
            fl::reshape(batch[1], {1, 1, 1, local_batch_size});
        auto loss_value = loss(predicted, fl::noGrad(target));
        // Add loss value to test meter
        test_meter.add(loss_value.scalar<float>());
    }
    // Logging the test metric
    // ...
}
```

Once we've checked that the current epoch is the 10th one, we defined an additional averaging meter object for the test loss metric and implemented evaluation mode. Then, we calculated the loss value for every batch and added these values to the averaging meter. At this point, we can implement a loss calculation for the test dataset and log the test metric to the tracking server:

```
// logging the test metric
auto avr_loss_value = test_meter.value()[0];
mlflow.log_metric("test loss", avr_loss_value, epoch_i);
```

Here, we logged the test loss value for the epoch with the `epoch_i` index and used `test loss` as its name. MLflow will provide a plot for this metric too. We'll be able to overlap this plot with the train metric plot to check whether there are issues such as overfitting.

Now that we've finished using the training cycle, we can end the run and log its parameters, like so:

```
mlflow.end_run();
mlflow.log_param("epochs", epochs);
mlflow.log_param("batch_size", batch_size);
mlflow.log_param("learning_rate", learning_rate);
mlflow.log_param("momentum", momentum);
```

Here, we logged the run parameters with the `end_run` call. This is a requirement when using the MLflow API. Note that parameter values can have different types and that they were only logged once.

Now, let's see how MLflow will display the program runs with different training parameters.

Experiments tracking process

The following figure shows the MLflow UI after the tracking server has been started:

The screenshot shows the MLflow UI's 'Experiments' page. On the left, a sidebar titled 'Experiments' contains a 'Default' experiment with a checkbox and edit icons. The main area has tabs for 'Runs', 'Evaluation', 'Experimental' (which is selected), and 'Traces'. A search bar at the top allows filtering by metrics and parameters. Below it, a table header includes columns for Run Name, Created, Dataset, Duration, Source, and Models. A large central message states 'No runs logged' with a small icon above it, and a note below explaining that no runs have been logged yet.

Figure 13.1 – Overview of the MLflow UI without experiments and runs

As we can see there are no experiments and there's run information. After executing a program run with a set of parameters, the UI will look as follows:

The screenshot shows the MLflow UI interface. At the top, there is a dark header bar with the 'mlflow' logo, version '2.15.0', and navigation links for 'Experiments' and 'Models'. On the right side of the header are 'Share', 'GitHub', and 'Docs' buttons. Below the header, the main content area has a title 'Linear regression' with options to 'Provide Feedback' and 'Add Description'. A 'Share' button is also present here. The left sidebar under 'Experiments' shows a search bar and two entries: 'Default' and 'Linear regression', with the latter being selected and highlighted in blue. The main panel is titled 'Runs' and contains a table with the following data:

Run Name	Created	Dataset	Duration	Source	Models
peaceful-ray-50	27 seconds ago	-	19.2s	-	-

Below the table, a message indicates '1 matching run'.

Figure 13.2 – Overview of the MLflow UI with a single experiment and one run

As you can see, **Linear regression** appeared in the left panel. Also, there's a new record for the run in the right-hand side table. Notice that the run name of **peaceful-ray-50** was automatically generated. Here, we can see the start time and how much time the run takes. Clicking on the run's name will open the run details page, which looks like this:

The screenshot shows the MLflow UI for a run named "peaceful-ray-50". At the top, there are tabs for "Overview", "Model metrics", "System metrics", and "Artifacts". Below the tabs, there's a "Details" section containing a table with various run attributes:

Created at	2024-08-18 20:57:44
Created by	(empty)
Experiment ID	984330160379813134 🔗
Status	<input checked="" type="checkbox"/> Finished
Run ID	4ea9b67d472b41ea883845766f6241dc 🔗
Duration	19.2s
Datasets used	—
Tags	Add
Source	—
Logged models	—
Registered models	—

Below the details, there are two tables: "Parameters (4)" and "Metrics (2)".

Parameters (4)

Parameter	Value
epochs	100
learning_rate	0.000100
batch_size	64
momentum	0.500000

Metrics (2)

Metric	Value
test loss	0.010005257092416286
train loss	0.009938654489815235

Figure 13.3 – Overview of the run details in the MLflow UI

Here, we can see the start time and date, the experiment ID that this run is associated with, the run ID, and its duration. Note that additional information might be provided here, such as a username, what datasets were used, tags, and the model source. These additional attributes can be also configured with the REST API.

At the bottom, we can see the **Parameters** table, where we can find the parameters we logged from our code. There's also the **Metrics** table, which shows the final values for our train and test loss value metrics.

If we click on the **Model metrics** tab, the following page will be displayed:

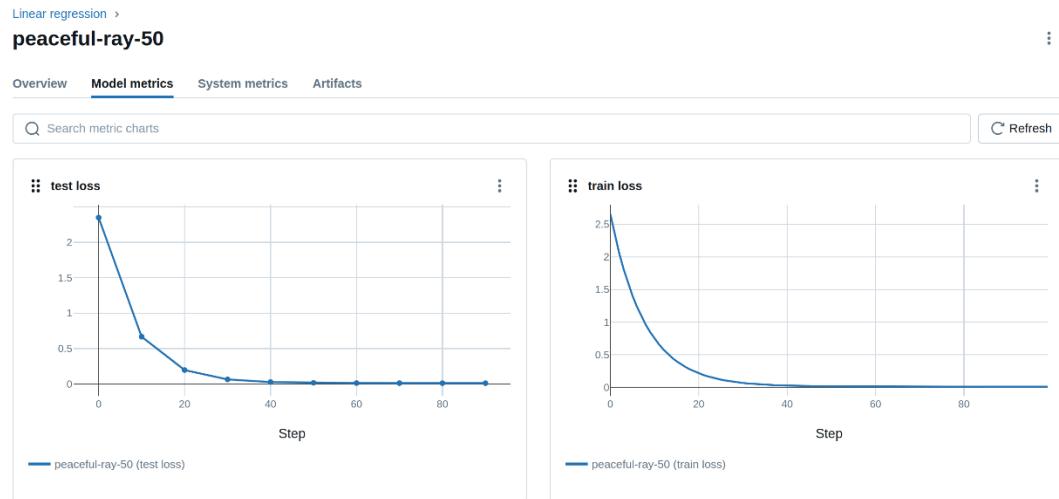


Figure 13.4 – The Model metrics page in the MLflow UI

Here, we can see the train and test loss metrics plots. These plots show how the loss values changed over epochs. Usually, it's useful to overlap the train and test loss plots to see some dependencies. We can do this by clicking on the metric's name on the page displayed in *Figure 13.3*. The following page will be displayed upon clicking **train loss**:

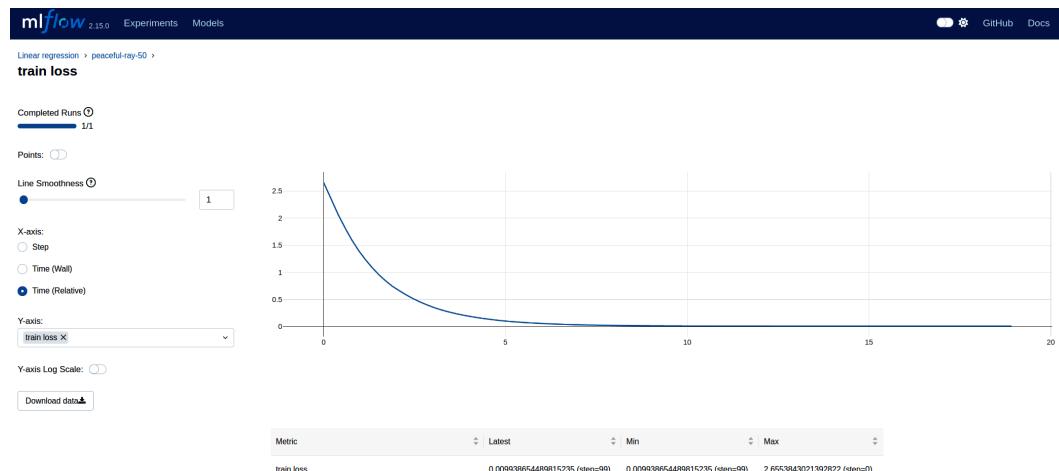


Figure 13.5 – The train loss metric plot

Here, we can see the plot for the single metric. On this page, we can configure some visualization parameters for the plot, such as the smoothness and the step. However, in this case, we're interested in the **Y-axis** field, which allows us to add additional metrics to the same plot. If we add the **test loss** metric, we'll see the following page:

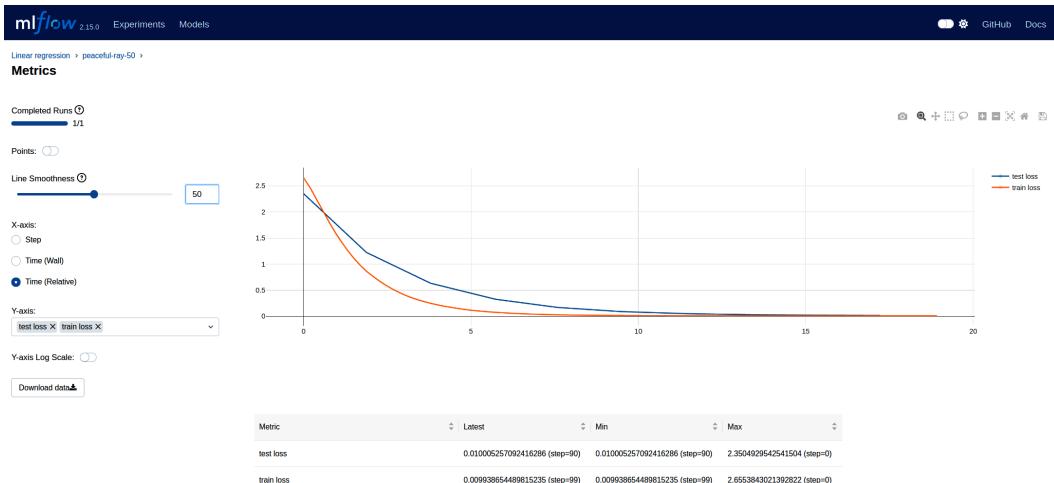


Figure 13.6 – Overlapping the metric plots

Now, we have two overlapped plots for the train and test metrics. In this visualization, we can see that for the first few epochs, the test loss was greater than the train loss, but after the 15th epoch, the loss values were pretty similar. This means that there's no model overfitting.

In this instance, we looked at the main regimes of the MLflow UI for the single train run. For more advanced cases, there will be pages that consist of artifacts and model sources, but we've skipped them here.

Next, let's learn how to work with several runs for an experiment. We ran our application again but with a different value for momentum. MLflow shows us that we have two runs for the same experiment, as follows:

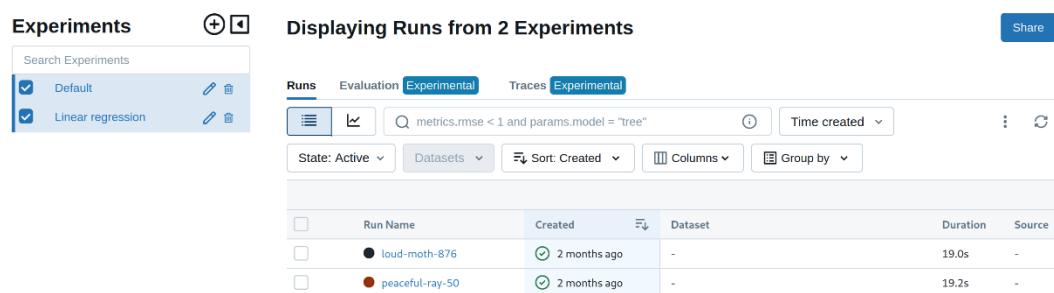


Figure 13.7 – The experiment with two runs

As we can see, there are two runs for the experiment. Also, there are only two minor differences between them—the names and their duration. To compare the runs, we must click on both checkboxes that precede the run names, as shown here:

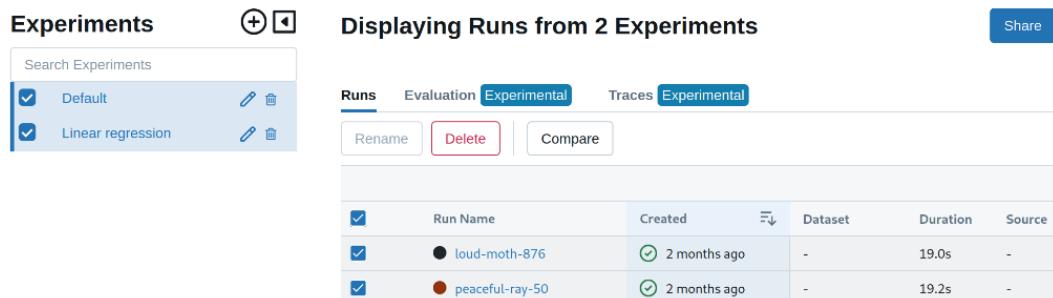
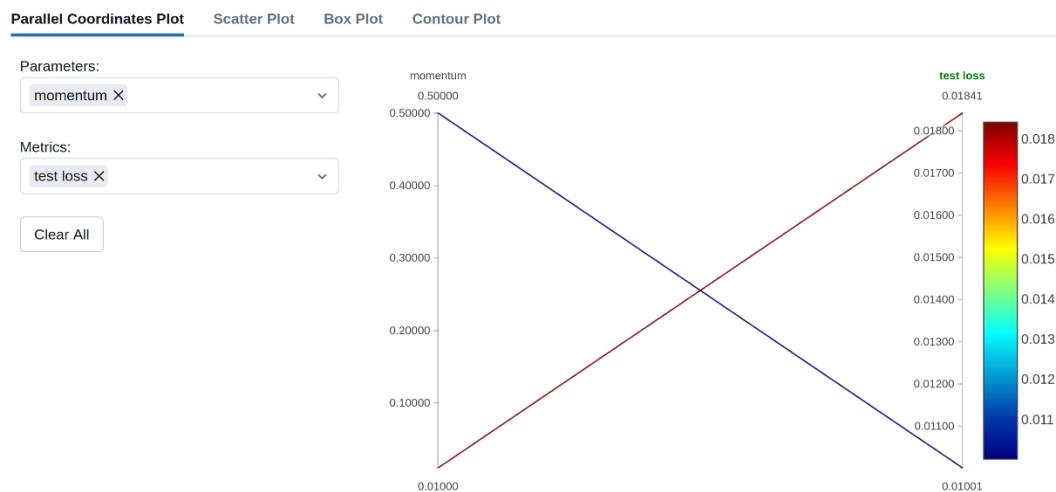


Figure 13.8 – Selecting both runs

After selecting both runs, the **Compare** button appears at the top of the **Runs** table. Clicking this button opens the following page:



Run details		
Run ID:	d3c3bfa5439146bd811a804083108f9d	4ea9b67d472b41ea883845766f6241dc
Run Name:	loud-moth-876	peaceful-ray-50
Start Time:	2024-08-18 22:05:13	2024-08-18 20:57:44
End Time:	2024-08-18 22:05:32	2024-08-18 20:58:03
Duration:	19.0s	19.2s

Parameters		
<input checked="" type="checkbox"/> Show diff only		
batch_size	64	64
epochs	100	100
learning_rate	0.000100	0.000100
momentum	0.010000	0.500000

Figure 13.9 – Overview of the runs comparison page

This page shows two runs side by side and shows different visualizations of the difference between various metrics and parameters. Note that the run parameter differences will also be highlighted. From the left top panel, you can select the parameters and metrics you wish to compare. By doing this, we can see that the new run with a lower momentum value performs worse. This is indicated by the top plot, where lines connect parameters with metrics and there are scales with values. This can also be seen at the bottom in the metrics rows, where you can compare final metric values.

In this section, we learned how to use the MLflow UI to explore experiment run behavior, as well as how to view metrics visualizations and how to compare different runs. All tracked information is saved by the tracking server and can be used later, after a server restart, so it's quite a useful tool for ML practitioners.

Summary

Visualization and experiment tracking systems are essential tools for ML engineers. They allow us to understand the performance of models, analyze results, and improve the overall process.

TensorBoard is a popular visualization system that provides detailed information about model training, including metrics, loss curves, histograms, and more. It supports multiple frameworks, including TensorFlow, and allows us to easily compare different runs.

MLflow is an open source framework that offers end-to-end solutions for model life cycle management. It includes features such as experiment tracking, Model Registry, artifact management, and deployment. MLflow helps teams collaborate, reproduce experiments, and ensure reproducibility.

Both TensorBoard and MLflow are powerful tools that can be used together or separately, depending on your needs.

After understanding both TensorBoard and MLflow, we implemented a linear regression training example with experiment tracking. By doing so, we learned how to implement the REST API client for the MLflow server and how to use it to log metrics and parameters for an experiment. Then, we explored the MLflow UI, where we learned how to view an experiment and its run details, as well as metrics plots, and learned how to compare different runs.

In the next chapter, we'll learn how to use ML models for computer vision on the Android mobile platform using C++.

Further reading

- **MLflow REST API:** <https://mlflow.org/docs/latest/rest-api.html>
- **MLflow documentation:** <https://mlflow.org/docs/latest/index.html>
- **TensorBoard documentation:** https://www.tensorflow.org/tensorboard/get_started
- *How to use TensorBoard with PyTorch:* https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html

14

Deploying Models on a Mobile Platform

In this chapter, we'll discuss deploying **machine learning (ML)** models on mobile devices running on the Android operating system. ML can be used to improve the user experience on mobile devices, especially since we can create more autonomous features that allow our devices to learn and adapt to user behavior. For example, ML can be used for image recognition, allowing devices to identify objects in photos and videos. This feature can be useful for applications such as augmented reality or photo editing tools. Additionally, ML-powered speech recognition can enable voice assistants to better understand and respond to natural language commands. Another important benefit of the autonomous features development is that they can work without an internet connection. This is particularly useful in situations where connectivity is limited or unreliable, such as when traveling in remote areas or during natural disasters.

Using C++ on mobile devices allows us to make programs faster and more compact. We can utilize as many computational resources as possible because modern compilers can optimize the program concerning the target CPU architecture. C++ doesn't use an additional garbage collector for memory management, which can have a significant impact on program performance. Program size can be reduced because C++ doesn't use an additional **Virtual Machine (VM)** and is compiled directly into machine code. Also, the use of C++ can help optimize battery life by more precise resource usage and adjusting accordingly. These facts make C++ the right choice for mobile devices with a limited amount of resources and can be used to solve heavy computational tasks.

By the end of the chapter, you will learn how to implement real-time object detection using a camera on an Android mobile platform using PyTorch and YOLOv5. But this chapter is not a comprehensive introduction to Android development; rather, it can be used as a starting point for experiments with ML and computer vision on an Android platform. It provides a complete minimal example of the project that you will be able to extend for your task.

This chapter covers the following topics:

- Creating the minimal required project for Android C++ development
- Implementing the minimal required Kotlin functionality for object detection
- Initializing the image-capturing session in the C++ part of the project
- Using OpenCV to process native camera images and draw results
- Using PyTorch script to launch the YOLOv5 model on the Android platform

Technical requirements

The following are the technical requirements for this chapter:

- Android Studio, **Android Software Development Kit (SDK)**, and **Android Native Development Kit (NDK)**
- The PyTorch library
- A modern C++ compiler with C++20 support
- CMake build system version ≥ 3.22

The code files for this chapter can be found at the following GitHub repository: <https://github.com/PacktPublishing/Hands-On-Machine-Learning-with-C-Second-edition/tree/main/Chapter14>.

Developing object detection on Android

There are many approaches regarding how to deploy an ML model to a mobile device with Android. We can use PyTorch, ExecuTorch, TensorFlow Lite, NCNN, ONNX Runtime, or others. We'll use the PyTorch framework in this chapter since we have discussed it in the previous chapters, and because it allows us to use almost any PyTorch model with minimal functional restrictions. Unfortunately, we will be able to use only the target device CPU for inference. Other frameworks, such as ExecuTorch, TensorFlow Lite, NCNN, and ONNX Runtime, allow you to use other inference backends, such as onboard GPU or **Neural Processing Unit (NPU)**. However, this option also comes with a notable restriction, which is the lack of certain operators or functions, which can limit the types of models that can be deployed on mobile devices. Dynamic shape support is usually limited, making it difficult to handle data with varying dimensions.

Another challenge is restricted control flow, which limits the ability to use models with dynamic computational graphs and implement advanced algorithms. These restrictions can make it more challenging to deploy ML models on mobile platforms using the frameworks described earlier. So, there is a trade-off between the model's functionality and the required performance when you deploy ML models on mobile devices. To balance functionality and performance, developers must carefully evaluate their requirements and choose a framework that meets their specific needs.

The mobile version of the PyTorch framework

There is an available binary distribution of PyTorch for mobile devices available in the Maven repository named `org.pytorch:pytorch_android_lite`. However, this distribution is outdated. So, to use the most recent version, we need to build it from source code. We can do this in the same way as we compile its regular version but with additional CMake parameters to enable mobile mode. You also have to install the Android NDK, which includes an appropriate version of the C/C++ compiler and the Android native libraries that are required to build the application.

The simplest way to install Android development tools is to download the Android Studio IDE and use the SDK Manager tool from that. You can find the SDK Manager under the **Tools | SDK Manager** menu. You can use this manager to install appropriate Android SDK versions. You can install the corresponding NDKs by using the **SDK Tools** tab in the manager's window. You can also use this tab to install the CMake utility. Another way to get NDK and build tools for Android is to use the `cmdline-tools` package. However, you need to have Java in your system; for Ubuntu, you can install Java as follows:

```
sudo apt install default-jre
```

The following command line script shows you how to install all required packages for CLI development:

```
# make the folder where to install components
mkdir android
cd android
# download command line tools
wget https://dl.google.com/android/repository/commandlinetools-
linux-9477386_latest.zip
# unzip them and move to the correct folder
unzip commandlinetools-linux-9477386_latest.zip
mv cmdline-tools latest
mkdir cmdline-tools
mv latest cmdline-tools
# install SDK, NDK and build tools for Android using sdkmanager
utility
yes | ./cmdline-tools/latest/bin/sdkmanager --licenses
yes | ./cmdline-tools/latest/bin/sdkmanager "platform-tools"
yes | ./cmdline-tools/latest/bin/sdkmanager "platforms;android-35"
yes | ./cmdline-tools/latest/bin/sdkmanager "build-tools;35.0.0"
yes | ./cmdline-tools/latest/bin/sdkmanager "system-images;android-
35;google_apis;arm64-v8a"
yes | ./cmdline-tools/latest/bin/sdkmanager --install
"ndk;26.1.10909125"
```

Here, we used the `sdkmanager` manager utility to install all required components with appropriate versions. Using this script, the path to NDK will be as follows:

```
android/ndk/26.1.10909125/
```

Having installed build tools and NDK, we can move on to the PyTorch mobile version compilation.

The following code snippet shows you how to use the command line environment to check out PyTorch and build it:

```
cd /home/[USER]
git clone https://github.com/pytorch/pytorch.git
cd pytorch/
git checkout v2.3.1
git submodule update --init --recursive
export ANDROID_NDK=[Path to the installed NDK]
export ANDROID_ABI='arm64-v8a'
export ANDROID_STL_SHARED=1
$START_DIR/android/pytorch/scripts/build_android.sh \
-DBUILD_CAFFE2_MOBILE=OFF \
-DBUILD_SHARED_LIBS=ON \
-DUSE_VULKAN=OFF \
-DCMAKE_PREFIX_PATH=$(python -c 'from distutils.sysconfig import get_python_lib; print(get_python_lib())') \
-DPYTHON_EXECUTABLE=$(python -c 'import sys; print(sys.executable)') \
```

Here, we assumed that `/home/ [USER]` is the user's home directory. The main requirement when it comes to building the mobile version of PyTorch is to declare the `ANDROID_NDK` environmental variable, which should point to the Android NDK installation directory. The `ANDROID_ABI` environment variable can be used to specify the **ARM (Advanced RISC Machines)** CPU architecture's compatibility for the compiler to generate architecture-specific code. In this example, we used the `arm64-v8a` architecture.

We used the `build_android.sh` script from the PyTorch source code distribution to build mobile PyTorch binaries. This script uses the CMake command internally, which is why it takes CMake parameter definitions as arguments. Notice that we passed the `BUILD_CAFFE2_MOBILE=OFF` parameter to disable building the mobile version of Caffe2, which is hard to use in the current version because the library is deprecated. The second important parameter we used was `BUILD_SHARED_LIBS=ON`, which enabled us to build shared libraries. Also, we disabled the Vulkan API support by using `DUSE_VULKAN=OFF` because it's still experimental and has some compilation problems. The other parameters that were configured were the Python installation paths for intermediate build code generation.

Now that we have the mobile PyTorch libraries, that is, `libc10.so` and `libtorch.so`, we can start developing the application. We are going to build an object detection application based on the YOLOv5 neural network architecture.

YOLOv5 is an object detection model based on the **You Only Look Once (YOLO)** architecture. It's a state-of-the-art deep learning model that can detect objects in images and videos with high accuracy and speed. The model is relatively small and lightweight, making it easy to deploy on resource-constrained devices. Also, it's fast enough, which is important for real-time applications that analyze a real-time video stream. It's open source software, which means that developers can freely access the code and modify it to suit their needs.

Using TorchScript for a model snapshot

In this section, we will discuss how to get the YOLOv5 model TorchScript file so that we can use it in our mobile application. In the previous chapters, we discussed how to save and load model parameters and how to use the ONNX format to share models between frameworks. When we use the PyTorch framework, there is another method we can use to share models between the Python API and C++ API called **TorchScript**.

This method uses real-time model tracing to get a special type of model definition that can be executed by the PyTorch engine, regardless of API. In PyTorch, only the Python API can create such definitions, but we can use the C++ API to load the model and execute it. Also, the mobile version of the PyTorch framework doesn't allow us to program neural networks with a full-featured C++ API. However, as was said earlier, TorchScript allows us to export and run models with complex control flow and dynamic shapes, which is not fully possible now for ONNX and other formats used in other mobile frameworks.

For now, the YOLOv5 PyTorch model can be directly exported only into TorchScript for inference on mobile CPUs. For example, there are YOLOv5 models adapted for TensorFlow Lite and NCNN frameworks, but we will not discuss these cases because we are using PyTorch mostly. I have to say that using NCNN will allow you to use a mobile GPU though the Vulkan API and using TensorFlow Lite or ONNX Runtime for Android will allow you to use a mobile NPU for some devices. However, you will need to adapt the model into another format by reducing some functionality or developing it with TensorFlow.

So, in this example, we are going to use the TorchScript model to perform object detection. To get the YOLOv5 model, we have to perform the following steps:

1. Clone the model repository from GitHub and install dependencies; run these commands in the terminal:

```
git clone https://github.com/ultralytics/yolov5  
cd yolov5  
pip install -r requirements.txt
```

- Run the export script in the terminal to get the PyTorch jit script of the model optimized for mobile:

```
python export.py --weights yolov5s.torchscript --include  
torchscript --optimize
```

The script from the second step automatically traces the model and saves the TorchScript file for us. After we made these steps, there will be the `yolo5s.torchscript` file, which we will be able to load and use in C++.

Now, we have all the prerequisites to move on and make an Android Studio project for our application.

The Android Studio project

In this section, we will use the Android Studio IDE to create our mobile application. We can use a default **native C++** wizard in the Android Studio IDE to create an application stub. If we name the project `objectdetection` and select **Kotlin** as the programming language, then Android Studio will create a particular project structure; the following sample shows the most valuable parts of it:

```
app  
| --src  
|   |--main  
|     |--cpp  
|       |--CmakeLists.txt  
|       `--native-lib.cpp  
|     |--java  
|       `--com  
|         |--example  
|           |--objectdetection  
|             `--MainActivity.kt  
|     |--res  
|       |--layout  
|         |--activity_main.xml  
|     |--values  
|       |--colors.xml  
|       |--strings.xml  
|       |--styles.xml  
|       `--...  
| --build.gradle  
`--...
```

The `cpp` folder contains the C++ part of the whole project. In this project, the Android Studio IDE created the C++ part as a native shared library project that had been configured with the CMake build generation system. The `java` folder contains the Kotlin part of the project. In our case, it is a single file that defines the main activity—the object that's used as a connection between the UI elements and event handlers. The `res` folder contains project resources, such as UI elements and string definitions.

We also need to create the `JniLibs` folder, under the `main` folder, with the following structure:

```
app
| --src
| | --main
| | | --...
| | | --JniLibs
| | | | --arm64-v8a
| | | | | --libc10.so
| | | | | --libtorch_cpu.so
| | | | | --libtorch_global_deps.so
| | | | `--libtorch.so
`--...
```

Android Studio requires us to place additional native libraries in such folders to correctly package them into the final application. It also allows the **Java Native Interface (JNI)** system to be able to find these libraries. Notice that we placed PyTorch libraries in the `arm64-v8a` folder because they have only been compiled for this CPU architecture. If you have libraries for other architectures, you have to create folders with corresponding names.

Also, in the previous subsection, we learned how to get the YOLOv5 torch script model. The model file and corresponding file, along with the class IDs, should be placed in the `assets` folder. This folder should be created beside the `JniLibs` folder on the same folder level, as follows:

```
app
| --src
| | --main
| | | --...
| | | --cpp
| | | --JniLibs
| | | --assets
| | | | --yolov5.torchscript
| | | | | --classes.txt
| | | --...
`--...
```

The file that maps string class names to numerical IDs, which the model returns, can be downloaded from <https://github.com/ultralytics/yolov5/blob/master/data/coco.yaml>.

In our example, we simply convert the YAML file into the text one to make its parsing simpler.

The IDE uses the Gradle build system for project configuration, so there are two files named `build.gradle.kts`, one for the application module and another one for the project properties. Look at the `build.gradle` file for the application module in our example. There are two variables that define paths to the PyTorch source code folder and to the OpenCV Android SDK folder. You need to update their values if you change these paths. The prebuilt OpenCV Android SDK can be downloaded from the official GitHub repository (<https://github.com/opencv/opencv/releases>) and simply unpacked.

The Kotlin part of the project

In this project, we are going to use the native C++ part to draw the captured picture with bounding boxes and class labels for detected objects. So, there will be no UI code and declarations in the Kotlin part. However, the Kotlin part will used to request and check required camera access permissions. Also, it will start a camera capture session if permissions are granted. All Kotlin code will be in the `MainActivity.kt` file.

Preserving camera orientation

In our project, we skip the implementation of device rotation handling to make code simpler and show just the most interesting parts of working with the object detection model. So, to make our code stable, we have to disable the landscape mode, which can be done in the `AndroidManifest.xml` file, as follows:

```
...
<activity
    ...
        android:screenOrientation="portrait"
    ...

```

We added the screen orientation instruction to the activity entity. This is not a good solution because there are devices that work only in landscape mode and our application will not work with them. In a real production-ready application, you should handle different orientation modes; for example, for most smartphones, this dirty solution should work.

Handling camera permission requests

There are no C++ APIs in Android NDK to request permissions. We can request the required permission only from the Java/Kotlin side or with JNI from C++. It's simpler to write the Kotlin code to request the camera permission than to write JNI calls.

The first step is modifying the declaration of the `MainActivity` class to be able to process permission request results. It's done as follows:

```
class MainActivity
    : NativeActivity(),
        ActivityCompat.OnRequestPermissionsResultCallback {
    ...
}
```

Here, we inherited the `MainActivity` class from the `OnRequestPermissionsResultCallback` interface. It gives us the possibility to override the `onRequestPermissionsResult` method where we will be able to check a result. However, to get a result, we have to make a request first, as follows:

```
override fun onResume() {
    super.onResume() val cameraPermission =
        android.Manifest.permission
        .CAMERA if (checkSelfPermission(
            cameraPermission) != PackageManager.PERMISSION_GRANTED) {
        requestPermissions(arrayOf(cameraPermission),
            CAM_PERMISSION_CODE)
    }
    else {
        val camId =
            getCameraBackCameraId() if (camId.isEmpty()) {
            Toast
                .makeText(
                    this,
                    "Camera probably won't work on this
                     device !",
                    Toast.LENGTH_LONG)
                .show() finish()} initObjectDetection(camId)
    }
}
```

We overrode the `onResume` method of the `Activity` class. This method is called every time when our application starts to work or is resumed from the background. We initialized the `cameraPermission` variable with the required camera permission constant value. Then, we checked whether we already granted this permission using the `checkSelfPermission` method. If we don't have the camera permission, we ask for it with the `requestPermissions` method.

Notice that we used the `CAM_PERMISSION_CODE` code to identify our request in the callback method. If we were granted access to a camera, we tried to get the back-facing camera ID and initialize the object detection pipeline for this camera. If we can't get access to a camera, we finish the Android activity with the `finish` method and the corresponding message. In the `onRequestPermissionsResult` method, we check if the required permission was granted, as follows:

```
override fun onRequestPermissionsResult(requestCode
        : Int, permissions
        : Array<out String>, grantResults
        : IntArray) {
    super.onRequestPermissionsResult(requestCode,
        permissions,
        grantResults)
    if (requestCode == CAM_PERMISSION_CODE &&
        grantResults[0] != PackageManager.PERMISSION_GRANTED)
    {
        Toast.makeText(this,
            "This app requires camera permission",
            Toast.LENGTH_SHORT).show()
        finish()
    }
}
```

At first, we called the parent method to preserve the standard application behavior. Then, we checked the permission identification code, `CAM_PERMISSION_CODE`, and whether or not the permission was granted. In the failure case, we just show the error message and finish the Android activity.

As we said before, in the success case, we looked for the back-facing camera ID, which is done as follows:

```
private fun getCameraBackCameraId(): String {
    val camManager = getSystemService(
        Context.CAMERA_SERVICE) as CameraManager
    for (camId in camManager.cameraIdList) {
        val characteristics =
            camManager.getCameraCharacteristics(camId)
        val hwLevel = characteristics.get(
            CameraCharacteristics.INFO_SUPPORTED_HARDWARE_LEVEL)
        val facing = characteristics.get(
            CameraCharacteristics.LENS_FACING)
        if (hwLevel != INFO_SUPPORTED_HARDWARE_LEVEL_LEGACY &&
            facing == LENS_FACING_BACK) {
            return camId
        }
    }
}
```

```
    }
    return ""
}
```

We got the instance of the `CameraManager` object and used this object to iterate over every camera on a device. For each camera object, we asked for its characteristics, supported hardware level, and where this camera faces. If a camera is a regular legacy device and faces back, we return its ID. If we didn't find a suitable device, we returned an empty string.

Having granted the camera access permission and the camera ID, we called the `initObjectDetection` function to start image capturing and object detection. This and the `stopObjectDetection` function are functions provided through the JNI from the C++ part to the Kotlin part. The `stopObjectDetection` function is used to stop the camera capturing session, as follows:

```
override fun onPause() {
    super.onPause()
    stopObjectDetection()
}
```

In the overridden `onPause` activity method, we just stopped the camera capturing session. This method is called every time the Android application is closed or goes into the background.

Native library loading

There are two methods, `initObjectDetection` and `stopObjectDetection`, which are JNI calls to the native library functions that are implemented with C++. To connect the native library with the Java or Kotlin code, we use JNI. This is a standard mechanism that's used for calling C/C++ functions from Kotlin or Java.

First, we have to load the native library with the `System.LoadLibrary` call and place it in the companion object for our activity. Then, we have to define the methods that are implemented in the native library by declaring them as `external`. The following snippet shows how to define these methods in Kotlin:

```
private external fun initObjectDetection(camId: String)
private external fun stopObjectDetection()
companion object {
    init {
        System.loadLibrary("object-detection")
    }
}
```

Such declarations allow Kotlin to find the corresponding native library binary, load it, and access the functions. JNI works by providing a set of APIs that allow Java code to call into native code and vice versa. The JNI API consists of a number of functions that can be called from Java or native code. These functions allow you to perform tasks, such as creating and accessing Java objects from native code, calling Java methods from native code, and accessing native data structures from Java.

Internally, JNI works by mapping Java objects and types to their corresponding native counterparts. This mapping is done using the `JNIEnv` interface, which provides access to the **Java Virtual Machine (JVM)** internal state. When a Java method is called from native code, `JNIEnv` is used to find the corresponding native method and pass it the necessary arguments. Similarly, when a native method returns a value, `JNIEnv` is used to convert the native value to a Java object. The JVM manages memory for both Java and native objects. However, native code must explicitly allocate and free its own memory. JNI provides functions for allocating and freeing memory, as well as for copying data between Java and native memory. JNI code must be thread-safe. This means that any data accessed by JNI must be properly synchronized to avoid race conditions. Using JNI can have performance implications. Native code is typically faster than Java code, but there is overhead associated with calling into native code through JNI.

In the next section, we will discuss the C++ part of the project.

The native C++ part of the project

The main functionality of this example project is implemented in the native C++ part. It's designed to use the OpenCV library to deal with camera images and the PyTorch framework for object detection model inference. Such an approach allows you to port this solution to another platform if needed and allows you to use standard desktop instruments, such as OpenCV and PyTorch, to develop and debug algorithms that will be used on mobile platforms.

There are two main C++ classes in this project. The `Detector` class is the application facade that implements a connection with the Android activity image-capturing pipeline, and delegates object detection to the second class, `YOLO`. The `YOLO` class implements the object detection model loading and its inference.

The following subsections will describe the implementation details of these classes.

Initialization of object detection with JNI

We finished our discussion of the Kotlin part by talking about the JNI function declarations. The corresponding C++ implementation for `initObjectDetection` and `stopObjectDetection` are located in the `native-lib.cpp` file. This file is automatically created by the Android Studio IDE for the native activity projects. The following code snippet shows the `initObjectDetection` function definition:

```
#include <jni.h>
...

```

```

std::shared_ptr<ObjectDetector> object_detector_;
extern "C" JNIEXPORT void JNICALL
Java_com_example_objectdetection_MainActivity_initObjectDetection(
    JNIEnv* env,
    jobject /* this */,
    jstring camId) {
    auto camera_id = env->GetStringUTFChars(camId, nullptr);

    LOGI("Camera ID: %s", camera_id);
    if (object_detector_) {
        object_detector_->allow_camera_session(camera_id);
        object_detector_->configure_resources();
    } else
        LOGE("Object Detector object is missed!");
}

```

We followed JNI rules to make the function declaration correct and visible from the Java/Kotlin part. The name of the function includes the full Java package name, including namespaces, and our first two required parameters are the `JNIEnv*` and `jobject` types. The third parameter is the string and corresponds to the camera ID; this is the parameter that exists in the Kotlin declaration of the function.

In the function implementation, we checked whether the `ObjectDetector` object was already instantiated and, in this case, we called the `allow_camera_session` method with the camera ID and then called the `configure_resources` method. These calls make the `ObjectDetector` object remember what camera to use and initialize, configure the output window, and initialize the image-capturing pipeline.

The second function we used in the Kotlin part is the `stopObjectDetection`, and its implementation is done as follows:

```

extern "C" JNIEXPORT void JNICALL
Java_com_example_objectdetection_MainActivity_stopObjectDetection(
    JNIEnv*,
    jobject /* this */) {
    if (object_detector_) {
        object_detector_->release_resources();
    } else
        LOGE("Object Detector object is missed!");
}

```

Here, we just released resources used for the image-capturing pipeline because when the application is suspended, access to the camera device is blocked. When the application is activated again, the `initObjectDetection` function will be called and the image-capturing pipeline will be initialized again.

You can see that we used the `LOGI` and the `LOGE` functions, which are defined as follows:

```
#include <android/log.h>
#define LOG_TAG "OBJECT-DETECTION"
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO,
                                    LOG_TAG, __VA_ARGS__)
#define LOGW(...) __android_log_print(ANDROID_LOG_WARN,
                                    LOG_TAG, __VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR,
                                    LOG_TAG, __VA_ARGS__)
#define ASSERT(cond, fmt, ...)
    if (!(cond))
    {
        __android_log_assert(#cond, LOG_TAG, fmt, ##__VA_ARGS__);
    }
    \
```

We defined these functions to log messages into the Android `logcat` subsystem more easily. This series of functions uses the same tag for logging and has fewer arguments than the original `__android_log_xxx` functions. Also, the log level was encoded in the function name.

Main application loop

This project will use the Native App Glue library. This is a library for Android developers that helps to create native applications. It provides an abstraction layer between the Java code and the native code, making it easier to develop applications using both languages.

The use of this library allows us to have the standard `main` function with a loop that runs continuously, updating the UI, processing user input, and responding to system events. The following code snippet shows how we implemented the main function in the `native-lib.cpp` file:

```
extern "C" void android_main(struct android_app* app) {
    LOGI("Native entry point");
    object_detector_ = std::make_shared<ObjectDetector>(app);
    app->onAppCmd = ProcessAndroidCmd;
    while (!app->destroyRequested) {
        struct android_poll_source* source = nullptr;
        auto result = ALooper_pollOnce(0, nullptr, nullptr,
                                       (void**)&source);
        ASSERT(result != ALOOPER_POLL_ERROR,
               "ALooper_pollOnce returned an error");
        if (source != nullptr) {
            source->process(app, source);
        }
        if (object_detector_)
            object_detector_->draw_frame();
    }
}
```

```
    object_detector_.reset();  
}
```

This `android_main` function takes the instance of the `android_app` type, instead of regular `argc` and `argv` parameters. The `android_app` is a C++ class that provides access to the Android framework and allows you to interact with system services. Also, you can use it to access the device hardware, such as sensors and cameras.

The `android_main` main function is the starting point for our native module. So, we initialized the global `object_detector_` object here, and it became available for the `initObjectDetection` and `stopObjectDetection` functions. For initialization, the `ObjectDetector` instance takes the pointer to the `android_app` object.

Then, we attached the command processing function to the Android application object. Finally, we started the main loop, and it worked until the application was destroyed (closed). In this loop, we used the `ALooper_pollOnce` Android NDK function to get a pointer to the commands (events) poller object.

We called the `process` method of this object to dispatch the current command to our `ProcessAndroidCmd` function through the `app` object. At the end of the loop, we used our object detector object to grab the current camera picture and process it in the `draw_frame` method.

The `ProcessAndroidCmd` function is implemented as follows:

```
static void ProcessAndroidCmd(struct android_app* /*app*/,  
                             int32_t cmd) {  
    if (object_detector_) {  
        switch (cmd) {  
            case APP_CMD_INIT_WINDOW:  
                object_detector_->configure_resources();  
                break;  
            case APP_CMD_TERM_WINDOW:  
                object_detector_->release_resources();  
                break;  
        }  
    }  
}
```

Here, we processed only two commands that correspond to the application window initialization and termination. We used them to initialize and clear the image-capturing pipeline in the object detector. When the window is created, we configure its dimensions according to the capturing resolution. The window termination command allows us to clear capturing resources to prevent access to the already blocked camera device.

That is all the information about the `native-lib.cpp` file. The next subsections will look at the `ObjectDetector` class implementation details.

The ObjectDetector class overview

This is the main facade of the whole object detection pipeline of our application. The following list shows the functionality items it implements:

- Camera device access management
- Application window dimensions configuration
- Image-capturing pipeline management
- Camera image converting into OpenCV matrix objects
- Drawing an object detection result into the application window
- Delegating the object detection to the YOLO inference object

Before we start looking at these item details, let's see how the constructor, the destructor, and some helper methods are implemented. The constructor implementation is as follows:

```
ObjectDetector::ObjectDetector(android_app *app) : android_app_(app) {
    yolo_ = std::make_shared<YOLO>(app->activity->assetManager);
}
```

We just saved the pointer to the `android_app` object and created the `YOLO` class inference object. Also, we used the `android_app` object to get a pointer to the `AssetManager` object, which is used to load files packaged into the **Android Application Package (APK)**. The destructor is implemented as follows:

```
ObjectDetector::~ObjectDetector() {
    release_resources();
    LOGI("Object Detector was destroyed!");
}
void ObjectDetector::release_resources() {
    delete_camera();
    delete_image_reader();
    delete_session();
}
```

We called the `release_resources` method, which is where we close the opened camera device and clear capturing pipeline objects. The following code snippet shows the methods that are used from the Kotlin part through the `initObjectDetection` function:

```
void ObjectDetector::allow_camera_session(std::string_view camera_id)
{
    camera_id_ = camera_id;
}
```

In `allow_camera_session`, we saved the camera ID string; the device with this ID will be opened in the `configure_resources` method. As we already know, the camera ID will be passed to `ObjectDetector` only if the required permission is granted and there is a back-facing camera on the Android device. So, we defined `is_session_allowed` as follows:

```
bool ObjectDetector::is_session_allowed() const {
    return !camera_id_.empty();
}
```

Here, we just checked if a camera ID is not empty.

The following subsections will show the main functionality items in detail.

Camera device and application window configuration

There is the `create_camera` method in the `ObjectDetection` class that implements the creation of a camera manager object and a camera device opening as follows:

```
void ObjectDetector::create_camera() {
    camera_mgr_ = ACameraManager_create();
    ASSERT(camera_mgr_, "Failed to create Camera Manager");
    ACameraManager_openCamera(camera_mgr_, camera_id_.c_str(),
                             &camera_device_callbacks,
                             &camera_device_);
    ASSERT(camera_device_, "Failed to open camera");
}
```

`camera_mgr_` is the `ObjectDetector` member variable and after initialization, it is used to open a camera device. The pointer to the opened camera device will be stored in the `camera_device_` member variable. Also, notice that we used the camera ID string to open the particular device. The `camera_device_callbacks` variable is defined as follows:

```
namespace {
void onDisconnected(
    [[maybe_unused]] void* context,
    [[maybe_unused]] ACameraDevice* device) {
    LOGI("Camera onDisconnected");
}
void onError([[maybe_unused]] void* context,
            [[maybe_unused]] ACameraDevice* device,
            int error) {
    LOGE("Camera error %d", error);
}
ACameraDevice_stateCallbacks camera_device_callbacks = {
    .context = nullptr,
```

```

    .onDisconnected = onDisconnected,
    .onError = onError,
};

} // namespace

```

We defined the `ACameraDevice_stateCallbacks` structure object with references to functions that simply report if the camera is opened or closed. These handlers can do some more useful work in other applications, but we can't initialize them with nulls due to the API requirements.

The `create_camera` method is called in the `configure_resources` method of the `ObjectDetection` class. This method is called every time the application is activated and it has the following implementation:

```

void ObjectDetector::configure_resources() {
    if (!is_session_allowed() || !android_app_ ||
        !android_app_->window) {
        LOGE("Can't configure output window!");
        return;
    }
    if (!camera_device_)
        create_camera();
    // configure output window size and format
    ...
    if (!image_reader_ && !session_output_) {
        create_image_reader();
        create_session();
    }
}

```

In the beginning, we checked that there are all required resources: the camera ID, the `android_app` object, and that this object has a pointer to the application window. Then, we created a camera manager object and opened a camera device. Using the camera manager, we got the camera sensor orientation to configure the appropriate width and height for the application window. Also, using values for image capture width and height, we configured the window dimensions, as follows:

```

ACameraMetadata *metadata_obj=nullptr;
ACameraManager_getCameraCharacteristics(camera_mngr_,
                                         camera_id_.c_str(),
                                         &metadata_obj);

ACameraMetadata_const_entry entry;
ACameraMetadata_getConstEntry(metadata_obj,
                             ACAMERA_SENSOR_ORIENTATION,
                             &entry);
orientation_ = entry.data.i32[0];

```

```
bool is_horizontal = orientation_ == 0 || orientation_ == 270;
auto out_width = is_horizontal ? width_ : height_;
auto out_height = is_horizontal ? height_ : width_;
ANativeWindow_setBuffersGeometry(android_app_->window,
                                 out_width,
                                 out_height,
                                 WINDOW_FORMAT_RGBA_8888);
```

Here, we used the `ACameraManager_getCameraCharacteristics` function to get the camera metadata characteristics object. Then, we read the `ACAMERA_SENSOR_ORIENTATION` property with the `ACameraMetadata_getConstEntry` function. After, we chose the appropriate width and height order based on the orientation used with the `ANativeWindow_setBuffersGeometry` function to set application output window dimensions and rendering buffer format.

The format we set is 32-bit **RGBA (Red Green Blue Alpha)**. If the orientation is horizontal, we swapped the width and height. The exact width and height values are defined in the header file and are equal to 800 for height and 600 for width in portrait mode. This orientation handling is very simple and is needed only to work with output window buffers correctly. Previously, we disabled the landscape mode for our application so we will ignore the camera sensor orientation in the camera image decoding.

At the end of the `configure_resources` method, we created the camera reader object and initialized the capturing pipeline.

Image-capturing pipeline construction

Previously, we saw that before the capturing pipeline initialization, we created the image reader object. It's done in the `create_image_reader` method, as follows:

```
void ObjectDetector::create_image_reader() {
    constexpr int32_t MAX_BUF_COUNT = 4;
    auto status = AImageReader_new(
        width_, height_, IMAGE_FORMAT_YUV_420_888,
        MAX_BUF_COUNT, &image_reader_);
    ASSERT(image_reader_ && status == AMEDIA_OK,
           "Failed to create AImageReader");
}
```

We used `AImageReader_new` to create the `AImageReader` object with a particular width and height, the YUV format, and four image buffers. The width and height values we used were the same that were used for the output window dimensions configuration. The YUV format was used because it's the native image format for most camera devices. Four image buffers were used to make image capturing slightly independent from their processing. It means that the image reader will fill one image buffer with camera data while we are reading another buffer and processing it.

The capture session initialization is a complex process that requires several objects' instantiation and their connection with each other. The `create_session` method implements it as follows:

```

void ObjectDetector::create_session() {
    ANativeWindow* output_native_window;
    AImageReader_getWindow(image_reader_,
                           &output_native_window);
    ANativeWindow_acquire(output_native_window);
    ACaptureSessionOutputContainer_create(&output_container_);
    ACaptureSessionOutput_create(output_native_window,
                                &session_output_);
    ACaptureSessionOutputContainer_add(output_container_,
                                       session_output_);
    ACameraOutputTarget_create(output_native_window,
                               &output_target_);
    ACameraDevice_createCaptureRequest(
        camera_device_, TEMPLATE_PREVIEW, &capture_request_);
    ACaptureRequest_ addTarget(capture_request_,
                               output_target_);
    ACameraDevice_createCaptureSession(camera_device_,
                                       output_container_,
                                       &session_callbacks,
                                       &capture_session_);
    // Start capturing continuously
    ACameraCaptureSession_setRepeatingRequest(capture_session_,
                                              nullptr,
                                              1,
                                              &capture_request_,
                                              nullptr);
}

```

We started with getting a native window from the image reader object and acquiring it. The window acquisition means that we took the reference to the window and the system should not delete it. This image reader window will be used as output for the capturing pipeline, so camera images will be drawn into it.

Then, we created the session output object and the container for the session output. The capturing session can have several outputs and they should be placed into a container. Every session output is a connection object for a concrete surface or a window output; in our case, it's the image reader window.

Having configured session outputs, we created the capture request object and made sure that its output target was the image reader window. We configured the capture request for our opened camera device and the preview mode. After that, we instantiated the capturing session object and pointed it to the opened camera device, which had the container with the outputs we created earlier.

Finally, we started the capturing by setting the repeated request for the session. The connection between the session and capture request is the follows: we created the capturing session that was configured with a list of possible outputs, and the capture request specifies what surfaces will actually be used. There can be several capture requests and several outputs. In our case, we have a single capture request with a single output that will be continuously repeated. So, in general, we will capture real-time pictures for the camera like a video stream. The following picture shows the logical scheme of an image data flow in the capturing session:

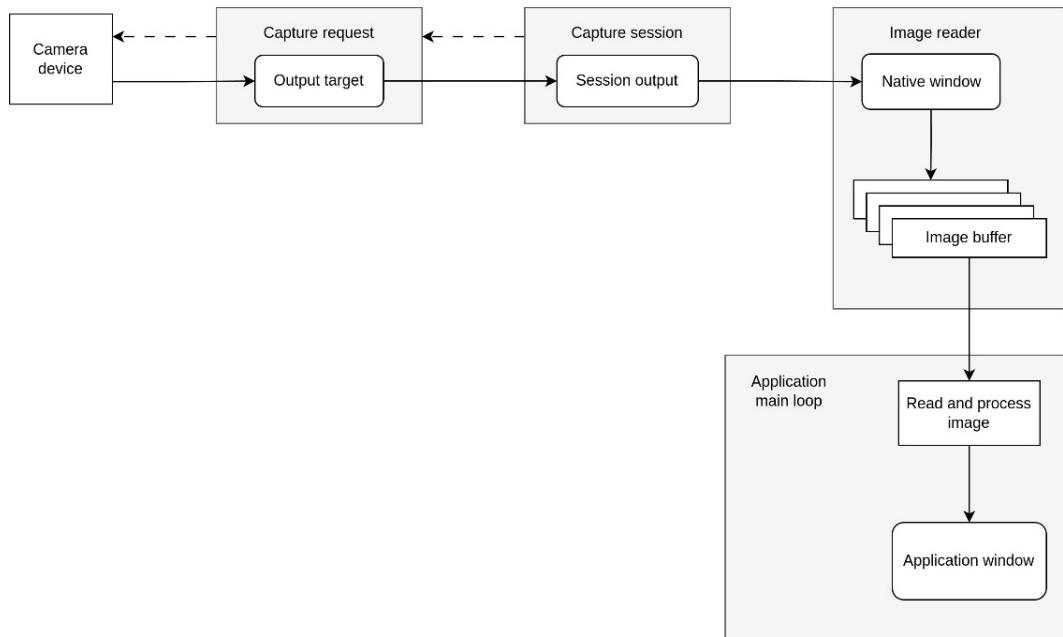


Figure 14.1 – The logical data flow in a capturing session

This is not the actual data flow scheme but the logical one that shows how the capture session objects are connected. The dotted line shows the request path and the solid line shows the logical image data path.

The capture image and the output window buffer management

When we discussed the main application loop, we mentioned the `draw_frame` method, which is called in this loop after command processing. This method is used to take a captured image from the image reader object, then detect objects on it and draw the detection results in the application window. The following code snippet shows the `draw_frame` method implementation:

```

void ObjectDetector::draw_frame() {
    if (image_reader_ == nullptr)
        return;
  
```

```

AImage *image = nullptr;
auto status = AImageReader_acquireNextImage(image_reader_, &image);

if (status != AMEDIA_OK) {
    return;
}
ANativeWindow_acquire(android_app_->window);
ANativeWindow_Buffer buf;
if (ANativeWindow_lock(android_app_->window,
                      &buf,
                      nullptr) < 0) {
    AImage_delete(image);
    return;
}
process_image(&buf, image);
AImage_delete(image);
ANativeWindow_unlockAndPost(android_app_->window);
ANativeWindow_release(android_app_->window);
}

```

We acquired the next image received by the image reader object. Remember that we initialized it to have four image buffers. So, we acquire images from these buffers one by one in the main loop, and while we process one image, the capturing session fills another one that has already been processed. It's done in a circular manner. Having the image from a camera, we acquired and locked the application window, but if the lock fails, we delete the current image reference, stop processing, and go to the next iteration of the main loop. Otherwise, if we successfully lock the application window, we process the current image, detect objects on it, and draw detection results into an application window—this is done in the `process_image` method. This method takes the `AImage` and the `ANativeWindow_Buffer` objects.

When we lock the application window, we get the pointer to the internal buffer that will be used for drawing. After we process the image and draw results, we unlock the application window to make its buffer available for the system, release the reference to the window, and delete the reference to the image object. So, this method is mostly about resource management, and the real image processing is done in the `process_image` method, which we will discuss in the following subsection.

The captured image processing

The `process_image` method implements the following tasks:

1. Convert Android YUV image data into the OpenCV matrix.
2. Dispatch the image matrix to the YOLO object detector.
3. Draw detection results into the OpenCV matrix.
4. Copy the OpenCV results matrix into the RGB (red, blue, green) window buffer.

Let's see implementations for these tasks one by one. The `process_image` method signature looks as follows:

```
void ObjectDetector::process_image(
    ANativeWindow_Buffer* buf,
    AImage* image);
```

This method takes the application window buffer object for results drawing and the image object for actual processing. To be able to process an image, we have to convert it into some appropriate data-structure format; in our case, this is the OpenCV matrix. We start with image format properties checking as follows:

```
int32_t src_format = -1;
AImage_getFormat(image, &src_format);
ASSERT(AIMAGE_FORMAT_YUV_420_888 == src_format,
       "Unsupported image format for displaying");
int32_t num_src_planes = 0;
AImage_getNumberOfPlanes(image, &num_src_planes);
ASSERT(num_src_planes == 3,
       "Image for display has unsupported number of planes");
int32_t src_height;
AImage_getHeight(image, &src_height);
int32_t src_width;
AImage_getWidth(image, &src_width);
```

We checked that the image format is YUV (Luminance (Y), blue luminance (U), and red luminance (V)) and the image has three planes, so we can proceed with its conversion. Then, we got image dimensions, which will be used later. After that, we verified the input data we extracted from the YUV plane data as follows:

```
int32_t y_stride{0};
AImage_getPlaneRowStride(image, 0, &y_stride);
int32_t uv_stride1{0};
AImage_getPlaneRowStride(image, 1, &uv_stride1);
int32_t uv_stride2{0};
AImage_getPlaneRowStride(image, 1, &uv_stride2);
uint8_t *y_pixel{nullptr}, *uv_pixel1{nullptr}, *uv_pixel2{nullptr};
int32_t y_len{0}, uv_len1{0}, uv_len2{0};
AImage_getPlaneData(image, 0, &y_pixel, &y_len);
AImage_getPlaneData(image, 1, &uv_pixel1, &uv_len1);
AImage_getPlaneData(image, 2, &uv_pixel2, &uv_len2);
```

We got strides, data sizes, and pointers to the actual YUV plane data. In this format, the image data is split into three components: luma (y), representing brightness, and two chroma components (u and v), which represent color information. The y component is usually stored at full resolution, while the u and v components may be subsampled. This allows for more efficient storage and transmission of video data. The Android YUV image uses the half-sized resolution for u and v. The strides will allow us to correctly access the row data in the plane buffers; these strides depend on the image resolution and a data memory layout.

Having the YUV plane data and its strides and lengths, we convert them into OpenCV matrix objects, as follows:

```
cv::Size actual_size(src_width, src_height);
cv::Size half_size(src_width / 2, src_height / 2);
cv::Mat y(actual_size, CV_8UC1, y_pixel, y_stride);
cv::Mat uv1(half_size, CV_8UC2, uv_pixel1, uv_stride1);
cv::Mat uv2(half_size, CV_8UC2, uv_pixel2, uv_stride2);
```

We created the two `cv::Size` objects to store the original image size for the Y plane and the half size for the u and v planes. Then, we used these sizes, pointers to data, and strides to create an OpenCV matrix for every plane. We didn't copy actual data into the OpenCV matrix objects; they will use data pointers that were passed for initialization. Such a view-creation approach saves memory and computational resources. The y-plane matrix has the 8-bit single-channel type but the u and v matrices have the 8-bit 2-channel type. We can use these matrices with the OpenCV `cvtColorTwoPlane` function to convert them into the RGBA format as follows:

```
cv::mat rgba_img_;
...
long addr_diff = uv2.data - uv1.data;
if (addr_diff > 0) {
    cvtColorTwoPlane(y, uv1, rgba_img_, cv::COLOR_YUV2RGB_NV12);
} else {
    cvtColorTwoPlane(y, uv2, rgba_img_, cv::COLOR_YUV2RGB_NV21);
}
```

We used the address difference to determine the ordering of the u and v planes: a positive difference indicates the NV12 format, while a negative difference indicates the NV21 format. NV12 and NV21 are types of the YUV format that differ in the order of the u and v components in the chroma plane. In NV12, the u component precedes the v component, while in NV21, it's the opposite. Such plane ordering plays a role in memory consumption and image processing performance, so the choice of which to use depends on the actual task and project. Also, the format can depend on the actual camera device, which is why we added this detection.

The `cvtColorTwoPlane` function takes the y-plane and uv-plane matrices as input arguments and outputs the RGBA image matrix into the `rgba_img_` variable. The last argument is the flag that tells the function what actual conversion it should perform. Now, this function can convert only YUV formats into RGB or RGBA formats.

As we said before, our application works only in portrait mode, but to make the image look normal, we need to rotate it as follows:

```
cv::rotate(rgba_img_, rgba_img_, cv::ROTATE_90_CLOCKWISE);
```

Android camera sensors return camera images rotated even if we fixed our orientation, so we used the `cv::rotate` function to make it look vertical.

Having prepared the RGBA image, we pass it to the YOLO object detector and get the detection results. For every result item, we draw rectangles and labels on the image matrix we have already used for detection. These steps are implemented as follows:

```
auto results = yolo_->detect(rgba_img_);
for (auto& result : results) {
    int thickness = 2;
    rectangle(rgba_img_, result.rect.tl(), result.rect.br(),
              cv::Scalar(255, 0, 0, 255), thickness,
              cv::LINE_4);
    cv::putText(rgba_img_, result.class_name,
               result.rect.tl(), cv::FONT_HERSHEY_DUPLEX,
               1.0, CV_RGB(0, 255, 0), 2);
}
```

We called the `detect` method of the YOLO object and got the `results` container. This method will be discussed later. Then, for each item in the container, we draw a bounding box and a text label for the detected object. We used the OpenCV `rectangle` function with the `rgba_img_` destination image argument. Also, the text was rendered into the `rgba_img_` object. The detection result is the structure defined in the `yolo.h` header file as follows:

```
struct YOLOResult {
    int class_index;
    std::string class_name;
    float score;
    cv::Rect rect;
};
```

So, a detection result has the class index and name properties, the model confidence score, and the bounding box in the image coordinates. For our results visualization, we used only rectangle and class name properties.

The last task that the `process_image` method does is to render the resulting image into the application window buffer. It's implemented as follows:

```
cv::Mat buffer_mat(src_width,
                    src_height,
                    CV_8UC4,
                    buf->bits,
                    buf->stride * 4);
rgba_img_.copyTo(buffer_mat);
```

We created the OpenCV `buffer_mat` matrix to wrap the given window buffer. Then, we simply used the OpenCV `copyTo` method to put the RGBA image with rendered rectangles and class labels into the `buffer_mat` object. `buffer_mat` is the OpenCV view for the Android window buffer. We created it to follow the window buffer format we configured in the `configure_resources` method, the `WINDOW_FORMAT_RGBA_8888` format. So, we created the OpenCV matrix with the 8-bit 4-channel type and used the buffer stride information to satisfy memory layout access. Such a view allows us to write less code and use OpenCV routines for memory management.

We discussed the main facade of our object detection application and in the following subsections, we will discuss details of how the YOLO model inference is implemented and how its results are parsed into the `YOLOResult` structures.

The YOLO wrapper initialization

There is only the constructor and the `detect` method in the `YOLO` class public API. We already saw that the `YOLO` object is initialized in the `ObjectDetector` class constructor, and the `detect` method is used in the `process_image` method. The `YOLO` class constructor takes only the asset manager object as a single argument and is implemented as follows:

```
YOLO::YOLO(AAssetManager* asset_manager) {
    const std::string model_file_name = "yolov5s.torchscript";
    auto model_buf = read_asset(asset_manager,
                                model_file_name);
    model_ = torch::jit::_load_for_mobile(
        std::make_unique<ReadAdapter>(model_buf));
    const std::string classes_file_name = "classes.txt";
    auto classes_buf = read_asset(asset_manager,
                                  classes_file_name);
    VectorStreamBuf<char> stream_buf(classes_buf);
    std::istream is(&stream_buf);
    load_classes(is);
}
```

Remember that we added the `yolov5s.torchscript` and the `classes.txt` files to the `assets` folder of our project. These files can be accessed in the application with the `AAssetManager` class object; this object was taken from the `android_app` object in the `android_main` function. So, in the constructor, we loaded the model binary and classes list file with a call to the `read_asset` function. Then, the model binary data was used to load and initialize the PyTorch script module with the `torch::jit::load_for_mobile` function.

Notice that the scripted model should be saved with optimization for mobile and loaded with the corresponding function. When PyTorch for mobile was compiled, the regular `torch::jit::load` functionality was automatically disabled. Let's look at the `read_asset` function that reads assets from the application bundle as `std::vector<char>` objects. The following code shows its implementation:

```
std::vector<char> read_asset(AAssetManager* asset_manager,
                           const std::string& name) {
    std::vector<char> buf;
    AAsset* asset = AAssetManager_open(
        asset_manager, name.c_str(), AASSET_MODE_UNKNOWN);
    if (asset != nullptr) {
        LOGI("Open asset %s OK", name.c_str());
        off_t buf_size = AAsset_getLength(asset);
        buf.resize(buf_size + 1, 0);
        auto num_read = AAsset_read(
            asset, buf.data(), buf_size);
        LOGI("Read asset %s OK", name.c_str());
        if (num_read == 0)
            buf.clear();
        AAsset_close(asset);
        LOGI("Close asset %s OK", name.c_str());
    }
    return buf;
}
```

There are four Android framework functions that we used to read an asset from the application bundle. The `AAssetManager_open` function opened the asset and returned the not null pointer to the `AAsset` object. This function assumes that the path to the asset is in the file path format and that the root of this path is the `assets` folder. After we opened the asset, we used the `AAsset_getLength` function to get the file size and allocated the memory for `std::vector<char>` with the `std::vector::resize` method. Then, we used the `AAsset_read()` function to read the whole file to the `buf` object.

This function does the following:

- It takes the pointer to the asset object to read from
- It takes the `void*` pointer to the memory buffer to read in
- It measures the size of the bytes to read

So, as you can see, the assets API is pretty much the same as the standard C library API for file operations. When we'd finished working with the asset object, we used the `AAsset_close` function to notify the system that we didn't need access to this asset anymore. If your assets are in the `.zip` archive format, you should check the number of bytes returned by the `AAsset_read` function because the Android framework reads archives chunk by chunk.

You might see that we didn't pass the vector of chars directly to the `torch::jit::load_for_mobile` function. This function doesn't work with standard C++ streams and types; instead, it accepts a pointer to an object of the `caffe2::serialize::ReadAdapterInterface` class. The following code shows how you to make the concrete implementation of the `caffe2::serialize::ReadAdapterInterface` class, which wraps the `std::vector<char>` object:

```
class ReadAdapter
    : public caffe2::serialize::ReadAdapterInterface {
public:
    explicit ReadAdapter(const std::vector<char>& buf)
        : buf_(&buf) {}
    size_t size() const override { return buf_->size(); }
    size_t read(uint64_t pos,
               void* buf,
               size_t n,
               const char* what) const override {
        std::copy_n(buf_->begin() + pos, n,
                   reinterpret_cast<char*>(buf));
        return n;
    }
private:
    const std::vector<char>* buf_;
};
```

The `ReaderAdapter` class overrides two methods, `size` and `read`, from the `caffe2::serialize::ReadAdapterInterface` base class. Their implementations are pretty obvious: the `size` method returns the size of the underlying vector object, while the `read` method copies the `n` bytes (chars) from the vector to the destination buffer with the standard algorithm function, that is, `std::copy_n`.

To load class information, we used the `VectorStreamBuf` adapter class to convert `std::vector<char>` into the `std::istream` type object. It was done because the `YOLO::load_classes` method takes an object of the `std::istream` type. The `VectorStreamBuf` implementation is as follows:

```
template<typename CharT, typename TraitsT = std::char_traits<CharT>>
struct VectorStreamBuf : public std::basic_streambuf<CharT, TraitsT> {
    explicit VectorStreamBuf(std::vector<CharT>& vec) {
        this->setg(vec.data(), vec.data(),
                    vec.data() + vec.size());
    }
}
```

We inherited from the `std::basic_streambuf` class and in the constructor, we initialized the `streambuf` internal data with char values from the input vector. Then, we used an object of this adapter class as regular C++ input stream. You can see it in the `load_classes` method implementation, which is shown in the following snippet:

```
void YOLO::load_classes(std::istream& stream) {
    LOGI("Init classes start OK");
    classes_.clear();
    if (stream) {
        std::string line;
        std::string id;
        std::string label;
        size_t idx = 0;
        while (std::getline(stream, line)) {
            auto pos = line.find_first_of(':');
            id = line.substr(0, pos);
            label = line.substr(pos + 1);
            classes_.insert({idx, label});
            ++idx;
        }
    }
    LOGI("Init classes finish OK");
}
```

The lines in `classes.txt` are in the following format:

```
[ID] space character [class name]
```

So, we read this file line by line and split each line at the position of the first space character. The first part of each line is the class identifier, while the second one is the class name. To match the model's evaluation result with the correct class name, we created the dictionary (map) object, where the key is the `id` value and the value is `label` (e.g., the class name)s.

The YOLO detection inference

The `detect` method of the `YOLO` class is the place where we do the actual object detection. This method takes the OpenCV matrix object that represents the RGB image as an argument and its implementation is as follows:

```
std::vector<YOLOResult> YOLO::detect(const cv::Mat& image) {
    constexpr int input_width = 640;
    constexpr int input_height = 640;
    cv::cvtColor(image, rgb_img_, cv::COLOR_RGBA2RGB);
    cv::resize(rgb_img_, rgb_img_,
               cv::Size(input_width, input_height));
    auto img_scale_x =
        static_cast<float>(image.cols) / input_width;
    auto img_scale_y =
        static_cast<float>(image.rows) / input_height;
    auto input_tensor = mat2tensor(rgb_img_);
    std::vector<torch::jit::IValue> inputs;
    inputs.emplace_back(input_tensor);
    auto output = model_.forward(inputs).toTuple() ->
        elements()[0].toTensor().squeeze(0);
    output2results(output, img_scale_x, img_scale_y);
    return non_max_suppression();
}
```

We defined constants that represent the width and height of the model input; it's 640×640 because the YOLO model was trained on images of this size. Using these constants, we resized the input image. Also, we removed the alpha channel and made the RGB image. We calculated scale factors for image dimensions, as they will be used to re-scale detected object boundaries to the original image size. Having scaled the image, we converted the OpenCV matrix into a PyTorch Tensor object using the `mat2tensor` function, whose implementation we will discuss later. The object type cast of the PyTorch Tensor object we added to the container of the `torch::jit::IValue` values was done automatically. There is a single element in this `inputs` container since the YOLO model takes a single RGB image input.

Then, we used the `forward` function of the YOLO model_ object to perform inference. The PyTorch API script modules return the `torch::jit::Tuple` type. So, we explicitly cast the returned `torch::jit::IValue` object to the tuple and took the first element. This element was cast to the PyTorch Tensor object and the batch dimension was removed from it with the `squeeze` method. So, we got the `torch::Tensor` type `output` object of size 25200 x 85. Here, 25200 is the number of detected objects and we will apply the non-max suppression algorithm to get the final reduced output. The 85 means 80 class scores, 4 bounding box locations (x, y, width, height), and 1 confidence score. The resulting tensor was parsed into the `YOLOResult` structures in the `output2results` method. As we said, we used the `non_max_suppression` method to select the best detection results.

Let's see the details of all the intermediate functions we used for inference.

Converting OpenCV matrix into torch::Tensor

The `mat2tensor` function converts an OpenCV `mat` object into a `torch::Tensor` object and is implemented as follows:

```
torch::Tensor mat2tensor(const cv::Mat& image) {
    ASSERT(image.channels() == 3, "Invalid image format");
    torch::Tensor tensor_image = torch::from_blob(
        image.data,
        {1, image.rows, image.cols, image.channels()},
        at::kByte);
    tensor_image = tensor_image.to(at::kFloat) / 255.;
    tensor_image = torch::transpose(tensor_image, 1, 2);
    tensor_image = torch::transpose(tensor_image, 1, 3);
    return tensor_image;
}
```

We used the `torch::from_blob` function to create the `torch Tensor` object from the raw data. The data pointer is what we took from the OpenCV object with the `data` property. The shape we used, `[HEIGHT, WIDTH, CHANNELS]`, follows the OpenCV memory layout where the last dimension is the channel number dimension. Then, we made tensor float and normalized it to the `[0, 1]` interval. PyTorch and the YOLO model use different shape layouts to `[CHANNELS, HEIGHT, WIDTH]`. So, we transpose the tensor channels appropriately.

Processing model output tensor

The next function we used is `output2results`, which converts the output Tensor object into the vector of the `YOLOResult` structures. It has the following implementation:

```
void YOLO::output2results(const torch::Tensor &output,
                           float img_scale_x,
                           float img_scale_y) {
    auto outputs = output.accessor<float, 2>();
    auto output_row = output.size(0);
    auto output_column = output.size(1);
    results_.clear();
    for (int64_t i = 0; i < output_row; i++) {
        auto score = outputs[i][4];
        if (score > threshold) {
            // read the bounding box
            // calculate the class id
            results_.push_back(YOLOResult{
                .class_index = cls,
                .class_name = classes_[cls],
                .score = score,
                .rect = cv::Rect(left, top, bw, bh),
            });
        }
    }
}
```

In the beginning, we used the `accessor<float, 2>` method of the `torch` Tensor object to get a very useful accessor for the tensor. This accessor allowed us to use the square brackets operator to access the elements in a multidimensional tensor. The number 2 means that the tensor is 2D. Then, we made a loop over tensor rows because every row corresponds to a single detection result. Inside the loop, we did the following steps:

1. We read the confidence score from the element with row index 4.
2. We continued result row processing if the confidence score was greater than the threshold.
3. We read the 0, 1, 2, 3 elements, which are the [x, y, width, height] coordinates of the bounding rectangle.
4. Using the previously calculated scale factors, we converted these coordinates into the [left, top, width, height] format.
5. We read the elements 5-84, which are class probabilities, and selected the class with the maximum value.
6. We created the `YOLOResult` structure with calculated values and inserted it into the `results_` container.

The bounding box calculation was done as follows:

```
float cx = outputs[i][0];
float cy = outputs[i][1];
float w = outputs[i][2];
float h = outputs[i][3];
int left = static_cast<int>(img_scale_x * (cx - w / 2));
int top = static_cast<int>(img_scale_y * (cy - h / 2));
int bw = static_cast<int>(img_scale_x * w);
int bh = static_cast<int>(img_scale_y * h);
```

The YOLO model returns X and Y coordinates for the center of a rectangle so we converted them into the image(screen) coordinate system: to the top-left point.

The class ID selection was implemented as follows:

```
float max = outputs[i][5];
int cls = 0;
for (int64_t j = 0; j < output_column - 5; j++) {
    if (outputs[i][5 + j] > max) {
        max = outputs[i][5 + j];
        cls = static_cast<int>(j);
    }
}
```

We used the loop over the last elements that represent 79 class probabilities to select the index of the maximum value. This index was used as a class ID.

NMS and IoU

Non-Maximum Suppression (NMS) and **Intersection over Union (IoU)** are two key algorithms used in YOLO for refining and filtering the output predictions to get the best results.

NMS is used to suppress or eliminate duplicate detections that overlap with each other. It works by comparing the predicted bounding boxes from the network and removing those that have high overlaps with others. For example, if there are two bounding boxes predicted for the same object, NMS will keep only the one with the highest confidence score and discard the rest. The following picture shows how NMS works:

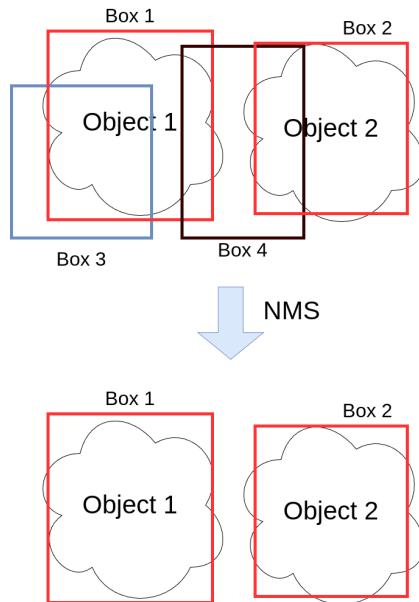


Figure 14.2 – NMS

IoU is another algorithm used in conjunction with NMS to measure the overlap between bounding boxes. IoU calculates the ratio of an intersection area to a union area between two boxes. The **intersection area** refers to the area where boxes overlap. The **union area**, on the other hand, refers to the total area covered by both boxes. The IoU value ranges from 0 to 1, where 0 means no overlap, and 1 indicates perfect overlap. The following picture shows how IoU works:

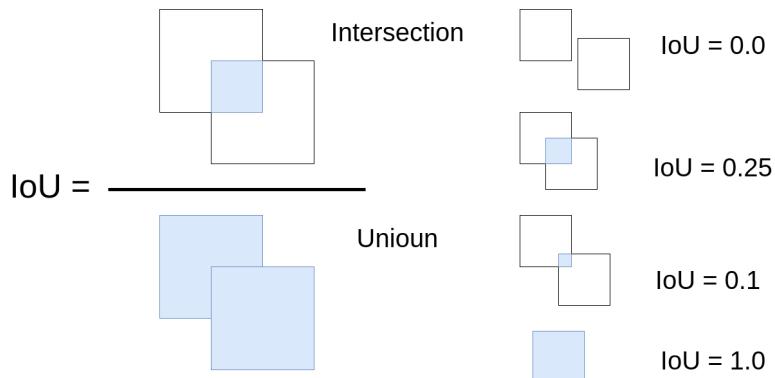


Figure 14.3 – IoU

We implemented NMS in the `non_max_suppression` method as follows:

```
std::vector<YOLOResult> YOLO::non_max_suppression() {
    // do an sort on the confidence scores, from high to low.
    std::sort(results_.begin(), results_.end(), [](
        auto &r1, auto &r2) {
            return r1.score > r2.score;
    ));
    std::vector<YOLOResult> selected;
    std::vector<bool> active(results_.size(), true);
    int num_active = static_cast<int>(active.size());
    bool done = false;
    for (size_t i = 0; i < results_.size() && !done; i++) {
        if (active[i]) {
            const auto& box_a = results_[i];
            selected.push_back(box_a);
            if (selected.size() >= nms_limit)
                break;
            for (size_t j = i + 1; j < results_.size(); j++) {
                if (active[j]) {
                    const auto& box_b = results_[j];
                    if (IOU(box_a.rect, box_b.rect) > threshold) {
                        active[j] = false;
                        num_active -= 1;
                        if (num_active <= 0) {
                            done = true;
                            break;
                        }
                    }
                }
            }
        }
    }
    return selected;
}
```

At first, we sorted all detection results by confidence score in descending order. We marked all results as active. If a detection result is active, then we can compare another result with it, otherwise, the result is already suppressed. Then, every active detection result was sequentially compared with the following active results; remember that the container is sorted. The comparison was done by calculating the IoU value for bounding boxes and comparing the IoU value with a threshold. If the IoU value is greater than the threshold, we marked the result with a lower confidence value as non-active; we suppressed it. So, we defined the nested comparison loop. In the outer loop, we ignored suppressed results too. Also, this nested loop has the check for the maximum allowed number of results; refer to the use of the `nms_limit` value.

The IoU algorithm for two bounding boxes is implemented in the `IOU` function as follows:

```
float IOU(const cv::Rect& a, const cv::Rect& b) {
    if (a.empty() <= 0.0)
        return 0.0f;
    if (b.empty() <= 0.0)
        return 0.0f;
    auto min_x = std::max(a.x, b.x);
    auto min_y = std::max(a.y, b.y);
    auto max_x = std::min(a.x + a.width, b.x + b.width);
    auto max_y = std::min(a.y + a.height, b.y + b.height);
    auto area = std::max(max_y - min_y, 0) *
                std::max(max_x - min_x, 0);
    return static_cast<float>(area) /
        static_cast<float>(a.area() + b.area() - area);
}
```

At first, we checked bounding boxes for emptiness; if one of the boxes is empty, the IoU value is zero. Then, we calculated the intersection area. This was done by finding the minimum and maximum values for X and Y, considering both bounding boxes, and then taking the product of the difference between these values. The union area was calculated by summing the areas of two bounding boxes minus the intersection area. You can see this calculation in the return statement where we calculated the area's ratio.

Together, NMS and IoU help improve the accuracy and precision of YOLO by discarding false positives and ensuring that only relevant detections are included in the final output.

In this section, we looked at the implementation of object detection applications for the Android system. We learned how to export a pre-trained model from a Python program as a PyTorch script file. Then, we delved into developing a mobile application with Android Studio IDE and the mobile version of the PyTorch C++ library. In the following figure, you can see an example of the application output window:



Figure 14.4 – Object detection application output

In this figure, you can see that our application successfully detected a laptop and a computer mouse in front of the smartphone camera. Every detection result was marked with the bounding box and corresponding label.

Summary

In this chapter, we discussed how to deploy ML models, especially neural networks, to mobile platforms. We examined that, on these platforms, we usually need a customized build of the ML framework that we used in our project. Mobile platforms use different CPUs, and sometimes, they have specialized neural network accelerator devices, so you need to compile your application and ML framework in regard to these architectures. These architectures differ from development environments, and you often use them for two different purposes. The first case is to use powerful machine configuration with GPUs to accelerate the ML training process, so you need to build your application while taking the use of one or multiple GPUs into account. The other case is using a device for inference only. In this case, you typically don't need a GPU at all because a modern CPU can, in many cases, satisfy your performance requirements.

In this chapter, we developed an object detection application for the Android platform. We learned how to connect the Kotlin module with the native C++ library through JNI. Then, we examined how to build the PyTorch C++ library for Android using the NDK and saw what limitations there are to using the mobile version.

This was the last chapter of the book; I hope you have enjoyed this book and found it helpful in your journey to mastering the use of C++ for ML. I hope that by now, you have gained a solid understanding of how to leverage the power of C++ to build robust and efficient ML models. Throughout the book, I have aimed to provide clear explanations of complex concepts, practical examples, and step-by-step guides to help you get started with C++ for ML. I have also included tips and best practices to help you avoid common pitfalls and optimize your models for performance. I want to remind you that the possibilities are endless when it comes to using C++ for ML.

Whether you are a beginner or an experienced developer, there is always something new to learn and explore. With that in mind, I encourage you to continue to push your boundaries and experiment with different approaches and techniques. The world of ML is constantly evolving, and by staying up to date with the latest trends and developments, you can stay ahead of the curve and build cutting-edge models that can solve complex problems.

Thank you again for choosing my book and for taking the time to learn about using C++ for ML. I hope that you find it to be a valuable resource and that it helps you on your journey toward becoming a skilled and successful ML developer.

Further reading

- PyTorch C++ API: <https://pytorch.org/cppdocs/>
- Documentation for app developers: <https://developer.android.com/develop>
- Android NDK: <https://developer.android.com/ndk>
- PyTorch guides for mobile development: <https://pytorch.org/mobile/android/>
- PyTorch guide for optimized mobile script exporting: https://pytorch.org/tutorials/recipes/script_optimized.html
- OpenCV Android SDK tutorial: https://docs.opencv.org/4.x/d5/df8/tutorial_dev_with_OCV_on_Android.html
- ExcuTorch – a new framework for running PyTorch on embedded devices: <https://pytorch.org/executorch/stable/index.html>

Index

A

absolute error tolerance parameter 160
activation functions 313

- hyperbolic tangent 317-319
- linear activation function 315
- properties 319
- sigmoid activation function 316
- stepwise activation function 314, 315

AdaBoost loss 274
Adaptive Boosting (AdaBoost) 271

- using, with mlpack 286

adjusted R squared 76
Advanced RISC Machines (ARM) 432
Alternating Least Squares (ALS) 242
Android

- object detection, developing 430

Android Application Package (APK) 444
Android Studio project 434-436

- Kotlin part 436
- native C++ part 440

anomaly detection 133

- applications 134-136
- C++ libraries, using 145
- Dlib library, using 153

anomaly detection, approaches

- density estimation approach 141
- density estimation tree (DET) 145
- isolation forest 139, 140
- KDE approach 143, 144
- Local Outlier Factor method 137, 138
- multivariate Gaussian distribution 141-143
- One-Class Support Vector Machine 140
- statistical tests 136, 137

anomaly model formula 141
application programming

- interface (API)** 12, 47, 205, 280

applied statistics 204
approaches, to ensemble construction

- bagging 268
- boosting 268
- random forest 268
- stacking 268

Area Under the Receiver Operating Characteristic Curve (AUC-ROC) 79, 80, 136
ArrayFire

- used, for solving linear regression tasks 32, 33
- using 21-24

ArrayFire library

- working with 57

artificial intelligence (AI) 204
artificial neural networks (ANNs) 6
ATen 338
autoencoders 182
axon 299

B

backpropagation method
example 306-311
modes 305
problems 306
used, for training 303, 304
backpropagation method modes
batch mode 305
mini-batch mode 305
stochastic mode 305
bagging approach 268
using, for creating ensembles 268-275
batch mode 305
batch normalization 321
Bernoulli loss 274
bias 81-83, 300
bidirectional encoder representations from transformers (BERT) 359
dataset loader, implementing 366-370
model and vocabulary, exporting 360-362
model class, implementing 371-373
model, training 373-375
tokenizer, implementing 362-366
used, for sentiment analysis model 360
biological neuron 299
Blaze library
used, for solving linear regression tasks 31
using 17-20
working with 57
blind signal separation (BSS) 172
boosting approach 268
bootstrap samples 268

C

C++
used, for data plotting 129-131
C++ data structures
data formats, parsing into 38-40
matrix and tensor objects, initializing 56-58
C++ libraries
ML model serialization APIs 380
used, for creating ensembles 280
using, to create neural networks 329
C++ libraries examples, for anomaly detection
Dlib library 153
isolation forest algorithms 146-153
central processing units (CPUs) 8
centroid 174
Chebyshev distance
measuring 105
Chinese Whispers algorithm
using, with Dlib library 127-129
class definition file
reading 404, 405
classes 203
classification 203, 204
classification algorithm metrics 76
accuracy 77
AUC-ROC 79
F-score 79
Log-Loss 81
precision 78
recall 78
classification algorithms 74
classification methods 205
kNN 213-215
KRR 208, 209
logistic regression 205-208
multi-class classification 215, 216
SVM 209-213

-
- classification task** 135
 applicable areas 204
 solving, on artificial datasets 217, 218
- classifiers** 204
- clustering** 103
 distance, measuring 104
 uses 106
- clustering algorithms**
 distance-based clustering algorithms 107
 graph theory-based clustering algorithms 108, 109
 hierarchical clustering algorithms 110-112
 model-based clustering algorithms 112, 113
 partition-based clustering algorithms 107
 spectral clustering algorithms 109
 types 106
- Clustering Large Applications based on RANdomized Search (CLARANS)** 108
- coefficient of determination** 75
- cold start problem** 246, 247
- collaborative filtering method** 248-252
- column-major ordering** 11
- command-line interface (CLI)** 93
- Comma-Separated Values (CSV)** 37, 38
- Common Objects in Context (COCO)** 45
- computer vision (CV)** 58, 88
- Condorcet's jury theorem** 266
- confidence interval (CI)** 246
- confusion matrix** 76, 77
- content-based recommendations** 241
- convolutional networks**
 architecture 327
 convolution operator 323-325
 exploring 323
 pooling operation 325, 326
 receptive field 326
- convolution kernel** 323
- convolution operator** 323, 325
- coordinate list format** 260
- Cosine distance** 245
- cosine proximity loss function** 313
- create_model function** 387
- cross-entropy loss function** 312
- cross-validation** 89
 K-fold cross-validation 89
- CSV files**
 preprocessing 42, 43
 reading, with Dlib library 44
 reading, with Fast-CPP-CSV-Parser library 40-42
 reading, with MLPack library 43
- curse of dimensionality (CoD)** 145
- ## D
- data augmentation** 88
- data formats, parsing into C++**
 data structures 38-40
 CSV files, preprocessing 42, 43
 CSV files, reading with Dlib library 44, 45
 CSV files, reading with Fast-CPP-CSV-Parser library 40-42
 CSV files, reading with MLPack library 43, 44
 HDF5 files, reading with HighFive library 53-56
 HDF5 files, writing with HighFive library 53-56
 JSON files, reading with Nlohmann json library 45-52
- data normalization** 66, 67
 centering (mean-centering) 246
 double standardization 246
 standardization (z-score) 246
 with Dlib library 69
 with Eigen 67
 with Flashlight 69, 70
 with MLPack 68

- DBSCAN**
using, with mlpack library 117
- decision boundary** 140
- decision tree algorithm**
overview 276-278
- decoder part** 357
using, separately 359, 360
- deep learning** 205, 328
using, examples 328
- dendrites** 299
- dendrogram** 110
- density-based clustering algorithms** 111, 112
- Density-Based Spatial Clustering of Applications with Noise (DBSCAN)** 111
- density estimation approach**
for anomaly detection 141
- density estimation tree (DET) algorithm** 145
for anomaly detection 145
- dimension-reduction algorithms**
linear methods, exploring 170
non-linear methods 177
using, with C++ libraries 182, 183
using, with Dlib library 183
using, with Tapkee library 191
- dimension-reduction methods** 170
feature selection methods 169
overview 168
- discriminant analysis problems** 204
- distance-based clustering algorithms** 107
- distributed stochastic neighbor embedding** 180, 181
- Dlib library** 380
Chinese Whispers algorithm,
using with 127-129
deinterleaving 65, 66
examples 281-283
hierarchical clustering, using with 123
- K-means clustering, using with 119-121
LDA 189, 190
linear regression, using with 34
Newman modularity-based graph clustering , using with 125
PCA 183, 184
Sammon mapping 190, 191
spectral clustering, using with 121, 123
using 24-26
used, for creating neural networks 329-331
used, for data normalization 69
used, for manipulating images 61-63
used, for reading CSV files 44
using 183, 224
using, with KRR 224-226
using, with SVM 227, 229
working with 57
- Dlib library, for anomaly detection**
multivariate Gaussian model 155, 156
OCSVM 153, 154
- Document Object Model (DOM)** 46
- dropout regularization** 320
- dying ReLU problem** 319
- E**
- each-against-each strategy** 216
- Eigen**
used, for data normalization 67
used, for solving linear regression tasks 30, 31
using 12-14
- Eigen library**
using 253-260
working with 56
- element-wise operations** 9
- EM algorithms**
using, with mlpack library 114, 115

-
- EMFit classes** 114
encoder part 356
 using, separately 359, 360
ensemble learning
 bagging approach, using 268-270
 gradient boosting method, using 271-274
 overview 266, 267
 random forest method, using 276-279
 stacking approach, using 275, 276
ensembles 265
ensembles of classifiers
 computational 267
 representative 267
 statistical 267
epoch 82
error backpropagation method 303
Euclidean distance
 measuring 104
Expectation-Maximization (EM) algorithm 112
Expectation-Maximization with Fit (EMFit) algorithm 157
experiment features
 artifacts location 412
 name 412
 tags 412
experiment tracking
 integrating, into linear regression
 training 418-421
 MLflow ‘s REST API, using for 411
 process 421-427
explicit ratings 240
extreme-value analysis 136
- F**
- factor analysis method** 175, 176, 198, 199
factorization algorithms 244
factor load 176
- False Positive Rate (FPR)** 79
Fast-CPP-CSV-Parser library
 used, for reading CSV files 40-42
feedforward network (FFN) 303, 332
fine-tuning 353
Flashlight
 used, for creating neural networks 335-337
 used, for data normalization 69, 70
Flashlight library 384
 using 229
 using, with logistic regression 229-232
 using, with logistic regression
 and kernel trick 232, 233
F-score 79
fully connected layer 303
- G**
- Gaussian** 141
Gaussian loss 273
Gaussian Mixture Models (GMM) 114, 157
generalization 275
generalized additive model (GAM) 274
generalized boosting machine (GBM) 275
general-purpose graphics processing units (GPUs) 8
generative pre-trained transformer (GPT) 360
GetInputCount method 399
GetInputNameAllocated method 399
GetInputTypeInfo method 399
GetOutputCount method 399
GetOutputNameAllocated method 399
GetOutputTypeInfo method 399
GMM algorithms
 using, with mlpack library 114, 115
gradient boosted decision trees (GBDT) 274
gradient boosted regression trees (GBRT) 275

gradient boosting machine (GBM) 274
gradient descent (GD) 28, 66, 251, 267
graphics processing unit (GPU) 8, 58
graph theory-based clustering
 algorithms 108, 109
grid search 90
grid search techniques
 cross-validation 89
 Dlib example 98, 99
 grid search 90
 MLPack example 91-93
 Optuna, with Flashlight example 93-97
 using, for model selection 89-99

H

HDF5 files
 reading and writing , with
 HighFive library 53-56
He initialization method
 using 322
hierarchical clustering
 using, with Dlib library 123-125
hierarchical clustering algorithms 110, 111
 bottom-up algorithms 110
 metrics or linkage criteria 110
 top-down-based algorithms 110
Hierarchical Data Format v5 (HDF5) 37, 39
high bias 82, 83
HighFive library
 used, for reading and writing
 HDF5 files 53-56
high-frequency filtering 244
hinge function 313
hinge loss function 313
hyperbolic tangent 317-319
hyperparameters 5
hyperplane 140
hypothesis 267

I

identity matrix 11
image classification, with LeNet
 architecture 337, 338
 dataset files, reading 341, 342
 image file, reading 343, 344
 neural network, defining 344-347
 neural network, training 347, 348-351
 training dataset, reading 339-341
images
 loading, into onnxruntime tensors 402-404
 manipulating, with OpenCV and
 Dlib libraries 58, 59
 transforming, into matrix or tensor
 objects of various libraries 64
implicit ratings 240
impurity function 277, 280
independent component analysis
 (ICA) 172, 173
internal covariance shift 321
inter-process communication (IPC) 93
intersection area 462
Intersection over Union (IoU) 461
inverting operation 11
isolation forest 139
 for anomaly detection 139, 140
isolation forest algorithms
 C++ implementation, for anomaly
 detection 146-153
Isomap algorithm 178, 179
Isomap method 197, 198
item-based collaborative filtering 243
item-based collaborative filtering
 example, with C++ 253
 Eigen library, using 253-260
 mlpack library, using 260-262
items 239

J

- Java Native Interface (JNI)** 435
- JavaScript Object Notation (JSON)** 37-39
- Java Virtual Machine(JVM)** 440
- JSON files**
 - reading, with Nlohmann json library 45-52
- just-in-time (JIT)** 22

K

- kernel** 144
- kernel density estimation (KDE)** 141, 143
 - for anomaly detection 143, 144
- kernel function** 178
- kernel matrix** 178
- Kernel PCA** 178, 195, 196
- kernel ridge regression (KRR)** 205, 208, 209
 - Dlib library, using with 224-226
- kernel trick** 178
- K-means clustering**
 - using, with Dlib library 119, 120, 121
 - using, with mlpack library 115, 116
- k-nearest neighbors**
 - (kNN) 118, 203, 213-215, 242
- Kotlin part, Android Studio project**
 - camera orientation, preserving 436
 - camera permission requests, handling 436-439
 - native library loading 439, 440

L

- L1 loss function** 312
- L2 loss function** 312
- L2 regularization** 208, 320
- Laplacian loss** 273
- Least Absolute Shrinkage and Selection Operator (Lasso)** 87

- least angle regression (LARS)** 88
- least squares support vector machine (LSSVM)** 88
- leave-one-out method** 248
- leave-p-out method** 248
- LeNet architecture**
 - using, for image classification 337, 338
- LibPyTorch** 338
- linear activation function** 315
- linear algebra**
 - matrix 9
 - operations 9
 - overview 7
 - samples 12
 - scalar 8
 - tensor 9
 - vector 8
- linear algebra operations**
 - dot product 10
 - element-wise operation 9
 - inverting 11
 - norm 10
 - Tensor representation in computing 11
 - transposing 10
- linear algebra samples**
 - ArrayFire, using 21-23
 - Blaze, using 17-20
 - Dlib, using 24, 25
 - Eigen, using 12-15
 - xtensor, using 15-17
- linear discriminant analysis (LDA)** 174, 175
 - implementing 189, 190
- linear discriminant (LD) function** 174
- linearly separable** 206
- linear methods, dimension-reduction**
 - factor analysis 175-177
 - independent component analysis 172-174
 - linear discriminant analysis 174, 175

principal component analysis
(PCA) 170, 171

singular value decomposition 172

linear regression 88, 208

- overview 26, 27
- with Dlib 34

linear regression algorithm

- mlpack library, using with 222-224

linear regression tasks

- solving, with ArrayFire 32, 33
- solving, with Blaze 31
- solving, with different libraries 27-30
- solving, with Eigen 30, 31

linear regression training

- experiment tracking, integrating into 418-421

Local Outlier Factor (LOF) method 137

- for anomaly detection 137, 138

local reachability density 138

logarithm of the odds 206

logarithm of the probability ratio 206

logistic loss function (Log-Loss) 81

logistic regression 203-208

- Flashlight library, using with 229-232

logistic regression, and kernel trick

- Flashlight library, using with 232-234

loss function 4, 7, 304, 311-313

- cosine proximity 313
- cross-entropy loss function 312
- L1 loss function 312
- L2 loss function 312
- mean squared error (MSE) 311
- mean squared logarithmic error (MSLE) 312
- modification 87
- negative log-likelihood 313

M

machine learning (ML) 37, 103, 379

- fundamentals 4
- techniques 4

machine learning operations (MLOps) 410

Manhattan distance

- measuring 105

manifold 178

matrix 9

MDS 196, 197

mean absolute error (MAE) 75, 277, 312

mean normalization 67

MeanShift approach 112

MeanShift clustering

- using, with mlpack library 118

mean squared error (MSE) 74, 269, 311

mean squared logarithmic error (MSLE) 312

medoid 108

meta-attribute 268

mini-batch mode 305

min-max normalization 66

MLflow 410

- components 410
- REST C++ client, implementing 411-416

ML models

- dealing with 5, 6
- performance metrics 74

ML model serialization APIs

- in C++ libraries 380
- types 380
- with Dlib library 380-383
- with Flashlight library 384-386
- with mlpack library 386-388
- with PyTorch 388

mlpack

- AdaBoost, using with 286, 287
- data preparation 284, 285

-
- examples 284
 - random forest, using with 285, 286
 - stacking ensemble, using with 287-293
 - mlpack library 386**
 - GMM and EM algorithms,
using with 114, 115
 - used, for creating neural networks 332-334
 - used, for data normalization 68
 - used, for reading CSV files 43, 44
 - using 218, 260-262
 - using, for dealing with clustering
 - task examples 114, 119
 - using, with DBSCAN 117
 - using, with GMM and EM 114, 115
 - using, with K-means clustering 115, 116
 - using, with linear regression
 - algorithm 222-224
 - using, with MaeanShift clustering 118, 119
 - using, with softmax regression 218
 - using, with SVMs 220, 222
 - working with 58
 - mlpack library, for anomaly detection**
 - DET method 162-164
 - KDE algorithm 159-162
 - multivariate Gaussian model 157-159
 - ML techniques**
 - supervised learning 5
 - unsupervised learning 5
 - model-based clustering algorithms 112, 113**
 - applying, examples 113
 - model_loaded object 385**
 - model parameter estimation 7**
 - Modified National Institute of Standards
and Technology (MNIST) 337**
 - Monte Carlo method 269**
 - Monte Carlo principle 139**
 - MSE 74**
 - multi-class classification 215, 216**
 - multidimensional scaling 176, 177**
 - multilayer perceptron (MLP) 300**
 - Multiple additive regression
trees (MART) 275**
 - multiple linear regression 27**
 - multivariable linear regression 27**
 - multivariate Gaussian distribution**
 - for anomaly detection 141-143

N

- named_parameters() method 391**
- native C++ part, Android Studio project 440**
 - application window configuration 445-447
 - camera device 445-447
 - captured image processing 450-454
 - capture image 449, 450
 - Image-capturing pipeline
construction 447-449
 - IoU 461-465
 - main application loop 442, 443
 - model output tensor, processing 460, 461
 - NMS 461-465
 - object detection, initialization
with JNI 440-442
 - ObjectDetector class, overview 444, 445
 - OpenCV matrix, converting
into torch::Tensor 459
 - output window buffer management 449, 450
 - YOLO detection inference 458, 459
 - YOLO wrapper initialization 454-458
- native C++ wizard 434**
- natural language processing (NLP) 353**
- negative log-likelihood loss function 313**
- net_to_xml function 382**
- neural network model 353**
- neural network (NN) module 81**

-
- neural networks (NNs)** 64
 activation functions 313
 backpropagation method 303
 creating, with C++ libraries 329
 creating, with Dlib library 329-331
 creating, with Flashlight 335-337
 creating, with mlpack library 332-334
 initialization 322
 initialization, with He initialization
 method 322
 initialization, with Xavier
 initialization method 322
 loss functions 311
 neurons 299, 300
 overview 298
 perceptron 300
 regularization 320
neural networks layers
 hidden 301
 input 301
 output 302
Neural Processing Unit (NPU) 430
neurons 299, 300
Newman modularity-based graph clustering
 using, with Dlib library 125
Nlohmann json library
 used, for reading JSON files 45-52
non-linear methods, dimension-reduction
 autoencoders 182
 distributed stochastic neighbor
 embedding 180, 181
 Isomap 178, 179
 kernel PCA 178
 Sammon mapping 179, 180
Non-Maximum Suppression (NMS) 461
non-negative matrix factorization (NMF) 244
non-parametric model 6
non-personalized recommendations 240
normalized cuts problem 109
normal training
 for balanced bias and variance 85, 87
norm operation 10
novelty detection 134, 135
- O**
- object detection, on Android** 430
 mobile version, of PyTorch
 framework 431, 432
 TorchScript, using for model snapshot 433
one-against-all strategy 215
One-Class Support Vector Machine (OCSVM) 140
 for anomaly detection 140
ONNX format 395
 class definition file, reading 404, 405
 images, loading into onnxruntime
 tensors 402-404
 ResNet architecture for image
 classification, using 396-401
onnxruntime framework 395
onnxruntime tensors
 used, for loading images 402-404
OpenCV library
 deinterleaving 64
 used, for manipulating images 59-61
Open Neural Network Exchange (ONNX) 379
operating system (OS) 129
optimizer 7
Optuna
 with Flashlight example 93-97
ordinary least squares (OLS) 28
outlier detection 134
overfitting 81

P

parametric model 6
partition-based clustering algorithms 107
Pearson's correlation coefficient 245
perceptron 300
performance metrics, ML models 74
 classification metrics 77
 regression metrics 74
planar (or linear) mapping 140
pooling layers 325
pooling operation 325
positive class 140
precision 78
prediction 240
principal component analysis
 (PCA) 145, 171, 183, 184, 194, 409
 using, for data compression 185-189
principal components 172
principal component scores 172
probability density function (PDF) 142
product factors 244
Projected Clustering (PROCLUS) 108
pruning 278
PyTorch
 archive objects, using 393-395
 neural network, initializing 388-392
 torch::load function, using 392
 torch::save function, using 392
PyTorch framework
 mobile version 431-433

Q

Quantile loss 274

R

Radial Basis Function (RBF) kernel 153, 229
random-access memory (RAM) 11
random forest method 268
 overview 279, 280
 using, for creating ensembles 276
 using, with mlpack 285, 286
read_classes function 404
recall 78
receiver operating characteristic (ROC) 80
receptive field 326
recommendation
 using 240
recommender system algorithms
 cold start problem 246
 data scaling and standardization 246
 factorization algorithms 244
 for content-based recommendations 241
 for non-personalized recommendations 240
 item-based collaborative filtering 243, 244
 overview 238-240
 relevance of recommendations 247
 similarity or preferences correlation 245
 system quality, assessing 247, 248
 user-based collaborative filtering 242, 243
rectangular diagonal matrices 172
rectified linear unit (ReLU) 317
Red Green Blue Alpha (RGB) 61, 63, 447
regression algorithm metrics
 adjusted R squared 76
 mean absolute error (MAE) 75
 MSE 74
 RMSE 75
 R squared 75
regression algorithms 74

-
- regularization** 87
 data augmentation 88
 early stopping 88
 for NNs 89
 Lasso 87
 ridge 88
- regularization, neural networks** 320
 batch normalization 321
 dropout regularization 320
 L2 regularization 320
- relative error tolerance parameter** 160
- repeatable** 239
- rescaling** 66
- ResNet architecture, for image classification**
 using 396-401
- REST API, MLflow**
 metric values, logging 416, 417
 parameters, running 416-418
 used, for experiment tracking 411
- ridge** 88
- ridge regularization** 88
- root mean squared error**
 (RMSE) 75, 244, 270
- Rosenblatt's perceptron** 302, 303
- row-major ordering** 11
- R squared** 75, 76
- S**
- Sammon mapping** 179, 180
 implementing 190, 191
- scalar** 8
- sensitivity** 78
- sentiment analysis model**
 example, with BERT 360
- shift** 300
- shooting tournament strategy** 216
- sigmoid activation function** 316, 317
- Simple API for XML (SAX)** 47
- simple linear regression** 27
- Single Instruction Multiple Data (SIMD)** 8, 68
- single-layer perceptron (SLP)** 300-302
- singular value decomposition**
 (SVD) 172, 244
- singular values** 172
- soft margin** 211
- softmax regression**
 mlpack library, using with 218
- Spearman's correlation** 245
- spectral clustering algorithms** 109
 using, with Dlib library 121, 123
- spectral decomposition** 244
- spherical mapping** 140
- Squared Euclidean distance**
 measuring 104
- stacking approach** 268
 using, for creating ensembles 275
- stacking ensemble**
 using, with mlpack 287-293
- standardization** 66
- standard template library (STL)** 24
- statistical tests** 136
 for anomaly detection 136, 137
- std**
 getline function 405
- stepwise activation function** 314, 315
- stochastic GBM** 274
- stochastic gradient boosting** 274
- stochastic mode** 305
- stochastic neighbor embedding (SNE)** 177
- stretching the axes technique** 214
- Subspace Clustering (SUBCLU)**
 algorithm 112
- supervised learning** 5

support vector machine (SVM)
66, 140, 203, 209-213
 Dlib, using with 227, 229
 mlpack library, using with 220, 222
support vector machine (SVM) algorithm 98
surfaces 178
synapse 299

T

Tapkee library
 factor analysis 198
 Isomap 197, 198
 kernel PCA 195, 196
 MDS 196, 197
 PCA, using 194, 195
 t-SNE method 199, 200
 using 191, 193
t-distributed stochastic neighbor embedding (t-SNE) 145
tensor 9
TensorBoard
 features 409
tokenization 358
 methods 358
torch::load function 392
torch::save function 392
TorchScript 433
 using, for model snapshot 433
training data preprocessing 87
transfer learning 353
Transformer architecture 353
 decoder part 357
 decoder parts, using separately 359, 360
 encoder part 356
 encoder parts, using separately 359, 360
 overview 354-356
 tokenization 358, 359

transposing 10
True Positive Rate (TPR) 79
t-SNE method 199, 200

U

underfitting 81, 82
union area 462
unrepeatable 239
unsupervised anomaly detection
 technique 133

unsupervised learning (UL) 5, 141
user-based collaborative filtering 242, 243
user factors (U) 244
user ratings, of objects 240

V

validation 81
vanishing gradient problem 317
variance 81, 84, 85
vector 8
vectorized computations 8
visualization and tracking systems,
 for ML experiments 408
 MLflow, using 410
 TensorBoard, using 409

W

weak models 272
weight decay 89
width, height, channels, batch (WHCN) 335
WordPiece tokenization 358

X

Xavier initialization method

using 322

xtensor

using 15-17

Y

You Only Look Once (YOLO) 433

Z

Z-score anomaly detection 137



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

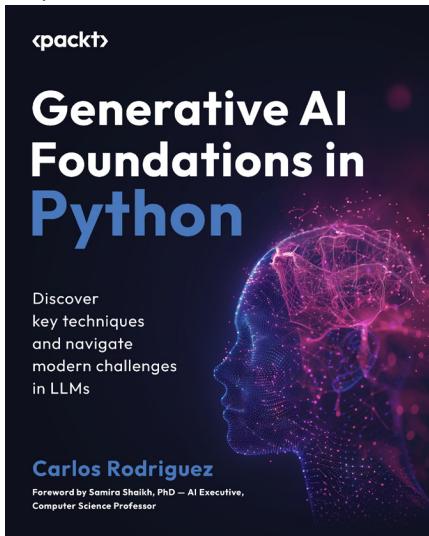
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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 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.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

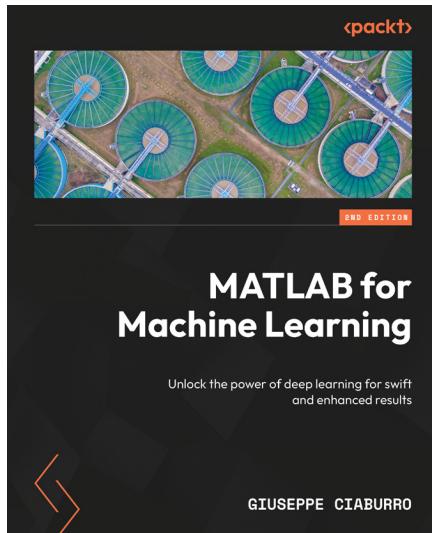


Generative AI Foundations in Python

Carlos Rodriguez

ISBN: 978-1-83546-082-5

- Discover the fundamentals of GenAI and its foundations in NLP
- Dissect foundational generative architectures including GANs, transformers, and diffusion models
- Find out how to fine-tune LLMs for specific NLP tasks
- Understand transfer learning and fine-tuning to facilitate domain adaptation, including fields such as finance
- Explore prompt engineering, including in-context learning, templatization, and rationalization through chain-of-thought and RAG
- Implement responsible practices with generative LLMs to minimize bias, toxicity, and other harmful outputs



MATLAB for Machine Learning

Second Edition

Giuseppe Ciaburro

ISBN: 978-1-83508-769-5

- Discover different ways to transform data into valuable insights
- Explore the different types of regression techniques
- Grasp the basics of classification through Naive Bayes and decision trees
- Use clustering to group data based on similarity measures
- Perform data fitting, pattern recognition, and cluster analysis
- Implement feature selection and extraction for dimensionality reduction
- Harness MATLAB tools for deep learning exploration

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Hands-On Machine Learning with C++*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781805120575>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly