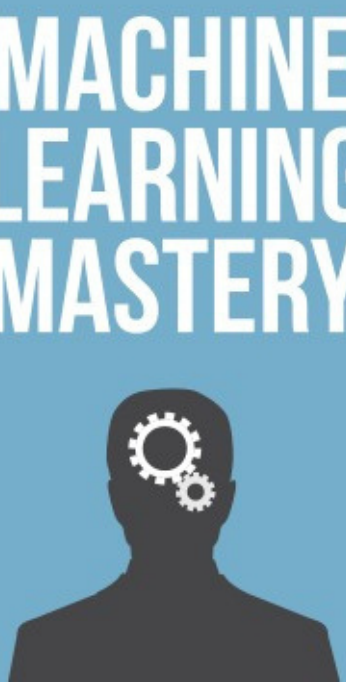
Basics of Linear Algebra for Machine Learning

Discover the Mathematical Language of Data in Python



Contents

i

ii

iii

I Intro duction v

Welcome vi
Who Is This Book For? vi
About Your Outcomes vi
How to Read This Book vii
About the Book Structure .. . .vii .
. . About Python Code. viii
Examples . About Further. ix
Reading . . . . . About. ix
Getting Help ix
Summary.....

II Foundations 1

1Introduction to Linear Algebra 2
1.1Tutorial Overview 2
1.2Linear Algebra 2
1.3Numerical Linear Algebra 3
1.4. . Linear Algebra and..... 4
1.5Statistics . Applications of..... 5
1.6Linear Algebra . Further..... 5
1.7Reading 6
Summary.....
2 Linear Algebra and Machine Learning 7
2.1Reasons to NOT Learn Linear. 7
2.Algebra Learn Linear Algebra. 8
2Notation . . . . Learn Linear. 8
2.Algebra Arithmetic . . . Learn. 9
3Linear Algebra for Statistics 9
2.Learn Matrix Factorization . . . . . .
4
2.ii
5

2.Learn Linear Least..... 9
6Squares One More..... 9
2.Reason 1
7 Summary.....
3 E2xa.mples of Linear Algebra in Machine Learning 10
83.1 Overview..... 3.2.....
DatasetandDataFiles..... 3.3..... 1
Images and Photographs 11
3.4 OneHotEncoding ..... 3.5..... 12
Linear Regression . . . . . 3.6..... 12
Regularization . . . . . 3.7..... 13
PrincipalComponentAnalysis. 13
3.8 Singular-..... 13
ValueDecomposition . 3.9..... 14
LatentSemanticAnalysis ... 3.10..... 14
Recommender Systems 14
3.11 Deep Learning 15
3.12Summary..... 15
15

III NumPy 16

4Introduction to NumPy Arrays 17
4.1Tutorial Overview 17
4.NumPy N-dimensional..... 17
2Array Functions to Create..... 18
4.Arrays . . Combining..... 19
3Arrays 21
4.Extensions..... Further..... 21
4Reading 2
4. Summary..... 2
5I5ndex, Slice and Reshape NumPy Arrays 23
45..1Tutorial Overview..... 2
65.. . From List to..... 3
42.Arrays . Array..... 2
75.Indexing . . . Array..... 3
3Slicing 2
5.ArrayReshaping 5
4Extensions 2
5.Further Reading 7
5. . Summary 3
6 N5u.mPyArrayBroadcasting 3
6.1Tutorial Overview 0
65.Limitation with Array... 5
72Arithmetic . Array... 3
65.Broadcasting 25
83Broadcasting in NumPy 3
6.Limitations of Broadcasting 35
4. . Extensions..... 3
6. 3
5 6
6. 3
6 6
3
9

6. Further 4
7Reading 1
6.Summary 4
8 4
IV Matrices 43

7 VectorsandVectorArithmetic 44
7.1 TutorialOverview..... 7.2. 4
WhatisaVector 7.3. 4
Defining a Vector 7.4. 4
VectorArithmetic..... 7.5. 4
VectorDotProduct..... 7.6. 4
Vector-ScalarMultiplication .. 5
7.7 Extensions..... 7.8. 4
FurtherReading..... 7.9. 5
Summary 4
8 VectorNorms 9
8.1 Tutorial 53
8.Overview 50
2Vector Norm 351
8.. Vector L1 Norm..... 51
3. . Vector L2..... 35
8.Norm . . Vector..... 52
4Max Norm 4
8.Extensions 5
5Further Reading 5
9 M8a.tric. e. sSaunmdmMatrryi x. A. .r .i t.hmetic 58
96.1 TutorialOverview..... 9.2. 5
W8.hat is a Matrix 9.3. 86
D7efiningaMatrix..... 9.4. 5
M8.atrixArithmetic 9.5. 86
M8atrix-MatrixMultiplication. 5
9.6 Matrix-. 97
VectorMultiplication. 9.7. 5
Matrix-ScalarMultiplication. . 9
9.8 Extensions..... 9.9. 6
FurtherReading..... . 4
9.10Summary..... 6
10 Types of Matrices 761
10.1 Tutorial Overview 671
10.2 Square Matrix 7
10.3 Symmetric Matrix..... 62
10.4 Triangular Matrix 97
10.5 Diagonal Matrix 62
10.6 Identity Matrix 97
10.7 Orthogonal Matrix..... 73
07
4
7
5
7
6

10.8 Extensions 7
10.9 Further Reading 8
10.10Summary 8870
11 Matrix Operations
11.1 Tutorial Overview 87
11.2 Transpose 09
11.3 Inverse 8
11.4Trace 0
11.5 Determinant 81
11.6Rank 8
11.7 Extensions 3
11.8 Further Reading 8
11.9 Summary 4
12 Sparse Matrices 8950
12.1 Tutorial Overview 89
12.2SparseMatrix . . . . . 12.3 Problems with Sparsity . . . . . 12.4 Sparse Matrices in Machine Learning 12.5 Working with Sparse Matrices . . . . 12.6 Sparse Matrices in Python . . . . . 12.7Extensions . . . . . 12.8 Further Reading . . . . . 12.9Summary . . . . . .07 .891 .971 .89 .82 9 3 9
13 Tensors and Tensor Arithmetic 4
13.1 Tutorial Overview 98
13.2 What are Tensors 985
13.3 Tensors in Python 989
13.4 Tensor Arithmetic 995
13.5 Tensor Product 1009
13.6 Extensions 1046
13.7 Further Reading 105
13.8 Summary 106
.107

V Factorization 108

14 Matrix Decompositions 109
14.1 Tutorial Overview 109
14.2 What is a Matrix Decomposition 109
14.3 LU Decomposition 110
14.4 QR Decomposition 111
14.5 Cholesky Decomposition 112
14.6Extensions 114
14.7 Further Reading 114

14.8 Summary 115

15 Eigendecomposition 116
15.1 Tutorial Overview 15.2. 116
Eigendecomposition of a Matrix 117
15.3 Eigenvectors and Eigenvalues 118
. 15.4 Calculation of. 118
Eigendecomposition . . . 15.5 Confirm. 119
an Eigenvector and Eigenvalue . 15.6. 120
Reconstruct Matrix 121
15.7Extensions. 121
15.8FurtherReading. 122
15.9Summary.
16 Singular Value Decomposition 123
16.1 Tutorial Overview . . . . . 16.2. .123
What is the Singular-Value. 124
Decomposition . 16.3 Calculate Singular-. 124
Value Decomposition . . 16.4 Reconstruct. 125
Matrix 127
16.5Pseudoinverse. 16.6. 129
Dimensionality Reduction 131
16.7Extensions. 132
16.8FurtherReading. 133
16.9Summary.

VI Statistics 135

17 Introduction to Multivariate Statistics 136
17.1 Tutorial Overview 136
17.2 Expected Value and Mean 136
. 17.3 Variance and Standard. 138
Deviation 17.4 Covariance and. 141
Correlation . . . 17.5 Covariance. 143
Matrix 144
17.6Extensions. 17.7 Further. 144
Reading 146
17.8Summary.
18 Principal Component Analysis 147
18.1TutorialOverview. 18.2 What is. 147
Principal Component Analysis 18.3. 147
Calculate Principal Component Analysis 149
18.4 Principal Component Analysis in scikit-. 150
learn . 18.5Extensions. 151
18.6FurtherReading. 151
18.7API. 18.8Articles. 151
18.9Summary. 151
. 152

19 Linear Regression 153
19.1 Tutorial Overview 153
19.2 What is Linear Regression 154
19.3 Matrix Formulation of Linear Regression 154
19.4 Linear Regression Dataset 155
19.5 Solve via Inverse 157
19.6 Solve via QR Decomposition 158
19.7 Solve via SVD and Pseudoinverse 160
19.8 Solve via Convenience Function 162
19.9 Extensions 163
19.10 Further Reading 164
19.11 Summary 165

VII App endix 167

A Getting Help 168
A.1 Linear Algebra on Wikipedia 168
A.2 Linear Algebra Textbooks 168
A.3 Linear Algebra University Courses 169
A.4 Linear Algebra Online Courses 170
A.5 NumPy Resources 170
A.6 Ask Questions About Linear Algebra 171
A.7 How to Ask Questions 171
A.8 Contact the Author 172
B How to Setup a Workstation for Python
B.1 Overview 172
B.2 Download Anaconda 172
B.3 Install Anaconda 174
B.4 Start and Update Anaconda 176
B.5 Further Reading 179
B.6 Summary 179

C Linear Algebra Cheat Sheet 180
C.1 Array Creation 180
C.2 Vectors 180
C.3 Matrices 181
C.4 Types of Matrices 182
C.5 Matrix Operations 183
C.6 Factorization 183
C.7 Statistics 184

D Basic Math Notation 186
D.1 Tutorial Overview 186
D.2 The Frustration with Math Notation 187
D.3 Arithmetic Notation 187

D.4 Greek Alphabet 189
D.5 Sequence Notation 190
D.6 Set Notation 191
D.7 Other Notation 192
D.8 TipsforGettingMoreHelp. 192
D.9 Further Reading 194
D.10Summary 194

VIII Conclusions 195

How Far You Have Come 196

Preface

I wrote this book to help machine learning practitioners, like you, get on top of linear algebra, fast.

Linear Algebra Is Important in Machine Learning

There is no doubt that linear algebra is important in machine learning. Linear algebra is the mathematics of data. It's all vectors and matrices of numbers. Modern statistics is described using the notation of linear algebra and modern statistical methods harness the tools of linear algebra. Modern machine learning methods are described the same way, using the notations and tools drawn directly from linear algebra. Even some classical methods used in the field, such as linear regression via linear least squares and singular-value decomposition, are linear algebra methods, and other methods, such as principal component analysis, were born from the marriage of linear algebra and statistics. To read and understand machine learning, you must be able to r

Peraadc atnitdio unnedresr sStatundd ylin Leianre aalgre Abirga.ebra Too Early

If you ask how to get started in machine learning, you will very likely be told to start with linear algebra. We know that knowledge of linear algebra is critically important, but it does not have to be the place to start. Learning linear algebra first, then calculus, probability, statistics, and eventually machine learning theory is a long and slow bottom-up path. A better fit for developers is to start with systematic procedures that get results, and work back to the deeper understanding of theory, using working results as a context. I call this the top-down or results-first approach to machine learning, and linear algebra is not the first step, but perhaps the second or third.

Practitioners Study Too Much Linear Algebra

When practitioners do circle back to study linear algebra, they learn far more of the field than is required for or relevant to machine learning. Linear algebra is a large field of study that has tendrils into engineering, physics and quantum physics. There are also theorems and derivations for nearly everything, most of which will not help you get better skill from or a deeper understanding of your machine learning model. Only a specific subset of linear algebra is required, though you can always go deeper once you have the basics.

Part I

Intro duction

Welcome

Welcome to Basics of Linear Algebra for Machine Learning. Linear algebra is a pillar of machine learning.

The field started to be formalized about 150 years ago, but it was only about 70 years ago that modern linear algebra came into existence. It's a huge field of study that has made an impact on other areas of mathematics, such as statistics, as well as engineering and physics. Thankfully, we don't need to know the breadth and depth of the field of linear algebra in order to improve our understanding and application of machine learning.

I designed this book to teach you step-by-step the basics of linear algebra with concrete and

eWxehcuota Ibsle T ehxaims pBleos oink P Fytohorn?.

Before we get started, let's make sure you are in the right place. This book is for developers that may know some applied machine learning. Maybe you know how to work through a predictive modeling problem end-to-end, or at least most of the main steps, with popular tools. The lessons in this book do assume a few things about you, such as:

You know your way around basic Python for programming.

You may know some basic NumPy for array manipulation.

You want to learn linear algebra to deepen your understanding and oafp mpliaccahtiionne learning.

ATbhiso guutid Ye owuasr w Oritutetnc ion mthee tsop-down and results-first machine learning style

This book will teach you the basics of linear algebra that you need tot khnaot wyo aus'r ae musaecdh tion efr olemar Mniancgh ipnreaLcetiatrionninegr.M Aafsteter ryre.caodmin.g and working through this book, you will know:

leahranti nlign.ear algebra is and why it is relevant and important to machine

How to create, index, and generally manipulate data in NumPy arrays.

What a vector is and how to perfor nmo rvmecst.or arithmetic and calculate vector

What a matrix is and how to perform matrix arithmetic, including matrix multiplication.

A suite of types of matrices, their properties, and advanced operations involving matrices. What a tensor is and how to perform basic tensor arithmetic.

Matrix factorization methods, including the eigendecomposition and singular-value decomp osition.

How to calculate and interpret basic statistics using the tools of linear algebra.

How to implement methods using the tools of linear algebra such as principal componen analysis and linear least squares regression.
This new basic understanding of linear algebra will impact your practice of machine learn in the following ways:

Read the linear algebra mathematics in machine learning papers.

Implement the linear algebra descriptions of machine learning algorithms.

Describe your machine learning models using the notation and operations of linear algeb

HThois wbo otko i sR neota ad s uTbhstiistu Bteo for kan under

Tsuhcish b ao cooku wrsaes, awltrhitoteung hto i tb ceo rueldad c olimnepalrelym, fernotn yteoxut bkonookws , tahned b oatshicesr raensdo unreceeds ohne llpin weaitrh a lag espb yeoauch c atunt oflripia ls.traight to that section and get started. This book was designed f you to read on your workstation, on the screen, not on a tablet or eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them.
This book is not intended to be read passively or be placed in a folder as a reference text. It
is a playbook, a workbook, and a guidebook intended for you to learn by doing and then apply your new understanding with working Python examples. To get the most out of the book, I would recommend playing with the examples in each tutorial. Extend them, break them, then fix them. Try some of the extensions presented at the end of each lesson and let me know how you do.

About the Book Structure

This book was designed around major data structures, operations, and techniques in linear algebra that are directly relevant to machine learning algorithms. There are a lot of things you could learn about linear algebra, from theory to abstract concepts to APIs. My goal is to take

you straight to developing an intuition for the elements you must understand with laser-f
tutorials.
I designed the tutorials to focus on how to get things done with linear algebra. They give
you the tools to both rapidly understand and apply each technique or operation. Each of t
tutorials are designed to take you about one hour to read through and complete, excludir
extensions and further reading. You can choose to work through the lessons one per day,
week, or at your own pace. I think momentum is critically important, and this book is inter
to be read and used, not to sit idle. I would recommend picking a schedule and sticking to
The tutorials are divided into 5 parts:

Part 1: Foundation. Discover a gentle introduction to the field of linear algebra and the relationship it has with the field of machine learning.

Part 2: NumPy. Discover NumPy tutorials that show you how to create, index, slice, and reshape NumPy arrays, the main data structure used in machine learning and the basis for linear algebra examples in this book.

Part 3: Matrices. Discover the key structures for holding and manipulating data in linear algebra in vectors, matrices, and tensors.

Part 4: Factorization. Discover a suite of methods for decomposing a matrix into its constituent elements in order to make numerical operations more efficient and more stal

Part 5: Statistics. Discover statistics through the lens of linear algebra and its application to principal component analysis and linear regression.

Each part targets a specific learning outcome, and so does each tutorial within each part.

This acts as a filter to ensure you are only focused on the things you need to know to get t
specific result and do not get bogged down in the math or near-infinite number of digres
The tutorials were not designed to teach you everything there is to know about each of th
theories or techniq

hAobwo thuety Pues of linear algebra. They were designed to g

Ttoh ele acrond be ye dxoaimngp.les were carefully designed to demonstrate the purpose
given lesson. Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third parties required beyond the installation of the required packages. A complete working example is presented with each tutorial for you to inspect and copy-and-paste. All source code is also provided with the book and I would recommend running the provided files whenever possible to avoid any copy-paste issues.
The provided code was developed in a text editor and intended to be run on the command
line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead. All code examples were tested on a POSIX-compatible machine with Python 3.

About Further Reading

Each lesson includes a list of further reading resources. This may include:

Books and book chapters.

API documentation.

Articles and Webpages.

Wherever possible, I try to list and link to the relevant API documentation for key function
used in each lesson so you can learn more about them. I have tried to link to books on Am
so that you can learn more about them. I don't know everything, and if you discover a goo
resource related to a given lesson, please let me know so I can update the book.

About Getting Help

You might need help along the way. Don't worry; you are not alone.

Help with a Technique? If you need help with the technical aspects of a specific operation or technique, see the Further Reading sections at the end of each lesson.

resources in the

HelpA w.ith NumPy? If you need help with using the NumPy library, see the list of

Further Reading section at the end of each lesson, and also see Appendix

Help with your workstIaf tyioun ?need help setting up your environment, I would recommend using Anaconda and following my tutorial in Appendix B.

Help with the Im praothvi?ded a list of locations where you can search for answers and ask questions about linear algebra math in Appendix A. You can also see Appen
for a crash course on math notation.

HYeolup cinan g sehnoeorat lm?e an email. My details are in Appendix A.

Summary

Are you ready? Let's dive in!

Next

Next up you will discover a gentle introduction to the field of linear algebra.

Part II

Foundations

Chapter 1

Introduction to Linear Algebra

Linear algebra is a field of mathematics that is universally agreed to be a prerequisite to a deeper understanding of machine learning. Although linear algebra is a large field with many esoteric theories and findings, the nuts and bolts tools and notations taken from the field are practical for machine learning practitioners. With a solid foundation of what linear algebra is, it is possible to focus on just the good or relevant parts. In this tutorial, you will discover what exactly linear algebra is from a machine learning perspective. After completing this tutorial, you will know:

Linear algebra is the mathematics of data.

Linear algebra has had a marked impact on the field of statistics.

Linear algebra underlies many practical mathematical tools, such

saesr Fieosu arniedr

computer graphics.

L1.e1t's geTt ustatroterdia.lOverview

This tutorial is divided into 4 parts; they are:

1. Linear Algebra.

2. Numerical Linear Algebra.

3. Linear Algebra and Statistics.

4. Applications of Linear Algebra.

1.2 Linear Algebra

Linear algebra is a branch of mathematics, but the truth of it is that linear algebra is the mathematics of data. Matrices and vectors are the language of data. Linear algebra is about linear combinations. That is, using arithmetic on columns of numbers called vectors and arrays

1.3.NumericalLinearAlgebra 3 of numbers called matrices, to create new

columns and arrays of numbers. Linear algebra is the

s
fitnuedayr otrfa lninsefos ramnsd. planes, vector spaces and mappings that are required fo
It is a relatively young field of study, having initially been formalized in the 1800s

in order

stoe rfiiensd o uf ntekrnmosw annsd in systems of linear equations. A linear equation is jus
mathematical operations where some terms are unknown; for example:

y = 4

×x+1 (1.1)

Equations like this are linear in that they describe a line on a two-dimensional

××graph. The

wlinhea ct othmee esq fruoamtio pnlugging in differen×t values in×to the unknown x to fi
or model does to the value of y. We c×an line up× a system of equations with the

same form with···

two Tohr em coorleu munnk nofo yw vnasl;u feosr ecxaanm bpe ltea:ken as a column vect

y=0.1 x1+0.4 x2equation. The

two columns of integer values a
yr=e 0th.3e dx1a+ta0-c.9o luxm2hs, say a1 and a2, and can be
taken as a matrix(1.2)

A. The two unknown values x1y a=n0d. 2x 2x 1c+a0n. 3b ex 2taken as the coefficients of the
equation and together form a vector of unknowns b to be solved. We can write
this compactly using linear
algebra notation as:

y = A

·b (1.3)

Problems of this form are generally challenging to solve because there are more

unknowns

t(hheerree wis eo fhtaevne n 2o) than there are equations to solve (here we have 3). Furthe
single line that can satisfy all of the equations without error. Systems describing

1.3 NumericalLinearAlgebra

problems we

oaTrfh ese oo alfuptetpinoli ncinast.teiornes otef dlin inea(rs uaclghe absr a ilnin ceoamr rp

This gives a small taste of the very core of linear algebra that interests us as
“numerical” linear algebra is really applied linear algebra.
machine learning
p
anradc ptirtoiobnleemrs.s Much of the rest of the operati—o nPsa gaer eix ,a Nbuomuet rm
l
Iitk ies imt eoarsei ethr aton ujunsdt etrhseta imndp alenmd esnotlvaeti.on of linear algebr
includes the careful handling of the problems of applied mathematics, such as working w
limited floating point precision of digital computers. Computers are good at performing li
algebra calculations, and much of the dependence on Graphical Processing Units (GPUs)
modern machine learning methods such as deep learning is because of their ability to co
linear algebra operations fast.

Efficient implementations of vector and matrix operations were originally implemented in the FORTRAN programming language in the 1970s and 1980s and a lot of code, or code po from those implementations, underlies much of the linear algebra performed using mode programming languages, such as Python. Three popular open source numerical linear alg libraries that implement these functions are:

Linear Algebra Package, or LAPACK.

Basic Linear Algebra Subprograms, or BLAS (a standard for linear algebra libraries).

Automatically Tuned Linear Algebra Software, or ATLAS.

Often, when you are calculating linear algebra operations directly or indirectly via higher-order algorithms, your code is very likely dipping down to use one of these, or similar linea algebra libraries. The name of one of more of these underlying libraries may be familiar to if you have installed or compiled any of Python's numerical libraries such as SciPy and Nu

1.4 Linear Algebra and Statistics

Linear algebra is a valuable tool in other branches of mathematics, especially statistics. Usually students studying statistics are expected to have seen at least one semester of linear algebra (or applied algebra) at the undergraduate level.

— Page xv, Linear Algebra and Matrix Analysis for Statistics, 2014.

The impact of linear algebra is important to consider, given the foundational relationship both fields have with the field of applied machine learning. Some clear fingerprints of line algebra on statistics and statistical methods include:

Use of vector and matrix notation, especially with multivariate statistics.

Solutions to least squares and weighted least squares, such as for linear regression.

Estimates of mean and variance of data matrices.

The covariance matrix that plays a key role in multinomial Gaussian distributions.

Principal component analysis for data reduction that draws many of these elements together.

As you can see, modern statistics and data analysis, at least as far as the interests of a machine learning practitioner are concerned, depend on the understanding and tools of algebra.

1.5 ApplicationsofLinearAlgebra

As linear algebra is the mathematics of data, the tools of linear algebra are used in many domains. In his classical book on the topic titled Introduction to Linear Algebra, Gilbert Strang provides a chapter dedicated to the applications of linear algebra. In it, he demonstrates specific mathematical tools rooted in linear algebra. Briefly they are:

Matrices in Engineering, such as a line of springs.

Graphs and Networks, such as analyzing networks.

Markov Matrices, Population, and Economics, such as population growth.

Linear Programming, the simplex optimization method.

Fourier Series: Linear Algebra for functions, used widely in signal processing.

Linear Algebra for statistics and probability, such as least squares for regression.

C
imoamgpeus.ter Graphics, such as the various translation, rescaling and rotation of

1.6FuArntohtheerr iRnteearedstiing application of linear algebr

This section provides more resources on the topic if you are lomokainthge tmo agtoic dse
b
teyn Asolbre crat lEcuinlustse.in in parts of his theory of relativity. Specifically tensors and
H1.6e .a1lso iBntoro odkusced a new type of linear algebra notation to physi

E instIenintr ondoutcatitoino nto, oLirnear Algebra, 2016.
the Ehinttspte:/i/na msuzmn.mtoa/2tji2oJn0 cgo4nvention.

Numerical Linear Algebra, 1997.
http://amzn.to/2kjEF4S

Linear Algebra and Matrix Analysis for Statistics, 2014.

http://amzn.to/2A9ceNv

1.6.2 Articles

Linear Algebra on Wikipedia.
https://en.wikipedia.org/wiki/Linear_algebra

Linear Algebra Category on Wikipedia.
https://en.wikipedia.org/wiki/Category:Linear_algebra

Linear Algebra List of Topics on Wikipedia.
https://en.wikipedia.org/wiki/List_of_linear_algebra_topics

LAPACK on Wikipedia.
https://en.wikipedia.org/wiki/LAPACK

Basic Linear Algebra Subprograms on Wikipedia.
https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

Automatically Tuned Linear Algebra Software on Wikipedia.
https://en.wikipedia.org/wiki/Automatically_Tuned_Linear_Algebra_Software

Einstein notation on Wikipedia.
https://en.wikipedia.org/wiki/Einstein_notation

Mathematics of general relativity on Wikipedia.
https://en.wikipedia.org/wiki/Mathematics_of_general_relativity

1.7 Summary

In this tutorial, you discovered a gentle introduction to linear algebra from a machine learning perspective. Specifically, you learned:

Linear algebra is the mathematics of data.

Linear algebra has had a marked impact on the field of statistics.

Linear algebra underlies many practical mathematical tools, such
saesr Fieosu arniedr
1c.o7m.1 pNuetexrt graphics.

In the next chapter you will discover why linear algebra is important to machine learning.

Chapter 2

Linear Algebra and Machine Learning

Linear algebra is a field of mathematics that could be called the mathematics of data. It is undeniably a pillar of the field of machine learning, and many recommend it as a prerequisite subject to study prior to getting started in machine learning. This is misleading advice, as linear algebra makes more sense to a practitioner once they have a context of the applied machine learning process in which to interpret it. In this chapter, you will discover why machine learning practitioners should study linear algebra to improve their skills and capabilities as practitioners. After reading this chapter, you will know:

Not everyone should learn linear algebra, that it depends where you a
prreo icne ysso uorf
learning machine learning.

5 Reasons why a deeper understanding of linear algebra is
in rteeqrumireeddi afoter
machine learning practitioners.

T
2h.1at linReaer aalsgeobnras c taon bNe OfunT iLf aepaprrona
Let's get started.
Before we go through the reasons that you should learn linear algebra, let's start off by taking a small look at the reason why you should not. I think you should not study linear algebra if you are just getting started with applied machine learning.

It's not required. Having an appreciation for the abstract op
uenrdaetirolyn ss othmaet
machine learning algorithms is not required in order to use machine learning
as a tool to
solve problems.

It's slow. Taking months to years to study an entire related field before m
leaacrnhiinnge
will delay you achieving your goals of being able to work through predictive
7 modeling
problems.

It's a huge field. Not all of linear algebra is relevant to theoretic
leaalrmniancgh i lneet

I recommend a breadth-first approach to getting started in applied machine learning. I c
this approach a results-first approach. It is where you start by learning and practicing the
for working through a predictive modeling problem end-to-end (e.g. how to get results) w
tool (such as scikit-learn and Pandas in Python). This process then provides the skeleton a
context for progressively deepening your knowledge, such as how algorithms work and e
the math that underlies them. After you know how to work through a predictive

modeling
problem, let's look at why you should deepen your understanding of linear algebra.

Linear algebra is a branch of mathematics that is widely used throughout science
and engineering. However, because linear algebra is a form of continuous rather
than discrete mathematics, many computer scientists have little experience with it.

— Page 31, Deep Learning, 2016.

2.2 Learn Linear Algebra Notation

You need to be able to read and write vector and matrix notation. Algorithms are described in books, papers and on websites using vector and matrix notation. Linear algebra is the mathematics of data and the notation allows you to describe operations on data precisely with specific operators. You need to be able to read and write this notation. This skill will allow you to:

Read descriptions of existing algorithms in textbooks.

Interpret and implement descriptions of new methods in research papers.

Concisely describe your own methods to other practitioners.

Further, programming languages such as Python offer efficient ways of implementing linear

a
relgaelizberad nino tyaotuiorn directly. An understanding of the notation and how it is

l2a.n3guagLee oar rlinbr aLryin weilla arll oAwl gfoer sbhroart e

iImn plaermtnenrsthatipio nwsi tohf tmhea cnhointaetion of linear algebra are the arithm
lpeearrfnoirnmge adlg. oYroituh mnse.ed to know how to add, subtract, and multiply scala
vectors, and matrices. A challenge for newcomers to the field of linear algebra are operations such as matrix multiplication and tensor multiplication that are not implemented as the direct multiplication of the elements of these structures, and at first glance appear nonintuitive.

Again, most if not all of these operations are implemented efficiently and provided via API calls in modern linear algebra libraries. An understanding of how vector and matrix operations are implemented is required as a part of being able to effectively read and write matrix notation.

2.4 Learn Linear Algebra for Statistics

You must learn linear algebra in order to be able to learn statistics. Especially multivariate statistics. Statistics and data analysis are another pillar field of mathematics to support machine learning. They are primarily concerned with describing and understanding data. As the mathematics of data, linear algebra has left its fingerprint on many related fields of mathematics, including statistics.

In order to be able to read and interpret statistics, you must learn the notation and operations of linear algebra. Modern statistics uses both the notation and tools of linear algebra to describe the tools and techniques of statistical methods. From vectors for the means and variances of data, to covariance matrices that describe the relationships between multiple Gaussian variables. The results of some collaborations between the two fields are also staple machine learning

m2.e5thodLse. saurcnh aMs.tahter Pixri nFcaipcalt Coormizpaothieonnt Analysis, or PCA for short, used for data reduction.

Building on notation and arithmetic is the idea of matrix factorization, also called matrix decomposition. You need to know how to factorize a matrix and what it means. Matrix factorization is a key tool in linear algebra and used widely as an element of many more complex operations in both linear algebra (such as the matrix inverse) and machine learning (least squares).

Further, there are a range of different matrix factorization methods, each with different strengths and capabilities, some of which you may recognize as ”machine learning” methods, such as Singular-Value Decomposition, or SVD for short, for data reduction. In order to read and interpret higher-order matrix operations, you must understand matrix factorization.

2.6 Learn Linear Least Squares

You need to know how to use matrix factorization to solve linear least squares. Linear algebra was originally developed to solve systems of linear equations. These are equations where there are more equations than there are unknown variables. As a result, they are challenging to solve arithmetically because there is no single solution as there is no line or plane can fit the data without some error. Problems of this type can be framed as the minimization of squared error, called least squares, and can be recast in the language of linear algebra, called linear least squares.

Linear least squares problems can be solved efficiently on computers using matrix operations such as matrix factorization. Least squares is most known for its role in the solution to linear regression models, but also plays a wider role in a range of machine learning algorithms. In order to understand and interpret these algorithms, you must understand how to use matrix factorization methods to solve least squares problems.

2.7 OneMoreReason

If I could give one more reason, it would be: because it is fun. Seriously. Learning linear algebra, at least the way I teach it with practical examples and executable code, is a lot of fun. Once you

2.8.Summary 10 can see how the operations work on real data, it is hard to avoid developing

a strong intuition
for the methods. I am not alone in thinking that linear algebra can be fun if approached in the right way:

> Learning linear algebra can also be a lot of fun. Readers will experience knowledge buzz as they learn about the connections between concepts, and it's not uncommon to experience mind-expanding moments while studying this subject.
>
> — Page ix, No Bullshit Guide To Linear Algebra, 2017.

Why do you want to learn linear algebra? Let me know.

2.8 Summary

In this chapter, you discovered why, as a machine learning practitioner, you should deepen your understanding of linear algebra. Specifically, you learned:

Not everyone should learn linear algebra, that it depends where you are in your process of learning machine learning.

5 Reasons why a deeper understanding of linear algebra is required for intermediate machine learning practitioners.

2.8.1 Next

That linear algebra can be fun if approached in the right way.
In the next chapter you will discover 10 concrete examples of machine learning concepts and methods that require an understanding of linear algebra.

Chapter 3

Examples of Linear Algebra in Machine Learning

Linear algebra is a sub-field of mathematics concerned with vectors, matrices and linear transforms. It is a key foundation to the field of machine learning from notations used to describe the operation of algorithms, to the implementation of algorithms in code. Although linear algebra is integral to the field of machine learning, the tight relationship is often left unexplained or explained using abstract concepts such as vector spaces or specific matrix operations. In this chapter, you will discover 10 common examples of machine learning that you may be familiar with that use, require and are really best understood using linear algebra. After reading this chapter, you will know:

The use if linear algebra structures when working with data su
dcha taass etatsb aunladr
images.

Linear algebra concepts when working with data preparation such a
esn ocnoed hinogt
and dimensionality reduction.

The in-grained use of linear algebra notation and methods in sub-fields suc
dhe eaps
l3e.a1r nOinvge, nravtiuerawl language processing and recomn

LIne tt'sh gise tc shtaaprtedr,. we will review 10 obvious and concrete examples of linear algebra in machine learning. I tried to pick examples that you may be familiar with or have even worked with before. They are:

1. Dataset and Data Files

2. Images and Photographs

3. One Hot Encoding

4. Linear Regression

5. Regularization

6. Principal Component Analysis

7. Singular-Value Decomposition

8. Latent Semantic Analysis

9. Recommender Systems

10. Deep Learning

Do you have your own favorite example of linear algebra in machine learning? Let me kn

3.2 DatasetandDataFiles

In machine learning, you fit a model on a dataset. This is the table like set of numbers where each row represents an observation and each column represents a feature of the observation. For example, below is a snippet of the Iris flowers dataset[1]:

```
5.1,3.5,1.4,0.2,Iris
- setosa
4.9,3.0,1.4,0.2,Iris
- setosa
4.7,3.2,1.3,0.2,Iris
- setosa
4.6,3.1,1.5,0.2,Iris
- setosa
5.0,3.6,1.4,0.2,Iris
- setosa
```

Listing 3.1: Sample output of the iris flowers dataset.

This data is in fact a matrix, a key data structure in linear algebra. Further, when you split the data into inputs and outputs to fit a supervised machine learning model, such as measurements and the flower species, you have a matrix (X) and a vector (y). The vector is another key data structure in linear algebra. Each row has the same length, i.e. the same number of columns, therefore we can say that the data is vectorized where rows can be p to a model one at a time or in batch and the model can be pre-configured to expect rows fixed width.

3.3 Images and Photographs

Perhaps you are more used to working with images or photographs in computer vision applications. Each image that you work with is itself a table structure with a width and height and one pixel value in each cell for black and white images or 3 pixel values in each cell for a color image. A photo is yet another example of a matrix from linear algebra. Operations on the image, such as cropping, scaling, shearing and so on are all described using the notation and operations of linear algebra.

1 http://archive.ics.uci.edu/ml/datasets/Iris

3.4 OneHotEncoding

Sometimes you work with categorical data in machine learning. Perhaps the class labels for classification problems, or perhaps categorical input variables. It is common to encode categorical variables to make their easier to work with and learn by some techniques. A popular encoding for categorical variables is the one hot encoding. A one hot encoding is where a table is created to represent the variable with one column for each category and a row for each example in the dataset. A check or one-value is added in the column for the categorical value for a given row, and a zero-value is added to all other columns. For example, the variable color variable with the 3 rows:

```
red
green
blue
...
```

Listing 3.2: Example of a categorical variable.

Might be encoded as:

```
red, green, blue
       1, 0, 0
       0, 1, 0
       0, 0, 1
...
```

Listing 3.3: Example of a one hot encoded categorical variable.

Each row is encoded as a binary vector, a vector with zero or one values and this is an example of a sparse representation, a whole sub-field of linear algebra.

3.5 Linear Regression

Linear regression is an old method from statistics for describing the relationships between variables. It is often used in machine learning for predicting numerical values in simpler regression problems. There are many ways to describe and solve the linear regression problem, i.e. finding a set of coefficients that when multiplied by each of the input variables and added together results in the best prediction of the output variable. If you have used a machine learning tool or library, the most common way of solving linear regression is via a least squares optimization that is solved using matrix factorization methods from linear regression, such as an LU decomposition or an singular-value decomposition or SVD. Even the common way of summarizing the linear regression equation uses linear algebra notation:

y = A

·b (3.1)

W3.h6ere yR ise tgheu olautrpizuat vtaiorianble A is the dataset and b are the model coefficients.

In applied machine learning, we often seek the simplest possible models that achieve the best skill on our problem. Simpler models are often better at generalizing from specific examples

to unseen data. In many methods that involve coefficients, such as regression methods an artificial neural networks, simpler models are often characterized by models that have sm
coefficient values. A technique that is often used to encourage a model to minimize the si of coefficients while it is being fit on data is called regularization. Common implementatio include the L2 and L1 forms of regularization. Both of these forms of regularization are in f a measure of the magnitude or length of the coefficients as a vector and are methods lifte directly from linear algebra called the vector norm.

3.7 Principal Component Analysis

Often a dataset has many columns, perhaps tens, hundreds, thousands or more. Modeling data with many features is challenging, and models built from data that include irrelevant features are often less skillful than models trained from the most relevant data. It is hard to know which features of the data are relevant and which are not. Methods for automatically reducing the number of columns of a dataset are called dimensionality reduction, and perhaps the most popular is method is called the principal component analysis or PCA for short. This method is used in machine learning to create projections of high-dimensional data for both visualization and for training models. The core of the PCA method is a matrix factorization method from linear algebra. The eigendecomposition can be used and more robust implementations may use the singular-value decomposition or SVD.

3.8 Singular-Value Decomposition

Another popular dimensionality reduction method is the singular-value decomposition method or SVD for short. As mentioned and as the name of the method suggests, it is a matrix factorization method from the field of linear algebra. It has wide use in linear algebra and can be used directly in applications such as feature selection, visualization, noise reduction and more. We will see two more cases below of using the SVD in machine learning.

3.9 Latent Semantic Analysis

In the sub-field of machine learning for working with text data called natural language processing, it is common to represent documents as large matrices of word occurrences. For example, the columns of the matrix may be the known words in the vocabulary and rows may be sentences, paragraphs, pages or documents of text with cells in the matrix marked as the count or frequency of the number of times the word occurred. This is a sparse matrix representation of the text. Matrix factorization methods such as the singular-value decomposition can be applied to this sparse matrix which has the effect of distilling the representation down to its most relevant essence. Documents processed in thus way are much easier to compare, query and use as the basis for a supervised machine learning model. This form of data preparation is called Latent Semantic Analysis or LSA for short, and is also known by the name Latent Semantic Indexing or LSI.

3.10 Recommender Systems

Predictive modeling problems that involve the recommendation of products are called recom- mender systems, a sub-field of machine learning. Examples include the recommendation of books based on previous purchases and purchases by customers like you on Amazon, and the recommendation of movies and TV shows to watch based on your viewing history and viewing history of subscribers like you on Netflix. The development of recommender systems is primarily concerned with linear algebra methods. A simple example is in the calculation of the similarity between sparse customer behavior vectors using distance measures such as Euclidean distance or dot products. Matrix factorization methods like the singular-value decomposition are used widely in recommender systems to distill item and user data to their essence for querying and searching and comparison.

3.11 DeepLearning

Artificial neural networks are nonlinear machine learning algorithms that are inspired by elements of the information processing in the brain and have proven effective at a range of problems not least predictive modeling. Deep learning is the recent resurged use of artificial neural networks with newer methods and faster hardware that allow for the development and training of larger and deeper (more layers) networks on very large datasets. Deep learning methods are routinely achieve state-of-the-art results on a range of challenging problems such as machine translation, photo captioning, speech recognition and much more.

At their core, the execution of neural networks involves linear algebra data structures multiplied and added together. Scaled up to multiple dimensions, deep learning methods work with vectors, matrices and even tensors of inputs and coefficients, where a tensor is a matrix with more than two dimensions. Linear algebra is central to the description of deep learning methods via matrix notation to the implementation of deep learning methods such as Google's

T3e.1n2sorFloSwu Pmythmona lribyrary that has the word "tensor" in its name.

In this chapter, you discovered 10 common examples of machine learning that you may be familiar with that use and require linear algebra. Specifically, you learned:

The use of linear algebra structures when working with data su
dcha taass etatsb aunladr
images.

Linear algebra concepts when working with data preparation such a
esn ocnoed hinogt
and dimensionality reduction.

3.12.1 Next

TThhise iins -gthraei neendd u soef othf lein fiearsrt a lpgaerbt,r ain n otthaet ionne xatn d
mlearnniipnugla, ntea taurrraly lsa onfg duaatgae i np rPoyctehsosnin ugs ainngd N reucr

Part III

NumPy

Chapter 4

Introduction to NumPy Arrays

Arrays are the main data structure used in machine learning. In Python, arrays from the NumPy library, called N-dimensional arrays or the ndarray, are used as the primary data structure for representing data. In this tutorial, you will discover the N-dimensional array in NumPy for representing numerical and manipulating data in Python. After completing this tutorial, you will know:

What the ndarray is and how to create and inspect an array in Python.

Key functions for creating new empty arrays and arrays with default values.

How to combine existing arrays to create new arrays.

Let's get started.

4.1 TutorialOverview

This tutorial is divided into 3 parts; they

are: 1. NumPy N-dimensional Array

2. Functions to Create Arrays

3. Combining Arrays

4.2 NumPyN-dimensionalArray

NumPy is a Python library that can be used for scientific and numerical applications and is the tool to use for linear algebra operations. The main data structure in NumPy is the ndarray, which is a shorthand name for N-dimensional array. When working with NumPy, data in an ndarray is simply referred to as an array. It is a fixed-sized array in memory that contains data of the same type, such as integers or floating point values.

The data type supported by an array can be accessed via the dtype attribute on the array. The dimensions of an array can be accessed via the shape attribute that returns a tuple describing the length of each dimension. There are a host of other attributes. A simple way

to create an array from data or simple Python data structures like a list is to use the array function. The example below creates a Python list of 3 floating point values, then creates a ndarray from the list and access the arrays' shape and data type.

```
# create array
from numpy import
array
# create array
l = [1.0, 2.0, 3.0]
a = array(l)
# display array
print(a)
# display array shape
print(a.shape)
# display array data
type print(a.dtype)
```

Listing 4.1: Example of creating an array with the array() function.

Running the example prints the contents of the ndarray, the shape, which is a one-dimensional array with 3 elements, and the data type, which is a 64-bit floating point.

```
[ 1. 2. 3.]
(3,) float64
```

Listing 4.2: Sample output of creating an array wiathrr athy(e)function.

4.3 Functions to Create Arrays

There are more convenience functions for creating fixed-sized arrays that you may encounter or be required to use. Let's look at just a few.

4.3.1 Empty

The empty() function will create a new array of the specified shape. The argument to the function is an array or tuple that specifies the length of each dimension of the array to create. The values or content of the created array will be random and will need to be assigned before use. The example below creates an empty 3 x 3 two-dimensional array.

```
# create empty array
from numpy import
empty a = empty([3,3])
print(a)
```

Listing 4.3: Example of creating an array with the empty() function.

Running the example prints the content of the empty array. Your specific array contents will vary.

```
[[ 0.00000000e+000 0.00000000e+000 2.20802703e-
314] [ 2.20803350e-314 2.20803353e-314 2.20803356e-314]
[ 2.20803359e-314 2.20803362e-314 2.20803366e-314]]
```

Listing 4.4: Sample output of creating an array wietmh tphtey()function.

4.3.2 Zeros

The zeros() function will create a new array of the specified size with the contents filled with zero values. The argument to the function is an array or tuple that specifies the length of each dimension of the array to create. The example below creates a 3 × 5 zero two-dimensional array.

```
# create zero array
from numpy import
zeros a = zeros([3,5])
print(a)
```

Listing 4.5: Example of creating an array with thezeros()function.

Running the example prints the contents of the created zero array.

```
[[ 0. 0. 0. 0.
0.] [ 0. 0. 0. 0.
0.]
[ 0. 0. 0. 0.
0.]]
```

Listing 4.6: Sample output of creating an array wiztehr oths(e)function.

4.3.3 Ones

The ones() function will create a new array of the specified size with the contents filled with one values. The argument to the function is an array or tuple that specifies the length of each dimension of the array to create. The example below creates a 5-element one-dimensional array.

```
# create one array
from numpy import
ones a = ones([5])
print(a)
```

Listing 4.7: Example of creating an array with theones()function.

Running the example prints the contents of the created ones array.

```
[ 1. 1. 1. 1. 1.]
```

Listing 4.8: Sample output of creating an array wiothne tsh(e)function.

4.4Combining Arrays

NumPy provides many functions to create new arrays from existing arrays. Let's look at two of the most popular functions you may need or encounter.

4.4.1 VerticalStack

Given two or more existing arrays, you can stack them vertically using the vstack() function. For example, given two one-dimensional arrays, you can create a new two-dimensional array with two rows by vertically stacking them. This is demonstrated in the example below.

```
# create array with vstack
from numpy import array
from numpy import vstack
# create first array
a1 = array([1,2,3])
print(a1)
# create second array
a2 = array([4,5,6])
print(a2)
# vertical stack
a3 = vstack((a1, a2))
print(a3)
print(a3.shape)
```

Listing 4.9: Example of creating an array from other arrays using the vstack() function.

Running the example first prints the two separately defined one-dimensional arrays. The arrays are vertically stacked resulti
×n g3 alnrr aa yn, ethwe 2 contents and shape of which are printed.

```
[1 2 3]

[4 5 6]

[[1 2 3] [4
5 6]]

(2, 3)
```

Listing 4.10: Sample output of creating an array from other arraysv swtaitchk (t)hefunction

4.4.2 HorizontalStack

Given two or more existing arrays, you can stack them horizontally using the hstack() function. For example, given two one-dimensional arrays, you can create a new one-dimensional array or one row with the columns of the first and second arrays concatenated. This is demonstrated in the example below.

```
# create array with hstack
from numpy import array
from numpy import hstack
# create first array
a1 = array([1,2,3])
print(a1)
# create second array
a2 = array([4,5,6])
print(a2)
# create horizontal stack
a3 = hstack((a1, a2))
print(a3)
print(a3.shape)
```

Listing 4.11: Example of creating an array from other arrays ushinsgta tchke()function.

Running the example first prints the two separately defined one-dimensional arrays. The arrays are then horizontally stacked resulting in a new one-dimensional array with 6 elem
the contents and shape of which are printed.

```
[1 2 3]
[4 5 6]
[123456]
(6,)
```

Listing 4.12: Sample output of creating an array from other arraysh wstiatchk t(h)efunction

4.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Experiment with the different ways of creating arrays to your own sizes or with new data.

Locate and develop an example for 3 additional NumPy functions for creating arrays.

Locate and develop an example for 3 additional NumPy functions for combining arrays.
I
4f y.6ou exFpulorrteh aneyr oRf tehaesdei enxgtensions, I'd lov

This section provides more resources on the topic if you are looking to go deeper.

4.6.1 Bo oks

Python for Data Analysis, 2017.
http://amzn.to/2B1sfXi

Elegant SciPy, 2017.
http://amzn.to/2yujXnT

Guide to NumPy, 2015.
http://amzn.to/2j3kEzd

4.6.2 References

NumPy Reference.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/

The N-dimensional array.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/arrays.ndarray.html

Array creation routines.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/routines.array- creation.html

4.6.3 API

numpy.array() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.array.html

numpy.empty() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.empty.html

numpy.zeros() API.

https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.zeros.html

numpy.ones() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.ones.html

numpy.vstack() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.vstack.html

numpy.hstack() API.

https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.hstack.html

4.7 Summary

In this tutorial, you discovered the N-dimensional array in NumPy for representing numerical and manipulating data in Python. Specifically, you learned:

What the ndarray is and how to create and inspect an array in Python.

Key functions for creating new empty arrays and arrays with default values.

4.7.1 Next

How to combine existing arrays to create new arrays.
In the next chapter you will discover how to slice, index, and reshape NumPy arrays.

Chapter 5

Index, Slice and Reshape NumPy Arrays

Machine learning data is represented as arrays. In Python, data is almost universally represented as NumPy arrays. If you are new to Python, you may be confused by some of the Pythonic ways of accessing data, such as negative indexing and array slicing. In this tutorial, you will discover how to manipulate and access your data correctly in NumPy arrays. After completing this tutorial, you will know:

How to convert your list data to NumPy arrays.

How to access data using Pythonic indexing and slicing.

How to resize your data to meet the

A ePxIpse.ctations of some machine learning

L5e.1t's geTt ustatroterdia.lOverview

This tutorial is divided into 4 parts; they are:

1. From List to Arrays

2. Array Indexing

3. Array Slicing

4. Array Reshaping

5.2 From List to Arrays

In general, I recommend loading your data from file using Pandas or even NumPy functions. This section assumes you have loaded or generated your data by other means and it is now represented using Python lists. Let's look at converting your data in lists to NumPy arrays.

5.2.1 One-DimensionalListtoArray

You may load your data or generate your data and have access to it as a list. You can convert a one-dimensional list of data to an array by calling the array() NumPy function.

```
# create one-dimensional
array from numpy import
array
# list of data
data = [11, 22, 33, 44, 55]
# array of data
data = array(data)
print(data)
print(type(data))
```

Listing 5.1: Example of creating one-dimensional array.

Running the example converts the one-dimensional list to a NumPy array.

```
[11 22 33 44 55]
<class 'numpy.ndarray'>
```

Listing 5.2: Sample output of creating a one-dimensional array.

5.2.2 Two-Dimensional List of Lists to Array

It is more likely in machine learning that you will have two-dimensional data. That is a table of data where each row represents a new observation and each column a new feature. Perhaps you generated the data or loaded it using custom code and now you have a list of lists. Each list represents a new observation. You can convert your list of lists to a NumPy array the same way as above, by calling the array() function.

```
# create two-dimensional
array from numpy import
array
# list of data
data = [[11, 22],
[33, 44],
[55, 66]]
# array of data
data = array(data)
print(data)
print(type(data))
```

Listing 5.3: Example of creating two-dimensional array.

Running the example shows the data successfully converted.

```
[[11 22]
[33 44]
[55 66]]
<class 'numpy.ndarray'>
```

Listing 5.4: Sample output of creating a two-dimensional array.

5.3 ArrayIndexing

Once your data is represented using a NumPy array, you can access it using indexing. Let's look at some examples of accessing data via indexing.

5.3.1 One-DimensionalIndexing

Generally, indexing works just like you would expect from your experience with other program- ming languages, like Java, C#, and C++. For example, you can access elements using the bracket operator [] specifying the zero-offset index for the value to retrieve.

```
# index a one-dimensional array
from numpy import array
# define array
data = array([11, 22, 33, 44, 55]) #
index data
print(data[0])
print(data[4])
```

Listing 5.5: Example of indexing a one-dimensional array.

Running the example prints the first and last values in the array.

```
1
1
5
5
```

Listing 5.6: Sample output from indexing a one-dimensional array.

Specifying integers too large for the bound of the array will cause an error.

```
# index array out of bounds
from numpy import array
# define array
data = array([11, 22, 33, 44, 55]) #
index data
print(data[5])
```

Listing 5.7: Example of an error when indexing a one-dimensional array. Running the example prints the following error:

```
IndexError: index 5 is out of bounds for axis 0 with size 5
```

Listing 5.8: Sample error output from indexing a one-dimensional array.

One key difference is that you can use negative indexes to retrieve values offset from the of the array. For example, the index -1 refers to the last item in the array. The index -2 retur the second last item all the way back to -5 for the first item in the current example.

```
# negative array indexing
from numpy import array
# define array
data = array([11, 22, 33, 44, 55]) #
index data
print(data[-1])
print(data[-5])
```

Listing 5.9: Example of negative indexing a one-dimensional array.

Running the example prints the last and first items in the array.

```
5
5
11
```

Listing 5.10: Sample output from negative indexing a one-dimensional array.

5.3.2 Two-Dimensional Indexing

Indexing two-dimensional data is similar to indexing one-dimensional data, except that a comma is used to separate the index for each dimension.

```
data[0,0]
```

Listing 5.11: Example of indexing a two-dimensional array in Python.

This is different from C-based languages where a separate bracket operator is used for ea dimension.

```
data[0][0]
```

Listing 5.12: Example of indexing a two-dimensional array in C-like languages. For example, we can access the first row and the first column as follows:

```
# index two-dimensional
array from numpy import
array
# define array
data = array([
[11, 22],
[33, 44],
[55, 66]])
# index data
print(data[0,0])
```

Listing 5.13: Example of indexing a two-dimensional array.

Running the example prints the first item in the dataset.

```
11
```

Listing 5.14: Sample output from indexing a two-dimensional array.

If we are interested in all items in the first row, we could leave the second dimension inde empty, for example:

```
# index row of two-dimensional
array from numpy import array
# define array
data = array([
[11, 22],
[33, 44],
[55, 66]])
# index data
print(data[0,])
```

Listing 5.15: Example of indexing the first column of a two-dimensional array. This prints the first row of data.

```
[11 22]
```

Listing 5.16: Sample output from indexing the first column of a two-dimensional array.

5.4 ArraySlicing

So far, so good; creating and indexing arrays looks familiar. Now we come to array slicing, and this is one feature that causes problems for beginners to Python and NumPy arrays. Structures like lists and NumPy arrays can be sliced. This means that a subsequence of the structure can be indexed and retrieved. This is most useful in machine learning when specifying input variables and output variables, or splitting training rows from testing rows. Slicing is specified using the colon operator : with a from and to index before and after the column r

```
deastap[efrcomtiv:teol]y. The slice extends from the from index and ends one
item before
```

the to index.

Listing 5.17: Example of the syntax of array slicing.

5.4.1 One-DimensionalSlicing

You can access all data in an array dimension by specifying the slice ':' with no indexes.

```
# slice a one-dimensional array
from numpy import array
# define array
data = array([11, 22, 33, 44, 55])
print(data[:])
```

Listing 5.18: Example of slicing a one-dimensional array.

Running the example prints all elements in the array.

```
[11 22 33 44 55]
```

Listing 5.19: Sample output from slicing a one-dimensional array.

The first item of the array can be sliced by specifying a slice that starts at index 0 and end
at index 1 (one item before the to index).

```
# slice a subset of a one-dimensional
array from numpy import array
# define array
data = array([11, 22, 33, 44, 55]) print(data[0:1])
```

Listing 5.20: Example of slicing a subset of a one-dimensional array. Running the example returns a sub-array with the first

```
[11] element.
```

Listing 5.21: Sample output from slicing a subset of a one-dimensional array.

We can also use negative indexes in slices. For example, we can slice the last two items in the list by starting the slice at -2 (the second last item) and not specifying a to index; that takes the slice to the end of the dimension.

```
# negative slicing of a one-dimensional array from numpy import array
# define array
data = array([11, 22, 33, 44, 55]) print(data[-2:])
```

Listing 5.22: Example of slicing with a negative index on a one-dimensional array. Running the example returns a sub-array with the last two items only.

```
[44 55]
```

Listing 5.23: Sample output from slicing with a negative index on a one-dimensional arra

5.4.2 Two-Dimensional Slicing

Let's look at the two examples of two-dimensional slicing you are most likely to use in machine learning.

Split Input and Output Features

It is common to split your loaded data into input variables (X) and the output variable (y). We can do this by slicing all rows and all columns up to, but before the last column, then separately indexing the last column. For the input features, we can select all rows and all columns except the last one by

```
Xsp =e [:c, :i-f1y]ing : for in the rows index, and :-1 in the columns index.
```

Listing 5.24: Example of slicing input variables.

For the output column, we can select all rows again using : and index just the last column by specifying the -1 index.

```
y = [:, -1]
```

Listing 5.25: Example of slicing the output variable.

Putting all of this together, we can separate a 3-column 2D dataset into input and output data as follows:

```
# split input and output data
from numpy import array
# define array
data = array([
[11, 22, 33],
[44, 55, 66],
[77, 88, 99]])
# separate data
X, y = data[:, :-1], data[:, -1]
print(X) print(y)
```

Listing 5.26: Example of slicing a dataset into input and output variables.

Running the example prints the separated X and y elements. NXoiste a t2hDa tarray and yis a 1D array.

```
[[11 22]
[44 55]
[77 88]]
[33 66 99]
```

Listing 5.27: Sample output slicing a dataset into input and output variables.

Split Train and Test Rows

It is common to split a loaded dataset into separate train and test sets. This is a splitting of rows where some portion will be used to train the model and the remaining portion will be used to estimate the skill of the trained model. This would involve slicing all columns by specifying : in the second dimension index. The training dataset would be all rows from the beginning to the split point.

```
train = data[:split, :]
```

Listing 5.28: Example of slicing a train set from a dataset.

The test dataset would be all rows starting from the split point to the end of the dimen

```
test = data[split:, :]
```

Listing 5.29: Example of slicing a test set from a dataset.

Putting all of this together, we can split the dataset at the contrived split point of 2.

```
# split train and test data
from numpy import array
# define array
data = array([
[11, 22, 33],
[44, 55, 66],
[77, 88, 99]])
# separate data
split = 2
train,test = data[:split,:],data[split:,:]
print(train)
print(test)
```

Listing 5.30: Example of slicing a dataset into train and test subsets.

Running the example selects the first two rows for training and the last row for the tes

```
[[11 22 33]
[44 55 66]]
[[77 88 99]]
```

Listing 5.31: Sample output slicing a dataset into train and test subsets.

5.5 ArrayReshaping

After slicing your data, you may need to reshape it. For example, some libraries, such as scikit- learn, may require that a one-dimensional array of output variables (y) be shaped as a two- dimensional array with one column and outcomes for each column. Some algorithms, like the Long Short-Term Memory recurrent neural network in Keras, require input to be specified as a three-dimensional array comprised of samples, timesteps, and features. It is important to know how to reshape your NumPy arrays so that your data meets the expectation of specific Python libraries. We will look at these two examples.

5.5.1 DataShape

NumPy arrays have a shapeattribute that returns a tuple of the length of each dimension of the array. For example:

```
# shape of one-dimensional array
from numpy import array
# define array
data = array([11, 22, 33, 44, 55])
print(data.shape)
```

Listing 5.32: Example of accessing shape for a one-dimensional array.

Running the example prints a tuple for the one dimension.

```
(5,)
```

Listing 5.33: Sample output shape for a one-dimensional array.

A tuple with two lengths is returned for a two-dimensional array.

```
# shape of a two-dimensional array
from numpy import array
# list of data
data = [[11, 22],
[33, 44],
[55, 66]]
# array of data
data = array(data) print(data.shape)
```

Listing 5.34: Example of accessing shape for a two-dimensional array.

Running the example returns a tuple with the number of rows and columns.

```
(3, 2)
```

Listing 5.35: Sample output shape for a two-dimensional array.

You can use the size of your array dimensions in the shape dimension, such as specifying parameters. The elements of the tuple can be accessed just like an array, with the 0th index for the number of rows and the 1st index for the number of columns. For example:

```
# row and column shape of two-dimensional
array from numpy import array
# list of data
data = [[11, 22],
```

```
[33, 44],
[55, 66]]
# array of data
data = array(data)
print('Rows: %d' % data.shape[0])
print('Cols: %d' % data.shape[1])
```

Listing 5.36: Example of accessing shape for a two-dimensional array in terms of rows and columns.

Running the example accesses the specific size of each dimension.

```
Rows:
3 Cols:
2
```

Listing 5.37: Sample output for a two-dimensional array in terms of rows and columns. shape

5.5.2 Reshape 1D to 2D Array

It is common to need to reshape a one-dimensional array into a two-dimensional array with one column and multiple arrays. NumPy provides the reshape() function on the NumPy array object that can be used to reshape the data. The reshape() function takes a single argument that specifies the new shape of the array. In the case of reshaping a one-dimensional array into a two-dimensional array with one column, the tuple would be the shape of the array as the first dimension (data.shape[0]) and 1 for the second dimension.

```
data = data.reshape((data.shape[0], 1))
```

Listing 5.38: Example the reshape() function for reshaping from 1D to 2D data.

Putting this all together, we get the following worked example.

```
# reshape 1D array to 2D
from numpy import array
# define array
data = array([11, 22, 33, 44, 55])
print(data.shape)
# reshape
data = data.reshape((data.shape[0], 1))
print(data.shape)
```

Listing 5.39: Example of changing the shape of a one-dimensional array with the reshape() function.

Running the example prints the shape of the one-dimensional array, reshapes the array to have 5 rows with 1 column, then prints this new shape.

```
(5,)
(5, 1)
```

Listing 5.40: Sample output of changing the shape of a one-dimensional array with the reshape() function.

5.5.3 Reshape 2D to 3D Array

It is common to need to reshape two-dimensional data where each row represents a sequence into a three-dimensional array for algorithms that expect multiple samples of one or more time steps and one or more features. A good example is the LSTM recurrent neural network model in the Keras deep learning library. The reshape function can be used directly, specifying the new dimensionality. This is clear with an example where each sequence has multiple time steps with one observation (feature) at each time step. We can use the sizes in the shape attribute on the array to specify the number of samples (rows) and columns (time steps) and fix the number of features at 1.

```
data.reshape((data.shape[0], data.shape[1], 1))
```

Listing 5.41: Example the reshape() function for reshaping from 2D to 3D data.

Putting this all together, we get the following worked example.

```
# reshape 2D array to 3D
from numpy import array
# list of data
data = [[11, 22],
[33, 44],
[55, 66]]
# array of data
data = array(data)
print(data.shape)
# reshape
data = data.reshape((data.shape[0], data.shape[1], 1))
print(data.shape)
```

Listing 5.42: Example of changing the shape of a two-dimensional array with the reshape() function.

Running the example first prints the size of each dimension in the 2D array, reshapes the array, then summarizes the shape of the new 3D array.

```
(3, 2)
(3, 2, 1)
```

Listing 5.43: Sample output of changing the shape of a two-dimensional array reshape() with the function.

5.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Develop one example of indexing, slicing and reshaping your own small data arrays.

Load a small real dataset from CSV file and split it into input and output elements

Load a small real dataset from CSV file and split it into train and test elements.

If you explore any of these extensions, I'd love to know.

5.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

5.7.1 Bo oks

Python for Data Analysis , 2017.
http://amzn.to/2B1sfXi

Elegant SciPy, 2017.
http://amzn.to/2yujXnT

Guide to NumPy, 2015.
http://amzn.to/2j3kEzd

5.7.2 References

NumPy Reference.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/

The N-dimensional array.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/arrays.ndarray.html

Array creation routines.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/routines.array- creation.html

NumPy Indexing.
https://docs.scipy.org/doc/numpy- 1.13.0/user/basics.indexing.html

SciPy Indexing.
https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

Indexing routines.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/routines.indexing.html

5.7.3 API

numpy.array() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.array.html

numpy.reshape() API.
https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html

5.8 Summary

In this tutorial, you discovered how to access and reshape data in NumPy arrays with Python. Specifically, you learned:

How to convert your list data to NumPy arrays.

How to access data using Pythonic indexing and slicing.

How to resize your data to meet the expectations of some machine learning APIs.

5.8.1 Next

In the next chapter you will discover array broadcasting and the rules that govern it in Python.

Chapter 6

NumPy Array Broadcasting

Arrays with different sizes cannot be added, subtracted, or generally be used in arithmetic. A way to overcome this is to duplicate the smaller array so that it has the dimensionality and size as the larger array. This is called array broadcasting and is available in NumPy when performing array arithmetic, which can greatly reduce and simplify your code. In this tutorial, you will discover the concept of array broadcasting and how to implement it in NumPy. After completing this tutorial, you will know:

The problem of arithmetic with arrays with different sizes.

The solution of broadcasting and common examples in one and two dimensions.

The rule of array broadcasting and when broadcasting fails.
Let's get started.

6.1 TutorialOverview

This tutorial is divided into 4 parts; they

are: 1. Limitation with Array Arithmetic

2. Array Broadcasting

3. Broadcasting in NumPy

4. Limitations of Broadcasting

6.2 Limitation with Array Arithmetic

You can perform arithmetic directly on NumPy arrays, such as addition and subtraction. For example, two arrays can be added together to create a new array where the values at each index are added together. For example, an array a can be defined as [1, 2, 3] and array b can be defined as [1, 2, 3] and adding together will result in a new array with the values [2, 4, 6].

```
a = [1, 2, 3]
b = [1, 2, 3]
c=a+b
c=[1+1,2+2,3+3]
```

Strictly, arithmetic may only be performed on arrays that have the same dimensions and dimensions with the same size. This means that a one-dimensional array with the length of 10 can only perform arithmetic with another one-dimensional array with the length 10. This limitation on array arithmetic is quite limiting indeed. Thankfully, NumPy provides a built-in workaround to allow arithmetic between arrays with differing sizes.

6.3 ArrayBroadcasting

Broadcasting is the name given to the method that NumPy uses to allow array arithmetic between arrays with a different shape or size. Although the technique was developed for NumPy, it has also been adopted more broadly in other numerical computational libraries, such as Theano, TensorFlow, and Octave. Broadcasting solves the problem of arithmetic between arrays of differing shapes by in effect replicating the smaller array along the last mismatched dimension.

> Vectors are built from components, which are ordinary numbers. You can think of a vector as a list of numbers, and vector algebra as operations performed on the numbers in the list.
>
> — Broadcasting, SciPy.org.

NumPy does not actually duplicate the smaller array; instead, it makes memory and computationally efficient use of existing structures in memory that in effect achieve the same result. The concept has also permeated linear algebra notation to simplify the explanation of simple operations.

> In the context of deep learning, we also use some less conventional notation. We allow the addition of matrix and a vector, yielding another matrix: C = A+b, where Ci,j = Ai,j + bj. In other words, the vector b is added to each row of the matrix. This shorthand eliminates the need to define a matrix with b copied into each row before doing the addition. This implicit copying of b to many locations is called broadcasting.
>
> — Page 34, Deep Learning, 2016.

6.4Broadcasting in NumPy

We can make broadcasting concrete by looking at three examples in NumPy. The examples in this section are not exhaustive, but instead are common to the types of broadcasting you may see or implement.

6.4.1 Scalar and One-Dimensional Array

A single value or scalar can be used in arithmetic with a one-dimensional array. For example, we can imagine a one-dimensional array a with three values [a1, a2, a3] added to a scalar b.

```
a = [a1, a2, a3] b
```

The scalar will need to be broadcast across the one-dimensional array by duplicating the value it 2 more times.

```
b = [b1, b2, b3]
```

The two one-dimensional arrays can then be added directly.

```
c = a + b
c = [a1 + b1, a2 + b2, a3 + b3]
```

The example below demonstrates this in NumPy.

```
# broadcast scalar to one-dimensional
array from numpy import array
# define array
a = array([1, 2, 3])
print(a)
# define scalar
b = 2
print(b)
# broadcast
c = a + b
print(c)
```

Listing 6.1: Example of broadcasting a scalar to a one-dimensional array in NumPy.

Running the example first prints the defined one-dimensional array, then the scalar, followed by the result where the scalar is added to each value in the array.

```
[1 2 3]

2 [3 4

5]
```

Listing 6.2: Results from broadcasting a scalar to a one-dimensional array in NumPy.

6.4.2 Scalar and Two-Dimensional Array

A scalar value can be used in arithmetic with a two-dimensional array. For example, we can imagine a two-dimensional array A with 2 rows and 3 columns added to the scalar b.

$$A=\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix} \quad (6.1)$$

The scalar will need to be broadcast across each row of the two-dimensional array by duplicating it 5 more times.

$$B=\begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{pmatrix} \quad (6.2)$$

The two two-dimensional arrays can then be added directly.

(6.3)

(C=A+Ba1,1+b1,1a1,2+b1,2a1,3+bC=,a2,1+b2,1a2,2+b2,2a2,3+
b, The example below demonstrates this in NumPy.) (6.4)

1
3
2
3

```
# broadcast scalar to two-dimensional
array from numpy import array
# define array
A = array([
[1, 2, 3],
[1, 2, 3]])
print(A)
# define scalar
b = 2
print(b)
# broadcast
C = A + b
print(C)
```

Listing 6.3: Example of broadcasting a scalar to a two-dimensional array in NumPy.

Running the example first prints the defined two-dimensional array, then the scalar, then the result of the addition with the value 2 added to each value in the array.

```
[[1 2 3] [1
2 3]]

2

[[3 4 5] [3
4 5]]
```

Listing 6.4: Results from broadcasting a scalar to a two-dimensional array in NumPy.

6.4.3 One-DimensionalandTwo-DimensionalArrays

A one-dimensional array can be used in arithmetic with a two-dimensional array. For example, we can imagine a two-dimensional array A with 2 rows and 3 columns added to a one-dimensional array b with 3 values.

a a a
( )
a2,1 a2,2 a2,3
( ) A=1,11,21,3 (6.5)

b=b1b2b3 (6.6)

The one-dimensional array is broadcast across each row of the two-dimensional array by creating a second copy to result in a new two-dimensional array B.

(b,b,)b1,3b2,3
1,2
2,2

1 b
1 b
2
1

B= (6.7)

The two two-dimensional arrays can then be added directly.

C = A + B (6.8)

(a1,1+b1,1a1,2+b1,2a1,3+bC=,a2,1+b2,1a2,2+b2,2a2,3+b3,) (6.9)

The example below demonstrates this in NumPy.

```
# broadcast one-dimensional array to two-dimensional
array from numpy import array
# define two-dimensional array
A = array([
[1, 2, 3],
[1, 2, 3]])
print(A)
# define one-dimensional array
b = array([1, 2, 3])
print(b)
# broadcast
C = A + b
print(C)
```

Listing 6.5: Example of broadcasting a one-dimensional array to a two-dimensional array in NumPy.

Running the example first prints the defined two-dimensional array, then the defined one-dimensional array, followed by the result C where in effect each value in the two-dimensional array is doubled.

```
[[1 2 3] [1
2 3]]

[1 2 3]

[[2 4 6] [2
4 6]]
```

Listing 6.6: Results from broadcasting a one-dimensional to a two-dimensional array in NumPy.

6.5 Limitations of Broadcasting

Broadcasting is a handy shortcut that proves very useful in practice when working with NumPy arrays. That being said, it does not work for all cases, and in fact imposes a strict rule that must be satisfied for broadcasting to be performed. Arithmetic, including broadcasting, can only be performed when the shape of each dimension in the arrays are equal or one has the dimension size of 1. The dimensions are considered in reverse order, starting with the trailing dimension; for example, looking at columns before rows in a two-dimensional case.

This make more sense when we consider that NumPy will in effect pad missing dimensions with a size of 1 when comparing arrays. Therefore, the comparison between a two-dimensional array A with 2 rows and 3 columns and a vector b with 3 elements:

```
A.shape = (2 x 3)
b.shape = (3)
```

In effect, this becomes a comparison between:

```
A.shape = (2 x 3)
b.shape = (1 x 3)
```

This same notion applies to the comparison between a scalar that is treated as an array with the required number of dimensions:

```
A.shape = (2 x 3)
b.shape = (1)
```

This becomes a comparison between:

```
A.shape = (2 x 3)
b.shape = (1 x 1)
```

When the comparison fails, the broadcast cannot be performed, and an error is raised. The example below attempts to broadcast a two-element array to a 2 × 3 array. This comparison is in effect:

```
A.shape = (2 x 3)
b.shape = (1 x 2)
```

We can see that the last dimensions (columns) do not match and we would expect the broadcast to fail. The example below demonstrates this in NumPy.

```
# broadcasting error
from numpy import array
#     define     two-dimensional
array A = array([
[1, 2, 3],
[1, 2, 3]])
print(A.shape)
# define one-dimensional
array b = array([1, 2])
print(b.shape)
# attempt broadcast
C = A + b
print(C)
```

Listing 6.7: Example of broadcasting error in NumPy.

Running the example first prints the shapes of the arrays then raises an error when attempting to broadcast, as we expected.

```
(2, 3)
(2,)
ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

Listing 6.8: Example output of a broadcast error..

6.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Create three new and different examples of broadcasting with NumPy arrays.

Implement your own broadcasting function for manually broadcasting in one and two-dimensional cases.

Benchmark NumPy broadcasting and your own custom broadcasting functions with one and two dimensional cases with very large arrays.

I6f .y7ou expFlourer atnhy eofr t hRese eaxdteinnsiogns, I'd love to know.

This section provides more resources on the topic if you are looking to go deeper.

6.7.1 Bo oks

Chapter 2,Deep Learning, 2016.
http://amzn.to/2EnS7x5

6.7.2 Articles

Broadcasting, NumPy API, SciPy.org.
https://docs.scipy.org/doc/numpy- 1.13.0/user/basics.broadcasting.html

Broadcasting semantics in TensorFlow.
https://www.tensorflow.org/performance/xla/broadcasting

Array Broadcasting in NumPy, EricsBroadcastingDoc.
http://scipy.github.io/old- wiki/pages/EricsBroadcastingDoc

Broadcasting, Theano.
http://deeplearning.net/software/theano/tutorial/broadcasting.html

Broadcasting arrays in NumPy, 2015.
https://eli.thegreenplace.net/2015/broadcasting- arrays- in- numpy/

Broadcasting in Octave.
https://www.gnu.org/software/octave/doc/v4.2.1/Broadcasting.html

6.8 Summary

In this tutorial, you discovered the concept of array broadcasting and how to implement in NumPy. Specifically, you learned:

The problem of arithmetic with arrays with different sizes.

The solution of broadcasting and common examples in one and two dimensions.

6Th.8e .r1u lNe eofx atrray broadcasting and when broadcasting fails.

This is the end of this part, in the next part you will discover vectors and matrices, the main data structures used in linear algebra.

Part IV

Matrices

Chapter 7

Vectors and Vector Arithmetic

Vectors are a foundational element of linear algebra. Vectors are used throughout the field of machine learning in the description of algorithms and processes such as the target variable (y) when training an algorithm. In this tutorial, you will discover linear algebra vectors for machine learning. After completing this tutorial, you will know:

What a vector is and how to define one in Python with NumPy.

How to perform vector arithmetic such as addition, subtraction, multiplication and division.

How to perform additional operations such as dot product and multiplication with a scalar.

Let's get started.

7.1 TutorialOverview

This tutorial is divided into 5 parts; they

are: 1. What is a Vector

2. Defining a Vector

3. Vector Arithmetic

4. Vector Dot Product

5. Vector-Scalar Multiplication

7.2 WhatisaVector

A vector is a tuple of one or more values called scalars.

Vectors are built from components, which are ordinary numbers. You can think of a vector as a list of numbers, and vector algebra as operations performed on the numbers in the list.

— Page 69, No Bullshit Guide To Linear Algebra, 2017.

Vectors are often represented using a lowercase character such as v; for example:

v = (v1,v2,v3) (7.1)

Where v1, v2, v3 are scalar values, often real values.
Vectors are also shown using a vertical representation or a column; for example:

v= (7.2)

v 1v 2v 3

It is common to represent the target variable as a vector with the lowercase y when describing the training of a machine learning algorithm. It is common to introduce vectors using a geometric analogy, where a vector represents a point or coordinate in an n-dimensional space, where n is the number of dimensions, such as 2. The vector can also be thought of as a line from the origin of the vector space with a direction and a magnitude.

These analogies are good as a starting point, but should not be held too tightly as we often consider very high dimensional vectors in machine learning. I find the vector-as-coordinate the most compelling analogy in machine learning. Now that we know what a vector is, let's look at how to define a vector in Python.

7.3 DefiningaVector

We can represent a vector in Python as a NumPy array. A NumPy array can be created from a list of numbers. For example, below we define a vector with the length of 3 and the integer values 1, 2 and 3.

```
# create a vector
from numpy import
array # define vector
v = array([1, 2, 3]) print(v)
```

Listing 7.1: Example of defining a vector.

The example defines a vector with 3 elements. Running the example prints the defined vector.

```
[1 2 3]
```

Listing 7.2: Sample output from defining a vector.

7.4 VectorArithmetic

In this section will demonstrate simple vector-vector arithmetic, where all operations are performed element-wise between two vectors of equal length to result in a new vector with the same length

7.4.1 VectorAddition

Two vectors of equal length can be added together to create a new third vector.

$$c=a+b \quad (7.3)$$

The new vector has the same length as the other two vectors. Each element of the new vector is calculated as the addition of the elements of the other vectors at the same index; for example:

$$c=(a1+b1,a2+b2,a3+b3) \quad (7.4)$$

Or, put another way:

c[0] = a[0] + b[0]

$$c[1] = a[1] + b[1] \quad (7.5)$$

c[2] = a[2] + b[2]

We can add vectors directly in Python by adding NumPy arrays.

```
# vector addition
from numpy import
array # define first
vector
a = array([1, 2, 3])
print(a)
# define second vector
b = array([1, 2, 3])
print(b)
# add vectors
c = a + b
print(c)
```

Listing 7.3: Example of vector addition.

The example defines two vectors with three elements each, then adds them together. Running the example first prints the two parent vectors then prints a new vector that is the addition of the two vectors.

```
[1 2 3]

[1 2 3]

[2 4 6]
```

Listing 7.4: Sample output from vector addition.

7.4.2 VectorSubtraction

One vector can be subtracted from another vector of equal length to create a new third vector.

c=a

$$-b \quad (7.6)$$

As with addition, the new vector has the same length as the parent vectors and each element of the new vector is calculated as the subtraction of the elements at the same indices.

c = (a1

−b1,a2−b2,a3−b3) (7.7)

Or, put another way:

−c[0b[]0 =] a[0]

c[1] = a[1](7.8)

− b[1]

c[2] = a[2]

− b[2]

The NumPy arrays can be directly subtracted in Python.

```
# vector subtraction
from numpy import array
# define first vector
a = array([1, 2, 3]) print(a)
# define second vector
b = array([0.5, 0.5, 0.5])
print(b)
# subtract vectors
c = a - b
print(c)
```

Listing 7.5: Example of vector subtraction.

The example defines two vectors with three elements each, then subtracts the first from the second. Running the example first prints the two parent vectors then prints the new vector that is the first minus the second.

```
[1 2 3]

[ 0.5 0.5 0.5] [

0.5 1.5 2.5]
```

Listing 7.6: Sample output from vector subtraction.

7.4.3 VectorMultiplication

Two vectors of equal length can be multiplied together.

c = a

×b (7.9)

As with addition and subtraction, this operation is performed element-wise to result in a new vector of the same length.

c = (a1

or

×b1,a2×b2,a3×b3) (7.10)

c=(a1b1, a2b23, 3a) b(7.11)

Or, put another way:

$$\begin{aligned} c[0] &= a[0] \times b[0] \\ c[1] &= a[1] \times b[1] \\ c[2] &= a[2] \times b[2] \end{aligned} \tag{7.12}$$

We can perform this operation directly in NumPy.

```
# vector multiplication
from numpy import array
# define first vector
a = array([1, 2, 3])
print(a)
# define second vector
b = array([1, 2, 3])
print(b)
# multiply vectors
c = a * b
print(c)
```

Listing 7.7: Example of vector multiplication.

The example defines two vectors with three elements each, then multiplies the vectors together. Running the example first prints the two parent vectors, then the new vector is printed.

```
[1 2 3]
[1 2 3]
[1 4 9]
```

Listing 7.8: Sample output from vector multiplication.

7.4.4 VectorDivision

Two vectors of equal length can be divided.

$$a _ bc= \tag{7.13}$$

As with other arithmetic operations, this operation is performed element-wise to result in a new vector of the same length.

$$\frac{a1}{b1}\ \frac{a}{b2}\ \frac{}{b3} ,2, a3c=() \tag{7.14}$$

Or, put another way:

$$\begin{aligned} c[0] &= a[0]/b[0] \\ c[1] &= a[1]/b[1] \\ c[2] &= a[2]/b[2] \end{aligned} \tag{7.15}$$

We can perform this operation directly in NumPy.

```
# vector division
from numpy import
array # define first
vector
a = array([1, 2, 3])
print(a)
# define second vector
b = array([1, 2, 3])
print(b)
# divide vectors
c = a / b
print(c)
```

Listing 7.9: Example of vector division.

The example defines two vectors with three elements each, then divides the first by the second. Running the example first prints the two parent vectors, followed by the result of the vector division.

```
[1 2 3]

[1 2 3]

[ 1. 1. 1.]
```

Listing 7.10: Sample output from vector division.

7.5 VectorDotProduct

We can calculate the sum of the multiplied elements of two vectors of the same length to give a scalar. This is called the dot product, named because of the dot operator used when describing the operation.

> The dot product is the key tool for calculating vector projections, vector decompositions, and determining orthogonality. The name dot product comes from the symbol used to denote it.
>
> — Page 110, No Bullshit Guide To Linear Algebra, 2017.

c = a

·b (7.16)

The operation can be used in machine learning to calculate the weighted sum of a vector. The dot product is calculated as follows:

c = (a1

or×b1+a2×b2+a3×b3) (7.17)

c = (a1b1 + a2b2 + a3b3)(7.18)

We can calculate the dot product between two vectors in Python using thedot()function on a NumPy array.

```
# vector dot product
from numpy import array
# define first vector
a = array([1, 2, 3])
print(a)
# define second vector
b = array([1, 2, 3])
print(b)
# multiply vectors
c = a.dot(b)
print(c)
```

Listing 7.11: Example of vector dot product.

The example defines two vectors with three elements each, then calculates the dot product. Running the example first prints the two parent vectors, then the scalar dot product.

```
[1  2  3]

[1  2  3]

14
```

Listing 7.12: Sample output from vector dot product.

7.6 Vector-ScalarMultiplication

A vector can be multiplied by a scalar, in effect scaling the magnitude of the vector. To keep notation simple, we will use lowercase s to represent the scalar value.

c = s

or×v (7.19)

c=sv (7.20)

The multiplication is performed on each element of the vector to result in a new scaled vector of the same length.

c=(s

×v1,s×v2,s×v3) (7.21)

Or, put another way:

×c[0s] = v[0]

c[1] = v[1](7.22)

× s

c[2] = v[2]

× s

We can perform this operation directly with the NumPy array.

```
# vector-scalar multiplication
from numpy import array
# define vector
a = array([1, 2, 3])
```

```
print(a)
# define scalar
s = 0.5
print(s)
#
multiplication
c=s*a
print(c)
```

Listing 7.13: Example of vector-scalar multiplication.

The example first defines the vector and the scalar then multiplies the vector by the scalar. Running the example first prints the parent vector, then scalar, and then the result of multiplying the two together.

```
[1 2 3]

0.5

[ 0.5        1.5]
```

1. Listing 7.14: Sample output from vector-scalar multiplication.

Similarly, vector-scalar addition, subtraction, and division can be performed in the same way.

7.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Create one example using each operation using your own small array data.

Implement each vector arithmetic operation manually for vectors defined as lists.

Search machine learning papers and find 1 example of each operation being used.

If you explore any of these extensions, I'd love to know.

7.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

7.8.1 Bo oks

Section 1.15, Vectors.No Bullshit Guide To Linear Algebra , 2017.
http://amzn.to/2k76D4

Section 2.2, Vector operations.No Bullshit Guide To Linear Algebra , 2017.
http://amzn.to/2k76D4

Section 1.1 Vectors and LinearIntroduction to Linear Algebra, Fifth Edition, Combinations, 2016.
http://amzn.to/2j2J0g4

Section 2.1 Scalars, Vectors, Matrices and Tensors, Deep Learning, 2016.
http://amzn.to/2j4oKuP

Section 1.B Definition of Vector Space, Linear Algebra Done Right , Third Edition, 2015.
http://amzn.to/2BGuEqI

7.8.2 API

numpy.array() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.array.html

numpy.dot() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.dot.html

7.8.3 Articles

Vector space on Wikipedia.
https://en.wikipedia.org/wiki/Vector_space

Dot product on Wikipedia.
https://en.wikipedia.org/wiki/Dot_product

7.9 Summary

In this tutorial, you discovered linear algebra vectors for machine learning. Specifically, you learned:

What a vector is and how to define one in Python with NumPy.

How to perform vector arithmetic such as addition, subtraction, multiplication and division.

7.9.1 NHoewx tto perform additional operations such as dot product and multiplication with a

In the next chapter you will discover vector norms for calculating the magnitude of vectors.

Chapter 8

Vector Norms

Calculating the length or magnitude of vectors is often required either directly as a regularization method in machine learning, or as part of broader vector or matrix operations. In this tutorial, you will discover the different ways to calculate vector lengths or magnitudes, called the vector norm. After completing this tutorial, you will know:

TheL1 normthatiscalculatedasthesumoftheabsolutevaluesofthevector.

The L2 norm that is calculated as the square root of the sum of the squared vector values.

The max norm that is calculated as the maximum vector values.
Let's get started.

8.1 TutorialOverview

This tutorial is divided into 4 parts; they

are: 1. Vector Norm

2. Vector L1 Norm

3. Vector L2 Norm

4. Vector Max Norm

8.2 VectorNorm

Calculating the size or length of a vector is often required either directly or as part of a broader vector or vector-matrix operation. The length of the vector is referred to as the vector norm or the vector's magnitude.

> The length of a vector is a nonnegative number that describes the extent of the vector in space, and is sometimes referred to as the vector's magnitude or the norm.
>
> — Page 112, No Bullshit Guide To Linear Algebra, 2017.

8.3.VectorL1Norm 54 The length of the vector is always a positive number, except for a vector of all zero values.

It is calculated using some measure that summarizes the distance of the vector from the origin of the vector space. For example, the origin of a vector space for a vector with 3 elements is (0,0,0). Notations are used to represent the vector norm in broader calculations and the type of vector norm calculation almost always has its own unique notation. We will take a look at a few common vector norm calculations used in machine learning.

8.3 VectorL1Norm

The length of a vector can be calculated using the L1 norm, where the 1 is a superscript of the L. The notation for the L1 norm of a vector is ||v||1, where 1 is a subscript. As such, this length is sometimes called the taxicab norm or the Manhattan norm.

L1(v) = ||v||1 (8.1)

The L1 norm is calculated as the sum of the absolute vector values, where the absolute value of a scalar uses the notation |a1|. In effect, the norm is a calculation of the Manhattan distance from the origin of the vector space.

||v||1=|a1|+|a2|+|a3| (8.2)

> In several machine learning applications, it is important to discriminate between elements that are exactly zero and elements that are small but nonzero. In these cases, we turn to a function that grows at the same rate in all locations, but retains mathematical simplicity: the L1 norm.
>
> — Pages 39-40, Deep Learning, 2016.

The L1 norm of a vector can be calculated in NumPy using the norm() function with a parameter to specify the norm order, in this case 1.

```
# vector L1 norm
from numpy import array
from numpy.linalg import norm # define vector
a = array([1, 2, 3])
print(a)
# calculate norm
l1 = norm(a, 1)
print(l1)
```

Listing 8.1: Example of calculating the L1 vector norm.

First, a 3-element vector is defined, then the L1 norm of the vector is calculated. Running the example first prints the defined vector and then the vector's L1 norm.

```
[1 2 3]

6.0
```

Listing 8.2: Sample output from calculating the L^1 vector norm.

8.4.VectorL2Norm 55 The L1 norm is often used when fitting machine learning algorithms as a

regularization

method, i.e. a method to keep the coefficients of the model small and, in turn, the model less
complex.

8.4 VectorL2Norm

The length of a vector can be calculated using the L2 norm, where the 2 is a superscript of the L. The notation for the L2 norm of a vector is
||v||2 where 2 is a subscript.

L2(v) =

||v||2 (8.3)

The L2 norm calculates the distance of the vector coordinate from the origin of the vector space. As such, it is also known as the Euclidean norm as it is calculated as the Euclidean distance from the origin. The result is a positive distance value. The L2 norm is calculated as the square root of the sum of the squared ve

√ctorvalues.||v||=2+2+22a1a2a3(8.4)

The L2 norm of a vector can be calculated in NumPy using the norm() function with default parameters.

```
# vector L2 norm
from numpy import array
from numpy.linalg import
norm # define vector
a = array([1, 2, 3])
print(a)
# calculate norm
l2 = norm(a)
print(l2)
```

Listing 8.3: Example of calculating the L2 vector norm.

First, a 3-element vector is defined, then the L2 norm of the vector is calculated. Running the example first prints the defined vector and then the vector's L2 norm.

```
[1 2 3]

3.74165738677
```

Listing 8.4: Sample output from calculating the L2 vector norm.

Like the L1 norm, the L2 norm is often used when fitting machine learning algorithms as a regularization method, e.g. a method to keep the coefficients of the model small and, in turn, the model less complex. By far, the L2 norm is more commonly used than other vector norms in machine learning.

8.5 VectorMaxNorm

The length of a vector can be calculated using the maximum norm, also called max norm. Max norm of a vector is referred to as Linf where inf is a superscript and can be represented with

8.6.Extensions 56 the infinity symbol. The notation for max norm is
||v||inf, where inf is a subscript.
Linf (v) =

||v||inf (8.5)

The max norm is calculated as returning the maximum value of the vector, hence the name.

||v||inf=maxa1,a2,a3 (8.6)

The max norm of a vector can be calculated in NumPy using the norm() function with the order parameter set to inf.

```
# vector max norm
from math import inf
from numpy import array
from numpy.linalg import
norm # define vector
a = array([1, 2, 3])
print(a)
# calculate norm
maxnorm = norm(a, inf)
print(maxnorm)
```

Listing 8.5: Example of calculating the max vector norm.

First, a 3
× 3 vector is defined, then the max norm of the vector is calculated. Running the example first prints the defined vector and then the vector's max norm.

```
[1 2 3]

3.0
```

Listing 8.6: Sample output from calculating the max vector norm.

Max norm is also used as a regularization in machine learning, such as on neural network weights, called max norm regularization.

8.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Create one example using each operation using your own small array data.

Implement each operation manually for vectors defined as lists of lists.

Search machine learning papers and find 1 example of each operation being used.

If you explore any of these extensions, I'd love to know.

8.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

8.7.1 Books

Section 1.2 Lengths and Dot Products,Introduction to Linear Algebra, Fifth Edition, 2016.
http://amzn.to/2j2J0g4

Section 2.5 Norms,Deep Learning, 2016.
http://amzn.to/2j4oKuP

Section 6.A Inner Products and Norms, Linear Algebra Done Right , Third Edition, 2015.
http://amzn.to/2BGuEqI

Lecture 3 Norms,Numerical Linear Algebra, 1997.
http://amzn.to/2BI9kRH

8.7.2 API

numpy.linalg.norm() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.linalg.norm.html

8.7.3 Articles

Norm (mathematics) on Wikipedia.
https://en.wikipedia.org/wiki/Norm_(mathematics)

8.8 Summary

In this tutorial, you discovered the different ways to calculate vector lengths or magnitudes, called the vector norm. Specifically, you learned:

TheL1 normthatiscalculatedasthesumoftheabsolutevaluesofthevector.

The L2 norm that is calculated as the square root of the sum of the squared vector values.

The max norm that is calculated as the maximum vector values.

8.8.1 Next

In the next chapter you will discover matrices and basic matrix arithmetic.

Chapter 9

Matrices and Matrix Arithmetic

Matrices are a foundational element of linear algebra. Matrices are used throughout the field of machine learning in the description of algorithms and processes such as the input data variable (X) when training an algorithm. In this tutorial, you will discover matrices in linear algebra and how to manipulate them in Python. After completing this tutorial, you will know:

What a matrix is and how to define one in Python with NumPy.

How to perform element-wise operations such as addition, subtraction, and the Hadamard pro duct.

How to multiply matrices together and the intuition behind the operation.
Let's get started.

9.1 TutorialOverview

This tutorial is divided into 6 parts; they

are: 1. What is a Matrix

2. Defining a Matrix

3. Matrix Arithmetic

4. Matrix-Matrix Multiplication

5. Matrix-Vector Multiplication

6. Matrix-Scalar Multiplication

9.2 WhatisaMatrix

A matrix is a two-dimensional array of scalars with one or more columns and one or more

rows. A matrix is a two-dimensional array (a table) of numbers.

— Page 115, No Bullshit Guide To Linear Algebra, 2017.

The notation for a matrix is often an uppercase letter, such as A, and entries are referred to by their two-dimensional subscript of row (i) and column (j), such as ai,j. For example, we can define a 3-row, 2-column matrix:

A=((a1,1,a1,2),(a2,1,a2,2),(a3,1,a3,2)) (9.1)

It is more common to see matrices defined using a horizontal notation.

1,1 a1,2

2,1a2,2A= (9.2)

aa3a,1a3,2

A likely first place you may encounter a matrix in machine learning is in model training data comprised of many rows and columns and often represented using the capital letter X. The geometric analogy used to help understand vectors and some of their operations does not hold with matrices. Further, a vector itself may be considered a matrix with one column and multiple rows. Often the dimensions of the matrix are denoted as m and n or m

× n for the number of rows and the number of columns respectively. Now that we know what a matrix is, let's look at defining one in Python.

9.3 DefiningaMatrix

We can represent a matrix in Python using a two-dimensional NumPy array. A NumPy array can be constructed given a list of lists. For example, below is a 2 row, 3 column matrix.

```
# create matrix
from numpy import array
A = array([[1, 2, 3], [4, 5, 6]]) print(A)
```

Listing 9.1: Example of creating a matrix.

Running the example prints the created matrix showing the expected structure.

```
[[1  2  3]
[4 5 6]]
```

Listing 9.2: Sample output from creating a matrix.

9.4 MatrixArithmetic

In this section will demonstrate simple matrix-matrix arithmetic, where all operations are performed element-wise between two matrices of equal size to result in a new matrix with the same size.

9.4.1 MatrixAddition

Two matrices with the same dimensions can be added together to create a new third matrix.

C=A+B (9.3)

The scalar elements in the resulting matrix are calculated as the addition of the elements in each of the matrices being added.

a1,1 + b1,1 a1,2 + b1,2

C=a2,1+b2,1a2,2+b2,2 (9.4)

a3,1 + b3,1 a3,2 + b3,2

Or, put another way:

C[0, 0] = A[0, 0] + B[0, 0]
C[1, 0] = A[1, 0] + B[1, 0]
C[2, 0] = A[2, 0] + B[2, 0]

(9.5)

C[0, 1] = A[0, 1] + B[0, 1]
C[1, 1] = A[1, 1] + B[1, 1]
C[2, 1] = A[2, 1] + B[2, 1]

We can implement this in Python using the plus operator directly on the two NumPy arrays.

```
# matrix addition
from numpy import
array # define first
matrix
A = array([
[1, 2, 3],
[4, 5, 6]])
print(A)
# define second matrix
B = array([
[1, 2, 3],
[4, 5, 6]])
print(B)
# add matrices
C = A + B
print(C)
```

Listing 9.3: Example of matrix addition.

The example first defines two 2

× 3 matrices and then adds them together. Running the example first prints the two parent matrices and then the result of adding them together.

```
[[1 2 3]
[4 5 6]]

[[1 2 3]
[4 5 6]]

[[246]
[ 8 10 12]]
```

Listing 9.4: Sample output from matrix addition.

9.4.2 MatrixSubtraction

Similarly, one matrix can be subtracted from another matrix with the same dimensions.

C = A

−B (9.6)

The scalar elements in the resulting matrix are calculated as the subtraction of the elements in each of the matrices.

a= 1, − b1,1 a1, − b1 ,

C aa1, b2,1 a2, b2 (9.7)

2 − b3,1 a2 − b2 ,,

, ,

Or, put another way:1−2−2

3 3 3

C

− [B0[, 00,] 0 =] A[0, 0]122

C[1, 0] = A[1, 0]

− B[1, 0]

C[2, 0] = A[2, 0]

−B[2,0] (9.8)

C[0, 1] = A[0, 1]

− B[0, 1]

C[1, 1] = A[1, 1]

− B[1, 1]

C[2, 1] =A[2, 1]

− B[2, 1]

We can implement this in Python using the minus operator directly on the two NumPy arrays.

```
# matrix subtraction
from numpy import array
# define first matrix
A = array([
[1, 2, 3],
[4, 5, 6]])
print(A)
# define second matrix
B = array([
[0.5, 0.5, 0.5],
[0.5, 0.5, 0.5]])
print(B)
# subtract matrices
C=A-B
print(C)
```

Listing 9.5: Example of matrix subtraction.

The example first defines two 2 × 3 matrices and then subtracts one from the other. Running the example first prints the two parent matrices and then subtracts the first matrix from the second.

```
[[1 2 3]
[4 5 6]]

[[ 0.5 0.5 0.5]
[ 0.5 0.5 0.5]]

[[ 0.5 1.5 2.5]
```

```
 [ 3.5 4.5 5.5]]
```

Listing 9.6: Sample output from matrix subtraction.

9.4.3 MatrixMultiplication(HadamardProduct)

Two matrices with the same size can be multiplied together, and this is often called element-wise matrix multiplication or the Hadamard product. It is not the typical operation meant when referring to matrix multiplication, therefore a different operator is often used, such as a circle

$$\circ$$

$$C = A \circ B \quad (9.9)$$

As with element-wise subtraction and addition, element-wise multiplication involves the multiplication of elements from each parent matrix to calculate the values in the new matrix.

a= 1, × b1,1 a1, × b1 ,

C aa1, b2,1 a2, b2 (9.10)

2 × b3,1 a2 × b2 ,,

Or, put another way:1×2×2

3 3 3

C

× [B0[, 00,] 0 =] A[0, 0]122

C[1, 0] = A[1, 0]

× B[1, 0]

C[2, 0] = A[2, 0]

×B[2,0] (9.11)

C[0, 1] = A[0, 1]

× B[0, 1]

C[1, 1] = A[1, 1]

× B[1, 1]

C[2, 1] = A[2, 1]

× B[2, 1]

We can implement this in Python using the star operator directly on the two NumPy arrays.

```
# matrix Hadamard product from numpy import array
# define first matrix
A = array([
[1, 2, 3],
[4, 5, 6]])
print(A)
# define second matrix
B = array([
[1, 2, 3],
[4, 5, 6]])
print(B)
# multiply matrices
C = A * B
print(C)
```

Listing 9.7: Example of matrix Hadamard product.

The example first defines two 2 × 3 matrices and then multiplies them together. Running the example first prints the two parent matrices and then the result of multiplying them together with a Hadamard Product.

```
[[1 2 3]
[4 5 6]]

[[1 2 3]
[4 5 6]]

[[1 4 9]
[16 25 36]]
```

Listing 9.8: Sample output from matrix Hadamard product.

9.4.4 Matrix Division

One matrix can be divided by another matrix with the same dimensions.

A

BC= (9.12)

The scalar elements in the resulting matrix are calculated as the division of the elements in each of the matrices.

C (9.13)

Or, put another way:

C[0, 0] = A[0, 0]/B[0, 0]
C[1, 0] = A[1, 0]/B[1, 0]
C[2, 0] = A[2, 0]/B[2, 0]
C[0, 1] = A[0, 1]/B[0, 1]
C[1, 1] = A[1, 1]/B[1, 1]
C[2, 1] = A[2, 1]/B[2, 1] (9.14)

We can implement this in Python using the division operator directly on the two NumPy arrays.

```
# matrix division
from numpy import array # define first matrix
A = array([
[1, 2, 3],
[4, 5, 6]])
print(A)
# define second matrix
B = array([
[1, 2, 3],
[4, 5, 6]])
print(B)
# divide matrices
C = A / B
print(C)
```

Listing 9.9: Example of matrix division.

The example first defines two 2

× 3 matrices and then divides the first from the second matrix. Running the example first prints the two parent matrices and then divides the first matrix by the second.

```
[[1 2 3]
[4 5 6]]

[[1 2 3]
[4 5 6]]

[[ 1. 1. 1.]
[ 1. 1. 1.]]
```

Listing 9.10: Sample output from matrix division.

9.5 Matrix-MatrixMultiplication

Matrix multiplication, also called the matrix dot product is more complicated than the previous operations and involves a rule as not all matrices can be multiplied together.

C = A

or·B (9.15)

C=AB (9.16)

The rule for matrix multiplication is as follows:

The number of columns (n) in the first matrix (A) must equal the number of rows (m) in the second matrix (B).

For example, matrix A has the dimensions m rows and n columns and matrix B has the dimensions n and k. The n columns in A and n rows in B are equal. The result is a new matrix with m rows and k columns.

C(m, k) = A(m, n)

·B(n,k) (9.17)

This rule applies for a chain of matrix multiplications where the number of columns in one matrix in the chain must match the number of rows in the following matrix in the chain.

One of the most important operations involving matrices is multiplication of two matrices. The matrix product of matrices A and B is a third matrix C. In order for this product to be defined, A must have the same number of columns as B has rows.

×If nA aisn do fB s ihsa opfe s mhape n × p, then C is of shape m × p.

— Page 34, Deep Learning, 2016.

The intuition for the matrix multiplication is that we are calculating the dot product between each row in matrix A with each column in matrix B. For example, we can step down rows of column A and multiply each with column 1 in B to give the scalar values in column 1 of C. Below describes the matrix multiplication using matrix notation.

a,=

a1, a1 ,

A 1 a2 (9.18)

2 a2 ,,

(a,b,b1, b2

33B= (9.19)

bb),,12

a1,1 b ,a

1 1

×b1,1+a1,2× 2,1 1,1×b1,2+a1,2×b2,2b,a

C= a2,11,a2

2 2

12×b1,1+a2,2×2,12,1×b1,2+a2,2×b2,2 (9.20)

a3,1

×b1,1+a3,2× 2,1 3,1×b1,2+a3,2×b2,2

We can describe the matrix multiplication operation using array notation.

C[0, 0] = A[0, 0]

× B[0, 0] + A[0, 1] × B[1, 0]

C[1, 0] = A[1, 0]

× B[0, 0] + A[1, 1] × B[1, 0]

C[2, 0] = A[2, 0]

×B[0,0]+A[2,1]×B[1,0] (9.21)

C[0, 1] = A[0, 0]

× B[0, 1] + A[0, 1] × B[1, 1]

C[1, 1] = A[1, 0]

× B[0, 1] + A[1, 1] × B[1, 1]

C[2, 1] = A[2, 0]

× B[0, 1] + A[2, 1] × B[1, 1]

The matrix multiplication operation can be implemented in NumPy using the dot() function. It can also be calculated using the newer @ operator, since Python version 3.5. The example below demonstrates both methods.

```
# matrix dot product
from numpy import array
# define first matrix
A = array([
[1, 2],
[3, 4],
[5, 6]])
print(A)
# define second matrix
B = array([
[1, 2],
[3, 4]])
print(B)
# multiply matrices
C = A.dot(B)
print(C)
# multiply matrices with @
operator D=A@B
print(D)
```

Listing 9.11: Example of matrix-matrix dot product.

The example first defines two 3

× 2 matrices and then calculates their dot product using the dot() function and the @ operator. Running the example first prints the two parent matrices and then the results of the two dot product operations.

```
[[1 2]
[34]
[5 6]]

[[1 2]
[3 4]]

[[ 7 10]
[15 22]
[23 34]]

[[ 7 10]
[15 22]
[23 34]]
```

Listing 9.12: Sample output matrix-matrix dot product.

I recommend using the dot() function for matrix multiplication for now given the newness of the @ operator.

9.6 Matrix-VectorMultiplication

A matrix and a vector can be multiplied together as long as the rule of matrix multiplication is observed. Specifically, that the number of columns in the matrix must equal the number of items in the vector. As with matrix multiplication, the operation can be written using the dot notation. Because the vector only has one column, the result is always a vector.

c = A

·v (9.22)

Or without the dot in a compact form.

c=Av (9.23)

The result is a vector with the same number of rows as the parent matrix.

a1,1 a,
A= a2,1 a, (9.24)
1a23,122 a,

v
()32 (9.25)
v = 1
v2

=a1,1 v1+a, (9.26)

c
a2,1 v×1+2a×,v2×22×v
a3,1 v21×+3a2,×v2

Or, more compactly.

a1,1v1 + a1,2v2
c= a2,1v1+a2,2v2 (9.27)
a3,1v1 + a3,2v2

We can also represent this with array notation.

c[0] = A[0, 0]
× v[0] + A[0, 1] × v[1]
c[1] = A[1, 0] (9.28)
× v[0] + A[1, 1] × v[1]
c[2] = A[2, 0]
× v[0] + A[2, 1] × v[1]

The matrix-vector multiplication can be implemented in NumPy using the dot() function.

```
# matrix-vector multiplication
from numpy import array
# define matrix
A = array([
[1, 2],
[3, 4],
[5, 6]])
print(A)
# define vector
B = array([0.5, 0.5])
print(B)
# multiply
C = A.dot(B)
print(C)
```

Listing 9.13: Example of matrix-vector dot product.

The example first defines a 3 × 2 matrix and a 2 element vector and then multiplies them together. Running the example first prints the parent matrix and vector and then the result of multiplying them together.

```
[[1 2]
[3 4]
[5 6]]

[ 0.5 0.5]

[ 1.5 3.5 5.5]
```

Listing 9.14: Sample output matrix-vector dot product.

9.7 Matrix-ScalarMultiplication

A matrix can be multiplied by a scalar. This can be represented using the dot notation between the matrix and the scalar.

C = A ·b (9.29)

Or without the dot notation.

C=Ab (9.30)

The result is a matrix with the same size as the parent matrix where each element of the matrix is multiplied by the scalar value.

A= a1, a1 2,

1 a2 2 (9.31)

a2, a3 2,,
1
×a3,+a12
C a1,1=a2×
× + a3 2,×b
orb
b
a1,1b + a1,2bb

1,1a3+,1a22,×b,×b (9.32)

C= a2,1b+a2,2b(9.33)

a3,1b + a3,2b

We can also represent this with array notation.

C[0, 0] = A[0, 0]
× b
C[1, 0] = A[1, 0]

×b (9.34)

C[2, 0] = A[2, 0]
× b
C[0, 1] = A[0, 1]
× b
C[1, 1] = A[1, 1]
× b
C[2, 1] = A[2, 1]
× b

This can be implemented directly in NumPy with the multiplication operator.

```
# matrix-scalar multiplication
from numpy import array
# define matrix
A = array([[1, 2], [3, 4], [5, 6]])
print(A) # define scalar
b = 0.5
print(b)
# multiply
C = A * b
print(C)
```

Listing 9.15: Example of matrix-scalar dot product.

The example first defines a 3

× 2 matrix and a scalar and then multiplies them together.

Running the example first prints the parent matrix and scalar and then the result of multiplying them together.

```
[[1 2]
[3 4]
[5 6]]
```

```
0.5

[[ 0.5 1. ]
 [ 1.5 2. ]
 [ 2.5 3. ]]
```

Listing 9.16: Sample output matrix-scalar dot product.

9.8 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Create one example using each operation using your own small array data.

Implement each matrix arithmetic operation manually for matrices defined as lists of lists.

Search machine learning papers and find 1 example of each operation being used.
If you explore any of these extensions, I'd love to know.

9.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

9.9.1 Books

Section 2.3, Matrix operations. No Bullshit Guide To Linear Algebra , 2017.
http://amzn.to/2k76D4

Section 3.3, Matrix multiplication. No Bullshit Guide To Linear Algebra , 2017.
http://amzn.to/2k76D4

Section 1.3 Matrices,Introduction to Linear Algebra, Fifth Edition, 2016.
http://amzn.to/2AZ7R8j

Section 2.4 Rules for Matrix Operations,Introduction to Linear Algebra , Fifth Edition, 2016.
http://amzn.to/2AZ7R8j

Section 2.1 Scalars, Vectors, Matrices and Tensors, Deep Learning, 2016.
http://amzn.to/2j4oKuP

Section 2.2 Multiplying Matrices and Vectors, Deep Learning, 2016.
http://amzn.to/2B3MsuU

Section 3.C Matrices, Linear Algebra Done Right , Third Edition, 2015.
http://amzn.to/2BGuEqI

Lecture 1 Matrix-Vector Multiplication, Numerical Linear Algebra, 1997.
http://amzn.to/2BI9kRH

9.9.2 API

numpy.array() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.array.html

numpy.dot() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.dot.html

9.9.3 Articles

Matrix (mathematics).
https://en.wikipedia.org/wiki/Matrix_(mathematics)

Matrix multiplication on Wikipedia.
https://en.wikipedia.org/wiki/Matrix_multiplication

Hadamard product (matrices) on Wikipedia.
https://en.wikipedia.org/wiki/Hadamard_product_(matrices)

Dot product on Wikipedia.
https://en.wikipedia.org/wiki/Dot_product

9.10 Summary

In this tutorial, you discovered matrices in linear algebra and how to manipulate them in Python. Specifically, you learned:

What a matrix is and how to define one in Python with NumPy.

How to perform element-wise operations such as addition, subtraction, and the Hadamard pro duct.

9Ho.1w0 t.1o Nmuelxtiptly matrices together and the intuition behind the operation.

In the next chapter you will discover a suite of different types of matrices.

Chapter 10

Types of Matrices

A lot of linear algebra is concerned with operations on vectors and matrices, and there are many different types of matrices. There are a few types of matrices that you may encounter again and again when getting started in linear algebra, particularity the parts of linear algebra relevant to machine learning. In this tutorial, you will discover a suite of different types of matrices from the field of linear algebra that you may encounter in machine learning. After completing this tutorial, you will know:

Square, symmetric, triangular, and diagonal matrices that are much as their names suggest.

Identity matrices that are all zero values except along the main diagonal where the values are 1.

Orthogonal matrices that generalize the idea of perpendicular vectors and have useful computational properties.

L

1e0t'.s1 get stTarutedt.orialOverview

This tutorial is divided into 6 parts to cover the main types of matrices; they are:

1. Square Matrix
2. Symmetric Matrix
3. Triangular Matrix
4. Diagonal Matrix
5. Identity Matrix
6. Orthogonal Matrix

10.2 Square Matrix

A square matrix is a matrix where the number of rows (n) is equivalent to the number of columns (m).

n

≡

m (10.1)

The square matrix is contrasted with the rectangular matrix where the number of rows and columns are not equal. Given that the number of rows and columns match, the dimensions are usually denoted as n, e.g. n

× n. The size of the matrix is called the order, so an order 4 square matrix is 4

× 4. The vector of values along the diagonal of the matrix from the top left to the bottom right is called the main diagonal. Below is an example of an order 3 square matrix.

1 2 3

M=123 (10.2)

1 2 3

Square matrices are readily added and multiplied together and are the basis of many simple l1in0e a3r transSfoyrmatimonse, sturcihc a sM roatatironixs (as in the rotations of images).

A symmetric matrix is a type of square matrix where the top-right triangle is the same as

the

bottom-left triangle.

It is no exaggeration to say that symmetric matrices S are the most important matrices the world will ever see — in the theory of linear algebra and also in the applications.

— Page 338, Introduction to Linear Algebra, Fifth Edition, 2016.

To be symmetric, the axis of symmetry is always the main diagonal of the matrix, from the top left to the bottom right. Below is an

exampleofa5×5symmetricmatrix.

1234521234M=

32123 (10.3)4321254321

A symmetric matrix is always square and equal to its own transpose. The transpose is an operation that flips the number of rows and columns. It is explained in more detail in the next lesson.

M=MT (10.4)

10.4 TriangularMatrix

A triangular matrix is a type of square matrix that has all values in the upper-right or lower-left of the matrix with the remaining elements filled with zero values. A triangular matrix with values only above the main diagonal is called an upper triangular matrix. Whereas, a triangular matrix with values only below the main diagonal is called a lower triangular matrix. Below is an example of a 3
× 3 upper triangular matr
ix.

123M=023 (10.5)003

B
× e3l oloww ies ra tnr ieaxample of a 3

ngularmM
atrix.100=120 (10.6)123

NumPy provides functions to calculate a triangular matrix from an existing square matrix. The tril() function to calculate the lower triangular matrix from a given matrix and the triu() to calculate the upper triangular matrix from a given matrix The example below defines
×a 33 square matrix and calculates the lower and upper triangular matrix from it.

```
# triangular matrices
from numpy import array
from numpy import tril
from numpy import triu
# define square matrix
M = array([
[1, 2, 3],
[1, 2, 3],
[1, 2, 3]])
print(M)
# lower triangular matrix
lower = tril(M)
print(lower)
# upper triangular matrix
upper = triu(M)
print(upper)
```

Listing 10.1: Example of creating a triangular matrices.

Running the example prints the defined matrix followed by the lower and upper triangular matrices.

```
[[1 2 3] [1
2 3] [1 2
3]]

[[1 0 0] [1
2 0] [1 2
3]]

[[1 2 3]  [0
2 3]
```

```
[0 0 3]]
```

Listing 10.2: Sample output from creating triangular matrices.

10.5 DiagonalMatrix

A diagonal matrix is one where values outside of the main diagonal have a zero value, where the main diagonal is taken from the top left of the matrix to the bottom right. A diagonal matrix is often denoted with the variable D and may be represented as a full matrix or as a vector of values on the main diagonal.

> Diagonal matrices consist mostly of zeros and have non-zero entries only along the main diagonal.
>
> — Page 40, Deep Learning, 2016.

Below is an example of a 3×3 square diagonal matrix.

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix} \quad (10.7)$$

As a vector, it would be represented as:

$$v = [1, 2, 3] \quad (10.8)$$

Or, with the specified scalar values:

$$d = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad (10.9)$$

A diagonal matrix does not have to be square. In the case of a rectangular matrix, the diagonal would cover the dimension with the smallest length; for example:

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (10.10)$$

NumPy provides the function diag() that can create a diagonal matrix from an existing matrix, or transform a vector into a diagonal matrix. The example below defines a 3 × 3 square matrix, extracts the main diagonal as a vector, and then creates a diagonal matrix from the extracted vector.

```
# diagonal matrix
from numpy import array
from numpy import diag
# define square matrix
M = array([
[1, 2, 3],
[1, 2, 3],
[1, 2, 3]])
print(M)
# extract diagonal vector
d = diag(M)
print(d)
# create diagonal matrix from vector
D = diag(d)
print(D)
```

Listing 10.3: Example of creating a diagonal matrix.

Running the example first prints the defined matrix, followed by the vector of the main diagonal and the diagonal matrix constructed from the vector.

```
[[1 2 3]
[1 2 3]
[1 2 3]]

[1 2 3]

[[1 0 0]
[0 2 0]
[0 0 3]]
```

Listing 10.4: Sample output from creating a diagonal matrix.

10.6 Identity Matrix

An identity matrix is a square matrix that does not change a vector when multiplied. The values of an identity matrix are known. All of the scalar values along the main diagonal (top-left to bottom-right) have the value one, while all other values are zero.

> An identity matrix is a matrix that does not change any vector when we multiply that vector by that matrix.
>
> — Page 36, Deep Learning, 2016.

An identity matrix is often represented using the notation I or with the dimensionality In, where n is a subscript that indicates the dimensionality of the square identity matrix. In some notations, the identity may be referred to as the unit matrix, or U, to honor the one value it contains (this is different from a Unitary matrix). For example, an identity matrix with the size 3 or I3 would be as follows:

1 0 0

I=010 (10.11)

0 0 1

In NumPy, an identity matrix can be created with a specific size using the identity() function. The example below creates an I3 identity matrix.

```
# identity matrix
from numpy import
identity I = identity(3)
print(I)
```

Listing 10.5: Example of creating an identity matrix.

Running the example prints the created identity matrix.

```
[[ 1. 0.
0.] [ 0. 1.
0.]
[ 0. 0.
1.]]
```

Listing 10.6: Sample output from creating an identity matrix.

Alone, the identity matrix is not that interesting, although it is a component in other import matrix operations, such as matrix inversion.

10.7 OrthogonalMatrix

Two vectors are orthogonal when their dot product equals zero. The length of each vector is 1 then the vectors are called orthonormal because they are both orthogonal and normalized.

v

or·w=0 (10.12)

v

·wT=0 (10.13)

This is intuitive when we consider that one line is orthogonal with another if it is perpendicular to it. An orthogonal matrix is a type of square matrix whose columns and rows are orthonormal unit vectors, e.g. perpendicular and have a length or magnitude of 1.

An orthogonal matrix is a square matrix whose rows are mutually orthonormal and whose columns are mutually orthonormal

— Page 41, Deep Learning, 2016.

An Orthogonal matrix is often denoted as uppercase Q.

Multiplication by an orthogonal matrix preserves lengths.

— Page 277, No Bullshit Guide To Linear Algebra, 2017.

The Orthogonal matrix is defined formally as follows:

QT

·Q=Q·QT=I (10.14)

Where Q is the orthogonal matrix, QT indicates the transpose of Q, and I is the identity matrix. A matrix is orthogonal if its transpose is equal to its inverse.

QT=Q−1 (10.15)

Another equivalence for an orthogonal matrix is if the dot product of the matrix and itself equals the identity matrix.

Q

·QT=I (10.16)

Orthogonal matrices are used a lot for linear transformations, such as reflections and permutations. A simple 2 × 2 orthogonal matrix is listed below, which is an example of a reflection matrix or coordinate reflection.

()10Q=0−(10.17)1

The example below creates this orthogonal matrix and checks the above equivalences.

```
# orthogonal matrix
from numpy import array
from numpy.linalg import
inv # define orthogonal
matrix
Q = array([
[1, 0],
[0, -1]])
print(Q)
# inverse equivalence
V = inv(Q)
print(Q.T)
print(V)
# identity equivalence
I = Q.dot(Q.T)
print(I)
```

Listing 10.7: Example of creating an orthogonal matrix.

Running the example first prints the orthogonal matrix, the inverse of the orthogonal matrix, and the transpose of the orthogonal matrix are then printed and are shown to be equivalent. Finally, the identity matrix is printed which is calculated from the dot product of the orthogonal matrix with its transpose.

```
[[ 1 0]
[ 0 -1]]

[[ 1 0]
[ 0 -1]]

[[ 1. 0.]
[-0. -1.]]

[[1 0]
[0 1]]
```

Listing 10.8: Sample output from creating an orthogonal matrix.

Note, sometimes a number close to zero can be represented as -0 due to the rounding of floating point precision. Just take it as 0.0. Orthogonal matrices are useful tools as they are computationally cheap and stable to calculate their inverse as simply their transpose.

10.8 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Modify each example using your own small contrived array data.

Write your own functions for creating each matrix type.

Research one example where each type of array was used in machine learning. If you explore any of these extensions, I'd love to know.

10.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

10.9.1 Bo oks

Section 6.2 Special types of matrices.No Bullshit Guide To Linear Algebra , 2017.
http://amzn.to/2k76D4

Introduction to Linear Algebra, 2016.
http://amzn.to/2j2J0g4

Section 2.3 Identity and Inverse Matrices,Deep Learning, 2016.
http://amzn.to/2B3MsuU

Section 2.6 Special Kinds of Matrices and Vectors, Deep Learning, 2016.
http://amzn.to/2B3MsuU

10.9.2 API

numpy.tril() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.tril.html

numpy.triu() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.triu.html

numpy.diag() API.

https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.diag.html

numpy.identity() API.

https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.identity.html

10.9.3 Articles

Square matrix on Wikipedia.
https://en.wikipedia.org/wiki/Square_matrix

Main diagonal on Wikipedia.
https://en.wikipedia.org/wiki/Main_diagonal

Symmetric matrix on Wikipedia.
https://en.wikipedia.org/wiki/Symmetric_matrix

Triangular Matrix on Wikipedia.
https://en.wikipedia.org/wiki/Triangular_matrix

Diagonal matrix on Wikipedia.
https://en.wikipedia.org/wiki/Diagonal_matrix

Identity matrix on Wikipedia.
https://en.wikipedia.org/wiki/Identity_matrix

Orthogonal matrix on Wikipedia.
https://en.wikipedia.org/wiki/Orthogonal_matrix

10.10 Summary

In this tutorial, you discovered a suite of different types of matrices from the field of linear algebra that you may encounter in machine learning. Specifically, you learned:

Square, symmetric, triangular, and diagonal matrices that are much as their name suggests.

Identity matrices that are all zero values except along the main diagonal where the values are 1.

Orthogonal matrices that generalize the idea of perpendicular vectors and have useful 1c0om.1p0u.t1a Ntioenxalt properties.

In the next chapter you will discover basic operations that you can perform on matrices.

Chapter 11

Matrix Operations

Matrix operations are used in the description of many machine learning algorithms. Some operations can be used directly to solve key equations, whereas others provide useful shorthand or foundation in the description and the use of more complex matrix operations. In this tutorial, you will discover important linear algebra matrix operations used in the description of machine learning methods. After completing this tutorial, you will know:

The Transpose operation for flipping the dimensions of a matrix.

The Inverse operations used in solving systems of linear equations.

The Trace and Determinant operations used as shorthand notation in other matrix op erations.

Let's get started.

11.1 TutorialOverview

This tutorial is divided into 5 parts; they

are: 1. Transpose

2. Inverse

3. Trace

4. Determinant

5. Rank

11.2 Transpose

A defined matrix can be transposed, which creates a new matrix with the number of columns and rows flipped. This is denoted by the superscript T next to the matrix AT.

C=AT (11.1)

An invisible diagonal line can be drawn through the matrix from top left to bottom right on which the matrix can be flipped to give the tr
anspo

se.12A=34 (11.2)

(56)1AT35=(11.3)246

The operation has no effect if the matrix is symmetrical, e.g. has the same number of columns and rows and the same values at the same locations on both sides of the invisible diagonal line.

The columns of AT are the rows of A.

— Page 109, Introduction to Linear Algebra, Fifth Edition, 2016.

We can transpose a matrix in NumPy by calling the T attribute.

```
# transpose matrix
from numpy import
array # define matrix
A = array([
[1, 2],
[3, 4],
[5, 6]])
print(A)
# calculate transpose
C = A.T
print(C)
```

Listing 11.1: Example of creating a transpose of a matrix.

Running the example first prints the matrix as it is defined, then the transposed version.

```
[[1 2] [34]
[5 6]]

[[1 3 5] [2
4 6]]
```

Listing 11.2: Sample output from creating a transpose of a matrix.

The transpose operation provides a short notation used as an element in many matrix op erations.

11.3 Inverse

Matrix inversion is a process that finds another matrix that when multiplied with the matrix, results in an identity matrix. Given a matrix A, find matrix B, such that AB = In or BA = In.

AB=BA=In (11.4)

The operation of inverting a matrix is indicated by a

−1 superscript next to the matrix; for example, A−1. The result of the operation is referred to as the inverse of the original matrix; for example, B is the inverse of A.

B=A−1 (11.5)

A matrix is invertible if there exists another matrix that results in the identity matrix, where not all matrices are invertible. A square matrix that is not invertible is referred to as singular.

Whatever A does, A−1 undoes.

— Page 83, Introduction to Linear Algebra, Fifth Edition, 2016.

The matrix inversion operation is not computed directly, but rather the inverted matrix is discovered through a numerical operation, where a suite of efficient methods may be used, often involving forms of matrix decomposition.

However, A1 is primarily useful as a theoretical tool, and should not actually be used in practice for most software applications.

— Page 37, Deep Learning, 2016.

A matrix can be inverted in NumPy using the inv() function.

```
# invert matrix
from numpy import array
from numpy.linalg import
inv # define matrix
A = array([
[1.0, 2.0],
[3.0, 4.0]])
print(A)
# invert matrix
B = inv(A)
print(B)
# multiply A and B
I = A.dot(B)
print(I)
```

Listing 11.3: Example of creating the inverse of a matrix.

First, we define a small 2

× 2 matrix, then calculate the inverse of the matrix, and then confirm the inverse by multiplying it with the original matrix to give the identity matrix.

Running the example prints the original, inverse, and identity matrices.

```
[[ 1. 2.]
[ 3. 4.]]

[[-2. 1.]
[ 1.5 -0.5]]

[[ 1.00000000e+00 0.00000000e+00]
[ 8.88178420e-16 1.00000000e+00]]
```

Listing 11.4: Sample output from creating the inverse of a matrix.

Note, your specific results may vary given differences in floating point precision on different hardware and software versions. Matrix inversion is used as an operation in solving systems of equations framed as matrix equations where we are interested in finding vectors of unknowns. A good example is in finding the vector of coefficient values in linear regression.

11.4 Trace

A trace of a square matrix is the sum of the values on the main diagonal of the matrix (top-left to bottom-right).

The trace operator gives the sum of all of the diagonal entries of a matrix

— Page 46, Deep Learning, 2016.

The operation of calculating a trace on a square matrix is described using the notation tr(A) where A is the square matrix on which the operation is being performed.

tr(A) (11.6)

The trace is calculated as the sum of the diagonal values; for example, in the case of a 3 × 3 matrix:

tr(A)=a1,1+a2,2+a3,3 (11.7)

Or, using array notation:

$$tr(A) = A[0, 0] + A[1, 1] + A[2, 2] \quad (11.8)$$

We can calculate the trace of a matrix in NumPy using the trace() function.

```
# matrix trace
from numpy import
array from numpy
import trace # define
matrix
A = array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
print(A)
# calculate trace
B = trace(A)
print(B)
```

Listing 11.5: Example of creating the trace of a matrix.

First, a 3 × 3 matrix is created and then the trace is calculated. Running the example, first the array is printed and then the trace.

```
[[1 2 3]
[4 5 6]
[7 8
9]]

15
```

Listing 11.6: Sample output from creating the trace of a matrix.

Alone, the trace operation is not interesting, but it offers a simpler notation and it is used as an element in other key matrix operations.

11.5 Determinant

The determinant of a square matrix is a scalar representation of the volume of the matrix.

> The determinant describes the relative geometry of the vectors that make up the rows of the matrix. More specifically, the determinant of a matrix A tells you the volume of a box with sides given by rows of A.
>
> — Page 119, No Bullshit Guide To Linear Algebra, 2017.

det(IAt)i sn doetantoiotend o bry |A|, where A is the matrix on which we are calculating the the determinant.

det(A) (11.9)

The determinant of a square matrix is calculated from the elements of the matrix. More technically, the determinant is the product of all the eigenvalues of the matrix. Eigenvalues are introduced in the lessons on matrix factorization. The intuition for the determinant is that it describes the way a matrix will scale another matrix when they are multiplied together. For example, a determinant of 1 preserves the space of the other matrix. A determinant of 0 indicates that the matrix cannot be inverted.

> The determinant of a square matrix is a single number. [...] It tells immediately whether the matrix is invertible. The determinant is a zero when the matrix has no inverse.
>
> — Page 247, Introduction to Linear Algebra, Fifth Edition, 2016.

In NumPy, the determinant of a matrix can be calculated using the det() function.

```
# matrix determinant
from numpy import array
from numpy.linalg import
det # define matrix
A = array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
print(A)
# calculate determinant
B = det(A)
print(B)
```

Listing 11.7: Example of creating the determinant of a matrix.

First, a 3 × 3 matrix is defined, then the determinant of the matrix is calculated. Running the example first prints the defined matrix and then the determinant of the matrix.

```
[[1 2 3]
[4 5 6]
[7 8 9]]

-9.51619735393e-16
```

Listing 11.8: Sample output from creating the determinant of a matrix.

Like the trace operation, alone, the determinant operation is not interesting, but it offers a simpler notation and it is used as an element in other key matrix operations.

11.6 Rank

The rank of a matrix is the estimate of the number of linearly independent rows or columns in a matrix. The rank of a matrix M is often denoted as the function rank().

rank(A) (11.10)

An intuition for rank is to consider it the number of dimensions spanned by all of the vectors within a matrix. For example, a rank of 0 suggest all vectors span a point, a rank of 1 suggests all vectors span a line, a rank of 2 suggests all vectors span a two-dimensional plane. The rank is estimated numerically, often using a matrix decomposition method. A common approach is to use the Singular-Value Decomposition or SVD for short. NumPy provides the matrix rank() function for calculating the rank of an array. It uses the SVD method to estimate the rank. The example below demonstrates calculating the rank of a matrix with scalar values and another vector with all zero values.

```
# vector rank
from numpy import array
from numpy.linalg import matrix_rank # rank
v1 = array([1,2,3])
print(v1)
vr1 = matrix_rank(v1)
print(vr1)
# zero rank
v2 = array([0,0,0,0,0])
print(v2)
vr2 = matrix_rank(v2)
print(vr2)
```

Listing 11.9: Example of calculating the rank of vectors.

Running the example prints the first vector and its rank of 1, followed by the second zero vector and its rank of 0.

```
[1 2 3]

1

[00000] 0
```

Listing 11.10: Sample output from creating the rank of vectors.

The next example makes it clear that the rank is not the number of dimensions of the matrix, but the number of linearly independent directions. Three examples of a 2 × 2 matrix are provided demonstrating matrices with rank 0, 1 and 2.

```
# matrix rank
from numpy import array
from numpy.linalg import matrix_rank # rank 0
M0 = array([
[0,0],
[0,0]])
print(M0)
mr0 = matrix_rank(M0)
print(mr0)
# rank 1
M1 = array([
[1,2],
[1,2]])
print(M1)
mr1 = matrix_rank(M1)
print(mr1)
# rank 2
M2 = array([
[1,2],
[3,4]])
print(M2)
mr2 = matrix_rank(M2)
print(mr2)
```

Listing 11.11: Example of creating the rank of matrices.

Running the example first prints a zero 2× 2 with a rank of 0 followed by the rank, then a 2 × 2 matrix with a rank of 1 and finally a 2 × 2 matrix with a rank of 2.

```
[[0 0]
[0 0]]

0

[[1 2]
[1 2]]

1

[[1 2]
[3 4]]

2
```

Listing 11.12: Sample output from creating the rank of matrices.

11.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Modify each example using your own small contrived array data.

Write your own functions to implement one operation.

Research one example where each operation was used in machine learning.

If you explore any of these extensions, I'd love to know.

11.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

11.8.1 Bo oks

Section 3.4 Determinants.No Bullshit Guide To Linear Algebra , 2017.
http://amzn.to/2k76D4

Section 3.5 Matrix inverse. No Bullshit Guide To Linear Algebra , 2017.
http://amzn.to/2k76D4

Section 5.1 The Properties of Determinants,Introduction to Linear Algebra, Fifth Edition, 2016.
http://amzn.to/2AZ7R8j

Section 2.3 Identity and Inverse Matrices,Deep Learning, 2016.
http://amzn.to/2B3MsuU

Section 2.11 The Determinant, Deep Learning, 2016.
http://amzn.to/2B3MsuU

Section 3.D Invertibility and Isomorphic Vector Spaces,Linear Algebra Done Right, Third Edition, 2015.
http://amzn.to/2BGuEqI

Section 10.A Trace,Linear Algebra Done Right , Third Edition, 2015.
http://amzn.to/2BGuEqI

Section 10.B Determinant, Linear Algebra Done Right , Third Edition, 2015.
http://amzn.to/2BGuEqI

11.8.2 API

numpy.ndarray.T API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.ndarray.T.html

numpy.linalg.inv() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.linalg.inv.html

numpy.trace() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.trace.html

numpy.linalg.det() API.

https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.linalg.det.html

numpy.linalg.matrix r̄ank() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.linalg.matrix_rank.html

11.8.3 Articles

Transpose on Wikipedia.
https://en.wikipedia.org/wiki/Transpose

Invertible matrix on Wikipedia.
https://en.wikipedia.org/wiki/Invertible_matrix

Trace (linear algebra) on Wikipedia.
https://en.wikipedia.org/wiki/Trace_(linear_algebra)

Determinant on Wikipedia.
https://en.wikipedia.org/wiki/Determinant

Rank (linear algebra) on Wikipedia.
https://en.wikipedia.org/wiki/Rank_(linear_algebra)

11.9 Summary

In this tutorial, you discovered important linear algebra matrix operations used in the description of machine learning methods. Specifically, you learned:

The Transpose operation for flipping the dimensions of a matrix.

The Inverse operations used in solving systems of linear equations.

The Trace and Determinant operations used as shorthand notation in other matrix op erations.

11.9.1 Next

In the next chapter you will discover sparsity and sparse matrices.

Chapter 12

Sparse Matrices

Matrices that contain mostly zero values are called sparse, distinct from matrices where most of the values are non-zero, called dense. Large sparse matrices are common in general and especially in applied machine learning, such as in data that contains counts, data encodings that map categories to counts, and even in whole subfields of machine learning such as natural language processing. It is computationally expensive to represent and work with sparse matrices as though they are dense, and much improvement in performance can be achieved by using representations and operations that specifically handle the matrix sparsity. In this tutorial, you will discover sparse matrices, the issues they present, and how to work with them directly in Python. After completing this tutorial, you will know:

That sparse matrices contain mostly zero values and are distinct from dense matrices.

The myriad of areas where you are likely to encounter sparse matrices in data, data preparation, and sub-fields of machine learning.

That there are many efficient ways to store and work with sparse matrices and SciPy provides implementations that you can use directly.

L

1e2t'.s1 get stTarutedt.orialOverview

This tutorial is divided into 5 parts; they are:

1. Sparse Matrix
2. Problems with Sparsity
3. Sparse Matrices in Machine Learning
4. Working with Sparse Matrices
5. Sparse Matrices in Python

12.2 Sparse Matrix

A sparse matrix is a matrix that is comprised of mostly zero values. Sparse matrices are distinct from matrices with mostly non-zero values, which are referred to as dense matrices.

> A matrix is sparse if many of its coefficients are zero. The interest in sparsity arises because its exploitation can lead to enormous computational savings and because many large matrix problems that occur in practice are sparse.
>
> — Page 1, Direct Methods for Sparse Matrices, Second Edition, 2017.

The sparsity of a matrix can be quantified with a score, which is the number of zero values in the matrix divided by the total number of elements in the matrix.

$$\text{sparsity} = \frac{\text{count of non-zero elements}}{\text{total elements}} \quad (12.1)$$

Below is an example of a small 3 × 6 sparse matrix.

$$A = \begin{matrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \end{matrix} \quad (12.2)$$

The example has 13 zero values of the 18 elements in the matrix, giving this matrix a sparsity score of 0.722 or about 72%.

12.3 Problems with Sparsity

Sparse matrices can cause problems with regards to space and time complexity.

12.3.1 Space Complexity

Very large matrices require a lot of memory, and some very large matrices that we wish to work with are sparse.

In practice, most large matrices are sparse — almost all entries are zeros.

> — Page 465, Introduction to Linear Algebra, Fifth Edition, 2016.

An example of a very large matrix that is too large to be stored in memory is a link matrix that shows the links from one website to another. An example of a smaller sparse matrix might be a word or term occurrence matrix for words in one book against all known words in English. In both cases, the matrix contained is sparse with many more zero values than data values. The problem with representing these sparse matrices as dense matrices is that memory is required and must be allocated for each 32-bit or even 64-bit zero value in the matrix. This is clearly a waste of memory resources as those zero values do not contain any information.

12.3.2 Time Complexity

Assuming a very large sparse matrix can be fit into memory, we will want to perform operations on this matrix. Simply, if the matrix contains mostly zero-values, i.e. no data, then performing operations across this matrix may take a long time where the bulk of the computation performed will involve adding or multiplying zero values together.

> It is wasteful to use general methods of linear algebra on such problems, because most of the O(N3) arithmetic operations devoted to solving the set of equations or inverting the matrix involve zero operands.
>
> — Page 75, Numerical Recipes: The Art of Scientific Computing, Third Edition, 2007.

This is a problem of increased time complexity of matrix operations that increases with the size of the matrix. This problem is compounded when we consider that even trivial machine learning methods may require many operations on each row, column, or even across the entire matrix, resulting in vastly longer execution times.

12.4 Sparse Matrices in Machine Learning

Sparse matrices turn up a lot in applied machine learning. In this section, we will look at some common examples to motivate you to be aware of the issues of sparsity.

12.4.1 Data

Sparse matrices come up in some specific types of data, most notably observations that record the occurrence or count of an activity. Three examples include:

Whether or not a user has watched a movie in a movie catalog.

Whether or not a user has purchased a product in a product catalog.

1C2o.u4n.t2 o fD thaet nauPmrbeepr oafr laistteionsn of a song in a song catalog.

Sparse matrices come up in encoding schemes used in the preparation of data. Three common examples include:

One hot encoding, used to represent categorical data as sparse binary vectors.

Count encoding, used to represent the frequency of words in a vocabulary for a document

TF-IDF encoding, used to represent normalized word frequency scores in a vocabulary.

12.4.3 AreasofStudy

Some areas of study within machine learning must develop specialized methods to address sparsity directly as the input data is almost always sparse. Three examples include:

Natural language processing for working with documents of text.

Recommender systems for working with product usage within a catalog.

Computer vision when working with images that contain lots of black pixels.

> If there are 100,000 words in the language model, then the feature vector has length 100,000, but for a short email message almost all the features will have count zero.
>
> — Page 22, Artificial Intelligence: A Modern Approach, Third Edition, 2009.

12.5 WorkingwithSparseMatrices

The solution to representing and working with sparse matrices is to use an alternate data structure to represent the sparse data. The zero values can be ignored and only the data or non-zero values in the sparse matrix need to be stored or acted upon. There are multiple data structures that can be used to efficiently construct a sparse matrix; three common examples are listed below.

a value.Dictionary of Keys. A dictionary is used where a row and column index is mapped to

List of Lists. Each row of the matrix is stored as a list, with each sublist containing the column index and the value.

Coordinate List. A list of tuples is stored with each tuple containing the row index, column index, and the value.

There are also data structures that are more suitable for performing efficient operations; two commonly used examples are listed below.

Compressed Sparse Row. The sparse matrix is represented using three one-dimensional arrays for the non-zero values, the extents of the rows, and the column indexes.

Compressed Sparse Column. The same as the Compressed Sparse Row method except the column indices are compressed and read first before the row indices.

The Compressed Sparse Row, also called CSR for short, is often used to represent sparse matrices in machine learning given the efficient access and matrix multiplication that it supports.

12.6 Sparse Matrices in Python

SciPy provides tools for creating sparse matrices using multiple data structures, as well as tools for converting a dense matrix to a sparse matrix. Many linear algebra NumPy and SciPy functions that operate on NumPy arrays can transparently operate on SciPy sparse arrays. Further, machine learning libraries that use NumPy data structures can also operate transparently on SciPy sparse arrays, such as scikit-learn for general machine learning and Keras for deep learning.

A dense matrix stored in a NumPy array can be converted into a sparse matrix using the CSR representation by calling the csr matrix() function. In the example below, we define a 3 ×6 sparse matrix as a dense array (e.g. an ndarray), convert it to a CSR sparse representation, and then convert it back to a dense array by calling the todense() function.

```
# sparse matrix
from numpy import array
from scipy.sparse import csr_matrix
# create dense matrix
A = array([
[1, 0, 0, 1, 0, 0],
[0, 0, 2, 0, 0, 1],
[0, 0, 0, 2, 0, 0]])
print(A)
# convert to sparse matrix (CSR
method) S = csr_matrix(A)
print(S)
# reconstruct dense matrix
B = S.todense()
print(B)
```

Listing 12.1: Example of converting between dense and sparse matrices.

Running the example first prints the defined dense array, followed by the CSR representation, and then the reconstructed dense matrix.

```
[[100100]
[002001]
[000200]]

(0, 0) 1
(0, 3) 1
(1, 2) 2
(1, 5) 1
(2, 3) 2

[[100100]
[002001]
[000200]]
```

Listing 12.2: Sample output from converting between dense and sparse matrices.

NumPy does not provide a function to calculate the sparsity of a matrix. Nevertheless, we can calculate it easily by first finding the density of the matrix and subtracting it from one. The number of non-zero elements in a NumPy array can be given by the count nonzero() function and the total number of elements in the array can be given by the size property of the array. Array sparsity can therefore be calculated as

```
sparsity = 1.0 - count_nonzero(A) / A.size
```

Listing 12.3: Example of the manual sparsity calculation.

The example below demonstrates how to calculate the sparsity of an array.

```
# sparsity calculation
from numpy import array
from numpy import count_nonzero
# create dense matrix
A = array([
[1, 0, 0, 1, 0, 0],
[0, 0, 2, 0, 0, 1],
[0, 0, 0, 2, 0, 0]])
print(A)
# calculate sparsity
sparsity = 1.0 - count_nonzero(A) / A.size
print(sparsity)
```

Listing 12.4: Example of calculating sparsity.

Running the example first prints the defined sparse matrix followed by the sparsity of the matrix.

```
[[100100]
[002001]
[000200]]

0.7222222222222222
```

Listing 12.5: Sample output from calculating sparsity.

12.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Develop your own examples for converting a dense array to sparse and calculating sparsity.

Develop an example for the each sparse matrix representation method supported by SciPy.

Select one sparsity representation method and implement it yourself from scratch.

If you explore any of these extensions, I'd love to know.

12.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

12.8.1 Books

Introduction to Linear Algebra, Fifth Edition, 2016.
http://amzn.to/2AZ7R8j

Section 2.7 Sparse Linear Systems,Numerical Recipes: The Art of Scientific Computing, Third Edition, 2007.
http://amzn.to/2CF5atj

Artificial Intelligence: A Modern Approach, Third Edition, 2009.
http://amzn.to/2C4LhMW

Direct Methods for Sparse Matrices, Second Edition, 2017.
http://amzn.to/2DcsQVU

12.8.2 API

Sparse matrices (scipy.sparse) API.
https://docs.scipy.org/doc/scipy/reference/sparse.html

scipy.sparse.csr_matrix() API.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html

numpy.count_nonzero() API.
https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.count_nonzero.html

numpy.ndarray.size API.
https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.ndarray.size.html

12.8.3 Articles

Sparse matrix on Wikipedia.
https://en.wikipedia.org/wiki/Sparse_matrix

12.9 Summary

In this tutorial, you discovered sparse matrices, the issues they present, and how to work with them directly in Python. Specifically, you learned:

That sparse matrices contain mostly zero values and are distinct from dense matrices.

The myriad of areas where you are likely to encounter sparse matrices in data, data preparation, and sub-fields of machine learning.

That there are many efficient ways to store and work with sparse matrices and SciPy provides implementations that you can use directly.

12.9.1 Next

In the next chapter you will discover tensors and tensor arithmetic.

Chapter 13

Tensors and Tensor Arithmetic

In deep learning it is common to see a lot of discussion around tensors as the cornerstone data structure. Tensor even appears in name of Google's flagship machine learning library: TensorFlow. Tensors are a type of data structure used in linear algebra, and like vectors and matrices, you can calculate arithmetic operations with tensors. In this tutorial, you will discover what tensors are and how to manipulate them in Python with NumPy. After completing this tutorial, you will know:

That tensors are a generalization of matrices and are represented using n-dimensional arrays.

How to implement element-wise operations with tensors.

How to perform the tensor product.

Let's get started.

13.1 TutorialOverview

This tutorial is divided into 3 parts; they

are: 1. What are Tensors

2. Tensors in Python

3. Tensor Arithmetic

4. Tensor Product

13.2 WhatareTensors

A tensor is a generalization of vectors and matrices and is easily understood as a multidimensional array.

In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a tensor.

— Page 33, Deep Learning, 2016.

A vector is a one-dimensional or first order tensor and a matrix is a two-dimensional or second order tensor. Tensor notation is much like matrix notation with a capital letter representing a tensor and lowercase letters with subscript integers representing scalar values within the tensor. For example, below defines a 3 × 3 × 3 three-dimensional tensor T with dimensions index as ti,j,k .

11,1 t1,2,1 t1,31 , 11,2 t1,2,2 t1,32 , 11,3 t1,2,3 t1,33 ,
21,1t2,2,1t2,3121,2t2,2,2t2,3221,3t2,2,3t2,33T= , , (13.1)
t,t3,1t,,1t3,2,1t3,31t,, t,, 3t1,,2t3,2,2t3,32t,, t,, 3t1,,3t3,2,3t3,33,,

Many of the operations that can be performed with scalars, vectors, and matrices can be reformulated to be performed with tensors. As a tool, tensors and tensor algebra is widely used in the fields of physics and engineering. Some operations in machine learning such as the training and operation of deep learning models can be described in terms of tensors.

13.3 Tensors in Python

Like vectors and matrices, tensors can be represented in Python using the N-dimensional array (ndarray). A tensor can be defined in-line to the constructor of array() as a list of lists. The example below defines a 3 × 3 × 3 tensor as a NumPy ndarray. Three dimensions is easier to wrap your head around. Here, we first define rows, then a list of rows stacked as columns, then a list of columns stacked as levels in a cube.

```
# create tensor
from numpy import array
T = array([
[[1,2,3], [4,5,6], [7,8,9]],
        [[11,12,13], [14,15,16], [17,18,19]],
            [[21,22,23], [24,25,26], [27,28,29]]])
print(T.shape)
print(T)
```

Listing 13.1: Example of creating a tensor.

Running the example first prints the shape of the tensor, then the values of the tensor itself. You can see that, at least in three-dimensions, the tensor is printed as a series of matrices, one for each layer. For this 3D tensor, axis 0 specifies the level (like height), axis 1 specifies the column, and axis 2 specifies the row.

```
(3, 3, 3)
[[[ 1 2 3]
    [4 5 6]
        [789]]

   [[11 12 13]
   [14 15 16]
      [17 18 19]]

  [[21 22 23]
  [24 25 26]
```

```
[27 28 29]]]
```

Listing 13.2: Sample output from creating a tensor.

13.4 TensorArithmetic

As with matrices, we can perform element-wise arithmetic between tensors. In this section, we will work through the four main arithmetic operations.

13.4.1 Tensor Addition

The element-wise addition of two tensors with the same dimensions results in a new tensor with the same dimensions where each scalar value is the element-wise addition of the scalars in the parent tensors.

(1a1,, a12,1 a,13)(,a,, 11,2 a12,2 a1,32),,
1,22,1a,121,2,22,2aA= , (13.2)2,32
2 1 a 2 3)(a,,b,,,b, a),,
B(a,,b1=11 ,,b,,,12,1b11311,2,12,2b13(13.3)
1 22,1 1 21,2 22,2 2
2 1 b , A + B2 3 b , 2 3
1b1b
, ,=) (2, , (13.4)
b C
,b,
C(a1,1,1+b1,1,1a1,2,1+b1,2,1a1,3,1+b1,3,1a1,1,2+b=,,a2,1,1+11b,2,1,a11a,22,,2+,1b+1b22,2,2,1aa12,3,2)3
21,2 a2,22, 22,2 a2,32 (13.5)
In NumPy, we can add tensors directly by adding arrays.,+b

```
# tensor addition
from numpy import array
# define first tensor
A = array([
[[1,2,3], [4,5,6], [7,8,9]],
        [[11,12,13], [14,15,16], [17,18,19]],
            [[21,22,23], [24,25,26], [27,28,29]]])
# define second tensor
B = array([
[[1,2,3], [4,5,6], [7,8,9]],
        [[11,12,13], [14,15,16], [17,18,19]],
            [[21,22,23], [24,25,26], [27,28,29]]])
# add tensors
C = A + B
print(C)
```

Listing 13.3: Example of adding tensors.

Running the example prints the addition of the two parent tensors.

```
[[[ 2 4 6]
  [ 8 10 12]
  [14 16 18]]

 [[22 24 26]
  [28 30 32]
  [34 36 38]]

 [[42 44 46]
  [48 50 52]
  [54 56 58]]]
```

Listing 13.4: Sample output from adding tensors.

13.4.2 Tensor Subtraction

The element-wise subtraction of one tensor from another tensor with the same dimensions results in a new tensor with the same dimensions where each scalar value is the element-wise subtraction of the scalars in the parent tensors.

(1a1,, a12,1 a,13)(,a,, 11,2 a12,2 a1,32),,
,a,,aA= , (13.6)122,1121,222,22,32
2 1 a 2 3)(a,,b,,,b, a),,

B(a,,b1=11 ,,b,,,12,1b11311,2,12,2b13(13.7)

1 22,1 1 21,2 22,2 2
2 1 b , A2 3 – B b , 2 3
1 b
,1= b2

, , , (13.8)

b C

,)(,a,,,a, b,

C = (a,a1,1,1 – b1,1,1 a1,2,1 – b1,2,1 a1,3,1 – b1,31 11,2 – b1,1,2 a1,2,2 – b1,2,2 a1,3,2 – b1,3,2)
21,1 – b2,1,1 a2,2,1 – b2,2,1 a2,3,1 – b2,31 21,2 – b2,1,2 a2,2,2 – b2,2,2 a2,3,2 – b2,3,2
(13.9)

In NumPy, we can subtract tensors directly by subtracting arrays.

```
# tensor subtraction
from numpy import array
# define first tensor
A = array([
[[1,2,3], [4,5,6], [7,8,9]],
        [[11,12,13], [14,15,16], [17,18,19]],
            [[21,22,23], [24,25,26], [27,28,29]]])
# define second tensor
B = array([
[[1,2,3], [4,5,6], [7,8,9]],
        [[11,12,13], [14,15,16], [17,18,19]],
            [[21,22,23], [24,25,26], [27,28,29]]])
# subtract tensors
C = A - B
print(C)
```

Listing 13.5: Example of subtracting tensors.

Running the example prints the result of subtracting the first tensor from the second.

```
[[[0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]]]
```

Listing 13.6: Sample output from subtracting tensors.

13.4.3 Tensor Hadamard Product

The element-wise multiplication of one tensor with another tensor with the same dimensions results in a new tensor with the same dimensions where each scalar value is the element-wise multiplication of the scalars in the parent tensors. As with matrices, the operation is referred to as the Hadamard Product to differentiate it from tensor multiplication. Here, we will use the

∘

operator to indicate the Hadamard product operation between tensors.

(1a1,, a12,1 a13)(,a,, 11,2 a12,2 a1,32),,

1,22,1,121,2,22,2a2,32A= , (13.10)

2 1 a a 2 3)(a,,b,,,b, a),,

B(a,,b1=11 ,,b,,,12,1,11311,2,12,2b13(13.11)

1 22,1 1 21,2 22,2 2

2 1 b b A2 3 ∘ B b , 2 3

1 1= b (13.12)2

, , , ,

b

,b,b

(1 1, × 1,1,1 1,21, × 1,2,1 1,31,)(,a,,a, ,, ,

C=a,a,baba×b,1,311,1×b1,1,2a1,22×b1,2,2a1,32×bC1,3,2)

1, b2,1,1 a2,21, b2,2,1 a2,31, b2,31 2, b2,1,2 a2,22, b2,2,2 a2,32, b2,3,2

2 1 × × × 2 1 × × × (13.13)

In NumPy, we can multiply tensors directly by mult2iplying arrays.

```
# tensor Hadamard product
from numpy import array
# define first tensor
A = array([
[[1,2,3], [4,5,6], [7,8,9]],
        [[11,12,13], [14,15,16], [17,18,19]],
            [[21,22,23], [24,25,26], [27,28,29]]])
# define second tensor
B = array([
[[1,2,3], [4,5,6], [7,8,9]],
        [[11,12,13], [14,15,16], [17,18,19]],
            [[21,22,23], [24,25,26], [27,28,29]]])
# multiply tensors
C = A * B
print(C)
```

Listing 13.7: Example of tensor Hadamard product.

Running the example prints the result of multiplying the tensors.

```
[[[ 1 4 9]
[ 16 25 36]
      [ 49 64 81]]

[[121 144 169]
    [196 225 256]
   [289 324 361]]

[[441 484 529]
    [576 625 676]
     [729 784 841]]]
```

Listing 13.8: Sample output from tensor Hadamard product.

13.4.4 Tensor Division

The element-wise division of one tensor with another tensor with the same dimensions results in a new tensor with the same dimensions where each scalar value is the element-wise division of the scalars in the parent tensors.

(1a1,, a12,1 a1,3)(a, 11,2 a12,2 a1,32),,

,,aA= , (13.14)122,1,121,222,22,32

2 1 a a2,3 a),,

B(a,,b1=11 ,,b,,,12,1,113)(a,b, ,b11,,2,12,2b13(13.15)

1 22,1 1 21,2 22,2 2

2 1 b b2, A3= b , 2 3

1 1 b2)

(abab, , , , (13.16)

1,1,1 1,2,1 1,3,1 1,1,2 1,2,2b

,ab,a,)B(a, babb,aa1,3,2

C=1,1,11,2,11,3,11,1,21,2,22,b1,12,2b,12,3,12,1,22,b2,2b1,3,2a Ca a a2,3,2 (13.17)

2,1,1 2,2,1 2,3,1 2,1,2 2,2,2bbbb2,3,2

In NumPy, we can divide tensors directly by dividing arrays.

```
# tensor division
from numpy import array
# define first tensor
A = array([
[[1,2,3], [4,5,6], [7,8,9]],
          [[11,12,13], [14,15,16], [17,18,19]],
              [[21,22,23], [24,25,26], [27,28,29]]])
# define second tensor
B = array([
[[1,2,3], [4,5,6], [7,8,9]],
          [[11,12,13], [14,15,16], [17,18,19]],
              [[21,22,23], [24,25,26], [27,28,29]]])
# divide tensors
C = A / B
```

```
print(C)
```

Listing 13.9: Example of diving tensors.

Running the example prints the result of dividing the tensors.

```
[[[ 1. 1. 1.]
  [ 1. 1. 1.]
  [ 1. 1. 1.]]

 [[ 1. 1. 1.]
  [ 1. 1. 1.]
  [ 1. 1. 1.]]

 [[ 1. 1. 1.]
  [ 1. 1. 1.]
  [ 1. 1. 1.]]]
```

Listing 13.10: Sample output from dividing tensors.

13.5 Tensor Product

The tensor product operator is often denoted as a circle with a small x in the middle. We will denote it here as $\otimes$. Given a tensor A with q dimensions and tensor B with r dimensions, the product of these tensors will be a new tensor with the order of q + r or, said another way, q + r dimensions. The tensor product is not limited to tensors, but can also be performed on matrices and vectors, which can be a good place to practice in order to develop the intuition for higher dimensions. Let's take a look at the tensor product for vectors.

()a1

$$a= \tag{13.18}$$

$$(a2)bb= \tag{13.19}$$

$$1b2\ C=a \otimes b \tag{13.20}$$

$$C \quad () \quad b1 \quad a1\times b=(2) \quad a12\times bb2 \tag{13.21}$$

Or, unrolled:

$$2()ababa=21a\times bCa\ b121\times \times 1\ 2\ \times \tag{13.22}$$

Let's take a look at the tensor product for matrices.

$$A=( )a1,1a1,2a2,1a2,2 \tag{13.23}$$

Bb= 1,(1b1,)

b2,1 2 (13.24)

C = A b2,

⊗ 2

(13.25)

b1,1,) b1,2B(×

(

b2,1, b2,2a1,1,1),11×12

b1,1,) b1,2(bb×b2b,1b2(13.26)

aC=(b2,1, b2,2

,1,1,221×22

b),

,a, ’2,1 2

Or, unrolled:a,1

b 2

b1,1 a1,1 × b1,2 a

, 1,2 × b1,1 ,a12

b a ,×b1,2,×b

C,aa= 1,1× 2,1 1,1 × b2,2 a1,2 × b2,1 ba22

bbaba2,2,×b(13.27)1,1a2,1×1,22,2×1,11

b a b a 1,2,×b2,2

1,1×a2,1×ab22,1,1×a2,1× 2,2 2,2× 2,1 2

The tensor product can be implemented in NumPy usi2ng the tensordot() function. The function takes as arguments the two tensors to be multiplied a2nd the axis on which to sum the products over, called the sum reduction. To calculate the tenso2r product, also called the tensor dot product in NumPy, the axis must be set to 0. In the examp2le below, we define two order-1 tensors (vectors) with and calculate the tensor product.

```
# tensor product
from numpy import array
from numpy import tensordot # define first vector
A = array([1,2])
# define second vector
B = array([3,4])
# calculate tensor product
C = tensordot(A, B, axes=0) print(C)
```

Listing 13.11: Example of tensor product.

Running the example prints the result of the tensor product. The result is an order-2 tensor ×(m2.atrix) with the lengths 2

```
[[3 4]
[6 8]]
```

Listing 13.12: Sample output from tensor product.

The tensor product is the most common form of tensor multiplication that you may encounter, but there are many other types of tensor multiplications that exist, such as the tensor dot product and the tensor contraction.

13.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Update each example using your own small contrived tensor array data.

Implement three other types of tensor multiplication not covered in this tutorial with small vector or matrix data.

Write your own functions to implement each tensor arithmetic operation.

If you explore any of these extensions, I'd love to know.

13.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

13.7.1Bo oks

A Student's Guide to Vectors and Tensors , 2011.
http://amzn.to/2kmUvvF

Chapter 12, Special Topics,Matrix Computations, 2012.
http://amzn.to/2B9xnLD

Tensor Algebra and Tensor Analysis for Engineers , 2015.
http://amzn.to/2C6gzCu

13.7.2 API

The N-dimensional array.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/arrays.ndarray.html

numpy.tensordot() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.tensordot.html

13.7.3 Articles

Tensor algebra on Wikipedia.
https://en.wikipedia.org/wiki/Tensor_algebra

Tensor on Wikipedia.
https://en.wikipedia.org/wiki/Tensor

Tensor product on Wikipedia.
https://en.wikipedia.org/wiki/Tensor_product

Outer product on Wikipedia.
https://en.wikipedia.org/wiki/Outer_product

13.8 Summary

In this tutorial, you discovered what tensors are and how to manipulate them in Python with NumPy. Specifically, you learned:

That tensors are a generalization of matrices and are represented using n-dimensional arrays.

How to implement element-wise operations with tensors.

1Ho3w.8 to. 1pe Nrfoerxmt the tensor product.

This was the end of the part on matrices, next is the part on matrix factorization, starting with a gentle introduction to matrix decomposition methods.

Part V

Factorization

Chapter 14

Matrix Decompositions

Many complex matrix operations cannot be solved efficiently or with stability using the limited precision of computers. Matrix decompositions are methods that reduce a matrix into constituent parts that make it easier to calculate more complex matrix operations. Matrix decomposition methods, also called matrix factorization methods, are a foundation of linear algebra in computers, even for basic operations such as solving systems of linear equations, calculating the inverse, and calculating the determinant of a matrix. In this tutorial, you will discover matrix decompositions and how to calculate them in Python. After completing this tutorial, you will know:

What a matrix decomposition is and why these types of operations are important.

How to calculate an LU and QR matrix decompositions in Python.

How to calculate a Cholesky matrix decomposition in Python.

1Le4t's. 1get stTarutetdo.rialOverview

This tutorial is divided into 3 parts; they are:

1. What is a Matrix Decomposition

2. LU Decomposition

3. QR Decomposition

4. Cholesky Decomposition

14.2 WhatisaMatrixDecomposition

A matrix decomposition is a way of reducing a matrix into its constituent parts. It is an approach that can simplify more complex matrix operations that can be performed on the decomposed matrix rather than on the original matrix itself. A common analogy for matrix decomposition is the factoring of numbers, such as the factoring of 10 into 2 ×5. For this reason, matrix decomposition is also called matrix factorization. Like factoring real values, there are

14.3.LUDecomposition 110 many ways to decompose a matrix, hence there are a range of

different matrix decomposition

techniques. Two simple and widely used matrix decomposition methods are the LU matrix decomposition and the QR matrix decomposition. Next, we will take a closer look at each of these methods.

14.3 LU Decomposition

The LU decomposition is for square matrices and decomposes a matrix into L and U components.

A=L

Or, without the dot notation.·U (14.1)

A=LU (14.2)

Where A is the square matrix that we wish to decompose, L is the lower triangle matrix and U is the upper triangle matrix.

The factors L and U are triangular matrices. The factorization that comes from elimination is A = LU.

— Page 97, Introduction to Linear Algebra, Fifth Edition, 2016.

The LU decomposition is found using an iterative numerical process and can fail for those matrices that cannot be decomposed or decomposed easily. A variation of this decomposition that is numerically more stable to solve in practice is called the LUP decomposition, or the LU decomposition with partial pivoting.

A = L

·U·P (14.3)

The rows of the parent matrix are re-ordered to simplify the decomposition process and the additional P matrix specifies a way to permute the result or return the result to the original order. There are also other variations of the LU. The LU decomposition is often used to simplify the solving of systems of linear equations, such as finding the coefficients in a linear regression, as well as in calculating the determinant and inverse of a matrix.

The LU decomposition can be implemented in Python with the lu() function. More specifically, this function calculates an LPU decomposition. The example below first defines a 3 ×3 square matrix. The LU decomposition is calculated, then the original matrix is reconstructed from the components.

```
# LU decomposition
from numpy import array
from scipy.linalg import lu
# define a square matrix
A = array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
print(A)
# factorize
```

```
P, L, U = lu(A)
print(P)
print(L)
print(U)
# reconstruct
B = P.dot(L).dot(U)
print(B)
```

Listing 14.1: Example of calculating an LU decomposition.

Running the example first prints the defined 3Ucomp onents
× 3 matrix, then the P, L, and
of the decomposition, then finally the original matrix is reconstructed.

```
[[1 2 3]
[4 5 6]
[7 8 9]]

[[ 0. 1. 0.]
[ 0. 0. 1.]
[ 1. 0. 0.]]

[[1. 0. 0. ]
[0.142857141. 0. ]
[0.571428570.5 1. ]]

[[ 7.00000000e+00 8.00000000e+00 9.00000000e+00]
[ 0.00000000e+00 8.57142857e-01 1.71428571e+00]
[ 0.00000000e+00 0.00000000e+00 -1.58603289e-16]]

[[ 1. 2. 3.]
[ 4. 5. 6.]
[ 7. 8. 9.]]
```

Listing 14.2: Sample output from calculating an LU decomposition.

14.4 QRDecomposition

The QR decomposition is for n × m matrices (not limited to square matrices) and decomposes a matrix into Q and R components.

A = Q

Or, without the dot notation.·R (14.4)

A=QR (14.5)

Where A is the matrix that we wish to decompose, Q a matrix with the size m × m, and R is an upper triangle matrix with the size m × n. The QR decomposition is found using an iterative numerical method that can fail for those matrices that cannot be decomposed, or decomposed easily. Like the LU decomposition, the QR decomposition is often used to solve systems of linear equations, although is not limited to square matrices.

The QR decomposition can be implemented in NumPy using the qr() function. By default, the function returns the Q and R matrices with smaller or reduced dimensions that is more

14.5.CholeskyDecomposition 112 economical. We can change this to return the expected

sizes of m

×mforQandm×nfor

R by specifying the mode argument as ‘complete’, although this is not required for most applications. The example below defines a 3 × 2 matrix, calculates the QR decomposition, then reconstructs the original matrix from the decomposed elements.

```
# QR decomposition
from numpy import array
from numpy.linalg import
qr # define rectangular
matrix A = array([
[1, 2],
[3, 4],
[5, 6]])
print(A)
# factorize
Q, R = qr(A, 'complete')
print(Q)
print(R)
# reconstruct
B = Q.dot(R)
print(B)
```

Listing 14.3: Example of calculating an QR decomposition.

Running the example first prints the defined 3Relements, then
× 2 matrix, then the Q and
finally the reconstructed matrix that matches what we started with.

```
[[1 2]
[34]
[5 6]]

[[-0.16903085 0.89708523 0.40824829]
[-0.50709255 0.27602622 -0.81649658]
[-0.84515425 -0.34503278 0.40824829]]

[[-5.91607978 -7.43735744]
[0. 0.82807867]
[0. 0. ]]

[[ 1. 2.]
[ 3. 4.]
[ 5. 6.]]
```

Listing 14.4: Sample output from calculating an QR decomposition.

14.5Cholesky Decomposition

The Cholesky decomposition is for square symmetric matrices where all values are greater than zero, so-called positive definite matrices. For our interests in machine learning, we will focus on the Cholesky decomposition for real-valued matrices and ignore the cases when working with complex numbers. The decomposition is defined as follows:

A = L

·LT (14.6)

Or without the dot notation:

A=LLT (14.7)

Where A is the matrix being decomposed, L is the lower triangular matrix and LT is the transpose of L. The decompose can also be written as the product of the upper triangular matrix, for example:

A = U T

·U (14.8)

Where U is the upper triangular matrix. The Cholesky decomposition is used for solving linear least squares for linear regression, as well as simulation and optimization methods. When decomposing symmetric matrices, the Cholesky decomposition is nearly twice as efficient as the LU decomposition and should be preferred in these cases.

> While symmetric, positive definite matrices are rather special, they occur quite frequently in some applications, so their special factorization, called Cholesky decomposition, is good to know about. When you can use it, Cholesky decomposition is about a factor of two faster than alternative methods for solving linear equations.
>
> — Page 100, Numerical Recipes: The Art of Scientific Computing, Third Edition, 2007.

The Cholesky decomposition can be implemented in NumPy by calling the cholesky() function. The function only returns L as we can easily access the L transpose as needed. The example below defines a 3 ×3 symmetric and positive definite matrix and calculates the Cholesky decomposition, then the original matrix is reconstructed.

```
# Cholesky decomposition
from numpy import array
from numpy.linalg import
cholesky # define symmetrical
matrix
A = array([
[2, 1, 1],
[1, 2, 1],
[1, 1, 2]])
print(A)
# factorize
L = cholesky(A)
print(L)
# reconstruct
B = L.dot(L.T)
print(B)
```

Listing 14.5: Example of calculating an Cholesky decomposition.

Running the example first prints the symmetric matrix, then the lower triangular matrix from the decomposition followed by the reconstructed matrix.

```
[[2 1 1]
[1 2 1]
[1 1 2]]

[[1.414213560. 0.                [          ]
0.70710678 1.22474487 0.         ]
```

```
[ 0.70710678 0.40824829 1.15470054]]

[[ 2. 1. 1.]
 [ 1. 2. 1.]
 [ 1. 1. 2.]]
```

Listing 14.6: Sample output from calculating an Cholesky decomposition.

14.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Write a summary of matrix decomposition to explain the principle to other students.

Create one example using each operation with your own small array data.

Search machine learning papers and find 1 example of each operation being used.
If you explore any of these extensions, I'd love to know.

14.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

14.7.1Bo oks

Section 6.6 Matrix decompositions. No Bullshit Guide To Linear Algebra, 2017.
http://amzn.to/2k76D4

Lecture 7 QR Factorization, Numerical Linear Algebra, 1997.
http://amzn.to/2BI9kRH

Section 2.3 LU Decomposition and Its Applications, Numerical Recipes: The Art of Scientific Computing, Third Edition, 2007.
http://amzn.to/2BezVEE

Section 2.10 QR Decomposition, Numerical Recipes: The Art of Scientific Computing, Third Edition, 2007.
http://amzn.to/2BezVEE

Section 2.9 Cholesky Decomposition, Numerical Recipes: The Art of Scientific Computing, Third Edition, 2007.
http://amzn.to/2BezVEE

Lecture 23, Cholesky Decomposition, Numerical Linear Algebra, 1997.
http://amzn.to/2BI9kRH

14.7.2 API

scipy.linalg.lu() API.
https://docs.scipy.org/doc/scipy- 1.0.0/reference/generated/scipy.linalg.lu.html

numpy.linalg.qr() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.linalg.qr.html

numpy.linalg.cholesky() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.linalg.cholesky.html

14.7.3 Articles

Matrix decomposition on Wikipedia.
https://en.wikipedia.org/wiki/Matrix_decomposition

LU decomposition on Wikipedia.
https://en.wikipedia.org/wiki/LU_decomposition

QR Decomposition on Wikipedia.
https://en.wikipedia.org/wiki/QR_decomposition

Cholesky decomposition on Wikipedia.
https://en.wikipedia.org/wiki/Cholesky_decomposition

14.8 Summary

In this tutorial, you discovered matrix decompositions and how to calculate them in Python. Specifically, you learned:

What a matrix decomposition is and why these types of operations are important.

How to calculate an LU and QR matrix decompositions in Python.

1Ho4w.8 to. 1ca Nlcuelxatte a Cholesky matrix decomposition in Python.

In the next chapter you will discover the eigendecomposition, eigenvalues, and eigenvectors.

Chapter 15

Eigendecomp osition

Matrix decompositions are a useful tool for reducing a matrix to their constituent parts in order to simplify a range of more complex operations. Perhaps the most used type of matrix decomposition is the eigendecomposition that decomposes a matrix into eigenvectors and eigenvalues. This decomposition also plays a role in methods used in machine learning, such as in the Principal Component Analysis method or PCA. In this tutorial, you will discover the eigendecomposition, eigenvectors, and eigenvalues in linear algebra. After completing this tutorial, you will know:

What an eigendecomposition is and the role of eigenvectors and eigenvalues.

How to calculate an eigendecomposition in Python with NumPy.

How to confirm a vector is an eigenvector and how to reconstruct a matrix from eigenvectors and eigenvalues.

Let's get started.

15.1 TutorialOverview

This tutorial is divided into 5 parts; they are:

1. Eigendecomposition of a Matrix
2. Eigenvectors and Eigenvalues
3. Calculation of Eigendecomposition
4. Confirm an Eigenvector and Eigenvalue
5. Reconstruct Matrix

15.2 Eigendecomposition of a Matrix

Eigendecomposition of a matrix is a type of decomposition that involves decomposing a square matrix into a set of eigenvectors and eigenvalues.

> One of the most widely used kinds of matrix decomposition is called eigendecomposition, in which we decompose a matrix into a set of eigenvectors and eigenvalues.
>
> — Page 42, Deep Learning, 2016.

A vector is an eigenvector of a matrix if it satisfies the following equation.

A

·v=λ·v (15.1)

This is called the eigenvalue equation, where A is the parent square matrix that we are decomposing, v is the eigenvector of the matrix, and λ is the lowercase Greek letter lambda and represents the eigenvalue scalar. Or without the dot notation.

Av=λv (15.2)

A matrix could have one eigenvector and eigenvalue for each dimension of the parent matrix. Not all square matrices can be decomposed into eigenvectors and eigenvalues, and some can only be decomposed in a way that requires complex numbers. The parent matrix can be shown to be a product of the eigenvectors and eigenvalues.

A = Q

·Λ·QT (15.3)

Or, without the dot notation.

A=QΛQT (15.4)

Where Q is a matrix comprised of the eigenvectors, Λ is the uppercase Greek letter lambda and is the diagonal matrix comprised of the eigenvalues, and QT is the transpose of the matrix comprised of the eigenvectors.

> However, we often want to decompose matrices into their eigenvalues and eigenvectors. Doing so can help us to analyze certain properties of the matrix, much as decomposing an integer into its prime factors can help us understand the behavior of that integer.
>
> — Page 43, Deep Learning, 2016.

Eigen is not a name, e.g. the method is not named after “Eigen”; eigen (pronounced eye-gan) is a German word that means own or innate, as in belonging to the parent matrix. A decomposition operation does not result in a compression of the matrix; instead, it breaks it down into constituent parts to make certain operations on the matrix easier to perform. Like other matrix decomposition methods, Eigendecomposition is used as an element to simplify the calculation of other more complex matrix operations.

> Almost all vectors change direction, when they are multiplied by A. Certain exceptional vectors x are in the same direction as Ax. Those are the “eigenvectors”. Multiply an eigenvector by A, and the vector Ax is the number λ times the original x. [...] The eigenvalue λ tells whether the special vector x is stretched or shrunk or reversed or left unchanged — when it is multiplied by A.
>
> — Page 289, Introduction to Linear Algebra, Fifth Edition, 2016.

Eigendecomposition can also be used to calculate the principal components of a matrix in the Principal Component Analysis method or PCA that can be used to reduce the dimensionality of data in machine learning.

15.3 Eigenvectors and Eigenvalues

Eigenvectors are unit vectors, which means that their length or magnitude is equal to 1.0. They are often referred as right vectors, which simply means a column vector (as opposed to a row vector or a left vector). A right-vector is a vector as we understand them. Eigenvalues are coefficients applied to eigenvectors that give the vectors their length or magnitude. For example, a negative eigenvalue may reverse the direction of the eigenvector as part of scaling it. A matrix that has only positive eigenvalues is referred to as a positive definite matrix, whereas if the eigenvalues are all negative, it is referred to as a negative definite matrix.

> Decomposing a matrix in terms of its eigenvalues and its eigenvectors gives valuable insights into the properties of the matrix. Certain matrix calculations, like computing the power of the matrix, become much easier when we use the eigendecomposition of the matrix.
>
> — Page 262, No Bullshit Guide To Linear Algebra, 2017.

15.4Calculation of Eigendecomposition

An eigendecomposition is calculated on a square matrix using an efficient iterative algorithm, of which we will not go into the details. Often an eigenvalue is found first, then an eigenvector is found to solve the equation as a set of coefficients. The eigendecomposition can be calculated in NumPy using the eig() function. The example below first defines a 3 × 3 square matrix. The eigendecomposition is calculated on the matrix returning the eigenvalues and eigenvectors.

```
# eigendecomposition
from numpy import array
from numpy.linalg import
eig # define matrix
A = array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
print(A)
# factorize
values, vectors = eig(A)
```

```
print(values)
print(vectors)
```

Listing 15.1: Example of calculating an eigendecomposition.

Running the example first prints the defined matrix, followed by the eigenvalues and the eigenvectors. More specifically, the eigenvectors are the right-hand side eigenvectors and are normalized to unit length.

```
[[1 2 3]
[4 5 6]
[7 8 9]]

[ 1.61168440e+01 -1.11684397e+00 -9.75918483e-16]

[[-0.23197069 -0.78583024 0.40824829]
[-0.52532209 -0.08675134 -0.81649658]
[-0.8186735 0.61232756 0.40824829]]
```

Listing 15.2: Sample output from calculating an eigendecomposition.

15.5Confirm an Eigenvector and Eigenvalue

We can confirm that a vector is indeed an eigenvector of a matrix. We do this by multiplying the candidate eigenvector by the value vector and comparing the result with the eigenvalue. First, we will define a matrix, then calculate the eigenvalues and eigenvectors. We will then test whether the first vector and value are in fact an eigenvalue and eigenvector for the matrix. We know they are, but it is a good exercise.

The eigenvectors are returned as a matrix with the same dimensions as the parent matrix, where each column is an eigenvector, e.g. the first eigenvector is vectors[:, 0]. Eigenvalues are returned as a list, where value indices in the returned array are paired with eigenvectors by column index, e.g. the first eigenvalue at values[0] is paired with the first eigenvector at vectors[:, 0].

```
# confirm eigenvector
from numpy import array
from numpy.linalg import eig
# define matrix
A = array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
# factorize
values, vectors = eig(A)
# confirm first eigenvector
B = A.dot(vectors[:, 0]) print(B)
C = vectors[:, 0] * values[0]
print(C)
```

Listing 15.3: Example of calculating a confirmation of an eigendecomposition.

The example multiplies the original matrix with the first eigenvector and compares it to the first eigenvector multiplied by the first eigenvalue. Running the example prints the results of these two multiplications that show the same resulting vector, as we would expect.

```
[ -3.73863537  -8.46653421  -13.19443305]

[ -3.73863537 -8.46653421 -13.19443305]
```

Listing 15.4: Sample output from calculating a confirmation of an eigendecomposition.

15.6Reconstruct Matrix

We can reverse the process and reconstruct the original matrix given only the eigenvectors and eigenvalues. First, the list of eigenvectors must be taken together as a matrix, where each vector becomes a row. The eigenvalues need to be arranged into a diagonal matrix. The NumPy diag() function can be used for this. Next, we need to calculate the inverse of the eigenvector matrix, which we can achieve with the inv() NumPy function. Finally, these elements need to be multiplied together with the dot() function.

```
# reconstruct matrix
from numpy import diag
from numpy.linalg import inv
from numpy import array
from numpy.linalg import eig
# define matrix
A = array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
print(A)
# factorize
values, vectors = eig(A)
# create matrix from eigenvectors
Q = vectors
# create inverse of eigenvectors matrix
R = inv(Q)
# create diagonal matrix from
eigenvalues L = diag(values)
# reconstruct the original matrix
B = Q.dot(L).dot(R)
print(B)
```

Listing 15.5: Example of reconstructing a matrix from an eigendecomposition.

The example calculates the eigenvalues and eigenvectors again and uses them to reconstruct the original matrix. Running the example first prints the original matrix, then the matrix reconstructed from eigenvalues and eigenvectors matching the original matrix.

```
[[1 2 3]
[4 5 6]
[7 8 9]]

[[ 1. 2. 3.] [ 4. 5.
6.]
```

```
[ 7. 8. 9.]]
```

Listing 15.6: Sample output from reconstructing a matrix from an eigendecomposition.

15.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Develop an eigendecomposition and reconstruction of your own small contrived array data.

List ten high-level operations that make use of the eigendecomposition.

Implement the eigendecomposition operation from scratch for matrices defined as lists of lists.

I

1f y5o.u8 exploFreu arntyh oef trh eRsee eaxtdenisniogns, I'd love to know.

This section provides more resources on the topic if you are looking to go deeper.

15.8.1Bo oks

Section 6.1 Eigenvalues and eigenvectors. No Bullshit Guide To Linear Algebra, 2017.
http://amzn.to/2k76D4

Chapter 6 Eigenvalues and Eigenvectors, Introduction to Linear Algebra, Fifth Edition, 2016.
http://amzn.to/2AZ7R8j

Section 2.7 Eigendecomposition, Deep Learning, 2016.
http://amzn.to/2B3MsuU

Chapter 5 Eigenvalues, Eigenvectors, and Invariant Linear Algebra Done Right, Subspaces, Third Edition, 2015.
http://amzn.to/2BGuEqI

Lecture 24, Eigenvalue Problems, Numerical Linear Algebra, 1997.
http://amzn.to/2BI9kRH

15.8.2 API

numpy.linalg.eig() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.linalg.eig.html

numpy.diag() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.diag.html

numpy.dot() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.dot.html

numpy.linalg.inv() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.linalg.inv.html

15.8.3 Articles

eigen on Wiktionary.
https://en.wiktionary.org/wiki/eigen

Eigenvalues and eigenvectors.
https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors

Eigendecomposition of a matrix.
https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix

Eigenvalue algorithm.
https://en.wikipedia.org/wiki/Eigenvalue_algorithm

Matrix decomposition.
https://en.wikipedia.org/wiki/Matrix_decomposition

15.9 Summary

In this tutorial, you discovered the eigendecomposition, eigenvectors, and eigenvalues in linear algebra. Specifically, you learned:

What an eigendecomposition is and the role of eigenvectors and eigenvalues.

How to calculate an eigendecomposition in Python with NumPy.

How to confirm a vector is an eigenvector and how to reconstruct a matrix from eigenvectors

1an5d. e9ig.1en Nvaeluxets.

In the next chapter you will discover the singular-value decomposition method.

Chapter 16

Singular Value Decomposition

Matrix decomposition, also known as matrix factorization, involves describing a given matrix using its constituent elements. Perhaps the most known and widely used matrix decomposition method is the Singular-Value Decomposition, or SVD. All matrices have an SVD, which makes it more stable than other methods, such as the eigendecomposition. As such, it is often used in a wide array of applications including compressing, denoising, and data reduction. In this tutorial, you will discover the Singular-Value Decomposition method for decomposing a matrix into its constituent elements. After completing this tutorial, you will know:

What Singular-value decomposition is and what is involved.

How to calculate an SVD and reconstruct a rectangular and square matrix from SVD elements.

How to calculate the pseudoinverse and perform dimensionality reduction using the SVD.

Let's get started.

16.1 TutorialOverview

This tutorial is divided into 5 parts; they are:

1. What is the Singular-Value Decomposition
2. Calculate Singular-Value Decomposition
3. Reconstruct Matrix
4. Pseudoinverse
5. Dimensionality Reduction

16.2 WhatistheSingular-ValueDecomposition

The Singular-Value Decomposition, or SVD for short, is a matrix decomposition method for reducing a matrix to its constituent parts in order to make certain subsequent matrix calculations simpler. For the case of simplicity we will focus on the SVD for real-valued matrices and ignore the case for complex numbers.

A = U

·Σ·VT (16.1)

Where A is the real n

× m matrix that we wish to decompose, U is an m × m matrix, Σ represented by the uppercase Greek letter sigma) is an m

× n diagonal matrix, and V T is the V transpose of an n
× n matrix where T is a superscript.

The Singular Value Decomposition is a highlight of linear algebra.

— Page 371, Introduction to Linear Algebra, 2016.

The diagonal values in the Σ matrix are known as the singular values of the original matrix A. The columns of the U matrix are called the left-singular vectors of A, and the columns of V are called the right-singular vectors of A. The SVD is calculated via iterative numerical methods. We will not go into the details of these methods. Every rectangular matrix has a singular value decomposition, although the resulting matrices may contain complex numbers and the limitations of floating point arithmetic may cause some matrices to fail to decompose neatly.

The singular value decomposition (SVD) provides another way to factorize a matrix,

into singular vectors and singular values. The SVD allows us to discover some of the same kind of information as the eigendecomposition. However, the SVD is more generally applicable.

— Pages 44-45, Deep Learning, 2016.

The SVD is used widely both in the calculation of other matrix operations, such as matrix inverse, but also as a data reduction method in machine learning. SVD can also be used in least squares linear regression, image compression, and denoising data.

The singular value decomposition (SVD) has numerous applications in statistics,

machine learning, and computer science. Applying the SVD to a matrix is like looking inside it with X-ray vision...

— Page 297, No Bullshit Guide To Linear Algebra, 2017.

16.3Calculate Singular-Value Decomposition

The SVD can be calculated by calling the svd() function. The function takes a matrix and returns the U, Σ and V T elements. The Σ diagonal matrix is returned as a vector of singular values. The V matrix is returned in a transposed form, e.g. V T. The example below defines a 3 × 2 matrix and calculates the singular-value decomposition.

```
# singular-value
decomposition from numpy
import array
from scipy.linalg import svd
# define a matrix
A = array([
[1, 2],
[3, 4],
[5, 6]])
print(A)
# factorize
U, s, V = svd(A)
print(U)
print(s)
print(V)
```

Listing 16.1: Example of calculating a singular-value decomposition.

Running the example first prints the defined 3 ×2 matrix, then the 3×3 U matrix, 2 element Σ vector, and 2 ×2 V T matrix elements calculated from the decomposition.

```
[[1 2]
[3 4]
[5 6]]

[[-0.2298477 0.88346102 0.40824829]
[-0.52474482 0.24078249 -0.81649658]
[-0.81964194 -0.40189603 0.40824829]]

[ 9.52551809 0.51430058]

[[-0.61962948 -0.78489445]
[-0.78489445 0.61962948]]
```

Listing 16.2: Sample output from calculating a singular-value decomposition.

16.4Reconstruct Matrix

The original matrix can be reconstructed from the U, Σ, and V T elements. The U, s, and V elements returned from the svd() cannot be multiplied directly. The s vector must be converted into a diagonal matrix using the diag() function. By default, this function will create a square matrix that is m × m, relative to our original matrix. This causes a problem as the size of the matrices do not fit the rules of matrix multiplication, where the number of columns in a matrix must match the number of rows in the subsequent matrix. After creating the square Σ diagonal matrix, the sizes of the matrices are relative to the original n × m matrix that we are decomposing, as follows:

U(m

Where, in fact, we require:

×m)·Σ(m×m)·VT(n×n) (16.2)

U(m

× m) ·Σ(m × n) · V T (n × n) (16.3)

We can achieve this by creating a new Σ matrix of all zero values that is m × n (e.g. more rows) and populate the first n × n part of the matrix with the square diagonal matrix calculated

```
# reconstruct rectangular matrix from svd
vfrioam d niaugm()p.y import array
from numpy import diag
from numpy import zeros
from scipy.linalg import svd
# define matrix
A = array([
[1, 2],
[3, 4],
[5, 6]])
print(A)
# factorize
U, s, V = svd(A)
# create m x n Sigma matrix
Sigma = zeros((A.shape[0], A.shape[1]))
# populate Sigma with n x n diagonal
matrix Sigma[:A.shape[1], :A.shape[1]] =
diag(s)
# reconstruct matrix
B = U.dot(Sigma.dot(V))
print(B)
```

Listing 16.3: Example of reconstructing a rectangular matrix from a SVD.

Running the example first prints the original matrix, then the matrix reconstructed from the SVD elements.

```
[[1 2]
[34]
[5 6]]

[[ 1.  2.]
[ 3.  4.]
[ 5. 6.]]
```

Listing 16.4: Sample output from reconstructing a rectangular matrix from a SVD.

The above complication with the Σ diagonal only exists with the case where m and n are not equal. The diagonal matrix can be used directly when reconstructing a square matrix, as follows.

```
# reconstruct square matrix from svd
from numpy import array
from numpy import diag
from scipy.linalg import svd
# define matrix
A = array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
print(A)
# factorize
U, s, V = svd(A)
# create n x n Sigma matrix
```

```
Sigma = diag(s)
# reconstruct matrix
B =
U.dot(Sigma.dot(V))
print(B)
```

Listing 16.5: Example of reconstructing a square matrix from a SVD.

Running the example prints the original 3 × 3 matrix and the version reconstructed directly from the SVD elements.

```
[[1 2 3]
[4 5 6]
[7 8 9]]

[[ 1. 2. 3.]
[ 4. 5. 6.]
[ 7. 8. 9.]]
```

Listing 16.6: Sample output from reconstructing a square matrix from a SVD.

16.5 Pseudoinverse

The pseudoinverse is the generalization of the matrix inverse for square matrices to rectangular matrices where the number of rows and columns are not equal. It is also called the Moore- Penrose Inverse after two independent discoverers of the method or the Generalized Inverse.

> Matrix inversion is not defined for matrices that are not square. [...] When A has more columns than rows, then solving a linear equation using the pseudoinverse provides one of the many possible solutions.
>
> — Page 46, Deep Learning, 2016.

The pseudoinverse is denoted as A+, where A is the matrix that is being inverted and + is a superscript. The pseudoinverse is calculated using the singular value decomposition of A:

A+=V ·D+·UT (16.4)

Or, without the dot notation:

A+=V ·D+·UT (16.5)

Where A+ is the pseudoinverse, D+ is the pseudoinverse of the diagonal matrix Σ and V T is the transpose of V T. We can get U and V from the SVD operation.

A = U ·Σ·VT (16.6)

The D+ can be calculated by creating a diagonal matrix from Σ, calculating the reciprocal of each non-zero element in Σ, and taking the transpose if the original matrix was rectangular.

s1,1 0 0

Σ=0s2,20 (16.7)

0 0 s3,3

1
0 0 —
s1,1
D+=010 —
s(16.8)2,2
0 0 1 —
s3,3

The pseudoinverse provides one way of solving the l inear regression equation, specifically when there are more rows than there are columns, which is often the case. NumPy provides the function pinv() for calculating the pseudoinverse of a rectangular matrix. The example below

xde 2fi mneast raix 4 and calculates the pseudoinverse.

```
# pseudoinverse
from numpy import array
from numpy.linalg import
pinv # define matrix
A = array([
[0.1, 0.2],
[0.3, 0.4],
[0.5, 0.6],
[0.7, 0.8]])
print(A)
# calculate pseudoinverse
B = pinv(A)
print(B)
```

Listing 16.7: Example of calculating the pseudoinverse.

Running the example first prints the defined matrix, and then the calculated pseudoinverse.

```
[[ 0.1 0.2]
[ 0.3 0.4]
[ 0.5 0.6]
[ 0.7 0.8]]

[[ -1.00000000e+01 -5.00000000e+00 9.04289323e-15 5.00000000e+00]
[ 8.50000000e+00 4.50000000e+00 5.00000000e-01 -3.50000000e+00]]
```

Listing 16.8: Sample output from calculating the pseudoinverse.

We can calculate the pseudoinverse manually via the SVD and compare the results to the pinv() function. First we must calculate the SVD. Next we must calculate the reciprocal of each value in the s array. Then the s array can be transformed into a diagonal matrix with an added row of zeros to make it rectangular. Finally, we can calculate the pseudoinverse from the elements. The specific implementation is:

A+=VT

·DT·UT (16.9)

The full example is listed below.

```
# pseudoinverse via svd
from numpy import array
from numpy.linalg import
svd from numpy import
zeros
from numpy import diag
# define matrix
A = array([
[0.1, 0.2],
```

```
[0.3, 0.4],
[0.5, 0.6],
[0.7, 0.8]])
print(A)
# factorize
U, s, V = svd(A)
# reciprocals of s
d=1.0/s
# create m x n D matrix
D = zeros(A.shape)
# populate D with n x n diagonal
matrix D[:A.shape[1], :A.shape[1]] =
diag(d)
# calculate pseudoinverse
B = V.T.dot(D.T).dot(U.T)
print(B)
```

Listing 16.9: Example of calculating the pseudoinverse via the SVD.

Running the example first prints the defined rectangular matrix and the pseudoinverse that matches the above results from the pinv() function.

```
[[ 0.1 0.2]
[ 0.3 0.4]
[ 0.5 0.6]
[ 0.7 0.8]]

[[ -1.00000000e+01 -5.00000000e+00 9.04831765e-15 5.00000000e+00]
[ 8.50000000e+00 4.50000000e+00 5.00000000e-01 -3.50000000e+00]]
```

Listing 16.10: Sample output from calculating the pseudoinverse via the SVD.

16.6 Dimensionality Reduction

A popular application of SVD is for dimensionality reduction. Data with a large number of features, such as more features (columns) than observations (rows) may be reduced to a smaller subset of features that are most relevant to the prediction problem. The result is a matrix with a lower rank that is said to approximate the original matrix. To do this we can perform an SVD operation on the original data and select the top k largest singular values in Σ. These columns can be selected from Σ and the rows selected from V T. An approximate B of the original vector A can then be reconstructed.

B = U

·Σ·TkVk (16.10)

In natural language processing, this approach can be used on matrices of word occurrences or word frequencies in documents and is called Latent Semantic Analysis or Latent Semantic Indexing. In practice, we can retain and work with a descriptive subset of the data called T. This is a dense summary of the matrix or a projection.

T = U

·Σk (16.11)

Further, this transform can be calculated and applied to the original matrix A as well as other similar matrices.

T = A

·VTk (16.12)

The example below demonstrates data reduction with the SVD. First a 3

× 10 matrix is

defined, with more columns than rows. The SVD is calculated and only the first two features are selected. The elements are recombined to give an accurate reproduction of the original matrix. Finally the tranform is calculated two different ways.

```
# data reduction with svd
from numpy import array
from numpy import diag
from numpy import zeros
from scipy.linalg import svd
# define matrix
A = array([
[1,2,3,4,5,6,7,8,9,10],
[11,12,13,14,15,16,17,18,19,20],
[21,22,23,24,25,26,27,28,29,30]])
print(A)
# factorize
U, s, V = svd(A)
# create m x n Sigma matrix
Sigma = zeros((A.shape[0], A.shape[1]))
# populate Sigma with n x n diagonal matrix
Sigma[:A.shape[0], :A.shape[0]] = diag(s)
# select
n_elements = 2
Sigma = Sigma[:, :n_elements]
V = V[:n_elements, :]
# reconstruct
B = U.dot(Sigma.dot(V))
print(B)
# transform
T = U.dot(Sigma)
print(T)
T = A.dot(V.T)
print(T)
```

Listing 16.11: Example of calculating data reduction with the SVD.

Running the example first prints the defined matrix then the reconstructed approximation, followed by two equivalent transforms of the original matrix.

```
[[12345678910]
[11 12 13 14 15 16 17 18 19 20]
[21 22 23 24 25 26 27 28 29 30]]

[[ 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.]
[ 11. 12. 13. 14. 15. 16. 17. 18. 19. 20.]
[ 21. 22. 23. 24. 25. 26. 27. 28. 29. 30.]]

[[-18.52157747 6.47697214]
[-49.81310011 1.91182038]
[-81.10462276 -2.65333138]]
```

```
[[-18.52157747 6.47697214]
[-49.81310011 1.91182038]
[-81.10462276 -2.65333138]]
```

Listing 16.12: Sample output from calculating data reduction with the SVD.

The scikit-learn provides a TruncatedSVD class that implements this capability directly. The TruncatedSVD class can be created in which you must specify the number of desirable features or components to select, e.g. 2. Once created, you can fit the transform (e.g. calculate V^T_k) by calling the fit() function, then apply it to the original matrix by calling the transform() function. The result is the transform of A called T above. The example below demonstrates the TruncatedSVD class.

```
# svd data reduction in scikit-learn
from numpy import array
from sklearn.decomposition import
TruncatedSVD # define matrix
A = array([
[1,2,3,4,5,6,7,8,9,10],
[11,12,13,14,15,16,17,18,19,20],
[21,22,23,24,25,26,27,28,29,30]])
print(A)
# create transform
svd = TruncatedSVD(n_components=2)
# fit transform
svd.fit(A)
# apply transform
result = svd.transform(A)
print(result)
```

Listing 16.13: Example of calculating data reduction with the SVD in scikit-learn.

Running the example first prints the defined matrix, followed by the transformed version of the matrix. We can see that the values match those calculated manually above, except for the sign on some values. We can expect there to be some instability when it comes to the sign given the nature of the calculations involved and the differences in the underlying libraries and methods used. This instability of sign should not be a problem in practice as long as the transform is trained for reuse.

```
[[12345678910]
[11 12 13 14 15 16 17 18 19 20]
[21 22 23 24 25 26 27 28 29 30]]

[[ 18.52157747 6.47697214]
[ 49.81310011 1.91182038]
[ 81.10462276 -2.65333138]]
```

Listing 16.14: Sample output from calculating data reduction with the SVD in scikit-learn.

16.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Experiment with the SVD method on your own data.

Research and list 10 applications of SVD in machine learning.

Apply SVD as a data reduction technique on a real-world tabular dataset.

If you explore any of these extensions, I'd love to know.

16.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

16.8.1Bo oks

Chapter 12, Singular-Value and Jordan Decompositions, Linear Algebra and Matrix Analysis for Statistics, 2014.
http://amzn.to/2A9ceNv

Chapter 4, The Singular Value Decomposition and Chapter 5, More on the SVD, Numerical Linear Algebra, 1997.
http://amzn.to/2kjEF4S

Section 2.4 The Singular Value Decomposition, Matrix Computations, 2012.
http://amzn.to/2B9xnLD

Chapter 7 The Singular Value Decomposition (SVD), Introduction to Linear Algebra, Fifth Edition, 2016.
http://amzn.to/2AZ7R8j

Section 2.8 Singular Value Decomposition, Deep Learning, 2016.
http://amzn.to/2B3MsuU

Section 7.D Polar Decomposition and Singular Value Decomposition, Linear Algebra Done R, Tighhirtd Edition, 2015.
http://amzn.to/2BGuEqI

Lecture 3 The Singular Value Decomposition, Numerical Linear Algebra, 1997.
http://amzn.to/2BI9kRH

Section 2.6 Singular Value Decomposition, , Numerical Recipes: The Art of Scientific Computing, Third Edition, 2007.
http://amzn.to/2BezVEE

Section 2.9 The Moore-Penrose Pseudoinverse, Deep Learning, 2016.
http://amzn.to/2B3MsuU

16.8.2 API

numpy.linalg.svd() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.linalg.svd.html

numpy.matrix.H API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.matrix.H.html

numpy.diag() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.diag.html

numpy.linalg.pinv() API.

https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.linalg.pinv.html

sklearn.decomposition.TruncatedSVD API.
http://scikit- learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD
html

16.8.3 Articles

Matrix decomposition on Wikipedia.
https://en.wikipedia.org/wiki/Matrix_decomposition

Singular-value decomposition on Wikipedia.
https://en.wikipedia.org/wiki/Singular- value_decomposition

Singular value on Wikipedia.
https://en.wikipedia.org/wiki/Singular_value

Moore-Penrose inverse on Wikipedia.
https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse

Latent semantic analysis on Wikipedia.
https://en.wikipedia.org/wiki/Latent_semantic_analysis

16.9 Summary

In this tutorial, you discovered the Singular-value decomposition method for decomposing a matrix into its constituent elements. Specifically, you learned:

What Singular-value decomposition is and what is involved.

How to calculate an SVD and reconstruct a rectangular and square matrix from SVD elements.

How to calculate the pseudoinverse and perform dimensionality reduction using the SVD.

16.9.1 Next

This is the end of the part on matrix factorization. In the next part you will discover the intersection of statistics and linear algebra, starting with simple statistical calculations on vectors and matrices.

Part VI

Statistics

Chapter 17

Introduction to Multivariate Statistics

Fundamental statistics are useful tools in applied machine learning for a better understanding your data. They are also the tools that provide the foundation for more advanced linear algebra operations and machine learning methods, such as the covariance matrix and principal component analysis respectively. As such, it is important to have a strong grip on fundamental statistics in the context of linear algebra notation. In this tutorial, you will discover how fundamental statistical operations work and how to implement them using NumPy with notation and terminology from linear algebra.
After completing this tutorial, you will know:

What the expected value, average, and mean are and how to calculate them.

What the variance and standard deviation are and how to calculate them.

What the covariance, correlation, and covariance matrix are and how to calculate them.

1e7t’s. g1et stTarutetdo.rialOverview

This tutorial is divided into 4 parts; they are:

1. Expected Value and Mean

2. Variance and Standard Deviation

3. Covariance and Correlation

4. Covariance Matrix

17.2 Expected Value and Mean

In probability, the average value of some random variable X is called the expected value or the expectation. The expected value uses the notation E with square brackets around the name of the variable; for example:

E[X] (17.1)

137 It is calculated as the proba

∑bilityweightedsumofvaluesthatcanbedrawn.E[X]=x1×p1,x2×p2,x3×p3,⋯,xn×pn(17.2)Insimplecases,suchas just as likely. Therefore, the expected value can be calculated as the sum of all values multiplied by the reciprocal of the number of values.

1

E[X] =∑ ×x1,x2,x3,⋯,xn (17.3)n

In statistics, the mean, or more technically the arithmetic mean or sample mean, can be estimated from a sample of examples drawn from the domain. It is confusing because mean, average, and expected value are used inter

∑changeably. In the abstract, the mean is denoted by the lower case Greek letter mu μ and is calculated from the sample of observations, rather than all possible values.

1

∑

×μx=1,x2,x3,⋯,xn (17.4)n

Or, written more compactly:

μ = P(x)

∑×x (17.5)

Where x is the vector of observations and P(x) is the calculated probability for each value.

When calculated for a specific variable, such as x, the mean is denoted as a lower case variable name with a line above, called x-bar e.g. x̄.

n

1

x̄ = ×xi (17.6)n

i=1

The arithmetic mean can be calculated for a vector or matrix in NumPy by using the mean() function. The example below defines a 6-element vector and calculates the mean.

```
# vector mean
from numpy import
array from numpy
import mean
# define vector
v = array([1,2,3,4,5,6])
print(v)
# calculate mean
result = mean(v)
print(result)
```

Listing 17.1: Example of calculating a vector mean.

Running the example first prints the defined vector and the mean of the values in the vector.

```
[123456] 3.5
```

Listing 17.2: Sample output from calculating a vector mean.

The mean function can calculate the row or column means of a matrix by specifying the axis argument and the value 0 or 1 respectively. The example below defines a 2 × 6 matrix and calculates both column and row means.

```
# matrix means
from numpy import array
from numpy import mean
# define matrix
M = array([
[1,2,3,4,5,6],
[1,2,3,4,5,6]])
print(M)
# column means
col_mean = mean(M,
axis=0) print(col_mean)
# row means
row_mean = mean(M,
axis=1) print(row_mean)
```

Listing 17.3: Example of calculating matrix means.

Running the example first prints the defined matrix, then the calculated column and row mean values.

```
[[123456]
[123456]]

[ 1. 2. 3. 4. 5.  6.]

[ 3.5 3.5]
```

Listing 17.4: Sample output from calculating matrix means.

17.3 VarianceandStandardDeviation

In probability, the variance of some random variable X is a measure of how much values in the distribution vary on average with respect to the mean. The variance is denoted as the function V ar() on the variable.

Var[X] (17.7)

Variance is calculated as the average squared difference of each value in the distribution from the expected value. Or the expected squared difference from the expected value.

V ar[X] = E[(X

−E[X])2] (17.8)

Assuming the expected value of the variable has been calculated (E[X]), the variance of the random variable can be calculated as the sum of the squared difference of each example from the expected value multiplied by the probability of that value.

V ar[X]=∑p(x1)×(x1−E [X])2, p(x2)×(x2−E [X])2 ,⋯, p(xn)×(xn−E [X])2(17.9)

If the probability of each example in the distribution is equal, variance calculation can drop the individual probabilities and multiply the sum of squared differences by the reciprocal of the number of examples in the dis

∑tribution.1Var[X]=×(x−E[X])2,(x−EX2,⋯,x−EX212[])(n[])

(17.10)nInstatistics,thevariancecanbeestimatedfrom domain. In the abstract, the sample variance is denote by the lower case sigma with a 2
superscript indicating the units are squared (e.g. σ2), not that you must square the final value. The sum of the squared differences is multiplied by the reciprocal of the number of examples minus 1 to correct for a bias (bias is related to a deeper discussion on degrees of freedom and I refer you to references at the end of the lesson).

1

∑nσ2=×(x−μ2i)(17.11)n−1i=1

In NumPy, the variance can be calculated for a vector or a matrix using the var() function. By default, the var() function calculates the population variance. To calculate the sample variance, you must set the ddof argument to the value 1. The example below defines a 6-element vector and calculates the sample variance.

```
# vector variance
from numpy import
array from numpy
import var
# define vector
v = array([1,2,3,4,5,6])
print(v)
# calculate variance
result = var(v, ddof=1)
print(result)
```

Listing 17.5: Example of calculating a vector variance.

Running the example first prints the defined vector and then the calculated sample variance of the values in the vector.

```
[123456] 3.5
```

Listing 17.6: Example of calculating a vector variance.

The var function can calculate the row or column variances of a matrix by specifying the axis argument and the value 0 or 1 respectively, the same as the mean function above. The example below defines a 2
× 6 matrix and calculates both column and row sample variances.

```
# matrix variances
from numpy import
array from numpy
import var
# define matrix
M = array([
[1,2,3,4,5,6],
[1,2,3,4,5,6]])
print(M)
```

```
# column variances
col_var = var(M, ddof=1, axis=0)
print(col_var)
# row variances
row_var = var(M, ddof=1, axis=1)
print(row_var)
```

Listing 17.7: Example of calculating matrix variances.

Running the example first prints the defined matrix and then the column and row sample variance values.

```
[[123456]
[123456]]

[ 0. 0. 0. 0. 0.          0.]

[ 3.5 3.5]
```

Listing 17.8: Sample output from calculating matrix variances.

The standard deviation is calculated as the square root of the variance and is denoted as lowercase s.

$$s=\sqrt{\sigma^2} \quad (17.12)$$

To keep with this notation, sometimes the variance is indicated as s2, with 2 as a superscript, again showing that the units are squared. NumPy also provides a function for calculating the standard deviation directly via the std() function. As with the var() function, the ddof argument must be set to 1 to calculate the unbiased sample standard deviation and column and row standard deviations can be calculated by setting the axis argument to 0 and 1 respectively. The example below demonstrates how to calculate the sample standard deviation for the rows and columns of a matrix.

```
# matrix standard deviation
from numpy import array
from numpy import std
# define matrix
M = array([
[1,2,3,4,5,6],
[1,2,3,4,5,6]])
print(M)
# column standard deviations
col_std = std(M, ddof=1, axis=0)
print(col_std)
# row standard deviations
row_std = std(M, ddof=1, axis=1)
print(row_std)
```

Listing 17.9: Example of calculating matrix standard deviations.

Running the example first prints the defined matrix and then the column and row sample standard deviation values.

```
[[123456]
[123456]]
```

```
[ 0. 0. 0. 0. 0. 0.]
[ 1.87082869 1.87082869]
```

Listing 17.10: Sample output from calculating matrix standard deviations.

17.4Covariance and Correlation

In probability, covariance is the measure of the joint probability for two random variables. It describes how the two variables change together. It is denoted as the function cov(X, Y), where X and Y are the two random variables being considered.

cov(X,Y) (17.13)

Covariance is calculated as expected value or average of the product of the differences of each random variable from their expected values, where E[X] is the expected value for X and E[Y] is the expected value of y.

cov(X, Y) = E[(X

−E[X]×(Y−E[Y])] (17.14)

Assuming the expected values for X and Y have been calculated, the covariance can be calculated as the sum of the difference of x values from their expected value multiplied by the difference of the y values from their expected values multiplied by the reciprocal of the number of examples in the population.

1

∑cov(X,Y)=×(x−E[X])×(y−E[Y])

(17.15)nInstatistics,thesamplecovariancecanbecalculatedinthesameway,althoughwithabias correction, the same as with the variance.

1

∑cov(X,Y)=×(x−E[X])×(y−E[Y])(17.16)n−1

The sign of the covariance can be interpreted as whether the two variables increase together (positive) or decrease together (negative). The magnitude of the covariance is not easily interpreted. A covariance value of zero indicates that both variables are completely independent. NumPy does not have a function to calculate the covariance between two variables directly. Instead, it has a function for calculating a covariance matrix called cov() that we can use to retrieve the covariance. By default, the cov()function will calculate the unbiased or sample covariance between the provided random variables.

The example below defines two vectors of equal length with one increasing and one decreasing. We would expect the covariance between these variables to be negative. We access just the covariance for the two variables as the [0, 1] element of the square covariance matrix returned.

```
# vector covariance
from numpy import array
from numpy import cov
# define first vector
x = array([1,2,3,4,5,6,7,8,9])
```

```
print(x)
# define second covariance
y = array([9,8,7,6,5,4,3,2,1])
print(y)
# calculate covariance
Sigma = cov(x,y)[0,1]
print(Sigma)
```

Listing 17.11: Example of calculating a vector covariance.

Running the example first prints the two vectors followed by the covariance for the values in the two vectors. The value is negative, as we expected.

```
[123456789]
[987654321]

-7.5
```

Listing 17.12: Sample output from calculating vector covariance.

The covariance can be normalized to a score between -1 and 1 to make the magnitude interpretable by dividing it by the standard deviation of X and Y . The result is called the correlation of the variables, also called the Pearson correlation coefficient, named for the developer of the method.

$$r = \frac{cov(X, Y)}{sX \times sY} \quad (17.17)$$

Where r is the correlation coefficient of X and Y , cov(X, Y) is the sample covariance of X and Y and sX and sY are the standard deviations of X and Y respectively. NumPy provides the corrcoef() function for calculating the correlation between two variables directly. Like cov(), it returns a matrix, in this case a correlation matrix. As with the results from cov() we can access just the correlation of interest from the [0,1] value from the returned squared matrix.

```
# vector correlation
from numpy import array
from numpy import corrcoef
# define first vector
x = array([1,2,3,4,5,6,7,8,9])
print(x)
# define second vector
y = array([9,8,7,6,5,4,3,2,1])
print(y)
# calculate correlation
corr = corrcoef(x,y)[0,1]
print(corr)
```

Listing 17.13: Example of calculating a vector correlation.

Running the example first prints the two defined vectors followed by the correlation coefficient. We can see that the vectors are maximally negatively correlated as we designed.

```
[123456789]
[987654321]

-1.0
```

Listing 17.14: Sample output from calculating vector correlation.

17.5Covariance Matrix

The covariance matrix is a square and symmetric matrix that describes the covariance between two or more random variables. The diagonal of the covariance matrix are the variances of each of the random variables, as such it is often called the variance-covariance matrix. A covariance matrix is a generalization of the covariance of two variables and captures the way in which all variables in the dataset may change together. The covariance matrix is denoted as the uppercase Greek letter Sigma, e.g. Σ. The covariance for each pair of random variables is calculated as ab ove.

Σ = E[(X

Where:−E[X]×(Y−E[Y])] (17.18)

Σi,j=cov(Xi,Xj) (17.19)

And X is a matrix where each column represents a random variable. The covariance matrix provides a useful tool for separating the structured relationships in a matrix of random variables. This can be used to decorrelate variables or applied as a transform to other variables. It is a key element used in the Principal Component Analysis data reduction method, or PCA for short.

The covariance matrix can be calculated in NumPy using the cov() function. By default, this function will calculate the sample covariance matrix. The cov() function can be called with a single 2D array where each sub-array contains a feature (e.g. column). If this function is called with your data defined in a normal matrix format (rows then columns), then a transpose of the matrix will need to be provided to the function in order to correctly calculate the covariance of the columns. Below is an example that defines a dataset with 5 observations across 3 features and calculates the covariance matrix.

```
# covariance matrix
from numpy import array
from numpy import cov
# define matrix of observations
X = array([
[1, 5, 8],
[3, 5, 11],
[2, 4, 9],
[3, 6, 10],
[1, 5, 10]])
print(X)
# calculate covariance matrix
Sigma = cov(X.T)
print(Sigma)
```

Listing 17.15: Example of calculating a covariance matrix.

Running the example first prints the defined dataset and then the calculated covariance matrix.

```
[[158]
[3511]
[2 4 9]
[3610]
[ 1 5 10]]

[[1. 0.250.75]
[ 0.25 0.5 0.25]
[ 0.75 0.25 1.3 ]]
```

Listing 17.16: Sample output from calculating a covariance matrix.

The covariance matrix is used widely in linear algebra and the intersection of linear algebra and statistics called multivariate analysis. We have only had a small taste in this chapter.

17.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Explore each example using your own small contrived array data.

Load data from a CSV file and apply each operation to the data columns.

Write your own functions to implement each statistical operation.

If you explore any of these extensions, I'd love to know.

17.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

17.7.1Bo oks

Applied Multivariate Statistical Analysis , 2012.
http://amzn.to/2AUcEc5

Applied Multivariate Statistical Analysis, 2015.
http://amzn.to/2AWIViz

Chapter 12 Linear Algebra in Probability & Statistics, Introduction to Linear Algebra, Fifth Edition, 2016.
http://amzn.to/2AZ7R8j

Chapter 3, Probability and Information Theory, Deep Learning 2016.
http://amzn.to/2j4oKuP

17.7.2 API

NumPy Statistics Functions.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/routines.statistics.html

numpy.mean() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.mean.html
numpy.var() API.

https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.var.html
numpy.std() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.std.html
numpy.cov() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.cov.html
numpy.corrcoef() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.corrcoef.html

17.7.3 Articles

Expected value on Wikipedia.
https://en.wikipedia.org/wiki/Expected_value

Mean on Wikipedia.
https://en.wikipedia.org/wiki/Mean

Variance on Wikipedia.
https://en.wikipedia.org/wiki/Variance

Standard deviation on Wikipedia.
https://en.wikipedia.org/wiki/Standard_deviation

Covariance on Wikipedia.
https://en.wikipedia.org/wiki/Covariance

Sample mean and covariance.
https://en.wikipedia.org/wiki/Sample_mean_and_covariance

Pearson correlation coefficient.
https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

Covariance matrix on Wikipedia.
https://en.wikipedia.org/wiki/Covariance_matrix

Estimation of covariance matrices on Wikipedia.
https://en.wikipedia.org/wiki/Estimation_of_covariance_matrices

17.8 Summary

In this tutorial, you discovered how fundamental statistical operations work and how to implement them using NumPy with notation and terminology from linear algebra. Specifically, you learned:

What the expected value, average, and mean are and how to calculate them in NumPy.

What the variance and standard deviation are and how to calculate them in NumPy.

17.8.1W Nhaetx thte covariance, correlation, and covariance matrix are and how to calculate

NumPy.
In the next chapter you will discover the principal component analysis method that makes use of the covariance matrix.

Chapter 18

Principal Component Analysis

An important machine learning method for dimensionality reduction is called Principal Component Analysis. It is a method that uses simple matrix operations from linear algebra and statistics to calculate a projection of the original data into the same number or fewer dimensions. In this tutorial, you will discover the Principal Component Analysis machine learning method for dimensionality reduction and how to implement it from scratch in Python. After completing this tutorial, you will know:

The procedure for calculating the Principal Component Analysis and how to choose principal components.

How to calculate the Principal Component Analysis from scratch in NumPy.

How to calculate the Principal Component Analysis for reuse on more data in scikit-learn.

Let's get started.

18.1 TutorialOverview

This tutorial is divided into 3 parts; they are:

1. What is Principal Component Analysis

2. Calculate Principal Component Analysis

3. Principal Component Analysis in scikit-learn

18.2 WhatisPrincipalComponentAnalysis

Principal Component Analysis, or PCA for short, is a method for reducing the dimensionality of data. It can be thought of as a projection method where data with m-columns (features) is projected into a subspace with m or fewer columns, whilst retaining the essence of the original data. The PCA method can be described and implemented using the tools of linear algebra.

18.2.WhatisPrincipalComponentAnalysis 148 PCA is an operation applied to a dataset, represented by an n × m matrix A that results in a projection of A which we will call B. Let's walk through the steps of this operation.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{bmatrix} \quad (18.1)$$

B=() (18.2)

The first step is to calculate the mean values of each column.

PCAA M=mean(A) (18.3)

Next, we need to center the values in each column by subtracting the mean column value.

C = A −M (18.4)

The next step is to calculate the covariance matrix of the centered matrix C. Correlation is a normalized measure of the amount and direction (positive or negative) that two columns change together. Covariance is a generalized and unnormalized version of correlation across multiple columns. A covariance matrix is a calculation of covariance of a given matrix with covariance scores for every column with every other column, including itself.

V=cov(C) (18.5)

Finally, we calculate the eigendecomposition of the covariance matrix V . This results in a list of eigenvalues and a list of eigenvectors.

values,vectors=eig(V) (18.6)

The eigenvectors represent the directions or components for the reduced subspace of B, whereas the eigenvalues represent the magnitudes for the directions. The eigenvectors can be sorted by the eigenvalues in descending order to provide a ranking of the components or axes of the new subspace for A. If all eigenvalues have a similar value, then we know that the existing representation may already be reasonably compressed or dense and that the projection may offer little. If there are eigenvalues close to zero, they represent components or axes of B that may be discarded. A total of m or less components must be selected to comprise the chosen subspace. Ideally, we would select k eigenvectors, called principal components, that have the k largest eigenvalues.

B=select(values,vectors) (18.7)

Other matrix decomposition methods can be used such as Singular-Value Decomposition, or SVD. As such, generally the values are referred to as singular values and the vectors of the subspace are referred to as principal components. Once chosen, data can be projected into the subspace via matrix multiplication.

P=BT ·A (18.8)

Where A is the original data that we wish to project, BT is the transpose of the chosen principal components and P is the projection of A. This is called the covariance method for calculating the PCA, although there are alternative ways to calculate it.

18.3Calculate Principal Component Analysis

There is no pca() function in NumPy, but we can easily calculate the Principal Component Analysis step-by-step using NumPy functions. The example below defines a small 3 × 2 matrix, centers the data in the matrix, calculates the covariance matrix of the centered data, and then the eigendecomposition of the covariance matrix. The eigenvectors and eigenvalues are taken as the principal components and singular values and used to project the original data.

```
# principal component analysis
from numpy import array
from numpy import mean
from numpy import cov
from numpy.linalg import eig
# define matrix
A = array([
[1, 2],
[3, 4],
[5, 6]])
print(A)
# column means
M = mean(A.T, axis=1)
# center columns by subtracting column means
C=A-M
# calculate covariance matrix of centered matrix
V = cov(C.T)
# factorize covariance matrix
values, vectors = eig(V)
print(vectors)
print(values)
# project data
P = vectors.T.dot(C.T)
print(P.T)
```

Listing 18.1: Example of calculating a PCA manually.

Running the example first prints the original matrix, then the eigenvectors and eigenvalues of the centered covariance matrix, followed finally by the projection of the original matrix. Interestingly, we can see that only the first eigenvector is required, suggesting that we could ×pr 2o jmecatt roiux ro 3nto a 3 × 1 matrix with little loss.

```
[[1 2]
[34]
[5 6]]

[[ 0.70710678 -0.70710678] [
0.70710678 0.70710678]]

[8. 0.]

[[-2.828427120. ] [0. 0. ]
[2.828427120. ]]
```

Listing 18.2: Sample output from calculating a PCA manually.

18.4Principal Component Analysis in scikit-learn

We can calculate a Principal Component Analysis on a dataset using the PCA() class in the scikit- learn library. The benefit of this approach is that once the projection is calculated, it can be applied to new data again and again quite easily. When creating the class, the number of components can be specified as a parameter. The class is first fit on a dataset by calling the fit() function, and then the original dataset or other data can be projected into a subspace with the chosen number of dimensions by calling the transform() function. Once fit, the singular values and principal components can be accessed on the PCA class via the explained_variance_ and components_attributes. The example below demonstrates using this class by first creating an instance, fitting it on a 3 × 2 matrix, accessing the values and vectors of the projection, and transforming the original data.

```
# principal component analysis with scikit-
learn from numpy import array
from sklearn.decomposition import PCA
# define matrix
A = array([
[1, 2],
[3, 4],
[5, 6]])
print(A)
# create the transform
pca = PCA(2)
# fit transform
pca.fit(A)
# access values and vectors
print(pca.components_)
print(pca.explained_variance_)
# transform data
B = pca.transform(A)
print(B)
```

Listing 18.3: Example of calculating a PCA with scikit-learn.

Running the example first prints the 3 × 2 data matrix, then the principal components and values, followed by the projection of the original matrix. We can see, that with some very minor floating point rounding that we achieve the same principal components, singular values, and projection as in the previous example.

```
[[1 2]
[34]
[5 6]]

[[ 0.70710678 0.70710678]
[ 0.70710678 -0.70710678]]

[ 8.00000000e+00 2.25080839e-33]

[[ -2.82842712e+00 2.22044605e-16]
[ 0.00000000e+00 0.00000000e+00]
[ 2.82842712e+00 -2.22044605e-16]]
```

Listing 18.4: Sample output from calculating a PCA with scikit-learn.

18.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Re-run the examples with your own small contrived array data.

Load a dataset and calculate the PCA on it and compare the results from the two methods.

Search for and locate 10 examples where PCA has been used in machine learning papers.

If you explore any of these extensions, I'd love to know.

18.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

18.6.1Bo oks

Section 7.3 Principal Component Analysis (PCA by the SVD), Introduction to Linear Algebra, Fifth Edition, 2016.
http://amzn.to/2CZgTTB

Section 2.12 Example: Principal Components Analysis, Deep Learning 2016.
http://amzn.to/2B3MsuU

18.7 API

numpy.mean() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.mean.html

numpy.cov() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.cov.html

numpy.linalg.eig() API.
https://docs.scipy.org/doc/numpy- 1.13.0/reference/generated/numpy.linalg.eig.html

sklearn.decomposition.PCA API.
http://scikit- learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

18.8 Articles

Principal component analysis on Wikipedia.
https://en.wikipedia.org/wiki/Principal_component_analysis

Covariance matrix.
https://en.wikipedia.org/wiki/Covariance_matrix

18.9 Summary

In this tutorial, you discovered the Principal Component Analysis machine learning method for dimensionality reduction. Specifically, you learned:

The procedure for calculating the Principal Component Analysis and how to choose principal components.

How to calculate the Principal Component Analysis from scratch in NumPy.

18.9.1H Nowe txot calculate the Principal Component Analysis for reuse on more data in scik

In the next chapter you will discover the linear algebra reformulation of linear regression.

Chapter 19

Linear Regression

Linear regression is a method for modeling the relationship between one or more independent variables and a dependent variable. It is a staple of statistics and is often considered a good introductory machine learning method. It is also a method that can be reformulated using matrix notation and solved using matrix operations. In this tutorial, you will discover the matrix formulation of linear regression and how to solve it using direct and matrix factorization methods. After completing this tutorial, you will know:

Linear regression and the matrix reformulation with the normal equations.

How to solve linear regression using a QR matrix decomposition.

How to solve linear regression using SVD and the pseudoinverse.
Let's get started.

19.1 TutorialOverview

This tutorial is divided into 7 parts; they are:

1. What is Linear Regression

    2. Matrix Formulation of Linear Regression

3. Linear Regression Dataset

4. Solve via Inverse

5. Solve via QR Decomposition

6. Solve via SVD and Pseudoinverse

7. Solve via Convenience Function

19.2 WhatisLinearRegression

Linear regression is a method for modeling the relationship between two scalar values: the input variable x and the output variable y. The model assumes that y is a linear function or a weighted sum of the input variable.

y=f(x) (19.1)

Or, stated with the coefficients.

y=b0+b1

×x1 (19.2)

The model can also be used to model an output variable given multiple input variables called multivariate linear regression (below, brackets were added for readability).

y=b0+(b1

×x1)+(b2×x2)+⋯ (19.3)

The objective of creating a linear regression model is to find the values for the coefficient values (b) that minimize the error in the prediction of the output variable y.

19.3 MatrixFormulationofLinearRegression

Linear regression can be stated using Matrix notation; for example:

y = (19.4)

X

· b Or, without the dot notation.

(19.5)

y=X Where X is the input data and each column is a data feature, b is a vector of coefficients and y is a vector of output variables for each row in . X

b

11 x1,2

X21 x2,2

31 x3,2

x ,x= (19.6)

4 1

,x,x,

b x4,2b1 (19.7)

=

2b

y b3 (19.8)

Reformulated, the problem becomes a system of linear equations where the b vector values are unknown. This type of system is referred to as overdetermined because there are more equations than there are unknowns, i.e. each coefficient is used on each row of data. It is a

y1

y=

2

y 3y

4

19.4.LinearRegressionDataset 155 challenging problem to solve analytically because there

are multiple inconsistent solutions, e.g.

multiple possible values for the coefficients. Further, all solutions will have some error because there is no line that will pass nearly through all points, therefore the approach to solving the equations must be able to handle that. The way this is typically achieved is by finding a solution where the values for b in the model minimize the squared error. This is called linear least squares.

$$\sum_{i=1}^{m}\sum_{j=1}^{n}||X \cdot b - y||_{2} = X_{i} \cdot b_{j} - y_{2,j(i)} \quad (19.9)$$

This formulation has a unique solution as long as the input columns are independent (e.g. uncorrelated).

We cannot always get the error e = b

− Ax down to zero. When e is zero, x is an

exact solution to Ax = b. When the length of e is as small as possible, $\hat{x}$ is a least squares solution.

— Page 219, Introduction to Linear Algebra, Fifth Edition, 2016.

In matrix notation, this problem is formulated using the so-named normal equation:

$$X^{T} \cdot X \cdot b = X^{T} \cdot y \quad (19.10)$$

This can be re-arranged in order to specify the solution for b as:

$$b = (X^{T} \cdot X)^{-1} \cdot X^{T} \cdot y \quad (19.11)$$

This can be solved directly, although given the presence of the matrix inverse can be numerically challenging or unstable.

19.4 Linear Regression Dataset

In order to explore the matrix formulation of linear regression, let's first define a dataset as a context. We will use a simple 2D dataset where the data is easy to visualize as a scatter plot and models are easy to visualize as a line that attempts to fit the data points. The example below defines a 5

× 2 matrix dataset, splits it into X and y components, and plots the dataset as a scatter plot.

```
# linear regression dataset
from numpy import array
from matplotlib import pyplot
# define dataset
data = array([
[0.05, 0.12],
[0.18, 0.22],
[0.31, 0.35],
[0.42, 0.38],
[0.5, 0.49]])
print(data)
# split into inputs and
outputs
```

```
X, y = data[:,0], data[:,1] X =
X.reshape((len(X), 1))
# scatter plot
pyplot.scatter(X, y)
pyplot.show()
```

Listing 19.1: Example of example linear regression dataset.

Running the example first prints the defined dataset.

```
[[ 0.05
0.12] [ 0.18
0.22]
[ 0.31 0.35]
[ 0.42 0.38]
[ 0.5 0.49]]
```

Listing 19.2: Sample output from example linear regression dataset.

A scatter plot of the dataset is then created showing that a straight line cannot fit this data exactly.

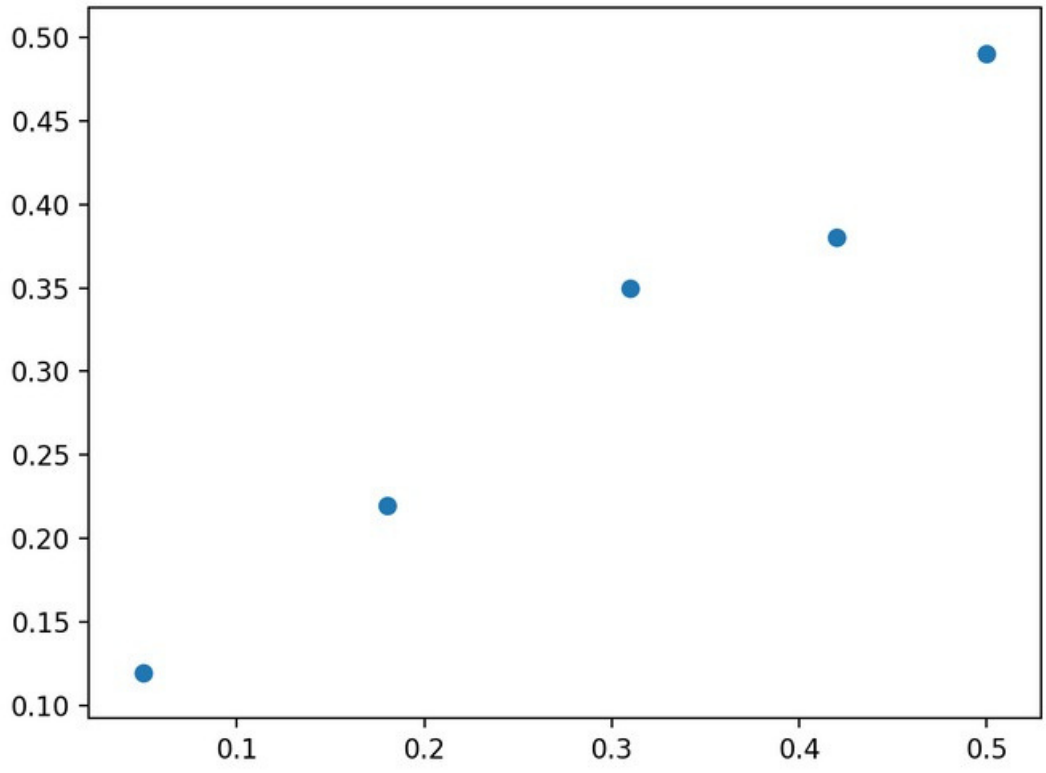


Figure 19.1: Scatter Plot of Linear Regression Dataset.

19.5 Solve via Inverse

The first approach is to attempt to solve the regression problem directly using the matrix inverse. That is, given X, what are the set of coefficients b that when multiplied by X will give y. As we saw in a previous section, the normal equations define how to calculate b directly.

b = (XT

·X)−1·XT·y (19.12)

This can be calculated directly in NumPy using the inv() function for calculating the matrix

```
ibn v=e irnsve(.X.T.dot(X)).dot(X.T).dot(y)
```

Listing 19.3: Example code for solving linear least squares directly.

Once the coefficients are calculated, we can use them to predict outcomes givenX.

```
yhat = X.dot(b)
```

Listing 19.4: Example code for using the coefficients to make a prediction.

Putting this together with the dataset defined in the previous section, the complete example is listed below.

```
# direct solution to linear least squares
from numpy import array
from numpy.linalg import inv
from matplotlib import pyplot
# define dataset
data = array([
[0.05, 0.12],
[0.18, 0.22],
[0.31, 0.35],
[0.42, 0.38],
[0.5, 0.49]])
# split into inputs and outputs
X, y = data[:,0], data[:,1]
X = X.reshape((len(X), 1))
# linear least squares
b = inv(X.T.dot(X)).dot(X.T).dot(y) print(b)
# predict using coefficients
yhat = X.dot(b)
# plot data and predictions
pyplot.scatter(X, y)
pyplot.plot(X, yhat, color='red')
pyplot.show()
```

Listing 19.5: Example of calculating a linear regression solution directly.

Running the example performs the calculation and prints the coefficient vector b.

```
[ 1.00233226]
```

Listing 19.6: Sample output from calculating a linear regression solution directly.

A scatter plot of the dataset is then created with a line plot for the model, showing a reasonable fit to the data.

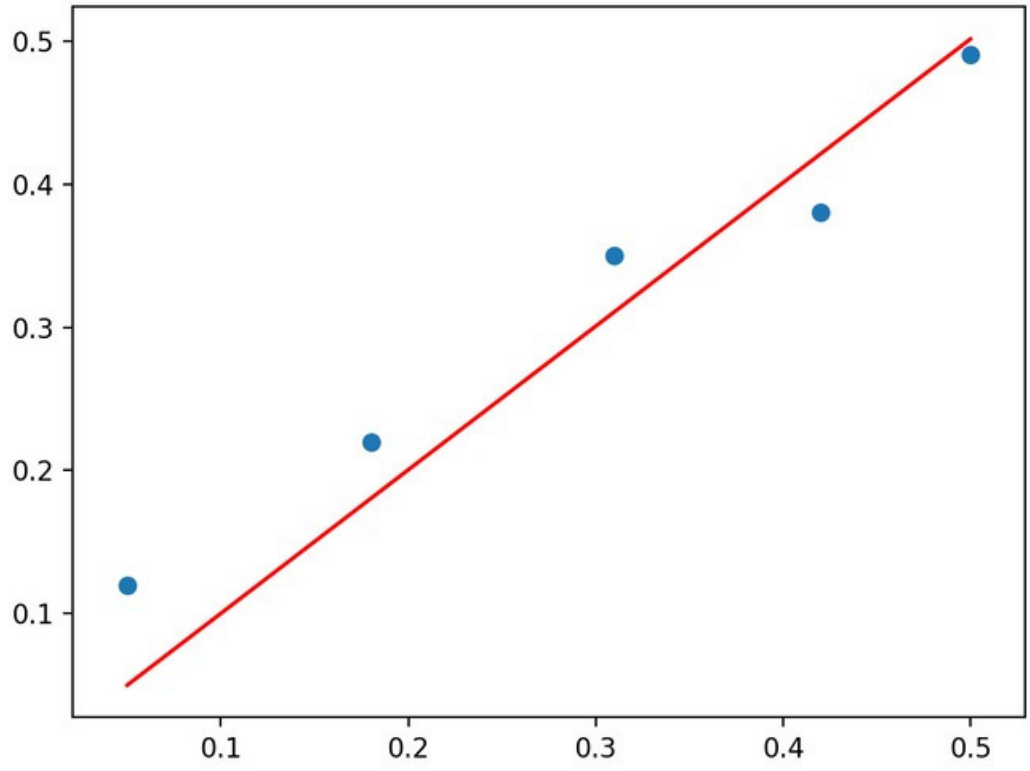


Figure 19.2: Scatter Plot of Direct Solution to the Linear Regression Problem.

A problem with this approach is the matrix inverse that is both computationally expensive and numerically unstable. An alternative approach is to use a matrix decomposition to avoid this operation. We will look at two examples in the following sections.

19.6 Solve via QR Decomposition

The QR decomposition is an approach of breaking a matrix down into its constituent elements.

A = Q

·R (19.13)

Where A is the matrix that we wish to decompose, Q a matrix with the size m

× m, and R

is an upper triangle matrix with the size m

× n. The QR decomposition is a popular approach for solving the linear least squares equation. Stepping over all of the derivation, the coefficients can be found using the Q and R elements as follows:

b = R−1

·QT·y (19.14)

The approach still involves a matrix inversion, but in this case only on the simpler R matrix. The QR decomposition can be found using the qr() function in NumPy. The calculation of the coefficients in NumPy looks as follows:

```
# QR decomposition
Q, R = qr(X)
b = inv(R).dot(Q.T).dot(y)
```

Listing 19.7: Example of calculating a QR decomposition.

Tying this together with the dataset, the complete example is listed below.

```
# QR decomposition solution to linear least
squares from numpy import array
from numpy.linalg import inv
from numpy.linalg import qr
from matplotlib import pyplot
# define dataset
data = array([
[0.05, 0.12],
[0.18, 0.22],
[0.31, 0.35],
[0.42, 0.38],
[0.5, 0.49]])
# split into inputs and outputs
X, y = data[:,0], data[:,1]
X = X.reshape((len(X), 1))
# factorize
Q, R = qr(X)
b = inv(R).dot(Q.T).dot(y)
print(b)
# predict using coefficients
yhat = X.dot(b)
# plot data and predictions
pyplot.scatter(X, y)
pyplot.plot(X, yhat, color='red')
pyplot.show()
```

Listing 19.8: Example of calculating a linear regression solution using a QR decomposition.

Running the example first prints the coefficient solution and plots the data with the model.

```
[ 1.00233226]
```

Listing 19.9: Sample output from calculating a linear regression using a QR decomposition.

The QR decomposition approach is more computationally efficient and more numerically stable than calculating the normal equation directly, but does not work for all data matrices.

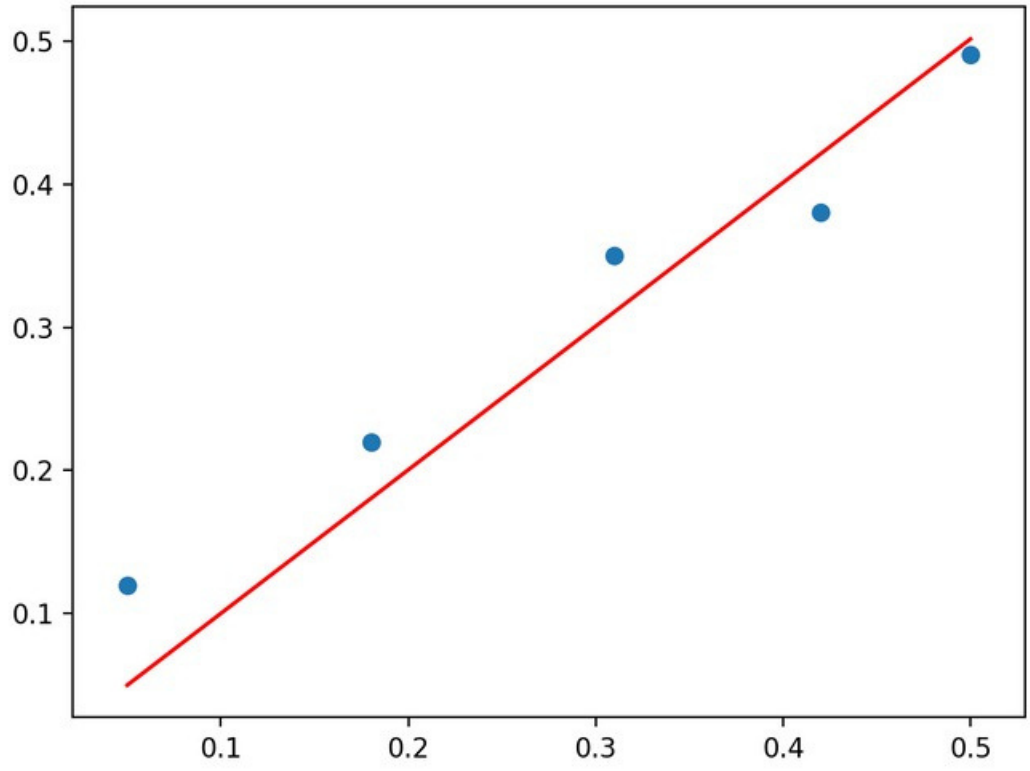


Figure 19.3: Scatter Plot of QR Decomposition Solution to the Linear Regression Problem.

19.7 Solve via SVD and Pseudoinverse

The Singular-Value Decomposition, or SVD for short, is a matrix decomposition method like the QR decomposition.

X = U

·Σ·VT (19.15)

Where A is the real n

× m matrix that we wish to decompose, U is a m × m matrix, Σ (often represented by the uppercase Greek letter Sigma) is an m

× n diagonal matrix, and V T is the transpose of an n

× n matrix. Unlike the QR decomposition, all matrices have a singular-value decomposition. As a basis for solving the system of linear equations for linear regression, SVD is more stable and the preferred approach. Once decomposed, the coefficients can be found by calculating the pseudoinverse of the input matrix X and multiplying that by the output vector y.

b = X +

·y (19.16)

Where the pseudoinverse X+ is calculated as following:

X+=U

·D+·VT (19.17)

Where X+ is the pseudoinverse of X and the + is a superscript, D+ is the pseudoinverse of the diagonal matrix Σ and V T is the transpose of V . NumPy provides the function pinv() to calculate the pseudoinverse directly. The complete example is listed below.

```
# SVD solution via pseudoinverse to linear least
squares from numpy import array
from numpy.linalg import pinv
from matplotlib import pyplot
# define dataset
data = array([
[0.05, 0.12],
[0.18, 0.22],
[0.31, 0.35],
[0.42, 0.38],
[0.5, 0.49]])
# split into inputs and outputs
X, y = data[:,0], data[:,1]
X = X.reshape((len(X), 1))
# calculate coefficients
b = pinv(X).dot(y)
print(b)
# predict using coefficients
yhat = X.dot(b)
# plot data and predictions
pyplot.scatter(X, y)
pyplot.plot(X, yhat, color='red')
pyplot.show()
```

Listing 19.10: Example of calculating a linear regression solution using an SVD.

Running the example prints the coefficient and plots the data with a red line showing the predictions from the model.

```
[ 1.00233226]
```

Listing 19.11: Sample output from calculating a linear regression using an SVD.

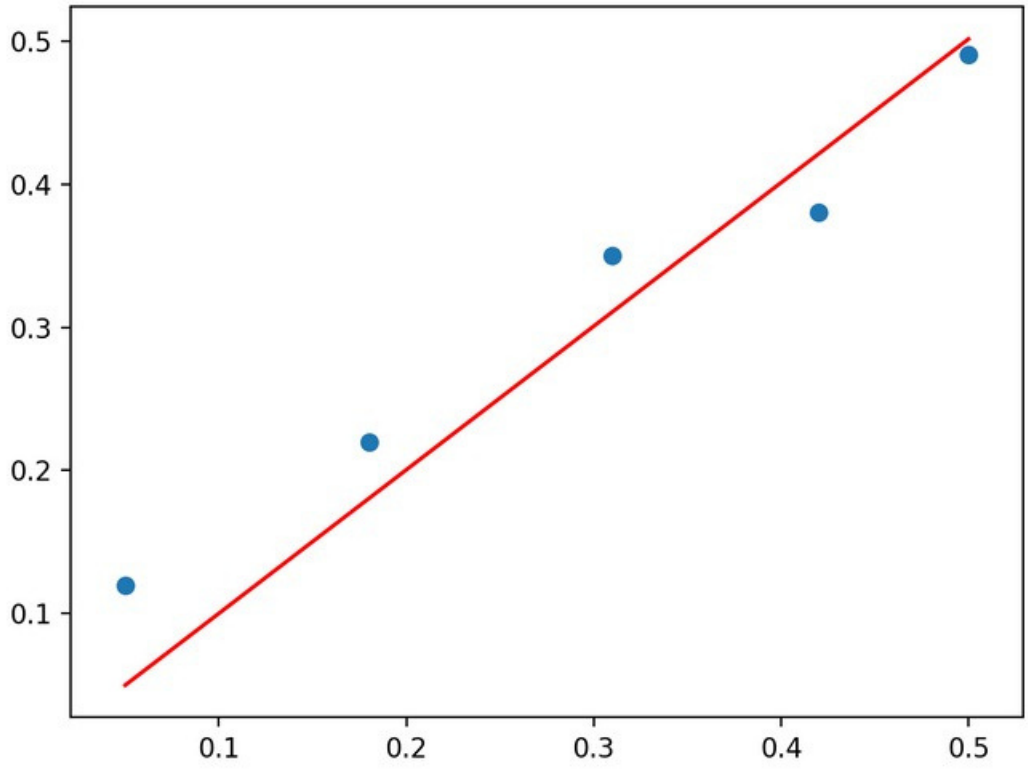


Figure 19.4: Scatter Plot of SVD Solution to the Linear Regression Problem.

19.8 Solve via Convenience Function

The pseudoinverse via SVD approach to solving linear least squares is the de facto standard. This is because it is stable and works with most datasets. NumPy provides a convenience function named lstsq() that solves the linear least squares function using the SVD approach. The function takes as input the X matrix and y vector and returns the b coefficients as well as residual errors, the rank of the provided X matrix and the singular values. The example below demonstrate the lstsq() function on the test dataset.

```
# least squares via convenience function
from numpy import array
from numpy.linalg import lstsq
from matplotlib import pyplot
# define dataset
data = array([
[0.05, 0.12],
[0.18, 0.22],
[0.31, 0.35],
[0.42, 0.38],
[0.5, 0.49]])
# split into inputs and outputs
X, y = data[:,0], data[:,1]
```

```
X = X.reshape((len(X), 1))
# calculate coefficients
b, residuals, rank, s = lstsq(X, y)
print(b)
# predict using coefficients
yhat = X.dot(b)
# plot data and predictions
pyplot.scatter(X, y)
pyplot.plot(X, yhat, color='red')
pyplot.show()
```

Listing 19.12: Example of calculating a linear regression solution using lstsq().

Running the example prints the coefficient and plots the data with a red line showing the predictions from the model.

```
[ 1.00233226]
```

Listing 19.13: Sample output from calculating a linear regression solution using lstsq().

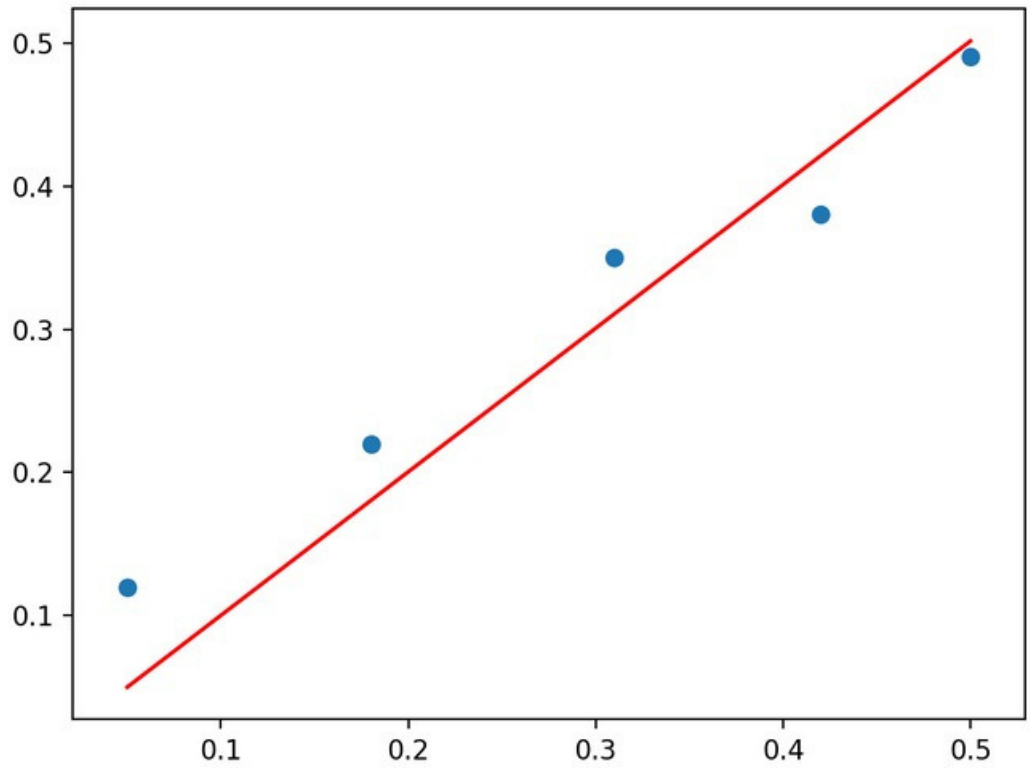


Figure 19.5: Scatter Plot of the lstsq() Solution to the Linear Regression Problem.

19.9 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

Test each linear regression on your own small contrived dataset.

Load a tabular dataset and test each linear regression method and compare the results.

Research and implement alternate ways of solving linear least squares using linear algebra.

If you explore any of these extensions, I'd love to know.

19.10Further Reading

This section provides more resources on the topic if you are looking to go deeper.

19.10.1Bo oks

Section 7.7 Least squares approximate solutions. 2017. No Bullshit Guide To Linear Algebra,

Section 4.3 Least Squares Approximations, 2016. Introduction to Linear Algebra, Fifth Edition,

Lecture 11, Least Squares Problems, Numerical Linear Algebra, 1997.

Chapter 5, Orthogonalization and Least Squares, Matrix Computations, 2012.

Chapter 12, Singular-Value and Jordan Decompositions, Linear Algebra and Matrix Analysis for Statistics, 2014.

Section 2.9 The Moore-Penrose Pseudoinverse, Deep Learning, 2016.

Section 15.4 General Linear Least Squares, Numerical Recipes: The Art of Scientific Computing, Third Edition, 2007.

19.10.2 API

19.10.3 Articles

Linear regression on Wikipedia.

Least squares on Wikipedia.

Linear least squares (mathematics) on Wikipedia.

Overdetermined system on Wikipedia.

QR decomposition on Wikipedia.

Singular-value decomposition on Wikipedia.

Moore-Penrose inverse.

19.11 Summary

In this tutorial, you discovered the matrix formulation of linear regression and how to solve it using direct and matrix factorization methods. Specifically, you learned:

Linear regression and the matrix reformulation with the normal equations.

How to solve linear regression using a QR matrix decomposition.

How to solve linear regression using SVD and the pseudoinverse.

19.11.1 Next

This was the end of the part on statistics and the final chapter. Next you can get additional help in the appendix.

Part VII

App endix

A.6 AskQuestionsAboutLinearAlgebra

There are a lot of places that you can ask questions about linear algebra online given the current abundance of question-and-answer platforms. Below is a list of the top places I recommend posting a question. Remember to search for your question before posting in case it has been asked and answered before.

Linear Algebra tag on Stack Overflow.

Linear Algebra on Quora.

Math Subreddit.

A.7 HowtoAskQuestions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

Boil your question down to the simplest form. E.g. not something broad like my model does not work or how does x work.

Search for answers before asking questions.

Provide complete code and error messages.

A.8BoilC yoourn ctoadec dto wthn teo tAheu smthalolerst possible working ex

You are not alone. If you ever have any questions about deep learning, natural language processing, or this book, please contact me directly. I will do my best to help.

Jason Brownlee
Jason@MachineLearningMastery.com

Appendix B

How to Setup a Workstation for Python

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda.

After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning software. These instructions are suitable for Windows, Mac OS X, and Linux platforms. I will demonstrate them on OS X, so you may see some mac dialogs and file extensions.

B.1 Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda

2. Install Anaconda

3. Start and Update Anaconda

Note: The specific versions may differ as the software and libraries are updated frequently.

B.2 DownloadAnaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

1. Visit the Anaconda homepage.

from the menu and click

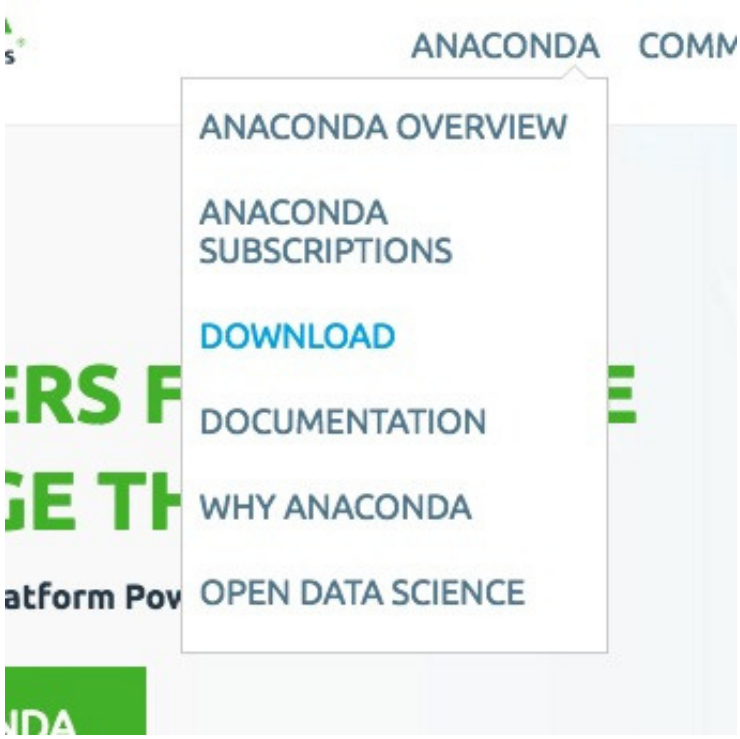


Figure B.1: Click Anaconda and Download.

3. Choose the download suitable for your platform (Windows, OSX, or Linux):

– Choose Python 3.6
– Choose the Graphical Installer



Figure B.2: Choose Anaconda Download for Your Platform.

This will download the Anaconda Python package to your workstation. I'm on OS X, so I chose the OS X version. The file is about 426 MB. You should have a file with a name like:

```
Anaconda3-4.4.0-MacOSX-x86_64.pkg
```

Listing B.1: Example filename on Mac OS X.

B.3 Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

1. Double click the downloaded file.

2. Follow the installation wizard.

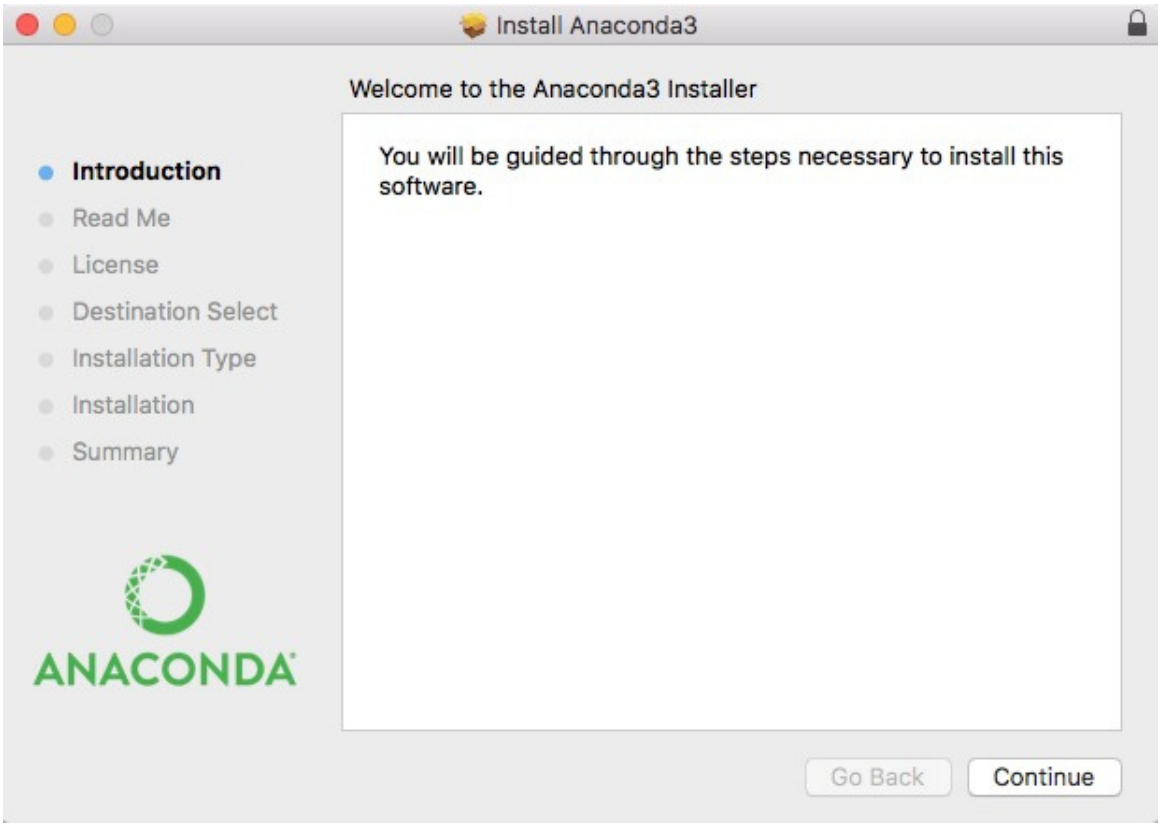


Figure B.3: Anaconda Python Installation Wizard.

Installation is quick and painless. There should be no tricky questions or sticking points.