



Marina Marjanović

Introduction to Machine Learning

A Python Approach to Concepts, Models and Applications



Belgrade, 2023

Singidunum University
Belgrade

Undergraduate Topics in Software and
Data Engineering

Marina Marjanović

INTRODUCTION TO MACHINE LEARNING

A Python Approach to Concepts, Models and Applications

1st edition

Belgrade, 2023

INTRODUCTION TO MACHINE LEARNING - A Python Approach to Concepts, Models and Applications

Author:

Marina Marjanović

Reviewers:

Marko Tanasković

Mladen Veinović

Publisher:

SINGIDUNUM UNIVERSITY

32 Danijelova Street, Bgrade

www.singidunum.ac.rs

For the publisher:

Milovan Stanišić

Prepress:

Miloš Višnjić

Design:

Aleksandar Mihajlović

Year:

2023

Circulation:

400

Printed by:

Birograf, Belgrade

ISBN: 978-86-7912-788-4

Copyright © 2023 Singidunum University, Belgrade

All rights reserved. No part of this work covered by the copyright herein may be reproduced, transmitted, stored or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, without the prior written permission of the publisher.

Contents

INTRODUCTION	VII
CHAPTER 1: INTRODUCTION TO MACHINE LEARNING	3
1.1. The relationship between artificial intelligence, machine learning and deep learning	5
1.2. Machine learning definition	8
1.3. Types of machine learning	9
CHAPTER 2: LINEAR ALGEBRA – REVIEW	15
CHAPTER 3: LINEAR REGRESSION MODEL WITH ONE ATTRIBUTE	25
3.1. System model	27
3.2. Cost function	28
3.3. Gradient descent algorithm	32
3.4. Scaling attribute values	41
3.4.1. Python implementation	44
3.4.2. Python implementation: cost function using numpy library	47
3.4.3. Python implementation: cost function using numpy library	49
3.4.4. Python implementation: cost function depending on the number of iterations	50
3.4.5. Python implementation: plotting a hypothesis with optimal parameters	51
3.4.6. Python implementation: linear regression using numpy library with attribute normalization	51

CHAPTER 4: LINEAR REGRESSION WITH MULTIPLE VARIABLES	59
4.1. Multivariable linear regression model	61
4.2. Multivariable linear regression: gradient descent	62
4.3. Attribute scaling	63
4.4. Attributes and polynomial regression	64
4.5. Normal equation	65
4.5.1. Python implementation: linear regression using the Sklearn library for multivariate minear regression	66
4.5.2. Python implementation: linear regression using only numpy library for multivariate linear regression	70
4.5.3. Python implememtation: normal equation	74
CHAPTER 5: PYTHON (NUMPY, PANDAS, SKLEARN)	77
CHAPTER 6: LOGISTIC REGRESSION	81
6.1. Classification	83
6.2. Hypothesis presentation	83
6.2.1. Python implementation: logistic regression (breast cancer dataset)	85
6.2.2. Python implementation: logistic regression (Iris dataset)	89
CHAPTER 7: REGULARIZATION	93
7.1. Addressing overfitting	96
7.2. Cost function optimization for regularization	97

CHAPTER 8: NEURAL NETWORKS - REPRESENTATION	103
8.1. Model representation	107
8.1.1. Python implementation: feed forward propagation	108
8.1.2. Backward propagation	110
8.1.3. Python implementation: backward propagation	116
CHAPTER 9: ADVICE FOR APPLYING MACHINE LEARNING TECHNIQUES	121
CHAPTER 10: MACHINE LEARNING SYSTEM DESIGN	131
CHAPTER 11: CLUSTERING (Kmeans)	143
11.1. Intuition for k-means clustering	147
CHAPTER 12: DIMENSIONALITY REDUCTION (PCA)	167
Literature	187
Author's bio	189

Introduction

This manual was created because of the need for appropriate written material from the subject Machine Learning, which the author holds in the third year of the Technical Faculty of Singidunum University in Belgrade. During writing, an effort has been made to make the textbook acceptable to readers who have a basic knowledge of the Python or Mat lab programming language. In one part of the textbook, a separate chapter will be devoted to the parts of mathematics (such as linear algebra and mathematical analysis) that are needed to understand algorithms. Through the chapters of this book, readers will learn the most effective machine learning techniques and gain experience in application using the Python programming language. In addition to machine learning theory, readers will learn how to apply these theories in solving new problems and some of Silicon Valley's best practices in innovation in machine learning and Artificial Intelligence. is divided into 12 chapters:

- 01: Introduction to Machine Learning
- 02: Linear Algebra – review
- 03: Linear regression model with one attribute
- 04: Linear Regression with multiple variables
- 05: Python (numpy, pandas, Sklearn)
- 06: Logistic Regression
- 07: Regularization
- 08: Neural Networks - Representation
- 09: Advice for applying machine learning techniques
- 10: Machine Learning System Design
- 11: Clustering (K means)
- 12: Dimensionality Reduction (PCA)

The book is organized to guide the reader from the simplest algorithms such as Linear Regression described in the chapter 3 of the book to the most complex algorithms such as Neural Networks described in the in the chapter 8, allowing the reader to gradually understand even the most complicated algorithms. Chapter 11 of the book describes the Kmeans algorithm as the best-known algorithm used for clustering in cases where we have unlabeled data.

All theoretical considerations are accompanied by examples in Python to familiarize students with this software package. All algorithms are made from scratch using only the numpy library, so that readers can understand each algorithm from start to finish without using generic functions so that later in practice they will be able to create the algorithm that best suits the problem they have.

I will be grateful for any useful suggestions and advice for future corrections and additions to this material.

Belgrade, January 2023

Author

CHAPTER 1



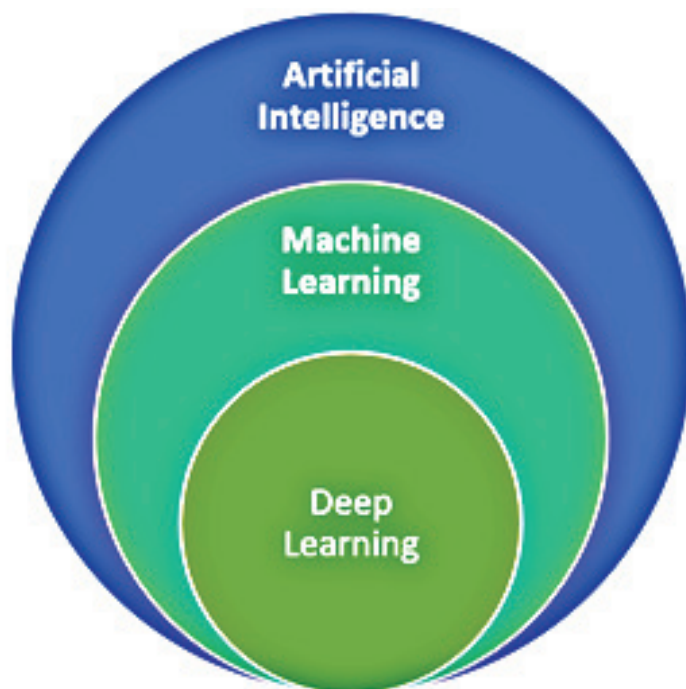
INTRODUCTION TO MACHINE LEARNING

1. INTRODUCTION TO MACHINE LEARNING

Make sure to unconsciously apply machine learning several times throughout the day. When you browse the internet and use Google to search for a particular term, you are utilizing machine learning techniques whose algorithm is intended to rank pages. Machine learning is also used when Facebook or Apple's photo app can identify your pals in your images. The spam filter saves you from having to go through a lot of spam each time you check your email. The goal of machine learning is to teach computers to perform tasks that would otherwise need explicit programming. One suggestion is to have people perform tasks instead of robots. How was this created? What can be done is to use robots for the jobs we want. What can be done is to show the robot performing the task so that it may learn how to do the jobs we want it to do. The robot may then watch what you pick up and put down and attempt to perform the same thing even if you are not present. We are capable of practically anything if we create highly intelligent machines. Neural networks are learning algorithms that imitate how the human brain works, and many experts believe that this is the best way to accomplish this.

1.1. THE RELATIONSHIP BETWEEN ARTIFICIAL INTELLIGENCE, MACHINE LEARNING AND DEEP LEARNING

Deep learning, machine learning, and artificial intelligence are all phrases that have recently become popular despite there not appearing to be any clear distinctions between them. Artificial intelligence is now a part of our daily lives; it is not the future anymore. The simplest way to understand their relationship is to visualize them as concentric circles, with Artificial Intelligence—i.e., the concept that first emerged—fitting into both, followed by Machine Learning—which was later introduced—and finally Deep Learning—which sparks the current explosion of Artificial Intelligence.

Figure 1. Artificial intelligence, machine learning, and deep learning relationship

Source: Nadia BERCHANE (M2 IESCI, 2018)

Artificial intelligence, as the name suggests, is simply the incorporation of human intelligence into robots. A broader idea known as artificial intelligence encompasses everything from traditional machine learning to cutting-edge innovations like deep learning. Artificial intelligence is the term used to describe any "intelligent" behavior displayed by a computer when it carries out tasks in accordance with a set of predetermined rules that address issues (algorithms). Such devices, for instance, can manipulate things, detecting raised hands, and solving other issues. Artificially intelligent machines are typically divided into two categories: general and narrow. Problems like those stated above can be intelligently solved by machines having general-purpose artificial intelligence. Although they have a restricted range of capabilities, narrowly focused artificial intelligence computers can execute tasks very well, sometimes even better than humans. Pinterest's image classification system is an illustration of artificial intelligence with a specific use. Machine learning, as its name suggests, is essentially the process of giving computer systems the capacity to "learn." Machine learning aims to give computers the ability to learn on their own utilizing data that is already accessible and make precise predictions.

Machine learning represents a subset of artificial intelligence; in fact, it is simply a technique for its realization. Machine learning can be understood as a set of mathematical models for training that can teach how to make decisions based on the data at their disposal. Machine learning training involves giving algorithms a lot of data that allows him to learn more about the information processed.

Here is an illustration of a table that classifies fruits according to their characteristics:

Table 1. Example of machine learning dataset

Weight (grams)	Texture	Label
155	Rough	orange
180	Rough	orange
135	Smooth	apple
110	Smooth	apple
120	Smooth	?

This table illustrates how different fruits vary in terms of their weight and feel. The type of fruit is not provided; just weight and texture are in the last row. It is possible to create a machine learning system to attempt and determine if the fruit is an orange or an apple. The algorithm will learn the variations between orange and apple after being trained using well-known data. Therefore, the sort of fruit with these qualities may be correctly predicted given weight and texture information.

As it is already mentioned, machine learning is a subset of deep learning, which is just a method for attaining machine learning. In other words, deep learning is the next development in machine learning, and its algorithms are modeled after the ways the human brain processes information. Deep learning algorithms can be trained to carry out the same functions for computers in the same way that our brains classify various types of information and recognize patterns.

Before we even realize it, the brain compares any new input to something it already knows, which is the same idea as deep learning algorithms. While machine learning needs that these functions be manually input, deep learning may automatically identify the traits that will be utilized for categorization. Deep learning, as opposed to machine learning, needs better processors and a lot more training data to produce reliable results.

1.2. MACHINE LEARNING DEFINITION

Even among machine learning practitioners, there is no single definition of what is and what is not machine learning. As far back as 1950, the man who developed games, Arthur Samuel, defined machine learning as a field of learning that gives computers the ability to learn without explicit learning. The fascinating thing about the gameplay program he created for the checkerboard is that Arthur Samuel wasn't a particularly skilled player of the game. He programmed perhaps tens of thousands of games against himself, and by seeing which board situations typically result in victories and which positions typically result in losses, the software ultimately learnt which moves were excellent and which moves were poor. Finally, the program accumulated enough expertise to become a greater checkers player than Arthur Samuel himself. It was an impressive outcome. This was achievable because, unlike humans, computers can play tens of thousands of games against one another. In a later definition of machine learning, Thomas Mitchel of Carnegie Mellon University states that a well-posed learning issue is characterized as follows: 'a computer program learns from experience E in relation to some task T and some measure of performance P , if its performance on T is measured by performance P , improves with experience E .' For an example of playing checkers, experience E would be an experience where the program itself could play tens of thousands of games. Task T would be the task of playing checkers, and a measure of P 's performance would be the probability of winning the next game against a new opponent. Let's observe the example of spam separation. Let's say an e-mail program looks at which e-mails a client has marked or not as spam, and based on that, the e-mail program better learns how to filter spam. In this example, classifying emails is a T task. Observing how a client marks email as spam or e-mail represents experience E . The number of e-mails that is properly classified is a measure of P 's performance that will improve with experience E .

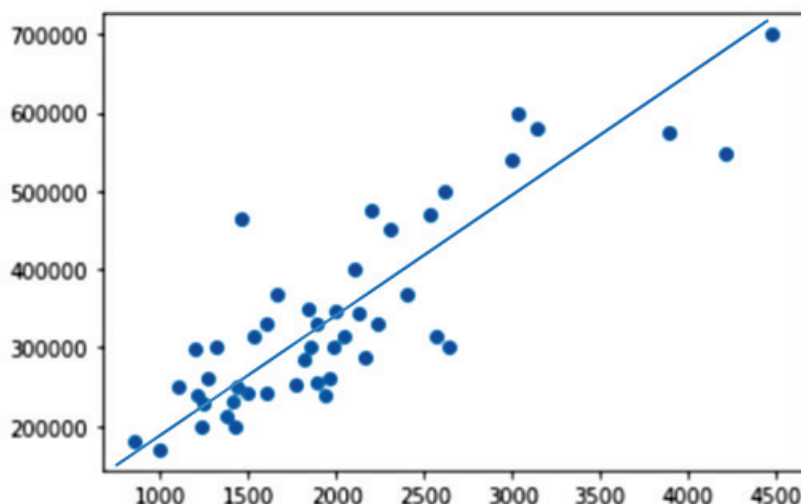
1.3. TYPES OF MACHINE LEARNING

There are many different types of learning algorithms. The two main categories are supervised learning and unsupervised learning. While unsupervised learning entails letting the computer figure things out on its own, supervised learning aims to educate the computer how to complete a task. As an illustration of supervised learning, we'll use the prediction of apartment prices. The data collection included apartment rent costs from Portland and other cities in Oregon. If we wanted to represent something graphically, we would put the square footage of various homes on the x-axis and their costs in thousands of dollars on the y-axis.

Finding the optimal learning system that can forecast the price of a 750 square meter property based on these facts is therefore important. Making a straight line across the data is one thing a learning system might desire. That suggests that this home might be sold for around \$150,000. (Figure 2). But perhaps there is a more effective learning algorithm that can be utilized instead of this one. For instance, we may decide that a quadratic function or a second-order polynomial is a better fit for the data than a straight line. The house can be estimated to cost \$200,000 and if this is done and a prediction is made. How to pick the optimal function will be covered among other topics later. This is an example of a 'supervised learning' algorithm, where the word supervised learning refers to the fact that we have provided the algorithm with a collection of data that contains only correct answers, where each example in this dataset has the appropriate price provided. What was the exact price at which the house was sold? The algorithm's job is to generate more of these accurate results, such as a brand-new 750-square-foot home that is not included in the statistics. This issue is supervised learning-related as well as regression model.

Table 2. Machine learning example (regression problem)

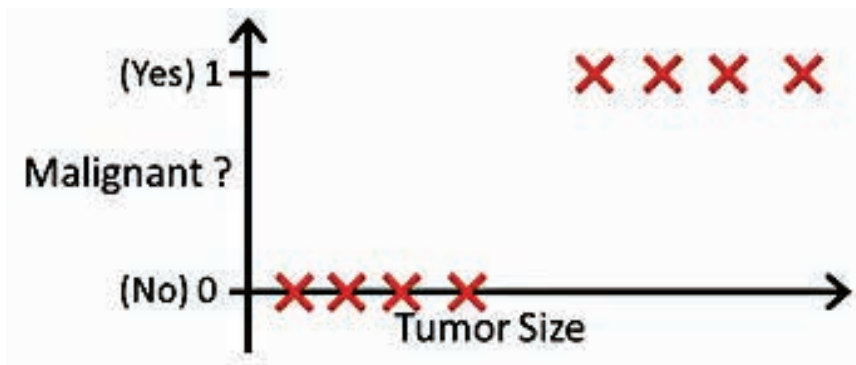
Size of the house (square meters)	House price (\$1000)
2104	460
1416	232
1534	315
852	178
...	...
750	?

Figure 2. Graphic representation of the price of the house in relation to the size of the house

Regression problems involve predicting a continuous value, in this case a price. Another example of supervised learning is the grading of breast cancer as malignant or benign. If a person discovers a breast tumor or lump in the breast, a benign tumor is one that is harmless, but a malignant tumor is one that is destructive and deadly. Depending on whether we are examining the dataset and if a given sample is in the dataset, the y-axis has a value of 0 or 1, i.e., yes, or no, i.e., malignant, or benign, respectively. The tumor's size is shown on the x-axis. Let's assume that the dataset contains information about a tumor of this size that was discovered to be benign. Unfortunately, we also discovered several cancerous tumor cells. In this illustration, five examples of benign tumors are shown with a value of one, and five examples of malignant tumors are also shown. The question of machine learning is: can you determine how probable it is that a friend's breast tumor, which is unpleasant to say has a size that corresponds to a value of one, is malignant as opposed to benign? Here is an illustration of supervised learning applied to a categorization issue. Classification refers to the fact that we are attempting to forecast the result of a discrete value equal to zero or one, malignant or benign, in this situation. It turns out that classification problems occasionally have more than two alternative output values using the function we wished to apply to regression problems. For instance, there could be three different forms of breast cancer. Therefore, you can attempt to forecast a discrete value of output of zero, one, two, or three, where zero can denote a

benign tumor, indicating that there is no cancer, and one can denote type one cancer, a value of two another types of cancer, and three can denote a possible cancer. Since they are a way out of a discrete set of values, this will also be a classification challenge. Another graphical method of presenting the categorization problem employs various symbols for negative and positive examples. Now, in this case, we only use one characteristic or attribute—namely, the tumor's size—to determine the tumor's malignancy or benignity. We can have more characteristics or qualities in other machine learning situations, such as patient age and tumor size. Given this set of data, the learning algorithm can attempt to distinguish between benign and malignant tumors by drawing a straight line on the data. As a result, the learning algorithm may choose to draw this line to distinguish between the two causes of tumors. Using this information, we expect to determine that your friend's tumor is more likely to be benign than malignant. In this example, the patient's age and the size of the tumor were two factors. Other machine learning issues will frequently require extra functions. The learning algorithms that will be discussed in this textbook will consider an infinite number of qualities, not just one or two. We can employ an endless number of attributes and functions to solve some learning issues. How should we handle the limitless variety of functions? Due to a memory shortage, how can you store an infinite number of objects in the computer? This handbook will address all of this in more details later.

Figure 3. Graphic representation of tumor classification in relation to its size (classification problem)



On the other hand, unsupervised learning represents a sort of machine learning that involves previously unidentified patterns in a dataset, minimal human supervision, and no pre-existing data labeling. Unsupervised learning has allowed us to approach issues with very little understanding of what our solutions might look like, in contrast to supervised learning, which often includes human-labeled data. Without human intervention, a structure can be created based on the correlations between the data attributes. If you have a collection of 1,000,000 unique genes and figure out how to automatically arrange those genes into groups that are related in some manner or use different characteristics, such lifetime, location, and so on, you may use clustering as an example. The "cocktail party algorithm," permits the detection of structures in data without clustering.

CHAPTER 2



LINEAR ALGEBRA - REVIEW

2. LINEAR ALGEBRA - REVIEW

Machine learning's use of vector space and matrices to represent linear equations is referred to as linear algebra. This branch of mathematics covers anything from coding techniques and algorithms (such as Regularization, Deep learning, One hot encoding, Principal Component Analysis (PCA), Single Value Decomposition (SVD), etc.) to the notations that are used to describe the operations of the machine learning algorithm. It serves as the main pillar of machine learning.

Here are a few instances that show the various applications of linear algebra:

Dataset and Data Files: In linear algebra, a data is a matrix or a data structure. A dataset is a collection of numbers or data organized in a table. While columns indicate the features of it, rows represent the observations. The length of each row is the same. Data is therefore vectorized. To make calculations simpler and more accurate, rows are pre-configured and added to the model one at a time. Linear algebra might be used in some of the following examples that:

One hot encoding: For categorical variables, it is a well-liked encoding that makes algebraic operations simpler. Each category has its own column, and each example has its own row in a table.

Linear regression: One of the statistical techniques is linear regression, which is used to forecast numerical results for regression issues as well as to describe the relationship between variables. A dataset or matrix is X , a coefficient is k , and the result is y .

Recommender Systems: SVD functions to cleanse data using Euclidean distance or dot products with the use of Linear Algebra. For instance, recommendations for books on Amazon are based on our past purchases, excluding other unrelated things.

Let's start with the terms.

Table 3. Linear algebra terms example

Scalar	Vector	Matrix
3	1 4 6	1 0 3 3 1 0 0 3 1

- Scalar represents a single value.
- Arrays represent a bunch of values.
- Matrices represent a bunch of arrays.
- Vector represents matrix with one column or one row.
- The dimensions of a matrix are given as #rows x #columns

This chapter provides a summary of the basic operations implemented in python such as:

- Checking vector and matrix size
- Vector and matrix creation
- Vector and matrix addition
- Matrix multiplication
- Dot product

Python implementation

```
▶ #1. Create vector  
y=[123,56778,666,444]
```

```
[ ] #2. find vector size  
def size_vector(u):  
    return len(u)
```

```
[ ] size_vector(y)
```

```
4
```

```
[ ] len(y)
```

```
4
```

```
[ ] y[0]
```

```
123
```

```
[ ] #3.create matrix M  
    M=[[1,2,3,4],[5,6,7,8]]
```

```
[ ] len(M)#number of rows
```

```
2
```

```
[ ] len(M[0])
```

```
4
```

```
[ ] #4.compute the addition of u and v  
    #u=[1,2,3,4]  
    #v=[5,6,7,8]  
    #u+v=w=[1+5,2+6,3+7,4+8]=[6,8,10,12]
```

```
[ ] u=[1,2,3,4]  
    v=[5,6,7,8]  
    u+v
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
[ ] def add(u,v):  
    w=[]  
    for i in range(len(u)):  
        w.append(u[i]+v[i])  
    return w
```

```
[ ] add(u,v)
```

```
[6, 8, 10, 12]
```

```
[ ] #5.Compute the addition of two matrices M and N
#M=[[1,2,3],[4,5,6]] R2X3
#N=[[7,8,9],[10,11,12]] R2X3
#W=[[1+7,2+8,3+9 ], [4+10,5+11,6+12]]=[[8,10,12],[14,16,18]]
```

```
[ ] def add_matrix(U,V):
    W=[]
    for i in range(len(U)):
        temp=[]
        for j in range(len(U[0])):
            temp.append(U[i][j]+V[i][j])
        W.append(temp)
    return W
```

```
[ ] M=[[1,2,3],[4,5,6]]
    N=[[7,8,9],[10,11,12]]
```

```
[ ] add_matrix(M,N)
```



```
[[8, 10, 12], [14, 16, 18]]
```

```
[ ] #6.Multiply matrix M with vector v

A=[[1,4,5,12],[-5,8,9,0],[-6,7,11,19]] #R3X4
v=[1,2,3,4] #R4X1
#Multiply matrix M with vector v
def dot(M,v):
    u=[]
    for i in range(len(M)):
        val=0.0
        for j in range(len(M[0])):
            val=val+M[i][j]*v[j]
        u.append(val)
    return u #R3X1
```

```
[ ] #7. Cross product between u and v (u*v')
```

```
[ ] def cross(u,v):
    M=[]
    for i in range(len(u)):
        N=[]
        for j in range(len(v)):
            N.append(u[i]*v[j])
        M.append(N)
    return M
```

```
[ ] u=[1,2,3,4]
    v=[5,6,7,8]
```

```
[ ] cross(u,v)
```

```
👤 [[5, 6, 7, 8], [10, 12, 14, 16], [15, 18, 21, 24], [20, 24, 28, 32]]
```

```
[ ] #8. Compute the element dot product of u and v
    #u=[1,2,3,4]
    #v=[5,6,7,8]
    #z=[5,12,21,32]
```

```
[ ] def ew_dot(u,v):
    w=[]
    for i in range(len(u)):
        w.append(u[i]*v[i])
    return w
```

```
[ ] ew_dot(u,v)
```

```
[5, 12, 21, 32]
```

```
[ ] #9.multiply the scale a with vector v
```

```
#v=[1,2,3,4,5]
#a=5
#w=a*v=[5,10,15,20,25]
```

```
[ ] a=5
v=[1,2,3,4,5]
a*v
```

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

```
[ ] def scale(a,v):
    u=[]
    for i in range(len(v)):
        u.append(a*v[i])
    return u
```

```
[ ] scale(a,v)
```

```
[5, 10, 15, 20, 25]
```

```
[ ] #10.Multiply the scale a with matrix M
```

```
def scale_matrix(a,M):
    N=[]
    for i in range(len(M)):
        U=[]
        for j in range(len(M[i])):
            U.append(a*M[i][j])
```

```
[ ] A=[[1,4,5,12],[-5,8,9,0],[-6,7,11,19]]
    a=5
    scale_matrix(a,A)
```

```
[[5, 20, 25, 60], [-25, 40, 45, 0], [-30, 35, 55, 95]]
```

```
[ ] #11. Transpose a matrix M
```

```
[ ] def transpose(M):
    N=[]
    for j in range(len(M[0])): #over columns
        U=[]
        for i in range(len(M)): #over rows
            U.append(M[i][j])
        N.append(U)
    return N
```

```
[ ] A=[[1,2,3],[4,5,6]]
```

```
[ ] transpose(A)
```

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


CHAPTER 3



LINEAR REGRESSION MODEL WITH ONE ATTRIBUTE

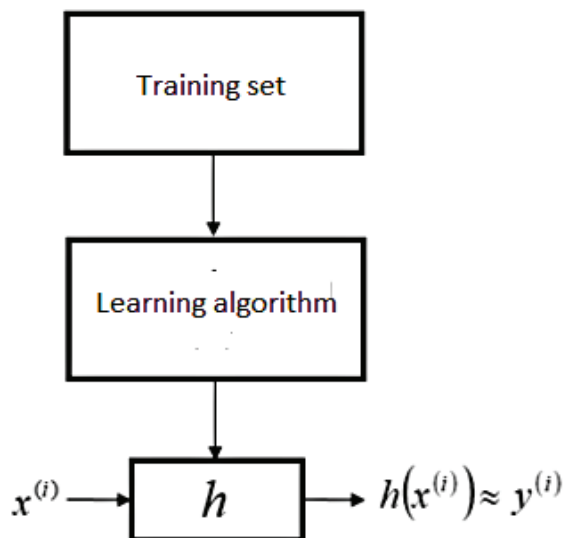
3. LINEAR REGRESSION MODEL WITH ONE ATTRIBUTE

3.1. SYSTEM MODEL

To be able to get acquainted with different algorithms in machine learning in the continuation of this book, it is necessary to adopt a model and notations that we will use in the future. Some label $x^{(i)}$ denotes the input variable of the i -th sample, a $y^{(i)}$ indicates its output. If we look at Table 2, $x^{(1)}=2104$ represents the size of the first house in the dataset while its corresponding output $y^{(1)}=460$ represents its appropriate price. A pair $(x^{(i)}, y^{(i)})$ the input-output is called the i -th training sample and dataset $(x^{(i)}, y^{(i)})$ where $i=1,2,...,m$ is called a training set. Later, we will use the notation X to denote the space of input variables and Y for output label.

In order to describe the problem of supervised learning, we will define the function $h: X \rightarrow Y$ such that $h(x)$ represents the most approximate assumption of the value of y , called the hypothesis. Figure 4 presents a model for machine learning.

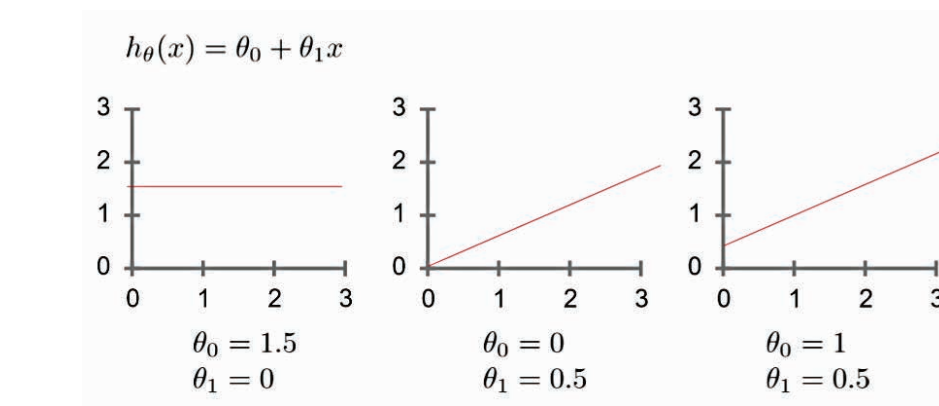
Figure 4. Machine learning model



3.2. COST FUNCTION

Hypothesis $h(x^{(i)})$ will be used as $h_{\theta}(x^{(i)})$ because it is a function of a parameter θ , where in case of linear regression model with one attribute will be represented such as $h_{\theta}(x) = \theta_0 + \theta_1 x$, where θ_1 represents the slope, and θ_0 represents an intercept on the y axis. Figure 5 shows the hypothesis for the case of linear regression with one attribute for different parameter values θ_0 and θ_1 .

Figure 5. Hypothesis in the case of linear regression model with one variable for different parameter values θ_0 i θ_1



The accuracy of the hypothesis function can be measured using the cost function. This requires an average difference of all hypothesis results $h_{\theta}(x^{(i)})$ and real value $y^{(i)}$.

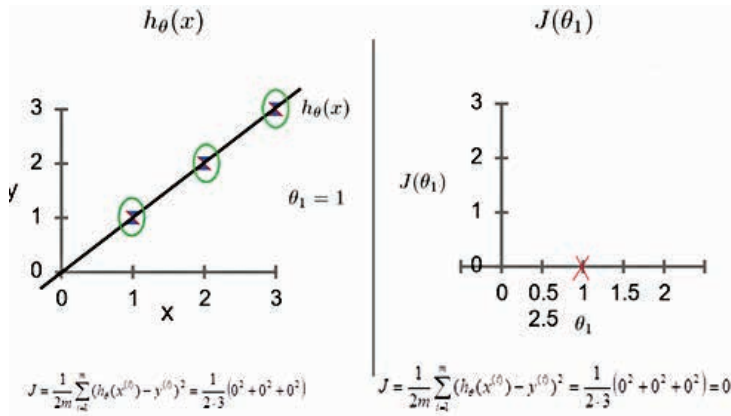
$$J = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

This function represents the mean square error generated by a hypothesis prediction $h_{\theta}(x^{(i)})$ of the samples $x^{(i)}$ and their real values $y^{(i)}$ for $i=1,2,...,m$ where m represents the number of samples in the training set.

For the sake of simplicity, we take the hypothesis model where one of the parameters is equal to zero, e.g. $\theta_0=0$ and change the parameter values θ_1 . We want to see how the cost function behaves. The figure shows the case when the parameter $\theta_1=1$ where three samples were given $(x^{(1)}, y^{(1)})=(1, 1)$, $(x^{(2)}, y^{(2)})=(2, 2)$ i $(x^{(3)}, y^{(3)})=(3, 3)$. For the given parameter values $\theta_0=0$ and $\theta_1=1$, hypotheses is $h_{\theta}(x)=x$. Then, the given samples we have $h_{\theta}(x^{(1)})=y^{(1)}=1$, $h_{\theta}(x^{(2)})=y^{(2)}=2$ and $h_{\theta}(x^{(3)})=y^{(3)}=3$. Given that the assumption of values by the hypothesis in all three samples coincides

with the actual values, the cost function has a value $J(\theta_1)=0$ which also represents its global minimum because the cost function cannot have a value less than 0.

Figure 6. The value of the cost function for $\theta_0=0$ i $\theta_1=1$ and 3 samples $(x^{(1)}, y^{(1)}) = (1, 1)$, $(x^{(2)}, y^{(2)}) = (2, 2)$ and $(x^{(3)}, y^{(3)}) = (3, 3)$

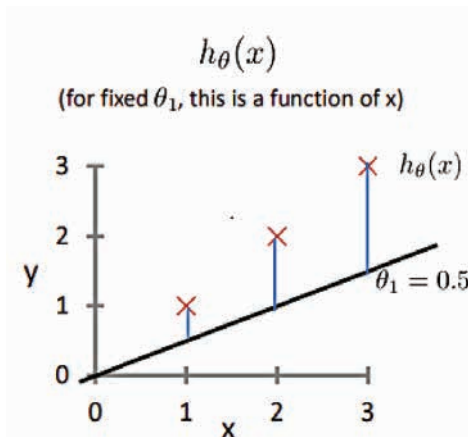


If for different values θ_0 and θ_1 we calculate the value of cost function taking into account, the differences between hypothesis $h_{\theta}(x)$ and the true values y are:

$$h_{\theta}(x^{(1)}) - y^{(1)}, h_{\theta}(x^{(2)}) - y^{(2)}, h_{\theta}(x^{(3)}) - y^{(3)}$$

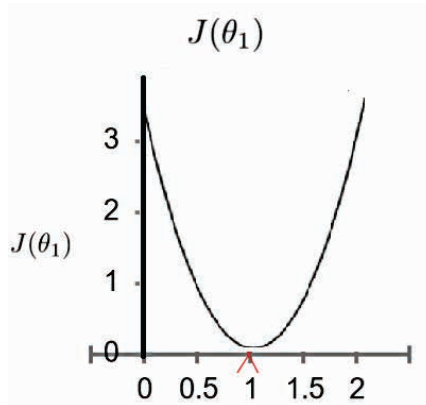
changes in parameters θ_0 and θ_1 will affect the change in cost function. Figure 7 shows these distances when $\theta_1=0.5$.

Figure 7. Example of a hypothesis $h_{\theta}(x) = \theta_0 + \theta_1 x$ for $\theta_0=0$ and $\theta_1=0.5$ and 3 samples $(x^{(1)}, y^{(1)}) = (1, 1)$, $(x^{(2)}, y^{(2)}) = (2, 2)$ and $(x^{(3)}, y^{(3)}) = (3, 3)$



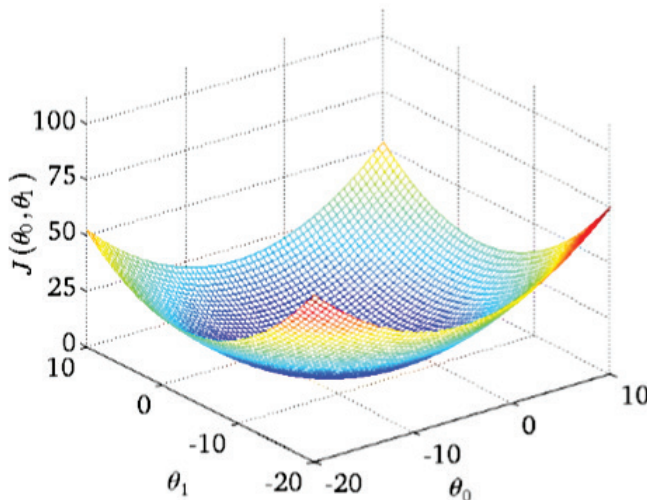
If we continue to change the parameter θ_1 for fixed value of θ_0 ($\theta_0=0$) and by calculating the differences between the predicted values by the hypothesis and the actual values, we obtain the shape of the cost function as it is presented in the Figure 8. We see that for value $\theta_1=1$, cost function reaches its global minimum and becomes $J(\theta_1)=0$.

Figure 8. Cost function value for different values of θ_1 ($\theta_0=0$)



For the sake of simplicity and representing the cost function on 2D graphic, we changed only one parameter θ_1 . If we change both parameters θ_0 and θ_1 , the cost function looks like in the Figure 9.

Figure 9. Cost function when parameters are changed at the same time θ_0 and θ_1



You can notice that this function has the shape of a bowl whose bottom represents the global minimum. To see more clearly the values θ_0 and θ_1 for which this function reaches the global minimum, it is more illustrative if the cross section is shown as in Figures 10, 11 and 12.

A graph with numerous contour lines is a cross section. Every point along a contour line made up of two variables has the same value. One would anticipate getting the same value of the cost function for any hue when travelling along the "circle" with it. If we look at Figure 10, 3 red crosses located on the top green line have the same value for $J(\theta_0, \theta_1)$.

The second cross located on the blue contour corresponds to the cost function of a linear regression model whose hypothesis has a parameter $\theta_0=800$ and $\theta_1=-0.15$. Figure 11 shows the cost function whose regression model has a hypothesis with parameters $\theta_0=360$ and $\theta_1=1$.

Figure 12 shows the case when the cross is in the center of the contour which means that the cost function has reached the global minimum for the parameter values $\theta_0=250$ and $\theta_1=0.12$.

If we compare the figure of 10, 11 and 12, we see that if the cross goes towards the center, the more the position of the hypothesis corresponds to the given data.

Figure 10. $\theta_0=800$ and $\theta_1=-0.15$.

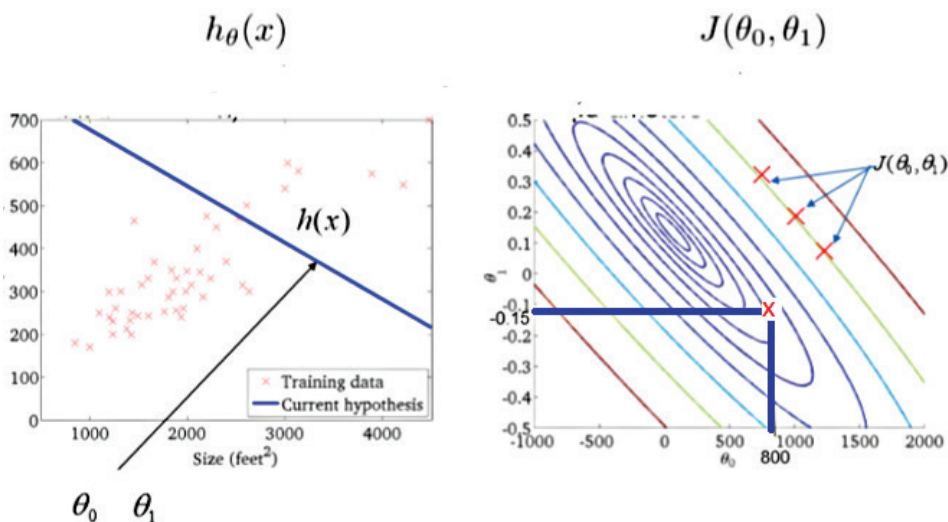
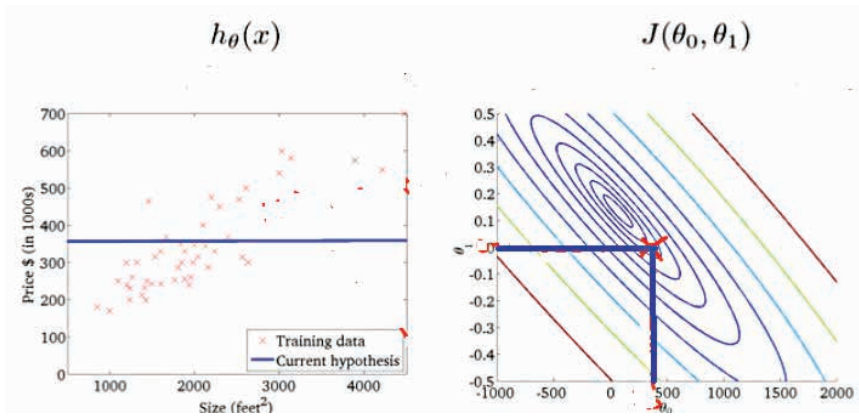
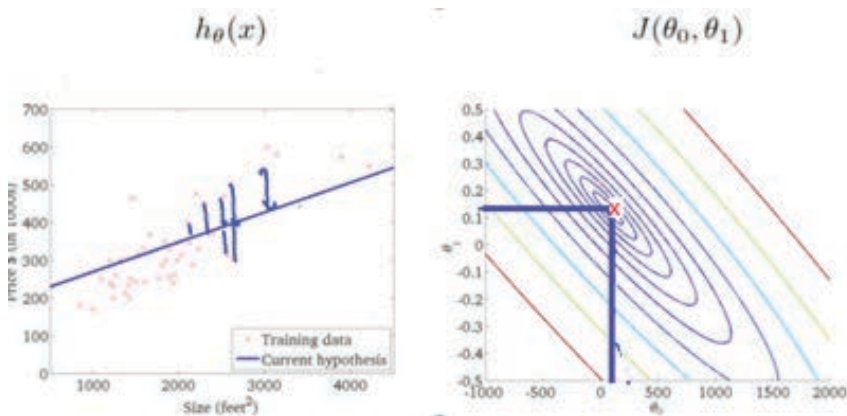
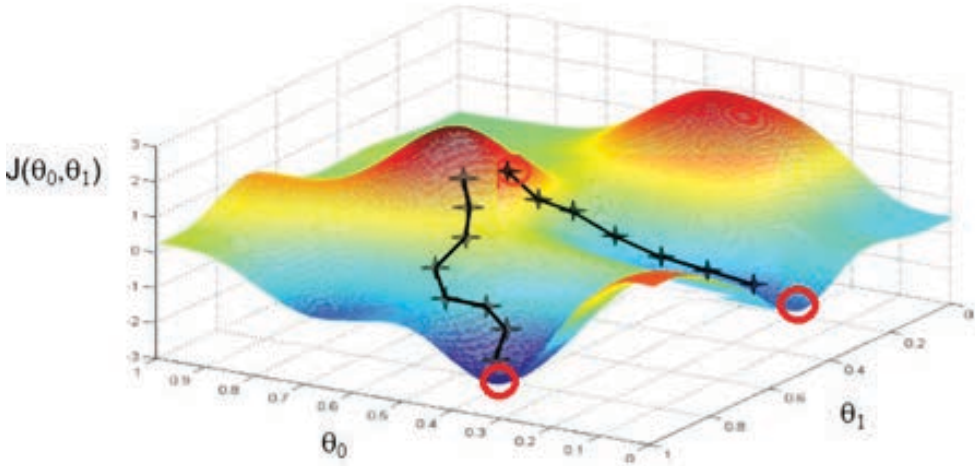


Figure 11. $\theta_0=360$ and $\theta_1=1$.Figure 12. $\theta_0=250$ and $\theta_1=0.12$.

3.3. GRADIENT DESCENT ALGORITHM

There is an explanation of a hypothesis function and a way to gauge how well it fits the facts. The parameters should now be estimated based on the hypothesis. The descent begins to descend here. Imagine that we graph the function of a hypothesis based on its parameters θ_0 and θ_1 at the same time, we graphically represent the cost function as a parameter estimation function. All possible parameter values for θ_0 are shown on the x-axis, and all possible parameter values θ_1 are shown on the y-axis. On the z axis we can see the value of the cost function $J(\theta_0, \theta_1)$. The crosses on this graph will be the result of the cost function using our hypothesis with those specific theta parameters. Figure 13 shows such a hypothesis settings.

Figure 13. Change of the cost function depending on parameters θ_0 and θ_1



When our cost function is at the absolute bottom of our chart, we know that we have been successful. The minimum cost function is shown by the red circles in Figure 13. Calculating the gradient (tangent lines to the function) of our cost function is how we arrive at the minimum. At that time, the slope of the tangent serves as an extract and directs us in the right path. We move in the direction of the steepest descent toward the cost function. The learning rate, a parameter α , controls how big each step is. For instance, our parameter α determines the step between each cross in the graph above. Lower value of α it would result in a smaller step and a larger parameter α with bigger step. The direction in which the step is performed is determined by the partial derivatives $\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0}$ and $\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1}$. Depending on where it starts on the chart, it could end at different points. The picture above shows us two different starting points that end in two different places. The gradient descent algorithm is a repetition if it converges.

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

where $j=0,1$ represents the feature index.

In every iteration j , we need to update the parameters simultaneously $\theta_0, \theta_1, \dots, \theta_n$ where n is the total number of features.

Table 4. Gradient descent algorithm (correct and incorrect way)

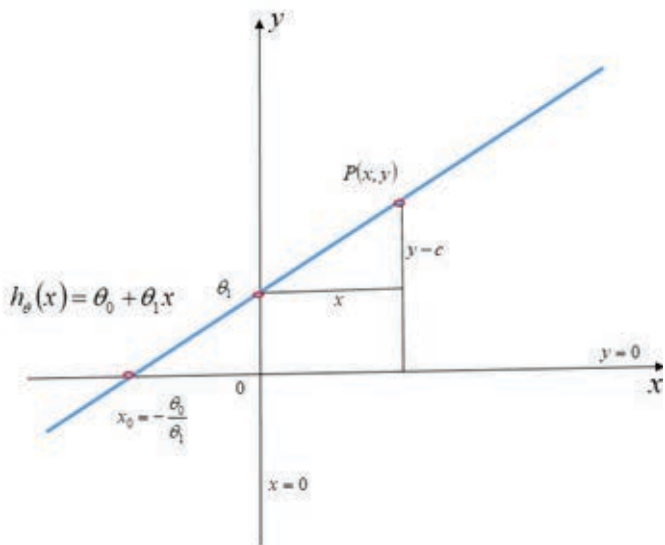
Correct	Incorrect
$temp0 := q_0 - a \frac{\partial}{\partial q_0} J(q_0, q_1)$	$temp0 := q_0 - a \frac{\partial}{\partial q_0} J(q_0, q_1)$
$temp1 := q_1 - a \frac{\partial}{\partial q_1} J(q_0, q_1)$	$\theta_0 := temp0$
$\theta_0 := temp0$	$temp1 := q_1 - a \frac{\partial}{\partial q_1} J(q_0, q_1)$
$\theta_1 := temp1$	$\theta_1 := temp1$

Table 4 shows the gradient descent algorithm. The difference between the correct and incorrect version is that in the correct version the parameters are updated simultaneously and thus in one iteration of the gradients of the cost function $\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ and $\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$ we use the same parameter values for θ_0 and θ_1 .

So, to clarify, let's repeat what has been done so far. We first applied a linear regression, then calculated the bad function, then calculated its derivatives for each of the variables, and finally updated the parameters in $h_{\theta}(x) = \theta_0 + \theta_1 x$.

This function is shown in Figure 14.

Figure 14. Linear function $h_{\theta}(x) = \theta_0 + \theta_1 x$.



This equation represents a linear function that we learned in high school. θ_1 is the slope of the line, and θ_0 represents the intercept on y axes. In this book we will use this equation to train our model with given datasets and predict the value of y for any value of x. Our challenge is to determine values θ_0 and θ_1 , so that line that has these parameter values gives minimal error.

Loss function represents an error in our predicted value for obtained values for θ_0 and θ_1 . The goal is to minimize this error to get the most accurate values for θ_0 and θ_1 . We will use the mean square error function to calculate the loss. There are three steps in this function:

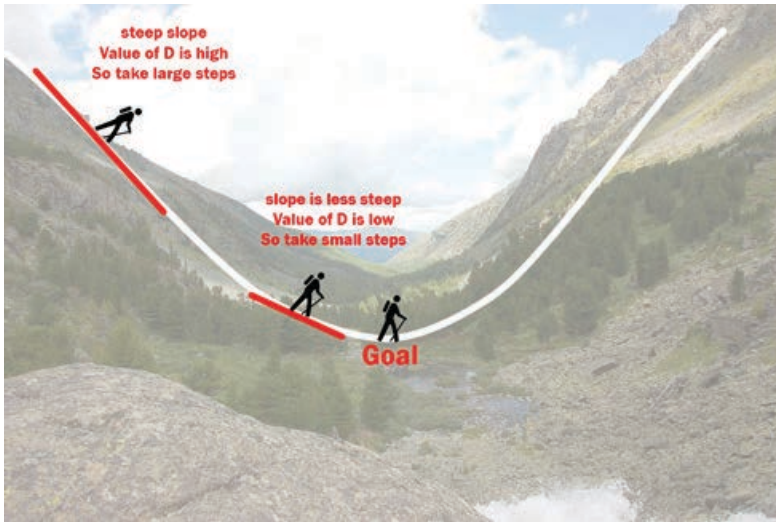
1. The difference between the actual y and the predicted value $h_\theta(x) = \theta_0 + \theta_1 x$, is found for given x.
2. The difference obtained in the first step is squared for every value x.

$$J = \frac{1}{2m} \sum_{i=0}^m \left(y^{(i)} - h_\theta(x^{(i)}) \right)^2$$

If we replace the value of the hypothesis, the cost function becomes

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m \left(y^{(i)} - (\theta_1 x + \theta_0) \right)^2$$

Gradient descent is an iterative optimization algorithm for finding the minimum loss function. As seen in Figure 15, visualize a valley and a traveler trying to reach its bottom who has no sense of direction. When descending a slope, a person takes larger steps when it is steep and shorter steps when it is not. Based on his present location, he calculates his future position and stops when he reaches the valley's bottom, which was his intended destination.

Figure 15. Illustrative representation of the cost function depending on one variable

Source: <https://towardsdatascience.com/linear-regression-using-gradient-descent-97a6c8700931>

Let's try to apply a gradient descent to θ_0 and θ_1 and to approach the bottom step by step. In the beginning, let's initialize those values on value 0. Let α be our learning step. It controls how much value θ_1 changes with each step. For better accuracy, α could be a small value like 0.0001. To update the parameters of our hypothesis, the partial value of the derivative of the loss function is calculated with respect to θ_1 , and it includes the current values θ_0 and θ_1 of x , y , to obtain derivative values $\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ and $\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$.

From the expression $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x + \theta_0))^2$, it turned out to be a

partial derivative $\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} = -\frac{1}{m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x^{(i)} + \theta_0))$

and $\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} = -\frac{1}{m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x^{(i)} + \theta_0)) x^{(i)}$

We simultaneously update the values θ_0 and θ_1 of variables

$$temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \theta_0 + \frac{\alpha}{m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x^{(i)} + \theta_0))$$

$$temp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \theta_1 + \frac{\alpha}{m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x^{(i)} + \theta_0)) x^{(i)}$$

$$\theta_0 := temp0$$

$$\theta_1 := temp1$$

We repeat this process until our cost function becomes very small or ideally 0 (which means 0 error or 100% accuracy). Values θ_0 and θ_1 what we have left will be the optimal values. Now back to our analogy, θ_0 and θ_1 can be considered as the current position of the person. $\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0}$ and $\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1}$ is equivalent to the gradient of the slope and α could be a person velocity.

If we look at the picture, we notice that if the current position of the person in the area where the gradient is negative, i.e. $\frac{\partial J(\theta_1)}{\partial \theta_1} < 0$, a person to reach the bottom of the valley needs to move downwards, i.e., to be shortened in the direction of parameter increase of θ_1 .

On the other hand, if the current position of the person in the area where the gradient is positive, i.e. $\frac{\partial J(\theta_1)}{\partial \theta_1} > 0$, a person to reach the bottom of the valley should move upwards, i.e.. to be shortened in the direction of parameter reduction θ_1 . The change in the current value of the parameters depending on the slope is shown in Figure 16. Now the new value for θ_0 and θ_1 which we calculate by updating the value to be its next position, and the gradients $\alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0}$ and $\alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1}$ will be the size of the steps the person will take. When the slope is steeper, the gradient is larger, and it makes longer steps, and when it is less steep, the gradient is smaller and makes smaller steps. Finally, the person arrives at the bottom of the valley which corresponds to our loss is equal to 0.

The number of repetitions is called number epochs that a person takes until he reaches the bottom. If the step size $\alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0}$ is bigger, fewer number of iterations are required, and if the step size is smaller, a larger number of iterations is required.

We should take care when choosing parameters α . It should move in range between 0.0001 i 1. Too small values give us more precision but makes the algorithm slower. Too big values for α give great steps, a risk that a person can cross from one side of the valley to the other by skipping the bottom of the valley. This illustration is shown in Figure 17.

Figure 16. Updating the current parameter value depending on the gradient sign

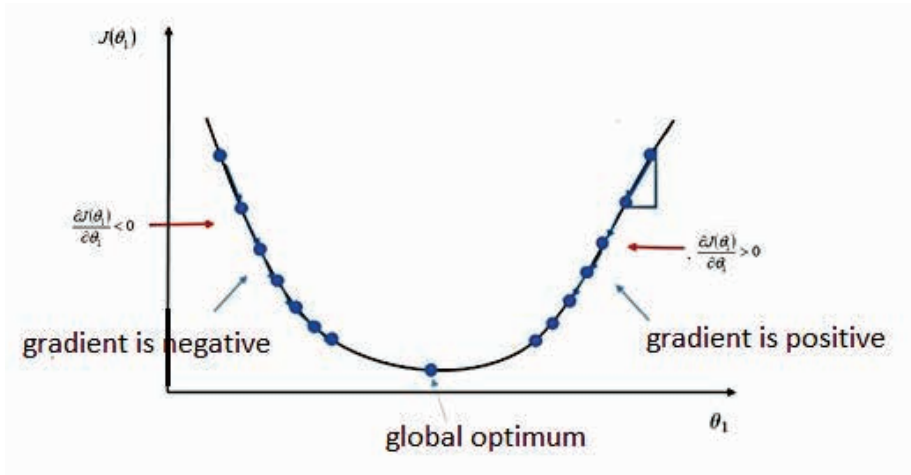
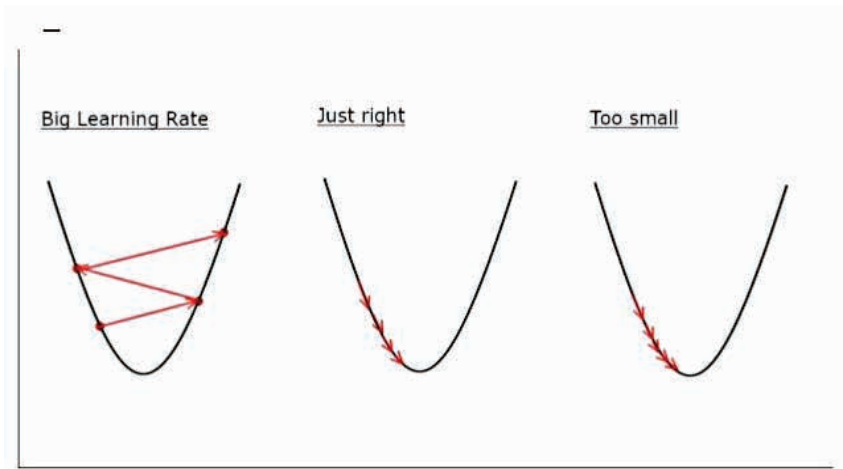


Figure 17. Updating the current parameter value depending on the parameter α



The gradient descent approach for linear regression parameter optimization is shown in Table 5. If the cost function approaches the minimal value and stabilizes, the number of iterations is adequate. The final values are the ideal values. If the cost function gets smaller as it goes through more rounds, the method has been implemented correctly. The parameter can be larger than it needs to be if the cost function grows with the number of iterations.

The cost function is dependent on the number of iterations. In general, the entire training process works best without a predetermined learning rate. Typically, we begin with the learning rate determined using the above-described method. To promote learning, we alter the learning rate during the training session. We adjust the learning rate during the training session to facilitate learning. This can be done in a variety of ways. In <https://towardsdatascience.com/the-subtle-art-of-fixing-and-modifying-learning-rate-f1e22b537303> methods for selecting a learning rate are described. Figure 18 shows how the cost function changes depending on the number of iterations and how the hypothesis approaches the optimal value through iterations for different values θ_0 and θ_1 with the learning rate Type equation here.

Table 5. Gradient descent algorithm for parameter optimization in linear regression

Gradient Descent Algorithm
1. Setting a hypothesis $h_{\theta}(x) = \theta_0 + \theta_1 x$
2. Parameter setting θ_0, θ_1, a number of iterations (eg $\theta_0 = \theta_1 = 0, a = 0.001$, number of iterations = 1000)
3. Calculate the cost function $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x + \theta_0))^2$
4. For i=1 to number of iterations
$temp0 := \theta_0 - a \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \theta_0 + \frac{a}{m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x^{(i)} + \theta_0))$ $temp1 := \theta_1 - a \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \theta_1 + \frac{a}{m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x^{(i)} + \theta_0)) x^{(i)}$ $\theta_0 := temp0$ $\theta_1 := temp1$ $h_{\theta}(x) = \theta_0 + \theta_1 x$

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x + \theta_0))^2$$

$$temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \theta_0 + \frac{\alpha}{m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x^{(i)} + \theta_0))$$

$$temp1 := \theta_0 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \theta_1 + \frac{\alpha}{m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x^{(i)} + \theta_0)) x^{(i)}$$

$$\theta_1 := temp1$$

$$\text{Calculating the cost function } J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (y^{(i)} - (\theta_1 x + \theta_0))^2$$

The number of iterations is sufficient if the cost function reaches the minimum value and becomes constant. Last values for θ_0, θ_1 represent optimal values.

The algorithm is properly implemented if the cost function decreases with the number of iterations.

Figure 18. Cost function depending on the number of iterations

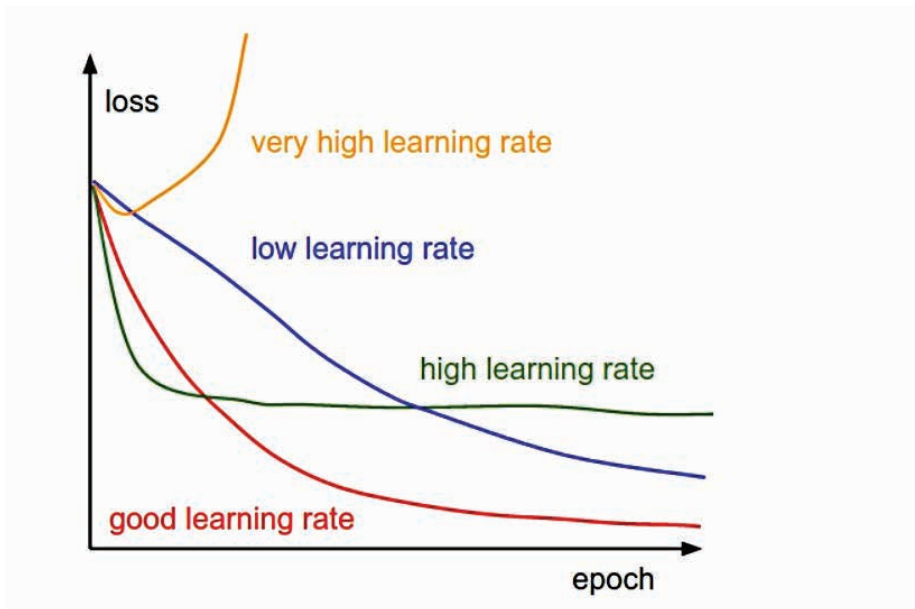
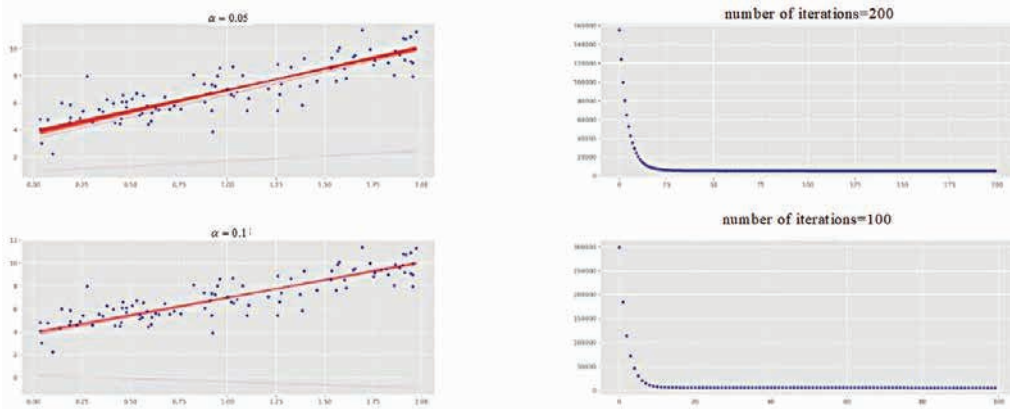


Figure 19. Achieving the optimal value of hypothesis (a) and cost function through iterations (b)

3.4. SCALING ATTRIBUTE VALUES

Let's start with some definitions before we discuss why normalization of attributes is vital in various algorithms such as gradient descent and neural networks. To alter data units, such as converting a temperature from Celsius to Fahrenheit, you would move a vector by first adding or subtracting a constant, followed by multiplying or dividing by a constant.

Typically, vector normalization entails multiplying by the vector norm. It frequently refers to scaling along the minimum and range of vectors so that all elements have values between 0 and 1, which uniformly scales all the attribute values in the data collection. If X represents a set of attribute values x from a dataset, $x_{\min}$ represents the minimum value from the set, and $x_{\max}$ represents the maximum value. Then the normalized value of the attribute is equal to

$$X_{\text{norm}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

This way of normalization is called "min-max normalization".

On the other hand, "standardization" is best when the algorithm used is a neural network. "Standardization" of vectors usually means subtracting the mean value μ and dividing by the standard deviation σ . For example, if a vector contains random values with a Gaussian distribution, the mean value can be subtracted and divided by the standard deviation, giving a "standard normalization".

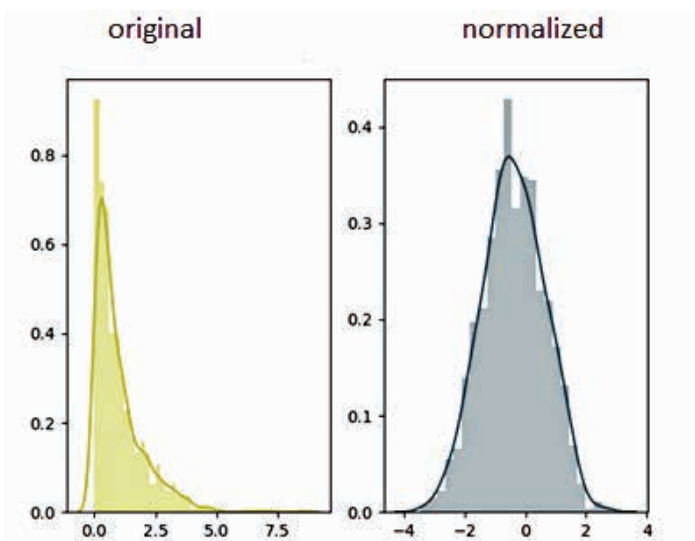
Obtaining a random variable with a mean value of 0 and a standard deviation of 1 as shown in Figure 20.

The normalized value is then calculated as

$$x_{\text{norm}} = \frac{x - \mu}{\sigma}$$

Where μ represents the mean value of the original value of the attribute, and σ^2 represents the standard deviation of the attribute.

Figure 20. Standard normalization of values with Gaussian distribution



Normalization is a good technique used when the distribution of attribute values is not known or when it is known that the distribution is not Gaussian. Normalization is useful when the data have different scales. This may not be strictly true, but the technique is more efficient if the distribution of attributes is Gaussian. Standardization is useful when the data has different scales and the algorithm used is linear regression and logistic regression.

Implementation/use in House prices

Assume we have the same set of data and three hypotheses. We can effectively apply all three hypotheses to all data by using matrix-matrix multiplication.

We want to forecast the price using a data matrix.

$$\text{House sizes}=[2104,1416,1534,852]$$

We have 3 competing hypotheses

$$h_{\theta}=-40+0.25x$$

$$h_{\theta}=200+0.1x$$

$$h_{\theta}=-150+0.4x$$

Due to the fact that our hypothesis only has one variable, we must convert our data (the sizes of the houses) into a 4x2 matrix in order for the matrices to match.

$$\begin{bmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1534 \\ 1 & 852 \end{bmatrix}$$

Then, we can transform this problem into matrix form

$$\begin{bmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1534 \\ 1 & 852 \end{bmatrix} \times \begin{bmatrix} -40 & 200 & -150 \\ 0.25 & 0.1 & 0.4 \end{bmatrix} = \begin{bmatrix} 486 & 410 & 692 \\ 314 & 342 & 416 \\ 344 & 353 & 464 \\ 173 & 285 & 191 \end{bmatrix}$$

3.4.1. Python implementation

Python Linear regression using the Sklearn library

First, we will show how the linear regression from the dataset from https://www.kaggle.com/shivachandel/kc-house-data#kc_house_data.csv is implemented with the help of a Sklearn library. The training test division is in the ratio of 80% -20%. There are several attributes in the dataset such as 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors', 'condition' etc. Among the variable reg. coef_ there is an optimal value for parameters θ_0 , θ_1 . Finally, the mean square value of the error between between the variables y_{pred} and y_{test} is calculated.

```
#importing libraries
import pandas as pd
import numpy as np
from sklearn import linear_model
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
```

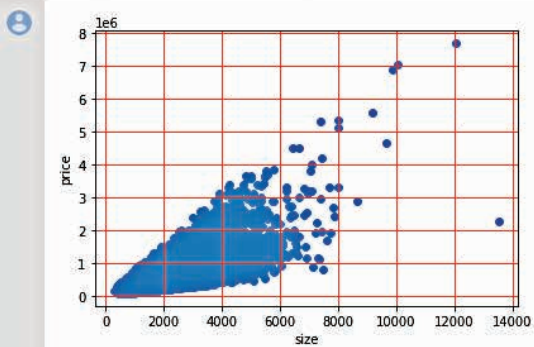
```
[ ] #Dataset Uploading
data= pd.read_csv("kc_house_data.csv")
data.head()
```

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0

```
[ ] data.loc[(data['id']==5631500400,'price')] #checking the price of the hous whose id is 5631500400
```

```
2    180000.0
Name: price, dtype: float64
```

```
#plotting price vs size
plt.scatter(data.sqft_living, data.price)
plt.xlabel('size')
plt.ylabel('price')
plt.grid(color='r', linestyle='-', linewidth=1)
```



```
x_train,x_test,y_train,y_test=train_test_split(x,y,train_size=0.8,random_state=42) #splitting training test dataset in ratio 80%-20% ratio
```

Double-click (or enter) to edit

```
[ ] x.describe() #dataset statistical parameters for every feature
```

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	condition
count	21613.000000	21613.000000	21613.000000	2.161300e+04	21613.000000	21613.000000
mean	3.370842	2.114757	2079.899736	1.510697e+04	1.494309	3.409430
std	0.930062	0.770163	918.440897	4.142051e+04	0.539989	0.650743
min	0.000000	0.000000	290.000000	5.200000e+02	1.000000	1.000000
25%	3.000000	1.750000	1427.000000	5.040000e+03	1.000000	3.000000
50%	3.000000	2.250000	1910.000000	7.618000e+03	1.500000	3.000000
75%	4.000000	2.500000	2550.000000	1.068800e+04	2.000000	4.000000
max	33.000000	8.000000	13540.000000	1.651359e+06	3.500000	5.000000

```

reg = linear_model.LinearRegression() #Linear Regression model using SKlearn Library

[ ] reg.fit(x_train, y_train)

LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

[ ] #Print the coefecients/weights for each feature/column of our model
print(reg.coef_)

[[-6.12394643e+04  1.33180754e+04  3.09478199e+02 -3.94150260e-01
   1.34504874e+04  5.15405801e+04]]

[ ] #print our price predictions on our test data
y_pred = reg.predict(x_test)
print(y_pred)

[[ 539279.5910362 ]
 [ 783838.22862308]
 [1016393.16829233]
 ...
 [ 605489.51146305]
 [ 574397.24931409]
 [ 644790.72179926]]

```

```
[ ] y_test.head()
```

	price
735	365000.0
2830	865000.0
4106	1038000.0
16218	1490000.0
19964	711000.0

```

[ ] # Two different ways to check model performance/accuracy using,
    # mean squared error which tells you how close a regression line is to a set of points.

    # 1. Mean squared error by numpy
    print(np.mean((y_pred-y_test)**2))

    # 2. Mean squared error by sklearn
    # Resource: https://stackoverflow.com/questions/42453875/precision-score-and-accuracy-score-showing-value-error?rq=1
    from sklearn.metrics import mean_squared_error
    print(mean_squared_error(y_test, y_pred))

```

```

price    7.258454e+10
dtype: float64
72584541062.1807

```

3.4.2. Python implementation: Cost function using numpy library

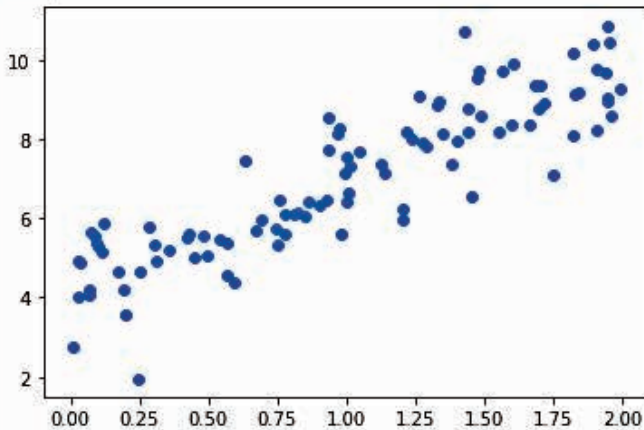
In this example, the algorithm for the cost function is implemented using the numpy library convenient to explain linear regression from scratch.

A dataset of 100 random samples was created.

```
[ ] X = 2 * np.random.rand(100,1)
    y = 4 +3 * X+np.random.randn(100,1)
```

```
import matplotlib.pyplot as plt
plt.scatter(X,y)
```

```
<matplotlib.collections.PathCollection at 0x7f8b75071358>
```



```
[ ] np.shape(y)
```

```
(100, 1)
```

```
[ ] np.shape(theta_best)
```

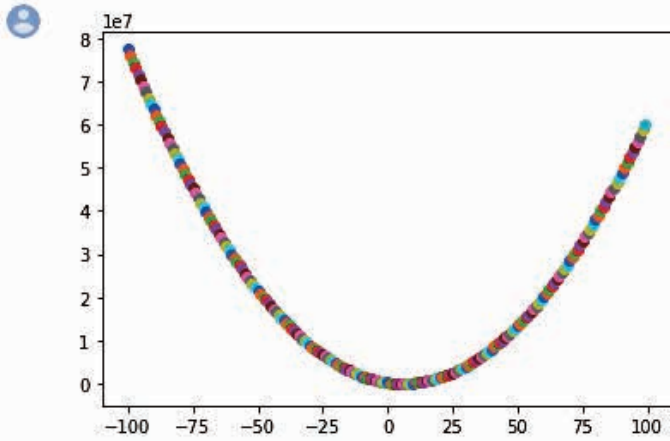
```
(1, 1)
```

```
[ ] theta_best
```

```
array([[5.86862229]])
```

```
[ ] def cal_cost(theta,X,y):  
    #calculates the cost function for given X and y  
    #X is matrix of data, m-number of samples X n-number of features  
    #y is vector of labels, m-number of samples X 1  
    #theta is a vector of parameters (1Xn)  
    m=len(y)  
    h=X.dot(theta)  
    J=(1/2*m)*np.sum(np.square(h-y))  
    return J
```

```
[ ] for theta in range(-100,100):  
    plt.scatter(theta,cal_cost(theta,X,y))
```



From the obtained results, the function reaches the minimum for parameters close to the values for coefficient ≈ 3 and intercept ≈ 4 .

3.4.3. Python implementation: Cost function using numpy library

In this example, the gradient descent algorithm was applied to the same data. To find the optimal parameters θ_0 and θ_1 .

```
# Building the model
theta1 = 0
theta0 = 0

alpha = 0.0001 # The learning Rate
epochs = 10000 # The number of iterations to perform gradient descent

m = float(len(X)) # Number of elements in X
cost = np.ones(epochs)
# Performing Gradient Descent
for i in range(epochs):
    h = theta1*X + theta0 # The current predicted value of Y
    D_theta1 = (-2/m) * sum(X * (y - h)) # Derivative theta1
    D_theta0 = (-2/m) * sum(y - h) # Derivative theta0
    theta1 = theta1 - alpha * D_theta1 # Update theta1
    theta0 = theta0 - alpha * D_theta0 # Update theat0
    cost[i] = (1/X.shape[0]) * 0.5 * sum(np.square(h - y))
print (theta1, theta0)
np.size(cost)
```

[3.43606396] [3.37752512]
10000

From the obtained results, it can be seen that the function reaches the minimum for parameters close to the values for coefficient ≈ 3 and intercept ≈ 4 .

3.4.4. Python implementation: Cost function depending on the number of iterations

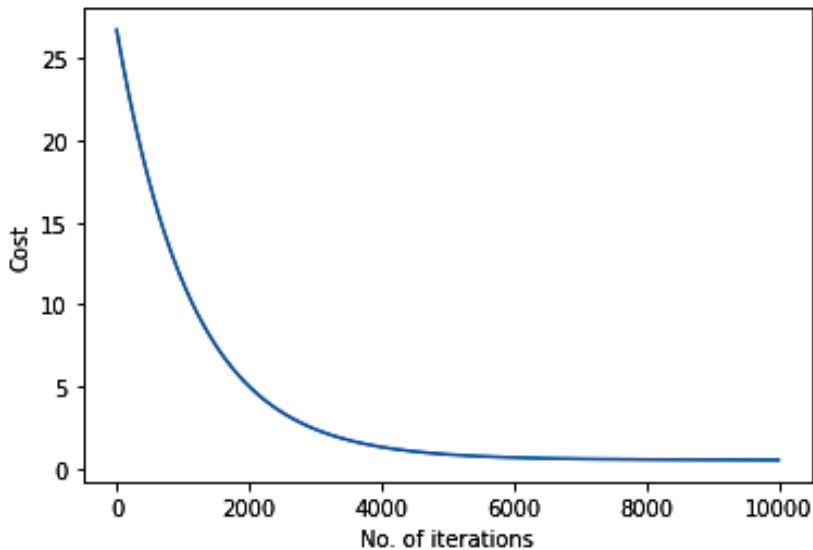
In this example, a cost function is drawn depending on the number of iterations. After 5000 iterations the cost function reaches a minimum value and becomes constant.

Also, it can be concluded that we choose the learning rate correctly because the cost function decreases with the number of iterations.

```
[ ] cost = list(cost)
epochs = [x for x in range(1,10001)]
plt.plot(epochs, cost)
plt.xlabel('No. of iterations')
plt.ylabel('Cost')
```



Text(0, 0.5, 'Cost')

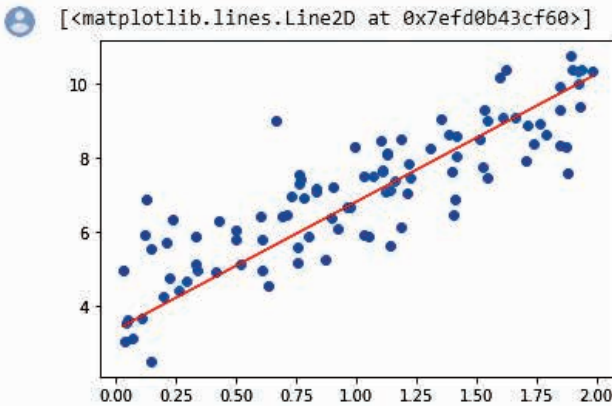


3.4.5. Python implementation: Plotting a hypothesis with optimal parameters

In this example, a hypothesis with optimal parameters is drawn.

```
[ ] # Making predictions
    h = theta1*X + theta0

plt.scatter(X, y)
plt.plot([min(X), max(X)], [min(h), max(h)], color='red') # regression line
```



We can see that the function plotted whose coefficients were calculated using gradient descent algorithm is perfectly fitting the data.

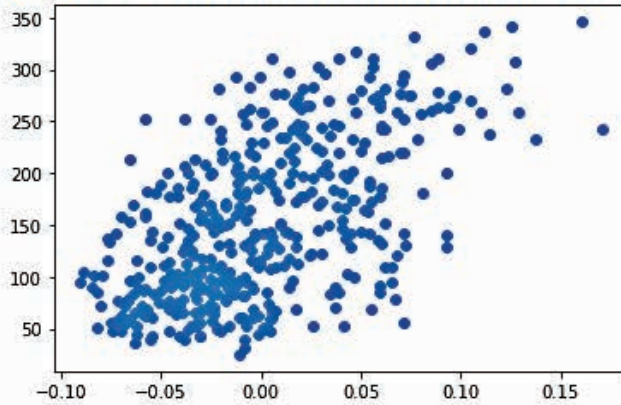
3.4.6. Python implementation: Linear regression using numpy library with attribute normalization

In this example, a gradient descent was applied to diabetes dataset from Sklearn library. From the attributes from this dataset, the third attribute was considered, whose values are ranged from -0.1 to 0.15. These values are too small and the smallest measurement error could affect the value of the output variable y . Therefore, before applying the gradient descent algorithm, standardization was applied to scale the attributes where the mean expected value was shifted to zero and the standard deviation to 1, so that the range of values for the third attribute is from -2 to 4. Finally, a hypothesis with optimal parameters is drawn.


```
[4] diabetes=load_diabetes()
```

```
X=diabetes.data
y=diabetes.target
X=X[:,2]
plt.scatter(X,y)
```

```
<matplotlib.collections.PathCollection at 0x7f83339539d0>
```



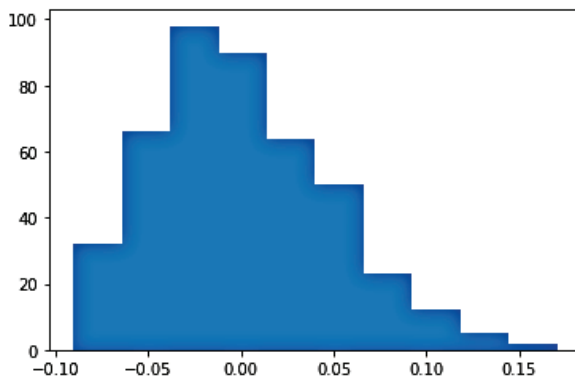
Without normalization



```
plt.hist(X)
```

```
(array([32., 66., 98., 90., 64., 50., 23., 12., 5., 2.]),
 array([-0.0902753, -0.06419224, -0.03810919, -0.01202614, 0.01405691,
        0.04013997, 0.06622302, 0.09230607, 0.11838912, 0.14447217,
        0.17055523]),
```

```
<a list of 10 Patch objects>)
```



```

▶ # Building the model #X to be normalized
theta1 = 0
theta0 = 0

alpha = 0.001 # The learning Rate
epochs = 10000 # The number of iterations to perform gradient descent

m = float(len(X)) # Number of elements in X

# Performing Gradient Descent
for i in range(epochs):
    h = theta1*X + theta0 # The current predicted value of Y
    D_theta1 = (-2/m) * sum(X * (y - h)) # Derivative theta1
    D_theta0 = (-2/m) * sum(y - h) # Derivative theta0
    theta1 = theta1 - alpha * D_theta1 # Update theta1
    theta0 = theta0 - alpha * D_theta0 # Update theta0

print (theta1, theta0)

```

📄 42.00349475168644 152.1334838555429

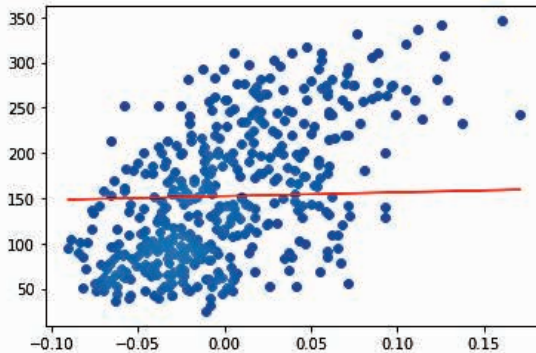
```

▶ # Making predictions
h = theta1*X + theta0

plt.scatter(X, y)
plt.plot([min(X), max(X)], [min(h), max(h)], color='red') # regression line

```

📄 [<matplotlib.lines.Line2D at 0x7f8333132ad0>]



We can see that without normalization, our function doesn't fit well our data due to very small range of feature values.

Let's apply standardization and compute coefficients θ_0 and θ_1 again.

```
[49] X.max()
```

```
0.17055522598066
```

```
[50] X.mean()
```

```
-8.013951493363262e-16
```

```
[51] X.std()
```

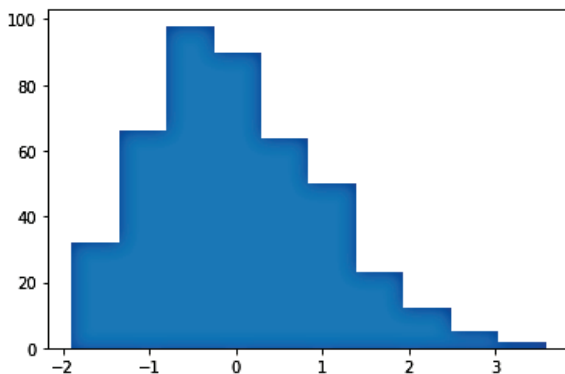
```
0.047565149415453895
```

```
[52] X=(X-X.min())/X.max()
```

```
[53] X=(X-X.mean())/X.std()
```

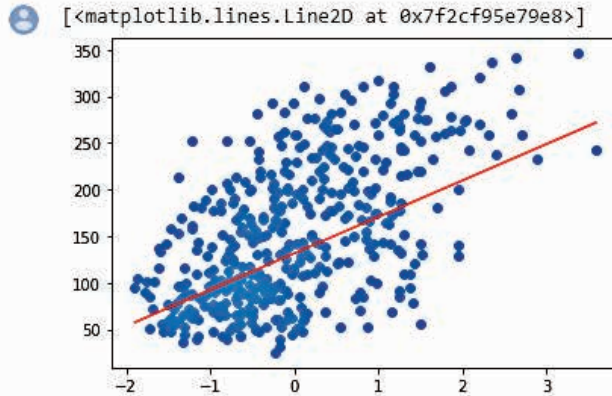
```
▶ plt.hist(X)
```

```
↳ (array([32., 66., 98., 90., 64., 50., 23., 12., 5., 2.]),  
    array([-1.89792941, -1.34956464, -0.80119987, -0.2528351 , 0.29552967,  
          0.84389444, 1.39225921, 1.94062398, 2.48898875, 3.03735352,  
          3.58571828]),  
    <a list of 10 Patch objects>)
```



```
[ ] # Making predictions
h = theta1*X + theta0

plt.scatter(X, y)
plt.plot([min(X), max(X)], [min(h), max(h)], color='red') # regression line
```



```
[ ] theta1
```

```
39.04950695650526
```

```
[ ] theta0
```

```
131.54857393683233
```

```
▶ # Building the model #X to be normalized
theta1 = 0
theta0 = 0

alpha = 0.001 # The learning Rate
epochs = 10000 # The number of iterations to perform gradient descent

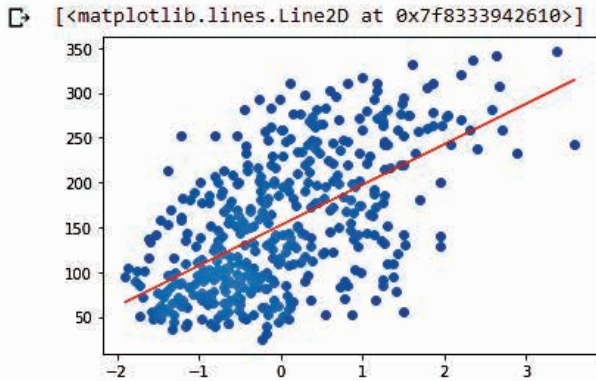
m = float(len(X)) # Number of elements in X

# Performing Gradient Descent
for i in range(epochs):
    h = theta1*X + theta0 # The current predicted value of Y
    D_theta1 = (-2/m) * sum(X * (y - h)) # Derivative theta1
    D_theta0 = (-2/m) * sum(y - h) # Derivative theta0
    theta1 = theta1 - alpha * D_theta1 # Update theta1
    theta0 = theta0 - alpha * D_theta0 # Update theta0

print (theta1, theta0)
```

```
↳ 45.160029929226646 152.13348385554286
```

```
# Making predictions  
h = theta1*X + theta0  
  
plt.scatter(X, y)  
plt.plot([min(X), max(X)], [min(h), max(h)], color='red') # regression line
```



We can see that function with new coefficients after applying standardization is better fitting our data.

CHAPTER 4



LINEAR REGRESSION WITH MULTIPLE VARIABLES

4. LINEAR REGRESSION FOR MODEL WITH MULTIPLE VARIABLES

4.1. MULTIVARIABLE LINEAR REGRESSION MODEL

Linear regression with multiple variables is also known as "multivariate linear regression".

In this chapter, model for multivariate linear regression will be presented where

$x_j^{(i)}$ represents the value of feature j in the i^{th} training example

$x^{(i)}$ represents the input (features) of the i^{th} training example

m represents the number of training examples

n represents the number of features

The multivariable form of the hypothesis function that encompasses these multiple characteristics is as follows:

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

In order to develop an intuition about this function, we can develop an intuition about the parameter θ_0 and θ_1 where x_1 represents the number of square meters and x_2 number of floors, etc. for the house estimation price.

Using the definition of matrix multiplication, our multivariate hypothesis will be

$$h_{\theta}(x) = \begin{bmatrix} \theta_0 & \theta_1 & \dots & \theta_n \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x$$

This expression represents the vectorization of our hypothesis function for a single training sample.

Note: $x_0^{(i)} = 1$ for $i \in 1, 2, \dots, m$.

This allows us to do matrix operations with θ and x . This gives rise to two vectors θ and $x^{(i)}$ which coincide with each other (i.e.. have the same number of elements: $n+1$).

4.2. MULTIVARIABLE LINEAR REGRESSION: GRADIENT DESCENT

The descent gradient for a multivariable model is the same as for a model with one attribute, other variables must be included.

$$\left\{ \begin{aligned} \theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_0^{(i)} \\ \theta_1 &:= \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_1^{(i)} \\ \theta_2 &:= \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_2^{(i)} \\ &\dots \end{aligned} \right\}$$

In other words:

Repeat as long as it converges:

$$\{\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad j = 1, 2, \dots, n$$

4.3. ATTRIBUTE SCALING

We can accelerate the gradient descent if all attributes are set to have the same range of values. That's because θ will fall quickly at both small and big ranges, and when the variables are very uneven, it will oscillate to the best level inefficiently. Modifying the range of our input variables to make them about equal is the best method to avoid this ideal:

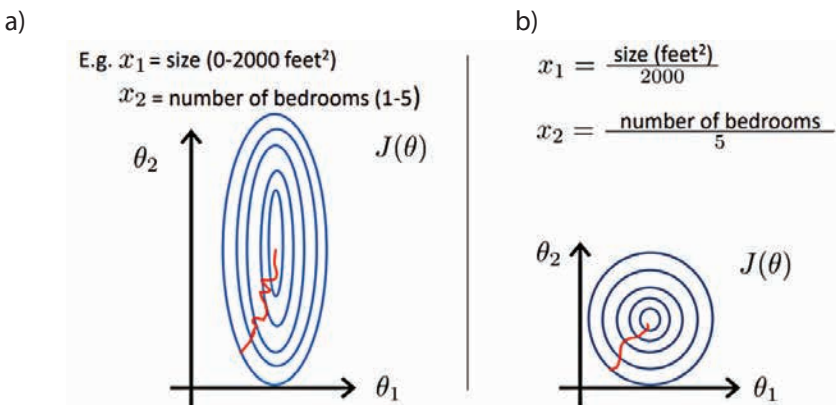
$$-1 \leq x_i \leq 1 \text{ or } -0.5 \leq x_i \leq 0.5$$

These are requirements; we're just trying to speed things up. The goal is to put all input variables in approximately one of these ranges. Two techniques to help with this we already mentioned in the previous chapter. Those are feature scaling (min-max normalization) and mean normalization (standardization). In feature scaling, the input values are divided by the input variable's range (i.e., For example, if it represents a price ranging from 0 to 2000 feet², and x_2 number of rooms from 1 to 5, then the new values for x_1 and x_2 will be $\frac{x_1}{2000}$ and, $\frac{x_2}{5}$ respectively.

Figure 21 shows the cross section of the cost function depending on the two parameters θ_1 and θ_2 .

It can be noticed that in case of normalized attributes the cross section of the cost function is a circle, and in case of non-normalized attributes it is an ellipse. Thus, in the case of normalized attributes, at any point where the current parameter value is within the same value of the cost function, the distance is the same.

Figure 21. Cost functions when attributes are unscaled a) and when they are scaled b) For the choice of learning rate, the same rules apply as in the previous chapter.



4.4. ATTRIBUTES AND POLYNOMIAL REGRESSION

We can improve our characteristics and the shape of the function of our hypothesis in several different ways. We can also change attributes in other ways. We can combine multiple attributes into one. For example, we can combine attributes x_1 and x_2 in a new attribute $x_3 = x_1 x_2$.

We can use polynomial Regression-Our hypothesis does not have to be a linear straight line to pass through data. We can change the function that describes our hypothesis by making the square or cubic or square root of the variable. For example, if our function

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1$$

doesn't fit well our data, then we might create an additional attribute based on existing attributes

to obtain a quadratic function such as

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$$

or a cubic function

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$$

In the cubic version, we made new attributes

$$x_3 = x_1^2, x_4 = x_1^3$$

To make the square root function, we could present the hypothesis as

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 \sqrt{x_1}$$

An important note is that if we select attributes in this way, the normalization of the attributes becomes very important because if it is

$$x_1 = 100, x_1^2 = 10000, \sqrt{x_1} = 10$$

4.5. NORMAL EQUATION

The cost function can be minimized by reducing the gradient. Let's discuss an alternative method, where the minimizing is done explicitly and without the use of an iterative algorithm. In "Normal equation" method, we reduce the cost function by explicitly taking its derivatives with respect to θ_j and equating them with zeros. This allows us to find the optimal one without iteration. Table 6 shows an example of multivariable linear regression when $m=4$.

Table 6. Example for multivariable regression (Example when $m=4$)

x_0	House size (square feet)	Number of bedrooms	Number of floors	House age (number of years)	House price (\$ 1000)
	x_1	x_2	x_3	x_4	
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1534	3	2	30	315
1	852	2	1	36	178

Using this table, data matrices X can be formed and vector output y

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix} \quad y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

Then the matrix θ can be obtained from the formula

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

In this case $X \in \mathbb{R}^{m \times (n+1)}$, $y \in \mathbb{R}^{m \times 1}$ and $\theta \in \mathbb{R}^{n+1}$.

When parameter θ is calculated, there is no need to scale the attributes. In practice, when we have a large amount of data, the gradient descent method is more often used, and rarely the normal equation. Table 7 shows the advantages and disadvantages of both methods.

Table 7. Comparison of Gradient Descent and Normal Equations

Gradient Descent	Normal equation
It is necessary to determine the rate of learning a	It is not necessary to determine the learning rate a
It takes a lot of iteration	It doesn't require much iteration
$O(kn^2)$	$O(n^3)$, it is necessary to calculate the inverse $X^T X$
It is quickly executed for large n	It executes slowly for a lot n

4.5.1. Python implementation: Linear regression using the Sklearn library for Multivariate Linear Regression

In this code, it is shown how linear regression from the dataset is implemented https://www.kaggle.com/shivachandel/kc-house-data#kc_house_data.csv with the help of a Sklearn library. The training-test division is in the ratio of 80% -20%. Attributes such as 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors' and 'condition' were used in the data. The variable `reg.coef_` contains the optimal coefficients for each attribute individually.

The mean square error rate was calculated between the assumed values by the hypothesis and the actual values, i.e. between the variables `y_pred` and `y_test`, respectively.

```

import pandas as pd
import numpy as np
from sklearn import linear_model
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline

```

```

[ ] data= pd.read_csv("kc_house_data.csv")
data.head()

```

	id	date	price	bedrooms	bathrooms	sqft_living
0	7129300520	20141013T000000	221900.0	3	1.00	1180
1	6414100192	20141209T000000	538000.0	3	2.25	2570
2	5631500400	20150225T000000	180000.0	2	1.00	770
3	2487200875	20141209T000000	604000.0	4	3.00	1960
4	1954400510	20150218T000000	510000.0	3	2.00	1680

```

[ ] data.loc[(data['id']==5631500400,'price')]

```

```

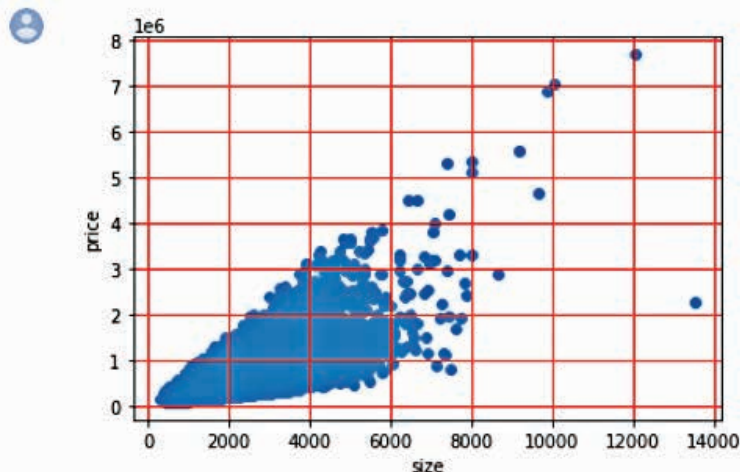
2    180000.0
Name: price, dtype: float64

```

```

[ ] plt.scatter(data.sqft_living, data.price)
plt.xlabel('size')
plt.ylabel('price')
plt.grid(color='r', linestyle='-', linewidth=1)

```



```
[ ] x = data[['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors', 'condition']]
    y = data[['price']] #izlaz
```

```
[ ]
```

```
[ ] x_train,x_test,y_train,y_test=train_test_split(x,y,train_size=0.8,random_state=42)
```

```
▶ x.describe()
```

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	condition
count	21613.000000	21613.000000	21613.000000	2.161300e+04	21613.000000	21613.000000
mean	3.370842	2.114757	2079.899736	1.510697e+04	1.494309	3.409430
std	0.930062	0.770163	918.440897	4.142051e+04	0.539989	0.650743
min	0.000000	0.000000	290.000000	5.200000e+02	1.000000	1.000000
25%	3.000000	1.750000	1427.000000	5.040000e+03	1.000000	3.000000
50%	3.000000	2.250000	1910.000000	7.618000e+03	1.500000	3.000000
75%	4.000000	2.500000	2550.000000	1.068800e+04	2.000000	4.000000
max	33.000000	8.000000	13540.000000	1.651359e+06	3.500000	5.000000

```
[ ] reg = linear_model.LinearRegression()
```

```
[ ] reg.fit(x_train, y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

```
[ ] #Print the coeficients/weights for each feature/column of our model
    print(reg.coef_)
```

```
[[ -6.12394643e+04  1.33180754e+04  3.09478199e+02 -3.94150260e-01
    1.34504874e+04  5.15405801e+04]]
```

```
[ ] #print our price predictions on our test data
    y_pred = reg.predict(x_test)
    print(y_pred)
```

```
[[ 539279.5910362 ]
 [ 783838.22862308]
 [1016393.16829233]
 ...
 [ 605489.51146305]
 [ 574397.24931409]
 [ 644790.72179926]]
```

```
[ ] y_test.head()
```

	price
735	365000.0
2830	865000.0
4106	1038000.0
16218	1490000.0
19964	711000.0

```
# 1. Mean squared numpy
print(np.mean((y_pred-y_test)**2))

# 2. Mean squared error sklearn
# Resource: https://stackoverflow.com/questions/42453875/precision-score-and-accuracy-score-showing-value-error?rq=1
from sklearn.metrics import mean_squared_error
print(mean_squared_error(y_test, y_pred))
```

price	7.258454e+10
dtype:	float64
	72584541062.1807

4.5.2. Python implementation: Linear regression using only numpy library for Multivariate Linear Regression

In this task, linear regression was applied using the gradient descent algorithm. The idea is that considering the size of the house and the number of bedrooms, we analyze and predict the possible price of the house from the dataset https://github.com/navoneel1092283/multivariate_regression/blob/master/data2.txt was used for this purpose.

```
[ ] import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd
data = np.loadtxt('data2.txt', delimiter=',')
```

```
[ ] np.shape(data)
```

```
(47, 3)
```

```
[ ] f = open("data2.txt", "r")
print(f.read())
```

```
2104,3,399900
1600,3,329900
2400,3,369000
1416,2,232000
3000,4,539900
1985,4,299900
1534,3,314900
1427,3,198999
1380,3,212000
1494,3,242500
1940,4,239999
```

Variable X_{train} represents the training dataset. We have $m=47$ samples and $n=2$ attributes.

```
[ ] x_train = data[:, :2] #feature set
    y_train = data[:, 2] #label set
```

```
[ ] transpose(x_train)
```



```
-0.7169663870682891,
-0.9688098626300412,
0.16702965088836602,
2.8164738912678087,
0.20518775324620725,
-0.42823674589395716,
0.3018549458860717,
0.7203221350770638,
-1.0184153956952349,
-1.461049383046193,
-0.1891126377848188,
-1.0145995854594507],
[-0.22609336757768864.
```

```
[ ] np.shape(x_train)
```



```
(47, 2)
```

```
[ ] m=x_train.shape[0]
```

```
[ ] m #Number of samples
```



```
47
```

```
[ ] n=x_train.shape[1]
```

```
[ ] n #number of features
```



```
2
```

In this part, all attributes are normalized so that their mean value is shifted to value 0 and the standard deviation has the value 1.

```
[ ] #Feature normalization

mean = np.ones(n)#mean for every feature
std = np.ones(n)#standard deviation for every feature
for i in range(n):#calculating mean value and standard deviation for every feature vector
    mean[i] = np.mean(transpose(X_train)[i])
    std[i] = np.std(transpose(X_train)[i])
    for j in range(0, m): #normalization in every sample
        X_train[j][i] = (X_train[j][i] - mean[i])/std[i]
```

In the next block, a function is created that calculates the hypothesis for linear regression.

```
def hypothesis(theta, X, n):
    h = np.ones((m,1)) #we want to calculate hypothesis for every sample
    theta = theta.reshape(1,n+1) #one column more to add the bias
    for i in range(0,m):#for every sample
        h[i] = dot(theta, X[i]) #multi
    h = h.reshape(m)
    return h
```

A function for Batch Gradient Descent was created in this block. In Batch Gradient Descent, all training data is considered in one step. We take the average gradient of all training examples, and then use that mean gradient to update our parameters. It is only one step of lowering the gradient in one iteration, that is epoch.

Batch Gradient Descent is great for convex or relatively smooth cost functions. In this case, we are slowly moving directly towards the optimal solution.

```
[ ] def BGD(theta, alpha, epochs, h, X, y, n):
    cost = np.ones(epochs)
    for i in range(0,epochs):
        theta[0] = theta[0] - (alpha*2/m) * sum(h - y)
        for j in range(1,n+1):
            theta[j] = theta[j] - (alpha*2/m) * sum((h-y) * transpose(X)[j])
        h = hypothesis(theta, X, n)
        cost[i] = (1/m) * 0.5 * sum(np.square(h - y))
        theta = theta.reshape(1,n+1)
    return theta, cost
```

In the next block, we define the linear regression function.

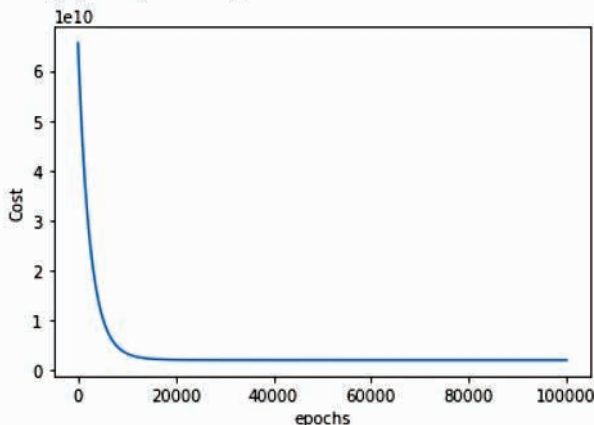
```
def linear_regression(X, y, alpha, epochs):
    n = X.shape[1]
    m=X.shape[0]
    bias = np.ones((m,1)) #adding feature  $x_0$ 
    X = np.concatenate((bias, X), axis = 1)
    # initializing the parameter vector
    theta = np.zeros(n+1) # $R^{n+1}$ 
    # hypothesis calculation
    h = hypothesis(theta, X, n)
    # returning the optimized parameters by Gradient Descent
    theta, cost = BGD(theta,alpha,epochs,h,X,y,n)
    return theta, cost
```

In the next block, we draw a cost function and calculate the optimal solution for the parameter θ .

```
[ ] # calling the principal function with learning_rate = 0.0001 and
    # num_iters = 100000
    theta, cost = linear_regression(X_train, y_train,0.0001, 100000)

[ ] cost = list(cost)
    n_iterations = [x for x in range(1,100001)] #creating the list for plotting
    plt.plot(n_iterations, cost)
    plt.xlabel('epochs')
    plt.ylabel('Cost')
```

Text(0, 0.5, 'Cost')



From the obtained result and the graph of the cost function in relation to the number of epochs, we can see that the algorithm is correctly implemented and that the cost function reaches a minimum after approximately 1000 epochs. The minimum value of the cost function is between 0 and 1.

4.5.3. Python implementation: Normal equation

In this example, an example is made in case we have $n=5$ and $m=4$ using normal equation. To do this, matrix multiplication, inverse matrix calculation and matrix transposition is required mentioned in the Chapter 2.

```
[ ] import numpy as np
    from numpy.linalg import inv

[ ] X = [[1, 2024, 5, 1, 45],
        [1, 1416, 3, 2, 40],
        [1, 1534, 3, 2, 30],
        [1, 852, 2, 1, 36]]

    y = [460, 232, 315, 178]
```

```
def transpose(M):
    N=[]
    for j in range(len(M[0])): #over columns
        U=[]
        for i in range(len(M)): #over rows
            U.append(M[i][j])
        N.append(U)
    return N

[ ] def matrix_multiplication(M, N):
    W = np.zeros((len(M), len(N[0])))
    for i in range(len(M)):
        for j in range(len(N[0])):
            for k in range(len(N)):
                W[i][j] += M[i][k] * N[k][j]
    return W

[ ] # matrix * vector
def dot(M,v):
    u=[]
    for i in range(len(M)):
        val=0.0
        for j in range(len(M[0])):
            val=val+M[i][j]*v[j]
        u.append(val)
    return u
```

```
[ ]
    theta = dot(inv(XXT), XTy)
    print(theta)
```

```
[ ] theta = dot(inv(XXT), XTy)
    print(theta)
```

```
[ ] XT = transpose(X)
    print(XT)
    print(X)
    print(y)
```

```
[ ] [[1, 1, 1, 1], [2024, 1416, 1534, 852], [5, 3, 3, 2], [1, 2, 2, 1], [45, 40, 30, 36]]
    [[1, 2024, 5, 1, 45], [1, 1416, 3, 2, 40], [1, 1534, 3, 2, 30], [1, 852, 2, 1, 36]]
    [460, 232, 315, 178]
```

```
[ ] XTX = matrix_multiplication(XT, X)
    print(XTX)
```

```
[ ] [[4.000000e+00 5.826000e+03 1.300000e+01 6.000000e+00 1.510000e+02]
    [5.826000e+03 9.180692e+06 2.067400e+04 8.776000e+03 2.244120e+05]
    [1.300000e+01 2.067400e+04 4.700000e+01 1.900000e+01 5.070000e+02]
    [6.000000e+00 8.776000e+03 1.900000e+01 1.000000e+01 2.210000e+02]
    [1.510000e+02 2.244120e+05 5.070000e+02 2.210000e+02 5.821000e+03]]
```

```
[ ] XTy = dot(XT, y)
    print(XTy)
```

```
[ ] [1185.0, 1894418.0, 4297.0, 1732.0, 45838.0]
```

```
[ ] theta = dot((XXT / 1), XTy)
    print(theta)
```

```
[ ] [11043871799.0, 17402465714532.0, 39188687870.0, 16635648639.0, 425399695500.0]
```

The result obtained and placed in theta vector represents the optimal value of the parameter θ for all 5 attributes wher $X \in R^{mx(n+1)}$, $y \in R^{mx1}$ and $\theta \in R^{n+1}$.

CHAPTER 5



PYTHON (NUMPY, PANDAS, SKLEARN)

5. PYTHON (NUMPY, PANDAS, SKLEARN)

Book <https://jakevdp.github.io/PythonDataScienceHandbook/> focuses on using Python to perform data science, which prompts the question, what exactly is data science? Considering how common the term has grown, it is surprisingly difficult to define. What science doesn't involve data, after all? According to vocal detractors, the term is either unnecessary or just a buzzword used to spice up resumes and capture the attention of overzealous IT recruiters.

Each chapter in this book focuses on a distinct package or tool that has significantly influenced the development of Python data science. Many data scientists operate in a computational environment provided by IPython and Jupyter.

NumPy: For effective storing and handling of dense data arrays in Python, this module offers the `ndarray`.

Pandas: This package offers the `DataFrame` for effective labeled/columnar data storage and processing in Python.

Matplotlib: This library gives Python users the tools they need to create a wide variety of data visualizations.

Scikit-Learn: This library gives Python users the tools they need to apply machine learning algorithms.

These five packages are only a small part of the PyData universe, which is constantly expanding. However, these five are already crucial to a lot of the work being done in the Python data science field.

CHAPTER 6



LOGISTIC REGRESSION

6. LOGISTIC REGRESSION

6.1. CLASSIFICATION

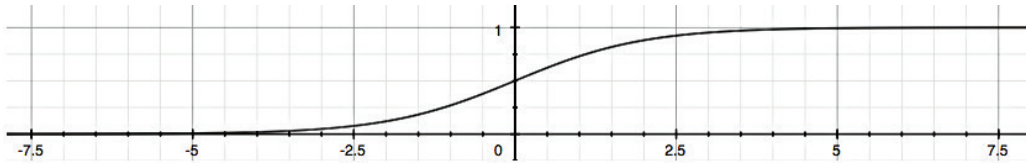
Using linear regression to map all predictions more than 0.5 as 1 and all predictions less than 0.5 as 0, is one way for categorization attempts. However, since classification isn't actually a linear function, this approach isn't the best. The main difference between the regression problem and the classification problem is that the values we are trying to predict now only take a small number of discrete values. We will concentrate on binary classification problem, in which there are only two possible values: 0 and 1 (most of what we say here will be generalized to the case of a higher number of classes). For example, if we try to create a classifier for spam-e-mail. For example, there may be some characteristics of the e-mail parts, and it can be 1 if it is part of the spam, and 0 otherwise. Then, $y \in \{0, 1\}$. Class 0 is also called a negative class, and 1 a positive class, and is sometimes denoted by symbols „-“ i „+“. Sample $(x^{(i)}, y^{(i)})$ is also called a training sample or instance.

6.2. HYPOTHESIS PRESENTATION

We could approach the problem of classification without regard to the fact that it is discrete, and with the help of our old linear regression algorithm. Intuitively, it also makes no sense $h_{\theta}(x)$ to have a smaller value than 0 and greater than 1 when we know that $y \in \{0, 1\}$. To change this, we need to set values for hypothesis to be $0 \leq h_{\theta}(x) \leq 1$.

This is solved by introducing a logistic parameter function with variable $\theta^T x$, i.e. $h_{\theta}(x) = g(\theta^T x)$, where $g(z) = \frac{1}{1 + e^{-z}}$ is sigmoid function shown in Figure 22.

$$h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T x}}$$

Figure 22. Sigmoid function.

In order to obtain our discrete variables 0 and 1 in the classification, we need to make a limit for our hypothesis. It is assumed that it is

$$h_{\theta}(x) \geq 0.5 \Rightarrow y = 1$$

$$h_{\theta}(x) \leq 0.5 \Rightarrow y = 0$$

The way the logistics function $g(z) = \frac{1}{1+e^{-z}}$ behaves is $g(z) \geq 0.5$ when the input variable is $z \geq 0$. $h_{\theta}(x)$ will represent the probability that the output $y=1$, ie. $h_{\theta}(x) = P(y=1|x;\theta)$. So for example if it is $h_{\theta}(x)=0.7$, probability that $y=1$ is 70%. The probability that $y=0$ represents the complement of this probability $h_{\theta}(x)=P(y=1|x;\theta)=1-P(y=0|x;\theta)=0.3$ i.e. 30%.

6.2.1. Python implementation: Logistic regression (breast cancer dataset)

In this code we can see python implementation for logistic regression (binary classification)

```
[1] import numpy as np
import sklearn
from sklearn.datasets import load_breast_cancer
import matplotlib.pyplot as plt
%matplotlib inline

[2] cancer=load_breast_cancer()
X=cancer.data
y=cancer.target
```

```
m=np.shape(X)[0]
n=np.shape(X)[1]
```

```
[34] m
```

```
↳ 569
```

```
[4] n
```

```
↳ 30
```

```
[5] X=(X-X.min())/(X.max()-X.min())
```

```
[6] from sklearn.model_selection import train_test_split
```

```
[7] X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.15)
```

```
[8] np.shape(X_train)
```

```
↳ (483, 30)
```

```
[9] X_train=X_train.T
     X_test=X_test.T
     y_train=y_train.T
     y_test=y_test.T
```

```
[10] def sigmoid(z):
      h=1/(1+np.exp(-z))
      return h
```

```
[11] def initialize_theta(n):
      theta=np.zeros([n,1])
      theta0=0.0
      return theta,theta0
```

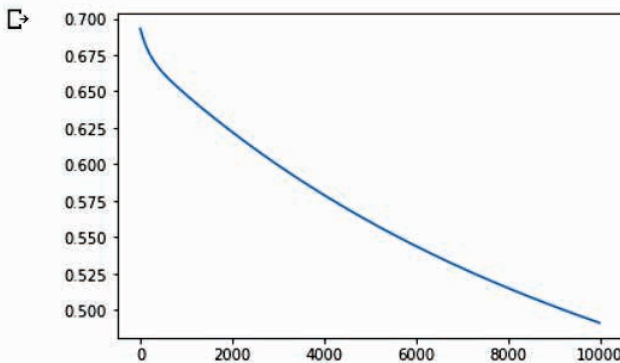
```
[15] def FP(theta,theta0,X_train,y_train):
      z=np.dot(theta.T,X_train)+theta0 #ThetaTranspose*X+X0
      h=sigmoid(z)#hypothesis=g(ThetaTranspose*X+X0), can variate between 0 and 1
      cost=-y_train*np.log(h)-(1-y_train)*np.log(1-h)#cost function for 1 sample
      cost=(np.sum(cost))/X_train.shape[1]
      derivative_weight=(np.dot(X_train,((h-y_train).T)))/X_train.shape[1]
      derivative_bias=np.sum(h-y_train)/X_train.shape[1]
      gradients={'derivative_weight':derivative_weight,'derivative_bias':derivative_bias}
      return cost,gradients
```

```
[16] cost,gradients=FP(theta,theta0,X_train,y_train)
```

```
[17] def update(theta,theta0,X_train,y_train,alpha,epochs):
    cost_list=[]
    cost_list2=[]
    index=[]
    for i in range(epochs):
        cost,gradients=FP(theta,theta0,X_train,y_train)
        cost_list.append(cost)
        theta=theta-alpha*gradients['derivative_weight']
        theta0=theta0-alpha*gradients['derivative_bias']
        if i%10==0:
            cost_list2.append(cost)
            index.append(i)
    parameters={'weight':theta,'derivative_bias':theta0}
    plt.plot(index,cost_list2)
    return parameters,gradients,cost_list
```

```
[19] parameters,gradients,cost_list=update(theta,theta0,X_train,y_train,0.01,10000)

theta=parameters['weight']#optimum theta
```



```
[20] theta0=parameters['derivative_bias']#optimum bias
```

```
[21] def predict(theta,theta0,X_test):
    z=sigmoid(np.dot(theta.T,X_test)+theta0)
    y_prediction=np.zeros((1,X_test.shape[1]))#size is number of samples of the test set, but X_test is transposed
    for i in range(z.shape[1]):
        if z[0,i]<0.5:
            y_prediction[0,i]=0
        else:
            y_prediction[0,i]=1
    return y_prediction
```


6.2.2. Python implementation: Logistic regression (iris dataset)

In this code we can see python implementation for logistic regression (multiclass classification).

```
import sklearn
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.datasets import load_iris
from sklearn.datasets import make_classification
```

```
[2] iris = sklearn.datasets.load_iris()
    x = iris.data[:, :2]
    y = (iris.target != 0) * 1
```

```
[6] def sigmoid(z):
    return 1/(1+np.exp(-z))

def cost(y_pred,target):
    return -np.mean((target*np.log(y_pred)+(1-target)*np.log(1-y_pred)))

def predict(X_test):
    decision = []
    for i in sigmoid(np.dot(X_test, Theta)):
        if i>0.5:
            decision.append(1)
        else:
            decision.append(0)
    return decision

print(X.shape[1]) #shape[0] gives the number of rows and shape[1] gives the number of columns(i.e. features)

np.random.seed(0)
Theta = np.random.uniform(0,1,size=(X.shape[1],1))
Theta0=0.5

for i in range(10):
    Z = np.dot(X, Theta)
    y_output = sigmoid(Z)
    E = cost(y_output,y)
    print("----->",E)
    grad= y_output - y
    grad_weight= np.dot(X.T,grad)/X.shape[0]
    grad_Theta0 = np.average(grad)
    Theta=Theta-.01*grad_weight
    Theta0=Theta0-.01*grad_Theta0
y_test = predict(X_test=[1,1])
print(y_test)
```

```
[6]      grad= y_output - y
      grad_weight= np.dot(X.T,grad)/X.shape[0]
      grad_Theta0 = np.average(grad)
      Theta=Theta-.01*grad_weight
      Theta0=Theta0-.01*grad_Theta0
      y_test = predict(X_test=[1,1])
      print(y_test)
```

```
↳ 2
-----> 1.802979528886028
-----> 1.6595805462919033
-----> 1.51698880116074
-----> 1.3756113975238937
-----> 1.2360489739952454
-----> 1.0991704453420377
-----> 0.9661946790093863
-----> 0.8387521484536271
-----> 0.7188717843839093
-----> 0.6088156696855678
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```


CHAPTER 7



REGULARIZATION

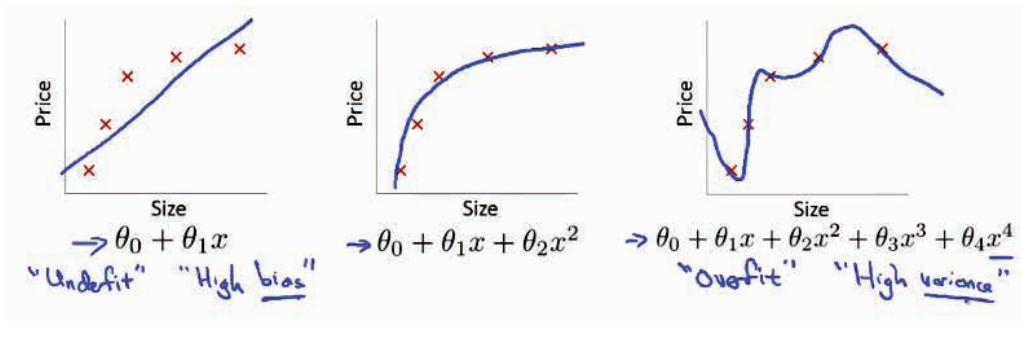
7. REGULARIZATION

We've seen a few algorithms so far, and while they are effective in many applications, they can experience overfitting issues. In this chapter it will be explained what is overfitting, what is regularization, and how can it be beneficial? We are going to use our house pricing example again.

The problem with fitting a linear function to the data is that it results in underfitting, or high bias. When fitting a straight line to the data, we have a strong preconception that there should be a linear fit. This bias is historical and technological. While a straight line can't help but be straight, this is false in this case. If we try to fit a quadratic function, we will realize that it works well. When we fit a 4th order polynomial, the curve fits through all five examples.

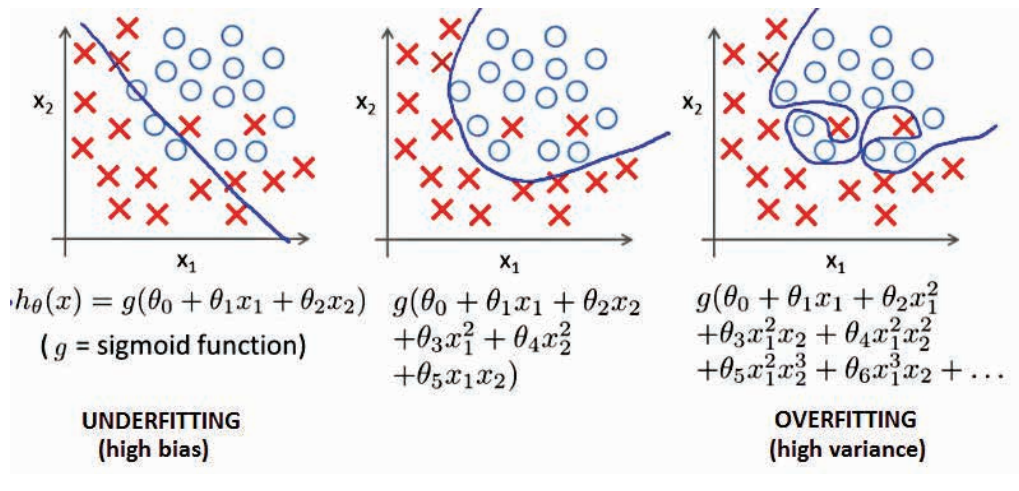
Seems to do a good job fitting the training set. However, although extremely well fitting the data we provided, this model is not that good. This represents overfitting, also known as high variance. In this case we say that algorithm has high variance. Basically, any data can be fit by the hypothesis if a high order polynomial is fitted. We can say that the space of hypothesis is too large.

Figure 23. High bias vs High variance



To sum up, the learnt hypothesis may result in a cost function that is exactly zero if we have too many features. However, this works too hard to suit the training set and is unable to generalize, failing to offer a solution (apply to new examples). The same can be said for logistic regression.

A high order polynomial is overfitting (high variance) and linear function is underfitting (high bias).

Figure 24. High bias vs High variance hypothesis

7.1. ADDRESSING OVERFITTING

Later, we'll discuss how to tell when overfitting and underfitting are taking place. We simply plotted a higher order function before and noticed that it appears "too wavy." One technique to determine is to plot hypotheses, however this method isn't always effective. We often have many features, making it more difficult to decide which features to maintain and which to discard. In this situation, choosing a degree polynomial is only one option. Overfitting can be a problem if you have a lot of features but insufficient data. What should we do about this?

1) Reduce number of features

- Choose manually which features to maintain.
- The discussion of model selection algorithms follows (good for reducing number of features).
- However, some information is lost when we reduce the number of characteristics.
- In the best-case scenario, choose features that reduce data loss, but even then, some data is lost.

2) Regularization

- Maintain all features while reducing parameter magnitudes.
- Performs well when there are several features, each of which helps predict y in a little way.

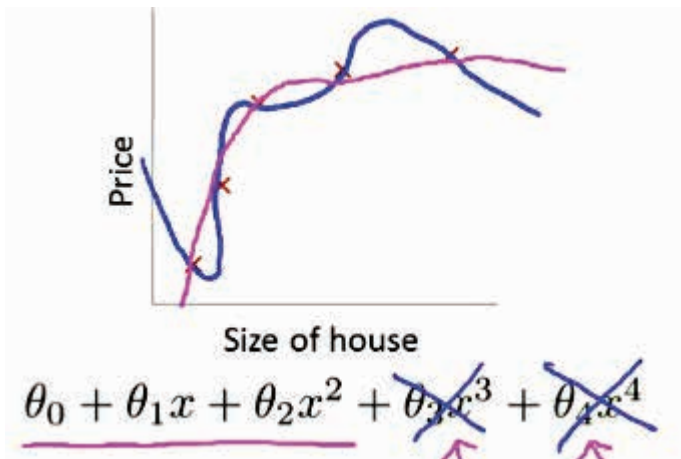
7.2. COST FUNCTION OPTIMIZATION FOR REGULARIZATION

Penalize and reduce some of the parameters θ and make them really small e.g., here θ_3 and θ_4

$$\min_{\theta} \frac{1}{2m} \sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right)^2 + 1000\theta_3^2 + 1000\theta_4^2$$

The blue addition is a change to our cost function that will help penalize θ_3 and θ_4 . So, here we end up with θ_3 and θ_4 being close to zero (because the constants are massive). As a result, we are essentially left with a quadratic function.

Figure 25. Regularization



We penalized two of parameter values in this case. Regularization more broadly is as follows: small parameter values correspond to a simpler theory (you effectively get rid of some of the terms). A more straightforward theory is less likely to overfit. Let's assume the following example: We have 100 features $x_1, x_2, \dots, x_{100}$. In contrast to the polynomial case, we are unsure of the high order terms. The issue is how to choose which people to shrink. Take the cost function, shrink all the parameters with regularization, and then add a term at the end. This regularization term shrinks every parameter. By convention you don't penalize θ_0 , minimization is from θ_1 onwards.

$$J(\theta) = \left[\frac{1}{2m} \sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right)^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

In practice, if you include θ_0 , this has little impact. The regularization parameter that determines how we tradeoff between our two objectives is called λ .

Our two objectives are:

- 1) Attempt to properly fit the training set.
- 2) Want to limit the parameters.

Using the regularized objective in our example, which is the cost function with the regularization factor, results in a much smoother curve that better fits the data and provides a hypothesis. If λ is very large, we end up penalizing ALL the parameters (θ_1, θ_2 etc.) As a result, all the parameters are close to zero. If this occurs, it will be the same as if we eliminated all the terms from the hypothesis. This leads to underfitting in this situation. So, this hypothesis is too biased because of the absence of any parameters (effectively). Thus, parameter λ should be chosen carefully not too big and not too small. There are some automatic ways to select λ but we will not mention them in this chapter.

Regularized linear regression

Prior to this, we examined the Gradient Descent and Normal Equation techniques for linear regression. Cost function for linear regression with regularization is presented as.

$$J(\theta) = \left[\frac{1}{2m} \sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right)^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

$$\frac{\min J(\theta)}{\theta}$$

Previously, the parameters θ_j , where $j=0,1,2,...,n$ would be updated simultaneously, regularly through gradient descent as it is shown below:

$$\{\theta_j := \theta_0 - a \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad j=1,2,...,n$$

Calculus can be used to demonstrate that the following equation is the partial derivative of the regularized $J(\theta)$.

$$J(\theta) = \left[\frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right]$$

$$\frac{\partial}{\partial \theta_j} J(\theta), \quad j=1,2,3,...,n$$

- ◆ The update for θ_j
 - θ_j gets updated to
 - $\theta_j - a * [\text{a big term which also depends on } \theta_j]$
- ◆ So, if you group the θ_j terms together

$$\theta_j := \theta_j \left(1 - a \frac{\lambda}{m} \right) - a \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

The term $\left(1 - a \frac{\lambda}{m} \right)$ is going to be a number less than 1 usually. Usually learning rate is small and m is large. So, this typically evaluates to $(1 - \text{a small number})$ and the term is often around 0.99 to 0.95. This means that θ_j gets multiplied by 0.99.

Regularization with the normal equation

Normal equation is the other linear regression model. We can minimize $J(\theta)$ using normal equation. To use regularization, we add a term $(+\lambda [n+1 \times n+1])$ to the equation. $\theta [n+1 \times n+1]$ is identity matrix.

$$\theta = \left(X^T X + \lambda \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1 \end{bmatrix} \right)^{-1} X^T y$$

Let's see how regularization for logistic regression looks like. We saw earlier that logistic regression can be prone to overfitting with lots of features.

Logistic regression cost function is as follows.

$$J(\theta) = - \left[\frac{1}{m} \sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(x^{(i)})) \right]$$

To modify it we have to add an extra term

$$+ \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

This has the effect of penalizing the parameters θ_1, θ_2 up to θ_n . We can obtain what seems to be a better fitting lower order hypothesis when using means, just like with linear regression. How do we put this into action? The initial results of the logistic regression with gradient descent looked like this.

$$\theta_j := \theta_j - a \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

$$j=0,1,2,3,\dots,n$$

Once more, all that is required to change the algorithm is to change the update rule for θ_j and onwards. Identical to linear regression in terms of appearance, but obviously significantly different in terms of the hypothesis

$$\theta_j := \theta_j \left(1 - a \frac{\lambda}{m} \right) - a \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

CHAPTER 8



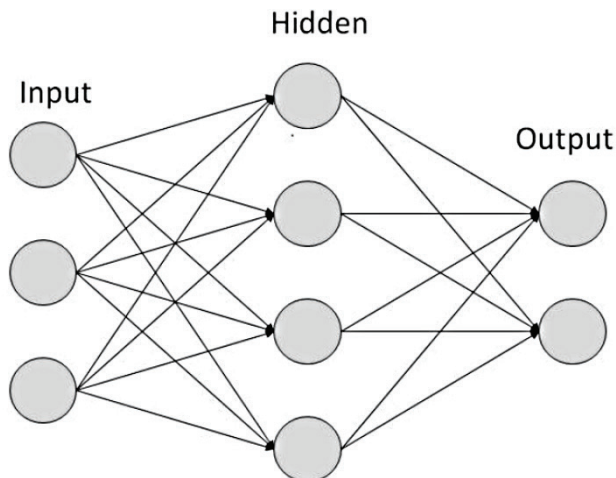
NEURAL NETWORKS- REPRESENTATION

8. NEURAL NETWORKS - REPRESENTATION

In this chapter it will be described what is a Neural Network (NN). Basic unit is a *single neuron* and is basically a linear regression without applying activation unit (perceptron). Thus, a single neuron will compute weighted sum of input $W.T*X$ and then we can set a threshold to predict output in a perceptron. If the weighted sum of the inputs is above the threshold, the perceptron activates; otherwise, it does not forecast. Perceptron input can be a real value input or boolean type value.

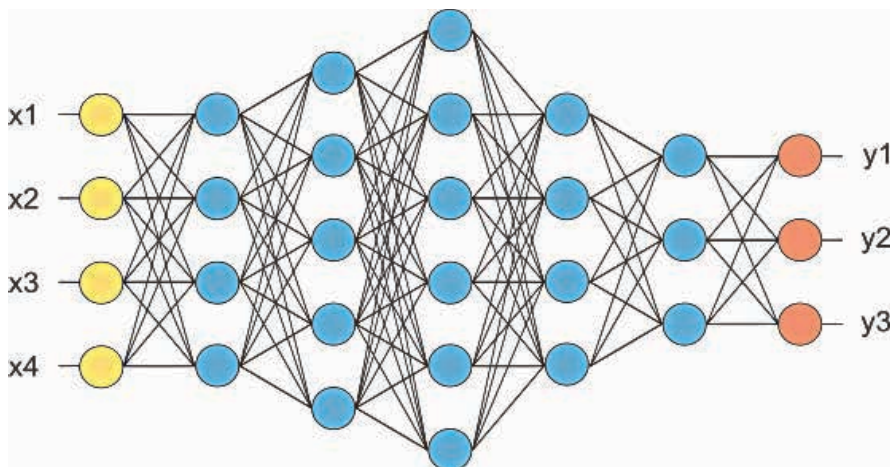
When $w*x+b=0$ the perceptron output is 0. A perceptron's output is only binary, therefore if we try to slightly change the weight or bias, the output may flip. We need a system that can subtly change the output in response to slight weight and bias changes. If we tweak the perceptron with a sigmoid function, we can perhaps change the outcome. For example, if output in perceptron is 0, if you slightly change weight and bias, the output might become 1, but actual output is 0.7. In case of sigmoid, output1 is 0, slight change in weight and bias, then output is 0.7. Applying the sigmoid activation function will make a single neuron operate as Logistic Regression. By examining the sigmoid function graph, we may comprehend the distinction between the perceptron and sigmoid function.

Simple NN graph:



The rectified linear unit, or RELU, is now the most popular activation function for accelerating the training of deep neural networks. Using hidden layers, deep learning excels at automatically anticipating connections between inputs.

Supervised learning with neural networks

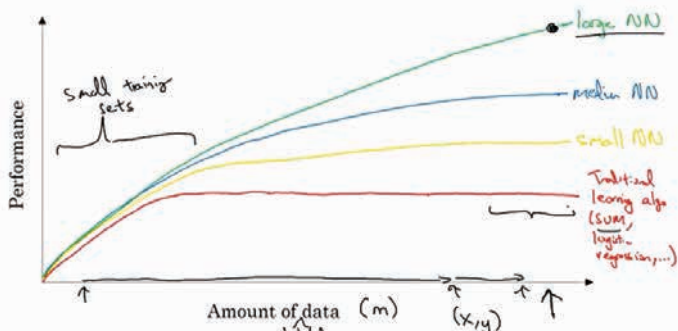


There are various neural network types for supervised learning, including: Convolutional Neural Networks (CNN) - appropriate for computer vision.

Recurrent Neural Networks (RNN) - appropriate for Speech recognition or NLP

Standard NN - appropriate for Structured data. Structured data is like the databases and tables. Unstructured data are images, video, audio, and text. Structured data are more expensive because companies rely on prediction on its big data.

Scale drives deep learning progress



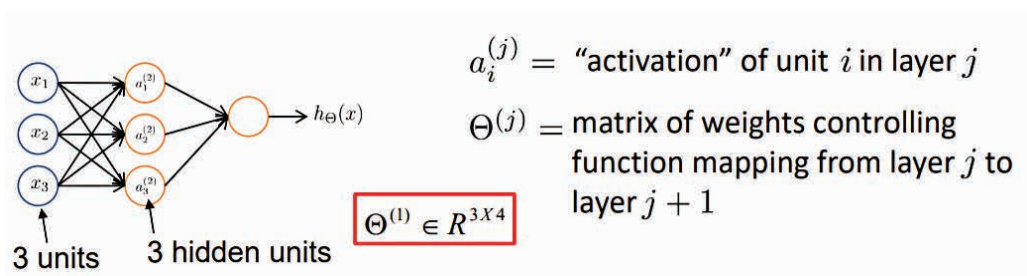
8.1. MODEL REPRESENTATION

Let's analyse how neural networks will be used to describe a hypothesis function. At their most fundamental level, neurons are simply computing units that accept inputs from dendrites in the form of electrical inputs (known as "spikes") and then guide those inputs to outputs (axons). Our dendrites function is equivalent to the input features $x_1, x_2, \dots, x_n$ output. In our model, and our hypothesis function as a function of input features produces the output. Our input node in this model is occasionally referred to as the "bias unit", and the output is the result of our hypothesis function. In this model our input node x_0 is sometimes called "bias unit" and it is always equal to 1. We employ the identical logistic function for classification and neural networks. $\frac{1}{1 + e^{-\theta^T x}}$, Sometimes we refer to it as a sigmoid (logistic) **activation** function. Our "theta" parameters in this case are also referred to as "weights." Visually, a crude illustration appears to be:

$$\begin{aligned} a_1^{(2)} &= g\left(\theta_{10}^{(1)}x_0 + \theta_{11}^{(1)}x_1 + \theta_{12}^{(1)}x_2 + \theta_{13}^{(1)}x_3\right) \\ a_2^{(2)} &= g\left(\theta_{20}^{(1)}x_0 + \theta_{21}^{(1)}x_1 + \theta_{22}^{(1)}x_2 + \theta_{23}^{(1)}x_3\right) \\ a_3^{(2)} &= g\left(\theta_{30}^{(1)}x_0 + \theta_{31}^{(1)}x_1 + \theta_{32}^{(1)}x_2 + \theta_{33}^{(1)}x_3\right) \\ h_\theta(x) &= a_1^{(3)} = g\left(\theta_{10}^2a_0^2 + \theta_{11}^2a_1^2 + \theta_{12}^2a_2^2 + \theta_{13}^2a_3^2\right) \end{aligned}$$

This indicates that our activation nodes are computed using 3x4 matrix of parameters. We apply each row of the parameters matrix to our inputs to get the value for one activation node. The logistic function, which we applied to the total of the input values of our activation nodes after multiplying them by another parameter matrix, produces our hypothesis as a result. $\Theta^{(2)}$ holding the weights for the nodes in our second layer. Each layer receives a unique weights matrix, $\Theta^{(j)}$. These weight matrices' dimensions are established as follows: If network has s_j units in layer j and s_{j+1} units layer $j+1$, then $\Theta^{(j)}$ will be dimensions $s_{j+1} \times (s_j + 1)$.

+1 comes from the addition in $\Theta^{(j)}$ of the 'bias nodes', x_0 and $\theta_0^{(j)}$. To put it another way, the bias nodes will not be present in the output nodes, but they will be present in the inputs. The representation of our model is shown in Figure 26.

Figure 26. Neural network model representation with 3 input units, 3 hidden units and one output unit

8.1.1. Python implementation: Feed forward propagation

```
[ ] def forprop(yi, Xi, Theta1, b1, Theta2, b2):
    z2 = dot(Theta1, Xi) #z2=Theta1*x1 #output of the layer 1
    z2 = add(z2, b1) #adding X0 from the first layer
    a2 = sigmoid(z2) # activation at layer 1
    z3 = dot(Theta2, a2)
    z3 = add(z3, b2) #adding a0 in the second layer
    a3 = sigmoid(z3) # activation at layer 2, and its derivative
    # Compute the objective function
    err = (a3[0] - yi)
    J = err * err;
    return [J, err, Xi, a2, a3]
```

```
[ ] from random import seed
    from random import random
```

```
[ ] Theta1 = [[random(), random()],
               [random(), random()],
               [random(), random()]]
    b1 = [0, 0, 0];
    Theta2 = [[random(), random(), random()]]
    b2 = [0];
```

```
[ ] Theta1
```

```
[[0.5564543226524334, 0.6422943629324456],  
 [0.1859062658947177, 0.9925434121760651],  
 [0.8599465287952899, 0.12088995980580641]]
```

```
[ ] type(Theta1)
```

```
list
```

```
[ ] type(X)
```

```
list
```

```
[ ] len(Theta1)
```

```
3
```

```
[ ] len(Theta1[0])
```

```
2
```

```
[ ] len(X)
```

```
11
```

```
[ ] len(X[0])
```

```
2
```

8.1.2. Backward Propagation

Like the method of gradient descent, we employed in the last note, this strategy is applied for minimizing the loss function. The approach we used here, which is like Gradient Descent, consists of two steps:

1. Do the partial derivative calculation for $J(\theta)$.
2. Update each element of weighting matrix θ .

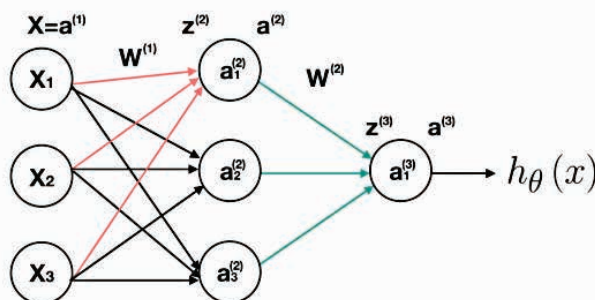
$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

$$j=0, 1, 2, 3, \dots, n$$

Idea about Backward propagation is explained by professor Ng in [5] in an intuitive way. Let's use the partial derivative to illustrate the process to compute the cost function $J(\theta)$. Let's begin with a different illustration and assume that the weighting matrices θ (or designated W) have been initialized. The weighting matrices will then be updated when we minimize the cost function $J(\theta)$. Note — We can use random method for initializing the weighting matrices θ if we have no prior knowledge of the problem.

$$\Theta^{(1)} = W^{(1)} = \begin{bmatrix} \Theta_{11}^{(1)} & \Theta_{21}^{(1)} & \Theta_{31}^{(1)} \\ \Theta_{12}^{(1)} & \Theta_{22}^{(1)} & \Theta_{32}^{(1)} \\ \Theta_{13}^{(1)} & \Theta_{23}^{(1)} & \Theta_{33}^{(1)} \end{bmatrix} = \begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \\ w_{13}^{(1)} & w_{23}^{(1)} & w_{33}^{(1)} \end{bmatrix}$$

$$\Theta^{(2)} = W^{(2)} = \begin{bmatrix} \Theta_{11}^{(2)} \\ \Theta_{12}^{(2)} \\ \Theta_{13}^{(2)} \end{bmatrix} = \begin{bmatrix} w_{11}^{(2)} \\ w_{12}^{(2)} \\ w_{13}^{(2)} \end{bmatrix}$$



Following a calculation in calculus, we can obtain the partial derivative of $J(\theta)$.

$$\frac{\partial J(\Theta)}{\partial \Theta^{(2)}} = \delta^{(3)} * a^{(2)}$$

$$\frac{\partial J(\Theta)}{\partial \Theta^{(1)}} = \delta^{(2)} * a^{(1)}$$

$$\delta^{(3)} = (a^{(3)} - y) * g'(z^{(3)})$$

$$\delta^{(2)} = \delta^{(3)} (W^{(2)})^T g'(z^{(2)})$$

Hint

$$\delta^{(3)} = (a^{(3)} - y) * g'(z^{(3)})$$

$$\begin{bmatrix} \delta_1^{(3)} \\ \delta_2^{(3)} \\ \delta_3^{(3)} \end{bmatrix} = \begin{bmatrix} (a_1^{(3)} - y_1) \\ (a_2^{(3)} - y_2) \\ (a_3^{(3)} - y_3) \end{bmatrix} * \begin{bmatrix} g'(z_1^{(3)}) \\ g'(z_2^{(3)}) \\ g'(z_3^{(3)}) \end{bmatrix} = \begin{bmatrix} (a_1^{(3)} - y_1) * g'(z_1^{(3)}) \\ (a_2^{(3)} - y_2) * g'(z_2^{(3)}) \\ (a_3^{(3)} - y_3) * g'(z_3^{(3)}) \end{bmatrix}$$

Hint

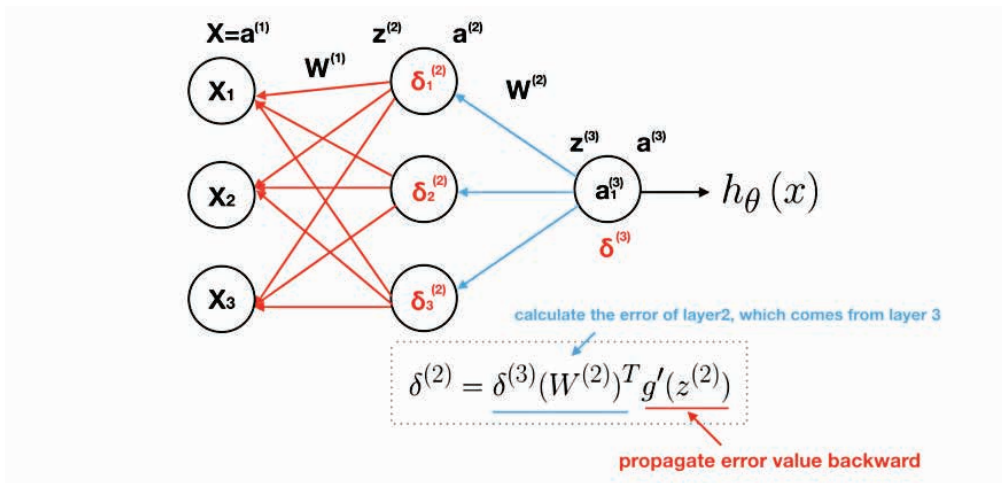
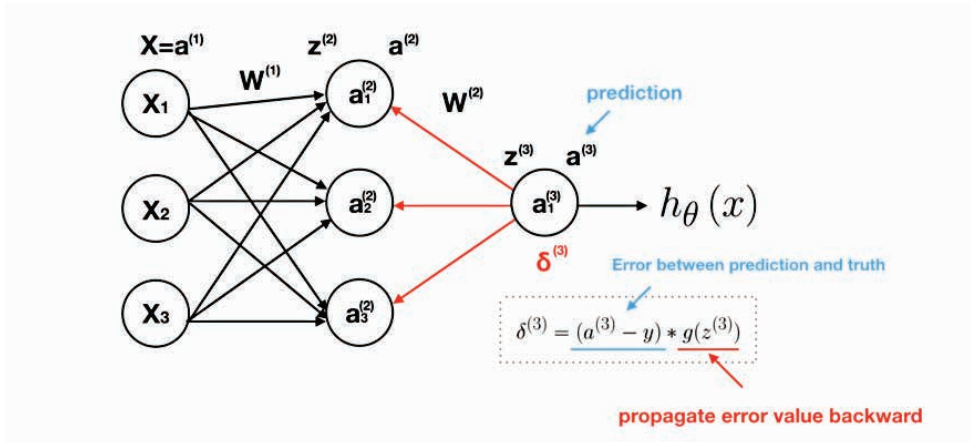
$$\delta^{(2)} = \delta^{(3)} (W^{(2)})^T g'(z^{(2)})$$

$$= \begin{pmatrix} \delta_1^{(3)} \\ \delta_2^{(3)} \\ \delta_3^{(3)} \end{pmatrix} \begin{bmatrix} w_1^{(2)} & w_2^{(2)} & w_3^{(2)} \end{bmatrix} * \begin{bmatrix} g'(z_1^{(2)}) \\ g'(z_2^{(2)}) \\ g'(z_3^{(2)}) \end{bmatrix}$$

$$= \begin{bmatrix} \delta_1^{(3)} * w_1^{(2)} & \delta_1^{(3)} * w_2^{(2)} & \delta_1^{(3)} * w_3^{(2)} \\ \delta_2^{(3)} * w_1^{(2)} & \delta_2^{(3)} * w_2^{(2)} & \delta_2^{(3)} * w_3^{(2)} \\ \delta_3^{(3)} * w_1^{(2)} & \delta_3^{(3)} * w_2^{(2)} & \delta_3^{(3)} * w_3^{(2)} \end{bmatrix} * \begin{bmatrix} g'(z_1^{(2)}) \\ g'(z_2^{(2)}) \\ g'(z_3^{(2)}) \end{bmatrix}$$

Let's explore the significance of δ

- δ^3 is the layer 3 output hidden layer in this illustration.
- δ^2 is the layer 2 output hidden layer in this illustration.
- δ^1 is the layer 1 output hidden layer in this illustration.



With the result of partial derivative of $J(\theta)$, you can update the weighting matrix θ .

Update each element of weighting matrix θ .

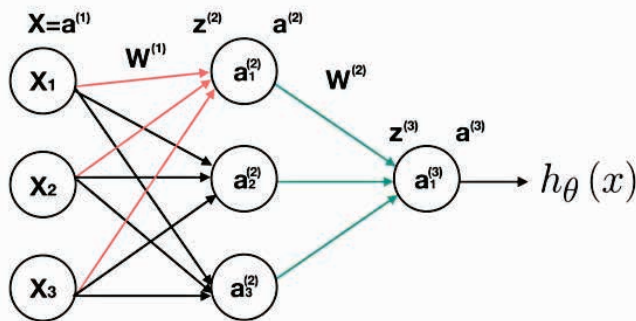
$$\theta_j := \theta_j - a \frac{\partial J(\theta)}{\partial \theta_j}$$

$$j=0,1,2,3,\dots,n$$

Additional information on partial derivative of $J(\theta)$

Let's provide some hints on the derivations of the aforementioned formulation in this section.

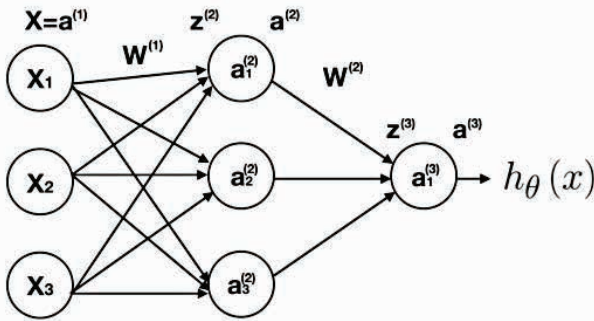
$$\Theta^{(1)} = W^{(1)} = \begin{bmatrix} \Theta_{11}^{(1)} & \Theta_{21}^{(1)} & \Theta_{31}^{(1)} \\ \Theta_{12}^{(1)} & \Theta_{22}^{(1)} & \Theta_{32}^{(1)} \\ \Theta_{13}^{(1)} & \Theta_{23}^{(1)} & \Theta_{33}^{(1)} \end{bmatrix} \quad \Theta^{(2)} = W^{(2)} = \begin{bmatrix} \Theta_{11}^{(2)} \\ \Theta_{12}^{(2)} \\ \Theta_{13}^{(2)} \end{bmatrix}$$



For convenience's sake, we use the same example above and suppose there is only one data. Hence, the loss function $J(\theta)$ is simpler than the general form.

$$J(\Theta) = - \left[y \log(h_{\Theta}(x)) + (1-y) \log(1-h_{\Theta}(x)) \right]$$

In order to do the backward propagation, we need to do the forward propagation first.



Forward Propagation

$$(\Theta^{(1)})^T X = z^{(2)}$$

$$a^{(2)} = g(z^{(2)})$$

$$(\Theta^{(2)})^T a^{(2)} = z^{(3)}$$

$$a^{(3)} = g(z^{(3)})$$

After that, we can perform $J(\theta)$ derivative. Here, we display two items from $W^{(1)}$ and $W^{(2)}$ partial derivatives. You can use hints to solve it if you're interested.

Hint

$$a^{(3)} = g(z^{(3)})$$

Hint

$$\Theta^{(2)} a^{(2)} = z^{(3)}$$

$$\frac{\partial J(\Theta)}{\partial \Theta_{11}^{(2)}} = \frac{\partial J(\Theta)}{\partial a_1^{(3)}} * \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} * \frac{\partial z_1^{(3)}}{\partial \Theta_{11}^{(2)}} = \delta_1^{(3)} * a_1^{(2)}$$

Hint

$$\begin{aligned} \frac{\partial J(\Theta)}{\partial a_1^{(3)}} &= -[y * \frac{1}{g(z)} * g'(z) + (1-y) * \frac{1}{1-g(z)} * (-g'(z))] \\ &= -[y * (1-g(z)) + (y-1) * g(z)] = g(z) - y \end{aligned}$$

Hint-The Property of Sigmoid Function

$$g'(z) = g(z) * (1-g(z))$$

$$\frac{\partial J(\Theta)}{\Theta_{11}^{(1)}} = \frac{\partial J(\Theta)}{\partial a_1^{(3)}} * \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} * \frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} * \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} * \frac{\partial z_1^{(2)}}{\Theta_{11}^{(1)}} = \delta_1^{(2)} * a_1^{(1)}$$

Diagram illustrating the backpropagation of error gradients through the network layers. The main equation shows the chain rule for the partial derivative of the cost function $J(\Theta)$ with respect to the weight $\Theta_{11}^{(1)}$. The terms are color-coded and connected by arrows to their corresponding variables in the network structure above:

- $\delta_1^{(3)}$ (red dashed box) is connected to $\frac{\partial J(\Theta)}{\partial a_1^{(3)}}$ (orange box).
- $g'(z_1^{(2)})$ (blue dashed box) is connected to $\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}}$ (blue box).
- $a_1^{(1)}$ or x_1 (red dashed box) is connected to $\frac{\partial z_1^{(2)}}{\Theta_{11}^{(1)}}$ (red box).
- $g'(z_1^{(2)})$ (blue dashed box) is also connected to $\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}$ (green box).

Below the main equation, three hints are provided in dashed boxes:

- Hint**
 $(\Theta^{(1)})^T X = z^{(2)}$
- Hint**
 $a^{(2)} = g(z^{(2)})$
- Hint**
 $\Theta^{(2)} a^{(2)} = z^{(3)}$

Finally, if you follow the technique, you may demonstrate the partial derivative formula for $J(\theta)$.

8.1.3. Pyhon implementation: Backward propagation

```

1 from math import exp
  from random import seed
  from random import random

  # Initialize a network #w
  def initialize_network(n_inputs, n_hidden, n_outputs):
      network = list()
      hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
      network.append(hidden_layer) #for the hidden layer we create n_hidden neurons and each neuron
      #in the hidden layer has n_inputs + 1 weights, one for each input column in a dataset and an additional one for the bias.
      output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
      network.append(output_layer) #output layer that connects to the hidden layer has n_outputs neurons, each with n_hidden + 1 weights.
      return network

```

```

# Sigmoid neuron activation
def sigmoid(z):
    return 1.0 / (1.0 + exp(-z))

```

```

[ ] def back_propagate_error(network,y):
    for i in reversed(range(len(network))):
        layer=network[i]
        errors=list()
        if i!=len(network)-1: #hidden layers
            for j in range(len(layer)):
                error=0.0
                for neuron in network[i+1]:
                    error=error+neuron['weights'][j]*neuron['delta']
                errors.append(error)
        else:
            for j in range(len(layer)): #the last layer
                neuron=layer[j]
                errors.append(y[j]-neuron['output'])
            for j in range(len(layer)):
                neuron=layer[j]
                neuron['delta']=errors[j]*sigmoid_derivative(neuron['output'])

    #error=(thetajk*errorj)*g'(output)

```

```

[ ] def update_weights(network, sample, alpha):
    for i in range(len(network)):
        inputs = sample[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] =neuron['weights'][j]+alpha * neuron['delta'] * inputs[j]
            neuron['weights'][-1] =neuron['weights'][-1]+ alpha * neuron['delta']

```

```
[ ] # Train a network for a fixed number of epochs
def train_network(network, train, alpha, epochs, n_outputs):
    for epoch in range(epochs):
        sum_error = 0
        for sample in train:
            outputs = forward_propagate(network, sample)
            y = [0 for i in range(n_outputs)]
            y[sample[-1]] = 1
            sum_error += sum([(y[i]-outputs[i])**2 for i in range(len(y))])
            back_propagate_error(network, y)
            update_weights(network, sample, alpha)
        print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, alpha, sum_error))

# Test training backprop algorithm
seed(1)

# Test making predictions with the network
dataset = [[2.7,2.5,0],
            [1.4,2.3,0],
            [3.3,4.4,0],
            [1.3,1.8,0],
            [3.0,3.0,0],
            [7.6,2.7,1],
            [5.3,2.0,1],
            [6.9,1.7,1],
            [8.6,1.0,1],
            [7.6,3.5,1]]
from random import random
from random import seed
n_inputs = len(dataset[0]) - 1
n_outputs = len(set([sample[-1] for sample in dataset]))#number of different outputs
network = initialize_network(n_inputs, 2, n_outputs)
train_network(network, dataset, 0.01, 20000, n_outputs)
for layer in network:
    print(layer)
```

```
[ ] test_data=[[1.4,2.3,0],
                [5.3,2.0,1],
                [5.3,3.0,1],
                [5.3,5.0,1]]
```

```
[ ] output=forward_propagate(network,[1.4,2.3,0])
```

```
[ ] max(output)
```



```
0.9806098075939801
```

```
[ ] def predict(network,sample):
    outputs=forward_propagate(network,sample)
    return outputs.index(max(outputs))
```

```
[ ] for sample in test_data:
    prediction=predict(network,sample)
    print(sample[-1], prediction)
```

```
0 0
1 1
1 1
1 0
```

```
[ ] n_inputs
```

```
2
```

```
[ ] y=[0,1,1,1,2,2]
```

```
[ ] sample=[1,3,0]
sample[:-1]
```

```
[1, 3]
```

```
[ ] len(set(y))
```

```
3
```

```
[ ] dataset=[[2.7,2.5,0],
             [1.4,2.3,0],
             [2.4,2.3,0],
             [3.4,2.3,0],
             [4.4,3.3,0],
             [8.4,2.3,1],
             [9.4,6.3,1],
             [8.4,7.3,1],
             [9.6,8.3,1],
             [9.4,7.3,1]]

[ ] from random import seed
    seed(1)
    n_inputs=len(dataset[0])-1
    n_outputs=len(set([sample[-1] for sample in dataset]))
    network=initialize_network(n_inputs, 2, n_outputs)
    alpha=0.3
    epochs=1000

[ ] train_network(network,dataset,alpha,epochs,n_outputs)
    for layer in network:
        print(layer)
```


CHAPTER 9



ADVICE FOR APPLYING MACHINE LEARNING TECHNIQUES

9. ADVICES FOR APPLYING MACHINE LEARNING TECHNIQUES

The difference between someone who is familiar with an algorithm and doesn't know how to use it, however, is enormous. Make sure you can pick the best routes for examining the numerous approaches. In this chapter, the major concern is selecting which choices to investigate. So, let's focus on learning algorithm's troubleshooting. Imagine that you utilized regularized linear regression to predict housing prices and trained it.

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right)^2 + \lambda \sum_{j=1}^m \theta_j^2 \right]$$

Unfortunately, you discover that it makes unacceptable-sized predictions mistakes when you test it with new data. What ought to you do next?

There are several options available to you, yet occasionally adding extra training data is ineffective. Often it does though, although you should always do some preliminary testing to make sure more data will make a difference. Try a more condensed set of features and pick a smaller subset with caution. You can either complete this manually or with the aid of a dimensionality reduction method (we'll discuss PCA later). Considering to acquire more features.

Isn't always useful because it could take a lot of time.

Building your own, adding new polynomial features based on your knowledge of the problem can be risky if you accidentally over fit your data by creating new features which are inherently specific/relevant to your training data. Try changing the value λ . Modify the regularization term's weighting in your computations. Changes can turn into MAJOR projects or hassles that last for at least six months. Sadly, choosing one of these cases is usually done based on intuition (randomly).

Half of the items on the list can be ruled out using some straightforward procedures. You'll save a lot of time!

Machine learning diagnostics

Test you can run to evaluate an algorithm's performance.

Examine what you can change to improve an algorithm's performance.

These could take a week to apply and understand, but they can save you from wasting months of effort trying to find a solution that won't work.

Evaluating a hypothesis

We aim to reduce an error when fitting parameters to training data. Though we may consider a low error to be desirable, this does not imply that our parameter settings are optimal.

This might be a sign of overfitting; in which case your model won't generalize.

How could we identify an overfitting hypothesis? Could plot $h_{\theta}(x)$, however it can be impossible to plot with many features. To examine a hypothesis, data are often divided into two groups. The first part is a training set, while the second part is a test set. A common split could be 70:30 (training: test).

Size	Price	
2104	400	70% Training set
1600	330	
2400	369	
1416	232	
3000	540	
1985	300	
1534	315	
1427	199	30% Test set
1380	212	
1494	243	

$(x^{(1)}, y^{(1)})$
 $(x^{(2)}, y^{(2)})$
 $\vdots$
 $(x^{(m)}, y^{(m)})$

$(x_{test}^{(1)}, y_{test}^{(1)})$
 $(x_{test}^{(2)}, y_{test}^{(2)})$
 $\vdots$
 $(x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$

Send a random percentage if the data is ordered (or randomly order, then send data).

In general, data is already arranged in some form. Consequently, a common train and test plan would be.

1. Learn parameters θ from training data, minimizing $J(\theta)$ using 70% of the training data]
2. Calculate the test error where $J_{\text{test}}(\theta)$ represents the average square error calculated using test set data.

$$J_{\text{test}}(\theta) = \frac{1}{2m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} (h_{\theta}(x_{\text{test}}^{(i)}) - y_{\text{test}}^{(i)})^2$$

This is how the test set error is defined. What if logistic regression were used instead? Similarly, study using 70% of the data and test using the remaining 30%

$$J_{\text{test}}(\theta) = \frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} y_{\text{test}}^{(i)} \log h_{\theta}(x_{\text{test}}^{(i)}) + (1 - y_{\text{test}}^{(i)}) \log h_{\theta}(x_{\text{test}}^{(i)})$$

There are situations when a better definition of misclassification error (0/1 misclassification) is available. The error is defined as follows:

$$\text{err}(h_{\theta}(x), y) = \begin{cases} 1 & \text{if } h_{\theta}(x) \geq 0.5, y = 0, \text{ or } h_{\theta}(x) < 0.5, y = 1 \\ 0, & \text{otherwise} \end{cases}$$

Then the test error is

$$\text{Test error} = \frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \text{err}(h_{\theta}(x_{\text{test}}^{(i)}), y_{\text{test}}^{(i)})$$

It is the portion of the test set that the hypothesis incorrectly labels. These are the methods that are typically used to assess learnt hypotheses.

Model selection and training validation test sets

How to select the regularization parameter or the polynomial's degree (model selection problems).

Overfitting has already caused us some anxiety.

Therefore, training set error is a subpar measure of the precision of a hypothesis for new data (generalization).

Model selection problem

Choose a polynomial's degree of fit to the data.

1. $h_{\theta}(x) = \theta_0 + \theta_1 x$
2. $h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2$
3. $h_{\theta}(x) = \theta_0 + \theta_1 x + \dots + \theta_3 x^3$
- .
- .
- .
10. $h_{\theta}(x) = \theta_0 + \theta_1 x + \dots + \theta_{10} x^{10}$

d represents a degree of polynomial you want to pick

$d=1$ (linear)

$d=2$ (quadratic)

...

$d=10$

Choose a model, fit it, and get a generalizability estimate for your hypothesis. You might think about model 1, which produces a parameter θ_1 by reducing with training data (where $d=1$). Consider model 2, carry out the identical processes, and get a different θ_2 (where $d=2$), and so on. Use the preceding formula to examine the test set error for each of these parameters $J_{test}(\theta_1), J_{test}(\theta_2), \dots, J_{test}(\theta_{10})$. It would then be possible to identify the model with the lowest test set error. Think about the situation where $d=5$ is the lowest and think about how well it generalizes. You could make use of $J_{test}(\theta_5)$. But, because our parameter fits that test set, this will be a pessimistic estimate of the generalization error (i.e. specifically chose it because the test set error is small). And is not a good technique to determine whether it will generalize.

To address this issue, we modify model selection and adopt a slightly different strategy:

Instead, a training set divided into three sections was provided

- 1 - Training set (60%) - m values
- 2 - Cross validation (CV) set (20%) m_{cv}
- 3 - Test set (20%) m_{test}

As before, training error, cross validation error, and test error can be calculated as follows:

Training error

$$J_{train}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right)^2$$

Cross Validation error

$$J_{cv}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{cv}} \left(h_{\theta}(x_{cv}^{(i)}) - y_{cv}^{(i)} \right)^2$$

Test error

$$J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} \left(h_{\theta}(x_{test}^{(i)}) - y_{test}^{(i)} \right)^2$$

Therefore, for each of the models, minimize cost function as previously.

The cross-validation error will then be generated by testing these hypotheses on the cross-validation set. Select the theory that has the lowest cross validation error, for example θ_5 . Lastly, calculate the model's generalization error using the test set.

The test error is frequently employed in machine learning since it is currently utilized to assess if a model is acceptable for generalization. This we have shown is problematic because it gives a bias analysis. With a HUGE test set, this would be acceptable. However, it is believed that having separate training and validation sets is a far better technique.

Diagnosis - bias vs. variance

Poor results are probably the result of one of the issues with strong bias or high variance. Understanding the problem is essential since it will help you improve the algorithm.

Regularization and bias/variance

How does regularization affect bias and variance?

Linear regression with regularization

Model:

$$h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

In the equation above, regularization is employed to fit a high order polynomial (used to keep parameter values small).

Here are three examples:

If λ is large.

All θ values are heavily penalized. So, most parameters end up being close to zero and hypothesis ends up being close to 0. So, high bias causes under fitting data. If all values θ receive severe penalties.

As a result, most parameters and the hypothesis are near to zero.

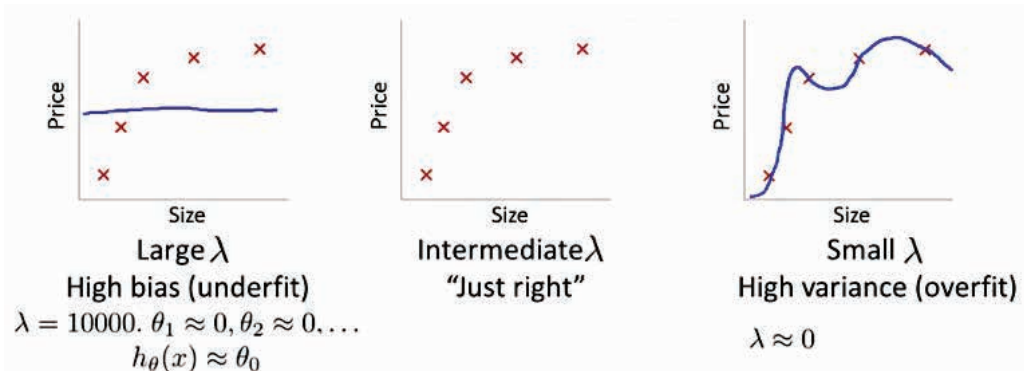
Therefore, underfitting data is due to excessive bias.

If λ is intermediate.

Only these values provide a fitting that makes sense.

If λ is small or $\lambda=0$

Then, we set the regularization term to 0, which indicates significant variance and overfitting (minimum regularization indicates that it clearly fails to achieve its goals).

Figure 27. High bias vs High variance hypothesis with regularization applied

What to do next

Let's summarize. How do these ideas help us choose how we approach a problem? Train a learning algorithm with regularized linear regression. But when you test on new data you find it makes unacceptably large errors in its predictions. What should you try next? How do we decide what to do?

More instances can help with high variance, but they are bad if you also have high bias.

Reduced number of features: Addresses high variance (overfitting). If your bias is strong, this is bad. Consider incorporating more features to address high bias (because hypothesis is too simple, make hypothesis more specific). Adding polynomial terms resolves the high bias issue.

Reduced λ corrects strong bias, increases high variance and fixes it. Referring to neural networks once more, choosing a network architecture. Use a tiny neural network with a few (perhaps one) hidden layers and hidden units as one method. Although these networks are computationally less expensive, they are prone to underfitting. Greater network size equates to more hidden layers. How do you assess whether a bigger network is beneficial? A single concealed layer is a suitable starting point. Try 1, 2, and 3 as well to see which performs the best on the cross validation set. So, take three sets as you did before (training, cross validation and test). The computational cost of adding more units makes them more likely to overfit. Regularization can be used to combat overfitting.

CHAPTER 10



MACHINE LEARNING SYSTEM DESIGN

10. MACHINE LEARNING SYSTEM DESIGN

Selecting a priority for work, let's take an illustration of spam classification.

Figure 28. Spam classification example

From: cheapsales@buystufffromme.com To: ang@cs.stanford.edu Subject: Buy now! Deal of the week! Buy now! Rolex w4tchs - \$100 <u>Medicine</u> (any kind) - \$50 Also low cost M0rgages available. Spam	From: Alfred Ng To: ang@cs.stanford.edu Subject: Christmas dates? Hey Andrew, Was talking to Mom about plans for Xmas. When do you get off work. Meet Dec 22? Alf Non-spam
---	---

What characteristics could we define?

Spam (1.) Misspelled word, and not spam (0), representing a genuine material. How can we create a classifier that can tell the two apart? Representation of features.

How should x (email features) be represented? whether $y = \text{spam}(1)$ or not (0). One technique is to choose your own traits, such a list of 100 words that can tell whether an email is spam or not spam. The most common spam triggered words are buy, discount, offer, etc. In this example non-spam triggered words are: Andrew, please. These words come together to make one long vector. This is encoded to a reference vector, then check the message for words. Describe the x feature vector which is 0 or 1 depending on whether a word matching a word in the reference vector is present. A bitmap of the word content is shown in Figure 29.

Within the x feature vector can contain either 0 or 1 depending on whether a word that matches a word in the reference vector is present?

Here, the text content is displayed as a bitmap.

Figure 29. Feature vector for Spam classification example

$$X = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 1 \\ \vdots \end{bmatrix} \begin{matrix} \text{andrew} \\ \text{buy} \\ \text{deal} \\ \text{discont} \\ \vdots \\ \text{now} \\ \vdots \end{matrix}$$

It is more typical to choose top n words from a training set, where n ranges from 10 000 to 50 000 words. In this case, you are choosing how to choose features from the training set data rather than expressly choosing your own features.

Which activity will most effectively increase system accuracy?

The desire to gather a lot of data is normal. Try to convince spammers to use phony email addresses so they may gather plenty of spam. Though it doesn't always work. Create specialized features depending on email routing data (contained in email header). Spammers frequently attempt to hide the source of emails. Send via peculiar pathways. Create powerful message body analysis features (i.e. Discount == discounts? DEAL == deal?)

Create a complex algorithm to identify misspellings. Spammers avoid detection by using misspelled words. A research team will frequently focus at random on one solution. It might not be the most productive use of your time. However, it is beneficial to come up with a variety of ideas, yet it might be very tempting to just try something.

Error analysis

There are several recommendations for improving ML issues. Talk about error analysis and how it can aid in your decision-making. When developing a machine learning system, it is always a good idea to start by developing an easy-to-implement algorithm. In little more than 24 hours, create an initial bootstrapped algorithm, test it on cross-validation data, put it into use, and draw learning curves to see whether further data, features, etc., can help with algorithmic optimization. It can be difficult to anticipate what is important.

For this problem, learning curves are quite helpful. How might early optimization be avoided? When selecting the development trajectory, we should base our conclusions on the available evidence. As part of the error analysis procedure, the samples (from the cross validation set) where your algorithm made faults should be manually checked. Look at the root of the problem. If we study systematic trends, we could design new features to circumvent these shortcomings. Using 500 instances from a CV set, a spam classifier used the following logic can be made. If there is a high rate of error, such as if we make 100 mistakes, then, we need manually review 100 of them and classify them based on criteria, such as the type of email. When we examine that email, we may discover that phishing and pharmaceutical emails are the most prevalent kind of spam emails. Next, determine which category is most prevalent and concentrate your efforts there. What characteristics, such as purposeful misspelling, strange email routing, or irregular punctuation, would have assisted in correctly classifying them. You may discover that a "spammer approach" is at blame for many of your misses. We should provide guidance and emphasize the relevance of the numerical evaluation. We ought to be able to evaluate the algorithm numerically. When constructing an algorithm, it's beneficial to run some performance calculations that provide a single real number to indicate how well the algorithm is performing. For instance, let's pretend we're choosing whether to regard a group of related words as the same word. Stemming in NLP is used to achieve this (e.g., "Porter stemmer" looks at the etymological stem of a word). As a result, your algorithm might become better or worse. Additionally, worth takes weighing error (false positive vs. false negative) into account. For example, is a bad situation when we have a false positive, or is it worthwhile to have a few of them to significantly boost performance? To compare the changes, we can apply numerical analysis. Check to see if the update has made the algorithm better afterwards. It may be difficult or complicated to compute a single real number, but it is far simpler to assess how modifications to your approach will affect it. Instead of using the test set for error analysis, use the cross validation set.

Error metrics for skewed analysis

Let's assume example for cancer classification.

Train logistic regression model $h_{\theta}(x)$ where, cancer means $y=1$, otherwise $y=0$. Test classifier on test set and i.e. we obtain 1% error. So, this looks pretty good, but in case that only 0.5% patient have cancer, 1% error looks very bad! So, when one number of examples is very small this is an example of skewed classes. If we have much more sample of one class than another, so standard error metrics aren't so good.

Another example: Algorithm has 99.2% accuracy. Make a change, now get 99.5% accuracy.

Does this genuinely reflect an algorithmic improvement? Did we do anything of value, or did we simply produce a model that forecasts $y=0$ more frequently? Classifier is still not very good despite having very low error.

Precision and recall

Precision and recall are two brand-new measurements. Each outputs a number between 0 and 1. The real class for a test set is either 1 or 0. The algorithm forecasts a value for each class and each example in the test set. Taking this into account, categorization

True positive (we guessed 1, it was 1),
False positive (we guessed 1, it was 0),
True negative (we guessed 0, it was 0) and
False negative (we guessed 0, it was 1)

Precision

How many false alarms does our algorithm generate?

How many of the people whose cancer we predicted will develop do.

= true positives / # predicted positive

= true positives / (true positive + false positive). High precision is good (i.e. closer to 1)

Because you want the false positive to be as near to 0 as feasible, you want a large number.

Recall

How sensitive is our algorithm?

Of all patients in set that have cancer, what fraction did we correctly detect?

= true positives / # actual positives

= true positive / (true positive + false negative)

High recall is good (i.e. closer to 1)

Because you want the false negative to be as near to 0 as feasible, you want a high number.

Get a better understanding of how well an algorithm is performing by computing precision and recall. There isn't really a way to game this.

As a result, we are far more certain that an algorithm is sound. In most cases, we declare that what we're attempting to ascertain is the existence of a rare class (for example, positive (1) is the existence of the rare thing).

Trading off precision and recall

We want to manage the trade-off between recall and precision for numerous applications.

Example

a logistic regression classifier was trained

Predict 1 if $h_{\theta}(x) \geq 0.5$

Predict 0 if $h_{\theta}(x) < 0.5$

This classifier may give some value for precision and some value for recall

Predict 1 only if very confident.

One way to do this modify the algorithm we could modify the prediction threshold.

Predict 1 if $h_{\theta}(x) \geq 0.8$

Predict 0 if $h_{\theta}(x) < 0.2$

Now we can be more confident a 1 is a true positive.

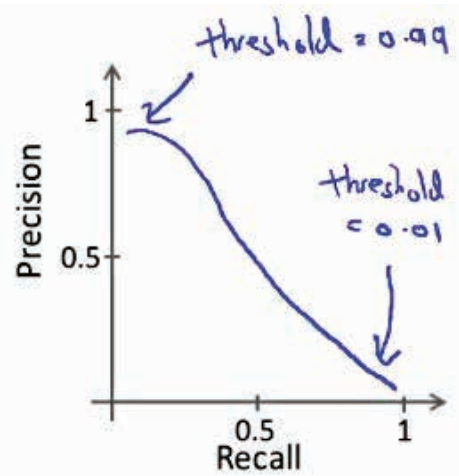
But classifier has lower recall - predict $y=1$ for a smaller number of patients. This represents a risk of false negatives. Another example - avoid false negatives. This is probably worse for the cancer example. Now we may set to a lower threshold

Predict 1 if $h_{\theta}(x) \geq 0.3$

Predict 0 if $h_{\theta}(x) < 0.3$

i.e., 30% chance they have cancer. So, while precision has decreased, recall has increased. Risk of false positives because we are less selective in determining what constitutes a cancer diagnosis. The trade-off is specified by this threshold. Plotting precision vs. recall allows us to illustrate this in a visual way.

Figure 30. Precision recall curve



Depending on the classifier details, this curve can take on a variety of shapes. Is it possible to have the threshold selected automatically? How can we compare various algorithms or parameter sets if we have several algorithms?

	Precision (P)	Recall (R)
Algorithm 1	0.5	0.4
Algorithm 2	0.7	0.1
Algorithm 3	0.02	1.0

How can we select the most effective algorithm among these? Prior to this, we discussed the use of a single real number assessment measure. As soon as we switch to precision/recall, we have two numbers. Comparison is now more difficult. Having just one number is preferable. How may P and R be combined into one number? The average - $(P + R)/2$ is one choice. This is a less than ideal solution. This indicates that if a classifier consistently predicts $y=1$, it will have a high recall and low precision. Like this, we rarely achieve high precision and low recall when we forecast that $y=0$. The averages would therefore be 0.45, 0.4, and 0.51. 0.51 is the best, despite having a recall of 1 - i.e. predict $y=1$ for everything. So average isn't great.

$$F1Score (fscore) = 2 * (PR / [P + R])$$

Fscore is like averaging recall and precision while giving the lower number more weight.

Numerous formulas for calculating numbers of equivalent precision and accuracy.

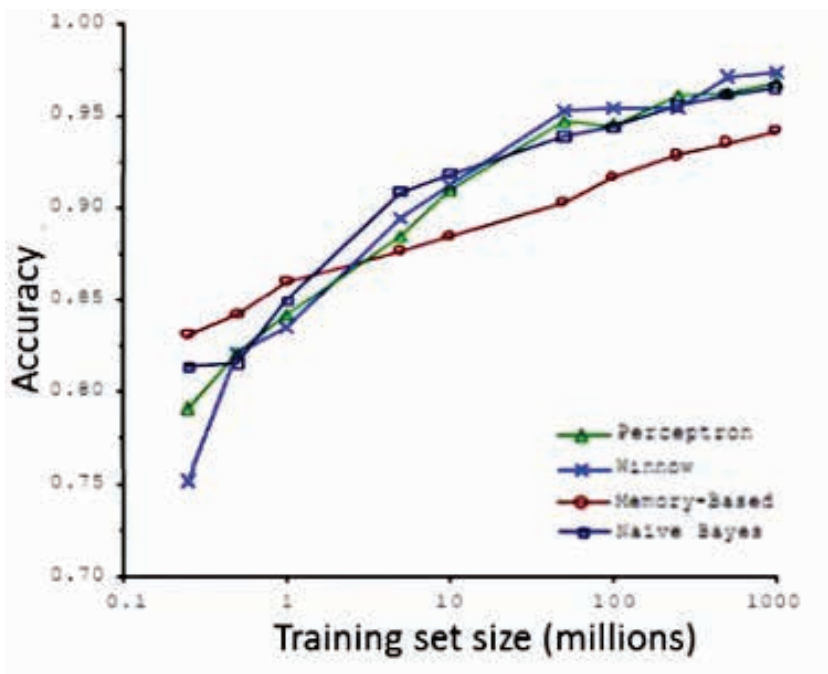
If $P=0$ or $R=0$ the $F_{score}=0$

If $P=1$ and $R=1$ then $F_{score}=1$

Fscore is like taking the average of precision and recall giving a higher weight to the lower value.

Many formulas for computing comparable precision/accuracy values. Now switch tracks and look at how much data to train on. One suggestion is to opportunistically get more information. It appears that in some situations, obtaining more data is a highly effective approach to enhance performance. Studies on the use of various algorithms on data have been conducted to build a learning system with high accuracy. Figure 31 shows different algorithms performance vs. range of sizes. Algorithms that are used are: Perceptron (logistic regression), Winnow, like logistic regression (used less now), memory based (used less now) and Naive Bayes.

Figure 31. Accuracy vs. Training set size for different algorithms



From all the previous discussion, what conclusion can we draw?

The performance of algorithms is strikingly similar. Accuracy rises as training set size rises. Give an algorithm more data, and it should outperform a "better" algorithm with fewer data shows that the choice of algorithm is equivalent. More data can be helpful, but this isn't always the case. More data will undoubtedly be helpful if we can confidently believe that features x have enough information to predict y accurately. Whether this is accurate can be tested by asking, "Can a human expert predict y given x ?". Let's look at an application of a learning method with numerous parameters, such as logistic regression, linear regression with a large amount of data, or neural networks with numerous hidden features. These are strong learning algorithms that, because of their many parameters can fit complex functions. These algorithms exhibit less bias.

Another way to think about this is we want our algorithm to have low bias and low variance. The best option is to have a low bias and a low variance. To achieve this we need to use complex algorithm and large training set, respectively.

CHAPTER 11



CLUSTERING (K means)

11. CLUSTERING (K means)

Unsupervised learning is primarily concerned with figuring out the data's structure or pattern.

For instance, clustering data, recommendation algorithms, etc. do not use labeled data (or the dependent variable is not there). Unsupervised learning yields astounding results since it enables the discovery of numerous undiscovered relationships between various features or attributes. After the data points are clustered, all data points with similar properties will be put into a single cluster to identify any hidden patterns in the data. So, what applications does clustering have?

Have you ever noticed how Google News has thousands of news stories in each category, including sports, politics, movies, and science? They are forming groups. The algorithm creates a cluster by grouping the news items according to the traits they have in common.

Market segmentation, recommendation systems, social network analysis, and others are more examples of clustering. One of the fundamental clustering methods is K-means technique.

Let's investigate the goal of clustering using some real-world examples. The plot of the dataset with features x_1 and x_2 is shown in the first picture. As you can see, the data are not clustered, making it impossible to infer any conclusions only from the plot. We obtained the second image after clustering was finished, in which we can see that data points that are close to one another are gathered as a cluster. In terms of application, this is a circumstance where we must divide the market depending on the earnings and volume of previous transactions of our clients. Assuming that the annual income x_1 and the number of transactions x_2 are the features in this case, we can group the data and divide it into three categories based on these features. Those groups could be customers with low annual income but a high number of transactions (like the orange cluster), customers with medium income and a medium number of transactions (like the green cluster), and customers with high income but a low number of transactions (like blue cluster). Based on these divisions, the company's marketing staff can adjust its marketing tactics to increase sales by suggesting goods that each cluster's customers would purchase.

Figure 32. Pre-clustering data

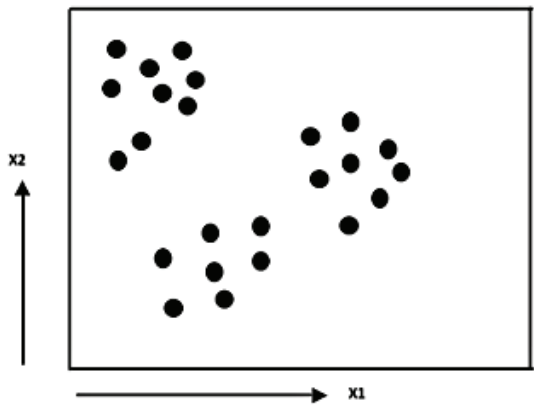
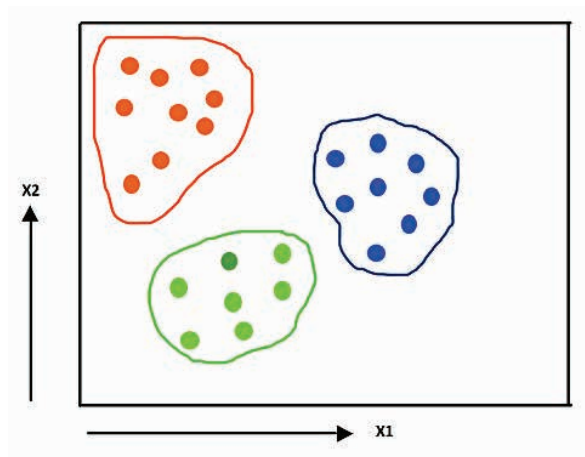


Figure 33. Post-clustering data

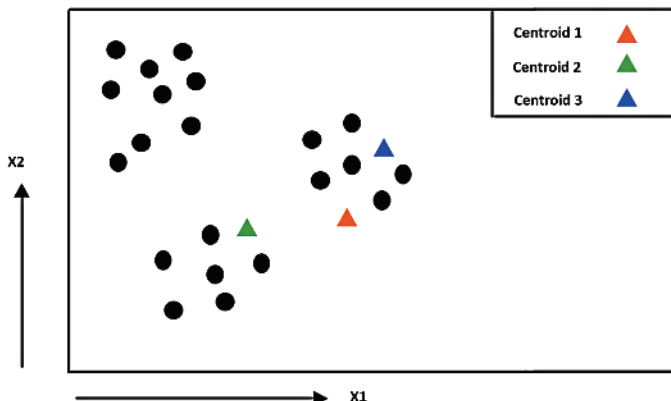


11.1.INTUITION FOR K-Means CLUSTERING

Now that we've covered the goal of clustering and a practical application, it's time to discuss K-means clustering's implementation and algorithm. As the name already implies, this algorithm will group n data points into K number of clusters. Therefore, we must cluster the enormous amount of data into K clusters. But how do you choose how many clusters to use? We will typically be aware of the type of clusters we should employ. For instance, in the case of Google News, we were aware that news stories should be categorized into one of four general types of clusters (sports, politics, science, and movies). However, in case of the market segmentation problem, as the objective is to cluster the categories of customers, we are unsure of the number of customer types existing in the dataset (such as rich, poor, those who enjoy shopping, those who do not, etc.). We don't know exactly how many clusters should be selected in this scenario, and we can't guesstimate the number of clusters based on some presumptions (such as those with annual incomes larger than 'x' amount should be in one cluster and lower than 'x' amount should be in another). We shall look at a solution to this problem later, but let's assume for the time being that our data will be split into three clusters. K-means clustering is connected to the mathematics ideas of centroids and Euclidian distance. Just a short reminder that the centroid of a data collection is its average or mean and that the Euclidean distance is the spacing between two points in the coordinate plane. Given two points $A(x_1, y_1)$ and $B(x_2, y_2)$, the Euclidian distance between these two points is:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Figure 34. Initialization of the centroid



Here, three cluster centers or centroids with the green, orange, and blue triangle markers are chosen randomly. Again, this is not an efficient method to choose the initial cluster centers.

Step 2:

Assign the cluster based on the shortest distance between all centroids after calculating the Euclidian distance from each data point to all centroids (3 in this case).

In our example, we must first determine the Euclidian distance of each black dot from each of the centroids (green, orange, and blue), and then we must assign each black dot the color of the closest centroid.

2b. Based on the calculations made in step 2a, adjust each cluster's centroid by averaging all the data points that make up that cluster. We need to compute the mean of each cluster in our example and move the centroid to the computed mean because we have already assigned all the data points to one of the clusters. Continue doing this until the clusters are clearly separated or convergence is reached.

Figure 35. 2.a Based on the calculated Euclidian distances, assign the centroids to each data point.

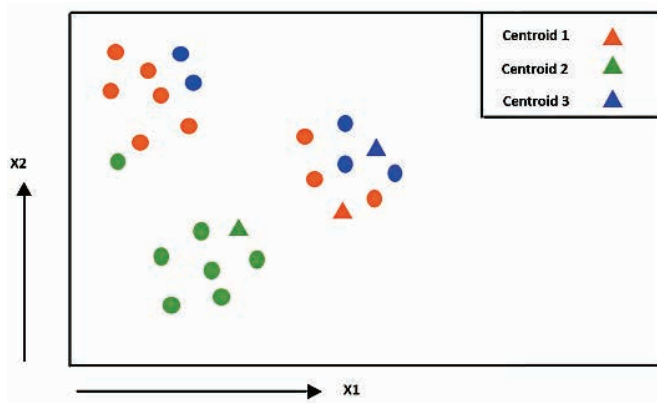
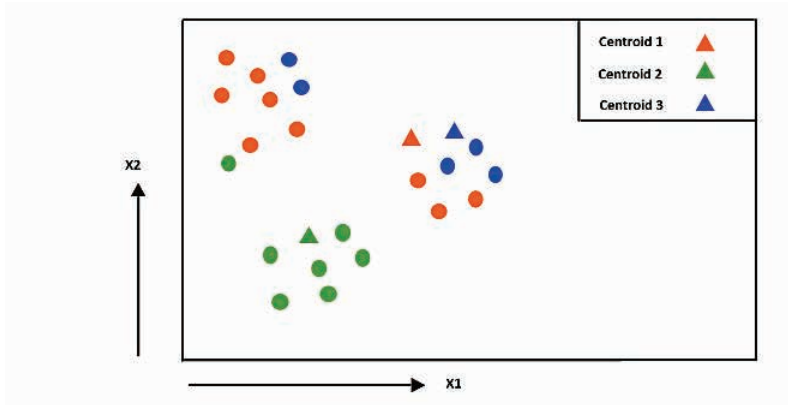
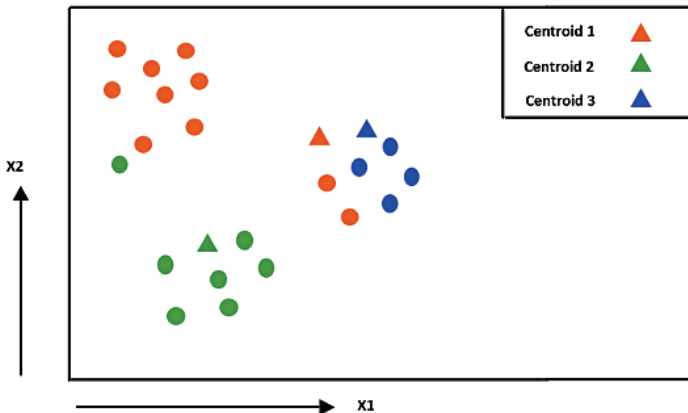


Figure 35. 2b. By averaging all of the data points in that cluster, adjust the centroids.



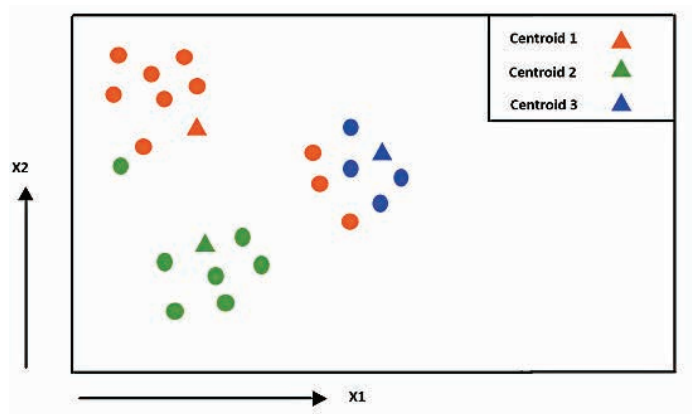
Follow these steps repeatedly until convergence is reached:

Figure 36. 2b. Adjust the centroids by taking the average of all data points which belong to that cluster



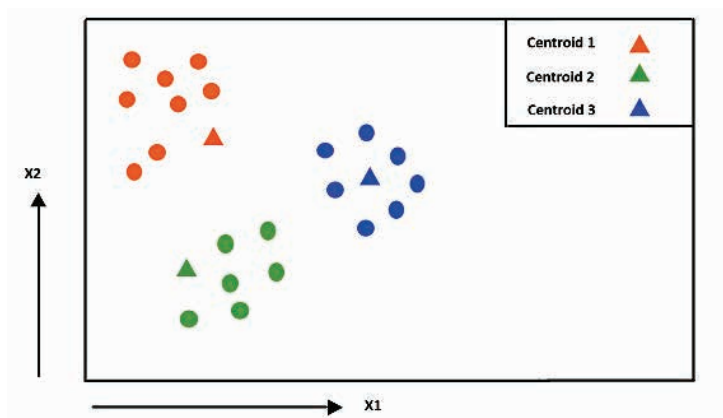
Repeat these steps until convergence is achieved:

Figure 37. 2b. By averaging all the data points in that cluster, adjust the centroids.



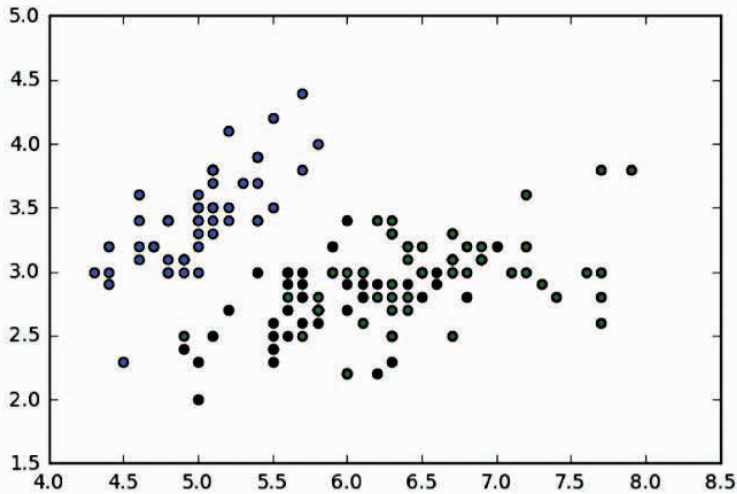
Follow these steps repeatedly until convergence is reached:

Figure 38. 2b. By averaging all the data points that make up that cluster, you can adjust the centroids.



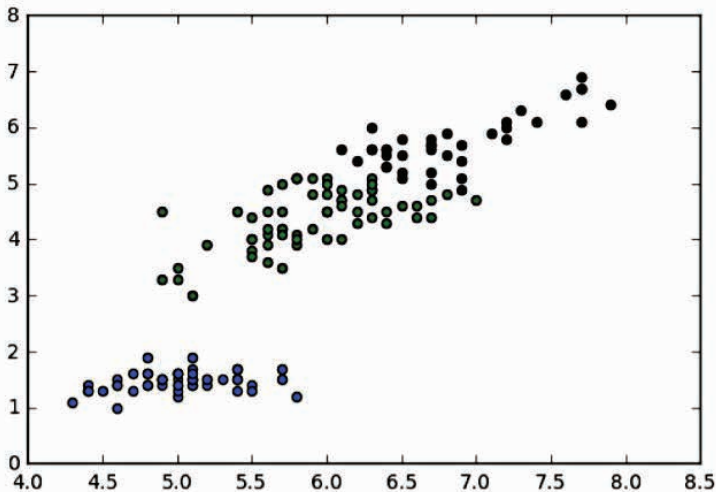

```
In [15]: plt.scatter(X[y==0,0], X[y==0,1])
plt.scatter(X[y==1,0], X[y==1,1], c='k')
plt.scatter(X[y==2,0], X[y==2,1], c='g')
```

Out[15]: <matplotlib.collections.PathCollection at 0xca01a90>



```
In [18]: plt.scatter(X[output==0,0], X[output==0,2])
plt.scatter(X[output==1,0], X[output==1,2], c='k')
plt.scatter(X[output==2,0], X[output==2,2], c='g')
```

Out[18]: <matplotlib.collections.PathCollection at 0xa06b38>



Exercise 1. K means using numpy library

Let's start from scratch and implement the method in Python now that we are more comfortable with intuition. The use of the Numpy, Pandas, and Matplotlib libraries is required because to the increased processing complexity and result visualization. The dataset known as "Mall Customers.csv" has a link on the GitHub page. This is the dataset that we'll be using. The dataset was retrieved from the SuperDataScience website. Let's read the dataset and gather a few data instances. The following traits are present in this situation:

1. CustomerID: It is the unique ID given to a customer.
2. Gender: Gender of the customer.
3. Age: The age of the customer.
4. Annual Income (k\$): It is the annual income of the customer
Spending Score: It is the score (out of 100) given to a customer by the mall authorities, based on the money spent and the behavior of the customer.

```
import pandas as pd
import numpy as np
import random
import matplotlib.pyplot as plt
dataset=pd.read_csv('Mall_Customers.csv')
dataset.head()
```

	CustomerID	Genre	Age	Annual Income (k\$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

We'll use the Annual Income and Spending Score as the base for our investigation.

```
[ ] X = dataset.iloc[:, [3, 4]].values # we are taking only third and fourth column
```

```
[ ] X[:5,:]
```

```
array([[15, 39],
       [15, 81],
       [16, 6],
       [16, 77],
       [17, 40]])
```

Now X is two shape matrices (200,2). Next step is choosing how many rounds could be required to ensure convergence. We must explore a wide range of options to discover the ideal number of iterations required for convergence.

There is no need to choose a particularly large number because, let's say, at the 100th iteration, if the centroids reached their true location or best viable location, the algorithm would still generate the same outcomes even after carrying out 1000 additional iterations.

Let's pick 100 iterations as the initial number of iterations to make things simpler.

K cluster selection comes next. As stated before, let's choose K to be 5, and we'll see a method later to help us determine the optimal K number of clusters.

```
[ ] m=X.shape[0] #number of training examples
    n=X.shape[1] #number of features. Here n=2
    n_iter=1000
    K=5 # number of clusters
```

```
[ ] m
```

```
200
```

```
[ ] n
```

```
2
```

```
[ ] K=5 # number of clusters
```

```
[ ] centroids=np.zeros([n*K])
    centroids=centroids.reshape(n,K)
    #Centroids is a n x K dimensional matrix,
    #where each column will be a centroid for one cluster.
```

```
[ ]
```

```
[ ] centroids
```

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

```
[ ] np.shape(centroids)
```

```
(2, 5)
```

```
[ ] rand=random.randint(0,m-1)
```

```
[ ] X[rand,:]
```

```
array([42, 60])
```

```
[ ] X[197,:]
```

```
array([126, 74])
```

'centroids' is a $n \times K$ dimensional matrix, where the centroid for one cluster is represented by each column.

```
[ ] for k in range(K): #K times-initialize the centroids randomly from the data points, taking K samples
    #from m samples
    rand=random.randint(0,m-1) #data index is random
    centroids[:,k]=X[rand]
    #Centroids is a n x K dimensional matrix,
    #where each column will be a centroid for one cluster.
```

```
[ ] centroids
```

```
array([[ 65.,  98.,  63.,  62., 101.],
       [ 50.,  88.,  50.,  41.,  68.]])
```

Step 2.a: For each training example, determine the Euclidian distance from the centroid, and then select the cluster with the least distance.

The only output from our application should be a dictionary with the cluster number as the Key and the data points in that cluster as the Values. Let's therefore begin the process from scratch. Each point's Euclidian distance from each centroid is determined, and the results are stored in a $m \times K$ matrix. As a result, each row of the EuclidianDistance matrix contains a list of the distances between each data point and each of the centroids. After we establish the minimal distance, the index of the column will then be kept in a vector C.

```
[ ] ''' Step 2.a. For each training example compute the euclidian distance
    from the centroid and assign the cluster based on the minimal distance
    The output of our algorithm should be a dictionary with cluster
    number as Keys and the data points which belong to
    that cluster as values. So let's initialize the dictionary.'''
```

 ' Step 2.a. For each training example compute the euclidian distance \nfrom the

```
[ ] EuclidianDistance=np.zeros([m*K])
    EuclidianDistance=EuclidianDistance.reshape(m,K)
```

```
[ ] EuclidianDistance
```

 array([[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., 0., 0., 0.],
 [0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0.]

```
[ ] EuclidianDistance=np.zeros([m*K])
EuclidianDistance=EuclidianDistance.reshape(m,K)
for k in range(K):
    tempDist=np.sum((X-centroids[:,k])**2,axis=1)
    EuclidianDistance[:,k]=tempDist
C=np.argmin(EuclidianDistance,axis=1)+1 #returns the k for which we have minimum distance

'''We find the euclidian distance from
each point to all the centroids and store in a m X K matrix.
So every row in EuclidianDistance matrix will have distances
of that particular data point from all the centroids.
Next, we are going to find the minimum distance and store
the index of the column in a vector C.'''
```

• We find the euclidian distance from \neach point to all the centroids and store in a $m \times K$:

[] C

```
 array([1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2,  
       1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 3,  
       1, 2, 3, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,  
       3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,  
       3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,  
       3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,  
       4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5,  
       4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5,  
       4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5, 4, 5,  
       4, 5])
```

```
[ ] np.shape(C)
```

 (200,)

We are locating samples' indexes for each cluster in this stage.

```
[ ] C1ind=np.where(C==1)#indexes of the samples in cluster1
    C2ind=np.where(C==2)#indexes of the samples in cluster2
    C3ind=np.where(C==3)#indexes of the samples in cluster3
    C4ind=np.where(C==4)#indexes of the samples in cluster4
    C5ind=np.where(C==5)#indexes of the samples in clusters5
```

[] C2

```
(array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33,
        35, 37, 39, 41, 45]),)
```

We are currently grouping the samples inside each cluster.

```
[ ] Y1=X[C1ind]#Group of samples in cluster1
    Y2=X[C2ind]#Group of samples in cluster2
    Y3=X[C3ind]#Group of samples in cluster3
    Y4=X[C4ind]#Group of samples in cluster4
    Y5=X[C5ind]#Group of samples in cluster5
    C1=Y1.mean(axis=0)#mean value of cluster1
    C2=Y2.mean(axis=0)#mean value of cluster2
    C3=Y3.mean(axis=0)#mean value of cluster3
    C4=Y4.mean(axis=0)#mean value of cluster4
    C5=Y5.mean(axis=0)#mean value of cluster5
```

```
[ ] C2
```

```
array([25.72727273, 79.36363636])
```

```
[ ] np.shape(Y1)
```

```
(26, 2)
```

```
[ ] Y1
```

```
array([[15, 39],
       [16, 6],
       [17, 40],
       [18, 6],
       [19, 3],
       [19, 14],
       [20, 15],
       [20, 13],
       [21, 35],
       [23, 29],
       [24, 35],
       [25, 5],
       [28, 14],
       [28, 32],
       [29, 31],
       [30, 4],
       [33, 4],
       [33, 14],
       [34, 17],
       [37, 26],
       [38, 35],
       [39, 36],
       [39, 28],
       [40, 47],
       [40, 42],
       [40, 42]])
```

[] Y2

```

array([[15, 81],
       [16, 77],
       [17, 76],
       [18, 94],
       [19, 72],
       [19, 99],
       [20, 77],
       [20, 79],
       [21, 66],
       [23, 98],
       [24, 73],
       [25, 73],
       [28, 82],
       [28, 61],
       [29, 87],
       [30, 73],
       [33, 92],
       [33, 81],
       [34, 73],
       [37, 75],
       [38, 92],
       [39, 65]])

```

Step 2: Repeat this process once more until convergence is achieved.

In other words, we loop through `n_iter` and repeat 2.a and 2.b as shown:

```

[ ] for k in range(K): #K times-initialize the centroids randomly from the data points, taking K samples
    #from m samples
    rand=random.randint(0,m-1) #data index is random
    centroids[:,k]=X[rand]
    n_iter=200
for i in range(n_iter):
    for k in range(K):
        tempDist=np.sum((X-centroids[:,k])**2,axis=1)
        EuclidianDistance[:,k]=tempDist
    C=np.argmin(EuclidianDistance,axis=1)+1 #returns the k for which we have minimum distance
    C1ind=np.where(C==1)
    C2ind=np.where(C==2)
    C3ind=np.where(C==3)
    C4ind=np.where(C==4)
    C5ind=np.where(C==5)
    Y1=X[C1ind]
    Y2=X[C2ind]
    Y3=X[C3ind]
    Y4=X[C4ind]
    Y5=X[C5ind]
    C1=Y1.mean(axis=0)
    C2=Y2.mean(axis=0)
    C3=Y3.mean(axis=0)
    C4=Y4.mean(axis=0)
    C5=Y5.mean(axis=0)
    centroids[:,0]=C1
    centroids[:,1]=C2
    centroids[:,2]=C3
    centroids[:,3]=C4
    centroids[:,4]=C5

```

[] C

```
array([3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4,
       3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 2,
       3, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 5, 1, 2, 1, 5, 1, 5, 1,
       5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 2, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1,
       5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1,
       5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1, 5, 1,
       5, 1])
```

[1] C3

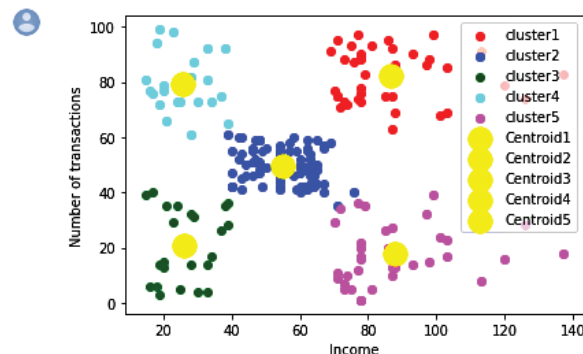
 array([99.88888889, 81.38888889])

[] C1

 array([62. , 84.93103448])

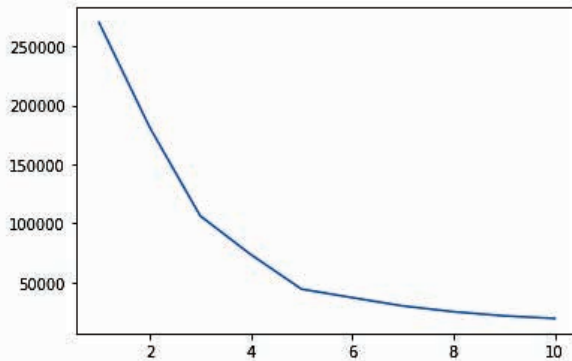
Plotting the grouped data now

```
[ ] color=['red','blue','green','cyan','magenta']
labels=['cluster1','cluster2','cluster3','cluster4','cluster5']
plt.scatter(Y1[:,0],Y1[:,1],c=color[0],label=labels[0])
plt.scatter(Y2[:,0],Y2[:,1],c=color[1],label=labels[1])
plt.scatter(Y3[:,0],Y3[:,1],c=color[2],label=labels[2])
plt.scatter(Y4[:,0],Y4[:,1],c=color[3],label=labels[3])
plt.scatter(Y5[:,0],Y5[:,1],c=color[4],label=labels[4])
plt.scatter(C1[0],C1[1],s=300,c='yellow',label='Centroid1')
plt.scatter(C2[0],C2[1],s=300,c='yellow',label='Centroid2')
plt.scatter(C3[0],C3[1],s=300,c='yellow',label='Centroid3')
plt.scatter(C4[0],C4[1],s=300,c='yellow',label='Centroid4')
plt.scatter(C5[0],C5[1],s=300,c='yellow',label='Centroid5')
plt.xlabel('Income')
plt.ylabel('Number of transactions')
plt.legend()
plt.show()
```



```
[ ] from sklearn.cluster import KMeans
cost=[]
for k in range(1,11):
    kmeans=KMeans(n_clusters=k,init='k-means++',random_state=42)
    kmeans.fit(X)
    cost.append(kmeans.inertia_)
plt.plot(range(1,11),cost)
```

[<matplotlib.lines.Line2D at 0x7f65168a3198>]



```
[ ] #Vectorization
```

```
[ ] '''Step 2.b: We need to regroup the data points based on the cluster index C and store
in the Output dictionary and also compute the mean of separated clusters and assign it as new centroids.
Y is a temporary dictionary which stores the solution for one particular iteration.'''
```

'Step 2.b: We need to regroup the data points based on the cluster index C and store \nin the Output dictionary

```
[ ] Y={}
for k in range(K):
    #Y[k+1]=np.zeros([numberOfSamplesinKthCluster,2 features]).reshape(numberofSamplesinKthCluster*features)
    Y[k+1]=np.array([]).reshape(2,0)
    for i in range(m):
        Y[C[i]]=np.c_[Y[C[i]],X[i]]

    for k in range(K):
        Y[k+1]=Y[k+1].T

    for k in range(K):
        centroids[:,k]=np.mean(Y[k+1],axis=0)
```

```
[ ] Y
```

```
{1: array([[15., 39.],
          [16., 6.],
          [17., 40.],
          [18., 6.],
          [19., 3.],
          [19., 14.],
          [20., 15.],
          [20., 13.],
          [21., 35.],
          [23., 29.],
          [24., 35.],
          [25., 5.],
          [28., 14.],
          [28., 32.],
          [29., 31.],
          [30., 4.],
          [33., 4.],
          [33., 14.],
          [34., 17.],
          [37., 26.],
          [38., 35.],
          [39., 36.],
          [39., 28.],
          [40., 47.],
          [40., 42.],
          [40., 42.]])}, 2: array([[15., 81.],
          [16., 77.]])
```

```
[ ] '''Now we need to repeat step 2 till convergence is achieved.
    In other words, we loop over n_iter and repeat the step 2.a and 2.b as shown:'''
```

```
⊞ 'Now we need to repeat step 2 till convergence is achieved. \nIn other words, we loop over n_iter and repeat the step 2.a and 2.b as shown:'
```

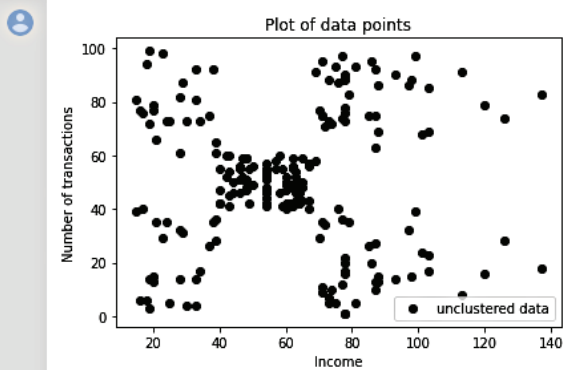
```
[ ] for i in range(n_iter):
    for k in range(K-1): #K times-initialize the centroids randomly from the data points, taking K samples
        #from m samples
        rand=random.randint(0,m-1) #data index is random
        centroids[:,k]=X[rand]
    C=np.argmin(EuclidianDistance,axis=1)+1 #returns the k for which we have minimum distance
    #step 2.b
    Y={}
    for k in range(K):
        Y[k+1]=np.array([]).reshape(2,0)
    for i in range(m):
        Y[C[i]]=np.c_[Y[C[i]],X[i]]#Translates slice objects to concatenation along the second axis.
    for k in range(K):
        Y[k+1]=Y[k+1].T
    for k in range(K):
        centroids[:,k]=np.mean(Y[k+1],axis=0)
    Output=Y
```

It's time to visualize the method and observe the clustering of the initial data.
Let's start by dispersing the initial unclustered data.

```
[ ] '''Now it's time to visualize the algorithm and notice how the original data is clustered.
    To start with, let's scatter the original unclustered data first.'''
```

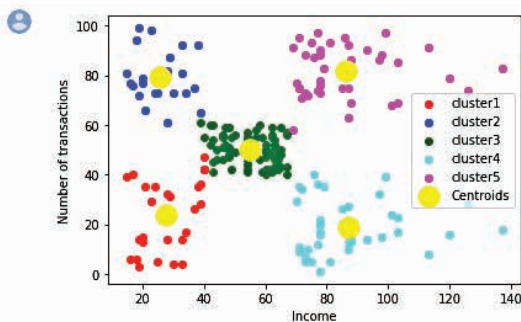
Now it's time to visualize the algorithm and notice how the original data is clustered. \nT

```
plt.scatter(X[:,0],X[:,1],c='black',label='unclustered data')
plt.xlabel('Income')
plt.ylabel('Number of transactions')
plt.legend()
plt.title('Plot of data points')
plt.show()
```

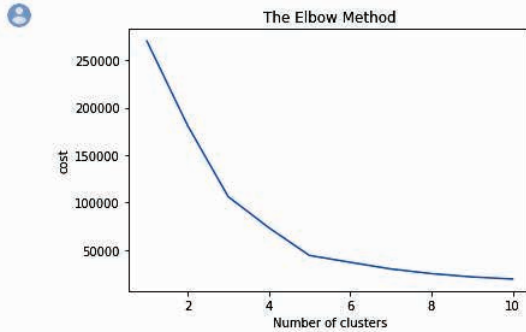


```
[ ] #Now let's plot the clustered data:
```

```
color=['red','blue','green','cyan','magenta']
labels=['cluster1','cluster2','cluster3','cluster4','cluster5']
for k in range(K):
    plt.scatter(Output[k+1][:,0],Output[k+1][:,1],c=color[k],label=labels[k])
plt.scatter(centroids[0,:],centroids[1,:],s=300,c='yellow',label='Centroids')
plt.xlabel('Income')
plt.ylabel('Number of transactions')
plt.legend()
plt.show()
```



```
[ ] #lets implement the same algorithm using sklearn libraries
# Using the elbow method to find the optimal number of clusters
from sklearn.cluster import KMeans
cost = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters = k, init = 'k-means++', random_state = 42)
    kmeans.fit(X)
    cost.append(kmeans.inertia_)#Sum of squared distances of samples to their closest cluster center.
plt.plot(range(1, 11), cost)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('cost')
plt.show()
```



CHAPTER 12



DIMENSIONALITY REDUCTION (PCA)

12. DIMENSIONALITY REDUCTION (PCA)

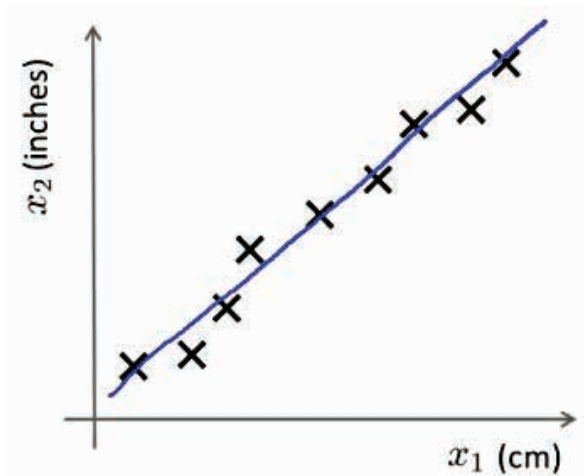
The Principle Component Analysis (PCA) method compresses a huge collection of variables into a smaller set while keeping the majority of the information in the larger set in order to minimize the dimensionality of enormous datasets. Accuracy always suffers when a dataset's variables are decreased, but the goal of dimensionality reduction is to exchange some accuracy for simplicity. Machine learning algorithms can analyze data much more rapidly and easily since there are fewer unimportant aspects to handle in smaller datasets, which are also simpler to examine and visualize.

In summary, the main goal of PCA is to preserve as much information while reducing the number of features in a data collection. For example, PCA can be used to find correlations between data points if there is a connection between the features.

Example 1

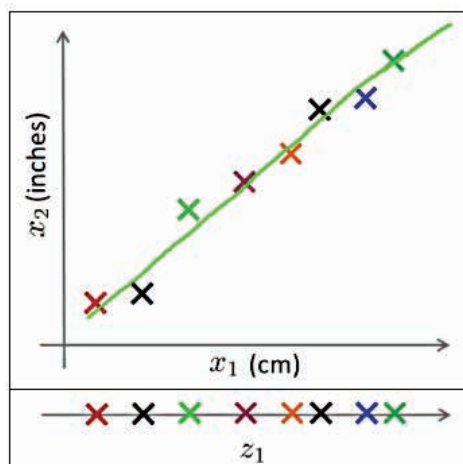
Let's assume a redundant dataset, which means that the same attribute has distinct units. Reduce data to 1D(2D->1D). If we consider that attribute x_1 represent a value in cm, and x_2 represents value in inches.

Figure 40. 2D dataset



Because of a round-off error, the example above isn't a perfectly straight line.

Figure 41. Dimensionality reduction (2D->1D)



Example 2

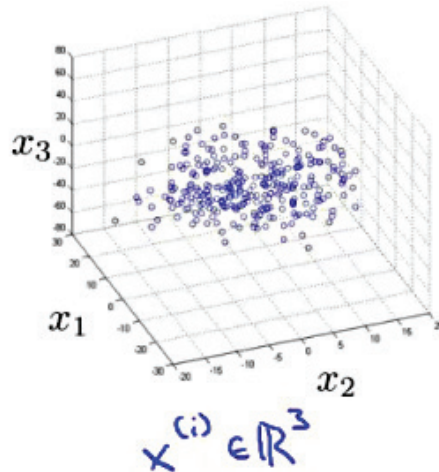
Let's use a different scenario. We wish to conduct a pilot survey where (x_1 =skill, x_2 =pilot enjoyment). These characteristics could be closely related. This association can be combined to form an aptitude trait (for example). What is meant by dimensionality reduction? In our illustration, a line is drawn. Record the position on the line using the precise example.

Before, our feature vector had two dimensions (X and Y), but now we can represent it as a single dimension (1D) number (Z dimension). Thus, we may reduce the quantity of storage to half using this approximation. However, the rounding mistake in the measurement is what causes the loss above.

Example 3

Another example 3D -> 2D

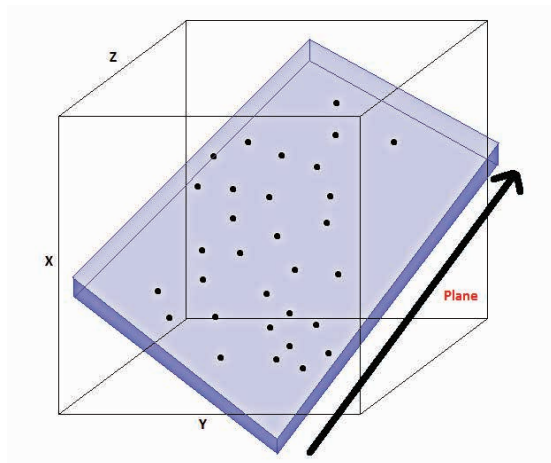
Figure 42. 3 D dataset



In a perfect world, perhaps all the data is contained on one plane. That plane might be aligned with one of the axes, although this is kind of difficult to convey in 2D images. In either case, the plane is a constrained, 3D spaces.

Consider that the blue tray in the diagram below contains all our data points (has a dark blue exterior face and a light blue inside).

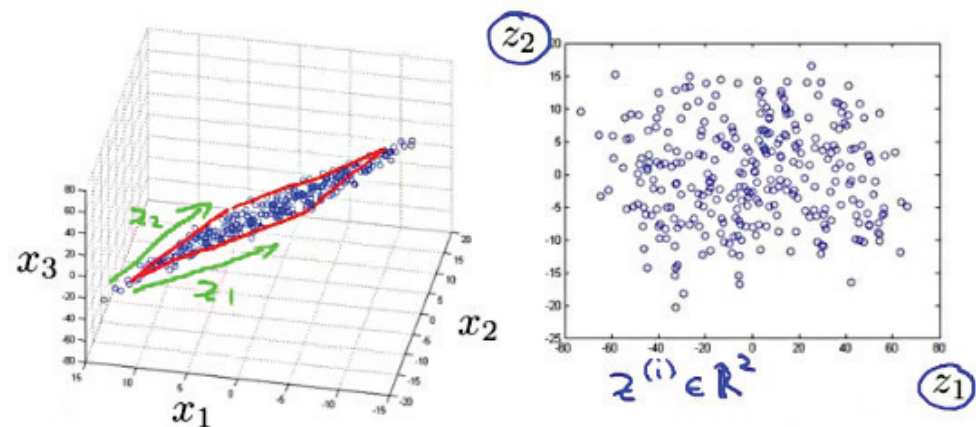
Figure 43. Dimensionality reduction (3D->2D)



We draw two new lines (z_1 and z_2) along the x and y planes of the box and plot the locations in that box, which means we lose the data in the z -dimension of our "shallow box" (NB "z-dimensions" refers to the dimension relative to the box (i.e., its depth) and NOT the z dimension of the axis we've got drawn above), but because the box is shallow it's OK to lose this.

Plotting values along those projections will help.

Figure 44. Dimensionality reduction (3D->2D)



Example 4

Assemble a sizable dataset containing details about several nations all across the world.

Figure 45. Multidimensionality dataset

Country	GDP (trillions of US\$)	Per capita GDP (thousands of intl. \$)	Human Develop- ment Index	Life expectancy	Poverty Index (Gini as percentage)	Mean household income (thousands of US\$)	...
Canada	1.577	39.17	0.908	80.7	32.6	67.293	...
China	5.878	7.54	0.687	73	46.9	10.22	...
India	1.632	3.41	0.547	64.7	36.8	0.735	...
Russia	1.48	19.84	0.755	65.5	39.9	0.72	...
Singapore	0.223	56.69	0.866	80	42.5	67.1	...
USA	14.527	46.86	0.91	78.3	40.8	84.3	...
...	...	...	...	...	...	...	...

So,

$x_1 = \text{GDP}$

...

$x_6 = \text{mean household}$

Let's say that each nation has 50 features.

How can we better interpret this data? Plotting data in 50 dimensions is really challenging.

Create a new feature representation (z values) that condenses these features instead of each country being represented by a 50-dimensional feature vector using dimensionality reduction.

Figure 46. Multidimensionality dataset after PCA implementation

Country	z_1	z_2
Canada	1.6	1.2
China	1.7	0.3
India	1.6	0.2
Russia	1.4	0.5
Singapore	0.5	1.7
USA	2	1.5
...	...	...

This gives us a 2-dimensional vector

Reduce 50D → 2D

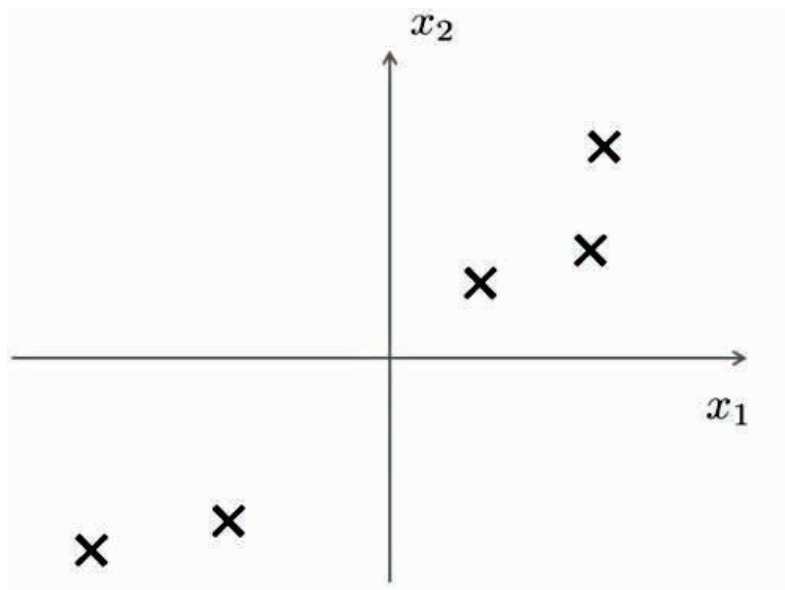
Plot as a 2D plot

Usually, you don't explain the meaning of the new features (so we have to figure out what these summary values mean); for instance, you might find that the horizontal axis corresponds to the size and economic activity of the entire country, while the y axis represents the economic activity and well-being of each individual. Therefore, even though there are 50 features, there may only be two "dimensions" of information, and each of those dimensions may have features associated with it. You must determine which features can be grouped to generate summary features and the most effective way to do so (feature scaling is probably important). It is beneficial to clearly depict the two primary aspects of variation.

Principle Component Analysis (PCA): Problem Formulation

The most widely used algorithm for the dimensionality reduction problem is PCA. We'll start discussing how to specify what we want PCA to accomplish at this point. Let's imagine we want to convert a 2D data collection to a 1D dataset.

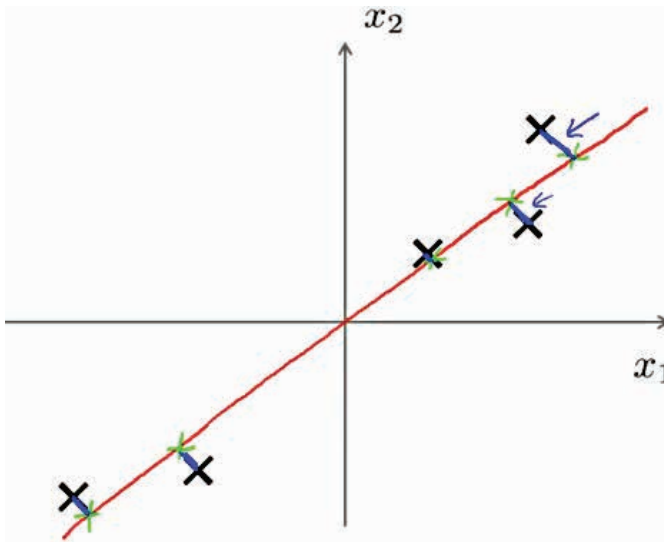
Figure 47. Dimensionality reduction (2D data)



Find a single line to project this data onto, in other words.

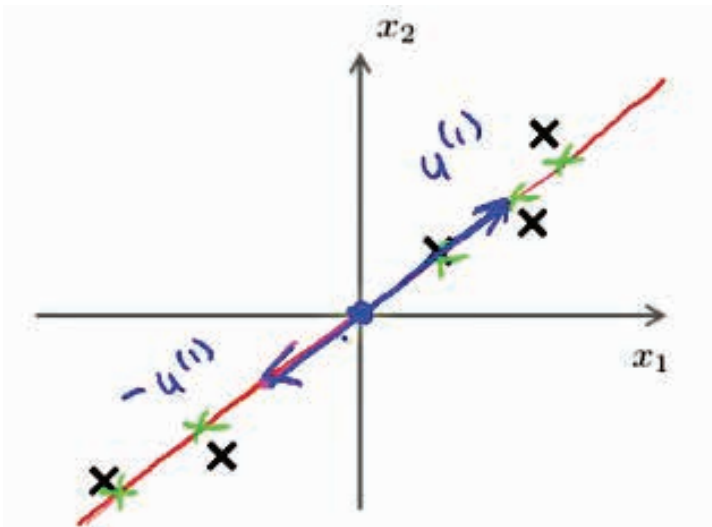
Where do we draw this line? Each point and the planned version should be close together (blue lines below are short). To reduce the sum of squares into a surface, PCA seeks out surfaces with lower dimensions. The projection mistake is another name for the blue lines.

PCA looks for the surface (in this case, a straight line) with the least amount of projection error.

Figure 48. dimensionality reduction (2D->1D data reduction)

An aside: Prior doing PCA, you should typically mean normalize and feature scale your data.

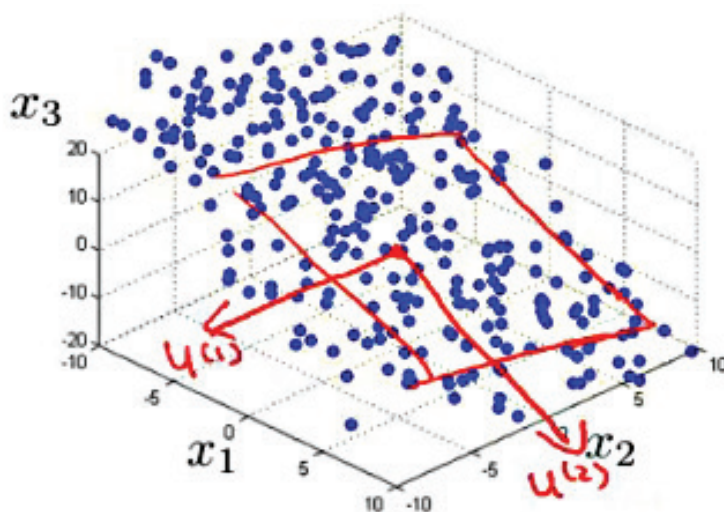
A more official explanation is Finding a vector $u(1)$ with a certain dimensionality onto which to project the data in 2D-1D is necessary in order to reduce the projection error.

Figure 49. Dimensionality reduction (2D->1D data reduction and data projection)

It makes no difference whether $u(1)$ is positive or negative ($-u(1)$). The identical red line is defined by each of the vectors. In a more typical scenario, we identify k vectors ($u(1), u(2), \dots, u(k)$) onto which to project the data in order to reduce the projection error from nD to kD . There are several vectors, then, onto which we project the data. Finding a set of vectors that spans a linear subspace will allow us to project the data onto that subspace.

We can define a point in a plane with k vectors e.g. $3D \rightarrow 2D$. Locate a pair of vectors that define the 2D plane (surface) that you'll be projecting your data onto. Like the "shallow box" example in compression, our goal is to build the shallowest box we can (by defining only two of its three dimensions, hence minimizing the box's depth).

Figure 50. Dimensionality reduction ($3D \rightarrow 2D$ data reduction and data projection)



Superficial similarities, PCA and linear regression are extremely distinct from one another. Fitting a straight line to minimize the straight line between a point and a squared line in linear regression. The size of the shortest orthogonal distance is reduced through PCA.

Produces incredibly diverse impacts. More specifically, we're attempting to forecast "y" using linear regression. There is no "y" in PCA; instead, we have a list of characteristics, and each feature is given the same consideration. If the features are handled symmetrically and the data is 3D to 2D **PCA Algorithm**.

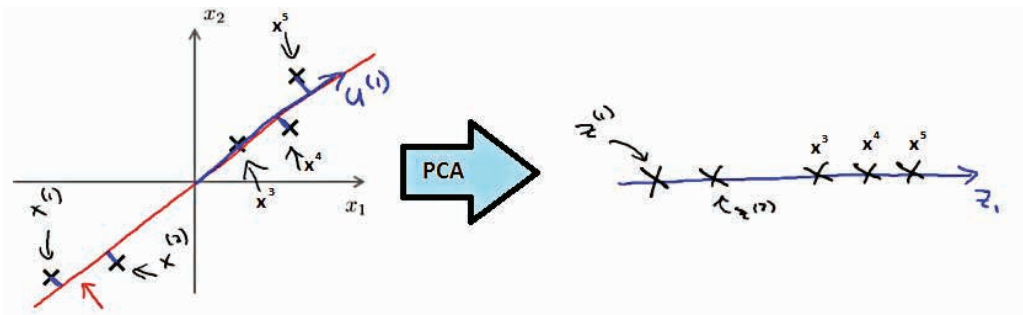
Data preprocessing must be done before PCA can be applied.

The first thing we must accomplish is for a given collection of m unlabeled instances.

1. Mean normalization, i.e. replace each x_{ji} with $x_{ji} - \mu_j$. In other words, determine the mean of each feature set, and then for each feature subtract the mean from the value, so we re-scale the mean to be 0
2. Feature scaling (depending on data). If features have very different scales, then scale so they all have a comparable range of values e.g. x_{ji} is set to $\frac{(x_{ji} - \mu_j)}{s_j}$, where s_j is some measure of the range, so could be biggest – smallest, standard deviation (more commonly).

With preprocessing done, PCA finds the lower dimensional sub-space which minimizes the sum of the square. In summary, for 2D->1D we'd be doing something like this.

Figure 51. Dimensionality reduction (3D->2D data reduction and data projection)



To get the new planes, you must compute the u vectors, and you also need to compute the z vectors. The newest feature vectors with lower dimensions are called z vectors. The u vectors' mathematical derivation is quite challenging. But after you've done it, finding each u vector is not a really difficult process.

Algorithm description

Reducing data from n -dimensional to k -dimensional and compute the covariance matrix

$$\Sigma = \frac{1}{m} \sum_{i=1}^n (x^{(i)})(x^{(i)})^T$$

This is commonly denoted as Σ (greek upper case sigma) - NOT summation symbol, Σ =sigma.

This is an $[n \times n]$ matrix. x_i is a $[n \times 1]$ matrix. Compute eigenvectors of matrix Σ
 $[U, S, V] = \text{svd}(\text{sigma})$

svd = singular value decomposition

More numerically stable than eig

eig = also gives eigenvector

U, S and V are matrices

U matrix is also an $[n \times n]$ matrix

Turns out the columns of U are the u vectors we want!

So to reduce a system from n -dimensions to k -dimensions. Just take the first k -vectors from U (first k columns)

$$U = \begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \dots & u^{(n)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n \times n}$$

Next, we need to find some way to change x (which is n dimensional) to z (which is k dimensional)

(reduce the dimensionality)

Take first k columns of the u matrix and stack in columns

$n \times k$ matrix - call this U_{reduce}

We calculate z as follows

$$z = (U_{\text{reduce}})^T * x$$

So $[k \times n] * [n \times 1]$

Generates a matrix which is

$k * 1$

Exactly the same as with supervised learning except we're now doing it with unlabeled data

So in summary

Preprocessing

Calculate sigma (covariance matrix)

Calculate eigenvectors with svd

Take k vectors from U ($U_{reduce} = U(:, 1:k)$)

Calculate z ($z = U_{reduce}^T * x$)

No mathematical derivation

Very complicated

But it works

Reconstruction from Compressed Representation

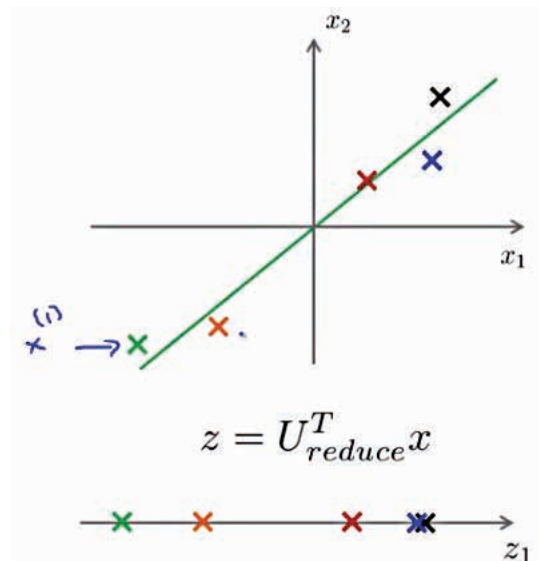
Earlier spoke about PCA as a compression algorithm

If this is the case, is there a way to decompress the data from low dimensionality back to a higher dimensionality format?

Reconstruction

Say we have an example as follows

Figure 52. Dimensionality reduction (2D->1D data reduction and data projection)



We have our examples (x_1, x_2 etc.)

Project onto z-surface

Given a point z_1 , how can we go back to the 2D space?

Considering

$$z(\text{vector}) = (\text{Ureduce})^T * x$$

To go in the opposite direction we must do

$$x_{\text{approx}} = \text{Ureduce} * z$$

To consider dimensions (and prove this really works)

$$\text{Ureduce} = [n \times k]$$

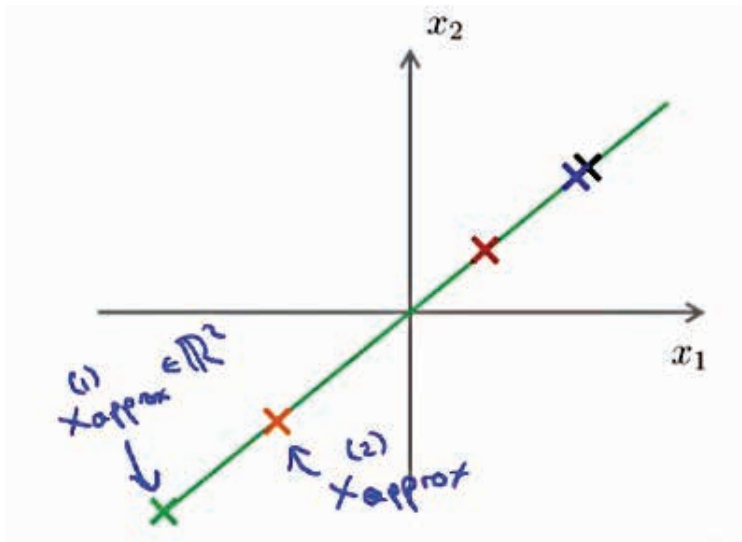
$$z[k \times 1]$$

So,

$$x_{\text{approx}} = [n \times 1]$$

So this creates the following representation

Figure 53. Dimensionality reduction (3D->2D data reduction and data projection)



We lose some of the information (i.e. everything is now perfectly on that line) but it is now projected into 2D space.

Choosing the number of Principle Components

How do we choose k ?

k = number of principle components

To choose k think about how PCA works.

PCA tries to minimize averaged squared projection error.

The average of data indicating how far away from the source the training examples are can be used to quantify total variation in data.

$$\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2$$

$$\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2$$

When we're choosing k typical to use something like this

$$\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} \leq 0.01 (1\%)$$

Ratio between averaged squared projection error with total variation in data.

Want ratio to be small - means we retain 99% of the variance.

If it's small (0) then this is because the numerator is small.

The numerator is small when $x^{(i)} = x_{approx}^{(i)}$, i.e. we lose very little information in the dimensionality reduction, so when we decompress we regenerate the same data. So, we choose k in terms of this ratio. Often can significantly reduce data dimensionality while retaining the variance. How is it possible to achieve this?

Algorithm:

Try PCA with $k=1$

Compute $U_{reduce}, z^{(1)}, z^{(2)}, \dots, z^{(m)}, x_{approx}^{(1)}, x_{approx}^{(m)}$

Check if
$$\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} \leq 0.01?$$

Advice for Applying PCA

We can use PCA to speed up algorithm running time. Say you have a supervised learning problem. Input x and y . x is a 10 000 dimensional feature vector e.g. 100 x 100 images = 10 000 pixels. Such a huge feature vector will make the algorithm slow. With PCA we can reduce the dimensionality and make it tractable

How

1. Extract x , So, we now have an unlabeled training set
2. Apply PCA to x vectors So, we now have a reduced dimensional feature vector z
3. This gives you a new training set, Each vector can be re-associated with the label
4. Take the reduced dimensionality dataset and feed to a learning algorithm
Use y as labels and z as feature vector
5. If you have a new example map from higher dimensionality vector to lower dimensionality vector, then feed into learning algorithm. PCA maps one vector to a lower dimensionality vector $x \rightarrow z$ defined by PCA only on the training set.

PCA algorithm presents a good solution for data compression but is not a good solution to address over fitting. It is better to use regularization. PCA throws away some data without knowing what the values it's losing. Probably is OK if you're keeping most of the data, but if you're throwing away some crucial data is not a good solution. So, you have to go to like 95-99% variance retained.

So, here regularization will give you AT LEAST as good a way to solve over fitting.

ONLY if you have a reason to believe PCA will help should you then add PCA. PCA is easy enough to add on as a processing step.

Try without first!

Python Implementation

A set of observations of potentially correlated variables are transformed into a set of values of linearly uncorrelated variables using principal component analysis, which is essentially a statistical method. Each of the principle components was chosen to describe the majority of the variance that is still there, and they are all orthogonal to one another. The first principal component has the highest variance among all principal components.

Uses of PCA:

- It is used to interpret and visualize data.
- It is used to discover relationships between variables in the data.
- As the number of variables decreases, further analysis becomes easier.
- It is widely used to show how genetic distance and population relatedness affect each other. Basically, a square symmetric matrix is used for them. A covariance matrix, correlation matrix, or a purely sums of squares and cross-products matrix can all be used.

If the individual variance varies widely, a correlation matrix is used.

Objectives of PCA:

- It is simply a non-dependent method in which it lowers attribute space from a big number of variables to a smaller number of components.
- PCA is essentially a process of dimension reduction, but there is no assurance that the reduced dimension will be comprehensible.
- The major aim in this PCA is to select a subset of variables from a wider collection, based on which original variables have the highest correlation with the principal amount.

Principal Axis Method: To extract the most variation from the variables, PCA essentially searches a linear combination of variables. Once this process is finished, it is taken out and another linear combination is looked for, one that provides an explanation for the greatest amount of remaining variance and, in essence, leads to orthogonal factors. With this approach, we examine overall variance.

Eigenvector: After matrix multiplication, it is a non-zero vector that continues to run in parallel. If Mx and x are parallel, let's assume that x is an eigenvector of dimension r of the matrix M with dimension $r \times r$. Then, to obtain the eigenvector and eigenvalues, we must solve $Mx = Ax$ where x and A are both unknown. Principal components exhibit both the common and unique variance of the variable, according to Eigen-Vectors. It is essentially a variance-focused method that aims to replicate overall variance and correlation across all components. The original variables' linear combinations, weighted by their contribution to understanding the variation in a specific orthogonal dimension, make up the main components.

Eigen Values: It is commonly referred to as distinctive roots. It essentially quantifies the variance across all variables that factor may account for. The eigenvalues ratio represents the explanatory power of the factors in relation to the variables. If the component is small, it will have a smaller impact on how variables are explained. In layman's terms, it calculates how much of the overall variation in the given database is accounted for by the factor. The total of the factor's squared factor loading for each variable allows us to determine the factor's eigenvalue. Now, Let's understand Principal Component Analysis with Python. To get the dataset used in the implementation.

Step 1: Importing the libraries

```
# importing required libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
# importing or loading the dataset
dataset=(178, 14) ... = pd.read_csv('wine.csv')

# distributing the dataset into two components X and Y
X=(178, 13) ... = dataset=(178, 14) ... .iloc[:, 0:13].values
y=(178,) arr... = dataset=(178, 14) ... .iloc[:, 13].values
# Splitting the X and Y into the
# Training set and Testing set
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test=(36,) arra... = train_test_split(X, y, test_size = 0.2, random_state = 0)
# performing preprocessing part
from sklearn.preprocessing import StandardScaler
sc=StandardSc... = StandardScaler()

X_train=(142, 2) a... = sc=StandardSc... .fit_transform(X_train=(142, 2) a...)
X_test=(36, 2) ar... = sc=StandardSc... .transform(X_test=(36, 2) ar...)
# Applying PCA function on training
# and testing set of X component
from sklearn.decomposition import PCA

pca=PCA(n_comp... = PCA(n_components = 2)

X_train=(142, 2) a... = pca=PCA(n_comp... .fit_transform(X_train=(142, 2) a...)
X_test=(36, 2) ar... = pca=PCA(n_comp... .transform(X_test=(36, 2) ar...)
```

```

explained_variance = pca.explained_variance_ratio_
# Fitting Logistic Regression To the training set
from sklearn.linear_model import LogisticRegression

classifier = LogisticRegression(random_state = 0)
classifier.fit(X_train, y_train)
# Predicting the test set result using
# predict function under LogisticRegression
y_pred = classifier.predict(X_test)
# making confusion matrix between
# test set of Y and predicted value.
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred)
# Predicting the training set
# result through scatter plot
from matplotlib.colors import ListedColormap

X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1,
                               stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1,
                               stop = X_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
                                                  X2.ravel()]).T).reshape(X1.shape), alpha = 0.75,
             cmap = ListedColormap(('yellow', 'white', 'aquamarine')))

plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())

```

```

plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green', 'blue'))(i), label = j)

plt.title('Logistic Regression (Training set)')
plt.xlabel('PC1') # for xlabel
plt.ylabel('PC2') # for ylabel
plt.legend() # to show legend

# show scatter plot
plt.show()
# Visualising the Test set results through scatter plot
from matplotlib.colors import ListedColormap

X_set, y_set = X_test, y_test

X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1,
                               stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1,
                               stop = X_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
                                                  X2.ravel()]).T).reshape(X1.shape), alpha = 0.75,
             cmap = ListedColormap(('yellow', 'white', 'aquamarine')))

plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())

```

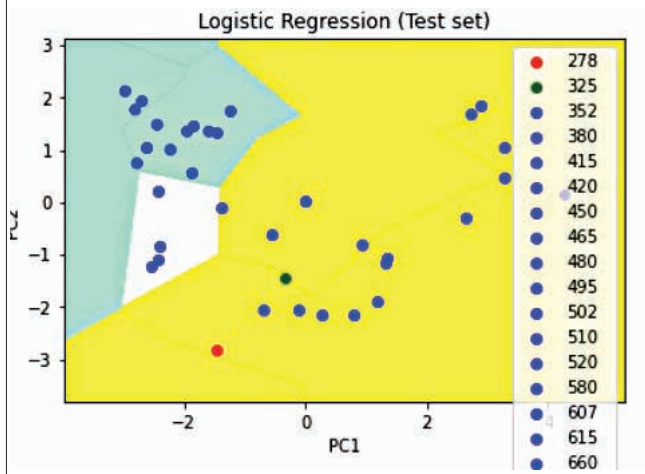
```

for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
               c = ListedColormap(('red', 'green', 'blue'))(i), label = j)

# title for scatter plot
plt.title('Logistic Regression (Test set)')
plt.xlabel('PC1') # for Xlabel
plt.ylabel('PC2') # for Ylabel
plt.legend()

# show scatter plot
plt.show()

```



Literature

1. <https://medium.com/machine-learning-algorithms-from-scratch/k-means-clustering-from-scratch-in-python-1675d38eee42>
2. <https://jakevdp.github.io/PythonDataScienceHandbook/>
3. Introduction to Linear Algebra, Fifth Edition (2016) by Gilbert Strang (gilstrang@gmail.com) ISBN : 978-09802327-7-6
4. Coursera's Machine Learning Notes — Week5, Neural Network — Lost Function, Forward-and-Backward Propagation | by Amber | Medium
5. Machine Learning | Coursera
6. The art of multiprocessor programming, Morgan Kaufmann (2008).
7. Distributed Systems, T.K.G.B.G. Coulouris (Pearson, 2012).

Figure source:

www.coursera.org

towardsdatascience.com

medium.com

Author's bio



Marina Marjanović

Marina Marjanović graduated from the Faculty of Electrical Engineering of Belgrade University, Serbia in 2002. In 2007 received PhD. diploma in the field of telecommunications (Signal Processing) with honors from the **Polytechnic University of Madrid**. During her PhD studies she was receiving a scholarship from the Spanish company “**Telefonica Moviles**”. From 2007-2013 she was working in the biggest Spanish company “Indra” on the development of equipment for Air Traffic Control Systems. From October 2013, she is Associate Professor at Faculty of Technical Sciences at “Singidunum”. She participated in several international funded projects.

Her areas of expertise are: Machine Learning, Deep Learning, Medical Image processing, Computer Vision.

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

001.891
004.85(035)
004.43PYTHON(035)

MARJANOVIĆ, Marina, 1976-

Introduction to Machine Learning : a Python approach to concepts, models and applications / Marina Marjanović. - 1st ed. - Begrade : Singidunum University, 2022 (Belgrade : BiroGraf). - 189 str. : ilustr. ; 24 cm

Na vrhu nasl. str.: Undergraduate Topics in Software and Data Engineering. - Autorkina slika. - Tiraž 400. - Author's bio: str. 189. - Bibliografija: str. 187.

ISBN 978-86-7912-788-4

a) Машинско учење -- Приручници b) Програмски језик "Python" -- Приручници

COBISS.SR-ID 108370697

© 2023

All rights reserved. No part of this work covered by the copyright herein may be reproduced, transmitted, stored or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, without the prior written permission of the publisher.



Marina Marjanović

Introduction to Machine Learning

**A Python Approach to
Concepts, Models and Applications**

This manual was created because of the need for appropriate written material from the subject Machine Learning, which the author holds in the third year of the Technical Faculty of Singidunum University in Belgrade. During writing, an effort has been made to make the textbook acceptable to readers who have a basic knowledge of the Python or Mat lab programming language. In one part of the textbook, a separate chapter will be devoted to the parts of mathematics (such as linear algebra and mathematical analysis) that are needed to understand algorithms. Through the chapters of this book, readers will learn the most effective machine learning techniques and gain experience in application using the Python programming language. In addition to machine learning theory, readers will learn how to apply these theories in solving new problems and some of Silicon Valley's best practices in innovation in machine learning and Artificial Intelligence.