Mathematical Foundations of Machine Learning

Lectures on YouTube:
https://www.youtube.com/@mathtalent

Seongjai Kim

Department of Mathematics and Statistics
Mississippi State University
Mississippi State, MS 39762 USA
Email: skim@math.msstate.edu

Updated: April 28, 2025

Seongjai Kim, Professor of Mathematics, Department of Mathematics and Statistics, Mississippi State University, Mississippi State, MS 39762 USA. Email: skim@math.msstate.edu.

Prologue

In organizing this lecture note, I am indebted by the following:

- S. RASCHKA AND V. MIRJALILI, *Python Machine Learning, 3rd Ed.*, 2019 [62].
- (Lecture note) http://fa.bianp.net/teaching/2018/eecs227at/, Dr. Fabian Pedregosa, UC Berkeley
- (Lecture note) Introduction To Machine Learning, Prof. David Sontag, MIT & NYU
- (Lecture note) Mathematical Foundations of Machine Learning, Dr. Justin Romberg, Geoigia Tech

This lecture note will grow up as time marches; various core algorithms, useful techniques, and interesting examples would be soon incorporated.

Seongjai Kim
April 28, 2025

Contents

Title ii

Prologue iii

Table of Contents ix

1 Introduction 1
- 1.1. Why and What in Machine Learning (ML)? . . . 2
  - 1.1.1. Inference problems . . . 2
  - 1.1.2. Modeling . . . 3
  - 1.1.3. Machine learning examples . . . 4
- 1.2. Three Different Types of ML . . . 5
  - 1.2.1. Supervised learning . . . 6
  - 1.2.2. Unsupervised learning . . . 7
  - 1.2.3. Reinforcement learning . . . 8
- 1.3. Issues in Machine Learning . . . 9
- 1.4. A Machine Learning Modelcode: Scikit-Learn Comparisons and Ensembling . . . 15

Exercises for Chapter 1 . . . 20

2 Python Basics 21
- 2.1. Why Python? . . . 22
- 2.2. Python Essentials in 30 Minutes . . . 25
- 2.3. Zeros of a Polynomial in Python . . . 31
- 2.4. Python Classes . . . 35

Exercises for Chapter 2 . . . 42

3 Simple Machine Learning Algorithms for Classification 43
- 3.1. Binary Classifiers – Artificial Neurons . . . 44
- 3.2. The Perceptron Algorithm . . . 46
  - 3.2.1. The perceptron: A formal definition . . . 46
  - 3.2.2. The perceptron learning rule . . . 47
  - 3.2.3. Problems with the perceptron algorithm . . . 52
- 3.3. Adaline: ADAptive LInear NEuron . . . 55
  - 3.3.1. The Adaline Algorithm . . . 55
  - 3.3.2. Feature Scaling and Stochastic Gradient Descent . . . 58

Exercises for Chapter 3 61

4 Gradient-based Methods for Optimization 63
4.1. Gradient Descent Method 64
4.1.1. The gradient descent method in 1D 67
4.1.2. The full gradient descent algorithm 69
4.1.3. Surrogate minimization: A unifying principle 73
4.2. Newton's Method 75
4.2.1. Derivation 75
4.2.2. Hessian and principal curvatures 78
4.3. Quasi-Newton Methods 80
4.4. The Stochastic Gradient Method 84
4.5. The Levenberg–Marquardt Algorithm, for Nonlinear Least-Squares Problems 89
4.5.1. The gradient descent method 90
4.5.2. The Gauss-Newton method 91
4.5.3. The Levenberg-Marquardt algorithm 92
Exercises for Chapter 4 94

5 Popular Machine Learning Classifiers 97
5.1. Logistic Sigmoid Function 99
5.1.1. The standard logistic sigmoid function 99
5.1.2. The logit function 101
5.2. Classification via Logistic Regression 103
5.2.1. The logistic cost function 104
5.2.2. Gradient descent learning for logistic regression 107
5.2.3. Regularization: bias-variance tradeoff 108
5.3. Support Vector Machine 110
5.3.1. Linear SVM 110
5.3.2. The method of Lagrange multipliers 112
5.3.3. Karush-Kuhn-Tucker conditions and Complementary slackness 115
5.3.4. The inseparable case: Soft-margin classification 120
5.3.5. Nonlinear SVM and kernel trick 124
5.3.6. Solving the dual problem with SMO 128
5.4. Decision Trees 130
5.4.1. Decision tree objective 131
5.4.2. Random forests: Multiple decision trees 135
5.5. k-Nearest Neighbors 137
Exercises for Chapter 5 139

6 Data Preprocessing in Machine Learning 141
6.1. General Remarks on Data Preprocessing 142
6.2. Dealing with Missing Data & Categorical Data 144

6.2.1. Handling missing data . . . 144
6.2.2. Handling categorical data . . . 145
6.3. Feature Scaling . . . 146
6.4. Feature Selection . . . 148
6.4.1. Selecting meaningful variables . . . 148
6.4.2. Sequential backward selection (SBS) . . . 150
6.4.3. Ridge regression vs. LASSO . . . 151
6.5. Feature Importance . . . 154
Exercises for Chapter 6 . . . 156

7 Feature Extraction: Data Compression **157**
7.1. Principal Component Analysis . . . 158
7.1.1. Computation of principal components . . . 159
7.1.2. Dimensionality reduction . . . 161
7.1.3. Explained variance . . . 163
7.2. Singular Value Decomposition . . . 164
7.2.1. Interpretation of the SVD . . . 166
7.2.2. Properties of the SVD . . . 170
7.2.3. Computation of the SVD . . . 173
7.2.4. Application of the SVD to image compression . . . 178
7.3. Linear Discriminant Analysis . . . 180
7.3.1. Fisher's LDA (classifier): two classes . . . 181
7.3.2. Fisher's LDA: the optimum projection . . . 183
7.3.3. LDA for Multiple Classes . . . 186
7.3.4. The LDA: Dimensionality Reduction . . . 194
7.4. Kernel Principal Component Analysis . . . 197
7.4.1. Principal components of the kernel PCA . . . 198
7.4.2. Computation of the kernel PCA . . . 201
Exercises for Chapter 7 . . . 204

8 Cluster Analysis **207**
8.1. Basics for Cluster Analysis . . . 208
8.1.1. Quality of clustering . . . 209
8.1.2. Types of clusters . . . 212
8.1.3. Types of clustering and Objective functions . . . 216
8.2. K-Means and K-Medoids Clustering . . . 219
8.2.1. The (basic) K-Means clustering . . . 219
8.2.2. Bisecting K-Means algorithm . . . 225
8.2.3. The K-Medoids algorithm . . . 229
8.2.4. CLARA and CLARANS . . . 230
8.3. Hierarchical Clustering . . . 232
8.3.1. Basics of AGNES and DIANA . . . 232

8.3.2. AGNES: Agglomerative clustering . . . 234
8.4. DBSCAN: Density-based Clustering . . . 239
8.5. Cluster Validation . . . 244
8.5.1. Basics of cluster validation . . . 244
8.5.2. Internal and external measures of cluster validity . . . 249
8.6. Self-Organizing Maps . . . 255
8.6.1. Basics of the SOM . . . 255
8.6.2. Kohonen SOM networks . . . 259
8.6.3. The SOM algorithm and its interpretation . . . 264
Exercises for Chapter 8 . . . 268

9 Neural Networks and Deep Learning 269
9.1. Basics for Deep Learning . . . 270
9.2. Neural Networks . . . 274
9.2.1. Sigmoid neural networks . . . 276
9.2.2. A simple network to classify handwritten digits . . . 278
9.3. Back-Propagation . . . 286
9.3.1. Notations . . . 286
9.3.2. The cost function . . . 288
9.3.3. The four fundamental equations behind the back-propagation . . . 289
9.4. Deep Learning: Convolutional Neural Networks . . . 293
9.4.1. Introducing convolutional networks . . . 294
9.4.2. CNNs, in practice . . . 300
Exercises for Chapter 9 . . . 303

10 Data Mining 305
10.1. Introduction to Data Mining . . . 306
10.2. Vectors and Matrices in Data Mining . . . 310
10.2.1. Two Examples . . . 310
10.2.2. Data compression: Low rank approximation . . . 314
10.3. Text Mining . . . 317
10.3.1. Vector space model: Preprocessing and query matching . . . 318
10.3.2. Latent Semantic Indexing . . . 324
10.4. Eigenvalue Methods in Data Mining . . . 327
10.4.1. Pagerank . . . 328
10.4.2. The Google matrix . . . 332
10.4.3. Solving the Pagerank equation . . . 334
Exercises for Chapter 10 . . . 337

11 Quadratic Programming 339
11.1. Equality Constrained Quadratic Programming . . . 340
11.2. Direct Solution for the KKT System . . . 345

11.2.1. Symmetric factorization . . . 345
11.2.2. Range-space approach . . . 347
11.2.3. Null-space approach . . . 348
11.3. Linear Iterative Methods . . . 349
11.3.1. Convergence theory . . . 350
11.3.2. Graph theory: Estimation of the spectral radius . . . 350
11.3.3. Eigenvalue locus theorem . . . 352
11.3.4. Regular splitting and M-matrices . . . 354
11.4. Iterative Solution of the KKT System . . . 355
11.4.1. Krylov subspace methods . . . 355
11.4.2. The transforming range-space iteration . . . 356
11.5. Active Set Strategies for Convex QP Problems . . . 357
11.5.1. Primal active set strategy . . . 358
11.6. Interior-point Methods . . . 360
11.7. Logarithmic Barriers . . . 362
Exercises for Chapter 11 . . . 365

A Appendix **367**
A.1. Optimization: Primal and Dual Problems . . . 368
A.1.1. The Lagrangian . . . 368
A.1.2. Lagrange Dual Problem . . . 370
A.2. Weak Duality, Strong Duality, and Complementary Slackness . . . 372
A.2.1. Weak Duality . . . 373
A.2.2. Strong Duality . . . 374
A.2.3. Complementary Slackness . . . 375
A.3. Geometric Interpretation of Duality . . . 376

P Projects **383**
P.1. Data Preparation for Heterogeneous Datasets . . . 384
P.2. Effective Preprocessing Technique for Filling Missing Data . . . 388
P.3. mCLESS . . . 390
P.3.1. Review: Simple classifiers . . . 391
P.3.2. The mCLESS classifier . . . 392
P.3.3. Feature expansion . . . 396
P.4. Noise-Removal and Classification . . . 401
P.5. Gaussian Sailing to Overcome Local Minima Problems . . . 407
P.6. Quasi-Newton Methods Using Partial Information of the Hessian . . . 409

Bibliography **411**

Index **417**

CHAPTER 1

Introduction

What are we “learning” in **Machine Learning (ML)**?

This is a hard question to which we can only give a somewhat fuzzy answer. But at a high enough level of abstraction, there are two answers:

- **Algorithms**, which solve some kinds of inference problems
- **Models** for datasets.

These answers are so abstract that they are probably completely unsatisfying. But let’s (start to) clear things up, by looking at some particular examples of “inference" and “modeling" problems.

Contents of Chapter 1

1.1. Why and What in Machine Learning (ML)? 2
1.2. Three Different Types of ML 5
1.3. Issues in Machine Learning 9
1.4. A Machine Learning Modelcode: Scikit-Learn Comparisons and Ensembling 15
Exercises for Chapter 1 20

1.1. Why and What in Machine Learning (ML)?

1.1.1. Inference problems

Definition 1.1. **Statistical inference** *is the process of using data analysis to deduce properties of an underlying probability distribution. Inferential statistical analysis infers properties of a population, for example* ***by testing hypotheses and deriving estimates****.*

Loosely speaking, inference problems take in data, then output some kind of decision or estimate. The output of a statistical inference is a statistical proposition; here are some common forms of statistical proposition.

- a point estimate
- an interval estimate
- a credible interval
- rejection of a hypothesis
- **classification** or **clustering** of the data points into discrete groups

Example 1.2. Inference algorithms can answer the following.

(a) Does this image have a tree in it?



(b) What words are in this picture?

(c) If I give you a recording of somebody speaking, can you produce text of what they are saying?

Remark 1.3. What does a machine learning algorithm do? ***Machine learning algorithms*** are not algorithms for performing inference. Rather, **they are algorithms for building inference algorithms from examples.** An inference algorithm takes a piece of data and outputs a decision (or a probability distribution over the decision space).

1.1.2. Modeling

A second type of problem associated with **machine learning** (ML) might be roughly described as:

> *Given a dataset, how can I **succinctly describe it** (in a quantitative, mathematical manner)?*

One example is **regression analysis**. Most models can be broken into two categories:

- **Geometric models**. The general problem is that we have example data points
$$\mathbf{x}_1,\ \mathbf{x}_2,\ \cdots,\ \mathbf{x}_N \in \mathbb{R}^D$$
and we want **to find some kind of geometric structure** that (approximately) describes them.

  Here is an example: given a set of vectors, what (low dimensional) subspace comes closest to containing them?
- **Probabilistic models**. The basic task here is **to find a probability distribution** that describes the dataset $\{\mathbf{x}_n\}$.

  The classical name for this problem is **density estimation** – given samples of a random variable, estimate its probability density function (pdf). This gets extremely tricky in high dimensions (large values of D) or when there are dependencies between the data points. Key to solving these problems is **choosing the right way to describe your probability model**.

Note: In both cases above, **having a concise model** can go a tremendous way towards analyzing the data.

- **As a rule, if you have a simple and accurate model, this is tremendously helping in solving inference problems, because there are fewer parameters to consider and/or estimate.**
- **The categories can either overlap with or complement each other.** It is often the case that the same model can be interpreted as a *geometric* model or a *probabilistic* model.

1.1.3. Machine learning examples

- **Classification**: from data to discrete classes
  - Spam filtering
  - Object detection (e.g., face)
  - Weather prediction (e.g., rain, snow, sun)
- **Regression**: predicting a numeric value
  - Stock market
  - Weather prediction (e.g., Temperature)
- **Ranking**: comparing items
  - Web search (keywords)
  - Given an image, find similar images
- **Collaborative Filtering** (e.g., Recommendation systems)
- **Clustering**: discovering structure in data
  - Clustering points or images
  - Clustering web search results
- **Embedding**: visualizing data
  - Embedding images, words
- **Structured prediction**: from data to discrete classes
  - Speech/image recognition
  - Natural language processing

1.2. Three Different Types of ML

Example 1.4. Three different types of ML:

- Supervised learning: classification, regression
- Unsupervised learning: clustering
- Reinforcement learning: chess engine

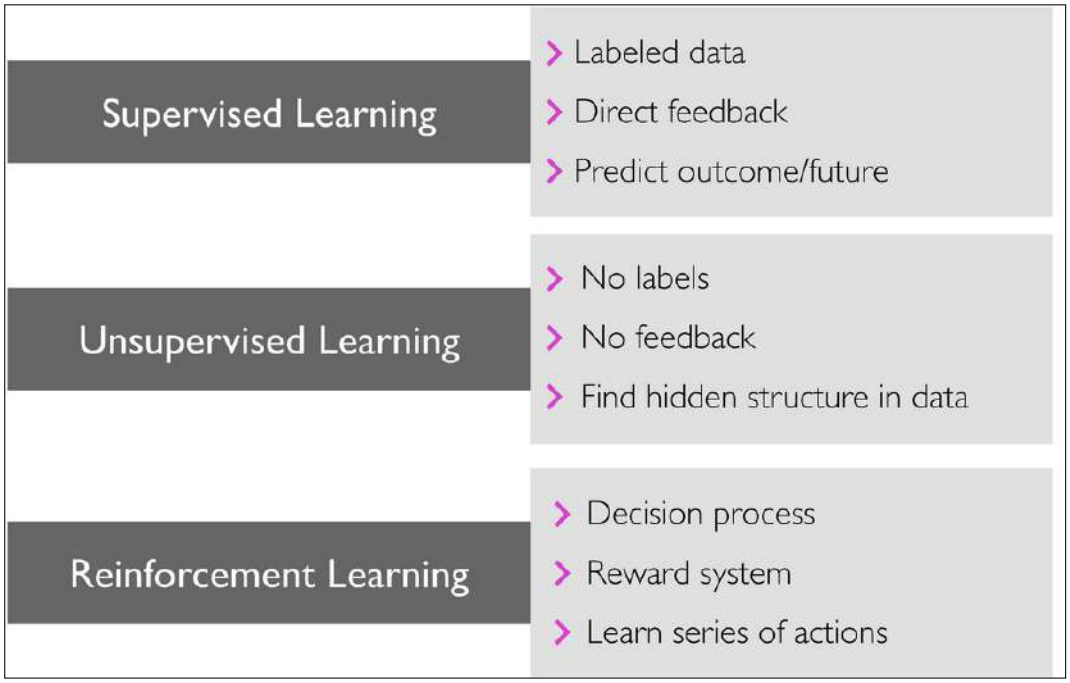


Figure 1.1: Three different types of ML (by methods)

1.2.1. Supervised learning

Assumption. Given a data set $\{(x_i, y_i)\}$, $\exists$ a relation $f : X \to Y$.

Supervised learning:

$$\begin{cases} \text{Given}: & \text{Training set } \{(x_i, y_i) \mid i = 1, \cdots, N\} \\ \text{Find}: & \widehat{f} : X \to Y, \text{ a good approximation to } f \end{cases} \tag{1.1}$$

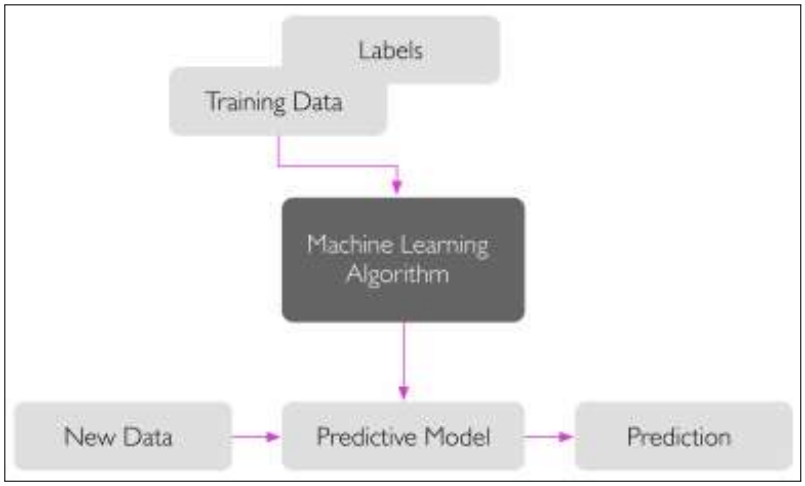


Figure 1.2: Supervised learning and prediction.

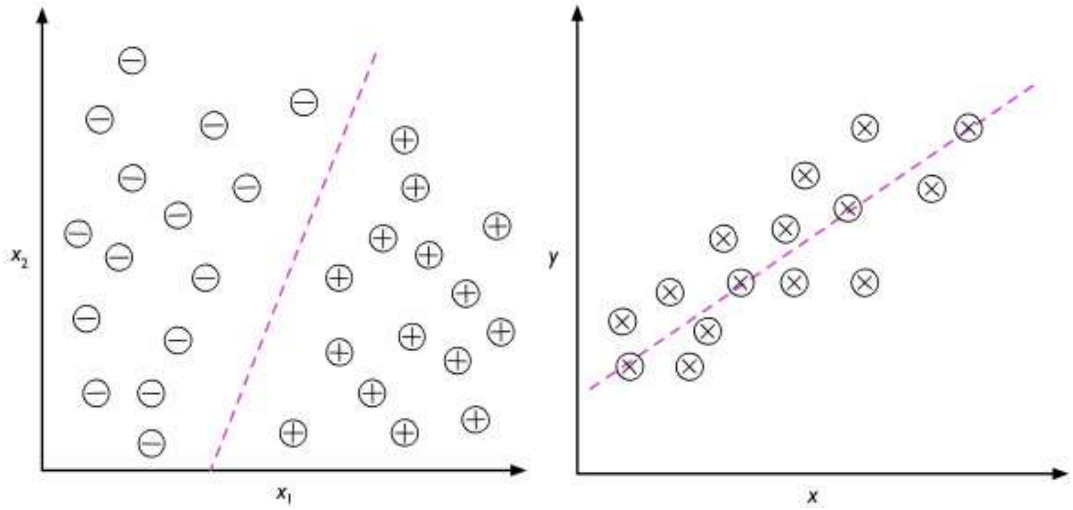


Figure 1.3: Classification and regression.

1.2.2. Unsupervised learning

Note:

- In supervised learning, we know the right answer beforehand when we train our model, and in reinforcement learning, we define a measure of reward for particular actions by the agent.
- In unsupervised learning, however, we are dealing with **unlabeled data** or **data of unknown structure**. Using unsupervised learning techniques, we are able **to explore the structure of our data** to **extract meaningful information** without the guidance of a known outcome variable or reward function.
- **Clustering** is an exploratory data analysis technique that allows us to organize a pile of information into **meaningful subgroups** (clusters) without having any prior knowledge of their group memberships.

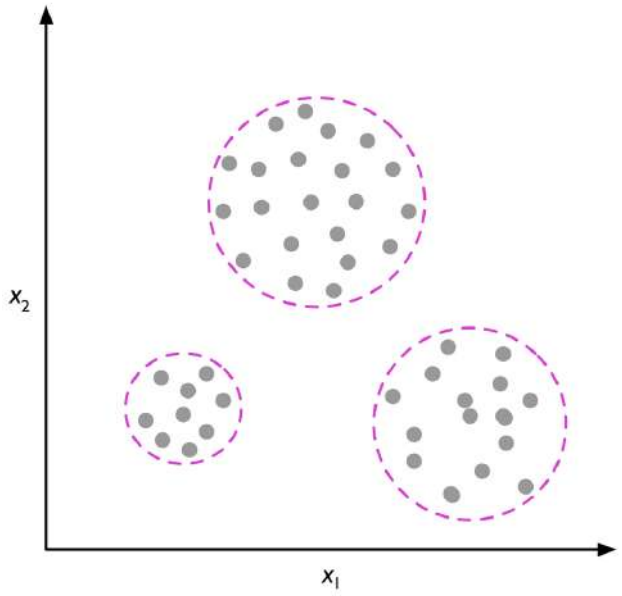


Figure 1.4: Clustering.

1.2.3. Reinforcement learning

Definition 1.5. **Reinforcement learning** (RL) is **the science of decision making**, combining **machine learning** and **optimal control**.

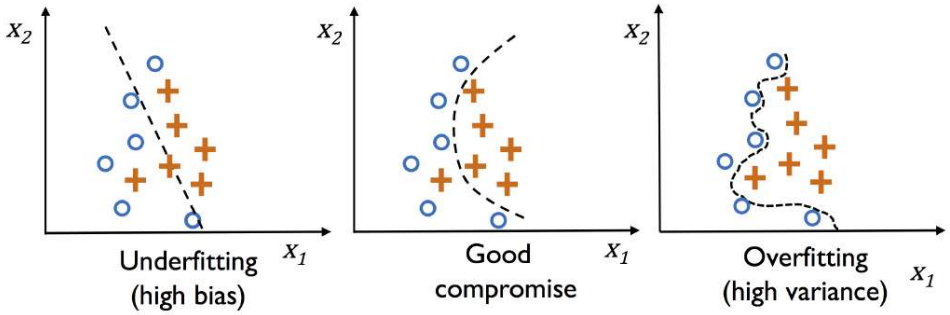
- It is about learning the optimal behavior in a dynamic environment in order to obtain maximum reward.
- This optimal behavior is learned through interactions with the environment and observations of how it responds.
- RL does not need labeled input/output pairs.
- In the absence of a supervisor, the learner must independently discover the sequence of actions that maximize the reward. This discovery process is similar to a trial-and-error search.
- **Examples**: AlphaGo, autonomous driving, robotics, ...

Key Points in RL

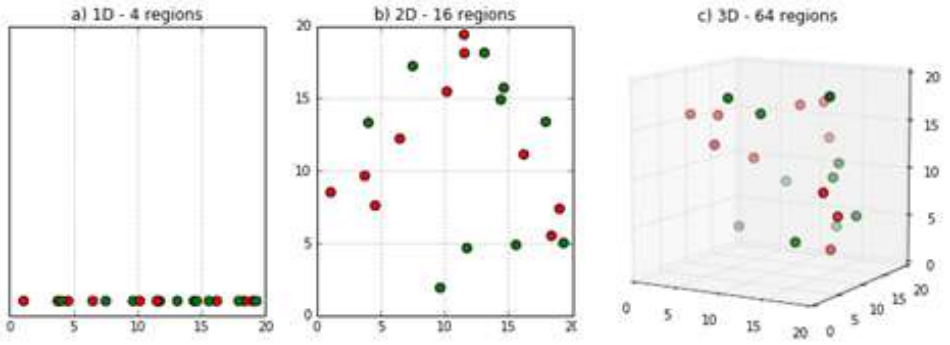
- **Input**: The input should be an **initial state** from which the model will start.
- **Output**: There are many possible outputs, as there are a variety of solutions to a particular problem
- **Training**: The training is based upon the input. The model will return a state; the user decides to reward or punish the model based on its output.
- The model keeps continue to learn.
- The best solution is decided based on the maximum reward.

1.3. Issues in Machine Learning

1. **Overfitting**: Fitting **training data** *too tightly*
   - **Difficulties**: Accuracy drops significantly for **test data**
   - **Remedies**:
     - More training data (often, impossible)
     - Early stopping; feature selection
     - Regularization; ensembling (multiple classifiers)



2. **Curse of Dimensionality**: **The feature space becomes increasingly sparse** for **an increasing number of dimensions** (of a fixed-size training dataset)
   - **Difficulties**: Larger error, more computation time;
     Data points **appear equidistant** from all the others
   - **Remedies**
     - More training data (often, impossible)
     - **Dimensionality reduction** (e.g., Feature selection, PCA)



3. **Multiple Local Minima Problem**
   Training often involves minimizing an objective function.

   - **Difficulties**: Larger error, unrepeatable
   - **Remedies**
     - Gaussian sailing; regularization
     - **Careful access to the data** (e.g., mini-batch)

4. **Interpretability**:
   Although ML has come very far, researchers still **don't know exactly how some algorithms (e.g., deep nets) work**.

   - If we don't know how training nets actually work, how do we make any **real progress**?

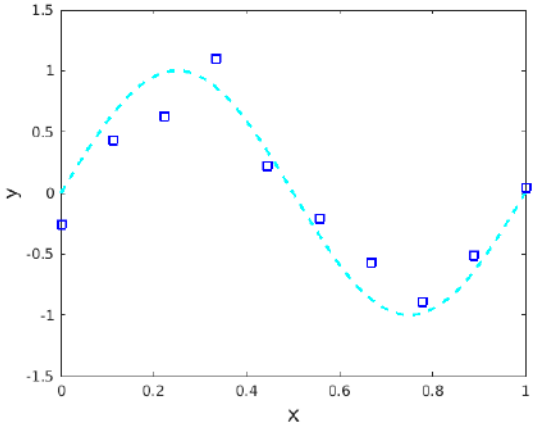
5. **One-Shot Learning**:
   We still haven't been able to achieve one-shot learning. ***Traditional gradient-based networks* need a huge amount of data**, and are often in the form of **extensive iterative training**.

   - Instead, we should find a way to **enable neural networks to learn, using just a few examples**.

Finding the Best Regression Model

Example 1.6. Consider a simple dataset: 10 points generated from a sine function, with noise.
Wanted: Find the best regression model for the dataset:

- Let's select a model from $\mathbb{P}_n$, polynomials of degree $\leq n$



Sine_Noisy_Data_Regression.m

```
close all; clear all

a=0; b=1; m=10;
f = @(t) sin(2*pi*t);
DATAFILE = 'sine-noisy-data.txt';
renew_data = 0;

%%-----------------------------------------------
if isfile(DATAFILE) && renew_data == 0
   DATA = readmatrix(DATAFILE);        % np.loadtxt()
else
   X = linspace(a,b,m); Y0 = f(X);
   noise = rand([1,m]); noise = noise-mean(noise(:));
   Y = Y0 + noise; DATA = [X',Y'];
   writematrix(DATA,DATAFILE);         % np.savetxt()
end

%%-----------------------------------------------
x = linspace(a,b,101); y = f(x);
x1 = DATA(:,1); y1 = DATA(:,2);
E = zeros(1,m);
for n = 0:m-1
    p = polyfit(x1,y1,n);              % np.polyfit()
    yhat = polyval(p,x1);              % np.polyval()
    E(n+1) = norm(y1-yhat,2)^2;
    %savefigure(x,y,x1,y1,polyval(p,x),n)
end

% figure
```

Which One is the Best?

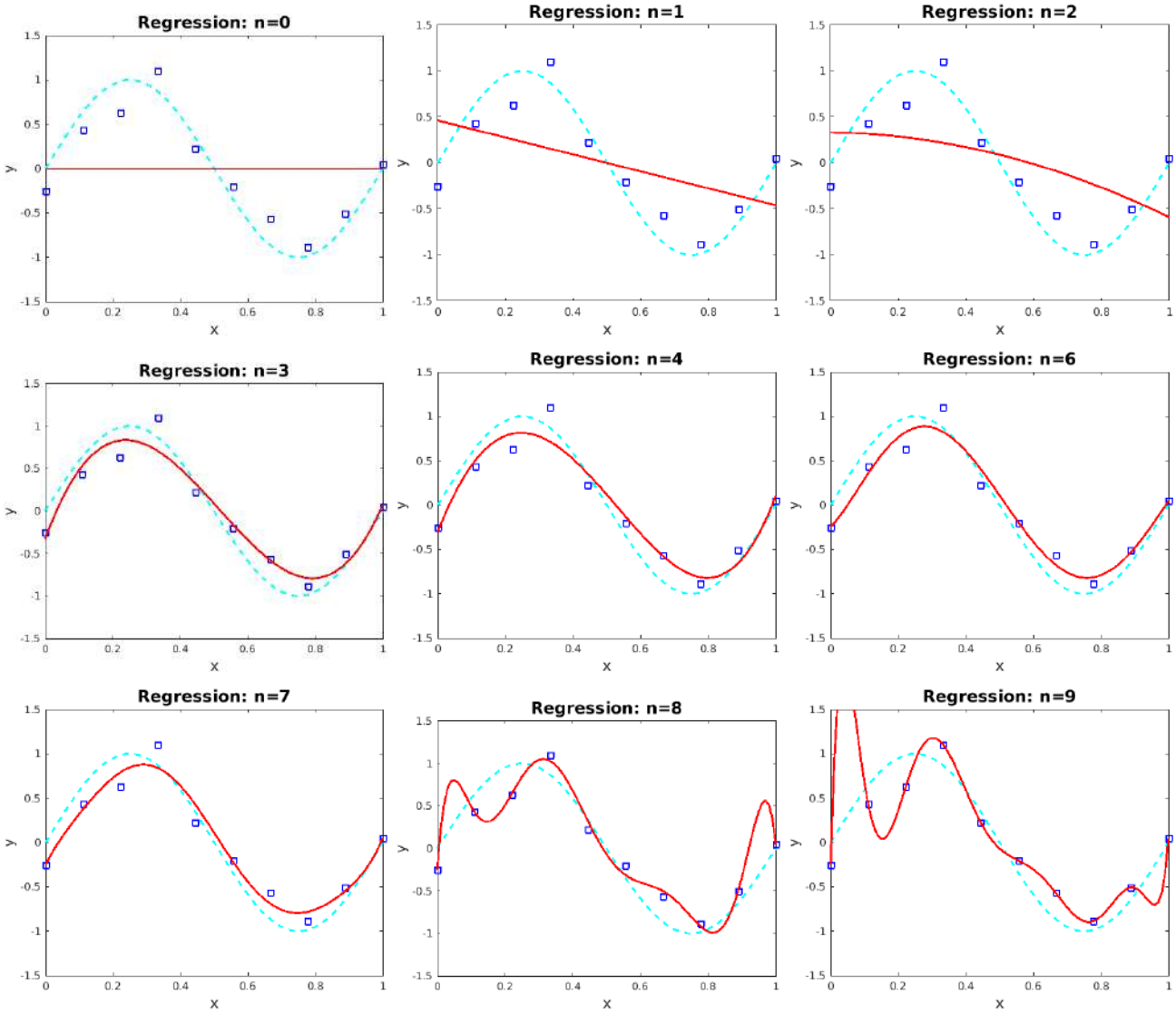


Figure 1.5: Regression models P_n, $n = 0, 1, \cdots, 9$.

Strategy 1.7. Given several models with similar explanatory ability, **the simplest is most likely to be the best choice.**

- Start simple, and only make the model more complex as needed.

The LS Error

Given the dataset $\{(x_i, y_i) \mid i = 1, 2, \cdots, m\}$ and the model P_n, define the LS-error

$$E_n = \sum_{i=1}^{m} \big(y_i - P_n(x_i)\big)^2, \qquad (m = 10), \tag{1.2}$$

which is also called the **mean square error**.

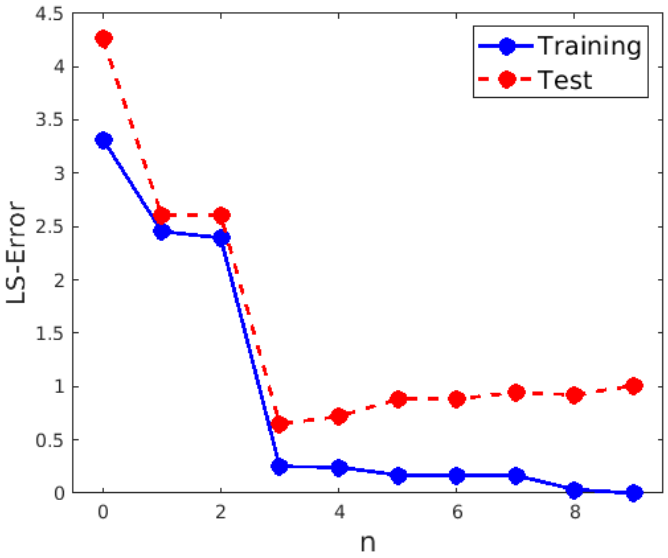


Figure 1.6: **The best choice is P_3, the third-order polynomial.**

Proposition 1.8. Occam's Razor Principle (a.k.a. Law of parsimony):

"One should not increase, beyond what is necessary,
the number of entities required to explain anything."

- William of Occam: A monk living in the 14-th century, England
- When **many** solutions are available for a given problem, we should select the **simplest** one.
- But what do we mean by **simple**?
- We will use **prior knowledge** of the problem to solve to define what is a simple solution (Example of a prior: smoothness).

Remark 1.9. Training and test performance. Assume that each training and test example–label pair (x, y) is drawn independently at random from the same (but unknown) population of examples and labels. Represent this population as a probability distribution $p(x, y)$, so that:

$$(x_i, y_i) \sim p(x, y).$$

- Then, given a loss function L:
  - Empirical (**training**) loss = $\frac{1}{N}\sum_{i=1}^{N} L(y_i, f(x_i))$.
    (Also called the **empirical risk**, $\widehat{R}(f, D_N)$.)
  - Expected (**test**) loss = $E_{(x,y)\sim p}\{L(y, f(x))\}$.
    (Also called the **risk** $R(f)$.)
- **Ideally, learning chooses the hypothesis that minimizes the risk.**
  - But this is impossible to compute!
  - The empirical risk is a good (unbiased) estimate of the risk (by linearity of expectation).
- **The principle of empirical risk minimization** reads

$$f^*(D_N) = \arg\min_{f\in\mathcal{F}} \widehat{R}(f, D_N). \tag{1.3}$$

1.4. A Machine Learning Modelcode: Scikit-Learn Comparisons and Ensembling

In machine learning, you can write a code **easily and effectively** using the following **modelcode**. It is also useful for **algorithm comparisons and ensembling**. You may download
https://skim.math.msstate.edu/LectureNotes/data/Machine-Learning-Modelcode.PY.tar.

Machine_Learning_Model.py

```
import numpy as np; import pandas as pd; import time
import seaborn as sbn; import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import datasets
np.set_printoptions(suppress=True)

#=====================================================================
# Upload a Dataset: print(dir(datasets))
# load_iris, load_wine, load_breast_cancer, ...
#=====================================================================
data_read = datasets.load_iris();  #print(data_read.keys())

X = data_read.data
y = data_read.target
dataname = data_read.filename
targets  = data_read.target_names
features = data_read.feature_names

#---------------------------------------------------------------------
# SETTING
#---------------------------------------------------------------------
N,d = X.shape; nclass=len(set(y));
print('DATA: N, d, nclass =',N,d,nclass)
rtrain = 0.7e0; run = 50; CompEnsm = 2;

def multi_run(clf,X,y,rtrain,run):
    t0 = time.time(); acc = np.zeros([run,1])
    for it in range(run):
        Xtrain, Xtest, ytrain, ytest = train_test_split(
            X, y, train_size=rtrain, random_state=it, stratify = y)
        clf.fit(Xtrain, ytrain);
        acc[it] = clf.score(Xtest, ytest)
    etime = time.time()-t0
    return np.mean(acc)*100, np.std(acc)*100, etime # accmean,acc_std,etime
```

```
35
36 #======================================================================
37 # My Classifier
38 #======================================================================
39 from myclf import *    # My Classifier = MyCLF()
40 if 'MyCLF' in locals():
41     accmean, acc_std, etime = multi_run(MyCLF(mode=1),X,y,rtrain,run)
42
43     print('%s: MyCLF()      : Acc.(mean,std) = (%.2f,%.2f)%%; E-time= %.5f'
44           %(dataname,accmean,acc_std,etime/run))
45
46 #======================================================================
47 # Scikit-learn Classifiers, for Comparisions && Ensembling
48 #======================================================================
49 if CompEnsm >= 1:
50     exec(open("sklearn_classifiers.py").read())
```

myclf.py

```
1  import numpy as np
2  from sklearn.base import BaseEstimator, ClassifierMixin
3  from sklearn.tree import DecisionTreeClassifier
4
5  class MyCLF(BaseEstimator, ClassifierMixin):        #a child class
6      def __init__(self, mode=0, learning_rate=0.01):
7          self.mode = mode
8          self.learning_rate = learning_rate
9          self.clf = DecisionTreeClassifier(max_depth=5)
10         if self.mode==1: print('MyCLF() = %s' %(self.clf))
11
12     def fit(self, X, y):
13         self.clf.fit(X, y)
14
15     def predict(self, X):
16         return self.clf.predict(X)
17
18     def score(self, X, y):
19         return self.clf.score(X, y)
```

Note: Replace `DecisionTreeClassifier()` with your own classier.

- The classifier must be implemented as **a child class** if if it is used in ensembling.

sklearn_classifiers.py

```
#=====================================================================
# Required: X, y, multi_run [dataname, rtrain, run, CompEnsm]
#=====================================================================
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.ensemble import VotingClassifier

#-----------------------------------------------
classifiers = [
    LogisticRegression(max_iter = 1000),
    KNeighborsClassifier(5),
    SVC(kernel="linear", C=0.5),
    SVC(gamma=2, C=1),
    RandomForestClassifier(max_depth=5, n_estimators=50, max_features=1),
    MLPClassifier(hidden_layer_sizes=[100], activation='logistic',
                  alpha=0.5, max_iter=1000),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis(),
    GaussianProcessClassifier(),
]
names = [
    "Logistic-Regr",
    "KNeighbors-5 ",
    "SVC-Linear   ",
    "SVC-RBF      ",
    "Random-Forest",
    "MLPClassifier",
    "AdaBoost     ",
    "Naive-Bayes  ",
    "QDA          ",
    "Gaussian-Proc",
]
```

```
#-------------------------------------------
if dataname is None: dataname = 'No-dataname';
if run      is None: run      = 50;
if rtrain   is None: rtrain   = 0.7e0;
if CompEnsm is None: CompEnsm = 2;

#=====================================================================
print('====== Comparision: Scikit-learn Classifiers ================')
#=====================================================================
import os;
acc_max=0; Acc_CLF = np.zeros([len(classifiers),1]);

for k, (name, clf) in enumerate(zip(names, classifiers)):
    accmean, acc_std, etime = multi_run(clf,X,y,rtrain,run)

    Acc_CLF[k] = accmean
    if accmean>acc_max: acc_max,algname = accmean,name
    print('%s: %s: Acc.(mean,std) = (%.2f,%.2f)%%; E-time= %.5f'
        %(os.path.basename(dataname),name,accmean,acc_std,etime/run))
print('---------------------------------------------------------------')
print('sklearn classifiers Acc: (mean,max) = (%.2f,%.2f)%%; Best = %s'
      %(np.mean(Acc_CLF),acc_max,algname))

if CompEnsm <2: quit()
#=====================================================================
print('====== Ensembling: SKlearn Classifiers ======================')
#=====================================================================
names = [x.rstrip() for x in names]
popped_clf = []
popped_clf.append(names.pop(9)); classifiers.pop(9);  #Gaussian Proc
popped_clf.append(names.pop(7)); classifiers.pop(7);  #Naive Bayes
popped_clf.append(names.pop(6)); classifiers.pop(6);  #AdaBoost
popped_clf.append(names.pop(4)); classifiers.pop(4);  #Random Forest
popped_clf.append(names.pop(0)); classifiers.pop(0);  #Logistic Regr
#print('popped_clf=',popped_clf[::-1])

CLFs = [(name, clf) for name, clf in zip(names, classifiers)]
#if 'MyCLF' in locals(): CLFs += [('MyCLF',MyCLF())]
EnCLF = VotingClassifier(estimators=CLFs, voting='hard')
accmean, acc_std, etime = multi_run(EnCLF,X,y,rtrain,run)

print('EnCLF =',[lis[0] for lis in CLFs])
print('%s: Ensemble CLFs: Acc.(mean,std) = (%.2f,%.2f)%%; E-time= %.5f'
       %(os.path.basename(dataname),accmean,acc_std,etime/run))
```

Output

```
DATA: N, d, nclass = 150 4 3
MyCLF() = DecisionTreeClassifier(max_depth=5)
iris.csv: MyCLF()       : Acc.(mean,std) = (94.53,3.12)%; E-time= 0.00074
====== Comparision: Scikit-learn Classifiers =================
iris.csv: Logistic-Regr: Acc.(mean,std) = (96.13,2.62)%; E-time= 0.01035
iris.csv: KNeighbors-5 : Acc.(mean,std) = (96.49,1.99)%; E-time= 0.00176
iris.csv: SVC-Linear   : Acc.(mean,std) = (97.60,2.26)%; E-time= 0.00085
iris.csv: SVC-RBF      : Acc.(mean,std) = (96.62,2.10)%; E-time= 0.00101
iris.csv: Random-Forest: Acc.(mean,std) = (94.84,3.16)%; E-time= 0.03647
iris.csv: MLPClassifier: Acc.(mean,std) = (98.58,1.32)%; E-time= 0.20549
iris.csv: AdaBoost     : Acc.(mean,std) = (94.40,2.64)%; E-time= 0.04119
iris.csv: Naive-Bayes  : Acc.(mean,std) = (95.11,3.20)%; E-time= 0.00090
iris.csv: QDA          : Acc.(mean,std) = (97.64,2.06)%; E-time= 0.00085
iris.csv: Gaussian-Proc: Acc.(mean,std) = (95.64,2.63)%; E-time= 0.16151
-------------------------------------------------------------
sklearn classifiers Acc: (mean,max) = (96.31,98.58)%; Best = MLPClassifier
====== Ensembling: SKlearn Classifiers ======================
EnCLF = ['KNeighbors-5', 'SVC-Linear', 'SVC-RBF', 'MLPClassifier', 'QDA']
iris.csv: Ensemble CLFs: Acc.(mean,std) = (97.60,1.98)%; E-time= 0.22272
```

Ensembling:
You may stack the best and its siblings of other options.

The exercise in the chapter is designed for you to

- install frequently-used machine learning packages in Python and
- run an example code, called a *modelcode*.

Exercises for Chapter 1

1.1. The modelcode in Section 1.4 will run without requiring any other implementation of yours.

(a) Download the code or save it by copy-and-paste.

(b) Install all imported packages to run the code.

(c) Modification:

- Select another dataset in `Machine_Learning_Model.py`, line 11.
  (You may use `print(dir(datasets))` to find datasets available.)
- Set different options for some of the classifiers in `sklearn_classifiers.py`, lines 20–30.

(d) Report the output.

Installation: If you are on Ubuntu, you may begin with

Install-Python-packages

```
sudo apt update
sudo apt install python3 -y
sudo apt install python3-pip -y
rehash
sudo pip3 install numpy scipy matplotlib sympy -y
sudo pip3 install sklearn seaborn pandas -y
```

CHAPTER 2

Python Basics

Contents of Chapter 2

2.1. Why Python? 22
2.2. Python Essentials in 30 Minutes 25
2.3. Zeros of a Polynomial in Python 31
2.4. Python Classes 35
Exercises for Chapter 2 42

2.1. Why Python?

Note: A good programming language must be **easy to learn and use** and **flexible and reliable**.

Advantages of Python

Python has the following characteristics.

- Easy to learn and use
- Flexible and reliable
- Extensively used in **Data Science**
- Handy for **Web Development** purposes
- Having **Vast Libraries** support
- Among the **fastest-growing** programming languages in the tech industry

Disadvantage of Python

Python is an interpreted and dynamically-typed language. The line-by-line execution of code, built with a high flexibility, most likely leads to **slow execution**. **Python scripts are way slow!**

Remark 2.1. Speed up Python Programs

- Use **numpy** and **scipy** for all mathematical operations.
- Always use **built-in functions** wherever possible.

- **Cython**: It is designed as **a C-extension for Python**, which is developed **for users not familiar with C**. **A good choice!**
- You may create and import your own **C/C++/Fortran-modules** into Python. If you extend Python with pieces of **compiled modules**, then the resulting code is easily **100× faster than Python scripts**. **The Best Choice!**

How to call C/C++/Fortran from Python

Functions in C/C++/Fortran can be compiled using the shell script.

Compile-f90-c-cpp

```
#!/usr/bin/bash

LIB_F90='lib_f90'
LIB_GCC='lib_gcc'
LIB_GPP='lib_gpp'

### Compiling: f90
f2py3 -c --f90flags='-O3' -m $LIB_F90 *.f90

### Compiling: C (PIC: position-independent code)
gcc -fPIC -O3 -shared -o $LIB_GCC.so *.c

### Compiling: C++
g++ -fPIC -O3 -shared -o $LIB_GPP.so *.cpp
```

The **shared objects** (*.so) can be imported to the **Python wrap-up**.

Python Wrap-up

```
#!/usr/bin/python3

import numpy as np
import ctypes, time
from lib_py3 import *
from lib_f90 import *
lib_gcc = ctypes.CDLL("./lib_gcc.so")
lib_gpp = ctypes.CDLL("./lib_gpp.so")

### For C/C++ -------------------------------------------
# e.g., lib_gcc.CFUNCTION(double array,double array,int,int)
#       returns a double value.
#-----------------------------------------------------------
IN_ddii = [np.ctypeslib.ndpointer(dtype=np.double),
   np.ctypeslib.ndpointer(dtype=np.double),
   ctypes.c_int, ctypes.c_int]  #input type
OUT_d = ctypes.c_double #output type

lib_gcc.CFUNCTION.argtypes = IN_ddii
lib_gcc.CFUNCTION.restype  = OUT_d

result = lib_gcc.CFUNCTION(x,y,n,m)
```

- The library numpy is designed for a **Matlab-like implementation**.
- Python can be used as a convenient **desktop calculator.**
  - First, set a startup environment
  - Use Python as a desktop calculator

~/.python_startup.py

```
#.bashrc: export PYTHONSTARTUP=~/.python_startup.py
#.cshrc:  setenv PYTHONSTARTUP ~/.python_startup.py
#------------------------------------------------
print("\t^[[1;33m~/.python_startup.py")

import numpy as np; import sympy as sym
import numpy.linalg as la; import matplotlib.pyplot as plt
print("\tnp=numpy; la=numpy.linalg; plt=matplotlib.pyplot; sym=sympy")

from numpy import zeros,ones
print("\tzeros,ones, from numpy")

import random
from sympy import *
x,y,z,t = symbols('x,y,z,t');
print("\tfrom sympy import *; x,y,z,t = symbols('x,y,z,t')")

print("\t^[[1;37mTo see details: dir() or dir(np)^[[m")
```

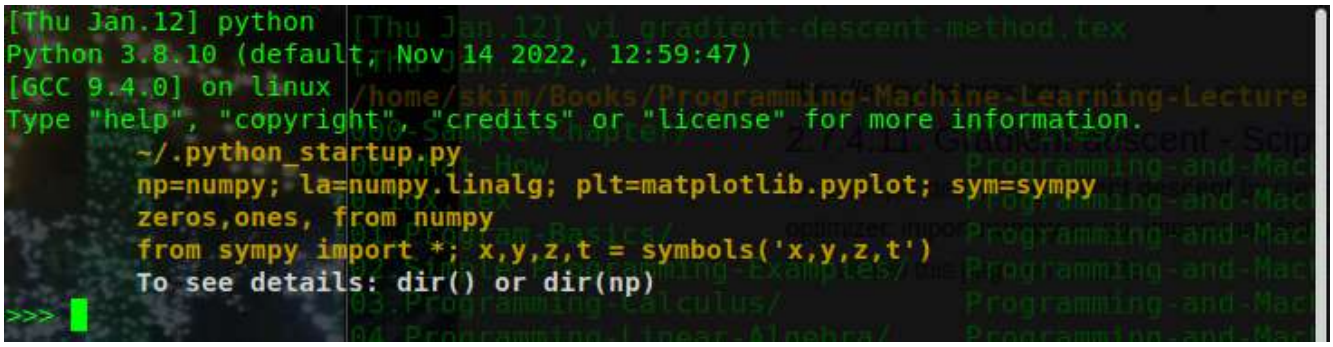


Figure 2.1: Python startup.

2.2. Python Essentials in 30 Minutes

Key Features of Python

- Python is a **simple, readable, open source** programming language which is easy to learn.
- It is an **interpreted** language, not a compiled language.
- In Python, **variables are untyped**; i.e., there is no need to define the data type of a variable while declaring it.
- Python supports **object-oriented programming** models.
- It is **platform-independent** and easily extensible and embeddable.
- It has a **huge standard library** with lots of modules and packages.
- Python is a **high level language** as it is easy to use because of simple syntax, **powerful** because of its rich libraries and extremely versatile.

Programming Features

- Python has **no support pointers**.
- Python codes are stored with **.py** extension.
- **Indentation**: Python uses indentation to define a block of code.
  - A **code block** (body of a function, loop, etc.) starts with indentation and ends with the first unindented line.
  - The amount of indentation is up to the user, but it must be consistent throughout that block.
- **Comments**:
  - The hash (#) symbol is used to start writing a comment.
  - **Multi-line comments**: Python uses triple quotes, either ''' or """.

Python Essentials

- **Sequence datatypes**: list, tuple, string
  - **[list]**: defined using square brackets (and commas)
    ```
    >>> li = ["abc", 14, 4.34, 23]
    ```
  - **(tuple)**: defined using parentheses (and commas)
    ```
    >>> tu = (23, (4,5), 'a', 4.1, -7)
    ```
  - **"string"**: defined using quotes (", ', or """)
    ```
    >>> st = 'Hello World'
    >>> st = "Hello World"
    >>> st = """This is a multi-line string
    ... that uses triple quotes."""
    ```
- **Retrieving elements**
  ```
  >>> li[0]
  'abc'
  >>> tu[1],tu[2],tu[-2]
  ((4, 5), 'a', 4.1)
  >>> st[25:36]
  'ng\nthat use'
  ```
- **Slicing**
  ```
  >>> tu[1:4] # be aware
  ((4, 5), 'a', 4.1)
  ```
- **The + and * operators**
  ```
  >>> [1, 2, 3]+[4, 5, 6,7]
  [1, 2, 3, 4, 5, 6, 7]
  >>> "Hello" + " " + 'World'
  Hello World
  >>> (1,2,3)*3
  (1, 2, 3, 1, 2, 3, 1, 2, 3)
  ```

- **Reference semantics**

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> b
[1, 2, 3, 4]
```

Be aware with copying lists and numpy arrays!

- **numpy, range, and iteration**

```
>>> range(8)
[0, 1, 2, 3, 4, 5, 6, 7]
>>> import numpy as np
>>> for k in range(np.size(li)):
...     li[k]
... <Enter>
'abc'
14
4.34
23
```

- **numpy array and deepcopy**

```
>>> from copy import deepcopy
>>> A = np.array([1,2,3])
>>> B = A
>>> C = deepcopy(A)
>>> A *= 4
>>> B
array([ 4, 8, 12])
>>> C
array([1, 2, 3])
```

Frequently used Python Rules

frequently_used_rules.py

```
## Multi-line statement
a =  1 + 2 + 3 + 4 + 5 +\
     6 + 7 + 8 + 9 + 10
b = (1 + 2 + 3 + 4 + 5 +
     6 + 7 + 8 + 9 + 10)  #inside (), [], or {}
print(a,b)
# Output: 55 55

## Multiple statements in a single line using ";"
a = 1; b = 2; c = 3

## Docstrings in Python
def double(num):
    """Function to double the value"""
    return 2*num
print(double.__doc__)
# Output: Function to double the value

## Assigning multiple values to multiple variables
a, b, c = 1, 2, "Hello"
## Swap
b, c = c, b
print(a,b,c)
# Output: 1 Hello 2

## Data types in Python
a = 5; b = 2.1
print("type of (a,b)", type(a), type(b))
# Output: type of (a,b) <class 'int'> <class 'float'>

## Python Set: 'set' object is not subscriptable
a = {5,2,3,1,4}; b = {1,2,2,3,3,3}
print("a=",a,"b=",b)
# Output: a= {1, 2, 3, 4, 5} b= {1, 2, 3}
```

```
35
36 ## Python Dictionary
37 d = {'key1':'value1', 'Seth':22, 'Alex':21}
38 print(d['key1'],d['Alex'],d['Seth'])
39 # Output: value1 21 22
40
41 ## Output Formatting
42 x = 5.1; y = 10
43 print('x = %d and y = %d' %(x,y))
44 print('x = %f and y = %d' %(x,y))
45 print('x = {} and y = {}'.format(x,y))
46 print('x = {1} and y = {0}'.format(x,y))
47 # Output: x = 5 and y = 10
48 #         x = 5.100000 and y = 10
49 #         x = 5.1 and y = 10
50 #         x = 10 and y = 5.1
51
52 print("x=",x,"y=",y, sep="#",end="&\n")
53 # Output: x=#5.1#y=#10&
54
55 ## Python Interactive Input
56 C = input('Enter any: ')
57 print(C)
58 # Output: Enter any: Starkville
59 #         Starkville
```

Looping and Functions

Example 2.2. Compose a Python function which returns cubes of natural numbers.

Solution.

get_cubes.py

```
1  def get_cubes(num):
2      cubes = []
3      for i in range(1,num+1):
4          value = i**3
5          cubes.append(value)
6      return cubes
7
8  if __name__ == '__main__':
9      num = input('Enter a natural number: ')
10     cubes = get_cubes(int(num))
11     print(cubes)
```

Remark 2.3. get_cubes.py

- Lines 8-11 are added for the function to be called directly. That is,

```
[Sun Nov.05] python get_cubes.py
Enter a natural number:  6
[1, 8, 27, 64, 125, 216]
```

- When get_cubes is called from another function, the last four lines will not be executed.

call_get_cubes.py

```
1  from get_cubes import *
2
3  cubes = get_cubes(8)
4  print(cubes)
```

Execusion

```
1  [Sun Nov.05] python call_get_cubes.py
2  [1, 8, 27, 64, 125, 216, 343, 512]
```

2.3. Zeros of a Polynomial in Python

In this section, we will implement **a Python code** for zeros of a polynomial and **compare it with a Matlab code**.

Recall: Let's begin with recalling how to find zeros of a polynomial.

- When the Newton's method is applied for finding an approximate zero of $P(x)$, the iteration reads

$$x_n = x_{n-1} - \frac{P(x_{n-1})}{P'(x_{n-1})}. \tag{2.1}$$

  Thus both $P(x)$ and $P'(x)$ must be evaluated in each iteration.

- **The derivative $P'(x)$ can be evaluated by using the Horner's method with the same efficiency.** Indeed, differentiating

$$P(x) = (x - x_0)Q(x) + P(x_0)$$

  reads

$$P'(x) = Q(x) + (x - x_0)Q'(x). \tag{2.2}$$

  Thus

$$P'(x_0) = Q(x_0). \tag{2.3}$$

  That is, the evaluation of Q at x_0 becomes the desired quantity $P'(x_0)$.

Example 2.4. Let $P(x) = x^4 - 4x^3 + 7x^2 - 5x - 2$. Use the Newton's method and the Horner's method to implement a code and find an approximate zero of P near 3.

Solution. First, let's try to use built-in functions.

zeros_of_poly_built_in.py

```
import numpy as np

coeff = [1, -4, 7, -5, -2]
P = np.poly1d(coeff)
Pder = np.polyder(P)

print(P)
print(Pder)
print(np.roots(P))
print(P(3), Pder(3))
```

Output

```
   4     3     2
1 x - 4 x + 7 x - 5 x - 2
   3      2
4 x - 12 x + 14 x - 5
[ 2. +0.j  1.1378411+1.52731225j  1.1378411-1.52731225j  -0.2756822+0.j ]
19 37
```

Observation 2.5. We will see:
Python programming is **as easy and simple as** Matlab programming.

- In particular, **numpy** is developed for **Matlab-like implementation, with enhanced convenience**.
- Numpy is used extensively in most of scientific Python packages: SciPy, Pandas, Matplotlib, scikit-learn, $\cdots$

Now, we implement **a code in Python** for Newton-Horner method to find an approximate zero of P near 3.

Zeros-Polynomials-Newton-Horner.py

```
def horner(A,x0):
    """ input:  A = [a_n,...,a_1,a_0]
        output: p,d = P(x0),DP(x0) = horner(A,x0) """
    n = len(A)
    p = A[0]; d = 0

    for i in range(1,n):
        d = p + x0*d
        p = A[i] +x0*p
    return p,d

def newton_horner(A,x0,tol,itmax):
    """ input:  A = [a_n,...,a_1,a_0]
        output: x: P(x)=0 """
    x=x0
    for it in range(1,itmax+1):
        p,d =  horner(A,x)
        h = -p/d;
        x = x + h;
        if(abs(h)<tol): break
    return x,it

if __name__ == '__main__':
    coeff = [1, -4, 7, -5, -2]; x0 = 3
    tol = 10**(-12); itmax = 1000
    x,it =newton_horner(coeff,x0,tol,itmax)
    print("newton_horner: x0=%g; x=%g, in %d iterations" %(x0,x,it))
```

Execution

```
[Sat Jul.23] python Zeros-Polynomials-Newton-Horner.py
newton_horner: x0=3; x=2, in 7 iterations
```

Note: The above Python code must be compared with the Matlab code.

horner.m

```
function [p,d] = horner(A,x0)
%   input:  A = [a_0,a_1,...,a_n]
%   output: p=P(x0), d=P'(x0)

n = size(A(:),1);
p = A(n); d=0;

for i = n-1:-1:1
    d = p + x0*d;
    p = A(i) +x0*p;
end
```

newton_horner.m

```
function [x,it] = newton_horner(A,x0,tol,itmax)
%   input:  A = [a_0,a_1,...,a_n]; x0: initial for P(x)=0
%   outpue: x: P(x)=0

x = x0;
for it=1:itmax
    [p,d] = horner(A,x);
    h = -p/d;
    x = x + h;
    if(abs(h)<tol), break; end
end
```

Call_newton_horner.m

```
a = [-2 -5 7 -4 1];
x0=3;
tol = 10^-12; itmax=1000;
[x,it] = newton_horner(a,x0,tol,itmax);
fprintf("  newton_horner: x0=%g; x=%g, in %d iterations\n",x0,x,it)
  Result:  newton_horner: x0=3; x=2, in 7 iterations
```

2.4. Python Classes

Remark 2.6. **Object-Oriented Programming (OOP)**
Classes are a key concept in the object-oriented programming.
Classes provide a means of bundling data and functionality together.

- A **class** is a user-defined template or prototype from which real-world objects are created.
- The major merit of using classes is on the **sharing mechanism** between functions/methods and objects.
  - **Initialization** and the **sharing boundaries** must be declared clearly and conveniently.
- A class tells us
  - what data an object should have,
  - what are the initial/default values of the data, and
  - what methods are associated with the object to take actions on the objects using their data.
- An object is an **instance** of a class, and creating an object from a class is called **instantiation**.

In the following, we would build a simple class, as Dr. Xu did in [82, Appendix B.5]; you will learn how to **initiate, refine, and use classes**.

Initiation of a Class

Polynomial_01.py

```
1  class Polynomial():
2      """A class of polynomials"""
3
4      def __init__(self,coefficient):
5          """Initialize coefficient attribute of a polynomial."""
6          self.coeff = coefficient
7
8      def degree(self):
9          """Find the degree of a polynomial"""
10         return len(self.coeff)-1
11
12 if __name__ == '__main__':
13     p2 = Polynomial([1,2,3])
14     print(p2.coeff)     # a variable; output: [1, 2, 3]
15     print(p2.degree())  # a method;   output: 2
```

- **Lines 1-2**: define a class called `Polynomial` with a docstring.
  - The parentheses in the class definition are empty because we create this class from scratch.
- **Lines 4-10**: define two functions, `__init__()` and `degree()`. A function in a class is called a **method**.
  - **The `__init__()` method** is a special method for initialization; it is called the `__init__()` **constructor**.
  - **The `self` Parameter and Its Sharing**
    - The `self` parameter is required and must come first before the other parameters in each method.
    - The variable `self.coeff` **(prefixed with `self`)** is **available** to every method and is **accessible** by any objects created from the class. (Variables prefixed with self are called **attributes**.)
    - We do not need to provide arguments for `self`.
- **Line 13**: The line `p2 = Polynomial([1,2,3])` creates an object `p2` (a polynomial $x^2 + 2x + 3$), by passing the coefficient list `[1,2,3]`.
  - When Python reads this line, it calls the method `__init__()` in the class `Polynomial` and creates the object named `p2` that represents this particular polynomial $x^2 + 2x + 3$.

Refinement of the Polynomial class

Polynomial_02.py

```
class Polynomial():
    """A class of polynomials"""

    count = 0     #Polynomial.count

    def __init__(self):
        """Initialize coefficient attribute of a polynomial."""
        self.coeff = [1]
        Polynomial.count += 1

    def __del__(self):
        """Delete a polynomial object"""
        Polynomial.count -= 1

    def degree(self):
        """Find the degree of a polynomial"""
        return len(self.coeff)-1

    def evaluate(self,x):
        """Evaluate a polynomial."""
        n = self.degree(); eval = []
        for xi in x:
            p = self.coeff[0]      #Horner's method
            for k in range(1,n+1): p = self.coeff[k]+ xi*p
            eval.append(p)
        return eval

if __name__ == '__main__':
    poly1 = Polynomial()
    print('poly1, default coefficients:', poly1.coeff)
    poly1.coeff = [1,2,-3]
    print('poly1, coefficients after reset:', poly1.coeff)
    print('poly1, degree:', poly1.degree())

    poly2 = Polynomial(); poly2.coeff = [1,2,3,4,-5]
    print('poly2, coefficients after reset:', poly2.coeff)
    print('poly2, degree:', poly2.degree())

    print('number of created polynomials:', Polynomial.count)
    del poly1
    print('number of polynomials after a deletion:', Polynomial.count)
    print('poly2.evaluate([-1,0,1,2]):',poly2.evaluate([-1,0,1,2]))
```

- **Line 4: (Global Variable)** The variable `count` is **a class attribute** of `Polynomial`.
  - It belongs to the class but not a particular object.
  - All objects of the class share this same variable (`Polynomial.count`).
- **Line 8: (Initialization)** Initializes the class attribute `self.coeff`.
  - Every object or class attribute in a class needs an initial value.
  - One can set a **default value** for an object attribute in the `__init__()` constructor; and we do not have to include a parameter for that attribute. See Lines 29 and 35.
- **Lines 11-13: (Deletion of Objects)** Define the `__del__()` method in the class for the deletion of objects. See Line 40.
  - `del` is a built-in function which deletes variables and objects.
- **Lines 19-28: (Add Methods)** Define another method called `evaluate`, which uses the *Horner's method*. See Example 2.4, p.32.

Output

```
1 poly1, default coefficients: [1]
2 poly1, coefficients after reset: [1, 2, -3]
3 poly1, degree: 2
4 poly2, coefficients after reset: [1, 2, 3, 4, -5]
5 poly2, degree: 4
6 number of created polynomials: 2
7 number of polynomials after a deletion: 1
8 poly2.evaluate([-1,0,1,2]): [-7, -5, 5, 47]
```

Inheritance

Note: If we want to write a class that is just *a specialized version of another class*, we do not need to write the class from scratch.

- We call the specialized class a **child class** and the other general class a **parent class**.
- The child class can inherit all the attributes and methods form the parent class.
  - It can also define its own special attributes and methods or even overrides methods of the parent class.

Classes can import functions implemented earlier, to define methods.

Classes.py

```
from util_Poly import *

class Polynomial():
    """A class of polynomials"""

    def __init__(self,coefficient):
        """Initialize coefficient attribute of a polynomial."""
        self.coeff = coefficient

    def degree(self):
        """Find the degree of a polynomial"""
        return len(self.coeff)-1

class Quadratic(Polynomial):
    """A class of quadratic polynomial"""

    def __init__(self,coefficient):
        """Initialize the coefficient attributes ."""
        super().__init__(coefficient)
        self.power_decrease = 1

    def roots(self):
        return roots_Quad(self.coeff,self.power_decrease)

    def degree(self):
        return 2
```

- **Line 1**: Imports functions implemented earlier.
- **Line 14**: We must include the name of the parent class in the parentheses of the definition of the child class (to indicate the parent-child relation for inheritance).
- **Line 19**: The `super()` function is to give an child object all the attributes defined in the parent class.
- **Line 20**: An additional child class attribute `self.power_decrease` is initialized.
- **Lines 22-23**: define a new method called `roots`, reusing a function implemented earlier.
- **Lines 25-26**: The method `degree()` overrides the parent's method.

util_Poly.py

```
1 def roots_Quad(coeff,power_decrease):
2     a,b,c = coeff
3     if power_decrease != 1:
4         a,c = c,a
5     discriminant = b**2-4*a*c
6     r1 = (-b+discriminant**0.5)/(2*a)
7     r2 = (-b-discriminant**0.5)/(2*a)
8     return [r1,r2]
```

call_Quadratic.py

```
1 from Classes import *
2
3 quad1 = Quadratic([2,-3,1])
4 print('quad1, roots:',quad1.roots())
5 quad1.power_decrease = 0
6 print('roots when power_decrease = 0:',quad1.roots())
```

Output

```
1 quad1, roots: [1.0, 0.5]
2 roots when power_decrease = 0: [2.0, 1.0]
```

Final Remarks on Python Implementation

- A proper **modularization** must precede implementation, as for other programming languages.
- Classes are used quite frequently.
  - You do not have to use classes for small projects.
- Try to use classes ***smartly***.
  Quite often, they add unnecessary complications and
  their methods are *hardly* applicable directly for other projects.
  - You may implement **stand-alone functions** to import.
  - This strategy enhances **reusability** of functions.
    For example, the function `roots_Quad` defined in `util_Poly.py` (page 40) can be used directly for other projects.
  - Afterwards, you will get **your own utility functions**; using them, you can complete various programming tasks effectively.

Programming Problems in Homework

- **Use Python.**
- You should report **your codes and results** as well.

Exercises for Chapter 2

2.1. Use nested `for` loops to assign entries of a 5×5 matrix A such that $A[i, j] = ij$.

2.2. The variable d is initially equal to 1. Use a `while` loop to keep dividing d by 2 until $d < 10^{-6}$.

(a) Determine how many divisions are made.
(b) Verify your result by algebraic derivation.

Note: A `while` loop has not been considered in the lecture. However, you can figure it out easily by yourself.

2.3. Write a function that takes as input a list of values and returns the largest value. Do this without using the Python `max()` function; you should combine a `for loop` and an `if statement`.

(a) Produce a random list of size 10-20 to verify your function.

2.4. Let $P_4(x) = 2x^4 - 5x^3 - 11x^2 + 20x + 10$. Solve the following.

(a) Plot P_4 over the interval $[-3, 4]$.
(b) Find all zeros of P_4, modifying `Zeros-Polynomials-Newton-Horner.py`, p.32.
(c) Add markers for the zeros to the plot.
(d) Find all roots of $P_4'(x) = 0$.
(e) Add markers for the zeros of P_4' to the plot.

***Hint*:** *For plotting, you may import: "*`import matplotlib.pyplot as plt`*" then use* `plt.plot()`*. You will see the Python plotting is quite similar to Matlab plotting.*

CHAPTER 3

Simple Machine Learning Algorithms for Classification

In this chapter, we will make use of one of the first algorithmically described machine learning algorithms for classification, the **perceptron** and **adaptive linear neurons** (*adaline*). We will start by implementing a perceptron step by step in Python and training it to classify different flower species in the Iris dataset.

Contents of Chapter 3

3.1. Binary Classifiers – Artificial Neurons 44
3.2. The Perceptron Algorithm 46
3.3. Adaline: ADAptive LInear NEuron 55
Exercises for Chapter 3 61

3.1. Binary Classifiers – Artificial Neurons

Definition 3.1. A **binary classifier** is a function which can decide whether or not an input vector belongs to some specific class (e.g., spam/ham).

- Binary classification often refers to those classification tasks that have two class labels. (**two-class classification**)
- It is **a type of linear classifier**, i.e. a classification algorithm that makes **its predictions based on a linear predictor function** combining a set of weights with the feature vector.
- Linear classifiers are **artificial neurons**.

Remark 3.2. Neurons are interconnected nerve cells that are involved in the processing and transmitting of chemical and electrical signals. Such a nerve cell can be described as a simple logic gate with binary outputs;

- multiple signals arrive at the dendrites,
- they are integrated into the cell body,
- and if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

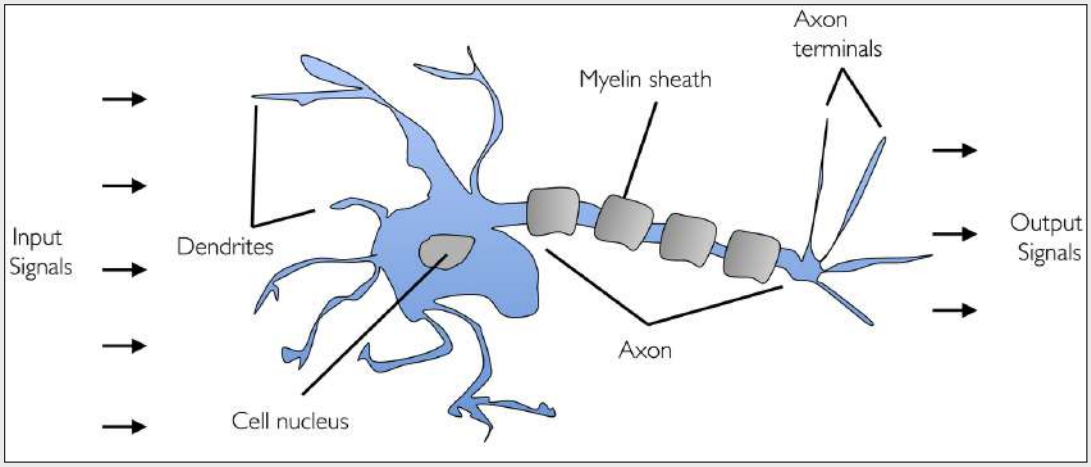


Figure 3.1: A schematic description of a neuron.

Linear classifiers

As artificial neurons, they have the following characteristics:

- Inputs are **feature values**: $\mathbf{x}$
- Each feature has a **weight**: $\mathbf{w}$
- Weighted sum (integration) is the **activation**

$$\text{activation}_{\mathbf{w}}(\mathbf{x}) = \sum_j w_j \, x_j = \mathbf{w} \cdot \mathbf{x} \tag{3.1}$$

- **Decision/output**: If the activation is

$$\begin{cases} \text{Positive} & \Rightarrow \quad \text{class 1} \\ \text{Negative} & \Rightarrow \quad \text{class 2} \end{cases}$$

Unknowns, in ML:

$$\begin{cases} \text{Training}: & \mathbf{w} \\ \text{Prediction}: & \text{activation}_{\mathbf{w}}(\mathbf{x}) \end{cases}$$

Examples:

- Perceptron
- Adaline (ADAptive LInear NEuron)
- Support Vector Machine (SVM) $\Rightarrow$ nonlinear decision boundaries, too

3.2. The Perceptron Algorithm

The **perceptron** is a binary classifier of supervised learning.

- 1957: Perceptron algorithm is invented by **Frank Rosenblatt**, Cornell Aeronautical Laboratory
  - Built on work of Hebbs (1949)
  - Improved by Widrow-Hoff (1960): Adaline
- 1960: Perceptron Mark 1 Computer – hardware implementation
- 1970's: Learning methods for two-layer neural networks

3.2.1. The perceptron: A formal definition

Definition 3.3. We can pose the **perceptron** as a **binary classifier**, in which we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity.

- **Input values**: $\mathbf{x} = (x_1, x_2, \cdots, x_m)^T$
- **Weight vector**: $\mathbf{w} = (w_1, w_2, \cdots, w_m)^T$
- **Net input**: $z = w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$
- **Activation function**: $\phi(z)$, defined by

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise}, \end{cases} \tag{3.2}$$

where θ is a threshold.

For simplicity, we can bring the threshold θ in (3.2) to the left side of the equation; define a weight-zero as $w_0 = -\theta$ and reformulate as

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise}, \end{cases} \quad z = \mathbf{w}^T \mathbf{x} = w_0 + w_1 x_1 + \cdots + w_m x_m. \tag{3.3}$$

In the ML literature, the variable w_0 is called the **bias**.
The equation $w_0 + w_1 x_1 + \cdots + w_m x_m = 0$ represents a hyperplane in $\mathbb{R}^m$, while w_0 decides the intercept.

3.2.2. The perceptron learning rule

The whole idea behind the **Rosenblatt's thresholded perceptron model** is to use a reductionist approach to mimic how a single neuron in the brain works: it either fires or it doesn't.

Algorithm 3.4. **Rosenblatt's Initial Perceptron Rule**

1. Initialize the weights to 0 or small random numbers.
2. For each training sample $\mathbf{x}^{(i)}$,
   (a) Compute the output value $\widehat{y}^{(i)}$ $(:= \phi(\mathbf{w}^T\mathbf{x}^{(i)}))$.
   (b) Update the weights.

The update of the weight vector w can be more formally written as:

$$\begin{aligned} \mathbf{w} &= \mathbf{w} + \Delta\mathbf{w}, \quad & \Delta\mathbf{w} &= \eta\,(y^{(i)} - \widehat{y}^{(i)})\,\mathbf{x}^{(i)}, \\ w_0 &= w_0 + \Delta w_0, & \Delta w_0 &= \eta\,(y^{(i)} - \widehat{y}^{(i)}), \end{aligned} \tag{3.4}$$

where η is the **learning rate**, $0 < \eta < 1$, $y^{(i)}$ is the true class label of the i-th training sample, and $\widehat{y}^{(i)}$ denotes the predicted class label.

Remark 3.5. A simple thought experiment for the perceptron learning rule:

- Let the perceptron predict the class label correctly. Then $y^{(i)} - \widehat{y}^{(i)} = 0$ so that the weights remain unchanged.
- Let the perceptron make a wrong prediction. Then

$$\Delta w_j = \eta\,(y^{(i)} - \widehat{y}^{(i)})\,x_j^{(i)} = \pm 2\,\eta\,x_j^{(i)}$$

  so that the weight w_j is pushed towards the direction of the positive or negative target class, respectively.

Note: It is important to note that **convergence of the perceptron** is only guaranteed if the two classes are **linearly separable** and the **learning rate is sufficiently small**. If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (epochs) and/or a threshold for the number of tolerated misclassifications.

Definition 3.6. ***(Linearly separable dataset).*** *A dataset* $\{(\mathbf{x}^{(i)}, y^{(i)})\}$ *is* ***linearly separable*** *if there exist* $\widehat{\mathbf{w}}$ *and* γ *such that*

$$y^{(i)}\, \widehat{\mathbf{w}}^T \mathbf{x}^{(i)} \geq \gamma > 0, \quad \forall\, i, \tag{3.5}$$

where γ *is called the* ***margin****.*

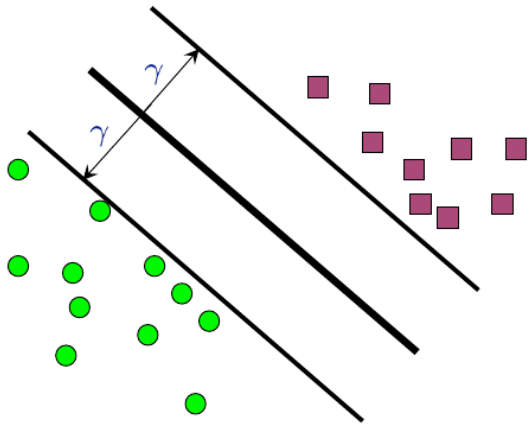


Figure 3.2: Linearly separable dataset.

Definition 3.7. ***(More formal/traditional definition).*** *Let* X *and* Y *be two sets of points in an* m*-dimensional Euclidean space. Then* X *and* Y *are* ***linearly separable*** *if there exist* $m+1$ *real numbers* $w_1, w_2, \cdots, w_m, k$ *such that every point* $\mathbf{x} \in X$ *satisfies* $\sum_{j=1}^{m} w_j x_j > k$ *and every point* $\mathbf{y} \in Y$ *satisfies* $\sum_{j=1}^{m} w_j y_j < k$.

Theorem 3.8. *Assume the data set $D = \{(\mathbf{x}^{(i)}, y^{(i)})\}$ is linearly separable with margin γ, i.e.,*

$$\exists\, \widehat{\mathbf{w}},\ \ \|\widehat{\mathbf{w}}\| = 1, \quad y^{(i)}\, \widehat{\mathbf{w}}^T \mathbf{x}^{(i)} \geq \gamma > 0,\ \ \forall\, i. \tag{3.6}$$

Suppose that $\|\mathbf{x}^{(i)}\| \leq R,\ \ \forall\, i$, for some $R > 0$. Then, the maximum number of mistakes made by the perceptron algorithm is bounded by R^2/γ^2.

Proof. Assume the perceptron algorithm makes *yet* a mistake for $(\mathbf{x}^{(\ell)}, y^{(\ell)})$. Then

$$\begin{aligned} \|\mathbf{w}^{(\ell+1)}\|^2 &= \|\mathbf{w}^{(\ell)} + \eta\,(y^{(\ell)} - \widehat{y}^{(\ell)})\mathbf{x}^{(\ell)}\|^2 \\ &= \|\mathbf{w}^{(\ell)}\|^2 + \|\eta\,(y^{(\ell)} - \widehat{y}^{(\ell)})\mathbf{x}^{(\ell)}\|^2 + 2\eta\,(y^{(\ell)} - \widehat{y}^{(\ell)}){\mathbf{w}^{(\ell)}}^T \mathbf{x}^{(\ell)} \\ &\leq \|\mathbf{w}^{(\ell)}\|^2 + \|\eta\,(y^{(\ell)} - \widehat{y}^{(\ell)})\mathbf{x}^{(\ell)}\|^2 \ \leq\ \|\mathbf{w}^{(\ell)}\|^2 + (2\eta\, R)^2, \end{aligned} \tag{3.7}$$

where we have used

$$(y^{(\ell)} - \widehat{y}^{(\ell)}){\mathbf{w}^{(\ell)}}^T \mathbf{x}^{(\ell)} \leq 0. \tag{3.8}$$

(See Exercise 1.) The inequality (3.7) implies

$$\|\mathbf{w}^{(\ell)}\|^2 \ \leq\ \ell \cdot (2\eta\, R)^2. \tag{3.9}$$

(Here we have used $\|\mathbf{w}^{(0)}\| = 0$.) On the other hand,

$$\widehat{\mathbf{w}}^T \mathbf{w}^{(\ell+1)} \ =\ \widehat{\mathbf{w}}^T \mathbf{w}^{(\ell)} + \eta\,(y^{(\ell)} - \widehat{y}^{(\ell)})\widehat{\mathbf{w}}^T \mathbf{x}^{(\ell)} \ \geq\ \widehat{\mathbf{w}}^T \mathbf{w}^{(\ell)} + 2\eta\,\gamma,$$

which implies

$$\widehat{\mathbf{w}}^T \mathbf{w}^{(\ell)} \ \geq\ \ell \cdot (2\eta\,\gamma) \tag{3.10}$$

and therefore

$$\|\mathbf{w}^{(\ell)}\|^2 \geq \ell^2 \cdot (2\eta\,\gamma)^2. \tag{3.11}$$

It follows from (3.9) and (3.11) that $\ell \leq R^2/\gamma^2$. □

Properties of the perceptron algorithm: For a linearly separable training dataset,

- **Convergence**: The perceptron will converge.
- **Separability**: Some weights get the training set perfectly correct.

Perceptron for Iris Dataset

perceptron.py

```
import numpy as np

class Perceptron():
    def __init__(self, xdim, epoch=10, learning_rate=0.01):
        self.epoch = epoch
        self.learning_rate = learning_rate
        self.weights = np.zeros(xdim + 1)

    def activate(self, x):
        net_input = np.dot(x,self.weights[1:])+self.weights[0]
        return 1 if (net_input > 0) else 0

    def fit(self, Xtrain, ytrain):
        for k in range(self.epoch):
            for x, y in zip(Xtrain, ytrain):
                yhat = self.activate(x)
                self.weights[1:] += self.learning_rate*(y-yhat)*x
                self.weights[0]  += self.learning_rate*(y-yhat)

    def predict(self, Xtest):
        yhat=[]
        #for x in Xtest: yhat.append(self.activate(x))
        [yhat.append(self.activate(x)) for x in Xtest]
        return yhat

    def score(self, Xtest, ytest):
        count=0;
        for x, y in zip(Xtest, ytest):
            if self.activate(x)==y: count+=1
        return count/len(ytest)

    #-----------------------------------------------------
    def fit_and_fig(self, Xtrain, ytrain):
        wgts_all = []
        for k in range(self.epoch):
            for x, y in zip(Xtrain, ytrain):
                yhat = self.activate(x)
                self.weights[1:] += self.learning_rate*(y-yhat)*x
                self.weights[0]  += self.learning_rate*(y-yhat)
                if k==0: wgts_all.append(list(self.weights))
        return np.array(wgts_all)
```

Iris_perceptron.py

```
import numpy as np; import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import datasets; #print(dir(datasets))
np.set_printoptions(suppress=True)
from perceptron import Perceptron

#-------------------------------------------------------------
data_read = datasets.load_iris();  #print(data_read.keys())
X = data_read.data;
y = data_read.target
targets = data_read.target_names; features = data_read.feature_names

N,d = X.shape; nclass=len(set(y));
print('N,d,nclass=',N,d,nclass)

#---- Take 2 classes in 2D ---------------------------------
X2 = X[y<=1]; y2 = y[y<=1];
X2 = X2[:,[0,2]]

#---- Train and Test ---------------------------------------
Xtrain, Xtest, ytrain, ytest = train_test_split(X2, y2,
        random_state=None, train_size=0.7e0)
clf = Perceptron(X2.shape[1],epoch=2)
#clf.fit(Xtrain, ytrain);
wgts_all = clf.fit_and_fig(Xtrain, ytrain);
accuracy = clf.score(Xtest, ytest); print('accuracy =', accuracy)
#yhat = clf.predict(Xtest);
```

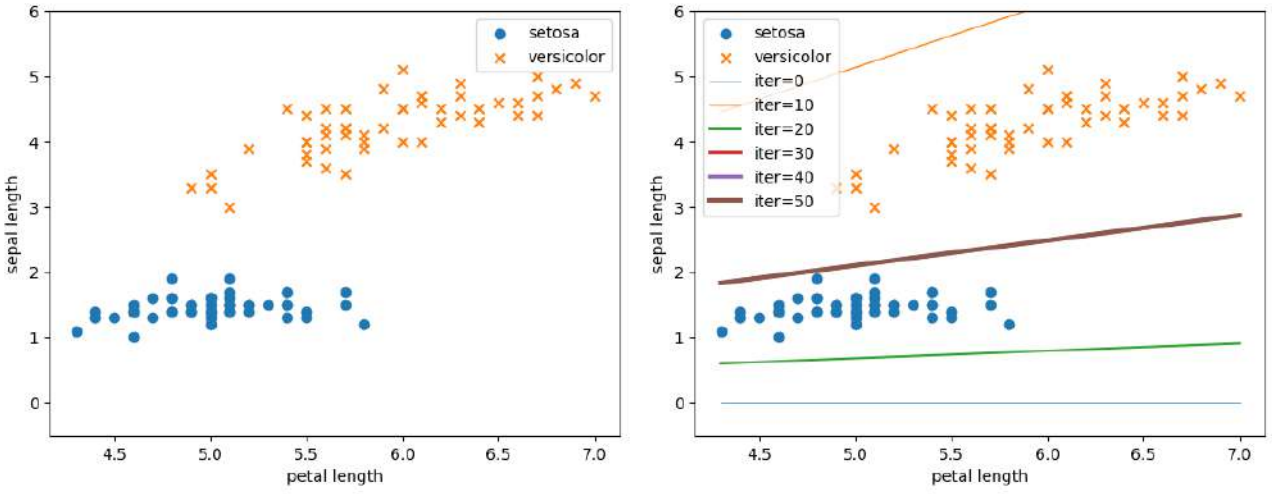
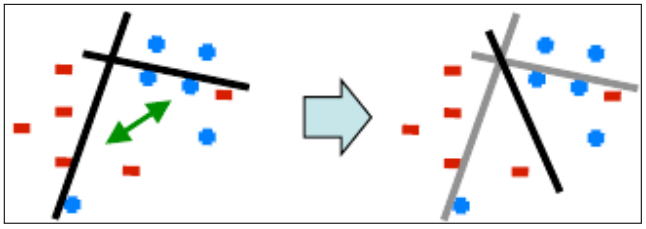


Figure 3.3: A part of Iris data (left) and the convergence of Perceptron iteration (right).

3.2.3. Problems with the perceptron algorithm

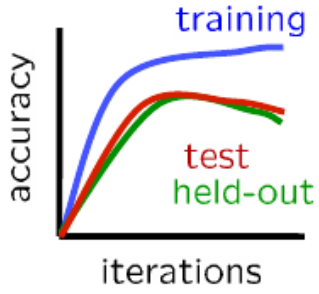
Inseparable Datasets

- If the data is **inseparable** (due to noise, for example), there is no guarantee for convergence or accuracy.
- **Averaged perceptron** is an algorithmic modification that helps with the issue.
  - Average the weight vectors, across all or a last part of iterations



Note: Frequently the training data ***is*** linearly separable! **Why?**

- For example, when the number of data points is much smaller than the number of features.
  - Perceptron can significantly **overfit** the data.
  - **An averaged perceptron** may help with this issue, too.



Definition 3.9. **Hold-out Method**: Hold-out is when you split up your dataset into a 'train' and 'test' set. The training set is what the model is trained on, and the test set is used to see how well that model performs on **unseen data.**

Optimal Separator?

Quesiton. Which of these **linear separators** is optimal?

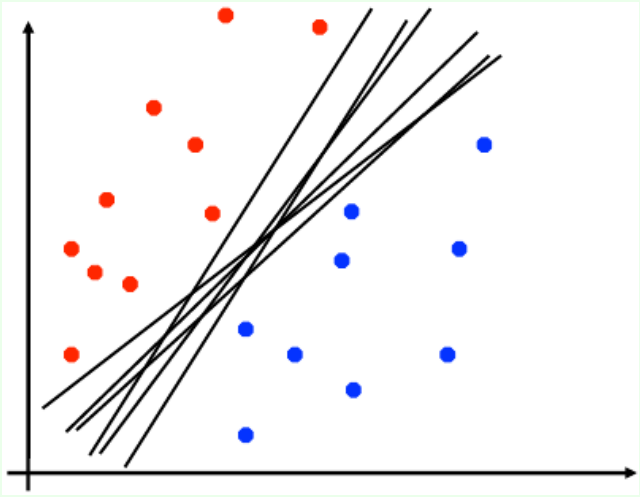


Figure 3.4

Example **3.10. Support Vector Machine** (Cortes & Vapnik, 1995) chooses the linear separator with the **largest margin**.

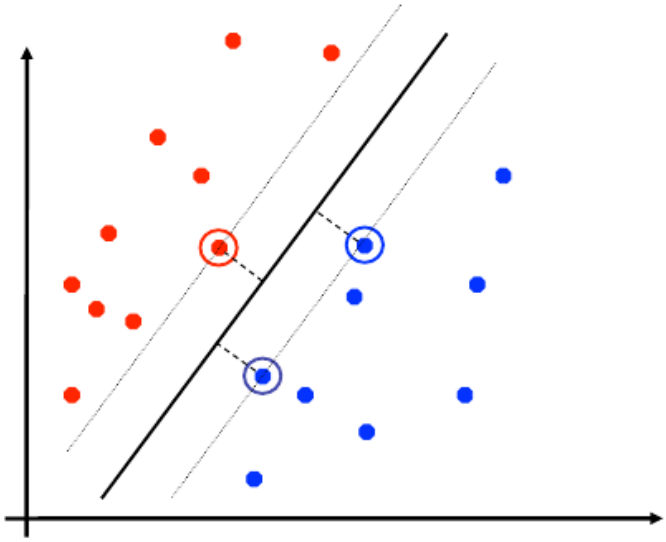


Figure 3.5

We will consider the SVM in Section 5.3.

How Multi-class Classification?

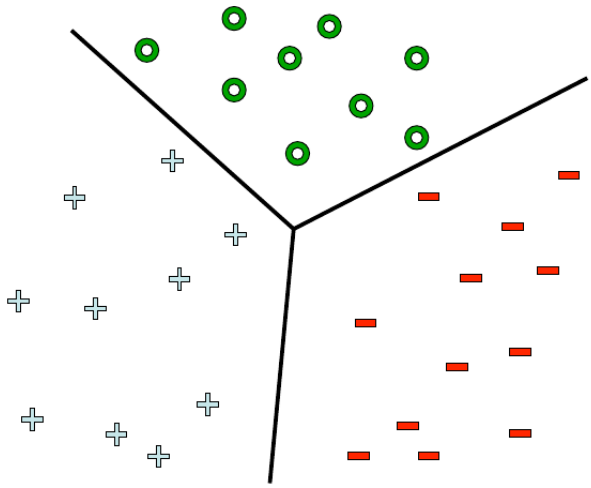


Figure 3.6: Classification for three classes.

One-versus-all (one-versus-rest) classification

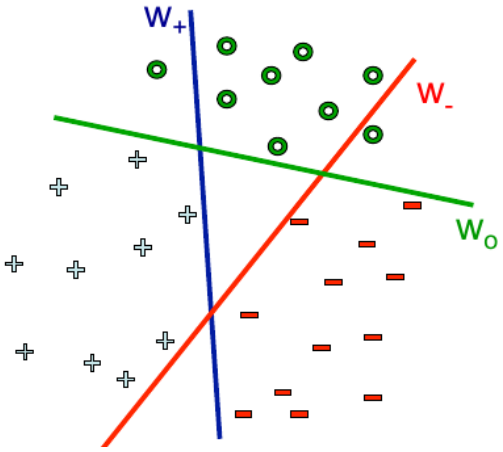


Figure 3.7: Three weights: $\mathbf{w}_-$, $\mathbf{w}_+$, and $\mathbf{w}_\circ$.

Learning: learn 3 classifiers

- $-$ vs $\{\circ, +\} \Rightarrow$ weights $\mathbf{w}_-$
- $+$ vs $\{\circ, -\} \Rightarrow$ weights $\mathbf{w}_+$
- $\circ$ vs $\{+, -\} \Rightarrow$ weights $\mathbf{w}_\circ$

Prediction: for a new data sample $\mathbf{x}$,

$$\widehat{y} = \arg \max_{i \in \{-,+,\circ\}} \phi(\mathbf{w}_i^T \mathbf{x}).$$

OVA (OVR) is readily applicable for classification of general n classes, $n \geq 2$.

3.3. Adaline: ADAptive LInear NEuron

3.3.1. The Adaline Algorithm

- (Widrow & Hoff, 1960)
- Weights are updated based on linear activation: e.g.,

$$\phi(\mathbf{w}^T\mathbf{x}) = \mathbf{w}^T\mathbf{x}.$$

  That is, ϕ is the **identity function**.
- Adaline algorithm is particularly interesting because it illustrates the key concept of defining and minimizing **continuous cost functions**, which will **lay the groundwork for understanding more advanced machine learning algorithms** for classification, such as logistic regression and support vector machines, as well as regression models.
- Continuous cost functions allow the ML optimization to incorporate **advanced mathematical techniques** such as **calculus**.

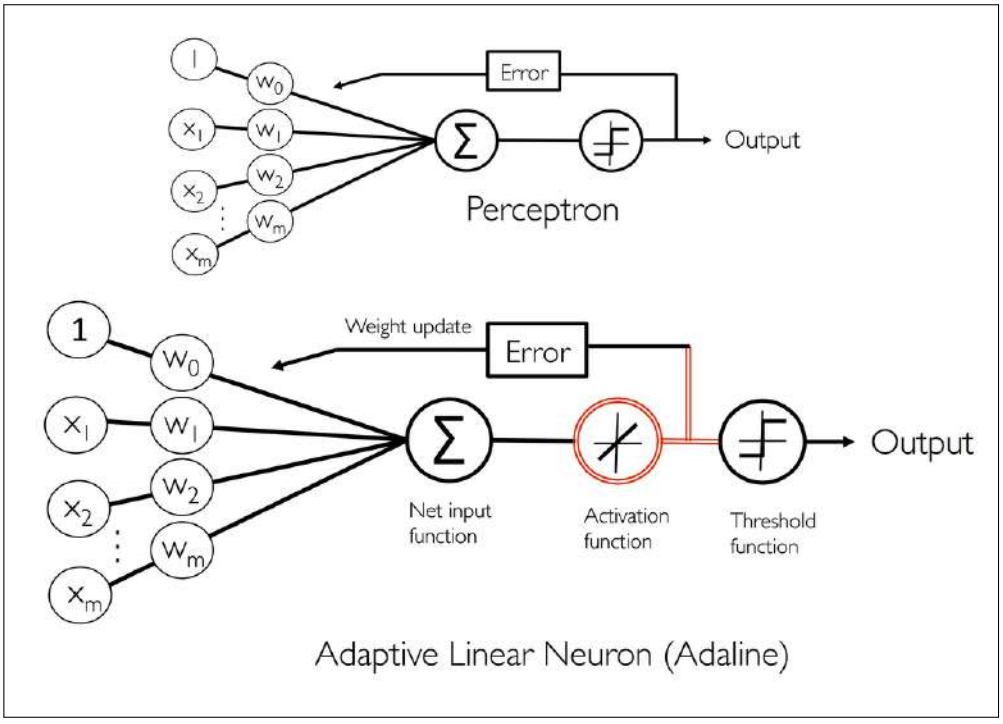


Figure 3.8: Perceptron vs. Adaline

Algorithm 3.11. Adaline Learning:
Given a dataset $\{(\mathbf{x}^{(i)}, y^{(i)}) \mid i = 1, 2, \cdots, N\}$, learn the weights $\mathbf{w}$ and bias $b = w_0$:

- **Activation function**: $\phi(z) = z$ (i.e., identity activation)
- **Cost function**: the SSE

$$\mathcal{J}(\mathbf{w}, b) = \frac{1}{2} \sum_{i=1}^{N} \left(y^{(i)} - \phi(z^{(i)}) \right)^2, \tag{3.12}$$

where $\boldsymbol{z^{(i)}} = \mathbf{w}^T \mathbf{x}^{(i)} + \boldsymbol{b}$ and $\phi = I$, the identity.

The dominant algorithm for the minimization of the cost function is the the Gradient Descent Method.

Algorithm 3.12. The Gradient Descent Method uses $-\nabla \mathcal{J}$ for the **search direction** (update direction):

$$\begin{array}{rcccl} \mathbf{w} & = & \mathbf{w} + \Delta \mathbf{w} & = & \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}, b), \\ b & = & b + \Delta b & = & b - \eta \nabla_b \mathcal{J}(\mathbf{w}, b), \end{array} \tag{3.13}$$

where $\eta > 0$ is the **step length** (learning rate).

Computation of $\nabla \mathcal{J}$ for Adaline:
The partial derivatives of the cost function $\mathcal{J}$ w.r.to w_j and b read

$$\begin{array}{rcl} \dfrac{\partial \mathcal{J}(\mathbf{w}, b)}{\partial w_j} & = & -\displaystyle\sum_i \left(y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)}, \\ \dfrac{\partial \mathcal{J}(\mathbf{w}, b)}{\partial b} & = & -\displaystyle\sum_i \left(y^{(i)} - \phi(z^{(i)}) \right). \end{array} \tag{3.14}$$

Thus, with $\phi = I$,

$$\begin{array}{rcccl} \Delta \mathbf{w} & = & -\eta \nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}, b) & = & \eta \displaystyle\sum_i \left(y^{(i)} - \phi(z^{(i)}) \right) \mathbf{x}^{(i)}, \\ \Delta b & = & -\eta \nabla_b \mathcal{J}(\mathbf{w}, b) & = & \eta \displaystyle\sum_i \left(y^{(i)} - \phi(z^{(i)}) \right). \end{array} \tag{3.15}$$

You will modify `perceptron.py` **for Adaline**; an implementation issue is considered in Exercise 3.4, p.61.

Convergence and Optimization Issues

- Depending on choices of certain **algorithmic parameters**, the gradient descent method may fail to converge to the the global minimizer.
- Data characteristics often determines both successability and speed of convergence; **data preprocessing** operations may improve convergence.
- For large-scale data, the gradient descent method is computationally expensive; a popular alternative is the **stochastic gradient descent method**.

Hyperparameters

Definition 3.13. In ML, a **hyperparameter** is a parameter whose value is set before the learning process begins. Thus it is an **algorithmic parameter**. Examples are

- The learning rate (η)
- The number of maximum epochs/iterations (`n_iter`)

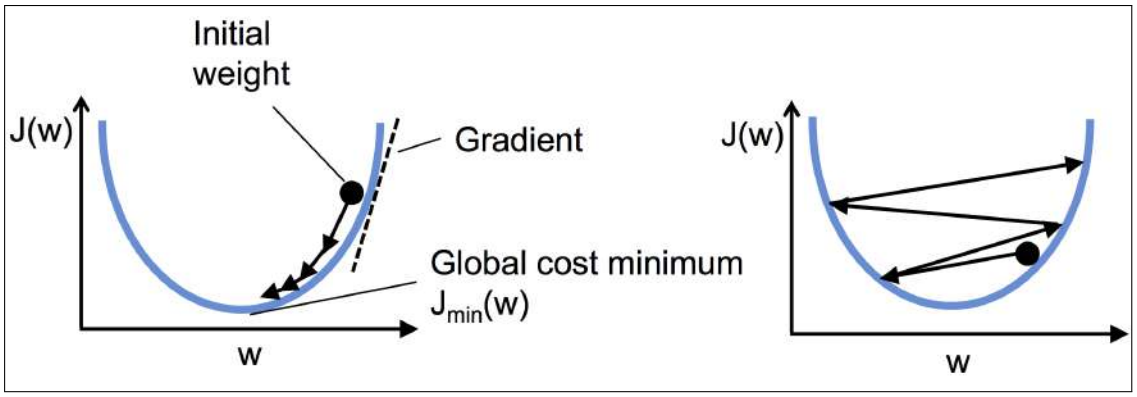


Figure 3.9: Well-chosen learning rate vs. a large learning rate

Hyperparameters must be selected to optimize the learning process:

- to converge **fast** to the global minimizer,
- avoiding overfit.

3.3.2. Feature Scaling and Stochastic Gradient Descent

Definition 3.14. **Feature Scaling Preprocessing**:
The gradient descent is one of the many algorithms that benefit from **feature scaling**. Here, we will consider a feature scaling method called **standardization**, which gives each feature of the data the property of a standard normal distribution.

- For example, to standardize the j-th feature, we simply need to subtract the sample mean μ_j from every training sample and divide it by its standard deviation σ_j:

$$\widetilde{x}_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j}. \tag{3.16}$$

Then,

$$\{\widetilde{x}_j^{(i)} \mid i = 1, 2, \cdots, n\} \sim \mathcal{N}(0, 1). \tag{3.17}$$

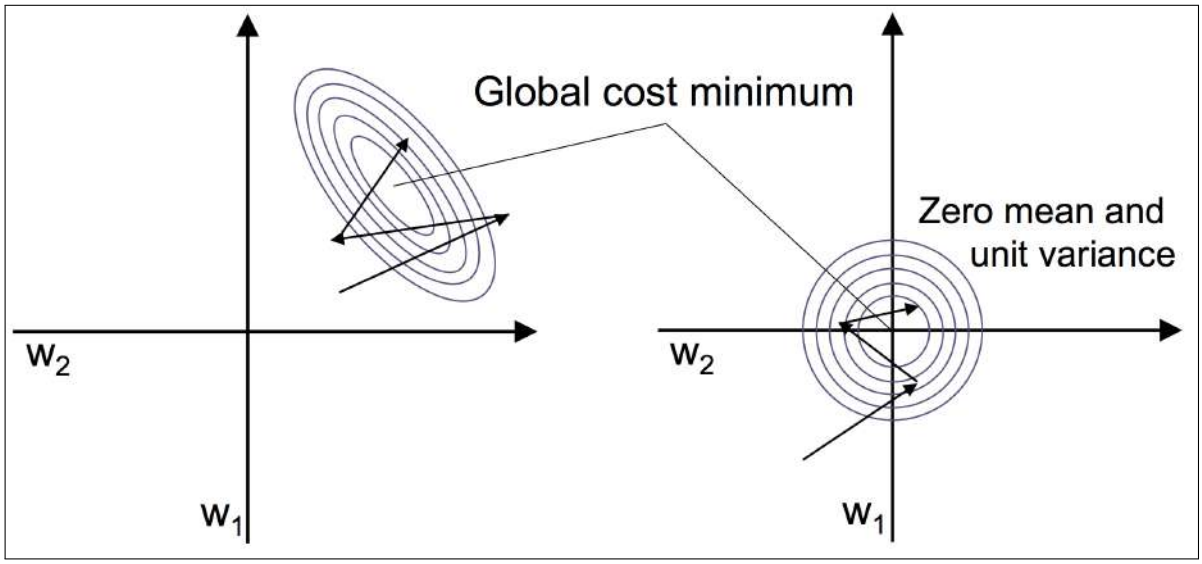


Figure 3.10: Standardization, which is one of **data normalization** techniques.

The gradient descent method has a tendency to converge faster with the standardized data.

Stochastic gradient descent method

Note: Earlier, we learned how to minimize a cost function with negative gradients that are calculated from the **whole training set**; this is why this approach is sometimes also referred to as **batch gradient descent**.

- Now imagine we have a very large dataset with millions of data points.
- Then, running with the gradient descent method can be computationally quite expensive, because we need to reevaluate the whole training dataset each time we take one step towards the global minimum.
- **A popular alternative** to the batch gradient descent algorithm is the **stochastic gradient descent** (**SGD**).

Algorithm 3.15. The SGD method updates the weights incrementally **for each training sample**:

$$\begin{array}{l}\text{Given a training set } D = \{(\mathbf{x}^{(i)}, y^{(i)}) \mid i = 1, 2, \cdots, n\} \\ \quad 1.\ \text{For } i = 1, 2, \cdots, n \\ \qquad \mathbf{w} = \mathbf{w} + \eta \left(y^{(i)} - \phi(\mathbf{w}^T \mathbf{x}^{(i)})\right) \mathbf{x}^{(i)}; \\ \quad 2.\ \text{If not convergent, shuffle } D \text{ and goto 1;}\end{array} \tag{3.18}$$

- The SGD method updates the weights based on a single training example.
- The SGD method typically reaches **convergence much faster** because of the **more frequent weight updates**.
- Since each search direction is calculated based on a single training example, the error surface is ***smoother*** (not noisier) than in the gradient descent method; **the SGD method can escape shallow local minima more readily.**
- To obtain accurate results via the SGD method, it is important **to present it with data in a random order**, which may prevent cycles with epochs.
- In the SGD method, the learning rate η is often set **adaptively**, decreasing over iteration k. For example, $\eta_k = c_1/(k + c_2)$.

◇ Mini-batch learning

Definition 3.16. A compromise between the batch gradient descent and the SGD is the so-called **mini-batch learning**. Mini-batch learning can be understood as applying batch gradient descent to smaller subsets of the training data – for example, 32 samples at a time.

The advantage over batch gradient descent is that convergence is reached faster via mini-batches because of the **more frequent weight updates**. Furthermore, mini-batch learning allows us to replace the for-loop over the training samples in stochastic gradient descent by **vectorized operations (vectorization)**, which can further improve the computational efficiency of our learning algorithm.

Exercises for Chapter 3

3.1. Verify (3.8).
Hint*:** *We assumed that the parameter* $\mathbf{w}^{(\ell)}$ *gave a mistake on* $\mathbf{x}^{(\ell)}$. *For example, let* $\mathbf{w}^{(\ell)^T}\mathbf{x}^{(\ell)} \geq 0$. *Then we* ***must *have* $(y^{(\ell)} - \widehat{y}^{(\ell)}) < 0$. *Why?*

3.2. Experiment all the examples on pp. 38–51, *Python Machine Learning, 3rd Ed.*. Through the examples, you will learn
(a) Gradient descent rule for Adaline,
(b) Feature scaling techniques, and
(c) Stochastic gradient descent rule for Adaline.

To get the **Iris dataset**, you have to use some lines on as earlier pages from 31.

3.3. Perturb the dataset (X) by a random Gaussian noise G_σ of an observable σ (so as for $G_\sigma(X)$ not to be linearly separable) and do the examples in Exercise 3.2 again.
Note: In most cases, classifiers become less accurate for noisy data.

3.4. Implement a code for Adaline, in the form of `perceptron.py`, p. 50, and verify it by classifying some appropriate datasets, e.g. the Iris dataset.
Note: This problem is asking you to implement a *complete and expandable* code for Adaline; you may adopt parts of the code used in Exercise 3.2.

- The correction terms in Adaline are accumulated from all data points in each iteration. As a consequence, the learning rate η may be chosen smaller as the number of points increases.
- **Implementation**: In order to overcome the problem, you may scale the correction terms by the number of data points.
  - Redefine the **cost function** (3.12):

$$\mathcal{J}(\mathbf{w}, b) = \frac{1}{2N}\sum_{i=1}^{N}\left(y^{(i)} - \phi(z^{(i)})\right)^2. \tag{3.19}$$

    where $z^{(i)} = \mathbf{w}^T\mathbf{x}^{(i)} + b$ and $\phi = I$, the identity.
  - Then the correction terms in (3.15) become correspondingly

$$\begin{aligned} \Delta\mathbf{w} &= \eta\frac{1}{N}\sum_{i}\left(y^{(i)} - \phi(z^{(i)})\right)\mathbf{x}^{(i)}, \\ \Delta b &= \eta\frac{1}{N}\sum_{i}\left(y^{(i)} - \phi(z^{(i)})\right). \end{aligned} \tag{3.20}$$

CHAPTER 4

Gradient-based Methods for Optimization

Optimization is the branch of research-and-development that aims to solve the problem of finding the elements which maximize or minimize a given real-valued function, while respecting constraints. Many problems in engineering and machine learning can be cast as optimization problems, which explains the growing importance of the field. An **optimization problem** is the problem of finding **the best solution** from all **feasible solutions**.
In this chapter, we will discuss details about

- Gradient descent method,
- Newton's method, and
- Their variants.

Contents of Chapter 4

4.1. Gradient Descent Method 64
4.2. Newton's Method 75
4.3. Quasi-Newton Methods 80
4.4. The Stochastic Gradient Method 84
4.5. The Levenberg–Marquardt Algorithm, for Nonlinear Least-Squares Problems 89
Exercises for Chapter 4 94

4.1. Gradient Descent Method

The first method that we will describe is one of the oldest methods in optimization: **gradient descent method**, a.k.a **steepest descent method**. The method was suggested by Augustin-Louis Cauchy in 1847 [47]. He was a French mathematician and physicist who made pioneering contributions to mathematical analysis. Motivated by the need to solve “large" quadratic problems (6 variables) that arise in Astronomy, he invented the method of gradient descent. Today, this method is used to comfortably solve problems with thousands of variables.

Figure 4.1: Augustin-Louis Cauchy

Problem 4.1. (Optimization Problem).
Let $\Omega \subset \mathbb{R}^d$, $d \geq 1$. Given a real-valued function $f : \Omega \to \mathbb{R}$, the general problem of finding the value that minimizes f is formulated as follows.

$$\min_{\mathbf{x} \in \Omega} f(\mathbf{x}). \tag{4.1}$$

In this context, f is the **objective function** (sometimes referred to as **loss function** or **cost function**). $\Omega \subset \mathbb{R}^d$ is the **domain** of the function (also known as the **constraint set**).

Example 4.2. (Rosenbrock function). For example, the **Rosenbrock function** in the two-dimensional (2D) space is defined as[1]

$$f(x, y) = (1 - x)^2 + 100\,(y - x^2)^2. \tag{4.2}$$

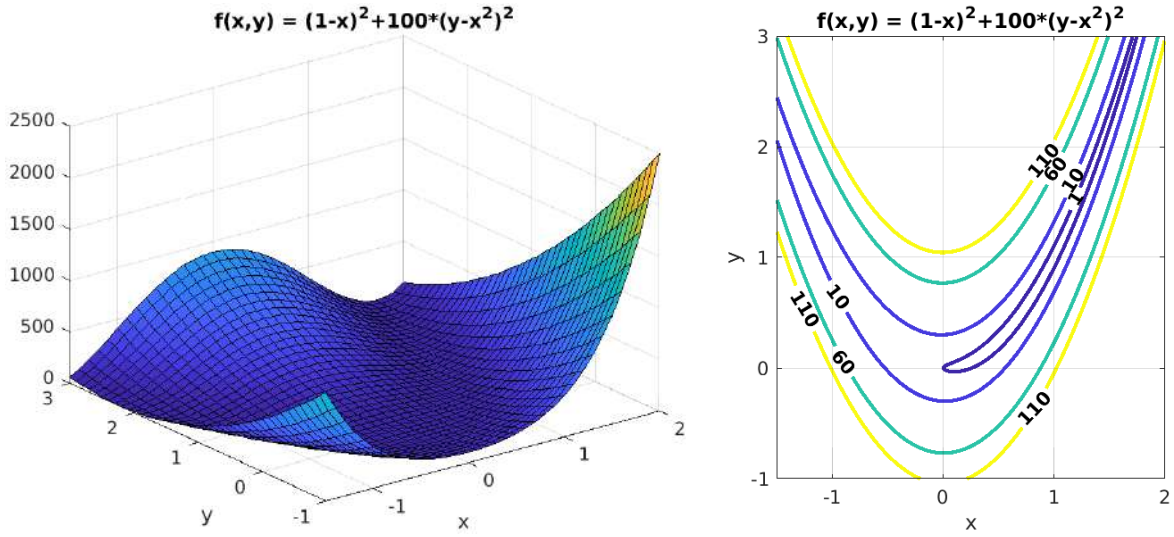


Figure 4.2: Plots of the Rosenbrock function $f(x, y) = (1 - x)^2 + 100\,(y - x^2)^2$.

Note: The Rosenbrock function is commonly used when evaluating the performance of an optimization algorithm; because

- its minimizer `x = np.array([1.,1.])` is found in curved valley, and so minimizing the function is non-trivial, and
- the Rosenbrock function is included in the `scipy.optimize` package (as `rosen`), as well as its gradient (`rosen_der`) and its Hessian (`rosen_hess`).

[1]The Rosenbrock function in 3D is given as $f(x, y, z) = [(1 - x)^2 + 100\,(y - x^2)^2] + [(1 - y)^2 + 100\,(z - y^2)^2]$, which has exactly one minimum at $(1, 1, 1)$. Similarly, one can define the Rosenbrock function in general N-dimensional spaces, for $N \geq 4$, by adding one more component for each enlarged dimension. That is, $f(\mathbf{x}) = \sum_{i=1}^{N-1} \left[(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2\right]$, where $\mathbf{x} = [x_1, x_2, \cdots, x_N] \in \mathbb{R}^N$. See Wikipedia (https://en.wikipedia.org/wiki/Rosenbrock_function) for details.

Remark 4.3. (Gradient)
The gradient ∇f is a vector (a direction to move) that is

- pointing in the **direction of greatest increase** of the function, and
- **zero ($\nabla f = 0$)** at local maxima or local minima.

The goal of the gradient descent method is to address directly the process of minimizing the function f, using the fact that $-\nabla f(\mathrm{x})$ is the direction of **steepest descent** of f at x. Given an initial point x_0, we move it to the direction of $-\nabla f(\mathrm{x}_0)$ so as to get a smaller function value. That is,

$$\mathbf{x}_1 = \mathbf{x}_0 - \gamma \nabla f(\mathbf{x}_0) \Rightarrow f(\mathbf{x}_1) < f(\mathbf{x}_0).$$

We repeat this process till reaching at a desirable minimum. Thus the method is formulated as follows.

Algorithm 4.4. ***(Gradient descent method)***
Given an initial point x_0*, find iterates* x_{n+1} *recursively using*

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma \nabla f(\mathbf{x}_n), \tag{4.3}$$

for some $\gamma > 0$*. The parameter* γ *is called the* ***step length*** *or the* ***learning rate****.* □

To understand the basics of *gradient descent* (GD) method thoroughly, we start with the algorithm for solving

- unconstrained minimization problems
- defined in the one-dimensional (1D) space.

4.1.1. The gradient descent method in 1D

Problem 4.5. Consider the minimization problem in 1D:

$$\min_x \ f(x), \quad x \in S, \tag{4.4}$$

where S is a closed interval in $\mathbb{R}$. Then its gradient descent method reads

$$x_{n+1} = x_n - \gamma\, f'(x_n). \tag{4.5}$$

Picking the step length γ: Assume that the step length was chosen to be independent of n, although one can play with other choices as well. The question is how to select γ in order to make the best gain of the method. To turn the right-hand side of (4.5) into a more manageable form, we invoke Taylor's Theorem:[2]

$$f(x+t) = f(x) + t\, f'(x) + \int_x^{x+t} (x+t-s)\, f''(s)\, ds. \tag{4.6}$$

Assuming that $|f''(s)| \le L$, we have

$$f(x+t) \le f(x) + t\, f'(x) + \frac{t^2}{2} L.$$

Now, letting $x = x_n$ and $t = -\gamma\, f'(x_n)$ reads

$$\begin{aligned} f(x_{n+1}) &= f(x_n - \gamma\, f'(x_n)) \\ &\le f(x_n) - \gamma\, f'(x_n)\, f'(x_n) + \frac{1}{2} L\, [\gamma\, f'(x_n)]^2 \\ &= f(x_n) - [f'(x_n)]^2 \Big(\gamma - \frac{L}{2}\gamma^2\Big). \end{aligned} \tag{4.7}$$

The gain (learning) from the method occurs when

$$\gamma - \frac{L}{2}\gamma^2 > 0 \quad \Rightarrow \quad 0 < \gamma < \frac{2}{L}, \tag{4.8}$$

and it will be best when $\gamma - \frac{L}{2}\gamma^2$ is maximal. This happens at the point

$$\boxed{\gamma = \frac{1}{L}}. \tag{4.9}$$

[2] **Taylor's Theorem with integral remainder**: Suppose $f \in C^{n+1}[a,b]$ and $x_0 \in [a,b]$. Then, for every $x \in [a,b]$, $f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!} (x-x_0)^k + R_n(x)$, $R_n(x) = \frac{1}{n!} \int_{x_0}^{x} (x-s)^n\, f^{(n+1)}(s)\, ds$.

Thus an effective **gradient descent method** (4.5) can be written as

$$x_{n+1} = x_n - \gamma f'(x_n) = x_n - \frac{1}{L} f'(x_n) = x_n - \frac{1}{\max |f''(x)|} f'(x_n). \tag{4.10}$$

Furthermore, it follows from (4.7) and (4.9) that

$$f(x_{n+1}) \leq f(x_n) - \frac{1}{2L}[f'(x_n)]^2. \tag{4.11}$$

Remark 4.6. Convergence of gradient descent method

Thus it is obvious that the method defines a sequence of points $\{x_n\}$ along which $\{f(x_n)\}$ decreases.

- If f is bounded from below and the level sets of f are bounded, $\{f(x_n)\}$ converges; so does $\{x_n\}$. That is, there is a point $\widehat{x}$ such that
$$\lim_{n\to\infty} x_n = \widehat{x}. \tag{4.12}$$
- Now, we can rewrite (4.11) as
$$[f'(x_n)]^2 \leq 2L\,[f(x_n) - f(x_{n+1})]. \tag{4.13}$$
Since $f(x_n) - f(x_{n+1}) \to 0$, also $f'(x_n) \to 0$.
- When f' is continuous, using (4.12) reads
$$f'(\widehat{x}) = \lim_{n\to\infty} f'(x_n) = 0, \tag{4.14}$$
which implies that the limit $\widehat{x}$ is a **critical point**.
- The method thus generally finds a critical point but that could still be a local minimum or a saddle point. Which it is cannot be decided at this level of analysis. □

4.1.2. The full gradient descent algorithm

We can implement the *full* gradient descent algorithm as follows. The algorithm has only one free parameter: γ.

Algorithm 4.7. (The Gradient Descent Algorithm).

input: initial guess $\mathbf{x}_0$, step size $\gamma > 0$;
for $n = 0, 1, 2, \cdots$ **do**
$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma \nabla f(\mathbf{x}_n); \tag{4.15}$$
end for
return $\mathbf{x}_{n+1}$;

Remark 4.8. In theory, the step length γ can be found as in (4.9):

$$\gamma = \frac{1}{L}, \quad \text{where} \quad L = \max_{\mathbf{x}} \|\nabla^2 f(\mathbf{x})\|. \tag{4.16}$$

Here $||\cdot||$ denotes an **induced matrix norm** and $\nabla^2 f(\mathbf{x})$ is the **Hessian** of f defined by

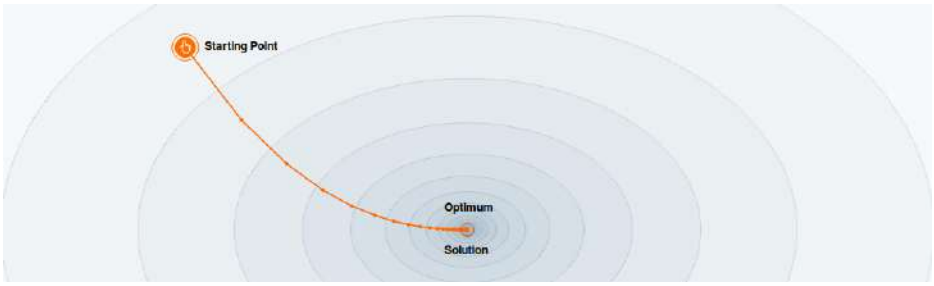
$$\nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial {x_1}^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial {x_2}^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \frac{\partial^2 f}{\partial x_d \partial x_2} & \cdots & \frac{\partial^2 f}{\partial {x_d}^2} \end{bmatrix} \in \mathbb{R}^{d \times d}. \tag{4.17}$$

- However, in practice, the computation of the Hessian (and L) can be expensive.

Remark 4.9. **Gradient Descent vs. Newton's Method**
The **gradient descent method** can be viewed as a simplification of the **Newton's method** (Section 4.2 below), replacing the inverse of Hessian, $(\nabla^2 f)^{-1}$, with a constant γ.

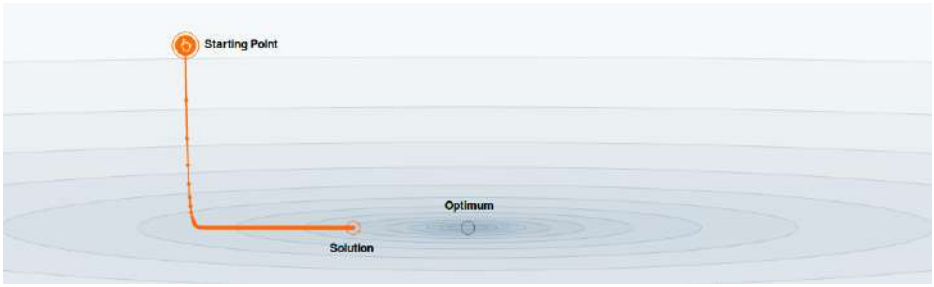
Convergence of Gradient Descent: Constant γ

Here we examine convergence of gradient descent on three examples: a *well-conditioned quadratic*, an *poorly-conditioned quadratic*, and a *non-convex function*, as shown by Dr. Fabian Pedregosa, UC Berkeley.



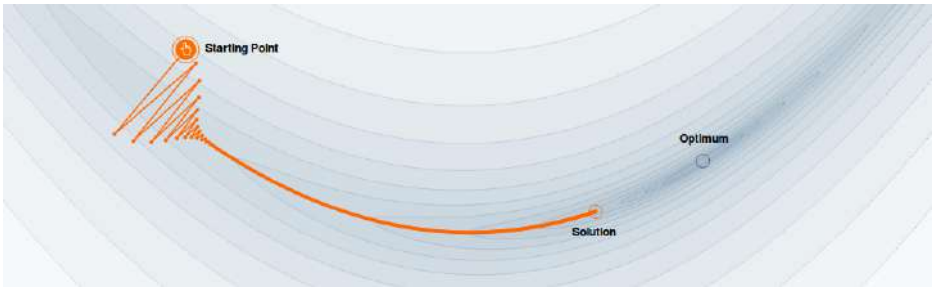
$\gamma = 0.2$

Figure 4.3: On a well-conditioned quadratic function, the gradient descent converges in a few iterations to the optimum



$\gamma = 0.02$

Figure 4.4: On a poorly-conditioned quadratic function, the gradient descent converges and takes many more iterations to converge than on the above well-conditioned problem. This is ***partially*** because gradient descent requires a ***much smaller step size*** on this problem to converge.



$\gamma = 0.02$

Figure 4.5: Gradient descent also converges on a poorly-conditioned non-convex problem. Convergence is slow in this case.

The Choice of Step Size: Backtracking Line Search

Note: The convergence of the gradient descent method can be extremely **sensitive to the choice of step size**. It often requires to choose the step size **adaptively**: the step size would better be chosen small in regions of large variability of the gradient, while in regions with small variability we would like to take it large.

Strategy 4.10. **Backtracking line search** procedures allow to select a step size depending on the current iterate and the gradient. In this procedure, we select an initial (optimistic) step size γ_n and evaluate the following inequality (known as **sufficient decrease condition**):

$$f(\mathbf{x}_n - \gamma_n \nabla f(\mathbf{x}_n)) \leq f(\mathbf{x}_n) - \frac{\gamma_n}{2} \|\nabla f(\mathbf{x}_n)\|^2. \tag{4.18}$$

If this inequality is verified, the current step size is kept. If not, the step size is divided by 2 (or any number larger than 1) repeatedly until (4.18) is verified. To get a better understanding, refer to (4.11) on p. 68, with (4.9).

The gradient descent algorithm with backtracking line search then becomes

Algorithm 4.11. (The Gradient Descent Algorithm, with Backtracking Line Search).

input: initial guess $\mathbf{x}_0$, step size $\gamma_0 > 0$;
for $n = 0, 1, 2, \cdots$ **do**
  initial step size estimate γ_n;
  while (TRUE) do
    if $f(\mathbf{x}_n - \gamma_n \nabla f(\mathbf{x}_n)) \leq f(\mathbf{x}_n) - \frac{\gamma_n}{2} \|\nabla f(\mathbf{x}_n)\|^2$
      break;
    else $\gamma_n = \gamma_n/2$;
  end while
  $\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla f(\mathbf{x}_n)$;
end for
return $\mathbf{x}_{n+1}$; (4.19)

Convergence of Gradient Descent: Backtracking line search

The following examples show the convergence of gradient descent with the aforementioned backtracking line search strategy for the step size.

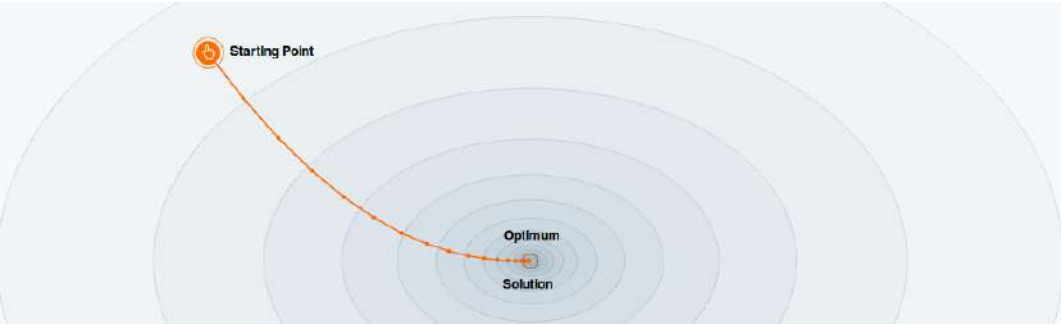


Figure 4.6: On a well-conditioned quadratic function, the gradient descent converges in a few iterations to the optimum. Adding the backtracking line search strategy for the step size does not change much in this case.

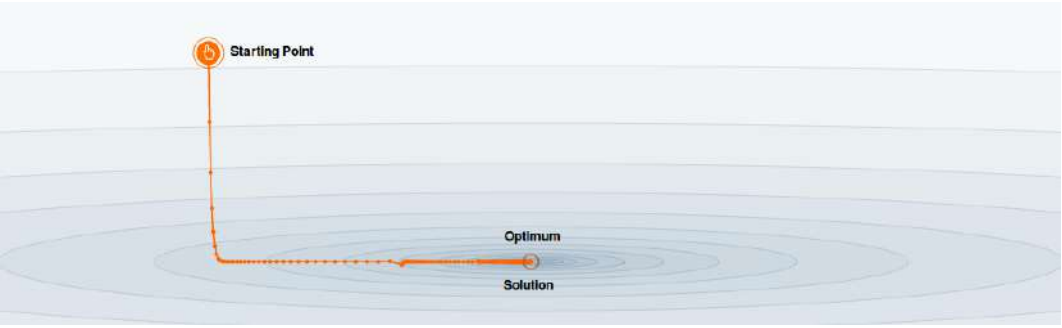


Figure 4.7: In this example we can clearly see the effect of the backtracking line search strategy: once the algorithm in a region of low curvature, it can take larger step sizes. The final result is a much improved convergence compared with the fixed step-size equivalent.

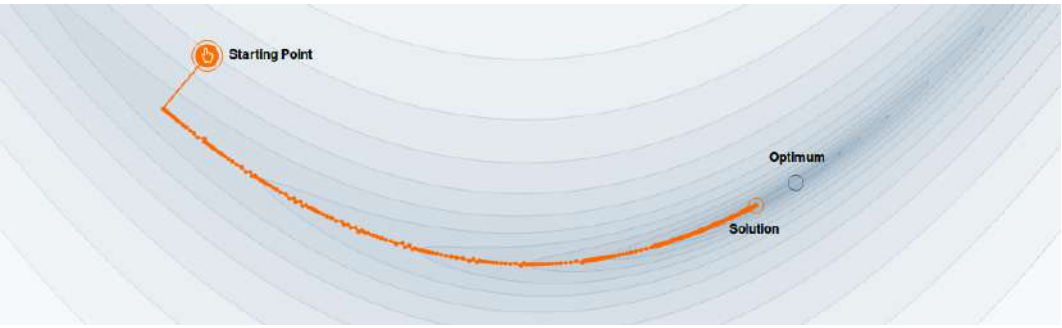


Figure 4.8: The backtracking line search also improves convergence on non-convex problems.

See Exercise 1 on p. 94.

4.1.3. Surrogate minimization: A unifying principle

Now, we aim to solve an optimization problem as in (4.1):

$$\min_{\mathbf{x}\in\Omega} f(\mathbf{x}). \tag{4.20}$$

Key Idea **4.12.** Start at an initial estimate $\mathbf{x}_0$ and successively minimize an **approximating function** $\mathcal{Q}_n(\mathbf{x})$ [43]:

$$\mathbf{x}_{n+1} = \arg\min_{\mathbf{x}\in\Omega} \mathcal{Q}_n(\mathbf{x}). \tag{4.21}$$

We will call $\mathcal{Q}_n$ a **surrogate function**. It is also known as a **merit function**. A good surrogate function should be:

- Easy to optimize.
- Flexible enough to approximate a wide range of functions.

Gradient descent method: Approximates the objective function near $\mathbf{x}_n$ with a quadratic surrogate of the form

$$\mathcal{Q}_n(\mathbf{x}) = \mathbf{c}_n + \boldsymbol{G}_n \cdot (\mathbf{x} - \mathbf{x}_n) + \frac{1}{2\gamma} (\mathbf{x} - \mathbf{x}_n)^T (\mathbf{x} - \mathbf{x}_n), \tag{4.22}$$

which coincides with f in its value and first derivative, i.e.,

$$\begin{aligned} \mathcal{Q}_n(\mathbf{x}_n) = f(\mathbf{x}_n) &\;\Rightarrow\; \mathbf{c}_n = f(\mathbf{x}_n), \\ \nabla\mathcal{Q}_n(\mathbf{x}_n) = \nabla f(\mathbf{x}_n) &\;\Rightarrow\; \boldsymbol{G}_n = \nabla f(\mathbf{x}_n). \end{aligned} \tag{4.23}$$

The gradient descent method thus updates its iterates minimizing the following surrogate function:

$$\mathcal{Q}_n(\mathbf{x}) = f(\mathbf{x}_n) + \nabla f(\mathbf{x}_n) \cdot (\mathbf{x} - \mathbf{x}_n) + \frac{1}{2\gamma} \|\mathbf{x} - \mathbf{x}_n\|^2. \tag{4.24}$$

Differentiating the function and equating to zero reads

$$\mathbf{x}_{n+1} = \arg\min_{\mathbf{x}} \mathcal{Q}_n(\mathbf{x}) = \mathbf{x}_n - \gamma \nabla f(\mathbf{x}_n). \tag{4.25}$$

Multiple Local Minima Problem

Remark 4.13. Optimizing Optimization Algorithms
Although you can choose the step size **smartly**, there is no guarantee for your algorithm to converge to the desired solution (the global minimum), particularly when the objective is not convex.
Here, we consider the so-called **Gaussian homotopy continuation** method [53], which may overcome the **local minima problem** for certain classes of optimization problems.

- The method begins by trying to find a convex approximation of an optimization problem, using a technique called **Gaussian smoothing**.
- Gaussian smoothing converts the cost function into a related function, each of whose values is a **weighted average** of all the surrounding values.
- This has the effect of smoothing out any abrupt dips or ascents in the cost function's graph, as shown in Figure 4.9.
- The weights assigned the surrounding values are determined by a Gaussian function, or normal distribution.

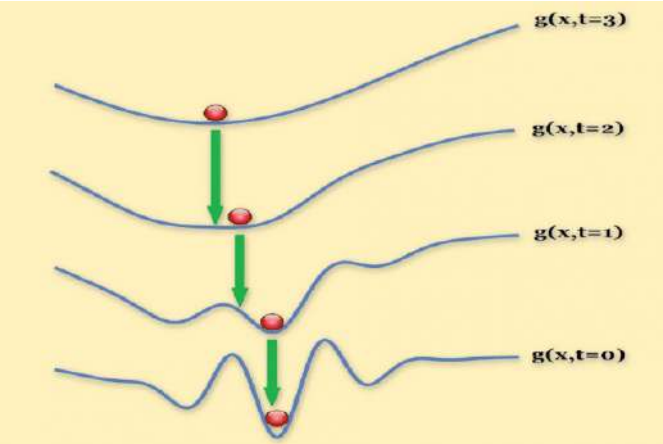


Figure 4.9: Smooth sailing, through a Gaussian smoothing.

However, there will be many ways to incorporate Gaussian smoothing; a realization of the method will be challenging, particularly for ML optimization. See P.5 (p. 407).

4.2. Newton's Method

4.2.1. Derivation

Scheme **4.14.** The **Newton's method** is an iterative method to solve the unconstrained optimization problem in (4.1), p. 64, when f is twice differentiable. In Newton's method, we approximate the objective with a **quadratic surrogate** of the form

$$\mathcal{Q}_n(\mathbf{x}) = \mathbf{c}_n + \boldsymbol{G}_n \cdot (\mathbf{x} - \mathbf{x}_n) + \frac{1}{2\gamma} (\mathbf{x} - \mathbf{x}_n)^T \boldsymbol{H}_n (\mathbf{x} - \mathbf{x}_n). \tag{4.26}$$

Compared with gradient descent, the quadratic term is not fixed to be the identity but instead incorporates an **invertible matrix** $\boldsymbol{H}_n$.

- A reasonable condition to impose on this surrogate function is that at $\mathbf{x}_n$ it coincides with f *at least* in **its value** and **first derivatives**, as in (4.23).
- **An extra condition** the method imposes is that

$$\boldsymbol{H}_n = \nabla^2 f(\mathbf{x}_n), \tag{4.27}$$

  where $\nabla^2 f$ is the **Hessian** of f defined as in (4.17).
- Thus the Newton's method updates its iterates **minimizing** the following surrogate function:

$$\mathcal{Q}_n(\mathbf{x}) = f(\mathbf{x}_n) + \nabla f(\mathbf{x}_n) \cdot (\mathbf{x} - \mathbf{x}_n) + \frac{1}{2\gamma} (\mathbf{x} - \mathbf{x}_n)^T \nabla^2 f(\mathbf{x}_n) (\mathbf{x} - \mathbf{x}_n). \tag{4.28}$$

- We can find the **optimum of the function** differentiating and equating to zero. This way we find (assuming the Hessian is invertible)

$$\mathbf{x}_{n+1} = \arg\min_{\mathbf{x}} \mathcal{Q}_n(\mathbf{x}) = \mathbf{x}_n - \gamma \left[\nabla^2 f(\mathbf{x}_n)\right]^{-1} \nabla f(\mathbf{x}_n). \tag{4.29}$$

Note: When $\gamma = 1$, $\mathcal{Q}_n(\mathbf{x})$ in (4.28) is the second-order approximation of the objective function near $\mathbf{x}_n$.

Remark 4.15. Where applicable, Newton's method **converges much faster** towards a local maximum or minimum than the gradient descent.

- In fact, every local minimum has a neighborhood such that, if we start within this neighborhood, Newton's method with step size $\gamma = 1$ **converges quadratically** assuming the Hessian is invertible and Lipschitz continuous.

Remark 4.16. The Newton's method can be seen as to find the **critical points** of f, i.e., $\widehat{\mathbf{x}}$ such that $\nabla f(\widehat{\mathbf{x}}) = 0$. Let

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}. \tag{4.30}$$

Then

$$\nabla f(\mathbf{x}_{n+1}) \;=\; \nabla f(\mathbf{x}_n + \Delta\mathbf{x}) \;=\; \nabla f(\mathbf{x}_n) + \nabla^2 f(\mathbf{x}_n)\,\Delta\mathbf{x} + \mathcal{O}(|\Delta\mathbf{x}|^2).$$

Truncating high-order terms of $\Delta\mathbf{x}$ and equating the result to zero reads

$$\Delta\mathbf{x} \;=\; -\left(\nabla^2 f(\mathbf{x}_n)\right)^{-1} \nabla f(\mathbf{x}_n). \tag{4.31}$$

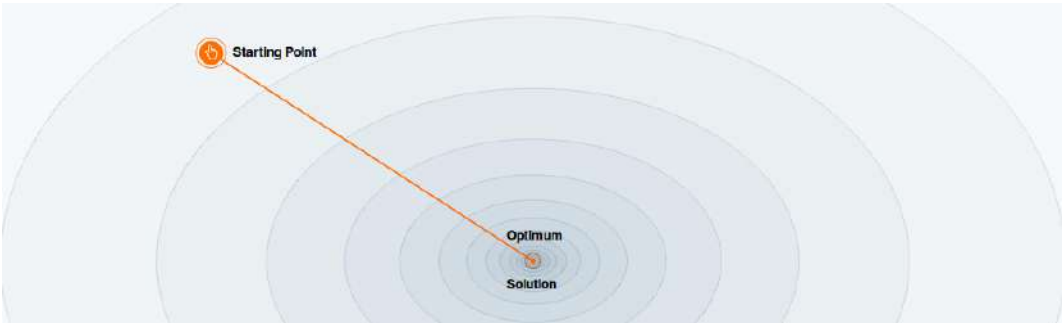
Implementation of Newton's Method

Only the difference from the gradient descent algorithm is to compute the Hessian matrix $\nabla^2 f(\mathbf{x}_n)$ to be applied to the gradient.

Algorithm 4.17. (Newton's method).

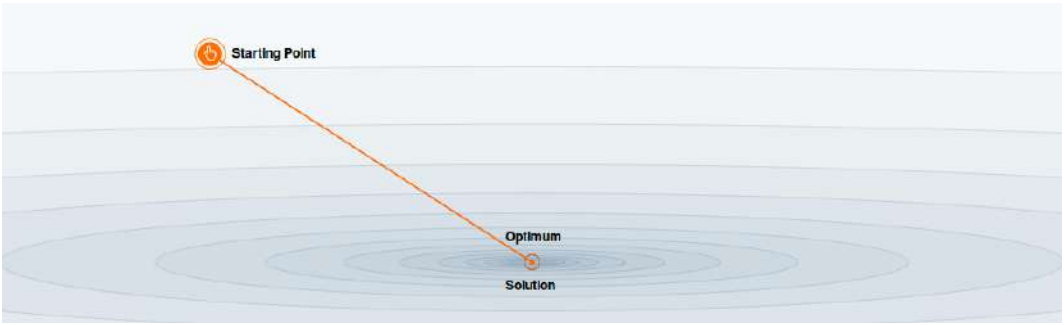
input: initial guess $\mathbf{x}_0$, step size $\gamma > 0$;
for $n = 0, 1, 2, \cdots$ **do**
$$\mathbf{x}_{n+1} \;=\; \mathbf{x}_n - \gamma\,\left[\nabla^2 f(\mathbf{x}_n)\right]^{-1} \nabla f(\mathbf{x}_n); \tag{4.32}$$
end for
return $\mathbf{x}_{n+1}$;

For the three example functions in Section 4.1.2, the Newton's method performs better as shown in the following.



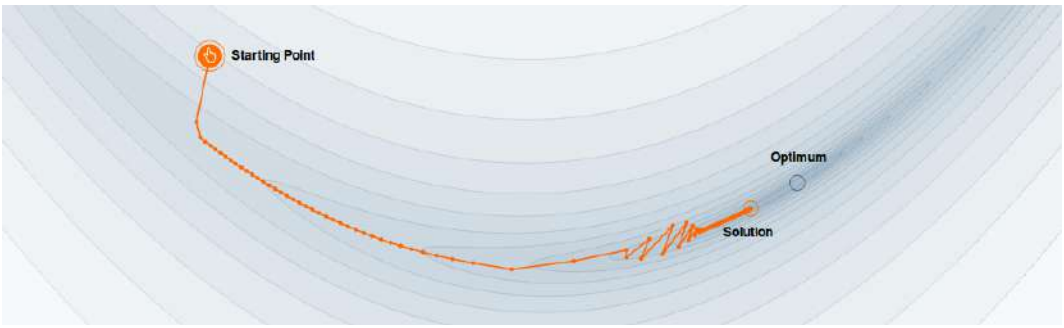
$\gamma = 1$

Figure 4.10: In this case the approximation is exact and it converges in a single iteration.



$\gamma = 1$

Figure 4.11: Although badly-conditioned, the cost function is quadratic; it converges in a single iteration.



$\gamma = 1$

Figure 4.12: When the Hessian is close to singular, there might be some numerical instabilities. However, it is better than the result of the gradient descent method in Figure 4.5.

4.2.2. Hessian and principal curvatures

Claim 4.18. The **Hessian** (or **Hessian matrix**) describes the **local curvature** of a function. The eigenvalues and eigenvectors of the Hessian have geometric meaning:

- The first principal eigenvector (corresponding to the largest eigenvalue in modulus) is the direction of **greatest curvature**.
- The last principal eigenvector (corresponding to the smallest eigenvalue in modulus) is the direction of **least curvature**.
- The corresponding eigenvalues are the respective amounts of these curvatures.

The eigenvectors of the Hessian are called **principal directions**, which are always orthogonal to each other. The eigenvalues of the Hessian are called **principal curvatures** and are invariant under rotation and always real-valued.

Observation 4.19. Let a Hessian matrix $\boldsymbol{H} \in \mathbb{R}^{d\times d}$ be positive definite and its eigenvalue-eigenvector pairs be given as $\{(\lambda_j, \mathbf{u}_j)\}$, $j = 1, 2, \cdots, d$.

- Then, given a vector $\mathbf{v} \in \mathbb{R}^d$, it can be expressed as

$$\mathbf{v} = \sum_{j=1}^{d} \xi_j \mathbf{u}_j,$$

and therefore

$$H^{-1}\mathbf{v} = \sum_{j=1}^{d} \xi_j \frac{1}{\lambda_j} \mathbf{u}_j, \tag{4.33}$$

where components of $\mathbf{v}$ in leading principal directions of $\boldsymbol{H}$ have been diminished with larger factors.

- Thus the angle measured from $H^{-1}\mathbf{v}$ to **the least principal direction of** H becomes smaller than the angle measured from $\mathbf{v}$.
- It is also true when $\mathbf{v}$ is the gradient vector (in fact, the negation of the gradient vector).

Note: The above observation can be rephrased mathematically as follows. Let $\mathbf{u}_d$ be the least principal direction of H. Then

$$\text{angle}(\mathbf{u}_d,\, H^{-1}\mathbf{v}) \,<\, \text{angle}(\mathbf{u}_d,\, \mathbf{v}), \quad \forall\, \mathbf{v}, \tag{4.34}$$

where

$$\text{angle}(\mathbf{a}, \mathbf{b}) \,=\, \arccos\left(\frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|\,\|\mathbf{b}\|}\right).$$

This implies that by setting $\mathbf{v} = -\nabla f(\mathbf{x}_n)$, the adjusted vector $H^{-1}\mathbf{v}$ is a rotation (and scaling) of the steepest descent vector towards the least curvature direction.

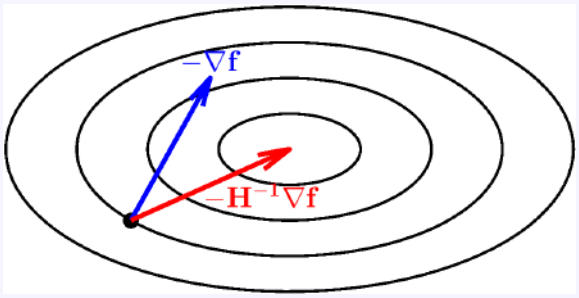


Figure 4.13: The effect of the Hessian inverse H^{-1}.

Claim 4.20. **The net effect of H^{-1}**
Rotate and scale the gradient vector to face towards the minimizer by a certain degree. This operation may make the Newton's method converge much faster than the gradient descent method.

Example 4.21. One can easily check that at each point (x, y) on the ellipsoid

$$z \,=\, f(x,y) \,=\, \frac{(x-h)^2}{a^2} + \frac{(y-k)^2}{b^2}, \tag{4.35}$$

the vector $-\left[\nabla^2 f(x,y)\right]^{-1} \nabla f(x,y)$ is always facing towards the minimizer (h, k). See Exercise 2. □

4.3. Quasi-Newton Methods

Note: The central issue with Newton's method is that we need to be able to ***compute efficiently*** the **Hessian matrix** and **its inverse**.

- For ML applications, the dimensionality of the problem can be of the **order of thousands or millions**; computing the Hessian or its inverse is often impractical.
- Because of these reasons, Newton's method is **rarely used in practice** to optimize functions corresponding to **large problems**.
- Luckily, Newton's method can still work even if the Hessian is replaced by a **good approximation**.

The BFGS Algorithm (1970)

Note: One of the most popular **quasi-Newton methods** is the BFGS algorithm, which is named after Charles George **Broyden** [9], Roger **Fletcher** [21], Donald **Goldfarb** [24], and David **Shanno** [72].

Key Idea **4.22.** As a byproduct of the optimization, we observe many gradients. Can we use these gradients to iteratively construct an approximation of the Hessian?

Derivation of BFGS algorithm

- At each iteration of the method, we consider the surrogate function:

$$\mathcal{Q}_n(\mathbf{x}) = \mathbf{c}_n + \boldsymbol{G}_n \cdot (\mathbf{x} - \mathbf{x}_n) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_n)^T \boldsymbol{H}_n (\mathbf{x} - \mathbf{x}_n), \tag{4.36}$$

  where in this case $\boldsymbol{H}_n$ is **an approximation to the Hessian matrix**, which is updated iteratively at each stage.

- **A reasonable thing to ask** to this surrogate is that its gradient coincides with ∇f at the last two iterates $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$:

$$\begin{aligned} \nabla \mathcal{Q}_{n+1}(\mathbf{x}_{n+1}) &= \nabla f(\mathbf{x}_{n+1}), \\ \nabla \mathcal{Q}_{n+1}(\mathbf{x}_n) &= \nabla f(\mathbf{x}_n). \end{aligned} \tag{4.37}$$

- From the definition of $\mathcal{Q}_{n+1}$:

$$\mathcal{Q}_{n+1}(\mathbf{x}) = \mathbf{c}_{n+1} + \boldsymbol{G}_{n+1} \cdot (\mathbf{x} - \mathbf{x}_{n+1}) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_{n+1})^T \boldsymbol{H}_{n+1} (\mathbf{x} - \mathbf{x}_{n+1}),$$

  we have

$$\nabla \mathcal{Q}_{n+1}(\mathbf{x}_{n+1}) - \nabla \mathcal{Q}_{n+1}(\mathbf{x}_n) = \boldsymbol{G}_{n+1} - \nabla \mathcal{Q}_{n+1}(\mathbf{x}_n) = -\boldsymbol{H}_{n+1}(\mathbf{x}_n - \mathbf{x}_{n+1}).$$

  Thus we reach at the following condition on $\boldsymbol{H}_{n+1}$:

$$\boldsymbol{H}_{n+1}(\mathbf{x}_{n+1} - \mathbf{x}_n) \;=\; \nabla f(\mathbf{x}_{n+1}) - \nabla f(\mathbf{x}_n), \tag{4.38}$$

  which is the **secant equation**.

- Let

$$\boldsymbol{s}_n = \mathbf{x}_{n+1} - \mathbf{x}_n \;\text{ and }\; \mathbf{y}_n = \nabla f(\mathbf{x}_{n+1}) - \nabla f(\mathbf{x}_n).$$

  Then $\boldsymbol{H}_{n+1}\boldsymbol{s}_n = \mathbf{y}_n$, which requires to satisfy the **curvature condition**

$$\mathbf{y}_n \cdot \boldsymbol{s}_n > 0, \tag{4.39}$$

  with which $\boldsymbol{H}_{n+1}$ becomes positive definite. (Pre-multiply $\boldsymbol{s}_n^T$ to the secant equation to prove it.)

- In order to maintain **the symmetry and positive definiteness of $\boldsymbol{H}_{n+1}$**, the update formula can be chosen as[3]

$$\boldsymbol{H}_{n+1} \;=\; \boldsymbol{H}_n + \alpha \mathbf{u}\mathbf{u}^T + \beta \mathbf{v}\mathbf{v}^T. \tag{4.40}$$

- Imposing the secant condition $\boldsymbol{H}_{n+1}\boldsymbol{s}_n = \mathbf{y}_n$ and with (4.40), we get the update equation of $\boldsymbol{H}_{n+1}$:

$$\boldsymbol{H}_{n+1} \;=\; \boldsymbol{H}_n + \frac{\mathbf{y}_n \mathbf{y}_n^T}{\mathbf{y}_n \cdot \boldsymbol{s}_n} - \frac{(\boldsymbol{H}_n \boldsymbol{s}_n)(\boldsymbol{H}_n \boldsymbol{s}_n)^T}{\boldsymbol{s}_n \cdot \boldsymbol{H}_n \boldsymbol{s}_n}. \tag{4.41}$$

- Let $\boldsymbol{B}_n = \boldsymbol{H}_n^{-1}$, the inverse of $\boldsymbol{H}_n$. Then, applying the **Sherman-Morrison formula**, we can update $\boldsymbol{B}_{n+1} = \boldsymbol{H}_{n+1}^{-1}$ as follows.

$$\boldsymbol{B}_{n+1} \;=\; \left(I - \frac{\boldsymbol{s}_n \mathbf{y}_n^T}{\mathbf{y}_n \cdot \boldsymbol{s}_n}\right) \boldsymbol{B}_n \left(I - \frac{\mathbf{y}_n \boldsymbol{s}_n^T}{\mathbf{y}_n \cdot \boldsymbol{s}_n}\right) + \frac{\boldsymbol{s}_n \boldsymbol{s}_n^T}{\mathbf{y}_n \cdot \boldsymbol{s}_n}. \tag{4.42}$$

  See Exercise 4.4.

[3] **Rank-one matrices**: Let A be an $m \times n$ matrix. Then $\operatorname{rank}(A) = 1$ if and only if there exist column vectors $\mathbf{v} \in \mathbb{R}^m$ and $\mathbf{w} \in \mathbb{R}^n$ such that $A = \mathbf{v}\mathbf{w}^T$.

Now, we are ready to summarize the BFGS algorithm.

Algorithm 4.23. ***(The BFGS algorithm).*** **The n-th step**:

1. *Obtain the search direction:* $\mathbf{p}_n = \boldsymbol{B}_n(-\nabla f(\mathbf{x}_n))$.
2. *Perform line-search to find an acceptable stepsize* γ_n.
3. *Set* $\boldsymbol{s}_n = \gamma_n \mathbf{p}_n$ *and update* $\mathbf{x}_{n+1} = \mathbf{x}_n + \boldsymbol{s}_n$.
4. *Get* $\mathbf{y}_n = \nabla f(\mathbf{x}_{n+1}) - \nabla f(\mathbf{x}_n)$.
5. *Update* $\boldsymbol{B} = \boldsymbol{H}^{-1}$*:*

$$\boldsymbol{B}_{n+1} = \left(I - \frac{\boldsymbol{s}_n \mathbf{y}_n^T}{\mathbf{y}_n \cdot \boldsymbol{s}_n}\right) \boldsymbol{B}_n \left(I - \frac{\mathbf{y}_n \boldsymbol{s}_n^T}{\mathbf{y}_n \cdot \boldsymbol{s}_n}\right) + \frac{\boldsymbol{s}_n \boldsymbol{s}_n^T}{\mathbf{y}_n \cdot \boldsymbol{s}_n}.$$

Remark 4.24. **The BFGS Algorithm**

- The algorithm begins with B_0, an estimation of H_0^{-1}.
  It is often better when $\boldsymbol{B_0 = H_0^{-1}}$**.**
- The resulting algorithm is a method which combines the ***low-cost of gradient descent*** with the ***favorable convergence properties of Newton's method***.

Examples, with the BFGS algorithm

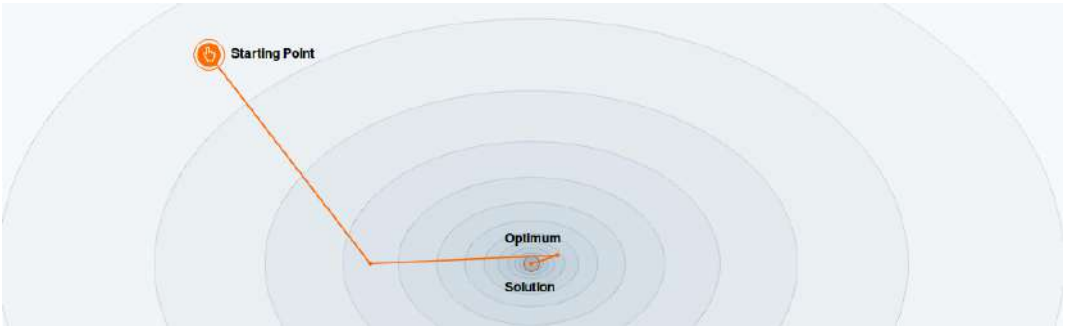


Figure 4.14: BFGS, on the **well-conditioned quadratic** objective function.

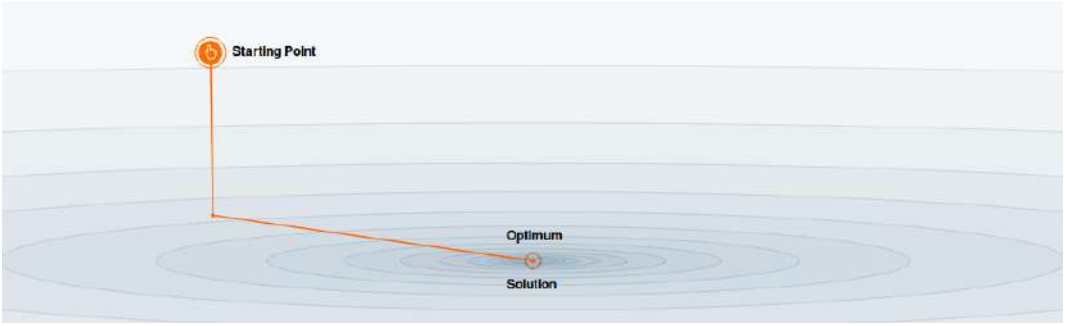


Figure 4.15: On the **poorly-conditioned quadratic** problem, the BFGS algorithm quickly builds a good estimator of the Hessian and is able to converge very fast towards the optimum. Note that this, just like the Newton method (and unlike gradient descent), BFGS does not seem to be affected (much) by a bad conditioning of the problem.

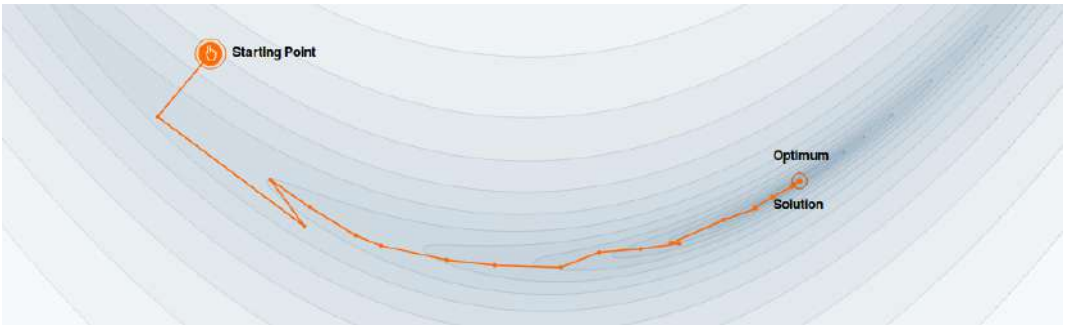


Figure 4.16: Even on the **ill-conditioned nonconvex** problem, the BFGS algorithm also converges extremely fast, with a convergence that is more similar to Newton's method than to gradient descent.

4.4. The Stochastic Gradient Method

The **stochastic gradient method** (**SGM**), introduced by Robbins-Monro in 1951 [63], is

- one of the most widely-used methods for large-scale optimization, and
- one of the main methods behind the current AI revolution.

Note: The SGM was considered earlier in Section 3.3.1, as a variant of the gradient descent method for Adaline classification. Here we will discuss it in details for more general optimization problems.

- The stochastic gradient method (a.k.a. **stochastic gradient descent** or **SGD**) can be used to solve optimization problems in which **the objective function is of the form**

$$f(x) = \mathbb{E}[f_i(x)],$$

  where the expectation is taken with respect to i.
- **The most common case** is when i can take a finite number of values, in which the problem becomes

$$\min_{\mathbf{x}\in\mathbb{R}^p} f(\mathbf{x}), \quad f(\mathbf{x}) = \frac{1}{m}\sum_{i=1}^{m} f_i(\mathbf{x}). \tag{4.43}$$

- The SGM can be motivated as an approximation to gradient descent in which at each iteration we approximate the gradient as

$$\nabla f(\mathbf{x}_n) \approx \nabla f_i(\mathbf{x}_n). \tag{4.44}$$

We can write the full stochastic gradient algorithm as follows. The algorithm has only one free parameter: γ.

Algorithm 4.25. (Stochastic Gradient Descent).

input: initial guess $\mathbf{x}_0$, step size sequence $\gamma_n > 0$;
for $n = 0, 1, 2, \cdots$ **do**
  Choose $i \in \{1, 2, \cdots, m\}$ uniformly at random;
  $\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla f_i(\mathbf{x}_n)$;
end for
return $\mathbf{x}_{n+1}$; (4.45)

The SGD can be much more efficient than gradient descent in the case in which **the objective consists of a large sum**, because at each iteration we only need to evaluate a partial gradient and not the full gradient.

Example 4.26. A least-squares problem can be written in the form acceptable by SGD since

$$\frac{1}{m}\|A\mathbf{x} - \mathbf{b}\|^2 = \frac{1}{m}\sum_{i=1}^{m}(A_i\mathbf{x} - \mathbf{b}_i)^2, \tag{4.46}$$

where A_i is the i-th row of A.

Step Size for the SGD

- The choice of step size is one of the most delicate aspects of the SGD.

  For the SGD, **the backtracking line search is not an option** since it would involve to evaluate the objective function at each iteration, which destroys the computational advantage of this method.

- **Two popular step size strategies exist for the SGD**: constant step size and decreasing step size.

  (a) **Constant step size**: In the constant step size strategy,

  $$\gamma_n = \gamma$$

  for some pre-determined constant γ.

  The method converges very fast to neighborhood of a local minimum and then bounces around. The radius of this neighborhood will depend on the step size γ [44, 51].

  (b) **Decreasing step size**: One can guarantee convergence to a local minimizer choosing a step size sequence that satisfies

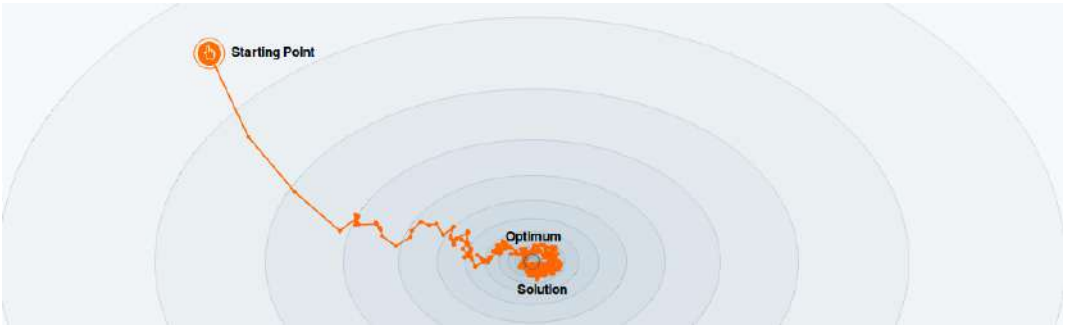
  $$\sum_{n=1}^{\infty} \gamma_n = \infty \quad \text{and} \quad \sum_{n=1}^{\infty} \gamma_n^2 < \infty. \tag{4.47}$$

  The most popular sequence to verify this is

  $$\gamma_n = \frac{C}{n}, \tag{4.48}$$

  for some constant C. This is often referred to as a **decreasing step-size sequence**, although in fact the sequence does not need to be monotonically decreasing.

Examples, with SGD



$\gamma = 0.2$

Figure 4.17: For the well-conditioned convex problem, stochastic gradient with constant step size converges quickly to a neighborhood of the optimum, but then bounces around.

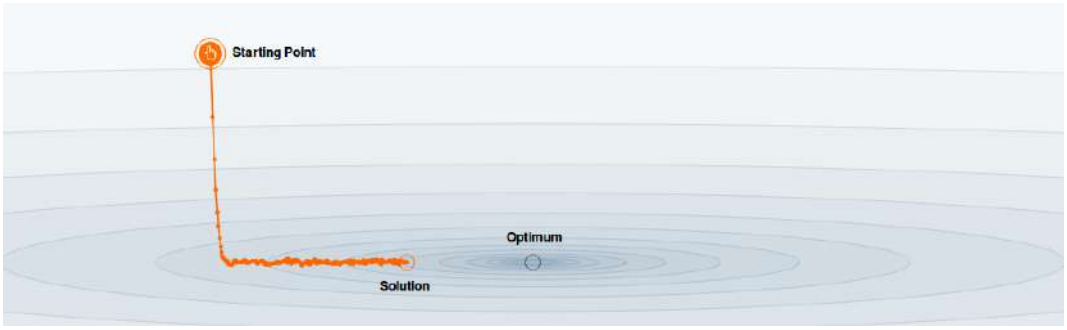


Figure 4.18: Stochastic Gradient **with decreasing step sizes** is quite robust to the choice of step size. On one hand there is really no good way to set the step size (e.g., no equivalent of line search for Gradient Descent) but on the other hand it converges for a wide range of step sizes.

Convergence of the SGD

Quesiton. **Why does the SGD converge**, despite its update being a very rough estimate of the gradient?

To answer this question mathematically, we must first understand the **unbiasedness property** of its update.

Proposition 4.27. (Unbiasedness of the SGD update).
Let $\mathbb{E}_n$ denote the expectation with respect to the choice of random sample (i) at iteration n. Then since the index i is chosen ***uniformly*** at random, we have

$$\begin{aligned} \mathbb{E}_n[\nabla f_{i_n}(\mathbf{x}_n)] &= \sum_{i=1}^{m} \nabla f_i(\mathbf{x}_n) P(i_n = i) \\ &= \frac{1}{m} \sum_{i=1}^{m} \nabla f_i(\mathbf{x}_n) \;=\; \nabla f(\mathbf{x}_n) \end{aligned} \tag{4.49}$$

This is the crucial property that makes SGD work. For a full proof, see e.g. [7].

4.5. The Levenberg–Marquardt Algorithm, for Nonlinear Least-Squares Problems

The **Levenberg-Marquardt algorithm** (LMA), a.k.a. the **damped least-squares (DLS)** method, is used for the solution of **nonlinear least-squares problems** which arise especially in curve fitting.

- In fitting a function $\widehat{y}(x;\mathbf{p})$ of an independent variable x and a parameter vector $\mathbf{p}\in\mathbb{R}^n$ to a set of m data points (x_i,y_i), it is customary and convenient to minimize the **sum of the weighted squares of the errors** (or **weighted residuals**) between the measured data y_i and the curve-fit function $\widehat{y}(x_i;\mathbf{p})$.

$$\begin{aligned} f(\mathbf{p}) &= \sum_{i=1}^{m}\left[\frac{y_i-\widehat{y}(x_i;\mathbf{p})}{\eta_i}\right]^2 \\ &= (\mathbf{y}-\widehat{\mathbf{y}}(\mathbf{p}))^{\mathbf{T}}\boldsymbol{W}\,(\mathbf{y}-\widehat{\mathbf{y}}(\mathbf{p})) \end{aligned} \tag{4.50}$$

  where η_i is the measurement error for y_i and the weighting matrix $\boldsymbol{W}$ is defined as

$$\boldsymbol{W}=\operatorname{diag}\{1/\eta_i^2\}\in\mathbb{R}^{m\times m}.$$

- However, more formally, $\boldsymbol{W}$ can be set to the inverse of the measurement error covariance matrix; more generally, the weights can be set to pursue other curve-fitting goals.

Definition 4.28. The **measurement error** (also called the **observational error**) is the difference between a measured quantity and its true value. It includes **random error** and **systematic error** (caused by a mis-calibrated instrument that affects all measurements).

Note: The **goodness-of-fit measure** in (4.50) is called the **chi-squared error criterion** because the sum of squares of normally-distributed variables is distributed as the χ-squared distribution.

If the function $\widehat{y}(x; \mathbf{p})$ is **nonlinear** in the model parameters $\mathbf{p}$, then **the minimization** of the χ-squared function f with respect to the parameters must be carried out **iteratively**:

$$\mathbf{p} := \mathbf{p} + \Delta\mathbf{p}. \tag{4.51}$$

The goal of each iteration is to find the parameter update $\Delta\mathbf{p}$ that reduces f. We will begin with the gradient descent method and the Gauss-Newton method.

4.5.1. The gradient descent method

Recall: The gradient descent method is a general minimization method which updates parameter values in the "steepest downhill" direction: the direction opposite to the gradient of the objective function.

- The gradient descent method converges well for problems with simple objective functions.
- For problems with thousands of parameters, gradient descent methods are **sometimes the only workable choice**.

The gradient of the objective function with respect to the parameters is

$$\begin{aligned} \frac{\partial}{\partial \mathbf{p}} f &= 2(\mathbf{y} - \widehat{\mathbf{y}}(\mathbf{p}))^T \boldsymbol{W} \frac{\partial}{\partial \mathbf{p}} (\mathbf{y} - \widehat{\mathbf{y}}(\mathbf{p})) \\ &= -2(\mathbf{y} - \widehat{\mathbf{y}}(\mathbf{p}))^T \boldsymbol{W} \left[\frac{\partial \widehat{\mathbf{y}}(\mathbf{p})}{\partial \mathbf{p}} \right] \\ &= -2(\mathbf{y} - \widehat{\mathbf{y}}(\mathbf{p}))^T \boldsymbol{W}\, \boldsymbol{J}, \end{aligned} \tag{4.52}$$

where $\boldsymbol{J} = \frac{\partial \widehat{\mathbf{y}}(\mathbf{p})}{\partial \mathbf{p}} \in \mathbb{R}^{m \times n}$ is the **Jacobian matrix**. The parameter update $\Delta\mathbf{p}$ that moves the parameters in the direction of steepest descent is given by

$$\Delta\mathbf{p}_{\text{gd}} = \gamma\, \boldsymbol{J}^T\, \boldsymbol{W}\, (\mathbf{y} - \widehat{\mathbf{y}}(\mathbf{p})), \tag{4.53}$$

where $\gamma > 0$ is the step length.

4.5.2. The Gauss-Newton method

The **Gauss-Newton method** is a method for minimizing a **sum-of-squares objective function**.

- It assumes that the objective function is **approximately quadratic** near the minimizer [6], and utilizes an **approximate Hessian**.
- For moderately-sized problems, the Gauss-Newton method typically converges much faster than gradient-descent methods [52].

Algorithm Derivation

- The function evaluated with perturbed model parameters may be locally approximated through a **first-order Taylor series expansion**.

$$\widehat{\mathbf{y}}(\mathbf{p}+\Delta\mathbf{p}) \approx \widehat{\mathbf{y}}(\mathbf{p}) + \left[\frac{\partial\widehat{\mathbf{y}}(\mathbf{p})}{\partial\mathbf{p}}\right]\Delta\mathbf{p} = \widehat{\mathbf{y}}(\mathbf{p}) + \boldsymbol{J}\Delta\mathbf{p}. \tag{4.54}$$

- Substituting the approximation into (4.50), p. 89, we have

$$\begin{aligned} f(\mathbf{p}+\Delta\mathbf{p}) \approx\ & \mathbf{y}^T\boldsymbol{W}\,\mathbf{y} - 2\mathbf{y}^T\boldsymbol{W}\,\widehat{\mathbf{y}}(\mathbf{p}) + \widehat{\mathbf{y}}(\mathbf{p})^T\boldsymbol{W}\,\widehat{\mathbf{y}}(\mathbf{p}) \\ & -2(\mathbf{y}-\widehat{\mathbf{y}}(\mathbf{p}))^T\boldsymbol{W}\,\boldsymbol{J}\Delta\mathbf{p} + (\boldsymbol{J}\Delta\mathbf{p})^T\boldsymbol{W}\,\boldsymbol{J}\Delta\mathbf{p}. \end{aligned} \tag{4.55}$$

  Note: The above approximation for $f(\mathbf{p}+\Delta\mathbf{p})$ is **quadratic** in the parameter perturbation $\Delta\mathbf{p}$.

- The parameter update $\Delta\mathbf{p}$ can be found from $\partial f/\partial\Delta\mathbf{p} = 0$:

$$\frac{\partial}{\partial\Delta\mathbf{p}} f(\mathbf{p}+\Delta\mathbf{p}) \approx -2(\mathbf{y}-\widehat{\mathbf{y}}(\mathbf{p}))^T\boldsymbol{W}\,\boldsymbol{J} + 2(\boldsymbol{J}\Delta\mathbf{p})^T\boldsymbol{W}\boldsymbol{J} = 0, \tag{4.56}$$

  and therefore the resulting normal equation for the Gauss-Newton update reads

$$[\boldsymbol{J}^T\boldsymbol{W}\,\boldsymbol{J}]\,\Delta\mathbf{p}_{\text{gn}} = \boldsymbol{J}^T\boldsymbol{W}(\mathbf{y}-\widehat{\mathbf{y}}(\mathbf{p})). \tag{4.57}$$

Note: The matrix $\boldsymbol{J}^T\boldsymbol{W}\,\boldsymbol{J} \in \mathbb{R}^{n\times n}$ is an **approximate Hessian** of the objective function. Here, we require $m \geq n$; otherwise, the approximate Hessian must be singular.

4.5.3. The Levenberg-Marquardt algorithm

The **Levenberg-Marquardt algorithm** *adaptively* varies the parameter updates between the **gradient descent** and the **Gauss-Newton** methods:

$$[\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J} + \lambda\boldsymbol{I}]\,\Delta\mathbf{p}_{\text{lm}} = \boldsymbol{J}^T\boldsymbol{W}(\mathbf{y} - \widehat{\mathbf{y}}(\mathbf{p})), \tag{4.58}$$

where $\lambda \geq 0$ is the **damping parameter**. Small values of λ result in a Gauss-Newton update and large values of it result in a gradient descent update.

Remark 4.29. **Implementation of the Levenberg-Marquardt Algorithm**.

- The damping parameter λ is **often initialized to be large** so that first updates are small steps in the steepest-descent direction.
- **As the solution improves, λ is decreased**; the Levenberg-Marquardt method approaches the Gauss-Newton method, and the solution typically accelerates to the local minimum [49, 52].
- If any iteration happens to result in **a bad approximation**, e.g.,

  $$f(\mathbf{p} + \Delta\mathbf{p}_{\text{lm}}) > f(\mathbf{p}),$$

  then **λ is increased.**

Acceptance of the Step

There have been many variations of the Levenberg-Marquardt method, particularly for **acceptance criteria**.

Example 4.30. **Acceptance Criterion**. Recall (4.50) and (4.54):

$$\begin{aligned} f(\mathbf{p}) &= \big(\mathbf{y}-\widehat{\mathbf{y}}(\mathbf{p})\big)^T \boldsymbol{W}\,\big(\mathbf{y}-\widehat{\mathbf{y}}(\mathbf{p})\big), && (4.50)\\ \widehat{\mathbf{y}}(\mathbf{p}+\Delta\mathbf{p}_{\rm lm}) &\approx \widehat{\mathbf{y}}(\mathbf{p})+\boldsymbol{J}\Delta\mathbf{p}_{\rm lm}. && (4.54)\end{aligned}$$

Then the **Sum of Squared Error (SSE)**, $f(\mathbf{p}+\Delta\mathbf{p}_{\rm lm})$, can be approximated by

$$\begin{aligned} f(\mathbf{p}+\Delta\mathbf{p}_{\rm lm}) &= \big(\mathbf{y}-\widehat{\mathbf{y}}(\mathbf{p}+\Delta\mathbf{p}_{\rm lm})\big)^T \boldsymbol{W}\,\big(\mathbf{y}-\widehat{\mathbf{y}}(\mathbf{p}+\Delta\mathbf{p}_{\rm lm})\big)\\ &\approx \big(\mathbf{y}-[\widehat{\mathbf{y}}(\mathbf{p})+\boldsymbol{J}\Delta\mathbf{p}_{\rm lm}]\big)^T \boldsymbol{W}\,\big(\mathbf{y}-[\widehat{\mathbf{y}}(\mathbf{p})+\boldsymbol{J}\Delta\mathbf{p}_{\rm lm}]\big). \end{aligned} \tag{4.59}$$

- At the k-th step, we first compute

$$\begin{aligned} \rho_k(\Delta\mathbf{p}_{\rm lm}) &= \frac{f(\mathbf{p})-f(\mathbf{p}+\Delta\mathbf{p}_{\rm lm})}{f(\mathbf{p})-\big(\mathbf{y}-\widehat{\mathbf{y}}-\boldsymbol{J}\Delta\mathbf{p}_{\rm lm}\big)^T \boldsymbol{W}\,\big(\mathbf{y}-\widehat{\mathbf{y}}-\boldsymbol{J}\Delta\mathbf{p}_{\rm lm}\big)}\\ &= \frac{f(\mathbf{p})-f(\mathbf{p}+\Delta\mathbf{p}_{\rm lm})}{\Delta\mathbf{p}_{\rm lm}^T\,(\lambda_k\Delta\mathbf{p}_{\rm lm}+\boldsymbol{J}^T\boldsymbol{W}(\mathbf{y}-\widehat{\mathbf{y}}(\mathbf{p}))}. \qquad [\Leftarrow (4.58)] \end{aligned} \tag{4.60}$$

- Then the step is accepted when $\rho_k(\Delta\mathbf{p}_{\rm lm})>\varepsilon_0$, for a threshold $\varepsilon_0>0$.

An example implementation reads

$$\begin{aligned} &\text{Initialize } \mathbf{p}_0,\ \lambda_0, \text{ and } \varepsilon_0;\ \ (\text{e.g. } \lambda_0=0.01 \ \&\ \varepsilon_0=0.1)\\ &\text{Compute } \Delta\mathbf{p}_{\rm lm} \text{ from (4.58)};\\ &\text{Evaluate } \rho_k \text{ from (4.60)};\\ &\text{If } \rho_k>\varepsilon_0:\\ &\qquad \mathbf{p}_{k+1}=\mathbf{p}_k+\Delta\mathbf{p}_{\rm lm};\ \lambda_{k+1}=\lambda_k\cdot\max[1/3,1-(2\rho_k)^3)];\ \nu_k=2;\\ &\text{otherwise: } \lambda_{k+1}=\lambda_k\,\nu_k;\ \ \nu_{k+1}=2\nu_k; \end{aligned} \tag{4.61}$$

Exercises for Chapter 4

4.1. **(Gradient descent method)**. Implement the gradient descent algorithm (4.15) and the gradient descent algorithm with backtracking line search (4.19).

(a) Compare their performances with the Rosenbrock function in 2D (4.2).
(b) Find an effective strategy for initial step size estimate for (4.19).

4.2. **(Net effect of the inverse Hessian matrix)**. Verify the claim in Example 4.21.

4.3. **(Newton's method)**. Implement a line search version of the Newton's method (4.32) with the Rosenbrock function in 2D.

(a) Recall the results in Exercise 1. With the backtracking line search, is the Newton's method better than the gradient descent method?
(b) Now, we will approximate the Hessian matrix by its diagonal. That is,

$$\mathcal{D}_n = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x^2} & 0 \\ 0 & \dfrac{\partial^2 f}{\partial y^2} \end{bmatrix}(\mathbf{x}_n) \;\approx\; \nabla^2 f(\mathbf{x}_n) \overset{\text{def}}{=\!=} \begin{bmatrix} \dfrac{\partial^2 f}{\partial x^2} & \dfrac{\partial^2 f}{\partial x \partial y} \\ \dfrac{\partial^2 f}{\partial y \partial x} & \dfrac{\partial^2 f}{\partial y^2} \end{bmatrix}(\mathbf{x}_n). \tag{4.62}$$

How does the Newton's method perform when the Hessian matrix is replaced by $\mathcal{D}_n$?

4.4. **(BFGS update)**. Consider $\boldsymbol{H}_{n+1}$ and $\boldsymbol{B}_{n+1}$ in (4.41) and (4.42), respectively:

$$\begin{aligned} \boldsymbol{H}_{n+1} &= \boldsymbol{H}_n + \frac{\mathrm{y}_n \mathrm{y}_n^T}{\mathrm{y}_n \cdot \boldsymbol{s}_n} - \frac{(\boldsymbol{H}_n \boldsymbol{s}_n)(\boldsymbol{H}_n \boldsymbol{s}_n)^T}{\boldsymbol{s}_n \cdot \boldsymbol{H}_n \boldsymbol{s}_n}, \\ \boldsymbol{B}_{n+1} &= \Big(I - \frac{\boldsymbol{s}_n \mathrm{y}_n^T}{\mathrm{y}_n \cdot \boldsymbol{s}_n}\Big) \boldsymbol{B}_n \Big(I - \frac{\mathrm{y}_n \boldsymbol{s}_n^T}{\mathrm{y}_n \cdot \boldsymbol{s}_n}\Big) + \frac{\boldsymbol{s}_n \boldsymbol{s}_n^T}{\mathrm{y}_n \cdot \boldsymbol{s}_n}. \end{aligned}$$

(a) Verify the the secant condition $\boldsymbol{H}_{n+1} \boldsymbol{s}_n = \mathrm{y}_n$.
(b) Verify $\boldsymbol{H}_{n+1} \boldsymbol{B}_{n+1} = I$, assuming that $\boldsymbol{H}_n \boldsymbol{B}_n = I$.

Continued on the next page $\Longrightarrow$

4.5. **(Curve fitting; Optional for undergraduates)**. Consider a set of data consisting of four points

	1	2	3	4
x_i	0.0	1.0	2.0	3.0
y_i	1.1	2.6	7.2	21.1

Fit the data with a fitting function of the form

$$\widehat{y}(x, \mathbf{p}) = a\, e^{bx}, \quad \text{where} \quad \mathbf{p} = [a, b], \tag{4.63}$$

by minimizing the sum of the square-errors:

(a) Implement the three algorithms introduced in Section 4.5: the gradient descent method, the Gauss-Newton method, and the Levenberg-Marquardt method.
(b) Ignore the weight vector $\boldsymbol{W}$, i.e., set $\boldsymbol{W} = \boldsymbol{I}$.
(c) For each method, set $\mathbf{p}_0 = [a_0, b_0] = [1.0, 0.8]$.
(d) Discuss how to choose γ for the gradient descent and λ for the Levenberg-Marquardt.

***Hint*:** *The Jacobian for this example must be in* $\mathbb{R}^{4\times 2}$*; more precisely,*

$$\boldsymbol{J} = \frac{\partial}{\partial \mathbf{p}} \widehat{\mathbf{y}}(x, \mathbf{p}) = \begin{bmatrix} 1 & 0 \\ e^{b} & a\, e^{b} \\ e^{2b} & 2a\, e^{2b} \\ e^{3b} & 3a\, e^{3b} \end{bmatrix},$$

because we have $\widehat{\mathbf{y}}(x, \mathbf{p}) = [a,\ a\, e^{b},\ a\, e^{2b},\ a\, e^{3b}]^T$ *from (4.63) and* $\{x_i\}$.

CHAPTER 5

Popular Machine Learning Classifiers

In this chapter, we will study a selection of popular and powerful machine learning algorithms, which are commonly used in academia as well as in the industry. While learning about the differences between several supervised learning algorithms for classification, we will also develop an intuitive appreciation of their individual strengths and weaknesses.

The topics that we will learn about throughout this chapter are as follows:

- Introduction to the concepts of popular classification algorithms such as **logistic regression**, **support vector machine** (SVM), **decision trees**, and k**-nearest neighbors**.
- Questions to ask when selecting a machine learning algorithm
- Discussions about the strengths and weaknesses of classifiers with linear and nonlinear decision boundaries

Contents of Chapter 5

5.1. Logistic Sigmoid Function 99
5.2. Classification via Logistic Regression 103
5.3. Support Vector Machine 110
5.4. Decision Trees 130
5.5. k-Nearest Neighbors 137
Exercises for Chapter 5 139

Choosing a classification algorithm

Choosing an appropriate classification algorithm for a particular problem task requires practice:

- Each algorithm has **its own quirks/characteristics** and is based on certain assumptions.
- **No Free Lunch theorem**: No single classifier works best across all possible scenarios.
- In practice, it is recommended that you **compare the performance of at least a handful of different learning algorithms** to select **the best model** for the particular problem.

Eventually, the performance of a classifier, computational power as well as predictive power, depends heavily on the underlying data that are available for learning. The **five main steps** that are involved in training a machine learning algorithm can be summarized as follows:

1. Selection of features.
2. Choosing a performance metric.
3. Choosing a classifier and optimization algorithm.
4. Evaluating the performance of the model.
5. Tuning the algorithm.

5.1. Logistic Sigmoid Function

A **logistic sigmoid function** (or **logistic curve**) is a common “S" shape curve with equation:

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}, \tag{5.1}$$

where L denotes the curve's maximum value, x_0 is the sigmoid's midpoint, and k is the logistic growth rate or steepness of the curve.

In statistics, the **logistic model** is a widely used statistical model that uses a logistic function to model a binary dependent variable; many more complex extensions exist.

5.1.1. The standard logistic sigmoid function

Setting $L = 1$, $k = 1$, and $x_0 = 0$ gives the **standard logistic sigmoid function**:

$$s(x) = \frac{1}{1 + e^{-x}}. \tag{5.2}$$

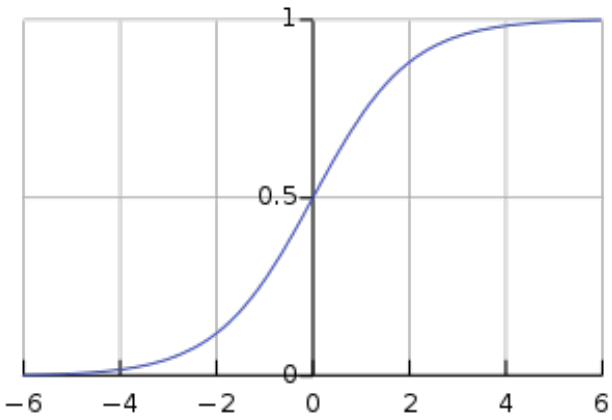


Figure 5.1: Standard logistic **sigmoid function** $s(x) = 1/(1 + e^{-x})$.

Remark 5.1. (The standard logistic sigmoid function):

$$s(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$$

- The standard logistic function is the solution of the simple first-order non-linear ordinary differential equation

$$\frac{d}{dx}y = y(1-y), \quad y(0) = \frac{1}{2}. \tag{5.3}$$

  It can be verified easily as

$$\boldsymbol{s'(x)} = \frac{e^x(1+e^x) - e^x \cdot e^x}{(1+e^x)^2} = \frac{e^x}{(1+e^x)^2} = \boldsymbol{s(x)(1-s(x))}. \tag{5.4}$$

- s' is even: $s'(-x) = s'(x)$.
- **Rotational symmetry** about $(0, 1/2)$:

$$s(x) + s(-x) = \frac{1}{1+e^{-x}} + \frac{1}{1+e^x} = \frac{2+e^x+e^{-x}}{2+e^x+e^{-x}} \equiv 1. \tag{5.5}$$

- $\int s(x)\,dx = \int \frac{e^x}{1+e^x}\,dx = \ln(1+e^x)$, which is known as the **softplus function** in **artificial neural networks**. It is a smooth approximation of the the **rectifier** (an activation function) defined as

$$f(x) = x^+ = \max(x, 0). \tag{5.6}$$

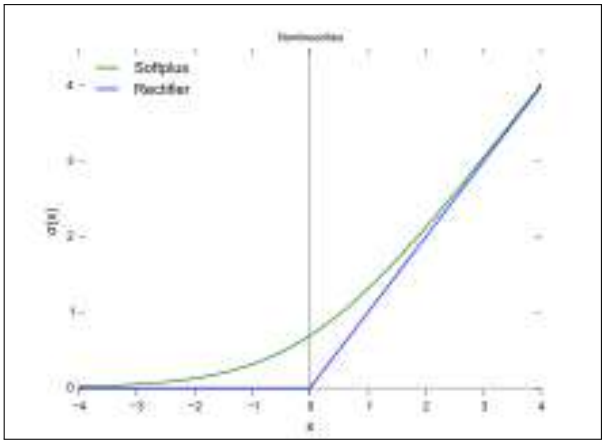


Figure 5.2: The rectifier and it smooth approximation, softplus function $\ln(1+e^x)$.

5.1.2. The logit function

Logistic regression uses the sigmoid function for activation. We first wish **to explain the idea behind logistic regression as a probabilistic model**.

- Let p be the probability of a particular event (having class label $y = 1$).
- Then the **odds ratio** of the particular event is defined as

$$\frac{p}{1-p}.$$

- We can then define the **logit** function, which is simply the logarithm of the odds ratio (**log-odds**):

$$\text{logit}(p) = \ln \frac{p}{1-p}. \tag{5.7}$$

- The logit function takes input values in $(0, 1)$ and transforms them to values over the entire real line,

  which we can use **to express a linear relationship between feature values and the log-odds**:

$$\text{logit}(p(y=1|\mathbf{x})) \; = \; w_0x_0 + w_1x_1 + \cdots + w_mx_m = \mathbf{w}^T\mathbf{x}, \tag{5.8}$$

  where $p(y = 1|\mathbf{x})$ is the conditional probability that a particular sample (given its features x) belongs to class 1.

Remark 5.2. What we are actually interested in is

predicting the probability

that a certain sample belongs to a particular class, which is the inverse form of the logit function:

$$p(y = 1|\mathbf{x}) = \text{logit}^{-1}(\mathbf{w}^T\mathbf{x}). \tag{5.9}$$

Quesiton. What is the inverse of the logit function?

Example 5.3. Find the inverse of the logit function

$$\text{logit}(p) = \ln \frac{p}{1-p}.$$

Solution.

Ans: $\text{logit}^{-1}(z) = \dfrac{1}{1+e^{-z}}$, the standard logistic sigmoid function.

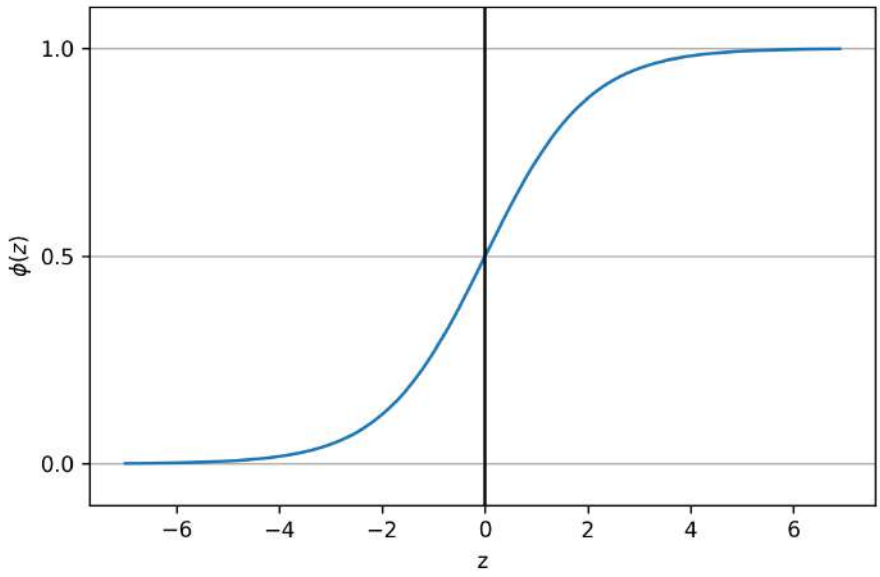


Figure 5.3: The standard logistic sigmoid function, again.

Note: **The Sigmoid Function as an Activation Function**

- When the standard logistic sigmoid function is adopted as an activation function, the prediction may be considered as the ***probability*** that a certain sample belongs to a particular class.
- This explains why the logistic sigmoid function is one of most popular activation functions.

5.2. Classification via Logistic Regression

Logistic regression is a probabilistic model.

- **Logistic regression maximizes the likelihood of the parameter** w; in realization, it is similar to **Adaline**.
- Only the difference is the **activation function** (the sigmoid function), as illustrated in the figure:

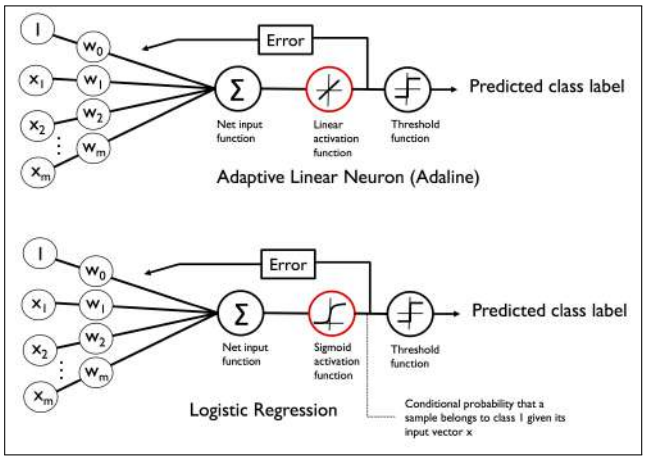


Figure 5.4: Adaline vs. Logistic regression.

- **The prediction** (the output of the sigmoid function) is interpreted as the **probability** of a particular sample belonging to class 1,

$$\phi(z) = p(y = 1|\mathbf{x}; \mathbf{w}), \tag{5.10}$$

given its features x parameterized by the weights w, $z = \mathbf{w}^T\mathbf{x}$.

Remark 5.4. Logistic Regression can be applied not only for **classification** (class labels) but also for **class-membership probability**.

- For example, logistic regression is used in **weather forecasting** (to predict the chance of rain).
- Similarly, it can be used to predict the **probability** that a patient has a particular disease given certain symptoms.
  - This is why logistic regression enjoys great popularity in the field of **medicine**.

5.2.1. The logistic cost function

Logistic regression incorporates a cost function in which **the likelihood is maximized**.[1]

Definition 5.5. *The **binomial distribution** with parameters n and $p \in [0, 1]$ is the discrete probability distribution of the number of successes in a sequence of n **independent** experiments, each asking a success–failure question, with probability of success being p.*

- The **probability** of getting exactly k successes in n trials is given by the probability mass function

$$f(k, n, p) = P(k; n, p) = {}_nC_k p^k (1-p)^{n-k}. \tag{5.11}$$

Definition 5.6. ***(Likelihood).*** *Let $X_1, X_2, \cdots, X_n$ have a joint density function $f(X_1, X_2, \cdots, X_n|\theta)$. Given $X_1 = x_1, X_2 = x_2, \cdots, X_n = x_n$ observed, the function of θ defined by*

$$L(\theta) = L(\theta|x_1, x_2, \cdots, x_n) = f(x_1, x_2, \cdots, x_n|\theta) \tag{5.12}$$

*is the **likelihood function**, or simply the **likelihood**.*

Note: The **likelihood** describes the joint probability of the observed data, as a function of the parameters of the chosen statistical model.

- The **likelihood function** indicates which parameter values are more **likely** than others, in the sense that they would make the observed data more probable.
- The **maximum likelihood estimator** selects the parameter values ($\theta = \mathbf{w}$) that give the observed data the largest possible probability.

[1] Note that the Adaline minimizes the sum-squared-error (SSE) cost function defined as $\mathcal{J}(\mathbf{w}) = \frac{1}{2}\sum_i \left(\phi(z^{(i)}) - y^{(i)}\right)^2$, where $z^{(i)} = \mathbf{w}^T\mathbf{x}^{(i)}$, using the gradient descent method; see Section 3.3.1.

Derivation of the Logistic Cost Function

- Assume that the individual samples in our dataset are **independent** of one another. Then we can define the **likelihood** L as

$$\begin{aligned} L(\mathbf{w}) &= P(\mathbf{y}|\mathbf{x};\mathbf{w}) = \Pi_{i=1}^{n} P(y^{(i)}|\mathbf{x}^{(i)};\mathbf{w}) \\ &= \Pi_{i=1}^{n} \left(\phi(z^{(i)})\right)^{y^{(i)}} \left(1-\phi(z^{(i)})\right)^{1-y^{(i)}}, \end{aligned} \tag{5.13}$$

  where $z^{(i)} = \mathbf{w}^T\mathbf{x}^{(i)}$.

- In practice, it is easier to maximize the (natural) log of this equation, which is called the **log-likelihood function**:

$$\ell(\mathbf{w}) = \ln(L(\mathbf{w})) = \sum_{i=1}^{n} \left[y^{(i)} \ln\left(\phi(z^{(i)})\right) + (1-y^{(i)}) \ln\left(1-\phi(z^{(i)})\right)\right]. \tag{5.14}$$

Remark 5.7. Log-Likelihood

- Firstly, applying the log function reduces the potential for **numerical underflow**, which can occur if the likelihoods are very small.
- Secondly, we can convert the product of factors into a summation of factors, which makes it easier to obtain the **derivative** of this function via the addition trick, as you may remember from calculus.
- We can adopt **the negation of the log-likelihood** as **a cost function $\mathcal{J}$** that can be minimized using gradient descent.

Now, we define the **logistic cost function** to be minimized:

$$\mathcal{J}(\mathbf{w}) = \sum_{i=1}^{n} \left[-\mathbf{y^{(i)}} \ln\left(\phi(\mathbf{z^{(i)}})\right) - (\mathbf{1}-\mathbf{y^{(i)}}) \ln\left(\mathbf{1}-\phi(\mathbf{z^{(i)}})\right)\right], \tag{5.15}$$

where $z^{(i)} = \mathbf{w}^T\mathbf{x}^{(i)}$.

Note: Looking at the equation, we can see that the first term becomes zero if $y^{(i)} = 0$, and the second term becomes zero if $y^{(i)} = 1$.

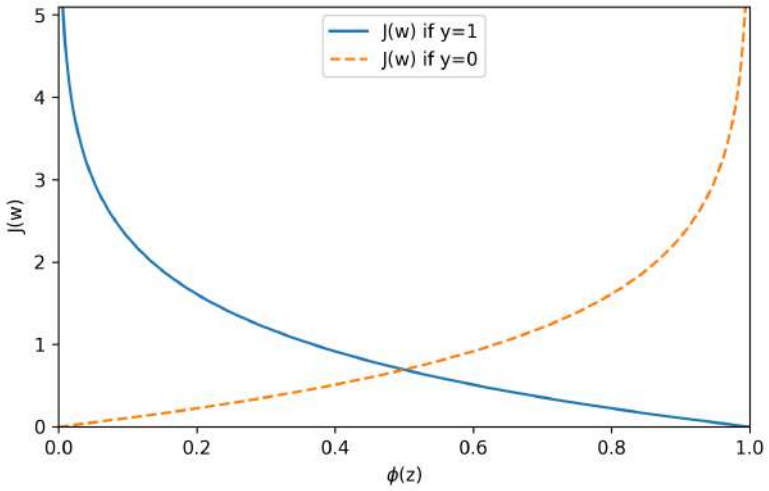


Figure 5.5: Plot of $\mathcal{J}(\mathbf{w})$, when $n = 1$ (one single-sample):

$$\mathcal{J}(\mathbf{w}) = \begin{cases} -\ln(\phi(\mathbf{z})), & \text{if } y = 1, \\ -\ln(\mathbf{1} - \phi(\mathbf{z})), & \text{if } y = 0. \end{cases}$$

Observation 5.8. We can see that

- **(Solid curve, in blue)**. If we correctly predict that a sample belongs to class 1, the cost approaches 0.
- **(Dashed curve, in orange)**. If we correctly predict $y = 0$, the cost also approaches 0.

- However, if the prediction is **wrong**, the cost goes towards **infinity**.
- Here, the main point is that we **penalize wrong predictions** with an increasingly larger cost, which will enforce the model to fit the sample.
- For general $n \geq 1$, it would try to fit all the samples in the training dataset.

5.2.2. Gradient descent learning for logistic regression

Let's start by calculating the partial derivative of the logistic cost function (5.15) with respect to the j–th weight, w_j:

$$\frac{\partial \mathcal{J}(\mathbf{w})}{\partial w_j} = \sum_{i=1}^{n} \left[-y^{(i)} \frac{1}{\phi(z^{(i)})} + (1-y^{(i)}) \frac{1}{1-\phi(z^{(i)})} \right] \frac{\partial \phi(z^{(i)})}{\partial w_j}, \tag{5.16}$$

where, using $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)}$ and (5.4),

$$\frac{\partial \phi(z^{(i)})}{\partial w_j} = \phi'(z^{(i)}) \frac{\partial z^{(i)}}{\partial w_j} = \phi(z^{(i)}) \left(1 - \phi(z^{(i)})\right) x_j^{(i)}.$$

Thus, if follows from the above and (5.16) that

$$\begin{aligned} \frac{\partial \mathcal{J}(\mathbf{w})}{\partial w_j} &= \sum_{i=1}^{n} \left[-y^{(i)} \left(1 - \phi(z^{(i)})\right) + (1-y^{(i)}) \phi(z^{(i)}) \right] x_j^{(i)} \\ &= -\sum_{i=1}^{n} \left[y^{(i)} - \phi(z^{(i)}) \right] x_j^{(i)} \end{aligned}$$

and therefore

$$\nabla \mathcal{J}(\mathbf{w}) = -\sum_{i=1}^{n} \left[y^{(i)} - \phi(z^{(i)}) \right] \mathbf{x}^{(i)}. \tag{5.17}$$

Algorithm 5.9. *Gradient descent learning for Logistic Regression is formulated as*

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \quad b := b + \Delta b, \tag{5.18}$$

*where $\eta > 0$ is the **step length** (learning rate) and*

$$\begin{aligned} \Delta \mathbf{w} &= -\eta \nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}, b) = \eta \sum_i \left[y^{(i)} - \phi(z^{(i)}) \right] \mathbf{x}^{(i)}, \\ \Delta b &= -\eta \nabla_b \mathcal{J}(\mathbf{w}, b) = \eta \sum_i \left[y^{(i)} - \phi(z^{(i)}) \right]. \end{aligned} \tag{5.19}$$

Note: The above gradient descent rule for Logistic Regression is of the same form as that of Adaline; see (3.15) on p. 56. Only the difference is the activation function ϕ.

5.2.3. Regularization: bias-variance tradeoff

- **Overfitting** is a common problem in ML.
  - If a model performs well on the training data but does not generalize well to unseen (test) data, then it is most likely **the sign of overfitting**.
  - Due to a **high variance**, from **randomness** (noise) in the training data.
  - **Variance** measures the consistency (or variability) of the model prediction for a particular sample instance.
- Similarly, our model can also suffer from **underfitting**.
  - Our model is **not complex enough** to capture the pattern in the training data well, and therefore also suffers from low performance on unseen data.
  - Due to a **high bias**.
  - **Bias** is the measure of the **systematic error** that is not due to randomness.

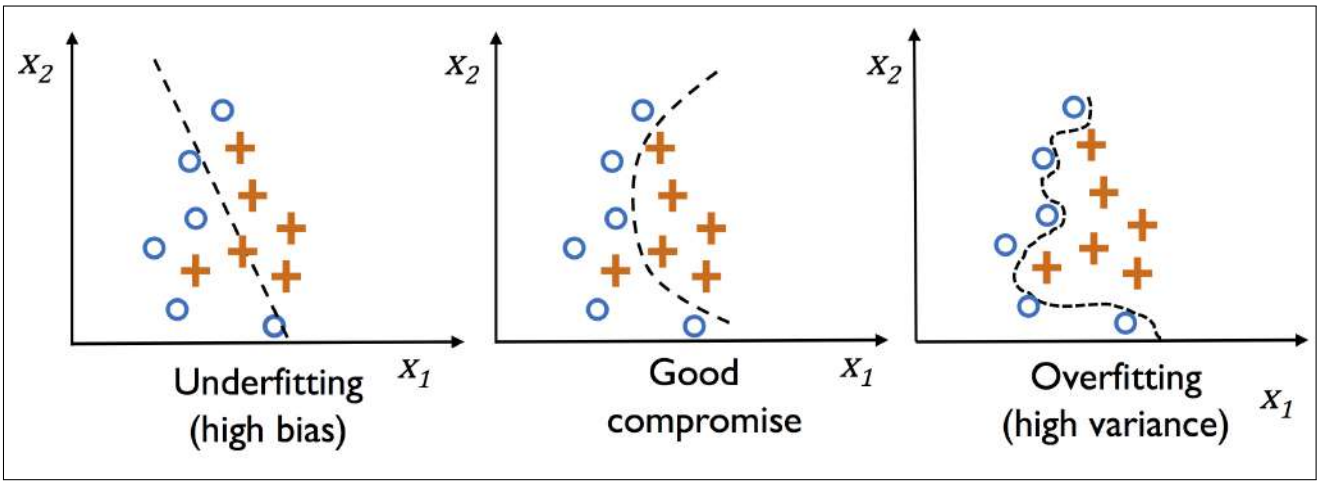


Figure 5.6

Regularization

- One way of finding a good **bias-variance tradeoff**.
- It is useful to prevent overfitting, also handling
  - collinearity (high correlation among features)
  - filter-out noise from data
  - **multiple local minima problem**
- The concept behind **regularization** is **to introduce additional information (bias)** to penalize extreme parameter (weight) values.
- The most common form of regularization is so-called L^2 **regularization** (sometimes also called L^2 **shrinkage** or **weight decay**):

$$\frac{\lambda}{2}\|\mathbf{w}\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2, \tag{5.20}$$

where λ is the regularization parameter.

The cost function for logistic regression can be regularized by adding a simple regularization term, which will *shrink the weights* during model training: for $z^{(i)} = \mathbf{w}^T\mathbf{x}^{(i)}$,

$$\mathcal{J}(\mathbf{w}) = \sum_{i=1}^{n}\left[-y^{(i)}\ln\left(\phi(z^{(i)})\right) - (1-y^{(i)})\ln\left(1-\phi(z^{(i)})\right)\right] + \frac{\lambda}{2}\|\mathbf{w}\|^2. \tag{5.21}$$

Note: **Regularization**

- Regularization is another reason why **feature scaling** such as **standardization** is important.
- For regularization to work properly, we need to ensure that **all our features are on comparable scales**.
- Then, via the regularization parameter λ, we can control how well we fit the training data while keeping the weights small. By increasing the value of λ, we increase the **regularization strength**.
- See § 6.3 for details on feature scaling.

5.3. Support Vector Machine

- **Support vector machine** (SVM), developed in 1995 by Cortes-Vapnik [12], can be considered as an extension of the Perceptron/Adaline, ***which maximizes the margin.***
- The **rationale** behind having decision boundaries with large margins is that they tend to have a **lower generalization error**, whereas **models with small margins are more prone to overfitting**.

5.3.1. Linear SVM

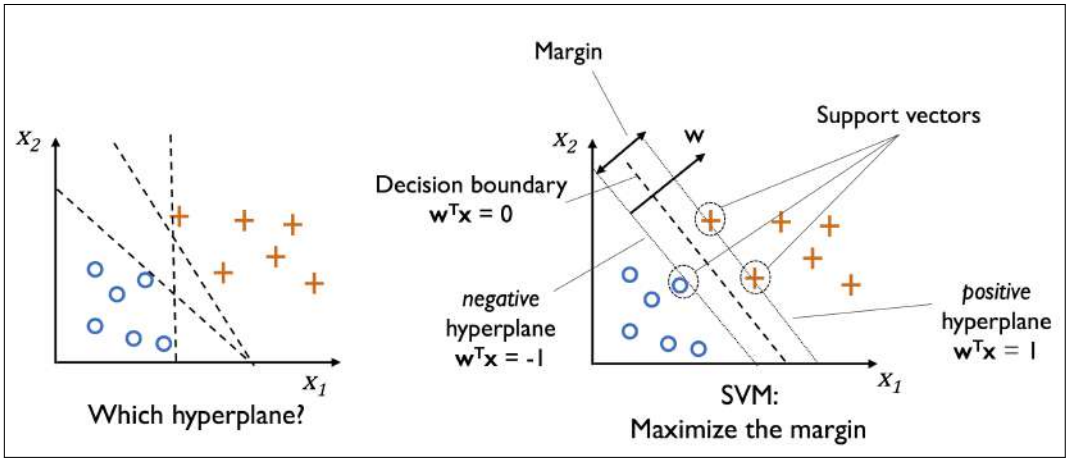


Figure 5.7: Linear support vector machine.

To find an optimal hyperplane that maximizes the margin, let's begin with considering the **positive** and **negative** hyperplanes that are parallel to the decision boundary:

$$\begin{aligned} w_0 + \mathbf{w}^T\mathbf{x}_+ &= 1, \\ w_0 + \mathbf{w}^T\mathbf{x}_- &= -1. \end{aligned} \tag{5.22}$$

where $\mathbf{w} = [w_1, w_2, \cdots, w_d]^T$. If we subtract those two linear equations from each other, then we have

$$\mathbf{w} \cdot (\mathbf{x}_+ - \mathbf{x}_-) = 2$$

and therefore

$$\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot (\mathbf{x}_+ - \mathbf{x}_-) = \frac{2}{\|\mathbf{w}\|}. \tag{5.23}$$

Note: $\mathbf{w} = [w_1, w_2, \cdots, w_d]^T$ is a **normal vector**[a] to the decision boundary (a hyperplane) so that the left side of (5.23) is the distance between the positive and negative hyperplanes.

[a] See Exercise 5.1.

Maximizing the distance (margin) is equivalent to minimizing its reciprocal $\frac{1}{2}\|\mathbf{w}\|$, or minimizing $\frac{1}{2}\|\mathbf{w}\|^2$.

Problem 5.10. *The **linear SVM** is formulated as*

$$\min_{\mathbf{w}, w_0} \frac{1}{2}\|\mathbf{w}\|^2, \quad \textit{subject to} \quad \begin{cases} w_0 + \mathbf{w}^T\mathbf{x}^{(i)} \geq 1 & \textit{if } y^{(i)} = 1, \\ w_0 + \mathbf{w}^T\mathbf{x}^{(i)} \leq -1 & \textit{if } y^{(i)} = -1. \end{cases} \tag{5.24}$$

Remark 5.11. The constraints in Problem 5.10 can be written as

$$y^{(i)}(w_0 + \mathbf{w}^T\mathbf{x}^{(i)}) - 1 \geq 0, \qquad \forall\, i. \tag{5.25}$$

- The beauty of linear SVM is that if the data is linearly separable, there is a unique global minimum value.
- An ideal SVM analysis should produce a hyperplane that completely separates the vectors (cases) into two non-overlapping classes.
- However, perfect separation may not be possible, or it may result in a model with so many cases that the model does not classify correctly.

Note: Constrained optimization problems such as (5.24) are typically solved using the method of Lagrange multipliers.

5.3.2. The method of Lagrange multipliers

In this subsection, we briefly consider Lagrange's method to solve the problem of the form

$$\min / \max_{\mathbf{x}} f(\mathbf{x}) \quad \text{subj.to} \quad g(\mathbf{x}) = c. \tag{5.26}$$

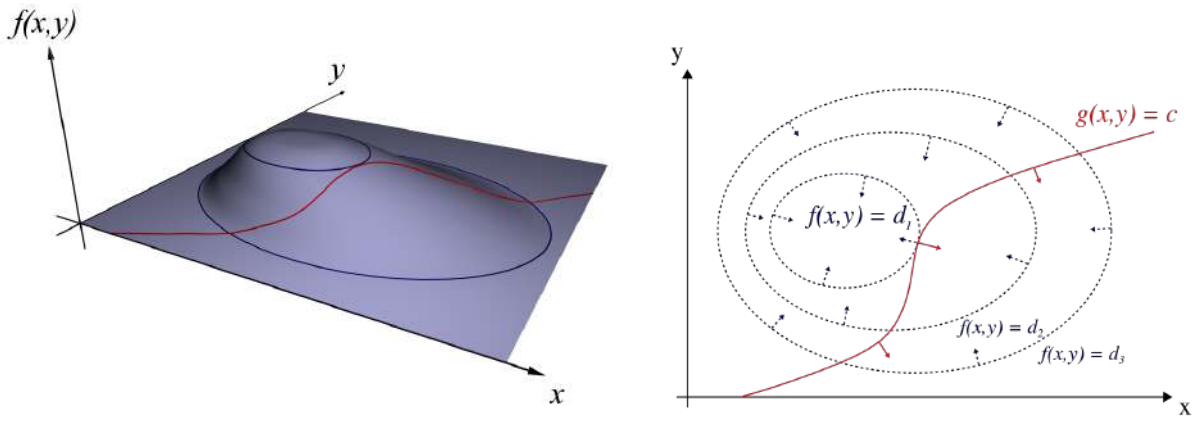


Figure 5.8: The method of Lagrange multipliers in $\mathbb{R}^2$: **∇f // ∇g, at optimum**.

Strategy 5.12. **(Method of Lagrange multipliers)**. For the maximum and minimum values of $\mathrm{f}(\mathrm{x})$ **subject to** $\mathrm{g}(\mathrm{x}) = \mathrm{c}$,

(a) Find x and λ such that

$$\nabla f(\mathbf{x}) = \lambda \nabla g(\mathbf{x}) \text{ and } g(\mathbf{x}) = c.$$

(b) Evaluate f at all these points, to find the maximum and minimum.

Self-study 5.13. Use the method of Lagrange multipliers to find the extreme values of $f(x, y) = x^2 + 2y^2$ on the circle $x^2 + y^2 = 1$.

Hint: $\nabla f = \lambda \nabla g \Longrightarrow \begin{bmatrix} 2x \\ 4y \end{bmatrix} = \lambda \begin{bmatrix} 2x \\ 2y \end{bmatrix}$. *Therefore,* $\begin{cases} 2x = 2x\,\lambda & ① \\ 4y = 2y\,\lambda & ② \\ x^2 + y^2 = 1 & ③ \end{cases}$

From ①, $x = 0$ ***or*** $\lambda = 1$.

Ans: min: $f(\pm 1, 0) = 1$; max: $f(0, \pm 1) = 2$

Lagrange multipliers – Dual variables

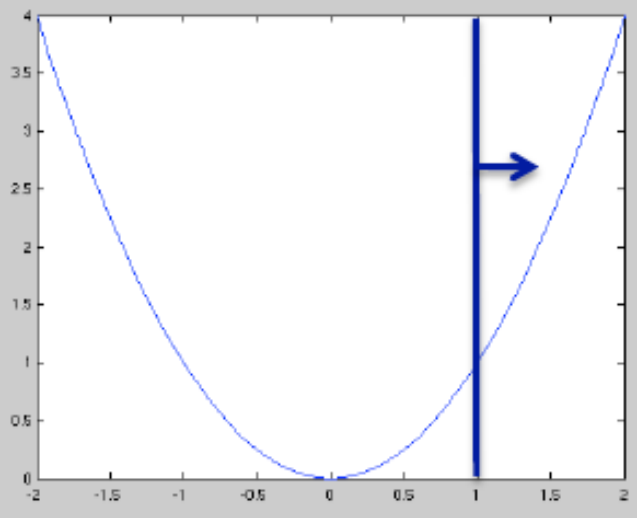


Figure 5.9: $\min_x x^2$ subj.to $x \geq 1$.

For simplicity, consider

$$\min_x x^2 \quad \text{subj.to } x \geq 1. \tag{5.27}$$

Rewriting the constraint

$$x - 1 \geq 0,$$

introduce **Lagrangian (objective)**:

$$\mathcal{L}(x, \alpha) = x^2 - \alpha\,(x - 1). \tag{5.28}$$

Now, consider

$$\min_x \max_\alpha \mathcal{L}(x, \alpha) \text{ subj.to } \alpha \geq 0. \tag{5.29}$$

Claim 5.14. The minimization problem (5.27) is equivalent to the min-max problem (5.29).

Proof. ① Let $x > 1$. $\Rightarrow \max_{\alpha \geq 0}\{-\alpha(x-1)\} = 0$ and $\alpha^* = 0$. Thus,

$$\mathcal{L}(x, \alpha) = x^2. \text{ (original objective)}$$

② Let $x = 1$. $\Rightarrow \max_{\alpha \geq 0}\{-\alpha(x-1)\} = 0$ and α is arbitrary. Thus, again,

$$\mathcal{L}(x, \alpha) = x^2. \text{ (original objective)}$$

③ Let $x < 1$. $\Rightarrow \max_{\alpha \geq 0}\{-\alpha(x-1)\} = \infty$. However, $\min_x$ won't make this happen! ($\min_x$ is fighting $\max_\alpha$) That is, when $x < 1$, the objective $\mathcal{L}(x, \alpha)$ becomes huge as α grows; then, $\min_x$ will push $x \nearrow 1$ or increase it to become $x \geq 1$. In other words, **$\min_x$ forces $\max_\alpha$ to behave, so constraints will be satisfied**. □

Now, the goal is **to solve (5.29)**. In the following, we will define the **dual problem** of (5.29), which is equivalent to the **primal problem**.

Recall: The min-max problem in (5.29), which is equivalent to the (original) primal problem:

$$\min_x \max_\alpha \mathcal{L}(x, \alpha) \quad \text{subj.to} \quad \alpha \geq 0, \qquad \text{(Primal)} \tag{5.30}$$

where

$$\mathcal{L}(x, \alpha) = x^2 - \alpha\,(x - 1).$$

Definition 5.15. *The **dual problem** of (5.30) is formulated by swapping* $\min_x$ *and* $\max_\alpha$ *as follows:*

$$\max_\alpha \min_x \mathcal{L}(x, \alpha) \quad \textit{subj.to} \quad \alpha \geq 0, \qquad \text{(Dual)} \tag{5.31}$$

The term $\min_x \mathcal{L}(x, \alpha)$ *is called the **Lagrange dual function** and the Lagrange multiplier* α *is also called the **dual variable**.*

How to solve it. For the Lagrange dual function $\min_x \mathcal{L}(x, \alpha)$, the minimum occurs where the gradient is equal to zero.

$$\frac{d}{dx}\mathcal{L}(x, \alpha) = 2x - \alpha = 0 \;\Rightarrow\; x = \frac{\alpha}{2}. \tag{5.32}$$

Plugging this to $\mathcal{L}(x, \alpha)$, we have

$$\mathcal{L}(x, \alpha) = \left(\frac{\alpha}{2}\right)^2 - \alpha\left(\frac{\alpha}{2} - 1\right) = \alpha - \frac{\alpha^2}{4}.$$

We can rewrite the dual problem (5.31) as

$$\max_{\alpha \geq 0} \left[\alpha - \frac{\alpha^2}{4}\right]. \qquad \text{(Dual)} \tag{5.33}$$

$\Rightarrow$ **the maximum is 1 when $\alpha^* = 2$ (for the dual problem).**
Plugging $\alpha = \alpha^*$ into (5.32) to get $x^* = 1$. Or, using the Lagrangian objective, we have

$$\mathcal{L}(x, \alpha) = x^2 - 2(x - 1) = (x - 1)^2 + 1. \tag{5.34}$$

$\Rightarrow$ **the minimum is 1 when $x^* = 1$ (for the primal problem).** □

5.3.3. Karush-Kuhn-Tucker conditions and Complementary slackness

Allowing **inequality constraints**, the **KKT approach** generalizes the **method of Lagrange multipliers** which allows only equality constraints.

Recall: The **linear SVM** formulated in Problem 5.10:

$$\min_{\mathbf{w},w_0} \frac{1}{2}\|\mathbf{w}\|^2, \quad \text{subj.to} \quad y^{(i)}(w_0 + \mathbf{w}^T\mathbf{x}^{(i)}) - 1 \geq 0, \ \ \forall\, i = 1, 2, \cdots, N. \qquad \text{(Primal)} \qquad (5.35)$$

To solve the problem, let's begin with its **Lagrangian**:

$$\mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{N} \alpha_i [y^{(i)}(w_0 + \mathbf{w}^T\mathbf{x}^{(i)}) - 1], \qquad (5.36)$$

where $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \cdots, \alpha_N]^T$, the **dual variables** (**Lagrange multipliers**).

- The primal problem of the SVM is formulated equivalently as

$$\min_{\mathbf{w},w_0} \max_{\alpha} \mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) \quad \text{subj.to} \quad \alpha \geq 0, \qquad \text{(Primal)} \qquad (5.37)$$

  while its dual problem reads

$$\max_{\alpha} \min_{\mathbf{w},w_0} \mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) \quad \text{subj.to} \quad \alpha \geq 0. \qquad \text{(Dual)} \qquad (5.38)$$

- Solve the "min" problem of (5.38) first, using calculus techniques.

Definition 5.16. **Karush-Kuhn-Tucker (KKT) conditions**
In optimization, the **KKT conditions** [36, 42] are **first derivative tests** for a solution in nonlinear programming to be optimized. It is also called the **first-order necessary conditions**.

Writing the **KKT conditions**, starting with Lagrangian stationarity, where we need to find the first-order derivatives w.r.t. $\mathbf{w}$ and w_0:

$$
\begin{aligned}
\nabla_{\mathbf{w}}\mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) &= \mathbf{w} - \sum_{i=1}^{N} \alpha_i y^{(i)} \mathbf{x}^{(i)} = 0 \;\Rightarrow\; \mathbf{w} = \sum_{i=1}^{N} \alpha_i y^{(i)} \mathbf{x}^{(i)}, \\
\frac{\partial}{\partial w_0}\mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) &= -\sum_{i=1}^{N} \alpha_i y^{(i)} = 0 \;\Rightarrow\; \sum_{i=1}^{N} \alpha_i y^{(i)} = 0, \\
&\alpha_i \geq 0, && \text{(dual feasibility)} \\
&\alpha_i \left[y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) - 1 \right] = 0, && \text{(complementary slackness)} \\
&y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) - 1 \geq 0. && \text{(primal feasibility)}
\end{aligned}
\tag{5.39}
$$

Complementary slackness will be discussed in detail on page 119.

Using the KKT conditions (5.39), we can simplify the Lagrangian:

$$
\begin{aligned}
\mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) &= \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{N} \alpha_i y^{(i)} w_0 - \sum_{i=1}^{N} \alpha_i y^{(i)} \mathbf{w}^T \mathbf{x}^{(i)} + \sum_{i=1}^{N} \alpha_i \\
&= \frac{1}{2}\|\mathbf{w}\|^2 - 0 - \mathbf{w}^T \mathbf{w} + \sum_{i=1}^{N} \alpha_i \\
&= -\frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{N} \alpha_i.
\end{aligned}
\tag{5.40}
$$

Again using the first KKT condition, we can rewrite the first term.

$$
\begin{aligned}
-\frac{1}{2}\|\mathbf{w}\|^2 &= -\frac{1}{2}\Big(\sum_{i=1}^{N} \alpha_i y^{(i)} \mathbf{x}^{(i)} \Big) \cdot \Big(\sum_{j=1}^{N} \alpha_j y^{(j)} \mathbf{x}^{(j)} \Big) \\
&= -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y^{(i)} y^{(j)}\, \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}.
\end{aligned}
\tag{5.41}
$$

Plugging (5.41) into the (simplified) Lagrangian (5.40), we see that the Lagrangian now depends on $\boldsymbol{\alpha}$ only.

Problem 5.17. *The **dual problem** of (5.35) is formulated as*

$$\max_{\boldsymbol{\alpha}} \Big[\sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y^{(i)} y^{(j)} \, \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \Big], \quad \textit{subj.to} \\ \begin{bmatrix} \alpha_i \geq 0, & \forall\, i, \\ \sum_{i=1}^{N} \alpha_i y^{(i)} = 0. \end{bmatrix} \tag{5.42}$$

Remark 5.18. (Solving the dual problem).

- We can solve the dual problem (5.42), by using either a generic quadratic programming solver or the **Sequential Minimal Optimization (SMO)**, which we will discuss in § 5.3.6, p. 128.
- For now, **assume** that we solved it to have $\boldsymbol{\alpha}^* = [\alpha_1^*, \cdots, \alpha_n^*]^T$.
- Then we can plug it into the first KKT condition to get

$$\mathbf{w}^* = \sum_{i=1}^{N} \alpha_i^* y^{(i)} \mathbf{x}^{(i)}. \tag{5.43}$$

- We still need to get w_0^*.

Remark 5.19. The objective function $\mathcal{L}(\boldsymbol{\alpha})$ in (5.42) is a linear combination of the dot products of data samples $\{\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}\}$, which will be used when we generalize the SVM for nonlinear decision boundaries; see § 5.3.5.

Support vectors

Assume momentarily that we have w_0^*. Consider the **complementary slackness KKT condition** along with the primal and dual feasibility conditions:

$$\begin{aligned} &\alpha_i^* \left[y^{(i)}(w_0^* + \mathbf{w}^{*T}\mathbf{x}^{(i)}) - 1 \right] = 0 \\ &\Rightarrow \begin{cases} \alpha_i^* > 0 \Rightarrow y^{(i)}(w_0^* + \mathbf{w}^{*T}\mathbf{x}^{(i)}) = 1 \\ \alpha_i^* < 0 \quad \text{(can't happen)} \\ y^{(i)}(w_0^* + \mathbf{w}^{T}\mathbf{x}^{(i)}) - 1 > 0 \Rightarrow \alpha_i^* = 0 \\ y^{(i)}(w_0^* + \mathbf{w}^{T}\mathbf{x}^{(i)}) - 1 < 0 \quad \text{(can't happen)}. \end{cases} \end{aligned} \tag{5.44}$$

We define the **optimal (scaled) scoring function**:

$$f^*(\mathbf{x}^{(i)}) = w_0^* + \mathbf{w}^{*T}\mathbf{x}^{(i)}. \tag{5.45}$$

Then

$$\begin{cases} \alpha_i^* > 0 & \Rightarrow \quad y^{(i)} f^*(\mathbf{x}^{(i)}) = \text{scaled margin} = 1, \\ y^{(i)} f^*(\mathbf{x}^{(i)}) > 1 & \Rightarrow \quad \alpha_i^* = 0. \end{cases} \tag{5.46}$$

Definition 5.20. The examples in the first category, for which the scaled margin is 1 and the constraints are active, are called **support vectors**. They are the closest to the decision boundary.

Finding the optimal value of w_0

To get w_0^*, use the primal feasibility condition:

$$y^{(i)}(w_0^* + \mathbf{w}^{*T}\mathbf{x}^{(i)}) \geq 1 \quad \text{and} \quad \min_i y^{(i)}(w_0^* + \mathbf{w}^{*T}\mathbf{x}^{(i)}) = 1.$$

If you take a positive support vector ($y^{(i)} = 1$), then

$$w_0^* = 1 - \min_{i:y^{(i)}=1} \mathbf{w}^{*T}\mathbf{x}^{(i)}. \tag{5.47}$$

Here, you'd better refer to **Summary of SVM** in Algorithm 5.27, p. 123.

Complementary Slackness

Definition 5.21. Types of Constraints

- A **binding constraint** is one where some optimal solution is on the hyperplane for the constraint (***tight***).
- A **non-binding constraint** is one where no optimal solution is on the line for the constraint (***loose/slack***).
- A **redundant constraint** is one whose removal would not change the feasible region.

Theorem 5.22. Complementary Slackness
Assume the primal problem (P) has a solution w^* *and the dual problem (D) has a solution* α^*.

(a) If $w_j^* > 0$, *then the* j*-th constraint in (D) is binding.*

(b) If $\alpha_i^* > 0$, *then the* i*-th constraint in (P) is binding.*

The term **complementary slackness** refers to a relationship between the slackness in a primal constraint and the slackness (positivity) of the associated dual variable.

- Notice that the number of variables in the dual is the same as the number of constraints in the primal, and the number of constraints in the dual is equal to the number of variables in the primal.
- This correspondence suggests that variables in one problem are complementary to constraints in the other.
- We say that ***a constraint has slack if it is not binding***.

Example 5.23. The **contrapositive** statement of Theorem 5.22 (b):

If the i-th constraint in (P) is not binding, then $\alpha_i^* = 0$.

or, equivalently,

If the i-th constraint in (P) has slack, then $\alpha_i^* = 0$.

See (5.46).

5.3.4. The inseparable case: Soft-margin classification

When the dataset is inseparable, there would be no separating hyperplane; there is no feasible solution to the linear SVM.

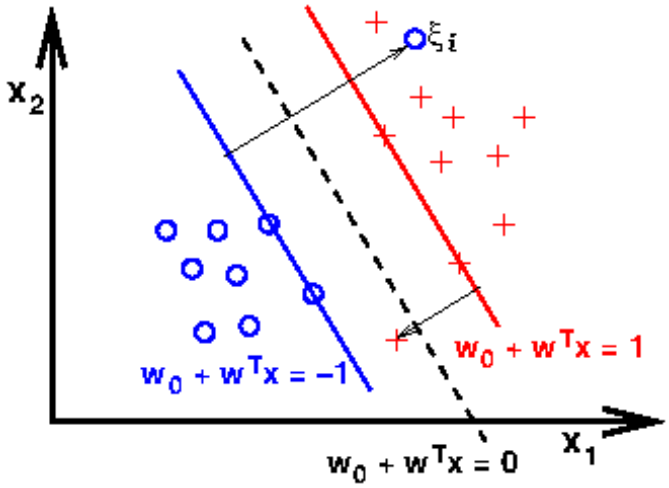


Figure 5.10: Slack variable: ξ_i.

Let's fix our SVM so it can accommodate the inseparable case.

- The new formulation involves the **slack variable**; it allows some instances to fall off the margin, but penalize them.
- So we are allowed to make mistakes now, but we pay a price.

Remark 5.24. The motivation for introducing the slack variable ξ is:

1. **The linear constraints need to be relaxed** for inseparable data.
2. Allow the optimization to converge
   - **under appropriate cost penalization,**
   - **in the presence of misclassifications.**

Such strategy of the SVM is called the **soft-margin classification**.

Recall: The **linear SVM** formulated in Problem 5.10:

$$\min_{\mathbf{w},w_0} \frac{1}{2}\|\mathbf{w}\|^2, \quad \text{subj.to} \quad y^{(i)}(w_0 + \mathbf{w}^T\mathbf{x}^{(i)}) - 1 \geq 0, \quad \forall\, i. \qquad \text{(Primal)} \qquad (5.48)$$

Let's change it to this new primal problem:

Problem 5.25. ***(Soft-margin classification)***. *The SVM with the slack variable is formulated as*

$$\min_{\mathbf{w},w_0,\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}\xi_i \quad \textit{subj.to} \quad \begin{cases} y^{(i)}(w_0 + \mathbf{w}^T\mathbf{x}^{(i)}) \geq 1 - \xi_i, \\ \xi_i \geq 0. \end{cases} \qquad \textit{(Primal)} \qquad (5.49)$$

Via the variable C, we can then control the penalty for misclassification. **Large values of C** correspond to large error penalties, whereas we are less strict about misclassification errors if we choose smaller values for C. We can then use the C parameter to control the width of the margin and therefore tune the **bias-variance trade-off**, as illustrated in the following figure:

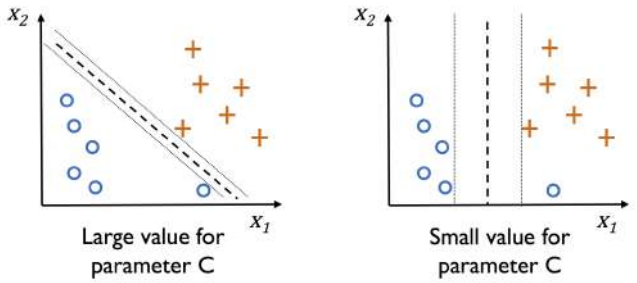


Figure 5.11: Bias-variance trade-off, via C.

The constraints allow some slack of size ξ_i, but we pay a price for it in the objective. That is,

if $y^{(i)}f(\mathbf{x}^{(i)}) \geq 1$, then $\xi_i = 0$ and penalty is 0. Otherwise, $y^{(i)}f(\mathbf{x}^{(i)}) = 1 - \xi_i$ and we pay price $\xi_i > 0$

.

The Dual for soft-margin classification

Form the Lagrangian of (5.49):

$$\begin{aligned}\mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{r}) \;&=\; \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N} r_i\xi_i \\ &\quad - \sum_{i=1}^{N}\alpha_i[y^{(i)}(w_0 + \mathbf{w}^T\mathbf{x}^{(i)}) - 1 + \xi_i],\end{aligned} \tag{5.50}$$

where α_i's and r_i's are Lagrange multipliers (constrained to be ≥ 0).

After some work, the dual turns out to be

Problem 5.26. The **dual problem** of (5.48) is formulated as

$$\max_{\boldsymbol{\alpha}} \Big[\sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y^{(i)}y^{(j)}\,\mathbf{x}^{(i)}\cdot\mathbf{x}^{(j)}\Big], \quad \text{subj.to} \quad \begin{bmatrix} 0 \leq \alpha_i \leq C, & \forall\, i, \\ \sum_{i=1}^{N}\alpha_i y^{(i)} = 0. \end{bmatrix} \tag{5.51}$$

So the only difference from the original problem's dual, (5.42), is that $\boldsymbol{\alpha_i \geq 0}$ **is changed to** $\boldsymbol{0 \leq \alpha_i \leq C}$**.** Neat!

See § 5.3.6, p. 128, for the solution of (5.51), using the SMO algorithm.

Algebraic expression for the dual problem:

Let

$$Z = \begin{bmatrix} y^{(1)}\mathbf{x}^{(1)} \\ y^{(2)}\mathbf{x}^{(2)} \\ \vdots \\ y^{(N)}\mathbf{x}^{(N)} \end{bmatrix} \in \mathbb{R}^{N\times m}, \quad \mathbf{1} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^{N}.$$

Then **dual problem** (5.51) can be written as

$$\max_{0\le\boldsymbol{\alpha}\le C} \left[\boldsymbol{\alpha}\cdot\mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^T ZZ^T\boldsymbol{\alpha}\right] \quad \text{subj.to} \quad \boldsymbol{\alpha}\cdot\mathbf{y} = 0. \tag{5.52}$$

Note:

- $G = ZZ^T \in \mathbb{R}^{N\times N}$ is called the **Gram matrix**. That is,

$$G_{ij} = y^{(i)}y^{(j)}\,\mathbf{x}^{(i)}\cdot\mathbf{x}^{(j)}. \tag{5.53}$$

- The optimization problem (5.52) is a typical **quadratic programming** (QP) problem.
- It admits **a unique solution**.

Algorithm 5.27. (Summary of SVM)

- **Training**
  - Compute Gram matrix: $G_{ij} = y^{(i)}y^{(j)}\,\mathbf{x^{(i)}}\cdot\mathbf{x^{(j)}}$
  - Solve QP to get $\boldsymbol{\alpha}^*$ (Chapter 11, or § 5.3.6)
  - Compute the weights: $\mathbf{w}^* = \sum_{i=1}^N \alpha_i^* y^{(i)}\mathbf{x}^{(i)}$ (5.43)
  - Compute the intercept: $w_0^* = 1 - \min_{i:y^{(i)}=1} \mathbf{w}^{*T}\mathbf{x}^{(i)}$ (5.47)
- **Classification** (for a new sample $\mathbf{x}$)
  - Compute $k_i = \mathbf{x}\cdot\mathbf{x^{(i)}}$ for support vectors $\mathbf{x}^{(i)}$
  - Compute $f(\mathbf{x}) = w_0^* + \sum_i \alpha_i^* y^{(i)} k_i \;\; (:= w_0^* + \mathbf{w}^{*T}\mathbf{x})$ (5.24)
  - Test $\text{sign}(f(\mathbf{x}))$.

5.3.5. Nonlinear SVM and kernel trick

Note: A reason why the SVM is popular is that

- It can be **easily kernelized** to solve nonlinear classification problems incorporating **linearly inseparable data**.

The basic idea behind **kernel methods** is

- To transform the data to **a higher-dimensional space where the data becomes linearly separable**.

For example, for the inseparable data set in Figure 5.12, we define

$$\phi(x_1, x_2) = (x_1, x_2, x_1^2 + x_2^2).$$

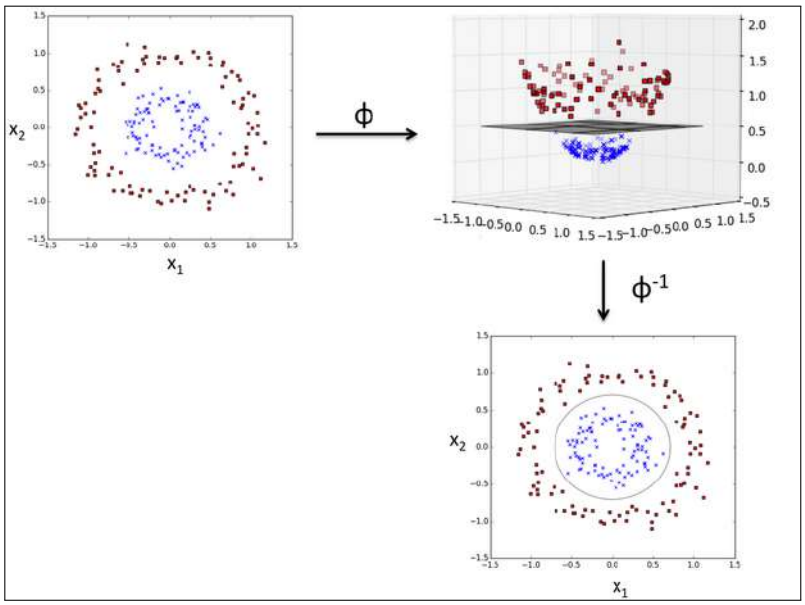


Figure 5.12: Inseparable dataset, feature expansion, and kernel SVM.

To solve a nonlinear problem using an SVM, we would

(a) ① **Transform the training data to a higher-dimensional space**, via a mapping ϕ, and ② **train a linear SVM model**.

(b) Then, **for new unseen data**, **classify using ① the same mapping ϕ to transform and ② the same linear SVM model**.

Kernel Trick

Recall: the **dual problem** to the soft-margin SVM given in (5.51):

$$\max_{\boldsymbol{\alpha}} \Big[\sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y^{(i)} y^{(j)} \, \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \Big], \quad \text{subj.to} \quad \begin{cases} 0 \leq \alpha_i \leq C, & \forall\, i, \\ \sum_{i=1}^{N} \alpha_i y^{(i)} = 0. \end{cases} \tag{5.54}$$

Observation 5.28. The objective is a linear combination of dot products $\{\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}\}$. Thus,

- If the kernel SVM transforms the data samples through ϕ, the dot product $\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}$ must be replaced by $\phi(\mathbf{x}^{(i)}) \cdot \phi(\mathbf{x}^{(j)})$.
- The dot product $\phi(\mathbf{x}^{(i)}) \cdot \phi(\mathbf{x}^{(j)})$ is performed in a higher-dimension, which may be costly.

Definition 5.29. In order **to save the expensive step of explicit computation** of this dot product (in a higher-dimension), we define a so-called **kernel function**:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \approx \phi(\mathbf{x}^{(i)}) \cdot \phi(\mathbf{x}^{(j)}). \tag{5.55}$$

One of the most widely used kernels is the **Radial Basis Function (RBF)** kernel or simply called the **Gaussian kernel**:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\Big(- \frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2} \Big) = \exp\Big(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2 \Big), \tag{5.56}$$

where $\gamma = 1/(2\sigma^2)$. Occasionally, the parameter γ plays an important role in controlling overfitting.

Note: Roughly speaking, the term **kernel** can be interpreted as a **similarity function** between a pair of samples.

This is the big picture behind the kernel trick.

Kernel SVM: It can also be summarized as in **Algorithm 5.27**, p. 123; **only the difference** is that dot products $\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}$ and $\mathbf{x} \cdot \mathbf{x}^{(i)}$ are replaced by $\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$ and $\mathcal{K}(\mathbf{x}, \mathbf{x}^{(i)})$, respectively.

Common Kernels

- Polynomial of degree exactly k (e.g. $k = 2$):
$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = (\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)})^k \tag{5.57}$$
- Polynomial of degree up to k: for some $c > 0$,
$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = (c + \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)})^k \tag{5.58}$$
- Sigmoid:
$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \tanh(a\, \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} + b) \tag{5.59}$$
- Gaussian RBF:
$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\Big(- \frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2} \Big) \tag{5.60}$$
- And many others: Fisher kernel, graph kernel, string kernel, ... **very active area of research!**

Example 5.30. (Quadratic kernels). Let $\mathcal{K}(\mathbf{x}, \mathbf{z}) = (c + \mathbf{x} \cdot \mathbf{z})^2$:

$$\begin{aligned} (c + \mathbf{x} \cdot \mathbf{z})^2 &= \Big(c + \sum_{j=1}^m x_j z_j\Big)\Big(c + \sum_{\ell=1}^m x_\ell z_\ell\Big) \\ &= c^2 + 2c \sum_{j=1}^m x_j z_j + \sum_{j=1}^m \sum_{\ell=1}^m x_j z_j x_\ell z_\ell \\ &= c^2 + \sum_{j=1}^m (\sqrt{2c} x_j)(\sqrt{2c} z_j) + \sum_{j,\ell=1}^m (x_j x_\ell)(z_j z_\ell). \end{aligned} \tag{5.61}$$

Define a **feature expansion** as

$$\phi([x_1, \cdots, x_m]) = [x_1^2, x_1 x_2, \cdots, x_m x_{m-1}, x_m^2, \sqrt{2c} x_1, \cdots, \sqrt{2c} x_m, c], \tag{5.62}$$

which is in $\mathbb{R}^{m^2+m+1}$. Then $\phi(\mathbf{x}) \cdot \phi(\mathbf{z}) = \mathcal{K}(\mathbf{x}, \mathbf{z}) = (c + \mathbf{x} \cdot \mathbf{z})^2$. □

Note: **Kernel Functions**

- Kernels may **not** be expressed as $\phi(\mathbf{x}) \cdot \phi(\mathbf{z})$.
- The mapping ϕ may transform $\mathbf{x}$ to **infinite dimensions**.

Summary **5.31.** **Linear Classifiers**

- They are a simple and popular way to learn a classifier
- They suffer from inefficient use of data, overfitting, or lack of expressiveness
- **SVM**
  - It can fix these problems using ① **maximum margins** and ② **feature expansion** (mapping to a higher-dimension).
  - In order to make feature expansion **computationally feasible**, we need the ③ **kernel trick**, which avoids writing out high-dimensional feature vectors explicitly.

Remark **5.32.** **Kernel Trick**

- There is no explicit feature expansion.
- The kernel $\mathcal{K}(\mathbf{x}, \mathbf{z})$ must be formulated **meaningfully**.

- The **kernel function** $\mathcal{K}$ must be considered as **a nonlinear measure for the data to become separable**.

5.3.6. Solving the dual problem with SMO

SMO (Sequential Minimal Optimization) is

- a type of coordinate ascent algorithm,
- but adapted to the SVM so that the solution always stays within the feasible region.

Recall: The **dual problem** of the soft-margin SVM, formulated in (5.51):

$$\max_{\boldsymbol{\alpha}} \Big[\sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y^{(i)} y^{(j)} \, \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \Big], \quad \text{subj.to} \quad \begin{bmatrix} 0 \leq \alpha_i \leq C, & \forall\, i, \\ \sum_{i=1}^{N} \alpha_i y^{(i)} = 0. \end{bmatrix} \tag{5.63}$$

Quesiton. Start with (5.63). Let's say you want to hold $\alpha_2, \cdots, \alpha_N$ fixed and take a coordinate step in the first direction. That is, change α_1 to maximize the objective in (5.63). Can we make any progress? Can we get a better feasible solution by doing this?

Turns out, no. Let's see why. Look at the constraint in (5.63), $\sum_{i=1}^{N} \alpha_i y^{(i)} = 0$. This means

$$\alpha_1 y^{(1)} = -\sum_{i=2}^{N} \alpha_i y^{(i)} \quad \Rightarrow \quad \alpha_1 = -y^{(1)} \sum_{i=2}^{N} \alpha_i y^{(i)}.$$

So, since $\alpha_2, \cdots, \alpha_N$ are fixed, α_1 is also fixed.

Thus, if we want to update any of the α_i's, **we need to update at least 2 of them simultaneously** to keep the solution feasible (i.e., to keep the constraints satisfied).

- Start with a feasible vector $\boldsymbol{\alpha}$.
- Let's **update α_1 and α_2**, holding $\alpha_3, \cdots, \alpha_N$ fixed.

Question: ***What values of α_1 and α_2 are we allowed to choose?***

- The constraint is: $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = -\sum_{i=3}^{N} \alpha_i y^{(i)} =: \xi$.

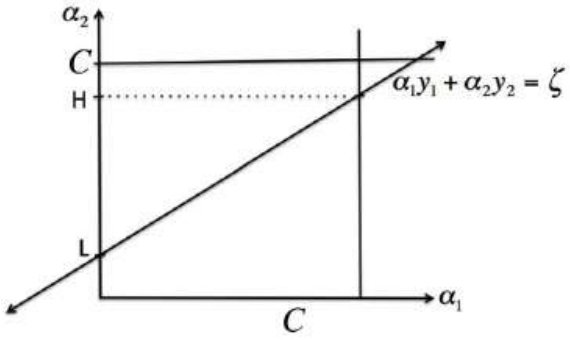


Figure 5.13

We are only allowed to choose α_1 and α_2 **on the line**.

- **When we pick α_2**, we can get α_1 from

$$\alpha_1 = \frac{1}{y^{(1)}}(\xi - \alpha_2 y^{(2)}) = y^{(1)}(\xi - \alpha_2 y^{(2)}). \tag{5.64}$$

- **Optimization for α_2**: The other constraints in (5.63) says $0 \le \alpha_1, \alpha_2 \le C$. Thus, α_2 needs to be within $[L, H]$ on the figure ($\because \alpha_1 \in [0, C]$). To do the coordinate ascent step, we will optimize the objective over α_2, keeping it within $[L, H]$. Using (5.64), (5.63) becomes

$$\max_{\alpha_2 \in [\mathbf{L},\mathbf{H}]} \left[y^{(1)}(\xi - \alpha_2 y^{(2)}) + \alpha_2 + \sum_{i=3}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y^{(i)} y^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right], \tag{5.65}$$

of which the objective is quadratic in α_2. This means we can just set its derivative to 0 to optimize it $\Longrightarrow$ **get α_2**.

- After updating α_1 using (5.64), move to the next iteration of SMO.

Note: There are heuristics to choose the order of α_i's chosen to update.

5.4. Decision Trees

Decision tree classifiers are attractive models if we care about **interpretability**. As the name decision tree suggests, we can think of this model as breaking down our data by making decision based on **asking a series of questions**. Decision tree was invented by a British researcher, William Belson, in 1959 [1].

Note: Decision trees are commonly used in **operations research**, specifically in **decision analysis**, to help identify a strategy most likely to reach a goal, but are also a popular tool in ML.

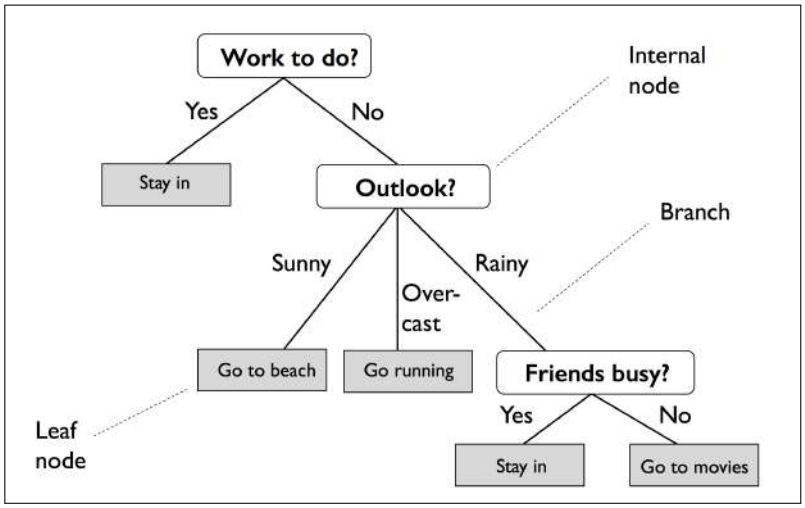


Figure 5.14: A decision tree to decide upon an activity on a particular day.

Key Idea 5.33. (Decision tree).

- **Start** at the tree root
- **Split** the data so as to result in the largest **Information Gain** (IG)
- **Repeat** the splitting at each child node until the leaves are pure
  (This means the samples at each node all belong to the same class)
- **In practice**, this can result in a very **deep tree** with many nodes, which can easily lead to **overfitting**
  (We typically set a limit for the maximal depth of the tree)

5.4.1. Decision tree objective

- Decision tree also needs to incorporate an **objective function**, to be optimized via the tree learning algorithm.
- Here, the objective function is to **maximize the information gain at each split**, which we define as follows:

$$IG(D_P, f) = I(D_P) - \sum_{j=1}^{m} \frac{N_j}{N_P} I(D_j), \tag{5.66}$$

where

f : the feature to perform the split
D_P : the parent dataset
D_j : the dataset of the j-th child node
I : the **impurity measure**
N_P : the total number of samples at the parent note
N_j : the number of samples in the j-th child node

- The **information gain** is simply the difference between the impurity of the parent node and the average of the child node impurities
  - The lower the impurity of the child nodes, the larger the information gain.
- However, for simplicity and to reduce the combinatorial search space, most libraries implement **binary decision trees**, where each parent node is split into two child nodes, D_L and D_R:

$$IG(D_P, f) = I(D_P) - \frac{N_L}{N_P} I(D_L) - \frac{N_R}{N_P} I(D_R). \tag{5.67}$$

Impurity measure?

Commonly used in binary decision trees:

- **Entropy**

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t) \tag{5.68}$$

- **Gini impurity**

$$I_G(t) = \sum_{i=1}^{c} p(i|t)\big(1 - p(i|t)\big) = 1 - \sum_{i=1}^{c} p(i|t)^2 \tag{5.69}$$

- **Classification error**

$$I_E(t) = 1 - \max_i\{p(i|t)\} \tag{5.70}$$

where $p(i|t)$ denotes the proportion of the samples that belong to class i for a particular node t.

Mind simulation: When $c = 2$

- **Entropy**: It is maximal, if we have a uniform class distribution; it is 0, if all samples at the node t belong to the same class.

$$I_H(t) = 0, \quad \text{if } p(i = 1|t) = 1 \text{ or } p(i = 2|t) = 0$$
$$I_H(t) = 1, \quad \text{if } p(i = 1|t) = p(i = 2|t) = 0.5$$

⇒ We can say that the entropy criterion attempts to maximize the mutual information in the tree.

- **Gini impurity**: Intuitively, it can be understood as a criterion to minimize the probability of misclassification. The Gini impurity is maximal, if the classes are perfectly mixed.

$$I_G(t) = 1 - \textstyle\sum_{i=1}^{c} 0.5^2 = 0.5$$

⇒ In practice, both Gini impurity and entropy yield very similar results.

- **Classification error**: It is less sensitive to changes in the class probabilities of the nodes.

⇒ The classification error is a useful criterion for pruning, but not recommended for growing a decision tree.

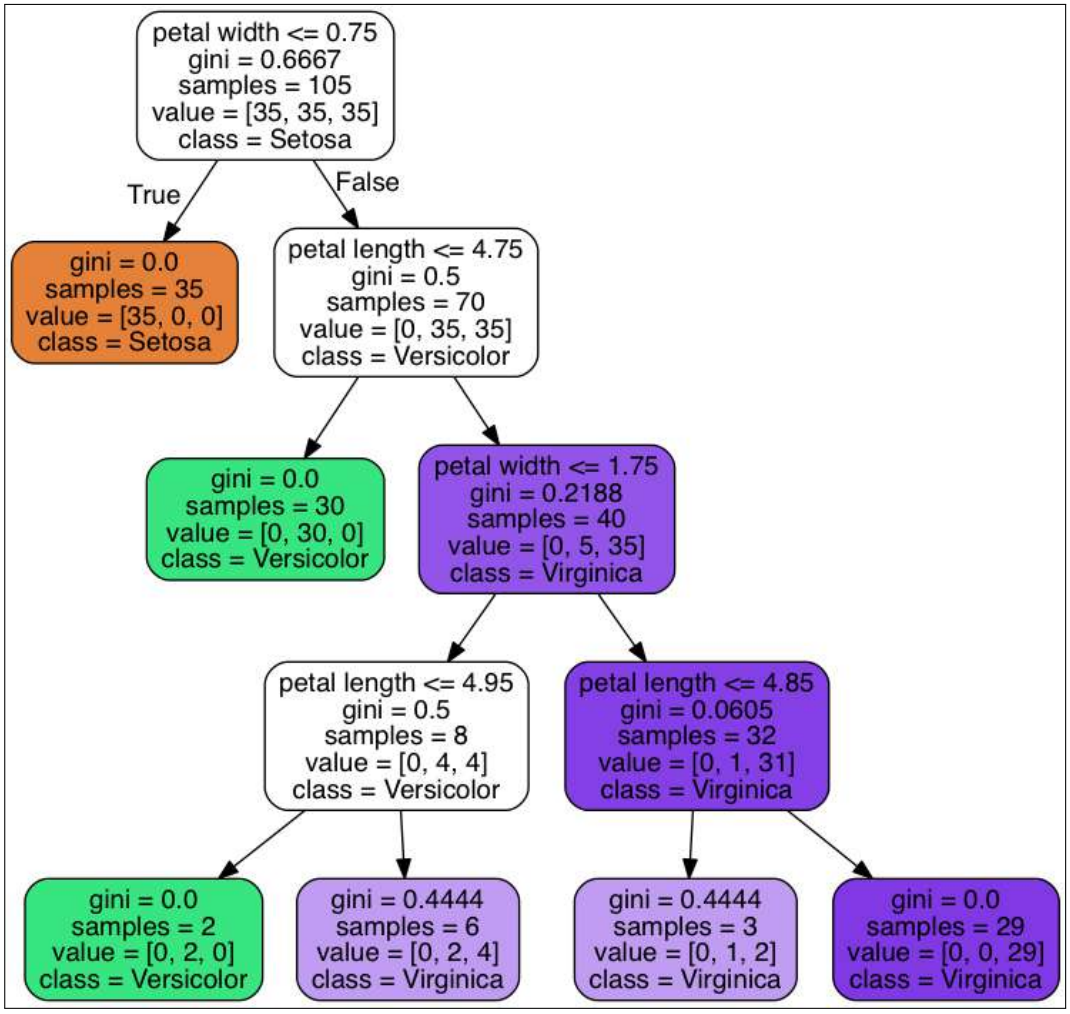


Figure 5.15: **A decision tree result** with Gini impurity measure, for three classes with two features (petal length, petal width). Page 99, *Python Machine Learning, 3rd Ed.*.

Quesiton. How can the decision tree find questions such as

`petal width <= 0.75` `petal length <= 4.75` ... ?

Algorithm 5.34. (Decision tree split rule).

1. **For each and every feature in D_P,** $f_j^{(i)}$:
   - make a question to split D_p into D_L and D_R (e.g. $f_j^{(k)} \leq f_j^{(i)}$, for which k's?)
   - compute the impurities: $I(D_L)$ and $I(D_R)$
   - compute the information gain:

$$IG(D_P, f_j^{(i)}) = I(D_P) - \frac{N_L}{N_P} I(D_L) - \frac{N_R}{N_P} I(D_R).$$

2. Let

$$f_q^{(p)} = \arg\max_{i,j} IG(D_P, f_j^{(i)}). \tag{5.71}$$

3. Then, the **best split** question (at the current node) is

$$f_q^{(k)} \leq f_q^{(p)}, \text{ for which } k\text{'s?} \tag{5.72}$$

The maximum in (5.71) often happens when one of the child impurities is zero or very small.

5.4.2. Random forests: Multiple decision trees

Random forests (or **random decision forests**) are an **ensemble learning** method for classification, regression, and other tasks that operates by constructing **multiple decision trees** at training time and outputting the class that is the **mode of the predicted classes** (classification) or **mean prediction** (regression) of the individual trees [32].

- Random forests have gained huge popularity in applications of ML **during the last decade** due to their good classification performance, scalability, and ease of use.
- The idea behind a random forest is **to average multiple (deep) decision trees** that individually suffer from high variance, to build a more robust model that has a better generalization performance and is less susceptible to overfitting.

Algorithm 5.35. Random Forest.
The algorithm can be summarized in four simple steps:

1. Draw a random **bootstrap sample** of size n
   (Randomly choose n samples from the training set *with replacement*).
2. Grow a decision tree from the bootstrap sample.
3. Repeat Steps 1-2 k times.
4. Aggregate the prediction by each tree to assign the class label by **majority vote**.

Note: In Step 2, when we are training the individual decision tree:

- instead of evaluating all features to determine the best split at each node,
- we can consider a random (without replacement) subset of those (of size d).

Remark **5.36.** A **big advantage** of random forests is that

we don't have to worry so much about choosing good hyperparameter values.

- A smaller n increases randomness of the random forest; the bigger n is, the larger the degree of overfitting becomes.

  Default n = size(the original training set), in most implementations
- Default $d = \sqrt{M}$, where M is the number of features in the training set
- The only parameter that we really need to care about in practice is

  the number of trees k (Step 3).

  Typically, the larger the number of trees, the better the performance of the random forest classifier at the expense of an increased computational cost.

5.5. k-Nearest Neighbors

The **k-nearest neighbor** (k-NN) classifier is a typical example of a **lazy learner**.

- It is called lazy not because of its apparent simplicity, but because it **doesn't learn a discriminative function** from the training data, but memorizes the training dataset instead.
- Analysis of the training data is **delayed until a query is made** to the system.

Algorithm 5.37. (k-NN algorithm). The algorithm itself is fairly straightforward and can be summarized by the following steps:

1. Choose the number k and a distance metric.
2. For the new sample, find the k-nearest neighbors.
3. Assign the class label by majority vote.

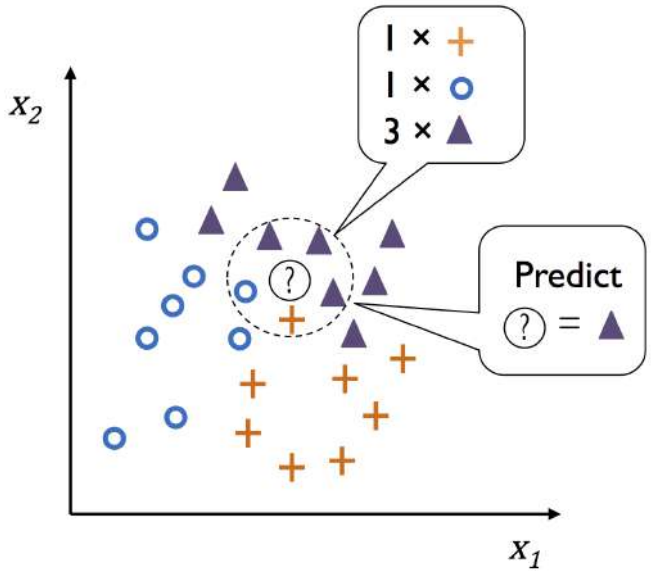


Figure 5.16: Illustration for how a new data point (?) is assigned the triangle class label, based on majority voting, when $k = 5$.

k-NN: pros and cons

- Since it is memory-based, the classifier **immediately adapts** as we collect new training data.
- The **computational complexity** for classifying new samples grows linearly with the number of samples in the training dataset in the worst-case scenario.[a]
- Furthermore, we can't discard training samples since *no training step* is involved. Thus, **storage space** can become a challenge if we are working with large datasets.

[a] J. H. Friedman, J. L. Bentley, and R.A. Finkel (1977). *An Algorithm for Finding Best Matches in Logarithmic Expected Time*, ACM transactions on Mathematical Software (TOMS), **3**, no. 3, pp. 209–226. The algorithm in the article is called the **KD-tree**.

k-NN: what to choose k and a distance metric?

- The **right choice of k is crucial** to find a good balance between overfitting and underfitting.
  (For `sklearn.neighbors.KNeighborsClassifier`, default `n_neighbors = 5`.)
- We also choose a distance metric that is appropriate for the features in the dataset. (e.g., the simple Euclidean distance, along with data standardization)
- Alternatively, we can choose the **Minkowski distance**:

$$d(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\|_p \overset{\text{def}}{=\!=} \left(\sum_{i=1}^{m} |x_i - z_i|^p \right)^{1/p}. \tag{5.73}$$

  (For `sklearn.neighbors.KNeighborsClassifier`, default `p = 2`.)

Remark 5.38. The k-NN algorithm is very susceptible (wide open) to **overfitting** due to the **curse of dimensionality**.[a]
Since regularization is not applicable for k-NN, we can use **feature selection** and **dimensionality reduction** techniques to help us avoid the curse of dimensionality and avoid overfitting. This will be discussed in more details later.

[a] The **curse of dimensionality** describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset. Intuitively, we can think of even the closest neighbors being too far away in a high-dimensional space to give a good estimate.

Exercises for Chapter 5

5.1. The equation $c_1x_1 + c_2x_2 + \cdots + c_nx_n = d$ determines a hyperplane in $\mathbb{R}^n$. Prove that the vector $[c_1, c_2, \cdots, c_n]$ is a normal vector of the hyperplane.

5.2. For this problem, you would modify the code used for Problem 3.2 in Chapter 3. For the standardized data (X_{SD}),

(a) Apply the logistic regression gradient descent (Algorithm 5.9).
(b) Compare the results with that of Adaline descent gradient.

5.3. *(Continuation of Problem 5.2).* Perturb the standardized data (X_{SD}) by a random Gaussian noise G_σ of an observable σ (so as for $G_\sigma(X_{SD})$ not to be linearly separable).

(a) Apply the logistic regression gradient descent (Algorithm 5.9) for the noisy data $G_\sigma(X_{SD})$.
(b) Modify the code for the logistic regression with regularization (5.21) and apply the resulting algorithm for $G_\sigma(X_{SD})$.
(c) Compare their performances

5.4. **(Optional for Undergraduate Students)** Verify the formulation in (5.51), which is dual to the minimization of (5.50).

5.5. Experiment examples on pp. 84–91, *Python Machine Learning, 3rd Ed.*, in order to optimize the performance of kernel SVM by finding a best kernel and optimal hyperparameters (`gamma` and C).

Choose one of Exercises 6 and 7 below to implement and experiment. The experiment will guide you to understand how the LM software has been composed from scratch. You may use the example codes thankfully shared by Dr. Jason Brownlee, who is the founder of machinelearningmastery.com.

5.6. Implement a **decision tree** algorithm that incorporates the Gini impurity measure, from scratch, to run for the data used on page 96, *Python Machine Learning, 3rd Ed.*. Compare your results with the figure on page 97 of the book. You may refer to
https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/

5.7. Implement a **k-NN** algorithm, from scratch, to run for the data used on page 106, *Python Machine Learning, 3rd Ed.*. Compare your results with the figure on page 103 of the book. You may refer to
https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/

CHAPTER 6

Data Preprocessing in Machine Learning

Data preprocessing (or, **data preparation**) is a data mining technique, which is **the most time consuming** (*often*, **the most important**) step in machine learning.

Contents of Chapter 6

6.1. General Remarks on Data Preprocessing 142
6.2. Dealing with Missing Data & Categorical Data 144
6.3. Feature Scaling 146
6.4. Feature Selection 148
6.5. Feature Importance 154
Exercises for Chapter 6 156

6.1. General Remarks on Data Preprocessing

Data preprocessing is a data mining technique.

- It involves transforming raw data into a **understandable and more tractable** format.
- Real-world data is often **incomplete, redundant, inconsistent**, and/or **lacking in certain behaviors or trends**, and is likely to contain many errors.
- Data preprocessing is a proven method of resolving such issues.
- Often, data preprocessing is the **most important phase** of a machine learning project, especially in computational biology.

Summary **6.1.** Different steps involved for data preprocessing can be summarized as follows.

1. **Data Cleaning**: In this first step, the primary focus is on handling missing data, noisy data, detection and removal of outliers, and minimizing duplication and computed biases within the data.
2. **Data Integration**: This process is used when data is gathered from various data sources and data are combined to form consistent data.
3. **Data Transformation**: This step is used to convert the raw data into a specified format according to the need of the model.
   (a) *Normalization* – Numerical data is converted into the specified range (e.g., **feature scaling** $\rightarrow \sim \mathcal{N}(0,1)$).
   (b) *Aggregation* – This method combines some features into one.
4. **Data Reduction**: Redundancy within the data can be removed and efficiently organize the data.

The more disciplined you are in your handling of data, the more consistent and better results you are likely to achieve.

Remark 6.2. **Data preparation** is **difficult** because the process is ***not objective***, and it is **important** because ML algorithms ***learn from data***. Consider the following.

- Preparing data for analysis is one of the most **important** steps in any data-mining project – and traditionally, one of the most **time consuming**.
- Often, it takes up to 80% of the time.
- Data preparation is **not a once-off process**; that is, it is iterative as you understand the problem deeper on each successive pass.
- It is critical that you **feed the algorithms with the right data** for the problem you want to solve. Even if you have a good dataset, you need to make sure that it is in a useful scale and format and that meaningful features are included.

Questions in ML, in practice

- What would reduce the **generalization error**?
- What is the **best form of the data** to describe the problem? (It is difficult to answer, because it is not objective.)
- Can we design effective methods and/or smart algorithms for **automated data preparation**?

6.2. Dealing with Missing Data & Categorical Data

6.2.1. Handling missing data

Software: `[pandas.DataFrame].isnull().sum() > 1`

For missing values, three different steps can be executed.

- **Removal of samples (rows) or features (columns)**:
  - It is **the simplest and efficient method** for handling the missing data.
  - However, we may end up removing too many samples or features.

  Software: `pandas.dropna`
- **Filling the missing values manually**:
  - This is **one of the best-chosen methods.**
  - But there is one limitation that when there are large data set, and missing values are significant.
- **Imputing missing values using computed values**:
  - The missing values can also be occupied by computing **mean, median, or mode** of the observed given values.
  - Another method could be the predictive values that are computed by using any ML or Deep Learning algorithms.
  - But one drawback of this approach is that **it can generate bias** within the data as the calculated values are not accurate concerning the observed values.

  Software: `from sklearn.preprocessing import Imputer`

6.2.2. Handling categorical data

It is common that real-world datasets contain one or more categorical feature columns. These categorical features must be effectively handled to fit in ***numerical computing libraries***.

When we are talking about categorical data, we should further distinguish between **ordinal features** and **nominal features**.

- Mapping ordinal features: e.g.,

$$\text{size}: \begin{bmatrix} \text{M} \\ \text{L} \\ \text{XL} \end{bmatrix} \longleftrightarrow \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}. \tag{6.1}$$

  - This is called an **ordinal encoding** or an **integer encoding**.
  - The integer values have a natural ordered relationship between each other; machine learning algorithms may understand and harness this relationship.

- Encoding nominal features: **one-hot encoding**, e.g.,

$$\text{color}: \begin{bmatrix} \text{blue} \\ \text{green} \\ \text{red} \end{bmatrix} \longleftrightarrow \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \longleftrightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{6.2}$$

Software: `from sklearn.preprocessing import OneHotEncoder`

Remark 6.3. For categorical variables where no ordinal relationship exists, the integer encoding is not enough.

- In fact, assuming a natural ordering between categories and using the integer encoding may result in poor performance or unexpected results.
- The **one-hot encoding** can be used, although ordinal relationship exists.

6.3. Feature Scaling

Note: **Feature scaling** is a method used **to standardize the range of independent variables or features** of the data.

- It is one of **data normalization** methods[a] in a broad sense.
- It is generally performed during the data preprocessing step.
- There are some scale-invariant algorithms such as decision trees and random forests.
- Most of other algorithms (we have learned) perform better with feature scaling.

[a] In a broad sense, **data normalization** is a process of reorganizing data, by cleaning and adjusting data values measured on different scales to a **notionally common scale**; its intention is to bring the entire probability distributions of adjusted values into alignment.

There are two common approaches to bring different features onto the same scale:

- **min-max scaling (normalization)**:

$$x_{j,\text{norm}}^{(i)} = \frac{x_j^{(i)} - x_{j,\min}}{x_{j,\max} - x_{j,\min}} \in [0, 1], \tag{6.3}$$

  where $x_{j,\min}$ and $x_{j,\max}$ are the minimum and maximum of the j-th feature column (in the training dataset), respectively.

- **standardization**:

$$x_{j,\text{std}}^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j}, \tag{6.4}$$

  where μ_j is the sample mean of the j-th feature column and σ_j is the corresponding standard deviation.

  - The standardized data has the **standard normal distribution**.

Remark **6.4.** **Standardization** is more practical than the **min-max scaling** for many ML methods, especially for **optimization algorithms** such as the gradient descent method.

- **Reason** ①: For many linear models such as the logistic regression and the SVM, we can **easily initialize the weights** to 0 or small random values close to 0.

  $\Leftarrow$ Standardization possibly results in $\mathbf{w}^*$ *small*.

- **Reason** ②: It makes **regularization** perform more effectively; see Sections 5.2.3 and 6.4.3 for regularization.

  $\Leftarrow$ The minimizer of the penalty term is $\mathbf{0}$.

6.4. Feature Selection

6.4.1. Selecting meaningful variables

Remark 6.5. **Overfitting**. If we observe that a model performs **much better on a training dataset** than on the test dataset, it is a strong indicator of **overfitting**.

- Overfitting means **the model fits the parameters too closely** with regard to the particular observations in **the training dataset**, but does not generalize well to new data. (The model has a high variance.)
- The reason for the overfitting is that our model is **too complex for the given training data**.

Common solutions to reduce the **generalization error** (via bias-variance tradeoff) are listed as follows:

- **Collect more training data** (often, not applicable)
- **Introduce regularization** (penalty for complexity)
- **Choose a simpler model** (fewer parameters)
- **Reduce the dimensionality** (feature selection)

Feature Selection (a.k.a. Variable Selection)
Its objective is four-fold:

- enhancing generalization by reducing overfitting/variance,
- providing faster and more cost-effective predictors,
- reducing training time, and
- providing a better understanding of the underlying process that generated the data.

Recall: **Curse of Dimensionality**. It describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset.

Methods for "automatic" feature selection

- **Filter methods**: **Filter methods suppress the least interesting features,** after assigning a scoring to **each feature** and ranking the features. The methods consider the feature *independently*, or with regard to the dependent variable.
  Examples: *Chi-squared test & correlation coefficient scores.*
- **Wrapper methods**: Wrapper methods evaluate **subsets of features** which allows, unlike filter approaches, **to detect the possible interactions between features**. They prepare **various combinations of features**, to evaluate and compare with other combinations. The two main disadvantages of these methods are:
  - Increasing overfitting risk, when the data size is not enough.
  - Significant computation time, for a large number of variables.

  Example: The recursive feature elimination algorithm
- **Embedded methods**: Embedded methods have been recently proposed that try **to combine the advantages of both previous methods**. They learn which features contribute the best to the accuracy of the model while the model is being created. The most common types of embedded feature selection methods are **regularization methods**.
  Examples: *ridge regression*[a], *LASSO*[b], *& elastic net regularization*[c]

[a] The **ridge regression** (a.k.a. **Tikhonov regularization**) is the most commonly used method of regularization of ill-posed problems. In machine learning, ridge regression is basically a regularized linear regression model: $\min_{\mathbf{w}} \mathcal{Q}(X, \mathbf{y}; \mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$, in which the regularization parameter λ should be learned as well, using a method called cross validation. It is related to the Levenberg-Marquardt algorithm for non-linear least-squares problems.

[b] **LASSO** (least absolute shrinkage and selection operator) is a regression analysis method that performs both variable selection and regularization in order to enhance the prediction accuracy and interpretability of the statistical model it produces. It includes an L^1 penalty term: $\min_{\mathbf{w}} \mathcal{Q}(X, \mathbf{y}; \mathbf{w}) + \lambda\|\mathbf{w}\|_1$. It was originally developed in Geophysics [68, (Santosa-Symes, 1986)], and later independently rediscovered and popularized in 1996 by Robert Tibshirani [75], who coined the term and provided further insights into the observed performance.

[c] The **elastic net regularization** is a regularized regression method that linearly combines the L^1 and L^2 penalties of the LASSO and ridge methods, particularly in the fitting of linear or logistic regression models.

We will see how L^1-regularization can reduce overfitting (serving as a feature selection method).

6.4.2. Sequential backward selection (SBS)

The idea behind the **sequential backward selection** (SBS) algorithm is quite simple:

- **The SBS sequentially removes features one-by-one** until the new feature subspace contains the desired number of features.
- In order to determine which feature is to be removed at each stage, we need to define the **criterion function** $\mathcal{C}$, e.g., performance of the classifier after the removal of a particular feature.
- Then, the feature to be removed at each stage can simply be defined as **the feature that maximizes this criterion**; or in more intuitive terms, at each stage we eliminate the feature that causes the **least performance loss after removal**.

Algorithm 6.6. Sequential Backward Selection

We can outline the algorithm in four simple steps:

1. Initialize the algorithm with $k = d$, where d is the dimensionality of the full feature space F_d.
2. Determine the feature $\widehat{f}$ such that
$$\widehat{f} = \arg\max_{f \in F_k} \mathcal{C}(F_k - f).$$
3. Remove the feature $\widehat{f}$ from the feature set:
$$F_{k-1} = F_k - \widehat{f}; \;\; k = k - 1;$$
4. Terminate if k equals the number of desired features; otherwise, go to step 2.

6.4.3. Ridge regression vs. LASSO

Observation 6.7. When an L^p-penalty term is involved ($p = 1, 2$), the minimization problem can be written as follows:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \mathcal{Q}(X, \mathbf{y}; \mathbf{w}) + \lambda \mathcal{R}_p(\mathbf{w}), \tag{6.5}$$

where

$$\mathcal{R}_p(\mathbf{w}) := \frac{1}{p}\|\mathbf{w}\|_p^p, \quad p = 1, 2. \tag{6.6}$$

- **Regularization** can be considered as adding a penalty term to the cost function to **encourage smaller weights**; or in other words, we penalize large weights.
- Thus, by **increasing the regularization strength** ($\lambda \uparrow$),
  - we can **shrink the weights** towards zero, and
  - **decrease the dependence** of our model on the training data.
- The **minimizer** $\mathbf{w}^*$ must be the point where the L^p-ball **intersects** with the minimum-valued contour of the unpenalized cost function.
  - The variable λ in (6.5) is a kind of **Lagrange multiplier**.

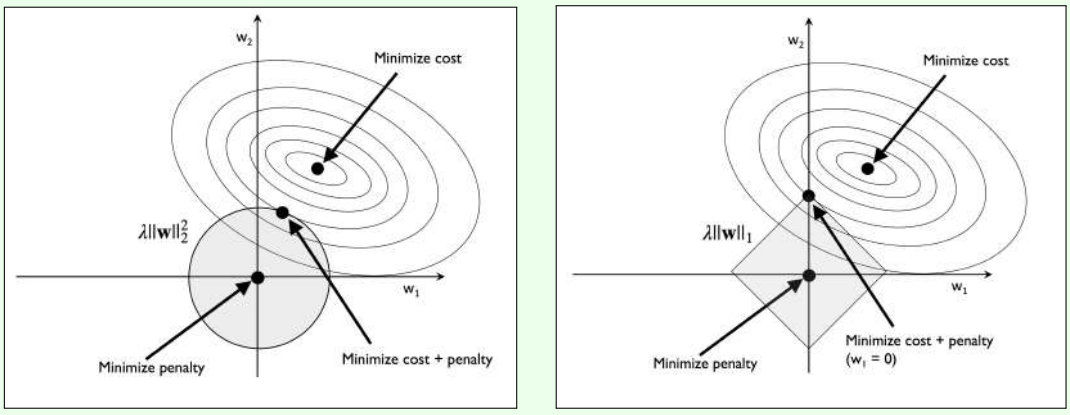


Figure 6.1: L^2-regularization ($\|\mathbf{w}\|_2^2 = \sum_{i=1}^m w_i^2$) and L^1-regularization ($\|\mathbf{w}\|_1 = \sum_{i=1}^m |w_i|$).

LASSO (L^1-regularization). In the right figure, the L^1-ball touches the minimum-valued contour of the cost function at $w_1 = 0$; the **optimum** is *more likely* located **on the axes**, which **encourages sparsity** (zero entries in w^*).

Remark 6.8. LASSO (L^1-regularization)

- We can **enforce sparsity** (more zero entries) by increasing the regularization strength λ.
- A **sparse model** is a model where many of the weights are 0 or close to 0. Therefore L^1**-regularization** is more suitable to create desired 0-weights, particularly for sparse models.

Remark 6.9. Regularization

In general, **regularization** can be understood as **adding bias** and preferring a **simpler model to reduce the variance (overfitting)**, ***in the absence of sufficient training data, in particular.***

- L^1-regularization **encourages sparsity**.
- We can **enforce sparsity** (more zero entries) by increasing the regularization strength λ.
- Thus it can **reduce overfitting**, serving as a **feature selection** method.

- L^1-regularization may introduce **oscillation**, particularly when the regularization strength λ is large.
- A **post-processing operation** may be needed to take into account oscillatory behavior of L^1-regularization.

Example 6.10. Consider a model consisting of the weights $\mathbf{w} = (w_1, \cdots, w_m)^T$ and

$$\mathcal{R}_1(\mathbf{w}) = \sum_{i=1}^{m} |w_i|, \quad \mathcal{R}_2(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m} w_i^2. \tag{6.7}$$

Let us minimize $\mathcal{R}_p(\mathbf{w})$, $p = 1, 2$, using gradient descent.

Solution. The gradients read

$$\nabla_{\mathbf{w}}\mathcal{R}_1(\mathbf{w}) = \text{sign}(\mathbf{w}), \quad \nabla_{\mathbf{w}}\mathcal{R}_2(\mathbf{w}) = \mathbf{w}, \tag{6.8}$$

where

$$\text{sign}(w_i) = \begin{cases} 1, & \text{if } w_i > 0 \\ -1, & \text{if } w_i < 0 \\ 0, & \text{if } w_i = 0. \end{cases}$$

Thus the gradient descent becomes

$$\begin{aligned} \mathcal{R}_1 \ &: \ \mathbf{w}_{k+1} = \mathbf{w}_k - \lambda\, \text{sign}(\mathbf{w}_k), \\ \mathcal{R}_2 \ &: \ \mathbf{w}_{k+1} = \mathbf{w}_k - \lambda\, \mathbf{w}_k = (1-\lambda)\, \mathbf{w}_k = (1-\lambda)^{k+1}\, \mathbf{w}_0. \ \square \end{aligned} \tag{6.9}$$

- **The L^2-gradient** is linearly decreasing towards 0 as the weight goes towards 0. Thus **L^2-regularization** will move any weight towards 0, but it will take smaller and smaller steps as a weight approaches 0. (The model never reaches a weight of 0.)
- In contrast, **L^1-regularization** will move any weight towards 0 with the same step size λ, regardless the weight's value.
  - The iterates for minimizing $\mathcal{R}_1$ **may oscillate endlessly** near 0. (e.g., $w_0 = 0.2$ and $\lambda = 0.5$
    $\Rightarrow w_1 = -0.3 \Rightarrow w_2 = 0.2 \Rightarrow w_3 = -0.3 \Rightarrow w_4 = 0.2 \Rightarrow \cdots$)
  - The oscillatory phenomenon may not be severe for real-world problems where $\mathcal{R}_1$ is used as a penalty term for a cost function.
  - However, we may need a **post-processing** to take account of oscillation, when λ is set large.

6.5. Feature Importance

The concept of **feature importance** is straightforward: it is *the increase in the model's prediction error after we permuted the feature's values*, which breaks the relationship between the feature and the true outcome.

- **A feature is "important" if shuffling its values increases the model error,** because in this case the model relied on the feature for the prediction.
- A feature is "unimportant" if shuffling its values leaves the model error unchanged, because in this case the model ignored the feature for the prediction.

- The **permutation feature importance** measurement was introduced by Breiman (2001) [8] for random forests.
- Based on this idea, Fisher, Rudin, and Dominici (2018) [19] proposed a **model-agnostic version** of the feature importance and called it **model reliance**.

Algorithm 6.11. **Permutation feature importance (FI)**

input: Trained model f, feature matrix X, target vector $\mathbf{y}$, error measure $\mathcal{L}(f, X, \mathbf{y})$;

1. Estimate the original model error $\varepsilon^{\text{orig}} = \mathcal{L}(f, X, \mathbf{y})$;
2. For each feature $j = 1, 2, \cdots, d$; do:
   - Permute feature j in the data X to get $X^{(j)}$;
   - Estimate error $\varepsilon^{(j)} = \mathcal{L}(f, X^{(j)}, \mathbf{y})$;
   - Calculate permutation FI: $FI^{(j)} = \varepsilon^{(j)}/\varepsilon^{\text{orig}}$ (or, $\varepsilon^{(j)} - \varepsilon^{\text{orig}}$);
3. Sort features by descending FI;

Should we compute FI on training or test data?

To answer the question, you need to decide whether

- you want to know **how much the model relies on each feature** for making predictions ($\rightarrow$ training data) or
- **how much the feature contributes** to the performance of the model on **unseen data** ($\rightarrow$ test data).

There is no research addressing the question of training vs. test data; more research and more experience are needed to gain a better understanding.

Exercises for Chapter 6

First, read pp. 135-143, *Python Machine Learning, 3rd Ed.*.

6.1. On pp. 135-143, the **sequential backward selection** (SBS) is implemented as a feature selection method and experimented with a k**-NN classifier** (`n_neighbors=5`), using the wine dataset.

(a) Perform the same experiment with the k-NN classifier replaced by the **support vector machine** (soft-margin SVM classification).
(b) In particular, analyze **accuracy of the soft-margin SVM** and plot the result as in the figure on p. 139.

6.2. On pp. 141-143, the **permutation feature importance** is assessed from the **random forest** classifier, using the wine dataset.

(a) Discuss whether or not you can derive feature importance for a k**-NN classifier**.
(b) Assess feature importance with the **logistic regression** classifier, using the same dataset.
(c) Based on the computed feature importance, analyze and plot **accuracy of the logistic regression** classifier for `k_features` $= 1, 2, \cdots, 13$.

CHAPTER 7

Feature Extraction: Data Compression

There are *two main categories* of **dimensionality reduction** methods:

- **Feature selection**: Select a subset of the original features.
- **Feature extraction**: Construct a new feature subspace.

Feature Extraction

- It can be understood as an approach to **dimensionality reduction** and **data compression**.
  - with the goal of maintaining most of the relevant information
- In practice, **feature extraction** is used
  - to improve storage space or the computational efficiency
  - **to improve the predictive performance** by reducing the curse of dimensionality

In this chapter, we will study **three fundamental techniques** for dimensionality reduction:

- **Principal component analysis** (PCA)
- **Linear discriminant analysis** (LDA), *maximizing class separability*
- **Kernel principal component analysis**, for nonlinear PCA

Contents of Chapter 7

7.1. Principal Component Analysis 158
7.2. Singular Value Decomposition 164
7.3. Linear Discriminant Analysis 180
7.4. Kernel Principal Component Analysis 197
Exercises for Chapter 7 204

7.1. Principal Component Analysis

- **Principal component analysis** (PCA) (a.k.a. **orthogonal linear transformation**) was invented in 1901 by K. Pearson [58], as an analogue of the principal axis theorem in mechanics; it was later independently developed and named by H. Hotelling in the 1930s [33, 34].
- The PCA is a statistical procedure that **uses an orthogonal transformation** to convert a set of observations (*of possibly correlated variables*) to **a set of linearly uncorrelated variables** called the **principal components**.
- The **orthogonal axes** of the new subspace can be interpreted as the **directions of maximum variance** given the constraint that the new feature axes are orthogonal to each other:

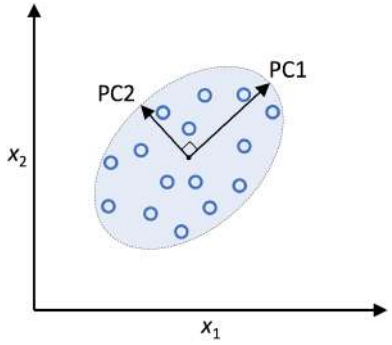


Figure 7.1: Principal components.

- As an **unsupervised**[a] linear transformation technique, the PCA is widely used across **various fields** – in ML, most prominently for feature extraction and dimensionality reduction.
- The PCA identifies **patterns in data** based on the **correlation between features**.
- The PCA directions are **highly sensitive to data scaling**, and we **need to standardize the features** prior to PCA.

[a] The PCA is a unsupervised technique, because it does not use any class label information.

7.1.1. Computation of principal components

- Consider a **data matrix** $X \in \mathbb{R}^{N \times d}$:
  - each of the N rows represents a different data point,
  - each of the d columns gives a particular kind of feature, and
  - each column has zero empirical mean (e.g., after standardization).
- **The goal** is **to find an orthogonal weight matrix** $W \in \mathbb{R}^{d \times d}$ such that
$$Z = XW \tag{7.1}$$
maximizes the variance ($\Rightarrow$ minimizes the reconstruction error).
- Here $Z \in \mathbb{R}^{N \times d}$ is called the **score matrix**, of which columns represent **principal components** of X.

First weight vector $\mathbf{w}_1$: the first column of W:
In order to maximize variance of $\mathbf{z}_1$, the first weight vector $\mathbf{w}_1$ should satisfy
$$\begin{aligned} \mathbf{w}_1 &= \arg\max_{\|\mathbf{w}\|=1} \|\mathbf{z}_1\|^2 = \arg\max_{\|\mathbf{w}\|=1} \|X\mathbf{w}\|^2 \\ &= \arg\max_{\|\mathbf{w}\|=1} \mathbf{w}^T X^T X \mathbf{w} = \arg\max_{\mathbf{w}\neq 0} \frac{\mathbf{w}^T X^T X \mathbf{w}}{\mathbf{w}^T \mathbf{w}}, \end{aligned} \tag{7.2}$$
where the quantity to be maximized can be recognized as a **Rayleigh quotient**.

Theorem 7.1. *For a **positive semidefinite matrix** (such as $X^T X$), the maximum of the Rayleigh quotient is the same as the largest eigenvalue of the matrix, which occurs when $\mathbf{w}$ is the corresponding eigenvector, i.e.,*
$$\mathbf{w}_1 = \arg\max_{\mathbf{w}\neq 0} \frac{\mathbf{w}^T X^T X \mathbf{w}}{\mathbf{w}^T \mathbf{w}} = \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|}, \quad (X^T X)\mathbf{v}_1 = \lambda_1 \mathbf{v}_1, \tag{7.3}$$
where λ_1 is the largest eigenvalue of $X^T X \in \mathbb{R}^{d \times d}$.

Example 7.2. With $\mathbf{w}_1$ found, the **first principal component** of a data vector $\mathbf{x}^{(i)}$ can then be given as a score $z_1^{(i)} = \mathbf{x}^{(i)} \cdot \mathbf{w}_1$.

Further weight vectors $\mathbf{w}_k$:

The k-th weight vector can be found by ① subtracting the first $(k-1)$ principal components from X:

$$\widehat{X}_k := X - \sum_{i=1}^{k-1} X \mathbf{w}_i \mathbf{w}_i^T, \tag{7.4}$$

and then ② finding the weight vector which **extracts the maximum variance** from this new data matrix $\widehat{X}_k$:

$$\mathbf{w}_k \;=\; \arg\max_{\|\mathbf{w}\|=1} \|\widehat{X}_k \mathbf{w}\|^2. \tag{7.5}$$

Claim 7.3. The above turns out to give the (normalized) eigenvectors of $X^T X$. That is, the **transformation matrix** W is the stack of eigenvectors of $X^T X$:

$$W = [\mathbf{w}_1 | \mathbf{w}_2 | \cdots | \mathbf{w}_d], \quad (X^T X)\, \mathbf{w}_j = \lambda_j\, \mathbf{w}_j, \;\; \mathbf{w}_i^T \mathbf{w}_j = \delta_{ij}, \tag{7.6}$$

where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$.

With W found, a **data vector** $\mathbf{x}$ is transformed to a d-dimensional row vector of principal components

$$\mathbf{z} = \mathbf{x} W, \tag{7.7}$$

of which components z_j, $j = 1, 2, \cdots, d$, are decorrelated.

Remark 7.4. From **Singular Value Decomposition**:

While the weight matrix $\boldsymbol{W} \in \mathbb{R}^{\boldsymbol{d \times d}}$ is the collection of eigenvectors of $\boldsymbol{X^T X}$, the score matrix $\boldsymbol{Z} \in \mathbb{R}^{\boldsymbol{N \times d}}$ is the stack of eigenvectors of $\boldsymbol{X X^T}$, scaled by the square-root of eigenvalues:

$$Z = [\sqrt{\lambda_1}\, \mathbf{u}_1 | \sqrt{\lambda_2}\, \mathbf{u}_2 | \cdots | \sqrt{\lambda_d}\, \mathbf{u}_d], \quad (X X^T)\, \mathbf{u}_j = \lambda_j\, \mathbf{u}_j, \;\; \mathbf{u}_i^T \mathbf{u}_j = \delta_{ij}. \tag{7.8}$$

See (7.14) and § 7.2.

7.1.2. Dimensionality reduction

The transformation $Z = XW$ maps data points in $\mathbb{R}^d$ to a new d-dimensional space of principal components. **Keeping only the first k principal components** ($k < d$) gives a **truncated transformation**:

$$Z_k = X\,W_k : \mathbf{x}^{(i)} \in \mathbb{R}^d \mapsto \mathbf{z}^{(i)} \in \mathbb{R}^k, \tag{7.9}$$

where $Z_k \in \mathbb{R}^{N\times k}$ and $W_k \in \mathbb{R}^{d\times k}$. Define the **truncated data** as

$$X_k \;:=\; Z_k W_k^T \;=\; X\,W_k W_k^T. \tag{7.10}$$

Quesitons. How can we choose k?
Is the difference $\|X - X_k\|$ small?

Remark 7.5. The principal components transformation can also be associated with the **singular value decomposition** (SVD) of X:

$$X = U\,\Sigma V^T, \tag{7.11}$$

where

U : $n \times d$ orthogonal (the **left singular vectors** of X.)
Σ : $d \times d$ diagonal (the **singular values** of X.)
V : $d \times d$ orthogonal (the **right singular vectors** of X.)

- The matrix Σ explicitly reads

$$\Sigma \;=\; \text{diag}(\sigma_1, \sigma_2, \cdots, \sigma_d), \tag{7.12}$$

  where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_d \geq 0$.
- In terms of this factorization, the matrix X^TX reads

$$X^TX = (U\,\Sigma V^T)^T U\,\Sigma V^T = V\,\Sigma U^T U\,\Sigma V^T = V\Sigma^2 V^T. \tag{7.13}$$

- Comparing with the **eigenvector factorization** of X^TX, we have
  - the right singular vectors $V \cong$ the eigenvectors of $X^TX \Rightarrow V \cong W$
  - the square of singular values of X are equal to the eigenvalues of X^TX $\Rightarrow \sigma_j^2 = \lambda_j$, $j = 1, 2, \cdots, d$.

Further considerations for the SVD

- Using the SVD, the **score matrix** Z reads

$$Z = XW = U\,\Sigma V^T W = U\,\Sigma, \tag{7.14}$$

  and therefore each column of Z is given by one of the left singular vectors of X multiplied by the corresponding singular value. This form is also the **polar decomposition** of Z. See (7.8) on p. 160.

- As with the eigen-decomposition, the SVD, the **truncated score matrix** $Z_k \in \mathbb{R}^{N\times k}$ can be obtained by considering only the first k largest singular values and their singular vectors:

$$Z_k = XW_k = U\,\Sigma V^T W_k = U\,\Sigma_k, \tag{7.15}$$

  where

$$\Sigma_k := \text{diag}(\sigma_1, \cdots, \sigma_k, 0, \cdots, 0). \tag{7.16}$$

- Now, using (7.15), the truncated data matrix reads

$$X_k = Z_k W_k^T = U\,\Sigma_k W_k^T = U\,\Sigma_k W^T = U\,\Sigma_k V^T. \tag{7.17}$$

Claim 7.6. It follows from (7.11) and (7.17) that

$$\begin{aligned} \|X - X_k\|_2 &= \|U\,\Sigma V^T - U\,\Sigma_k V^T\|_2 \\ &= \|U(\Sigma - \Sigma_k)V^T\|_2 \\ &= \|\Sigma - \Sigma_k\|_2 = \sigma_{k+1}, \end{aligned} \tag{7.18}$$

where $\|\cdot\|_2$ is the induced matrix L^2-norm.

Remark 7.7. Efficient algorithms exist to calculate the SVD of X without having to form the matrix $X^T X$. **Computing the SVD is now the standard way to carry out the PCA.** See [27, 79].

7.1.3. Explained variance

Note: Since we want to reduce the dimensionality of our dataset by compressing it onto a new feature subspace, we only select the subset of the eigenvectors (principal components) that contains **most of the information (variance). The eigenvalues define the magnitude of the eigenvectors**, so we have to sort the eigenvalues by decreasing magnitude; we are interested in the top k eigenvectors based on the values of their corresponding eigenvalues.

Definition 7.8. Let λ_j $(=\sigma_j^2)$ be eigenvalues of X^TX: $(X^TX)\mathbf{v}_j = \lambda_j\mathbf{v}_j$. Define the **explained variance ratio** of each eigenvalue as

$$evr(\lambda_i) = \frac{\lambda_i}{\sum_{j=1}^{d}\lambda_j}, \qquad i = 1, 2, \cdots, d, \tag{7.19}$$

and **cumulative explained variance** as

$$cev(\lambda_k) = \sum_{i=1}^{k} evr(\lambda_i) = \sum_{i=1}^{k}\lambda_i \Big/ \sum_{j=1}^{d}\lambda_j, \quad k = 1, 2, \cdots, d. \tag{7.20}$$

Then, we may choose k satisfying

$$cev(\lambda_{k-1}) < \varepsilon \ \text{ and } \ cev(\lambda_k) \geq \varepsilon, \tag{7.21}$$

for a tolerance ε. (**The smallest k such that $cev(\lambda_k) \geq \varepsilon$.**)

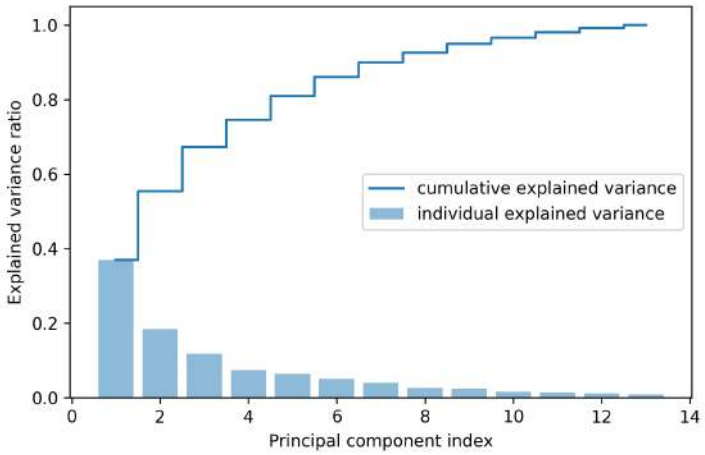


Figure 7.2: evr and cev for the wine dataset.

7.2. Singular Value Decomposition

Here we will deal with the SVD in detail.

Theorem 7.9. (SVD Theorem). *Let $A \in \mathbb{R}^{m\times n}$ with $m \geq n$. Then we can write*

$$A = U\,\Sigma\,V^T, \tag{7.22}$$

where $U \in \mathbb{R}^{m\times n}$ and satisfies $U^TU = I$, $V \in \mathbb{R}^{n\times n}$ and satisfies $V^TV = I$, and $\Sigma = diag(\sigma_1, \sigma_2, \cdots, \sigma_n)$, where

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0.$$

Remark 7.10. The matrices are illustrated pictorially as

$$\begin{bmatrix} \\ A \\ \\ \end{bmatrix} = \begin{bmatrix} \\ U \\ \\ \end{bmatrix} \begin{bmatrix} \\ \Sigma \\ \\ \end{bmatrix} \begin{bmatrix} \\ V^T \\ \\ \end{bmatrix}, \tag{7.23}$$

where

$$\begin{array}{rcl} U & : & m \times n \text{ orthogonal (the } \textbf{left singular vectors} \text{ of } A.) \\ \Sigma & : & n \times n \text{ diagonal (the } \textbf{singular values} \text{ of } A.) \\ V & : & n \times n \text{ orthogonal (the } \textbf{right singular vectors} \text{ of } A.) \end{array}$$

- For some $r \leq n$, the singular values may satisfy

$$\underbrace{\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r}_{\text{nonzero singular values}} > \sigma_{r+1} = \cdots = \sigma_n = 0. \tag{7.24}$$

  In this case, $\text{rank}(A) = r$.
- If $m < n$, the **SVD** is defined by considering A^T.

Proof. **(of Theorem 7.9)** Use induction on m and n: we assume that the SVD exists for $(m-1)\times(n-1)$ matrices, and prove it for $m\times n$. We assume $A\neq 0$; otherwise we can take $\Sigma = 0$ and let U and V be arbitrary orthogonal matrices.

- The basic step occurs when $n = 1$ ($m \geq n$). We let $A = U\Sigma V^T$ with $U = A/||A||_2$, $\Sigma = ||A||_2$, $V = 1$.
- For the induction step, choose $\mathbf{v}$ so that

$$||\mathbf{v}||_2 = 1 \text{ and } ||A||_2 = ||A\mathbf{v}||_2 > 0.$$

- Let $\mathbf{u} = \frac{A\mathbf{v}}{||A\mathbf{v}||_2}$, which is a unit vector. Choose $\tilde{U}$, $\tilde{V}$ such that

$$U = [\mathbf{u}\ \ \tilde{U}] \in \mathbb{R}^{m\times n} \text{ and } V = [\mathbf{v}\ \ \tilde{V}] \in \mathbb{R}^{n\times n}$$

  are orthogonal.
- Now, we write

$$U^TAV = \begin{bmatrix} \mathbf{u}^T \\ \tilde{U}^T \end{bmatrix} \cdot A \cdot [\mathbf{v}\ \ \tilde{V}] = \begin{bmatrix} \mathbf{u}^TA\mathbf{v} & \mathbf{u}^TA\tilde{V} \\ \tilde{U}^TA\mathbf{v} & \tilde{U}^TA\tilde{V} \end{bmatrix}$$

  Since

$$\begin{aligned} \mathbf{u}^TA\mathbf{v} &= \frac{(A\mathbf{v})^T(A\mathbf{v})}{||A\mathbf{v}||_2} = \frac{||A\mathbf{v}||_2^2}{||A\mathbf{v}||_2} = ||A\mathbf{v}||_2 = ||A||_2 \equiv \sigma, \\ \tilde{U}^TA\mathbf{v} &= \tilde{U}^T\mathbf{u}||A\mathbf{v}||_2 = 0, \end{aligned}$$

  we have

$$U^TAV = \begin{bmatrix} \sigma & 0 \\ 0 & U_1\Sigma_1V_1^T \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & U_1 \end{bmatrix} \begin{bmatrix} \sigma & 0 \\ 0 & \Sigma_1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & V_1 \end{bmatrix}^T,$$

  or equivalently

$$A = \left(U \begin{bmatrix} 1 & 0 \\ 0 & U_1 \end{bmatrix}\right) \begin{bmatrix} \sigma & 0 \\ 0 & \Sigma_1 \end{bmatrix} \left(V \begin{bmatrix} 1 & 0 \\ 0 & V_1 \end{bmatrix}\right)^T. \tag{7.25}$$

Equation (7.25) is our desired decomposition. □

7.2.1. Interpretation of the SVD

Algebraic interpretation of the SVD

Let $\text{rank}(A) = r$. let the SVD of A be $A = U\,\Sigma\,V^T$, with

$$\begin{aligned} U &= [\mathbf{u}_1 \quad \mathbf{u}_2 \quad \cdots \quad \mathbf{u}_n], \\ \Sigma &= \mathbf{diag}(\sigma_1, \sigma_2, \cdots, \sigma_n), \\ V &= [\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_n], \end{aligned}$$

and σ_r be the **smallest** positive singular value. Since

$$A = U\,\Sigma\,V^T \iff AV = U\Sigma V^T V = U\Sigma,$$

we have

$$\begin{aligned} AV &= A[\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_n] = [A\mathbf{v}_1 \quad A\mathbf{v}_2 \quad \cdots \quad A\mathbf{v}_n] \\ &= [\mathbf{u}_1 \quad \cdots \quad \mathbf{u}_r \quad \cdots \quad \mathbf{u}_n] \begin{bmatrix} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_r & & \\ & & & \ddots & \\ & & & & 0 \end{bmatrix} \\ &= [\sigma_1\mathbf{u}_1 \quad \cdots \quad \sigma_r\mathbf{u}_r \quad \mathbf{0} \quad \cdots \quad \mathbf{0}]. \end{aligned} \tag{7.26}$$

Therefore,

$$A = U\,\Sigma\,V^T \Leftrightarrow \begin{cases} A\mathbf{v}_j = \sigma_j\mathbf{u}_j, & j = 1, 2, \cdots, r \\ A\mathbf{v}_j = \mathbf{0}, & j = r+1, \cdots, n \end{cases} \tag{7.27}$$

Similarly, starting from $A^T = V\,\Sigma\,U^T$,

$$A^T = V\,\Sigma\,U^T \Leftrightarrow \begin{cases} A^T\mathbf{u}_j = \sigma_j\mathbf{v}_j, & j = 1, 2, \cdots, r \\ A^T\mathbf{u}_j = \mathbf{0}, & j = r+1, \cdots, n \end{cases} \tag{7.28}$$

Summary **7.11.** It follows from (7.27) and (7.28) that

- $(\mathbf{v}_j, \sigma_j^2)$, $j = 1, 2, \cdots, r$, are eigenvector-eigenvalue pairs of $A^T A$.

$$A^T A\, \mathbf{v}_j = A^T(\sigma_j \mathbf{u}_j) = \sigma_j^2 \mathbf{v}_j, \;\; j = 1, 2, \cdots, r. \tag{7.29}$$

  So, the singular values play the role of eigenvalues.

- Similarly, we have

$$AA^T\, \mathbf{u}_j = A(\sigma_j \mathbf{v}_j) = \sigma_j^2 \mathbf{u}_j, \;\; j = 1, 2, \cdots, r. \tag{7.30}$$

- Equation (7.29) gives how to find the **singular values** $\{\sigma_j\}$ and the **right singular vectors** V, while (7.27) shows a way to compute the **left singular vectors** U.

- **(Dyadic decomposition)** The matrix $A \in \mathbb{R}^{m\times n}$ can be expressed as

$$A = \sum_{j=1}^{n} \sigma_j \mathbf{u}_j \mathbf{v}_j^T. \tag{7.31}$$

  When $\mathrm{rank}(A) = r \leq n$,

$$A = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^T. \tag{7.32}$$

  This property has been utilized for various approximations and applications, e.g., by dropping singular vectors corresponding to *small* singular values.

Geometric interpretation of the SVD

The matrix A maps an **orthonormal basis**

$$\mathcal{B}_1 = \{\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_n\}$$

of $\mathbb{R}^n$ onto a new "scaled" **orthogonal basis**

$$\mathcal{B}_2 = \{\sigma_1\mathbf{u}_1, \sigma_2\mathbf{u}_2, \cdots, \sigma_r\mathbf{u}_r\}$$

for a subspace of $\mathbb{R}^m$:

$$\mathcal{B}_1 = \{\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_n\} \xrightarrow{A} \mathcal{B}_2 = \{\sigma_1\mathbf{u}_1, \sigma_2\mathbf{u}_2, \cdots, \sigma_r\mathbf{u}_r\} \tag{7.33}$$

Consider a unit sphere $\mathcal{S}^{n-1}$ in $\mathbb{R}^n$:

$$\mathcal{S}^{n-1} = \left\{\mathbf{x} \middle| \sum_{j=1}^{n} x_j^2 = 1\right\}.$$

Then, $\forall \mathbf{x} \in \mathcal{S}^{n-1}$,

$$\begin{aligned} \mathbf{x} &= x_1\mathbf{v}_1 + x_2\mathbf{v}_2 + \cdots + x_n\mathbf{v}_n \\ A\mathbf{x} &= \sigma_1 x_1\mathbf{u}_1 + \sigma_2 x_2\mathbf{u}_2 + \cdots + \sigma_r x_r\mathbf{u}_r \\ &= y_1\mathbf{u}_1 + y_2\mathbf{u}_2 + \cdots + y_r\mathbf{u}_r, \quad (y_j = \sigma_j x_j) \end{aligned} \tag{7.34}$$

So, we have

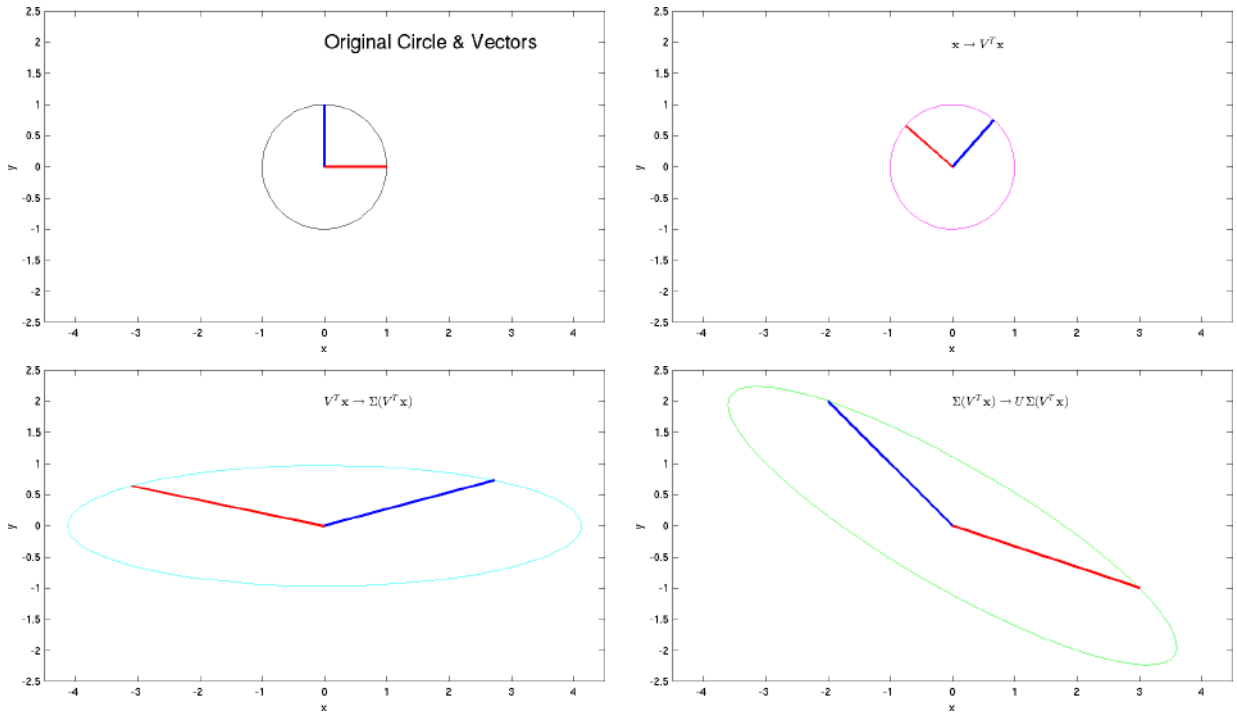
$$\begin{aligned} y_j = \sigma_j x_j \iff x_j &= \frac{y_j}{\sigma_j} \\ \sum_{j=1}^{n} x_j^2 = 1 \text{ (sphere)} \iff \sum_{j=1}^{r} \frac{y_j^2}{\sigma_j^2} &= \alpha \leq 1 \text{ (ellipsoid)} \end{aligned} \tag{7.35}$$

Example 7.12. *We build the set $A(\mathcal{S}^{n-1})$ by multiplying one factor of $A = U\Sigma V^T$ at a time. Assume for simplicity that $A \in \mathbb{R}^{2\times 2}$ and nonsingular. Let*

$$\begin{aligned} A &= \begin{bmatrix} 3 & -2 \\ -1 & 2 \end{bmatrix} = U\Sigma V^T \\ &= \begin{bmatrix} -0.8649 & 0.5019 \\ 0.5019 & 0.8649 \end{bmatrix} \begin{bmatrix} 4.1306 & 0 \\ 0 & 0.9684 \end{bmatrix} \begin{bmatrix} -0.7497 & 0.6618 \\ 0.6618 & 0.7497 \end{bmatrix} \end{aligned}$$

Then, for $\mathbf{x} \in \mathcal{S}^1$,

$$A\mathbf{x} = U\Sigma V^T\mathbf{x} = U\left(\Sigma(V^T\mathbf{x})\right)$$



In general,

- $V^T : \mathcal{S}^{n-1} \to \mathcal{S}^{n-1}$ (rotation in $\mathbb{R}^n$)
- $\Sigma : \mathbf{e}_j \mapsto \sigma_j \mathbf{e}_j$ (scaling from $\mathcal{S}^{n-1}$ to $\mathbb{R}^n$)
- $U : \mathbb{R}^n \to \mathbb{R}^m$ (rotation)

7.2.2. Properties of the SVD

Theorem 7.13. *Let $A \in \mathbb{R}^{m\times n}$ with $m \geq n$. Let $A = U\Sigma V^T$ be the SVD of A, with*

$$\sigma_1 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_n = 0.$$

Then,

$$\begin{cases} \textit{rank}(A) &= r \\ \textit{Null}(A) &= \textit{Span}\{\mathbf{v}_{r+1}, \cdots, \mathbf{v}_n\} \\ \textit{Range}(A) &= \textit{Span}\{\mathbf{u}_1, \cdots, \mathbf{u}_r\} \\ A &= \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^T \end{cases} \tag{7.36}$$

and

$$\begin{cases} ||A||_2 &= \sigma_1 \quad \text{(See Exercise 2.)} \\ ||A||_F^2 &= \sigma_1^2 + \cdots + \sigma_r^2 \quad \text{(See Exercise 3.)} \\ \min\limits_{\mathbf{x}\neq\mathbf{0}} \dfrac{||A\mathbf{x}||_2}{||\mathbf{x}||_2} &= \sigma_n \qquad (m \geq n) \\ \kappa_2(A) &= ||A||_2 \cdot ||A^{-1}||_2 = \dfrac{\sigma_1}{\sigma_n} \\ & \quad (\ \textit{when}\ m = n, \,\&\, \exists A^{-1}) \end{cases} \tag{7.37}$$

Theorem 7.14. *Let $A \in \mathbb{R}^{m\times n}$, $m \geq n$, $\textit{rank}(A) = n$, with singular values*

$$\sigma_1 \geq \sigma_2 \geq \cdots \sigma_n > 0.$$

Then

$$\begin{aligned} ||(A^TA)^{-1}||_2 &= \sigma_n^{-2}, \\ ||(A^TA)^{-1}A^T||_2 &= \sigma_n^{-1}, \\ ||A(A^TA)^{-1}||_2 &= \sigma_n^{-1}, \\ ||A(A^TA)^{-1}A^T||_2 &= 1. \end{aligned} \tag{7.38}$$

Definition 7.15. *$(A^TA)^{-1}A^T$ is called the* **pseudoinverse** *of A, while $A(A^TA)^{-1}$ is called the* **pseudoinverse** *of A^T. Let $A = U\Sigma V^T$ be the SVD of A. Then*

$$(A^TA)^{-1}A^T = V\Sigma^{-1}U^T \overset{\text{def}}{=\!=} A^+. \tag{7.39}$$

Theorem 7.16. *Let $A \in \mathbb{R}^{m\times n}$ with rank$(A) = r > 0$. Let $A = U\Sigma V^T$ be the SVD of A, with singular values*

$$\sigma_1 \geq \cdots \geq \sigma_r > 0.$$

Define, for $k = 1, \cdots, r-1$,

$$A_k = \sum_{j=1}^{k} \sigma_j \mathbf{u}_j \mathbf{v}_j^T \quad \text{(sum of rank-1 matrices)}.$$

Then, rank$(A_k) = k$ and

$$\begin{aligned} ||A - A_k||_2 &= \min\{||A - B||_2 \,\big|\, \textbf{rank}(B) \leq k\} \\ &= \sigma_{k+1}, \\ ||A - A_k||_F^2 &= \min\{||A - B||_F^2 \,\big|\, \textbf{rank}(B) \leq k\} \\ &= \sigma_{k+1}^2 + \cdots + \sigma_r^2. \end{aligned} \tag{7.40}$$

That is, of all matrices of rank $\leq k$, $\boldsymbol{A_k}$ is closest to $\boldsymbol{A}$.

Note: The matrix A_k can be written as

$$A_k = U\,\Sigma_k V^T, \tag{7.41}$$

where $\Sigma_k = \text{diag}(\sigma_1, \cdots, \sigma_k, 0, \cdots, 0)$. The **pseudoinverse of A_k** reads

$$A_k^+ = V\Sigma_k^+ U^T, \tag{7.42}$$

where

$$\Sigma_k^+ = \text{diag}(1/\sigma_1,\, 1/\sigma_2,\, \cdots,\, 1/\sigma_k,\, 0, \cdots, 0). \tag{7.43}$$

Corollary 7.17. *Suppose $A \in \mathbb{R}^{m\times n}$ has full rank; rank$(A) = n$. Let $\sigma_1 \geq \cdots \geq \sigma_n$ be the singular values of A. Let $B \in \mathbb{R}^{m\times n}$ satisfy*

$$||A - B||_2 < \sigma_n.$$

Then B also has full rank.

Full SVD

- For $A \in \mathbb{R}^{m \times n}$,

$$A = U\Sigma V^T \iff U^T A V = \Sigma,$$

where $U \in \mathbb{R}^{m \times n}$ and $\Sigma, V \in \mathbb{R}^{n \times n}$.

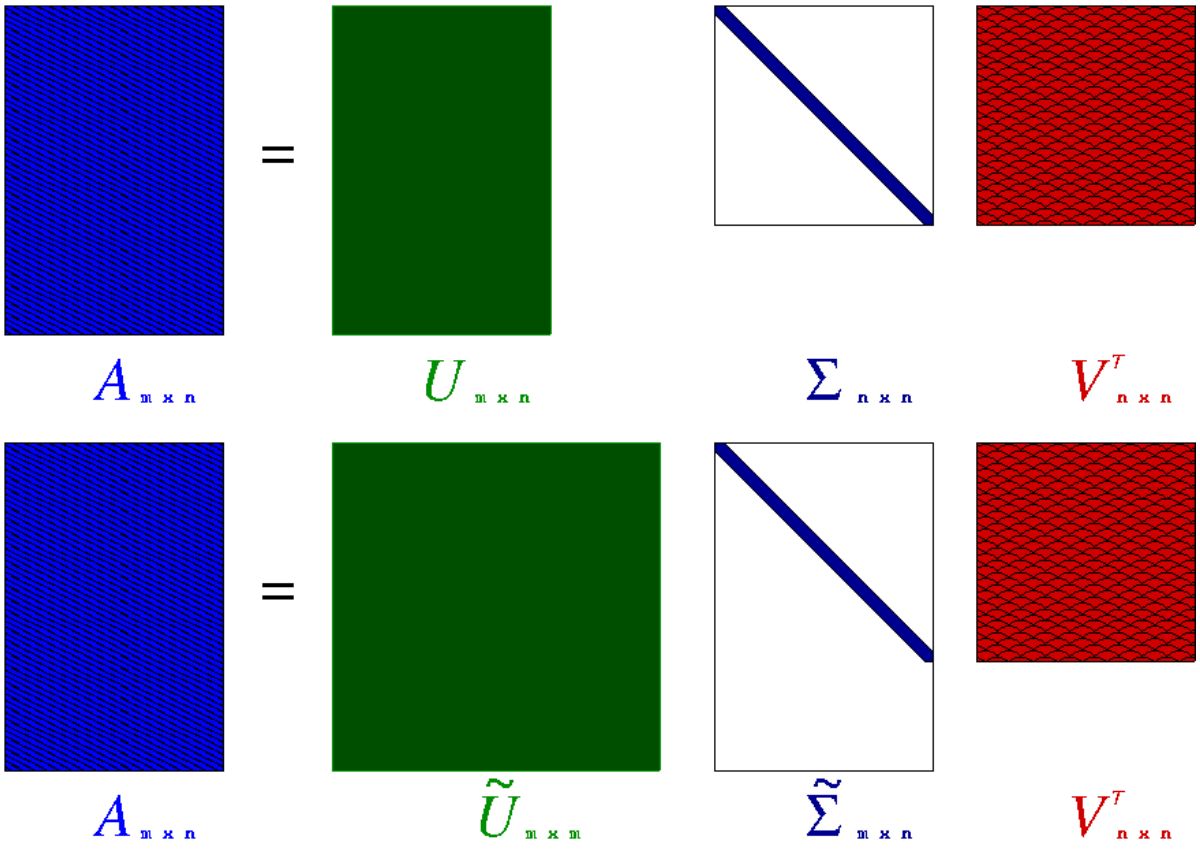
- Expand

$$\begin{aligned} U &\rightarrow \tilde{U} = [U \;\; U_2] \in \mathbb{R}^{m \times m}, \quad \text{(orthogonal)} \\ \Sigma &\rightarrow \tilde{\Sigma} = \begin{bmatrix} \Sigma \\ O \end{bmatrix} \in \mathbb{R}^{m \times n}, \end{aligned}$$

where O is an $(m-n) \times n$ zero matrix.

- Then,

$$\tilde{U}\tilde{\Sigma}V^T = [U \;\; U_2] \begin{bmatrix} \Sigma \\ O \end{bmatrix} V^T = U\Sigma V^T = A \tag{7.44}$$



7.2.3. Computation of the SVD

For $A \in \mathbb{R}^{m\times n}$, the procedure is as follows.

1. Form A^TA (A^TA – **covariance matrix** of A).
2. Find the eigen-decomposition of A^TA by orthogonalization process, i.e., $\Lambda = \text{diag}(\lambda_1, \cdots, \lambda_n)$,
$$A^TA = V\Lambda V^T,$$
where $V = [\mathbf{v}_1 \quad \cdots \quad \mathbf{v}_n]$ is orthogonal, i.e., $V^TV = I$.
3. Sort the eigenvalues according to their magnitude and let
$$\sigma_j = \sqrt{\lambda_j}, \;\; j = 1, 2, \cdots, n.$$
4. Form the U matrix as follows,
$$\mathbf{u}_j = \frac{1}{\sigma_j} A\mathbf{v}_j, \quad j = 1, 2, \cdots, r.$$
If necessary, pick up the remaining columns of U so it is orthogonal. (These additional columns must be in Null(AA^T).)
5. $A = U\Sigma V^T = [\mathbf{u}_1 \; \cdots \; \mathbf{u}_r \; \cdots \; \mathbf{u}_n] \,\text{diag}(\sigma_1, \cdots, \sigma_r, 0, \cdots, 0) \begin{bmatrix} \mathbf{v}_1^T \\ \vdots \\ \mathbf{v}_n^T \end{bmatrix}$

Lemma 7.18. *Let $A \in \mathbb{R}^{n\times n}$ be symmetric. Then (a) all the eigenvalues of A are real and (b) eigenvectors corresponding to distinct eigenvalues are orthogonal.*

Proof. See Exercise 4. □

Example 7.19. *Find the* SVD *for* $A = \begin{bmatrix} 1 & 2 \\ -2 & 1 \\ 3 & 2 \end{bmatrix}$.

Solution.

1. $A^TA = \begin{bmatrix} 14 & 6 \\ 6 & 9 \end{bmatrix}$.

2. Solving $\det(A^TA - \lambda I) = 0$ gives the eigenvalues of A^TA

$$\lambda_1 = 18 \text{ and } \lambda_2 = 5,$$

of which corresponding eigenvectors are

$$\tilde{\mathbf{v}}_1 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}, \tilde{\mathbf{v}}_2 = \begin{bmatrix} -2 \\ 3 \end{bmatrix}. \Longrightarrow V = \begin{bmatrix} \frac{3}{\sqrt{13}} & -\frac{2}{\sqrt{13}} \\ \frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix}$$

3. $\sigma_1 = \sqrt{\lambda_1} = \sqrt{18} = 3\sqrt{2}$, $\sigma_2 = \sqrt{\lambda_2} = \sqrt{5}$. So

$$\Sigma = \begin{bmatrix} \sqrt{18} & 0 \\ 0 & \sqrt{5} \end{bmatrix}$$

4. $\mathbf{u}_1 = \frac{1}{\sigma_1}A\mathbf{v}_1 = \frac{1}{\sqrt{18}}A \begin{bmatrix} \frac{3}{\sqrt{13}} \\ \frac{2}{\sqrt{13}} \end{bmatrix} = \frac{1}{\sqrt{18}}\frac{1}{\sqrt{13}} \begin{bmatrix} 7 \\ -4 \\ 13 \end{bmatrix} = \begin{bmatrix} \frac{7}{\sqrt{234}} \\ -\frac{4}{\sqrt{234}} \\ \frac{13}{\sqrt{234}} \end{bmatrix}$

$$\mathbf{u}_2 = \frac{1}{\sigma_2}A\mathbf{v}_2 = \frac{1}{\sqrt{5}}A \begin{bmatrix} \frac{-2}{\sqrt{13}} \\ \frac{3}{\sqrt{13}} \end{bmatrix} = \frac{1}{\sqrt{5}}\frac{1}{\sqrt{13}} \begin{bmatrix} 4 \\ 7 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{4}{\sqrt{65}} \\ \frac{7}{\sqrt{65}} \\ 0 \end{bmatrix}.$$

5. $A = U\Sigma V^T = \begin{bmatrix} \frac{7}{\sqrt{234}} & \frac{4}{\sqrt{65}} \\ -\frac{4}{\sqrt{234}} & \frac{7}{\sqrt{65}} \\ \frac{13}{\sqrt{234}} & 0 \end{bmatrix} \begin{bmatrix} \sqrt{18} & 0 \\ 0 & \sqrt{5} \end{bmatrix} \begin{bmatrix} \frac{3}{\sqrt{13}} & \frac{2}{\sqrt{13}} \\ -\frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix}$

Example 7.20. *Find the* ***pseudoinverse*** *of* A*,*

$$A^+ = (A^T A)^{-1} A^T = V \Sigma^{-1} U^T,$$

when $A = \begin{bmatrix} 1 & 2 \\ -2 & 1 \\ 3 & 2 \end{bmatrix}$.

Solution. From Example 7.19, we have

$$A = U \Sigma V^T = \begin{bmatrix} \frac{7}{\sqrt{234}} & \frac{4}{\sqrt{65}} \\ -\frac{4}{\sqrt{234}} & \frac{7}{\sqrt{65}} \\ \frac{13}{\sqrt{234}} & 0 \end{bmatrix} \begin{bmatrix} \sqrt{18} & 0 \\ 0 & \sqrt{5} \end{bmatrix} \begin{bmatrix} \frac{3}{\sqrt{13}} & \frac{2}{\sqrt{13}} \\ -\frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix}$$

Thus,

$$\begin{aligned} A^+ &= V \Sigma^{-1} U^T = \begin{bmatrix} \frac{3}{\sqrt{13}} & -\frac{2}{\sqrt{13}} \\ \frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{18}} & 0 \\ 0 & \frac{1}{\sqrt{5}} \end{bmatrix} \begin{bmatrix} \frac{7}{\sqrt{234}} & -\frac{4}{\sqrt{234}} & \frac{13}{\sqrt{234}} \\ \frac{4}{\sqrt{65}} & \frac{7}{\sqrt{65}} & 0 \end{bmatrix} \\ &= \begin{bmatrix} -\frac{1}{30} & -\frac{4}{15} & \frac{1}{6} \\ \frac{11}{45} & \frac{13}{45} & \frac{1}{9} \end{bmatrix} \end{aligned}$$

Computer implementation [25]

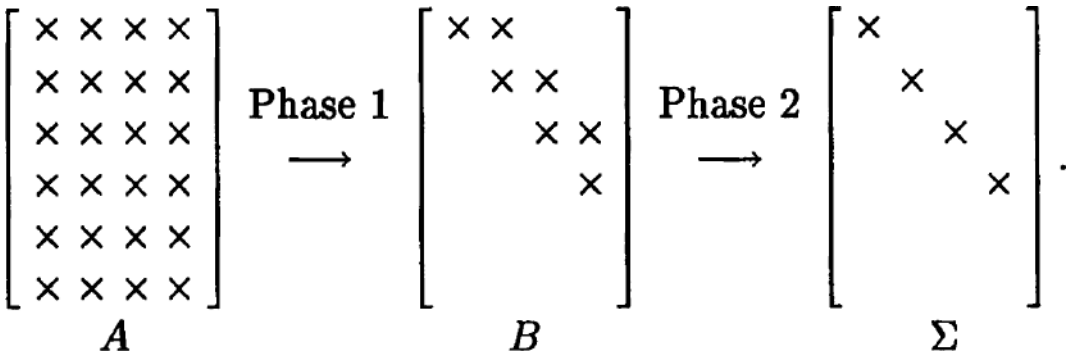


Figure 7.3: A two-phase procedure for the SVD: $A = U\Sigma V^T$.

Algorithm 7.21. (Golub and Reinsch, 1970) [26]. Let $A \in \mathbb{R}^{m\times n}$.

- **Phase 1**: It **constructs two finite sequences** of **Householder transformations** to find an **upper bidiagonal matrix**:

$$P_n \cdots P_1 \, A \, Q_1 \cdots Q_{n-2} \;=\; B \tag{7.45}$$

- **Phase 2**: It is to **iteratively diagonalize** B using the **QR method**.

Golub-Reinsch SVD algorithm

- It is extremely stable.
- Computational complexity:
  - Computation of U, V, and Σ: $4m^2n + 8mn^2 + 9n^3$.
  - Computation of V and Σ: $4mn^2 + 8n^3$.
- Phases 1 & 2 take $\mathcal{O}(mn^2)$ and $\mathcal{O}(n^2)$ flops, respectively. (when Phase 2 is done with $\mathcal{O}(n)$ iterations)
- Python: `U,S,V = numpy.linalg.svd(A)`
- Matlab/Maple: `[U,S,V] = svd(A)`
- Mathematica: `{U,S,V} = SingularValueDecomposition[A]`

Numerical rank

In the absence of round-off errors and uncertainties in the data, the SVD reveals the rank of the matrix. Unfortunately the presence of errors makes rank determination problematic. For example, consider

$$A = \begin{bmatrix} 1/3 & 1/3 & 2/3 \\ 2/3 & 2/3 & 4/3 \\ 1/3 & 2/3 & 3/3 \\ 2/5 & 2/5 & 4/5 \\ 3/5 & 1/5 & 4/5 \end{bmatrix} \tag{7.46}$$

- Obviously A is of rank 2, as its third column is the sum of the first two.
- Matlab “svd" (with IEEE double precision) produces

$$\sigma_1 = 2.5987, \quad \sigma_2 = 0.3682, \quad \text{and} \quad \sigma_3 = 8.6614 \times 10^{-17}.$$

- What is the rank of A, 2 or 3? What if σ_3 is in $\mathcal{O}(10^{-13})$?
- For this reason we must introduce a **threshold** T. Then we say that A has **numerical rank** r if A has r singular values larger than T, that is,

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > T \geq \sigma_{r+1} \geq \cdots \tag{7.47}$$

In Matlab

- Matlab has a “rank" command, which computes the numerical rank of the matrix with a default threshold

$$T = 2 \max\{m, n\} \, \epsilon \, ||A||_2 \tag{7.48}$$

  where ϵ is the unit round-off error.
- In Matlab, the unit round-off error can be found from the parameter “eps"

$$\text{eps} = 2^{-52} = 2.2204 \times 10^{-16}.$$

- For the matrix A in (7.46),

$$T = 2 \cdot 5 \cdot \text{eps} \cdot 2.5987 = 5.7702 \times 10^{-15}$$

  and therefore `rank(A)=2`.

See Exercise 5.

7.2.4. Application of the SVD to image compression

- $A \in \mathbb{R}^{m\times n}$ is a sum of rank-1 matrices (dyadic decomposition):

$$V = [\mathbf{v}_1, \cdots, \mathbf{v}_n], \quad U = [\mathbf{u}_1, \cdots, \mathbf{u}_n],$$
$$A = U\Sigma V^T = \sum_{i=1}^{n} \sigma_i \mathbf{u}_i \mathbf{v}_i^T, \quad \mathbf{u}_i \in \mathbb{R}^m, \quad \mathbf{v}_i \in \mathbb{R}^n. \tag{7.49}$$

- The approximation

$$A_k = U\Sigma_k V^T = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T \tag{7.50}$$

  is closest to A among matrices of rank$\leq k$, and

$$||A - A_k||_2 = \sigma_{k+1}. \tag{7.51}$$

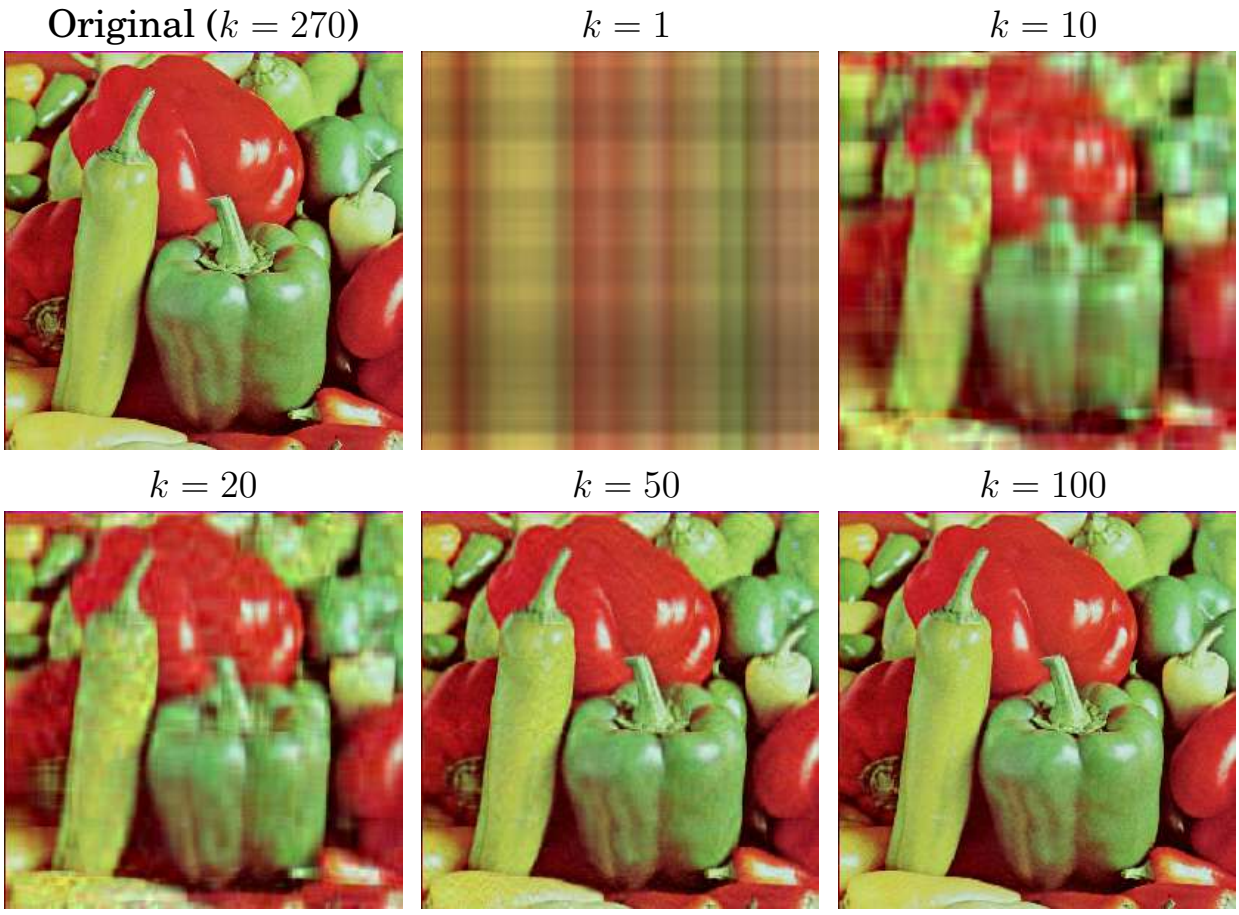
- It only takes $(m+n)\cdot k$ words to store $\mathbf{u}_1$ through $\mathbf{u}_k$, and $\sigma_1\mathbf{v}_1$ through $\sigma_k\mathbf{v}_k$, from which we can reconstruct A_k.

Image compression using k singular values

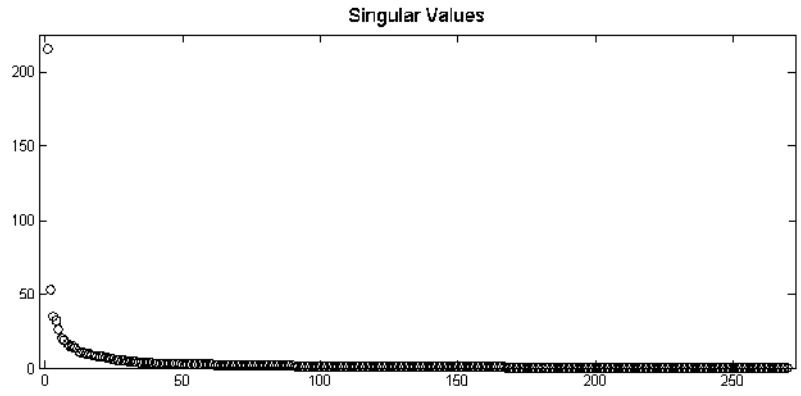
peppers_SVD.m

```
img = imread('Peppers.png'); [m,n,d]=size(img);
[U,S,V] = svd(reshape(im2double(img),m,[]));
%%---- select k <= p=min(m,n)
k = 20;
img_k = U(:,1:k)*S(1:k,1:k)*V(:,1:k)';
img_k = reshape(img_k,m,n,d);
figure, imshow(img_k)
```

The “Peppers" image is in $[270, 270, 3] \in \mathbb{R}^{270\times 810}$.



Peppers: Singular values



Peppers: Storage: It requires $(m+n) \cdot k$ words.For example, when $k = 50$,

$$(m+n) \cdot k = (270 + 810) \cdot 50 = \boxed{54{,}000}, \tag{7.52}$$

which is approximately **a quarter** the full storage space

$$270 \times 270 \times 3 = \boxed{218{,}700}.$$

7.3. Linear Discriminant Analysis

Linear discriminant analysis is a method to find a **linear combination of features** that **characterizes or separates** two or more classes of objects or events.

- The LDA is sometimes also called **Fisher's LDA**. Fisher *initially* formulated the LDA for **two-class classification problems** in 1936 [20], and later generalized for multi-class problems by C. Radhakrishna Rao under the assumption of **equal class covariances** and **normally distributed classes** in 1948 [61].
- The LDA may be used as a **linear classifier**, or, more commonly, for **dimensionality reduction** (§ 7.3.4) for a later classification.
- The general concept behind the LDA is very similar to PCA.[1]

LDA objective

- The LDA objective is to perform **dimensionality reduction**.
  - So what? PCA does that, too!
- However, we want to preserve as much of the **class discriminatory information** as possible.
  - OK, this is new!

LDA

- Consider a pattern classification problem, where we have c classes.
- Suppose each class has N_k samples in $\mathbb{R}^d$, where $k = 1, 2, \cdots, c$.
- Let $\mathcal{X}_k = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \cdots, \mathbf{x}^{(N_k)}\}$ be the set of d-dimensional samples for class k.
- Let $X \in \mathbb{R}^{d \times N}$ be the data matrix, stacking all the samples from all classes, such that each **column** represents a sample, where $N = \sum_k N_k$.
- The LDA seeks to obtain a transformation of X to Z through projecting the samples in X onto a **hyperplane** with dimension $c - 1$.

[1] In PCA, the main idea is to re-express the available dataset to extract the relevant information by **reducing the redundancy** and to **minimize the noise**. While (unsupervised) PCA attempts to *find the orthogonal component axes of maximum variance* in a dataset, the goal in the **(supervised)** LDA is to find the feature subspace that **optimizes class separability**.

7.3.1. Fisher's LDA (classifier): two classes

Let us define a transformation of samples $\mathbf{x}$ onto a line [$(c-1)$-space, for $c=2$]:

$$z = \mathbf{w}^T\mathbf{x} = \mathbf{w}\cdot\mathbf{x}, \tag{7.53}$$

where $\mathbf{w} \in \mathbb{R}^d$ is a **projection vector**.

Of all the possible lines, we would like to select the one that maximizes the separability of the scalars $\{z\}$.

- In order to find a good projection vector, we need to define a measure of separation between the projections.
- The mean vector of each class in $\mathbf{x}$ and z feature space is

$$\boldsymbol{\mu}_k = \frac{1}{N_k}\sum_{\mathbf{x}\in\mathcal{X}_k}\mathbf{x}, \quad \widetilde{\mu}_k = \frac{1}{N_k}\sum_{\mathbf{x}\in\mathcal{X}_k} z = \frac{1}{N_k}\sum_{\mathbf{x}\in\mathcal{X}_k}\mathbf{w}^T\mathbf{x} = \mathbf{w}^T\boldsymbol{\mu}_k, \tag{7.54}$$

  i.e., ***projecting*** $\mathbf{x}$ ***to*** z ***will lead to projecting the mean of*** $\mathbf{x}$ ***to the mean of*** z**.**
- We could then choose the **distance between the projected means** as our objective function:

$$\widehat{\mathcal{J}}(\mathbf{w}) = |\widetilde{\mu}_1 - \widetilde{\mu}_2| = |\mathbf{w}^T(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)|. \tag{7.55}$$

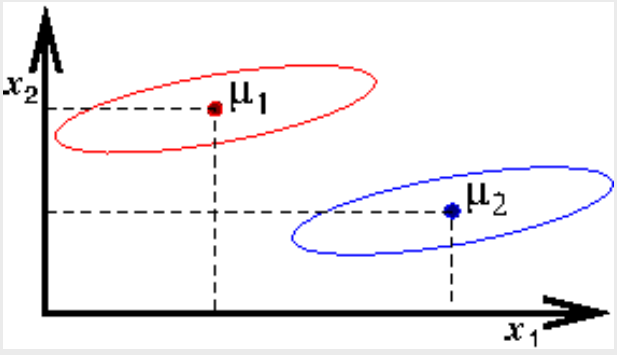


Figure 7.4: The x_1-axis has a larger distance between means, while the x_2-axis yields a better class separability.

- However, the distance between the projected means is **not a very good measure**, since it does not take into account the sample distribution within the classes.
- The maximizer $\mathbf{w}^*$ of (7.55) must be parallel to $(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$:

$$\mathbf{w}^* \,/\!/\, (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2);$$

  the projection to a parallel line of $(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$ is not an optimal transformation.

Fisher's LDA: The Key Idea

The solution proposed by Fisher is **to maximize a function that represents the difference between the means**, normalized by a measure of the **within-class variability** (called the **scatter**).

- For each class k, we define the **scatter** (an equivalent of the variance) as

$$\widetilde{s}_k^2 = \sum_{\mathbf{x} \in \mathcal{X}_k} (z - \widetilde{\mu}_k)^2, \quad z = \mathbf{w}^T \mathbf{x}. \tag{7.56}$$

- The quantity $\widetilde{s}_k^2$ measures the variability within class $\mathcal{X}_k$ after projecting it on the z-axis.
- Thus, $\widetilde{s}_1^2 + \widetilde{s}_2^2$ measures the variability within the two classes at hand after projection; it is called the **within-class scatter** of the projected samples.
- **Fisher's linear discriminant** is defined as the linear function $\mathbf{w}^T \mathbf{x}$ that maximizes the objective function:

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \mathcal{J}(\mathbf{w}), \quad \text{where} \quad \mathcal{J}(\mathbf{w}) = \frac{(\widetilde{\mu}_1 - \widetilde{\mu}_2)^2}{\widetilde{s}_1^2 + \widetilde{s}_2^2}. \tag{7.57}$$

- Therefore, Fisher's LDA searches for a projection where samples from the same class are projected very close to each other; at the same time, the projected means are as farther apart as possible.

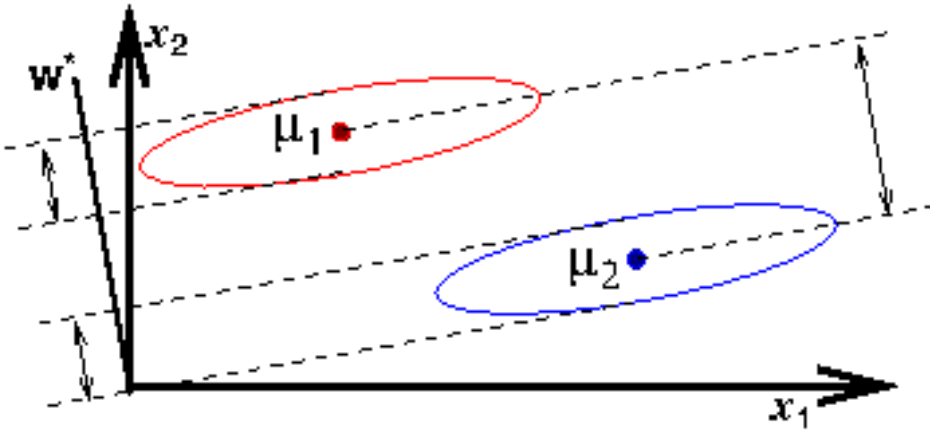


Figure 7.5: Fisher's LDA.

7.3.2. Fisher's LDA: the optimum projection

Rewrite the Fisher's objective function:

$$\mathcal{J}(\mathbf{w}) = \frac{(\widetilde{\mu}_1 - \widetilde{\mu}_2)^2}{\widetilde{s}_1^2 + \widetilde{s}_2^2}, \tag{7.58}$$

where $$\boldsymbol{\mu}_k = \frac{1}{N_k}\sum_{\mathbf{x}\in\mathcal{X}_k}\mathbf{x}, \quad \widetilde{\mu}_k = \mathbf{w}^T\boldsymbol{\mu}_k, \quad \widetilde{s}_k^2 = \sum_{\mathbf{x}\in\mathcal{X}_k}(z - \widetilde{\mu}_k)^2.$$

- In order to express $\mathcal{J}(\mathbf{w})$ as an explicit function of $\mathbf{w}$, we first define a measure of the scatter in the feature space $\mathbf{x}$:

$$S_w = S_1 + S_2, \quad \text{for } S_k = \sum_{\mathbf{x}\in\mathcal{X}_k}(\mathbf{x} - \boldsymbol{\mu}_k)(\mathbf{x} - \boldsymbol{\mu}_k)^T, \tag{7.59}$$

  where $S_w \in \mathbb{R}^{d\times d}$ is called the **within-class scatter matrix** of samples $\mathbf{x}$, while S_k is the **covariance matrix** of class $\mathcal{X}_k$.

  Then, the scatter of the projection z can then be expressed as

$$\begin{aligned}\widetilde{s}_k^2 &= \sum_{\mathbf{x}\in\mathcal{X}_k}(z - \widetilde{\mu}_k)^2 = \sum_{\mathbf{x}\in\mathcal{X}_k}(\mathbf{w}^T\mathbf{x} - \mathbf{w}^T\boldsymbol{\mu}_k)^2 \\ &= \sum_{\mathbf{x}\in\mathcal{X}_k}\mathbf{w}^T(\mathbf{x} - \boldsymbol{\mu}_k)(\mathbf{x} - \boldsymbol{\mu}_k)^T\mathbf{w} \\ &= \mathbf{w}^T\Big(\sum_{\mathbf{x}\in\mathcal{X}_k}(\mathbf{x} - \boldsymbol{\mu}_k)(\mathbf{x} - \boldsymbol{\mu}_k)^T\Big)\mathbf{w} = \mathbf{w}^T S_k\mathbf{w}.\end{aligned} \tag{7.60}$$

  Thus, the denominator of the objective function gives

$$\widetilde{s}_1^2 + \widetilde{s}_2^2 = \mathbf{w}^T S_1\mathbf{w} + \mathbf{w}^T S_2\mathbf{w} = \mathbf{w}^T S_w\mathbf{w} =: \widetilde{S}_w, \tag{7.61}$$

  where $\widetilde{S}_w$ is the **within-class scatter** of projected samples z.

- Similarly, the difference between the projected means (in z-space) can be expressed in terms of the means in the original feature space ($\mathbf{x}$-space).

$$\begin{aligned}(\widetilde{\mu}_1 - \widetilde{\mu}_2)^2 &= (\mathbf{w}^T\boldsymbol{\mu}_1 - \mathbf{w}^T\boldsymbol{\mu}_2)^2 = \mathbf{w}^T\underbrace{(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T}_{=:S_b}\mathbf{w} \\ &= \mathbf{w}^T S_b\mathbf{w} =: \widetilde{S}_b,\end{aligned} \tag{7.62}$$

where the rank-one matrix $S_b \in \mathbb{R}^{d\times d}$ is called the **between-class scatter matrix** of the original samples $\mathbf{x}$, while $\widetilde{S}_b$ is the **between-class scatter** of the projected samples z.

- Since S_b is the outer product of two vectors, $\text{rank}(S_b) \leq 1$.

We can finally express the Fisher criterion in terms of S_w and S_b as

$$\mathcal{J}(\mathbf{w}) = \frac{(\widetilde{\mu}_1 - \widetilde{\mu}_2)^2}{\widetilde{s}_1^2 + \widetilde{s}_2^2} = \frac{\mathbf{w}^T S_b \mathbf{w}}{\mathbf{w}^T S_w \mathbf{w}}. \tag{7.63}$$

Hence, $\mathcal{J}(\mathbf{w})$ is a measure of the difference between class means (encoded in the between-class scatter matrix), normalized by a measure of the within-class scatter matrix.

- To find the maximum of $\mathcal{J}(\mathbf{w})$, we differentiate it with respect to $\mathbf{w}$ and equate to zero. Applying some algebra leads (Exercise 6)

$$S_w^{-1} S_b \mathbf{w} = \mathcal{J}(\mathbf{w})\, \mathbf{w}. \tag{7.64}$$

Note that $S_w^{-1} S_b$ is a **rank-one matrix**.

Equation (7.64) is a **generalized eigenvalue problem**:

$$S_w^{-1} S_b\, \mathbf{w} = \lambda\, \mathbf{w} \iff S_b\, \mathbf{w} = \lambda\, S_w\, \mathbf{w}; \tag{7.65}$$

the maximizer $\mathbf{w}^*$ of $\mathcal{J}(\mathbf{w})$ is the eigenvector associated with the **nonzero eigenvalue** $\lambda^* = \mathcal{J}(\mathbf{w})$.

Summary **7.22.** Finding the eigenvector of $S_w^{-1} S_b$ associated with the largest eigenvalue yields

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \mathcal{J}(\mathbf{w}) = \arg\max_{\mathbf{w}} \frac{\mathbf{w}^T S_b \mathbf{w}}{\mathbf{w}^T S_w \mathbf{w}}. \tag{7.66}$$

This is known as **Fisher's linear discriminant analysis**, although it is not a discriminant but a specific choice of direction for the projection of the data down to one dimension.

Fisher's LDA: an example

We will compute the Linear Discriminant projection for the following two-dimensional dataset of two classes ($c = 2$).

lda_Fisher.m

```
m=2; n=5;

X1=[2,3; 4,3; 2,1; 3,4;  5,4];
X2=[7,4; 6,8; 7,6; 8,9; 10,9];

Mu1 = mean(X1)';              % Mu1 = [3.2,3.0]
Mu2 = mean(X2)';              % Mu2 = [7.6,7.2]

S1 = cov(X1,0)*n;
S2 = cov(X2,0)*n;
Sw = S1+S2;                   % Sw = [20,13; 13,31]

Sb = (Mu1-Mu2)*(Mu1-Mu2)';  % Sb = [19.36,18.48; 18.48,17.64]

invSw_Sb = inv(Sw)*Sb;
[V,L] = eig(invSw_Sb);        % V1 = [ 0.9503,0.3113]; L1 = 1.0476
                              % V2 = [-0.6905,0.7234]; L2 = 0.0000
```

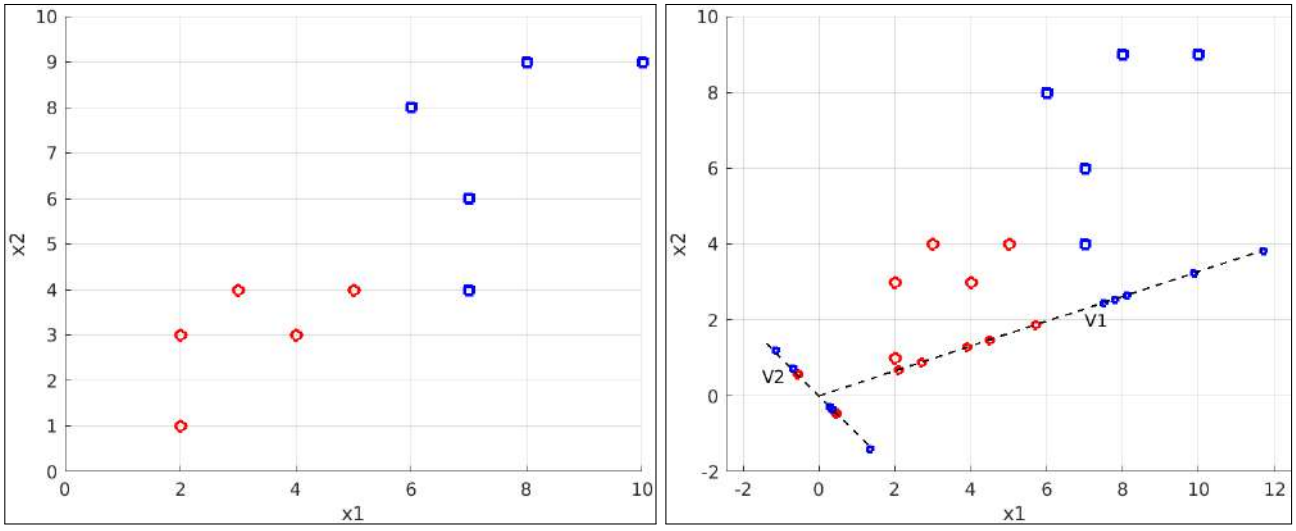


Figure 7.6: A synthetic dataset and Fisher's LDA projection.

7.3.3. LDA for Multiple Classes

- Now, we have c-classes instead of just two.
- We are now seeking $(c-1)$ projections $[z_1, z_2, \cdots, z_{c-1}]$ by means of $(c-1)$ projection vectors $\mathbf{w}_k \in \mathbb{R}^d$.
- Let $W = [\mathbf{w}_1|\mathbf{w}_2|\cdots|\mathbf{w}_{c-1}]$, a collection of column vectors, such that

$$z_k = \mathbf{w}_k^T\mathbf{x} \Longrightarrow \mathbf{z} = W^T\mathbf{x} \in \mathbb{R}^{c-1}. \tag{7.67}$$

- If we have N sample (column) vectors, we can stack them into one matrix as follows.

$$Z = W^T X, \tag{7.68}$$

where $X \in \mathbb{R}^{d\times N}$, $W \in \mathbb{R}^{d\times(c-1)}$, and $Z \in \mathbb{R}^{(c-1)\times N}$.

Recall: For the two classes case, the **within-class scatter matrix** was computed as

$$S_w = S_1 + S_2.$$

This can be generalized in the c-classes case as:

$$S_w = \sum_{k=1}^{c} S_k, \quad S_k = \sum_{\mathbf{x}\in\mathcal{X}_k} (\mathbf{x}-\boldsymbol{\mu}_k)(\mathbf{x}-\boldsymbol{\mu}_k)^T, \tag{7.69}$$

where $\boldsymbol{\mu}_k = \frac{1}{N_k}\sum_{\mathbf{x}\in\mathcal{X}_k}\mathbf{x}$, where N_k is the number of data samples in class $\mathcal{X}_k$, and $S_w \in \mathbb{R}^{d\times d}$.

Recall: For the two classes case, the **between-class scatter matrix** was computed as

$$S_b = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T.$$

For c-classes case, we will measure the **between-class scatter matrix** with respect to the mean of all classes as follows:

$$S_b = \sum_{k=1}^{c} N_k(\boldsymbol{\mu}_k - \boldsymbol{\mu})(\boldsymbol{\mu}_k - \boldsymbol{\mu})^T, \quad \boldsymbol{\mu} = \frac{1}{N}\sum_{\forall\mathbf{x}}\mathbf{x}, \tag{7.70}$$

where $\text{rank}(S_b) = c - 1$.

Definition 7.23. *As an analogue to (7.66), we may define the LDA optimization, for c classes case, as follows.*

$$W^* = \arg\max_W \mathcal{J}(W) = \arg\max_W \frac{W^T S_b W}{W^T S_w W}. \tag{7.71}$$

Recall: For two-classes case, when we set $\frac{\partial \mathcal{J}(\mathbf{w})}{\partial \mathbf{w}} = 0$, the optimization problem is reduced to the eigenvalue problem

$$S_w^{-1} S_b \, \mathbf{w}^* = \lambda^* \, \mathbf{w}^*, \;\; \text{where} \;\; \lambda^* = \mathcal{J}(\mathbf{w}^*).$$

For c-classes case, we have $(c-1)$ projection vectors. Hence the eigenvalue problem can be generalized to the c-classes case:

$$S_w^{-1} S_b \, \mathbf{w}_k^* = \lambda_k^* \, \mathbf{w}_k^*, \quad \lambda_k^* = \mathcal{J}(\mathbf{w}_k^*), \quad k = 1, 2, \cdots, c-1. \tag{7.72}$$

Thus, it can be shown that the optimal projection matrix

$$W^* = [\mathbf{w}_1^* | \mathbf{w}_1^* | \cdots | \mathbf{w}_{c-1}^*] \in \mathbb{R}^{d \times (c-1)} \tag{7.73}$$

is the one whose columns are the eigenvectors corresponding to the **eigenvalues** of the following generalized eigenvalue problem:

$$S_w^{-1} S_b \, W^* = \boldsymbol{\lambda}^* \cdot W^*, \quad \boldsymbol{\lambda}^* = [\lambda_1^*, \cdots, \lambda_{c-1}^*], \tag{7.74}$$

where $S_w^{-1} S_b \in \mathbb{R}^{d \times d}$ and $(\cdot)$ denotes the pointwise product.

Illustration – 3 classes

- Let us generate a dataset for each class to illustrate the LDA transformation.
- For each class:
  - Use the random number generator to generate a uniform stream of 500 samples that follows $\mathcal{U}(0,1)$.
  - Using the Box-Muller approach, convert the generated uniform stream to $\mathcal{N}(0,1)$.
  - Then use the method of eigenvalues and eigenvectors to manipulate the standard normal to have the required mean vector and covariance matrix .
  - Estimate the mean and covariance matrix of the resulted dataset.

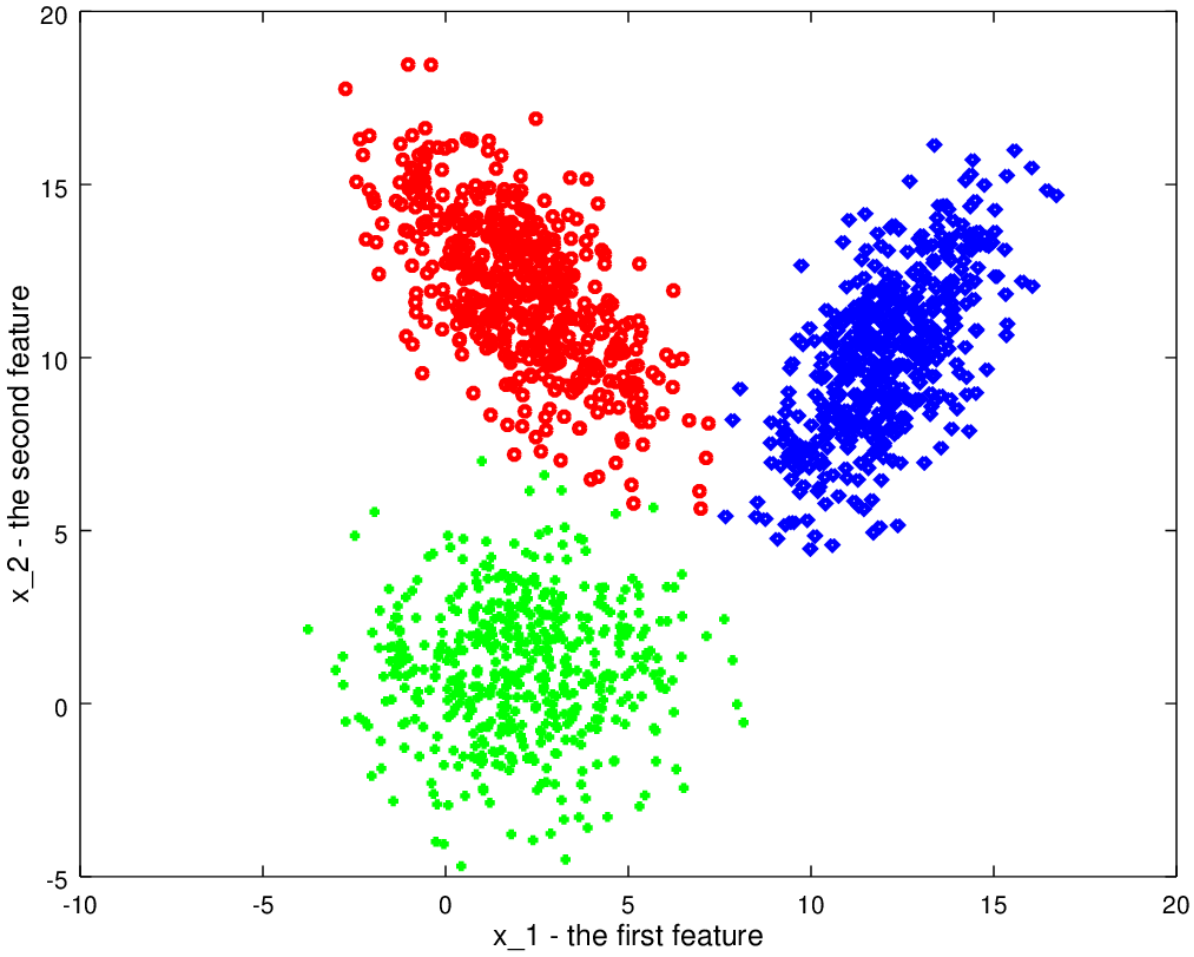


Figure 7.7: Generated and manipulated dataset, for 3 classes.

lda_c3.m

```
close all;
try, pkg load statistics; end  % for octave

%% uniform stream
U  = rand(2,1000); u1 = U(:,1:2:end); u2 = U(:,2:2:end);

%% Box-Muller method to convert to N(0,1)
X = sqrt((-2).*log(u1)).*(cos(2*pi.*u2));  % 2 x 500
clear u1 u2 U;

%% manipulate for required Mean and Cov
Mu = [5;5];

Mu1= Mu +[-3;7];  Cov1 =[5 -1; -3 3];
X1 = denormalize(X,Mu1,Cov1);
Mu2= Mu +[-3;-4]; Cov2 =[4 0; 0 4];
X2 = denormalize(X,Mu2,Cov2);
Mu3= Mu +[7; 5];  Cov3 =[4 1; 3 3];
X3 = denormalize(X,Mu3,Cov3);

%%Begin the comptation of the LDA Projection Vectors
% estimate mean and covariance
N1  = size(X1,2); N2  = size(X2,2); N3  = size(X3,2);
Mu1 = mean(X1')'; Mu2 = mean(X2')'; Mu3 = mean(X3')';
Mu = (Mu1+Mu2+Mu3)/3.;

% within-class scatter matrix
S1 = cov(X1'); S2 = cov(X2'); S3 = cov(X3');
Sw = S1+S2+S3;

% between-class scatter matrix
Sb1 = N1 * (Mu1-Mu)*(Mu1-Mu)'; Sb2 = N2 * (Mu2-Mu)*(Mu2-Mu)';
Sb3 = N3 * (Mu3-Mu)*(Mu3-Mu)';
Sb = Sb1+Sb2+Sb3;

% computing the LDA projection
invSw_Sb = inv(Sw)*Sb; [V,D] = eig(invSw_Sb);
w1 = V(:,1); w2 = V(:,2);
if D(1,1)<D(2,2), w1 = V(:,2); w2 = V(:,1); end

lda_c3_visualize;
```

Figure 7.8: lda_c3.m

denormalize.m

```
function Xnew = denormalize(X,Mu,Cov)
% it manipulates data samples in N(0,1) to something else.

[V,D] = eig(Cov); VsD = V*sqrt(D);

Xnew = zeros(size(X));
for j=1:size(X,2)
    Xnew(:,j)= VsD * X(:,j);
end

%Now, add "replicated and tiled Mu"
Xnew = Xnew +repmat(Mu,1,size(Xnew,2));
```

lda_c3_visualize.m

```
figure, hold on; axis([-10 20 -5 20]);
   xlabel('x_1 - the first feature','fontsize',12);
   ylabel('x_2 - the second feature','fontsize',12);
   plot(X1(1,:)',X1(2,:)','ro','markersize',4,"linewidth",2)
   plot(X2(1,:)',X2(2,:)','g+','markersize',4,"linewidth",2)
   plot(X3(1,:)',X3(2,:)','bd','markersize',4,"linewidth",2)
hold off
print -dpng 'LDA_c3_Data.png'

figure, hold on; axis([-10 20 -5 20]);
   xlabel('x_1 - the first feature','fontsize',12);
   ylabel('x_2 - the second feature','fontsize',12);
   plot(X1(1,:)',X1(2,:)','ro','markersize',4,"linewidth",2)
   plot(X2(1,:)',X2(2,:)','g+','markersize',4,"linewidth",2)
   plot(X3(1,:)',X3(2,:)','bd','markersize',4,"linewidth",2)

   plot(Mu1(1),Mu1(2),'c.','markersize',20)
   plot(Mu2(1),Mu2(2),'m.','markersize',20)
   plot(Mu3(1),Mu3(2),'r.','markersize',20)
   plot(Mu(1),Mu(2),'k*','markersize',15,"linewidth",3)
   text(Mu(1)+0.5,Mu(2)-0.5,'\mu','fontsize',18)

   t = -5:20; line1_x = t*w1(1); line1_y = t*w1(2);
   plot(line1_x,line1_y,'k-',"linewidth",3);
   t = -5:10; line2_x = t*w2(1); line2_y = t*w2(2);
   plot(line2_x,line2_y,'m--',"linewidth",3);
hold off
print -dpng 'LDA_c3_Data_projection.png'

%Project the samples through w1
wk = w1;
z1_wk = wk'*X1; z2_wk = wk'*X2; z3_wk = wk'*X3;
```

```
33
34 z1_wk_Mu = mean(z1_wk); z1_wk_sigma = std(z1_wk);
35 z1_wk_pdf = mvnpdf(z1_wk',z1_wk_Mu,z1_wk_sigma);
36
37 z2_wk_Mu = mean(z2_wk); z2_wk_sigma = std(z2_wk);
38 z2_wk_pdf = mvnpdf(z2_wk',z2_wk_Mu,z2_wk_sigma);
39
40 z3_wk_Mu = mean(z3_wk); z3_wk_sigma = std(z3_wk);
41 z3_wk_pdf = mvnpdf(z3_wk',z3_wk_Mu,z3_wk_sigma);
42
43 figure, plot(z1_wk,z1_wk_pdf,'ro',z2_wk,z2_wk_pdf,'g+',...
44      z3_wk,z3_wk_pdf,'bd')
45 xlabel('z','fontsize',12); ylabel('p(z|w1)','fontsize',12);
46 print -dpng 'LDA_c3_Xw1_pdf.png'
47
48 %Project the samples through w2
49 wk = w2;
50 z1_wk = wk'*X1; z2_wk = wk'*X2; z3_wk = wk'*X3;
51
52 z1_wk_Mu = mean(z1_wk); z1_wk_sigma = std(z1_wk);
53 z1_wk_pdf = mvnpdf(z1_wk',z1_wk_Mu,z1_wk_sigma);
54
55 z2_wk_Mu = mean(z2_wk); z2_wk_sigma = std(z2_wk);
56 z2_wk_pdf = mvnpdf(z2_wk',z2_wk_Mu,z2_wk_sigma);
57
58 z3_wk_Mu = mean(z3_wk); z3_wk_sigma = std(z3_wk);
59 z3_wk_pdf = mvnpdf(z3_wk',z3_wk_Mu,z3_wk_sigma);
60
61 figure, plot(z1_wk,z1_wk_pdf,'ro',z2_wk,z2_wk_pdf,'g+',...
62      z3_wk,z3_wk_pdf,'bd')
63 xlabel('z','fontsize',12); ylabel('p(z|w2)','fontsize',12);
64 print -dpng 'LDA_c3_Xw2_pdf.png'
```

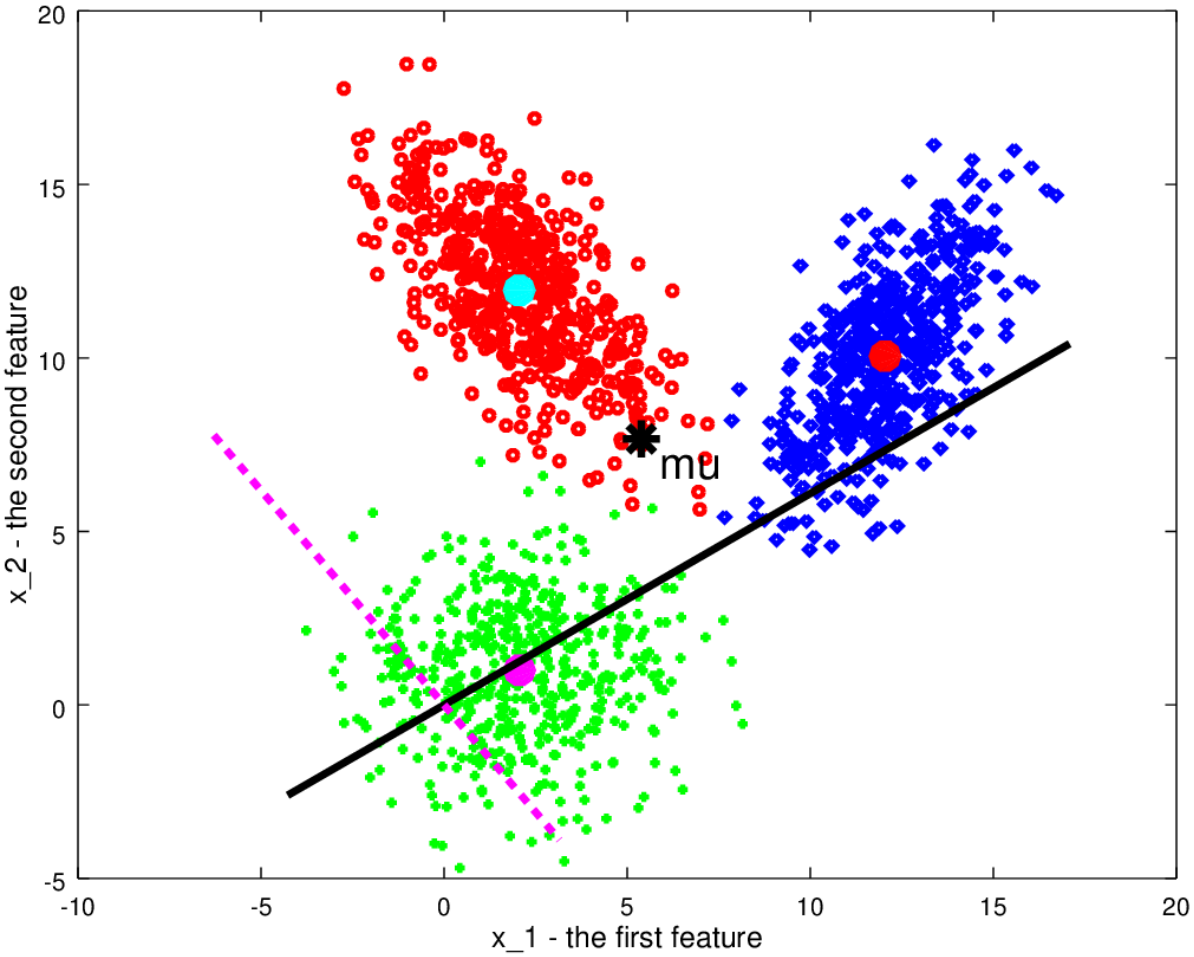


Figure 7.9: $\mathbf{w}_1^*$ (solid line in black) and $\mathbf{w}_2^*$ (dashed line in magenta).

- $\mathbf{w}_1^* = [0.85395, 0.52036]^T$, $\mathbf{w}_2^* = [-0.62899, 0.77742]^T$.
- Corresponding eigenvalues read

$$\lambda_1 = 3991.2, \quad \lambda_2 = 1727.7.$$

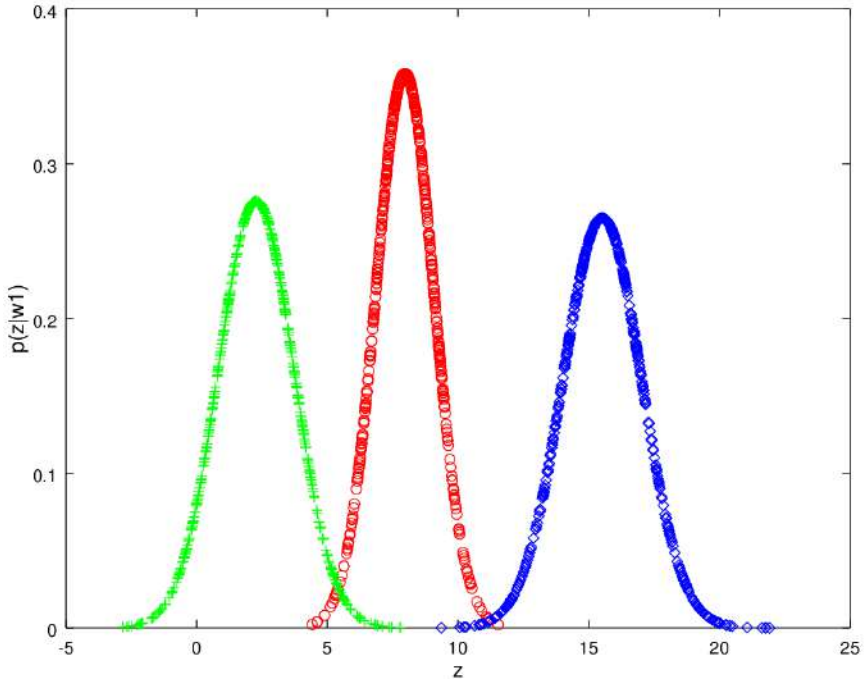


Figure 7.10: Classes PDF, along the first projection vector $\mathbf{w}_1^*$; $\lambda_1 = 3991.2$.

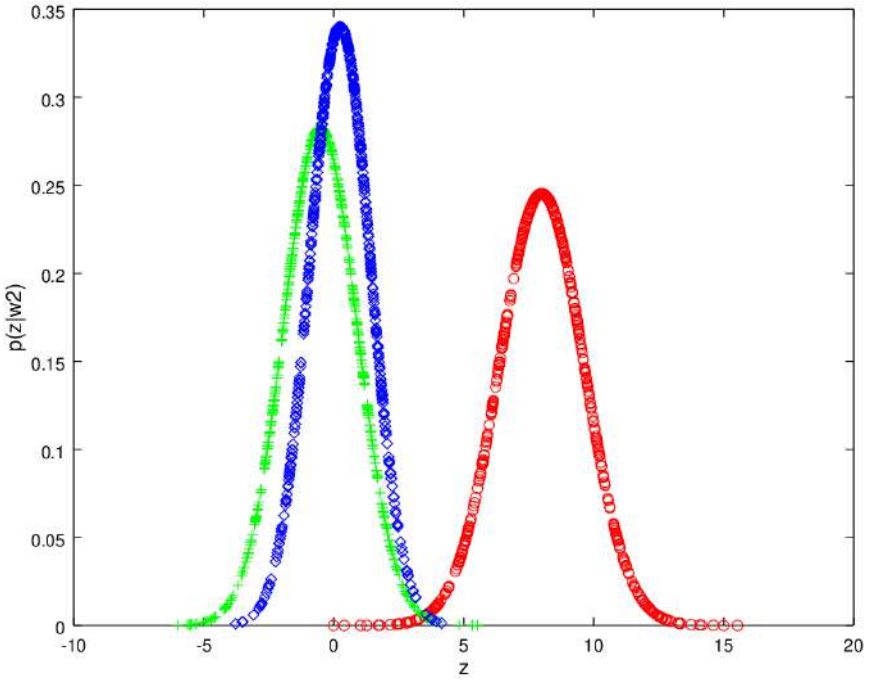


Figure 7.11: Classes PDF, along the second projection vector $\mathbf{w}_2^*$; $\lambda_2 = 1727.7$.

Apparently, the projection vector that has the highest eigenvalue provides higher discrimination power between classes.

7.3.4. The LDA: Dimensionality Reduction

Let $X \in \mathbb{R}^{N \times d}$ be the data matrix, in which each **row** represents a sample.

We summarize the main steps that are required to perform the LDA for dimensionality reduction.

1. **Standardize** the d-dimensional dataset (d is the number of features).
2. For each class j, compute the d-dimensional **mean vector** $\boldsymbol{\mu}_j$.
3. Construct the **within-class scatter matrix** S_w (7.69) and the **between-class scatter matrix** S_b (7.70).
4. Compute the **eigenvectors** and corresponding **eigenvalues** of the matrix $S_w^{-1} S_b$ (7.72).
5. Sort the eigenvalues by **decreasing order** to rank the corresponding eigenvectors.
6. Choose the k eigenvectors that correspond to the k largest eigenvalues to **construct a transformation matrix**

$$W = [\mathbf{w}_1 | \mathbf{w}_2 | \cdots | \mathbf{w}_k] \in \mathbb{R}^{d \times k}; \tag{7.75}$$

   the eigenvectors are the columns of this matrix.
7. **Project the samples** onto a new feature subspace: $X \to Z := XW$.

Remark 7.24.

- $\text{rank}(S_w^{-1} S_b) \leq c - 1$; we must have $k \leq c - 1$.
- The projected feature Z_{ij} is $\mathbf{x}^{(i)} \cdot \mathbf{w}_j$ in the projected coordinates and $(\mathbf{x}^{(i)} \cdot \mathbf{w}_j)\, \mathbf{w}_j$ in the original coordinates.

Limitations of the LDA (classifier) 

- The LDA produces **at most** $(c-1)$ **feature projections**.
  - If the classification error estimates establish that more features are needed, some other method must be employed to provide those additional features.
- The LDA is a parametric method, since it **assumes unimodal Gaussian likelihoods**.
  - If the distributions are significantly non-Gaussian, the LDA projections will not be able to preserve any complex structure of the data, which may be needed for classification.
- The LDA will **fail** when the discriminatory information is **not in the mean** but rather **in the variance of the data**.

LDA vs. PCA

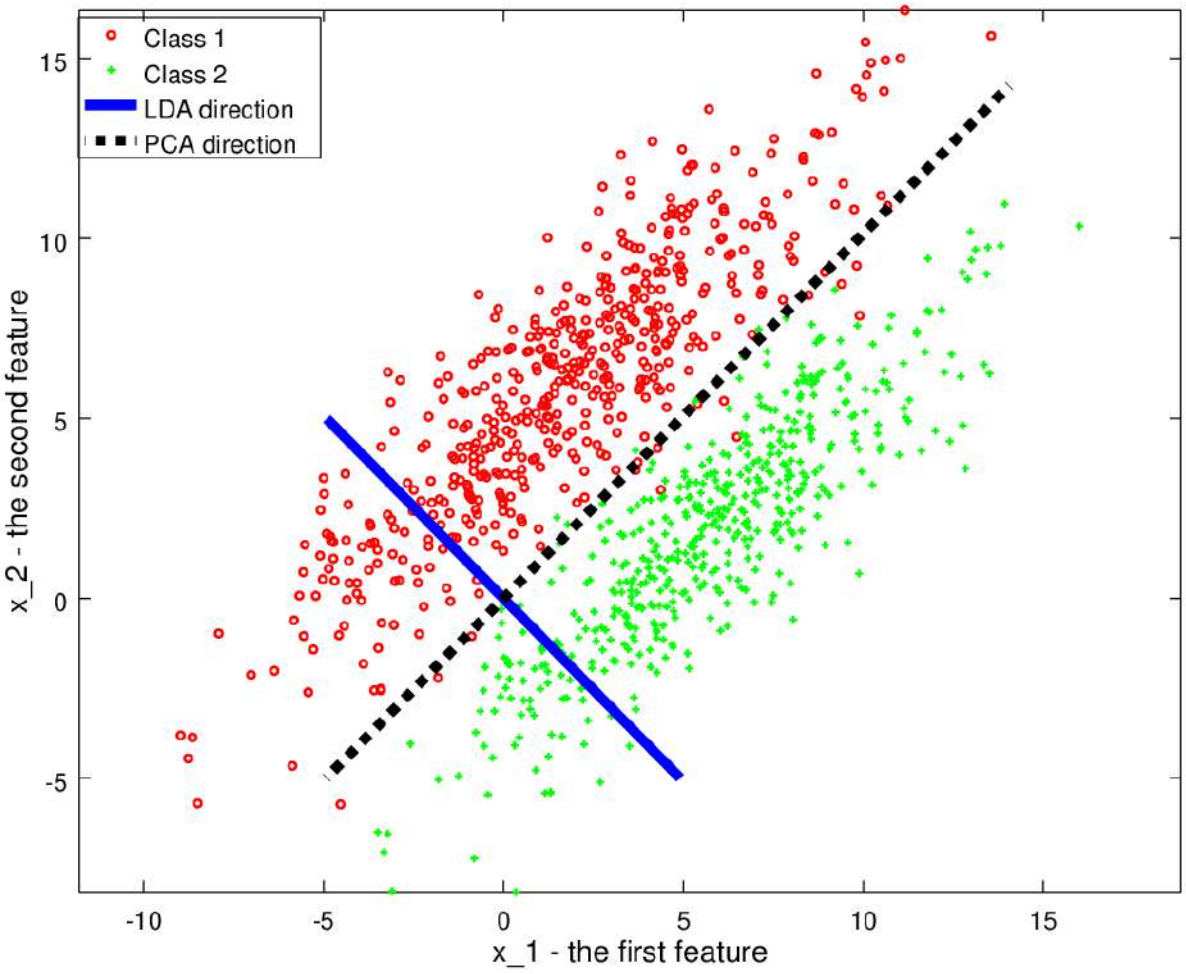


Figure 7.12: PCA vs. LDA.

- The (supervised) **LDA classifier must work better** than the (unsupervised) PCA, for datasets in Figures 7.9 and 7.12.

  Recall: Fisher's LDA was generalized under the assumption of **equal class covariances** and **normally distributed classes**.

- However, even if one or more of those assumptions are (slightly) violated, the LDA for dimensionality reduction can still work reasonably well.

7.4. Kernel Principal Component Analysis

The **kernel principal component analysis** (kernel PCA) [70] is **an extension of the PCA using kernel techniques** and performing the originally linear operations of the PCA in a **kernel Hilbert space**.

Recall: **(PCA)**. Consider a **data matrix** $X \in \mathbb{R}^{N\times d}$:
- each of the N rows represents a different data point,
- each of the d columns gives a particular kind of feature, and
- each column has zero empirical mean (e.g., after standardization).

- The goal of the standard PCA is to find an **orthogonal** weight matrix $W_k \in \mathbb{R}^{d\times k}$ such that
$$Z_k = X\,W_k, \quad k \le d, \tag{7.76}$$
where $Z_k \in \mathbb{R}^{N\times k}$ is call the **truncated score matrix** and $Z_d = Z$. Columns of Z represent the **principal components** of X.
- (Claim 7.3, p. 160). The transformation matrix W_k turns out to be the collection of normalized eigenvectors of X^TX:
$$W_k = [\mathbf{w}_1|\mathbf{w}_2|\cdots|\mathbf{w}_k], \quad (X^TX)\,\mathbf{w}_j = \lambda_j\,\mathbf{w}_j, \;\; \mathbf{w}_i^T\mathbf{w}_j = \delta_{ij}, \tag{7.77}$$
where $\lambda_1 \ge \lambda_2 \ge \cdots \ge \lambda_k \ge 0$.
- (Remark 7.4, p. 160). The matrix $Z_k \in \mathbb{R}^{N\times k}$ is scaled eigenvectors of XX^T:
$$Z_k = [\sqrt{\lambda_1}\,\mathbf{u}_1|\sqrt{\lambda_2}\,\mathbf{u}_2|\cdots|\sqrt{\lambda_k}\,\mathbf{u}_k], \quad (XX^T)\,\mathbf{u}_j = \lambda_j\,\mathbf{u}_j, \;\; \mathbf{u}_i^T\mathbf{u}_j = \delta_{ij}. \tag{7.78}$$
- A **data (row) vector** $\mathbf{x}$ **(new or old)** is transformed to a k-dimensional row vector of principal components
$$\mathbf{z} = \mathbf{x}W_k \in \mathbb{R}^{1\times k}. \tag{7.79}$$
- (Remark 7.5, p. 161). Let $\boldsymbol{X = U\,\Sigma V^T}$ be the **SVD** of X, where
$$\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \cdots, \sigma_d), \quad \sigma_1 \ge \sigma_2 \ge \cdots \ge \sigma_d \ge 0.$$
Then,
$$\begin{aligned} &V \cong W; \quad \sigma_j^2 = \lambda_j, \;\; j = 1, 2, \cdots, d, \\ &Z_k = [\sigma_1\,\mathbf{u}_1|\sigma_2\,\mathbf{u}_2|\cdots|\sigma_k\,\mathbf{u}_k]. \end{aligned} \tag{7.80}$$

7.4.1. Principal components of the kernel PCA

Note: Let $C = \frac{1}{N} X^T X$, the covariance matrix of X. Then,

$$C = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}^{(i)} \mathbf{x}^{(i)^T} \in \mathbb{R}^{d \times d}, \quad C_{jk} = \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} x_k^{(i)}. \tag{7.81}$$

Here, we consider $\mathbf{x}^{(i)}$ as a column vector (when standing alone), while it lies in X as a row.

- The kernel PCA is a generalization of the PCA, where the dataset X is **transformed into a higher dimensional space** (by creating non-linear combinations of the original features):

$$\phi : X \in \mathbb{R}^{N \times d} \to \phi(X) \in \mathbb{R}^{N \times p}, \quad d < p, \tag{7.82}$$

  and the **covariance matrix** is computed via outer products between such expanded samples:

$$C = \frac{1}{N} \sum_{i=1}^{N} \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T = \frac{1}{N} \phi(X)^T \phi(X) \in \mathbb{R}^{p \times p}. \tag{7.83}$$

- To obtain the eigenvectors – ***the principal components*** – from the covariance matrix, we should solve the eigenvalue problem:

$$C\mathbf{v} = \lambda \mathbf{v}. \tag{7.84}$$

- Assume (7.84) is solved.
  - Let, for $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k \geq \cdots \geq \lambda_p \geq 0$,

$$V_k = [\mathbf{v}_1 | \mathbf{v}_2 | \cdots | \mathbf{v}_k] \in \mathbb{R}^{p \times k}, \quad C\mathbf{v}_j = \lambda_j \mathbf{v}_j, \ \ \mathbf{v}_i^T \mathbf{v}_j = \delta_{ij}. \tag{7.85}$$

  - Then, the **score matrix Z_k (principal components)** for the kernel PCA reads

$$Z_k = \phi(X) V_k \in \mathbb{R}^{N \times k}, \tag{7.86}$$

    which is an analogue to (7.76).

However, it is computationally expensive or impossible to solve the eigenvalue problem (7.84), when p is large or infinity.

An Alternative to the Computation of the Score Matrix

Claim 7.25. Let $\mathbf{v}$ be an eigenvector of C as in (7.84). Then it can be expressed as linear combination of data points:

$$\mathbf{v} = \sum_{i=1}^{N} \alpha_i \, \phi(\mathbf{x}^{(i)}). \tag{7.87}$$

Proof. Since $C\mathbf{v} = \lambda\mathbf{v}$, we get

$$\frac{1}{N} \sum_{i=1}^{N} \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T \mathbf{v} = \lambda \mathbf{v}$$

and therefore

$$\mathbf{v} = \frac{1}{\lambda N} \sum_{i=1}^{N} \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T \mathbf{v} = \frac{1}{\lambda N} \sum_{i=1}^{N} [\phi(\mathbf{x}^{(i)}) \cdot \mathbf{v}] \, \phi(\mathbf{x}^{(i)}), \tag{7.88}$$

where $\phi(\mathbf{x}^{(i)}) \cdot \mathbf{v}$ is a scalar and $\alpha_i := (\phi(\mathbf{x}^{(i)}) \cdot \mathbf{v})/(\lambda N)$. □

Note:

- The above claim means that all eigenvectors $\mathbf{v}$ with $\lambda \neq 0$ lie in the span of $\phi(\mathbf{x}^{(1)}), \cdots, \phi(\mathbf{x}^{(N)})$.
- Thus, finding the eigenvectors in (7.84) is equivalent to finding the coefficients $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \cdots, \alpha_N)^T$.

How to find α

- Let $C\mathbf{v}_j = \lambda_j \mathbf{v}_j$ with $\lambda_j \neq 0$. Then, (7.87) can be written as

$$\mathbf{v}_j = \sum_{\ell=1}^{N} \alpha_{\ell j} \phi(\mathbf{x}^{(\ell)}) = \phi(X)^T \boldsymbol{\alpha}_j. \tag{7.89}$$

- By substituting this back into the equation and using (7.83), we get

$$C\mathbf{v}_j = \lambda_j \mathbf{v}_j \;\Rightarrow\; \frac{1}{N}\phi(X)^T\phi(X)\phi(X)^T\boldsymbol{\alpha}_j = \lambda_j\, \phi(X)^T\boldsymbol{\alpha}_j. \tag{7.90}$$

and therefore

$$\frac{1}{N}\phi(X)\phi(X)^T\phi(X)\phi(X)^T\boldsymbol{\alpha}_j = \lambda_j\, \phi(X)\phi(X)^T\boldsymbol{\alpha}_j. \tag{7.91}$$

- Let K be the **similarity (kernel) matrix**:

$$K \overset{\text{def}}{=\!=} \phi(X)\phi(X)^T \in \mathbb{R}^{N\times N}. \tag{7.92}$$

- Then, (7.91) can be rewritten as

$$K^2\boldsymbol{\alpha}_j = (N\lambda_j)\, K\boldsymbol{\alpha}_j. \tag{7.93}$$

- We can remove a factor of K from both sides of the above equation:[a]

$$\boxed{K\boldsymbol{\alpha}_j = \mu_j\boldsymbol{\alpha}_j}, \quad \mu_j = N\lambda_j. \tag{7.94}$$

which implies that $\boldsymbol{\alpha}_j$ **are eigenvectors of** K.

- It should be noticed that $\boldsymbol{\alpha}_j$ **are analogues of** $\mathbf{u}_j$**, where** $\boldsymbol{X = U\Sigma V^T}$**.**

[a] This will only affects the eigenvectors with zero eigenvalues, which will not be a principle component anyway.

Note: There is a **normalization condition** for the $\boldsymbol{\alpha}_j$ vectors:

$$\|\mathbf{v}_j\| = 1 \iff \|\boldsymbol{\alpha}_j\| = 1/\sqrt{\mu_j}.$$

$$\because \left[\begin{array}{rcll} 1 &=& \mathbf{v}_j^T\mathbf{v}_j = (\phi(X)^T\boldsymbol{\alpha}_j)^T\phi(X)^T\boldsymbol{\alpha}_j = \boldsymbol{\alpha}_j^T\phi(X)\phi(X)^T\boldsymbol{\alpha}_j & \Leftarrow (7.89) \\ &=& \boldsymbol{\alpha}_j^T K\boldsymbol{\alpha}_j = \boldsymbol{\alpha}_j^T(\mu_j\boldsymbol{\alpha}_j) = \mu_j\,\|\boldsymbol{\alpha}_j\|^2 & \Leftarrow (7.94) \end{array}\right. \tag{7.95}$$

7.4.2. Computation of the kernel PCA

Remark 7.26. Let the eigenvalue-eigenvector pairs of the kernel matrix K be given as

$$K\boldsymbol{\alpha}_j = \mu_j \boldsymbol{\alpha}_j, \quad j = 1, 2, \cdots, N; \qquad \boldsymbol{\alpha}_i^T \boldsymbol{\alpha}_j = \delta_{ij}. \tag{7.96}$$

- Then, referring (7.78) derived for the standard PCA, we may conclude that the ***k* principal components for the kernel PCA** are

$$\mathcal{A}_k = [\sqrt{\mu_1}\boldsymbol{\alpha}_1 | \sqrt{\mu_2}\boldsymbol{\alpha}_2 | \cdots | \sqrt{\mu_k}\boldsymbol{\alpha}_k] \in \mathbb{R}^{N\times k}. \tag{7.97}$$

- It follows from (7.86), (7.89), and (7.95)-(7.96) that for a **new point** $\mathbf{x}$, its projection onto the principal components is:

$$\begin{aligned} z_j = \phi(\mathbf{x})^T \mathbf{v}_j &= \frac{1}{\sqrt{\mu_j}} \phi(\mathbf{x})^T \sum_{\ell=1}^{N} \alpha_{\ell j} \phi(\mathbf{x}^{(\ell)}) = \frac{1}{\sqrt{\mu_j}} \sum_{\ell=1}^{N} \alpha_{\ell j} \phi(\mathbf{x})^T \phi(\mathbf{x}^{(\ell)}) \\ &= \frac{1}{\sqrt{\mu_j}} \sum_{\ell=1}^{N} \alpha_{\ell j} \mathcal{K}(\mathbf{x}, \mathbf{x}^{(\ell)}) = \frac{1}{\sqrt{\mu_j}} \mathcal{K}(\mathbf{x}, X)^T \boldsymbol{\alpha}_j. \end{aligned} \tag{7.98}$$

That is, due to (7.95) and (7.96), when $\mathbf{v}_j$ is expressed in terms of $\boldsymbol{\alpha}_j$, it must be scaled by $1/\sqrt{\mu_j}$.

Construction of the kernel matrix *K*

- The **kernel trick** is to avoid calculating the pairwise dot products of the transformed samples $\phi(\mathbf{x})$ explicitly by using a kernel function.
- For a selected kernel function $\mathcal{K}$,

$$K = \begin{bmatrix} \mathcal{K}(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}) & \mathcal{K}(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) & \cdots & \mathcal{K}(\mathbf{x}^{(1)}, \mathbf{x}^{(N)}) \\ \mathcal{K}(\mathbf{x}^{(2)}, \mathbf{x}^{(1)}) & \mathcal{K}(\mathbf{x}^{(2)}, \mathbf{x}^{(2)}) & \cdots & \mathcal{K}(\mathbf{x}^{(2)}, \mathbf{x}^{(N)}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{K}(\mathbf{x}^{(N)}, \mathbf{x}^{(1)}) & \mathcal{K}(\mathbf{x}^{(N)}, \mathbf{x}^{(2)}) & \cdots & \mathcal{K}(\mathbf{x}^{(N)}, \mathbf{x}^{(N)}) \end{bmatrix} \in \mathbb{R}^{N\times N}. \tag{7.99}$$

where $\mathcal{K}$ is called the **kernel function**.[a]

[a] As for nonlinear SVM, the most commonly used kernels are the polynomial kernel, the hyperbolic tangent (sigmoid) kernel, and the Gaussian Radial Basis Function (RBF) kernel. See (5.57)-(5.60), p. 126.

Normalizing the feature space

- In general, $\phi(\mathbf{x}^{(i)})$ may not be zero mean.
- Thus $K = \phi(X)\phi(X)^T$ would better be normalized before start finding its eigenvectors and eigenvalues.
- Centered features:

$$\widetilde{\phi}(\mathbf{x}^{(i)}) = \phi(\mathbf{x}^{(i)}) - \frac{1}{N}\sum_{k=1}^{N}\phi(\mathbf{x}^{(k)}), \quad \forall\, i. \tag{7.100}$$

- The corresponding kernel is

$$\begin{aligned}
\widetilde{\mathcal{K}}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) &= \widetilde{\phi}(\mathbf{x}^{(i)})^T\widetilde{\phi}(\mathbf{x}^{(j)}) \\
&= \left(\phi(\mathbf{x}^{(i)}) - \frac{1}{N}\sum_{k=1}^{N}\phi(\mathbf{x}^{(k)})\right)^T\left(\phi(\mathbf{x}^{(j)}) - \frac{1}{N}\sum_{k=1}^{N}\phi(\mathbf{x}^{(k)})\right) \\
&= \mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) - \frac{1}{N}\sum_{k=1}^{N}\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(k)}) - \frac{1}{N}\sum_{k=1}^{N}\mathcal{K}(\mathbf{x}^{(k)}, \mathbf{x}^{(j)}) \\
&\quad + \frac{1}{N^2}\sum_{k,\ell=1}^{N}\mathcal{K}(\mathbf{x}^{(k)}, \mathbf{x}^{(\ell)}).
\end{aligned} \tag{7.101}$$

- In a matrix form

$$\widetilde{K} = K - K\mathbf{1}_{1/N} - \mathbf{1}_{1/N}K + \mathbf{1}_{1/N}K\mathbf{1}_{1/N}, \tag{7.102}$$

where $\mathbf{1}_{1/N}$ is an $N \times N$ matrix where all entries are equal to $1/N$.

Summary **7.27. (Summary of the Kernel PCA).**

- Pick a kernel function $\mathcal{K}$.
- For data $X \in \mathbb{R}^{N\times d}$, construct the kernel matrix

$$K = [\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})] \in \mathbb{R}^{N\times N}. \tag{7.103}$$

- Normalize the kernel matrix K:

$$\widetilde{K} = K - K\mathbf{1}_{1/N} - \mathbf{1}_{1/N}K + \mathbf{1}_{1/N}K\mathbf{1}_{1/N}. \tag{7.104}$$

- Solve an eigenvalue problem:

$$\widetilde{K}\boldsymbol{\alpha}_j = \mu_j\boldsymbol{\alpha}_j, \quad \boldsymbol{\alpha}_i^T\boldsymbol{\alpha}_j = \delta_{ij}. \tag{7.105}$$

- Then, the ***k* principal components for the kernel PCA** are

$$\mathcal{A}_k = [\mu_1\boldsymbol{\alpha}_1|\mu_2\boldsymbol{\alpha}_2|\cdots|\mu_k\boldsymbol{\alpha}_k] \in \mathbb{R}^{N\times k}, \quad k \le N. \tag{7.106}$$

- For **a data point x (new or old)**, we can represent it as

$$z_j = \phi(\mathbf{x})^T\mathbf{v}_j = \phi(\mathbf{x})^T\sum_{\ell=1}^{N}\alpha_{\ell j}\phi(\mathbf{x}^{(\ell)}) = \sum_{\ell=1}^{N}\alpha_{\ell j}\mathcal{K}(\mathbf{x}, \mathbf{x}^{(\ell)}), \quad j = 1, 2, \cdots, k. \tag{7.107}$$

Note: Formulas in (7.106)-(7.107) are alternatives of (7.97)-(7.98).

Properties of the KPCA

- With an appropriate choice of kernel function, the kernel PCA can give a good re-encoding of the data that lies along a nonlinear manifold.
- The kernel matrix is in $(N \times N)$-dimensions, so the kernel PCA will have difficulties when we have lots of data points.

Exercises for Chapter 7

7.1. Read pp. 145–158, *Python Machine Learning, 3rd Ed.*, about the PCA.

(a) Find the optimal number of components k^* which produces the best classification accuracy (for logistic regression), by experimenting the example code with `n_components` $= 1, 2, \cdots, 13$.

(b) What is the corresponding **cumulative explained variance**?

7.2. Let $A \in \mathbb{R}^{m\times n}$. Prove that $||A||_2 = \sigma_1$, the largest singular value of A. **Hint**: Use the following

$$\frac{||A\mathbf{v}_1||_2}{||\mathbf{v}_1||_2} = \frac{\sigma_1||\mathbf{u}_1||_2}{||\mathbf{v}_1||_2} = \sigma_1 \Longrightarrow ||A||_2 \geq \sigma_1$$

and arguments around Equations (7.34) and (7.35) for the opposite directional inequality.

7.3. Recall that the Frobenius matrix norm is defined by

$$||A||_F = \Big(\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2\Big)^{1/2}, \quad A \in \mathbb{R}^{m\times n}.$$

Show that $||A||_F = (\sigma_1^2 + \cdots + \sigma_k^2)^{1/2}$, where σ_j are nonzero singular values of A. **Hint**: You may use the norm-preserving property of orthogonal matrices. That is, if U is orthogonal, then $||UB||_2 = ||B||_2$ and $||UB||_F = ||B||_F$.

7.4. Prove Lemma 7.18. **Hint**: For (b), let $A\mathbf{v}_i = \lambda_i\mathbf{v}_i$, $i = 1, 2$, and $\lambda_1 \neq \lambda_2$. Then

$$(\lambda_1\mathbf{v}_1)\cdot\mathbf{v}_2 = \underbrace{(A\mathbf{v}_1)\cdot\mathbf{v}_2 = \mathbf{v}_1\cdot(A\mathbf{v}_2)}_{\because\ A \text{ is symmetric}} = \mathbf{v}_1\cdot(\lambda_2\mathbf{v}_2).$$

For (a), you may use a similar argument, but with the dot product being defined for complex values, i.e.,

$$\mathbf{u}\cdot\mathbf{v} = \mathbf{u}^T\,\overline{\mathbf{v}},$$

where $\overline{\mathbf{v}}$ is the complex conjugate of $\mathbf{v}$.

7.5. Use Matlab to generate a random matrix $A \in \mathbb{R}^{8\times 6}$ with rank 4. For example,

```
A = randn(8,4);
A(:,5:6) = A(:,1:2)+A(:,3:4);
[Q,R] = qr(randn(6));
A = A*Q;
```

(a) Print out A on your computer screen. Can you tell by looking if it has (numerical) rank 4?

(b) Use Matlab's "svd" command to obtain the singular values of A. How many are "large?" How many are "tiny?" (You may use the command "format short e" to get a more accurate view of the singular values.)

(c) Use Matlab's "rank" command to confirm that the numerical rank is 4.

(d) Use the “rank" command with a small enough threshold that it returns the value 6. (Type “help rank" for information about how to do this.)

7.6. Verify (7.64). ***Hint*:** *Use the quotient rule for* $\frac{\partial \mathcal{J}(\mathbf{w})}{\partial \mathbf{w}}$ *and equate the numerator to zero.*

7.7. Try to understand the kernel PCA more deeply by experimenting pp. 175–188, *Python Machine Learning, 3rd Ed.*. Its implementation is slightly different from (but equivalent to) Summary 7.27.

(a) Modify the code, following Summary 7.27, and test if it works as expected as in *Python Machine Learning, 3rd Ed.*.

(b) The datasets considered are transformed via the Gaussian radial basis function (RBF) kernel only. What happens if you use the following kernels?

$$\mathcal{K}_1(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = (a_1 + b_1 \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)})^2 \qquad \text{(polynomial of degree up to 2)}$$
$$\mathcal{K}_2(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \tanh(a_2 + b_2 \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}) \qquad \text{(sigmoid)}$$

Can you find a_i and b_i, $i = 1, 2$, appropriately?

CHAPTER 8

Cluster Analysis

Cluster analysis or **clustering** is **the task of finding groups of objects** such that the objects in a group will be similar (or related) to one another and different from (or unrelated to) the objects in other groups. It is a main task of **exploratory data mining**, and a common technique for **statistical data analysis**, used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, bioinformatics, data compression, and computer graphics.

History. Cluster analysis was originated in **anthropology** by Driver and Kroeber in 1932 [16], introduced to **psychology** by Zubin in 1938 [84] and Robert Tryon in 1939 [76], and famously used by Cattell beginning in 1943 [11] for trait theory classification in **personality psychology**.

Contents of Chapter 8

8.1. Basics for Cluster Analysis . . . 208
8.2. K-Means and K-Medoids Clustering . . . 219
8.3. Hierarchical Clustering . . . 232
8.4. DBSCAN: Density-based Clustering . . . 239
8.5. Cluster Validation . . . 244
8.6. Self-Organizing Maps . . . 255
Exercises for Chapter 8 . . . 268

8.1. Basics for Cluster Analysis

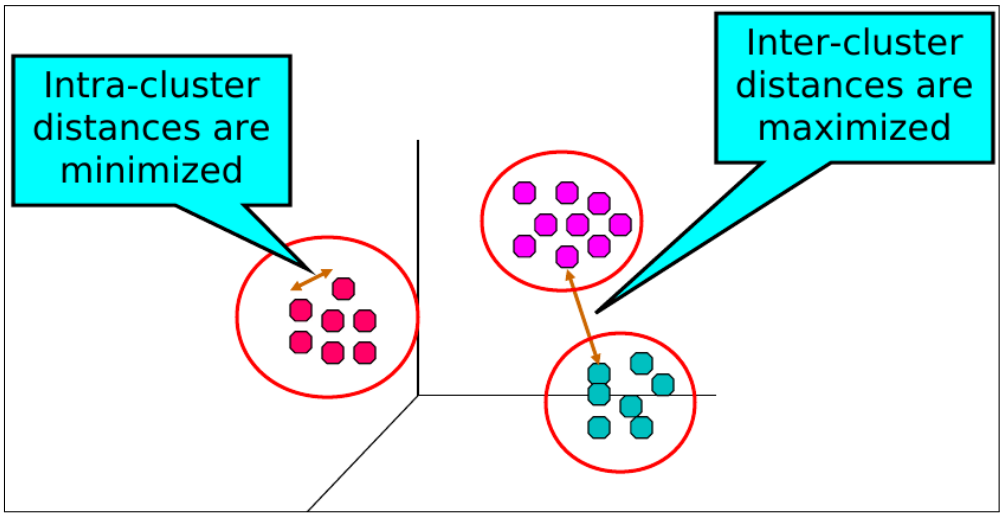


Figure 8.1: Intra-cluster distance vs. inter-cluster distance.

Applications of Cluster Analysis

- **Understanding**
  - group related documents or browsings
  - group genes/proteins that have similar functionality, or
  - group stocks with similar price fluctuations
- **Summarization**
  - reduce the size of large data sets

Not Cluster Analysis

- Supervised classification – Uses class label information
- Simple segmentation – Dividing students into different registration groups alphabetically, by last name
- Results of a query – Groupings are a result of an external specification

Clustering uses ***only the data*** (**unsupervised learning**):
to discover hidden structures in data

8.1.1. Quality of clustering

- A **good clustering** method will produce high quality clusters with
  - **high intra-class similarity**
  - **low inter-class similarity**
- The quality of a clustering result depends on both **the similarity measure** and **its implementation**
- The quality of a clustering method is also measured by its ability to discover some or all of the **hidden patterns**

Measuring the Quality of Clustering

- **Dissimilarity/Similarity/Proximity metric**: Similarity is expressed in terms of a distance function $d(i, j)$
- The definitions of **distance functions** are usually very different for interval-scaled, boolean, categorical, ordinal ratio, and vector variables.
- There is a separate "quality" function that measures the "goodness" of a cluster.
- *Weighted measures*: Weights should be associated with different variables based on applications and data semantics.
- It is hard to define "**similar enough**" or "**good enough**"
  - **the answer is typically highly subjective**

Notion of a Cluster can be Ambiguous

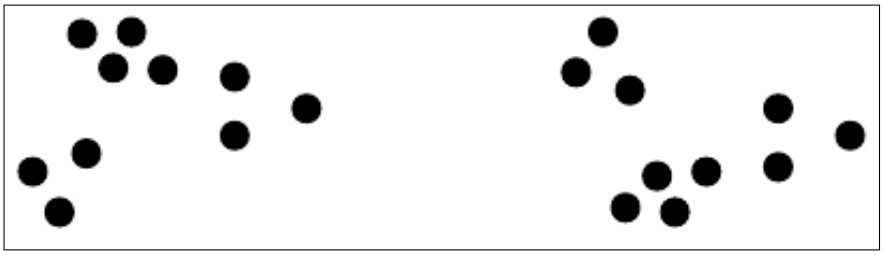


Figure 8.2: **How many clusters?**

The answer could be:

Figure 8.3: Two clusters.

Figure 8.4: Four clusters.

Figure 8.5: Six clusters.

Similarity and Dissimilarity Between Objects

- **Distances** are normally used to measure the similarity or dissimilarity between two data objects
- Some popular ones include: **Minkowski distance**

$$d(i,j) = \|\mathbf{x}_i - \mathbf{x}_j\|_p = (|x_{i1} - x_{j1}|^p + |x_{i2} - x_{j2}|^p \cdots |x_{id} - x_{jd}|^p)^{1/p}, \quad (8.1)$$

  where $\mathbf{x}_i$, $\mathbf{x}_j \in \mathbb{R}^d$, two d-dimensional data objects.

  - When $p = 1$, it is **Manhattan distance**
  - When $p = 2$, it is **Euclidean distance**
- *Other Distances*: Also, one can use weighted distance, parametric Pearson product moment correlation, or other dissimilarity measures
- Various similarity measures have been studied for
  - Binary variables
  - Nominal variables & ordinal variables
  - Ratio-scaled variables
  - Variables of mixed types
  - Vector objects

8.1.2. Types of clusters

- Center-based clusters
- Contiguity/connectivity-based clusters
- Density-based clusters
- Conceptual clusters

Note: **(Well-separated clusters)**. A cluster is a set of objects such that an object in a cluster is closer (more similar) to **every/some of points** in the cluster, than any points not in the cluster.

Center-based Clusters

- The center of a cluster is often
  - a **centroid**, the average of all the points in the cluster, or
  - a **medoid**, the most representative point of a cluster.
- *A cluster is a set of objects such that an object in a cluster is closer (more similar) to **the "center" of a cluster**, than to the center of any other clusters.*

Figure 8.6: Well-separated, 4 center-based clusters.

Contiguity-based Clusters

- Contiguous cluster (nearest neighbor or transitive)
- *A cluster is a set of points such that a point in a cluster is closer (or more similar) to* ***one or more other points*** *in the cluster, than to any points not in the cluster.*

Figure 8.7: 8 contiguous clusters.

Density-based Clusters

- *A cluster is a dense region of points, which is separated by low-density regions, from other regions of high density.*
- Used when the clusters are irregular or intertwined, and when noise and outliers are present.

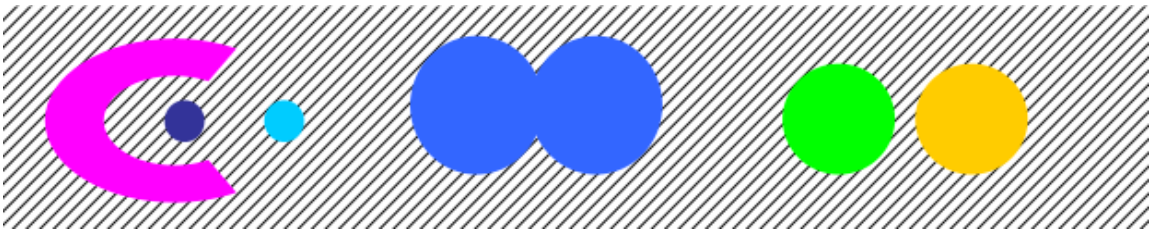


Figure 8.8: 6 density-based clusters.

Conceptual Clusters

- Points in a cluster share some general property.
  - Conceptual clusters are hard to detect, because they are often none of the center-based, contiguity-based, or density-based.
  - Points in the intersection of the circles belong to both.

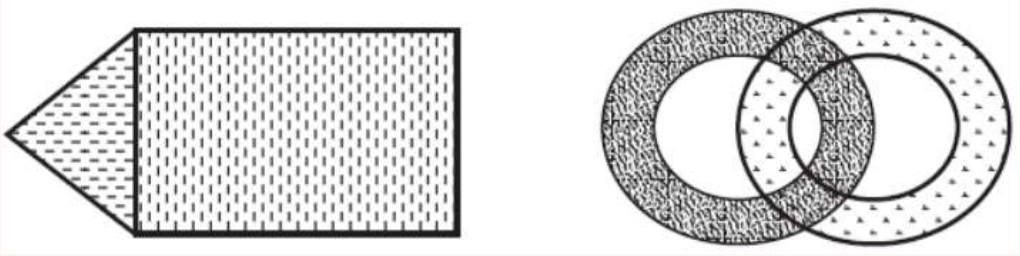


Figure 8.9: Conceptual clusters

Clusters Defined by an Objective Function

- Find clusters that minimize or maximize an objective function.
- Enumerate all possible ways of dividing the points into clusters and evaluate the "goodness" of each potential set of clusters by using the given objective function. (**NP-Hard**)
- Can have global or local objectives. *Typically*,
  - **Partitional clustering algorithms** have global objectives
  - **Hierarchical clustering algorithms** have local objectives

Computational Complexity Theory

Problem Types

- **P (Polynomial Time)**: Problems which are **solvable in polynomial time** (when running on a deterministic Turing machine[a]).
- **NP (Non-deterministic Polynomial Time)**: Decision problems which can be **verified in polynomial time**.
- **NP-Hard**: These are at least as hard as the hardest problems in NP, **in both solution and verification**.
- **NP-Complete**: These are the problems which are both NP and NP-Hard.

[a] A **Turing machine** is a theoretical machine that manipulates symbols on a strip of tape according to a table of rules. A deterministic Turing machine is a theoretical machine, used in thought experiments to examine the abilities and limitations of algorithms. In a deterministic Turing machine, the set of rules impose at most one action to be performed for any given situation. In a nondeterministic Turing machine, it may have a set of rules that prescribes more than one action for a given situation [13].

Problem Type	Verifiable in P-time	Solvable in P-time
P	Yes	Yes
NP	Yes	Yes or No
NP-Complete	Yes	Unknown
NP-Hard	Yes or No	Unknown

Quesiton. **P = NP?** (**P versus NP problem**)

- This one is the most famous problem in computer science, and one of the most important outstanding questions in the mathematical sciences.
- In fact, the **Clay Institute** is offering one million dollars for a solution to the problem.
  - It's clear that P is a subset of NP.
  - The open question is whether or not NP problems have deterministic polynomial time solutions.

8.1.3. Types of clustering and Objective functions

- Partitional clustering
- Hierarchical clustering (agglomerative; divisive)
- Density-based clustering (DBSCAN)

Partitional Clustering

Divide data objects into **non-overlapping subsets** (clusters) such that each data object is in exactly one subset

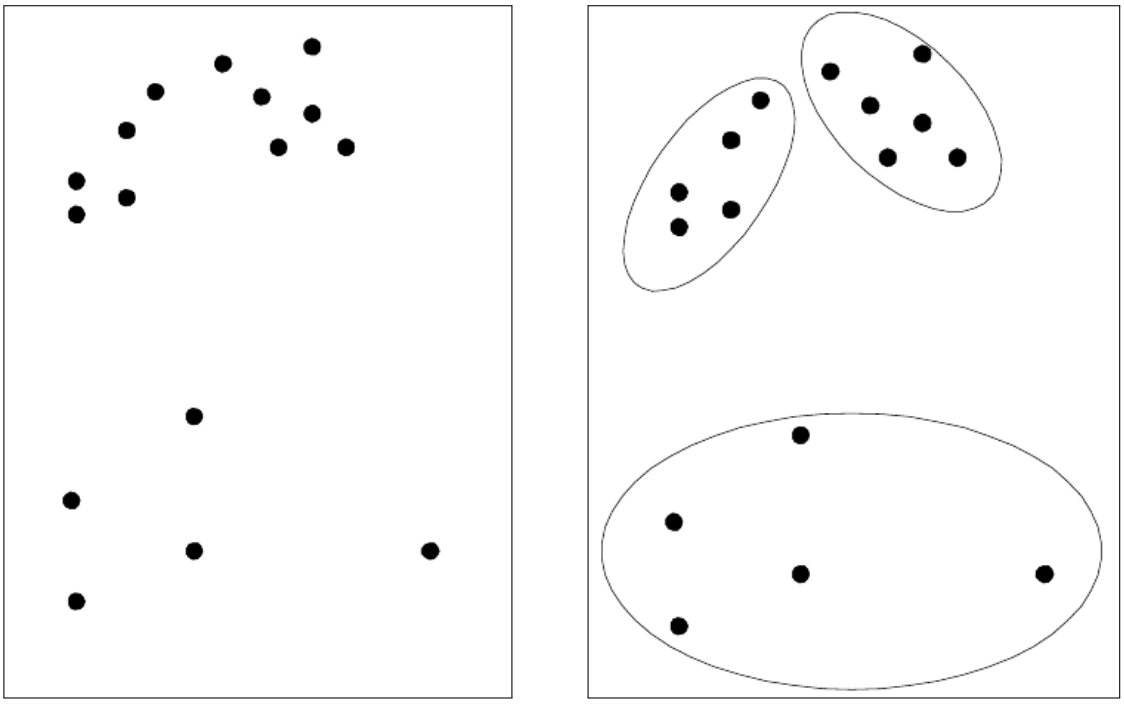


Figure 8.10: Original points & A partitional clustering

Examples are

- K-Means, Bisecting K-Means
- K-Medoids (PAM: partitioning around medoids)
- CLARA, CLARANS (Sampling-based PAMs)

Hierarchical Clustering

A set of **nested clusters**, organized as a hierarchical tree

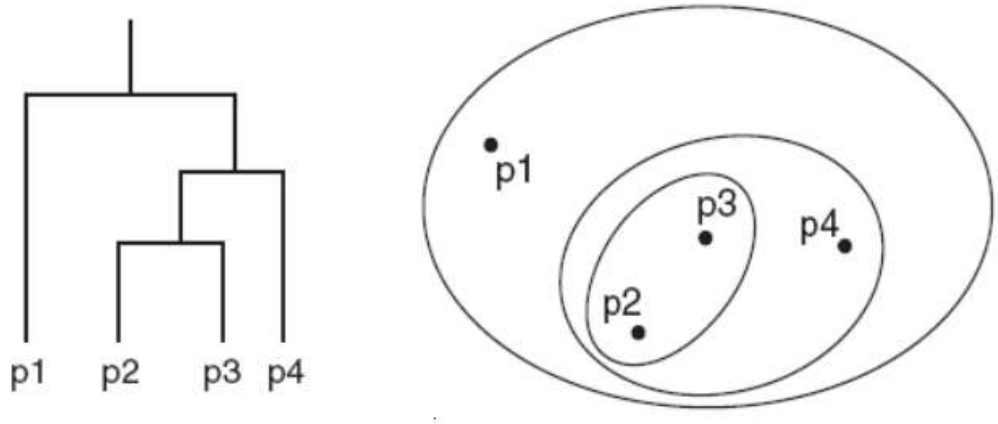


Figure 8.11: Dendrogram and Nested cluster diagram.

Other Distinctions Between Sets of Clusters

- **Exclusive (hard) vs. Non-exclusive (soft)**
  - In non-exclusive clusterings, points may belong to multiple clusters.
- **Fuzzy vs. Non-fuzzy**
  - In fuzzy clustering, a point belongs to every cluster with some **membership weight** between 0 and 1
  - Membership weights must sum to 1
  - **Probabilistic clustering** has similar characteristics
- **Partial vs. Complete**
  - In some cases, we only want to cluster some of the data
- **Homogeneous vs. Heterogeneous**
  - Cluster of widely different sizes, shapes, and densities

Objective Functions

Global objective function

- Typically used in partitional clustering
  - K-Means minimizes the **Sum of Squared Errors** (SSE):

$$SSE = \sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|_2^2, \tag{8.2}$$

    where x is a data point in cluster C_i and $\boldsymbol{\mu}_i$ is the center for cluster C_i as the mean of all points in the cluster.

- **Mixture models**: assume that the dataset is a "mixture" of a number of parametric statistical distributions
  (e.g., Gaussian mixture models).

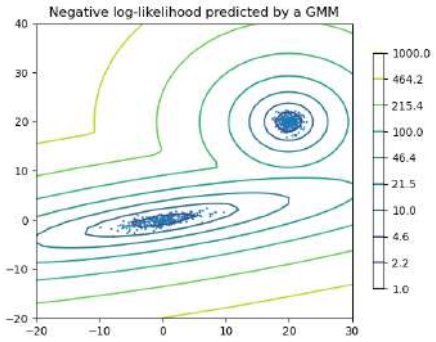


Figure 8.12: A two-component Gaussian mixture model: data points, and equi-probability surfaces of the model.

Local objective function

- **Hierarchical clustering** algorithms typically have local objectives
- **Density-based clustering** is based on local density estimates
- **Graph based approaches**: Graph partitioning and shared nearest neighbors

We will consider the objective functions when we talk about individual clustering algorithms.

8.2. K-Means and K-Medoids Clustering

- Given
  - X, a dataset of N objects
  - K, the number of clusters to form
- Organize the objects into K partitions ($K \leq N$), where each partition represents a cluster
- The clusters are formed to optimize an **objective partitioning criterion**:
  - Objects within a cluster are similar
  - Objects of different clusters are dissimilar

8.2.1. The (basic) K-Means clustering

- **Partitional clustering** approach
- Each cluster is associated with a **centroid** (mean)
- Each point is assigned to the cluster **with the closest centroid**
- Number of clusters, ***K***, must be specified

Algorithm **8.1. Lloyd's algorithm** (a.k.a. **Voronoi iteration**): (Lloyd, 1957) [48]

1. Select K points as the initial centroids;
2. **repeat**
3. Form K clusters by assigning all points to the closest centroid;
4. Recompute the centroid of each cluster;
5. **until** (the centroids don't change)

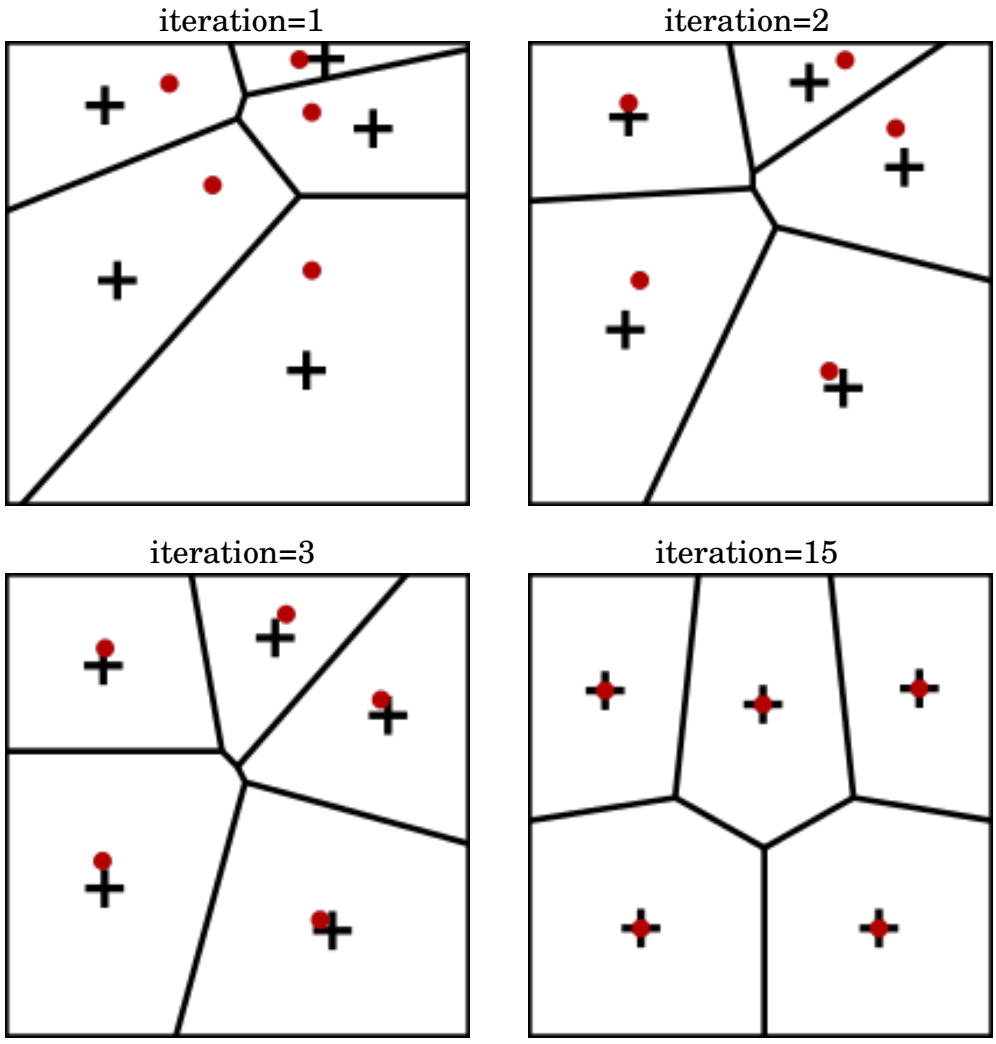


Figure 8.13: Lloyd's algorithm: The Voronoi diagrams, the given centroids (●), and the updated centroids (┼), for iteration = 1, 2, 3, and 15.

The K-Means Clustering – Details

- Initial centroids are often chosen randomly.
  - Clusters produced vary from one run to another.
- The centroid is (typically) the mean of the points in the cluster.
- "Closeness" is measured by **Euclidean distance**, cosine similarity, correlation, etc..
- The K-Means will converge typically in the first few iterations.
  - Often the stopping condition is changed to "**until** (relatively few points change clusters)" or some measure of clustering doesn't change.
- Complexity is $\mathcal{O}(N * d * K * I)$, where
  N: the number of points
  d: the number of attributes
  K: the number of clusters
  I: the number of iterations

Evaluating the K-Means Clusters

- Most common measure is Sum of Squared Error (SSE):

$$SSE = \sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|_2^2, \tag{8.3}$$

  where x is a data point in cluster C_i and $\boldsymbol{\mu}_i$ is the center for cluster C_i.
- **Multiple runs**: Given sets of clusters, we can choose the one with the smallest error.
- One easy way to reduce SSE is to increase K, the number of clusters.
  - A good clustering with smaller K can have a lower SSE than a poor clustering with higher K.

The K-Means is **heuristic** to minimize the SSE.

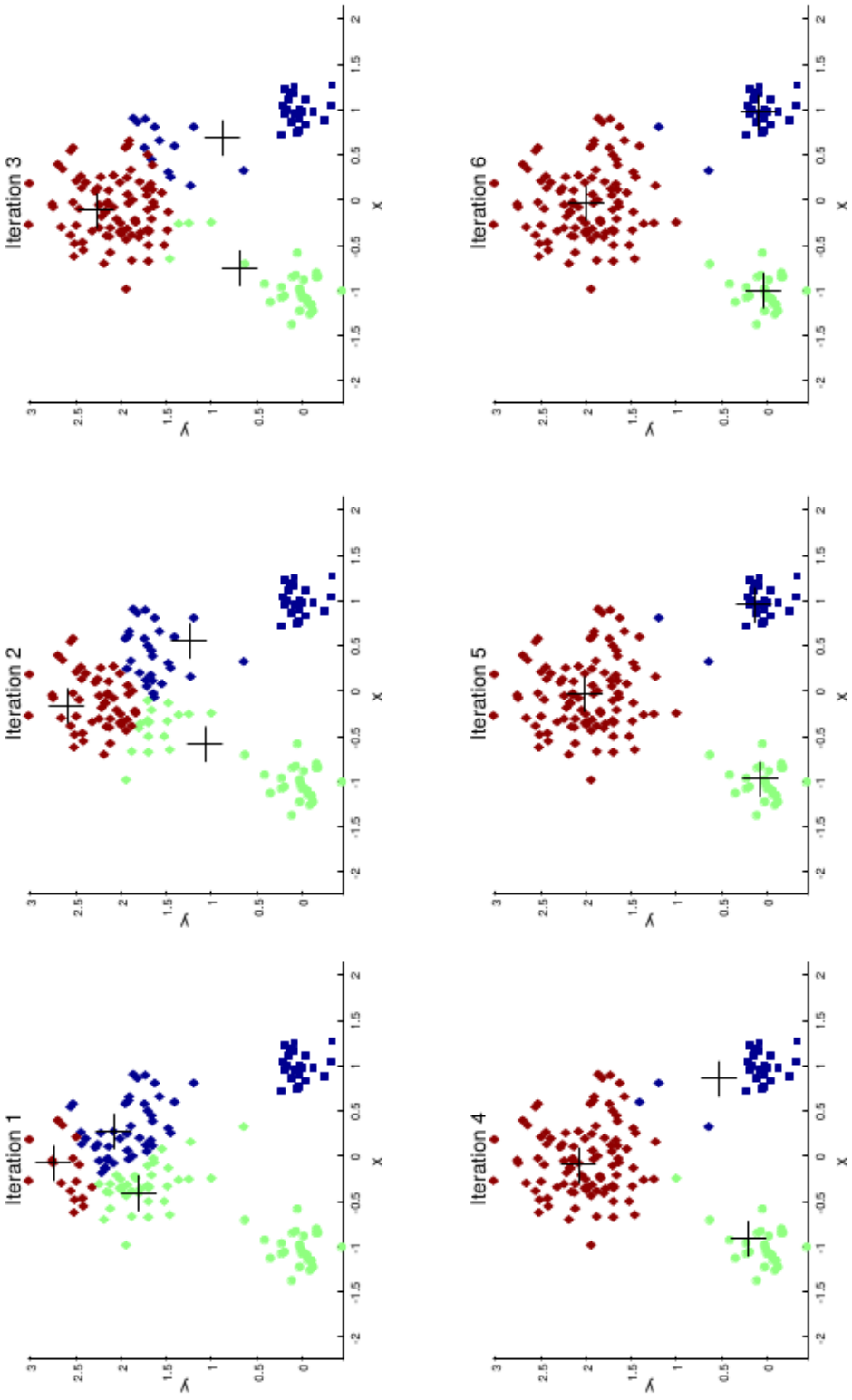


Figure 8.14: K-Means clustering example.

Problems with Selecting Initial Points

The chance of **selecting one centroid from each cluster** is small.

- Chance is relatively small when K is large
- If clusters are the same size, n, then

$$\begin{aligned} P &= \frac{\text{\# of ways to select a centroid from each cluster}}{\text{\# of ways to select } K \text{ centroids}} \\ &= \frac{K! n^K}{(Kn)^K} = \frac{K!}{K^K}. \end{aligned}$$

- For example, if $K = 5$ or 10, then probability is:

$$5!/5^5 = 0.0384, \quad 10!/10^{10} = 0.00036.$$

- Sometimes the initial centroids will readjust themselves in "right" way, and sometimes they don't.

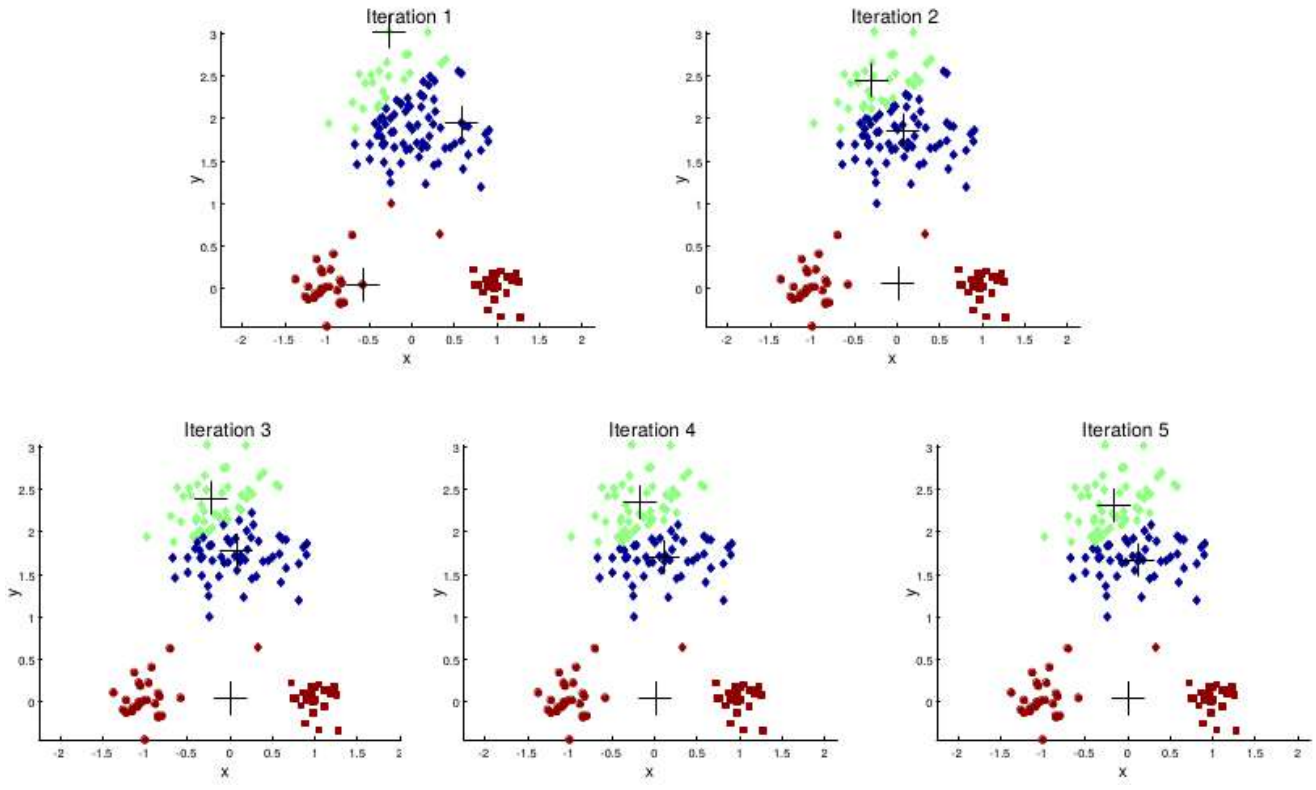


Figure 8.15: Importance of choosing initial centroids.

Solutions to Initial Centroids Problem

- **Multiple runs**
  - Helps, but probability is not on your side
- **Sample and use hierarchical clustering** to determine initial centroids
- Select **more than** K initial centroids and then, among these initial centroids
  - Select most widely separated
- **Post-processing**
- **Bisecting K-Means**
  - Not as susceptible to initialization issues

Pre-processing and Post-processing

- **Pre-processing**
  - Normalize the data
  - Eliminate outliers
- **Post-processing**
  - **Eliminate** small clusters that may represent outliers
  - **Split** "loose" clusters, i.e., clusters with relatively high SSE
  - **Merge** clusters that are "close" and that have relatively low SSE

  * Can use these steps during the clustering process – ISODATA

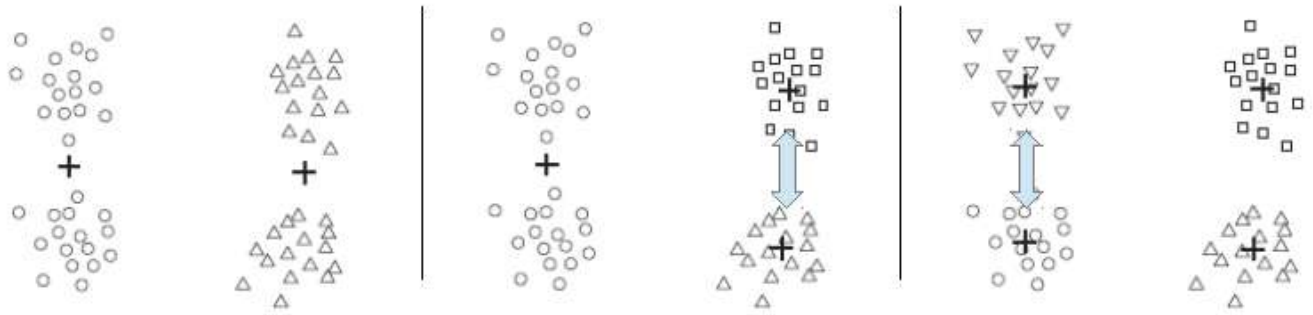
8.2.2. Bisecting K-Means algorithm

A variant of the K-Means that can produce a partitional or a hierarchical clustering

---

1. **Initialize** (a list of clusters), containing all points.
2. **Repeat**
   (a) Select a cluster from the list of clusters
   (b) **for** $i = 1$ to $iter\_runs$ **do**
   Bisect the selected cluster using the basic K-Means
   end for
   (c) Add the **two clusters from the bisection with the lowest SSE** to the list of clusters.

until (the list of clusters contains K clusters)



(a) Iteration 1. (b) Iteration 2. (c) Iteration 3.

Figure 8.16: Bisecting K-Means algorithm, with $K = 4$.

Note: The bisecting K-Means algorithm is not as susceptible to initialization issues as the basic K-Means clustering.

Limitations of K-Means Algorithms

- The K-Means have problems when clusters are of differing
  - **sizes, densities, and non-globular shapes**
- The K-Means have problems when the data contains **outliers**

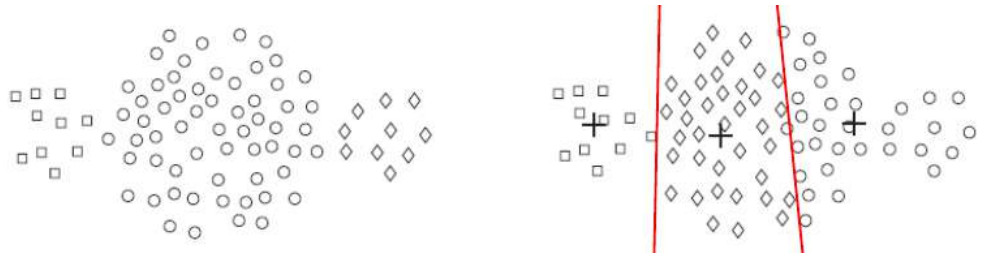


Figure 8.17: The K-Means with 3 clusters of different sizes.

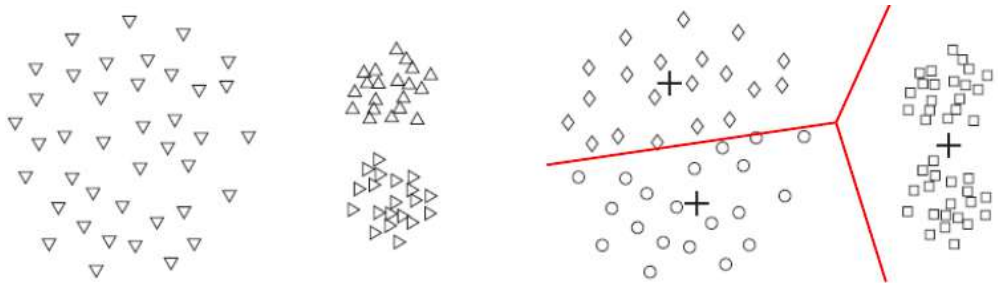


Figure 8.18: The K-Means with 3 clusters of different densities.

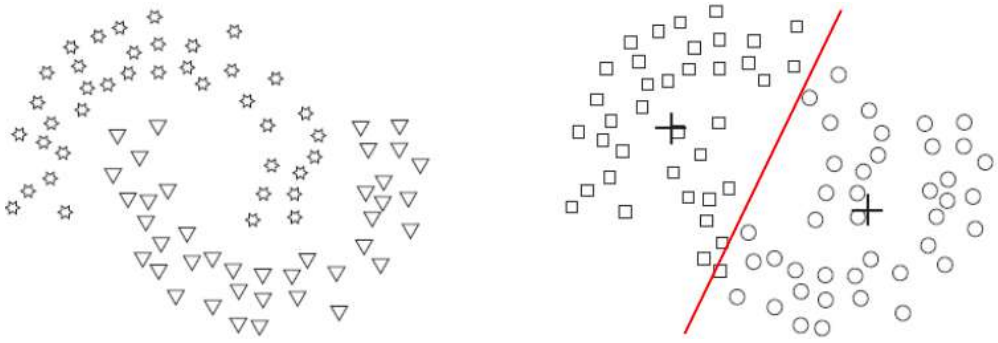


Figure 8.19: The K-Means with 2 non-globular clusters.

Overcoming K-Means Limitations

- Use a larger number of clusters
- **Several clusters represent a true cluster**

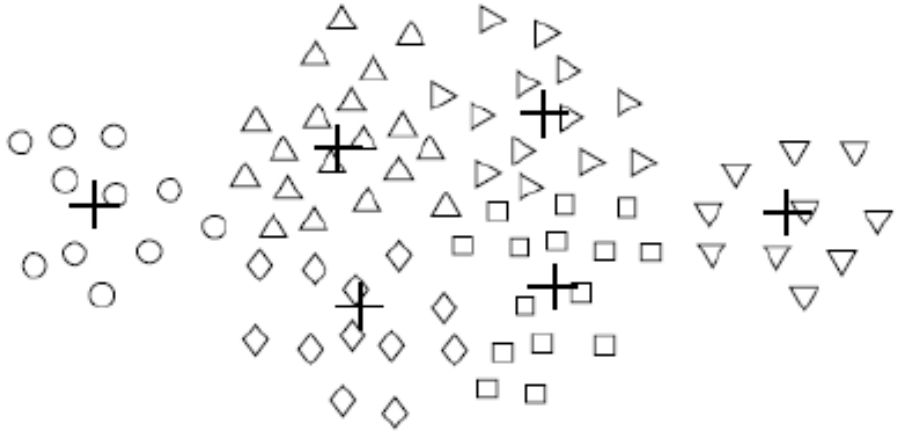


Figure 8.20: Unequal-sizes.

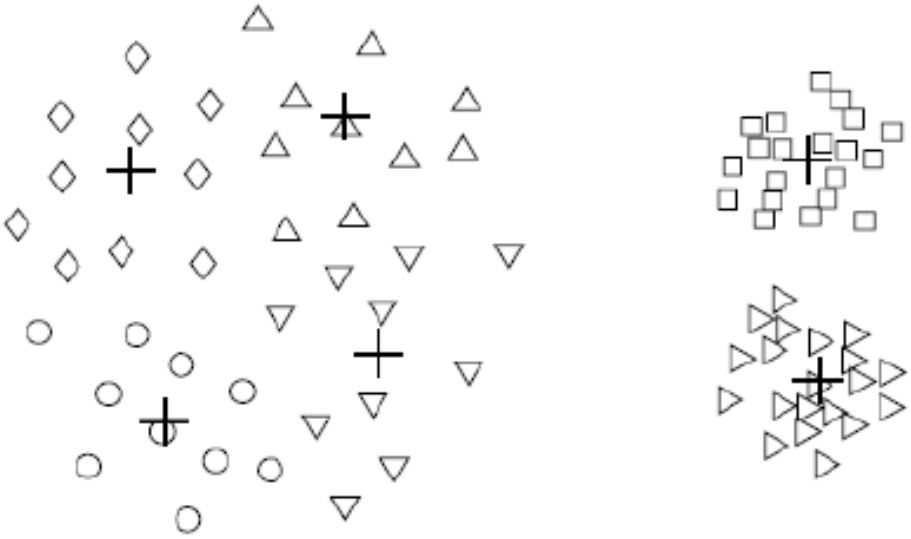


Figure 8.21: Unequal-densities.

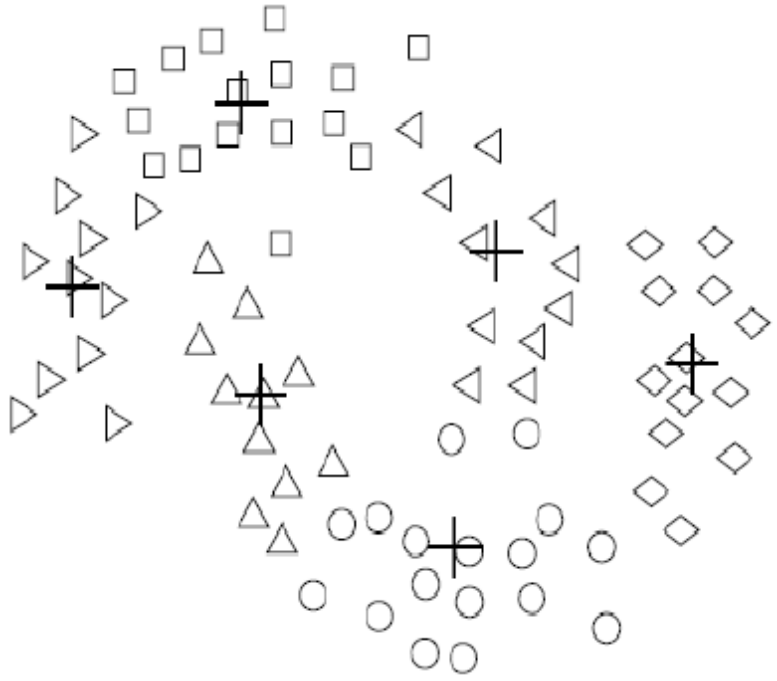


Figure 8.22: Non-spherical shapes.

Overcoming the K-Means Outlier Problem

- The K-Means algorithms are **sensitive to outliers**.
  - Since an object with an extremely large value may substantially distort the distribution of the data.
- **Solutions**:
  (a) Instead of taking the mean value of the object in a cluster as a reference point, **medoids** can be used, which is the **most centrally located object** in a cluster.
  (b) **Develop an effective outlier removal algorithm.** We will do it as a project which combines clustering and supervised learning for classification.

8.2.3. The K-Medoids algorithm

The **K-Medoids algorithm** (or **PAM algorithm**) is a clustering algorithm similar to the K-Means algorithm. Both the K-Means and K-Medoids algorithms are **partitional** (breaking the dataset up into groups) and both attempt to minimize the distance between points labeled to be in a cluster and a point designated as the center of that cluster. The K-Medoids chooses **data points as centers (medoids)** and can be used **with arbitrary distances**, while the K-Means only minimizes the squared Euclidean distances from cluster means. The PAM method was proposed by (Kaufman & Rousseeuw, 1987) [37] for the work **with L^1-norm and other distances**.

- Find representative objects, called **medoids**, in clusters.
- The **PAM** (partitioning around medoids) starts from an initial set of medoids and **iteratively replaces** one of the medoids by one of the non-medoids **if it improves the total distance** of the resulting clustering.
- The PAM works effectively for **small datasets**, but **does not scale well** for large data sets.
- **CLARA** (Clustering LARge Applications): sampling-based method (Kaufmann & Rousseeuw, 1990) [38]
- **CLARANS**: CLARA with randomized search (Ng & Han, 1994) [54]

PAM (Partitioning Around Medoids): Use real objects to represent the clusters (called medoids).

Initialization: select K representative objects;
Associate each data point to the closest medoid;
while (the cost of the configuration decreases) :
    For **each medoid** m and **each non-medoid data point** o :
        swap m **and** o;
        associate each data point to the closest medoid;
        recompute the cost (sum of distances of points to their medoid);
    If the total cost of the configuration increased, undo the swap;

Pros and cons of the PAM

- The PAM is **more robust** than the K-Means in the presence of noise and outliers, because a medoid is less influenced by outliers or other extreme values than a mean.
- The PAM works efficiently for small datasets but does **not scale well** for large data sets.
- The run-time complexity of the PAM is $\mathcal{O}(K(N-K)^2)$ for each iteration, where N is the number of data points and K is the number of clusters.

$\Rightarrow$ **CLARA** (Clustering LARge Applications): sampling-based method (Kaufmann & Rousseeuw, 1990) [38]

The PAM finds the best K-medoids among a given data, and the CLARA finds the best K-medoids among the selected samples.

8.2.4. CLARA and CLARANS

CLARA (Clustering LARge Applications)

- Sampling-based PAM (Kaufmann & Rousseeuw, 1990) [38]
- It draws **multiple samples** of the dataset, applies the PAM on each sample, and gives the best clustering as the output.

$\oplus$ **Strength**: deals with larger data sets than the PAM.

$\ominus$ **Weakness**:
  - Efficiency depends on the sample size.
  - A good clustering based on samples will not necessarily represent a good clustering of the whole data set, if the sample is biased.

- **Medoids are chosen from the sample**:
  $\ominus$ The algorithm cannot find the best solution if one of the best K-Medoids is not among the selected samples.

CLARANS ("Randomized" CLARA)

- **CLARANS** (CLARA with Randomized Search) (Ng & Han; 1994,2002) [54, 55]
- The CLARANS draws **sample of neighbors dynamically**.
  - CLARANS draws **a sample of neighbors** in each step of a search, while CLARA draws a sample of nodes at the beginning of a search.
- The clustering process can be presented as **searching a graph** where every node is a potential solution, that is, **a set of K medoids**.
- If a local optimum is found, **the CLARANS starts with new randomly selected node** in **search for a new local optimum**.
- Finds several local optimums and output the clustering with the **best local optimum**.

⊕ It is more efficient and scalable than both the PAM and the CLARA; handles outliers.

⊕ Focusing techniques and **spatial access structures** may further improve its performance; see (Ng & Han, 2002) [55] and (Schubert & Rousseeuw, 2018) [71].

⊖ Yet, the computational complexity of the CLARANS is $\mathcal{O}(N^2)$, where N is the number of objects.

⊖ The clustering quality depends on the sampling method.

8.3. Hierarchical Clustering

8.3.1. Basics of AGNES and DIANA

Hierarchical clustering can be divided into two main types: **agglomerative** and **divisive**.

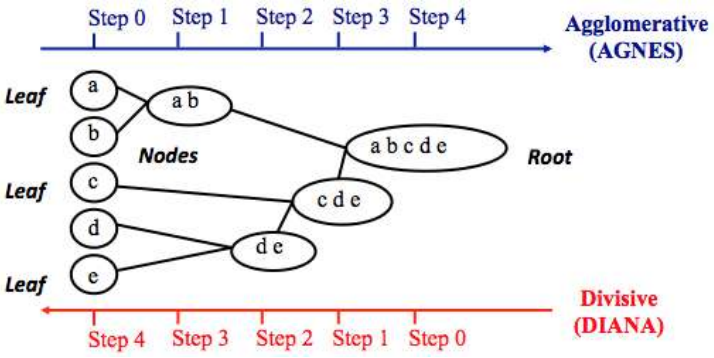


Figure 8.23: AGNES and DIANA

Agglomerative hierarchical clustering

(a.k.a. **AGNES**: Agglomerative Nesting). It works in a **bottom-up manner**.

- Each object is initially considered as its own singleton cluster (leaf).
- At each iteration, the **two closest clusters** are merged into a new bigger cluster (nodes).
- This procedure is iterated **until** all points are merged into a single cluster (root).
- The result is a tree which can be plotted as a dendrogram.

Divisive hierarchical clustering

(a.k.a. **DIANA**: Divisive Analysis). It works in a **top-down manner**; the algorithm is an inverse order of the AGNES.

- It begins with the root, where all objects are included in a single cluster.
- **Repeat**: the **most heterogeneous cluster** is divided into two.
- **Until**: all objects are in their own cluster.

Note: Agglomerative clustering is good at identifying small clusters, while divisive hierarchical clustering is good for large clusters.

Complexity

- The optimum cost is $\mathcal{O}(N^2)$, because it uses the proximity matrix. (N is the number of points)
- In practice, $\mathcal{O}(N^3)$ in many cases.
  - There are $\mathcal{O}(N)$ steps and at each step the proximity matrix of size $\mathcal{O}(N^2)$ must be updated and searched.
  - Complexity can be reduced to $\mathcal{O}(N^2 log(N))$ for some approaches.

Limitations

- **Greedy**: Once a decision is made to combine two clusters, it cannot be undone.
- **No global objective function** is directly minimized.
- Most algorithms have problems with one or more of:
  - Sensitivity to noise and outliers
  - Difficulty handling different sized clusters and non-convex shapes
  - Chaining, breaking large clusters

Hierarchical Clustering vs. K-Means

- Recall that K-Means or K-Medoids requires
  - The number of clusters K
  - An initial assignment of data to clusters
  - A distance measure between data $d(\mathbf{x}_i, \mathbf{x}_j)$
- Hierarchical clustering requires only a **similarity measure** between groups/clusters of data points.

8.3.2. AGNES: Agglomerative clustering

Quesiton. How do we measure the similarity (or dissimilarity) between two groups of observations?

A number of different **cluster agglomeration methods** (i.e, linkage methods) have been developed to answer to the question. The most popular choices are:

- Single linkage
- Complete linkage
- Group linkage
- Centroid linkage
- Ward's minimum variance

1. **Single linkage**: the similarity of the closest pair

$$d_{SL}(G, H) = \min_{i \in G, j \in H} d(i, j). \tag{8.4}$$

   - Single linkage can produce **"chaining"**, where a sequence of close observations in different groups cause early merges of those groups
   - It tends to produce **long "loose" clusters**.

2. **Complete linkage**: the similarity of the furthest pair

$$d_{CL}(G, H) = \max_{i \in G, j \in H} d(i, j). \tag{8.5}$$

   - Complete linkage has the opposite problem; it might not merge close groups because of **outlier members** that are far apart.
   - It tends to produce **more compact clusters**.

3. **Group average**: the average similarity between groups

$$d_{GA}(G, H) = \frac{1}{N_G N_H} \sum_{i \in G} \sum_{j \in H} d(i, j). \tag{8.6}$$

   - Group average represents a **natural compromise**, but depends on the scale of the similarities. Applying a monotone transformation to the similarities can change the results.

4. **Centroid linkage**: It computes the dissimilarity between the centroid for group G (a mean vector of length d variables) and the centroid for group H.

5. **Ward's minimum variance**: It minimizes the total within-cluster variance. More precisely, at each step, the method finds **the pair of clusters** that leads to **minimum increase in total within-cluster variance** after merging. It uses the squared error (as an objective function).

Interpretable Visualization of AGNES

- Each level of the resulting tree is a segmentation of data
- The algorithm results in a **sequence** of groupings
- It is **up to the user to choose a natural clustering** from this sequence

Dendrogram

- Agglomerative clustering is monotonic
  - The **similarity** between merged clusters is **monotone decreasing** with the level of the merge.
- **Dendrogram**: Plot each merge at the dissimilarity between the two merged groups
  - Provides an **interpretable visualization** of the algorithm and data
  - **Useful summarization tool**, part of why hierarchical clustering is popular

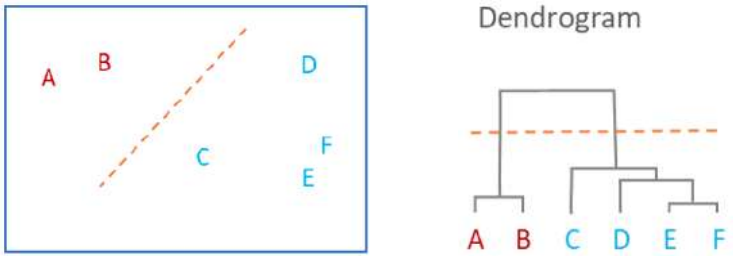


Figure 8.24: Six observations and a dendrogram showing their hierarchical clustering.

Remark 8.2.

- The height of the dendrogram indicates **the order** in which the clusters were joined; it reflects **the distance between the clusters**.
- The greater the difference in height, the more **dissimilarity**.
- Observations are allocated to clusters by drawing a **horizontal line** through the dendrogram. Observations that are joined together below the line are in the same clusters.

Single Link

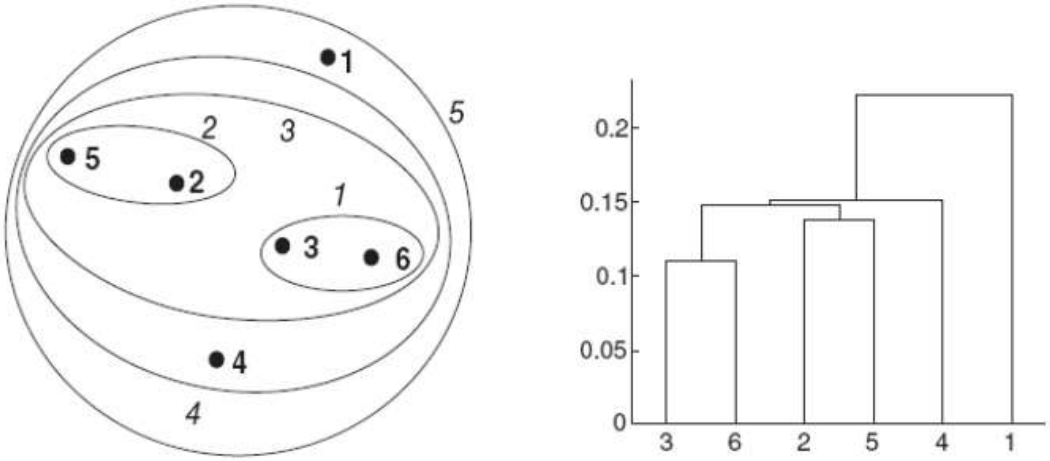


Figure 8.25: Single link clustering of six points.

- **Pros**: Non-spherical, non-convex clusters
- **Cons**: Chaining

Complete Link

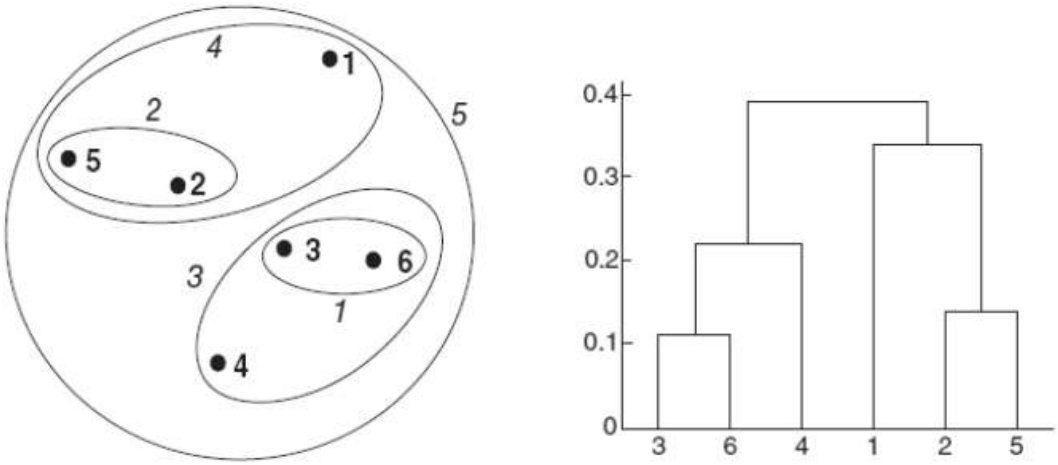


Figure 8.26: Complete link clustering of six points.

- **Pros**: more robust against noise (no chaining)
- **Cons**: Tends to break large clusters; biased towards globular clusters

Average Link

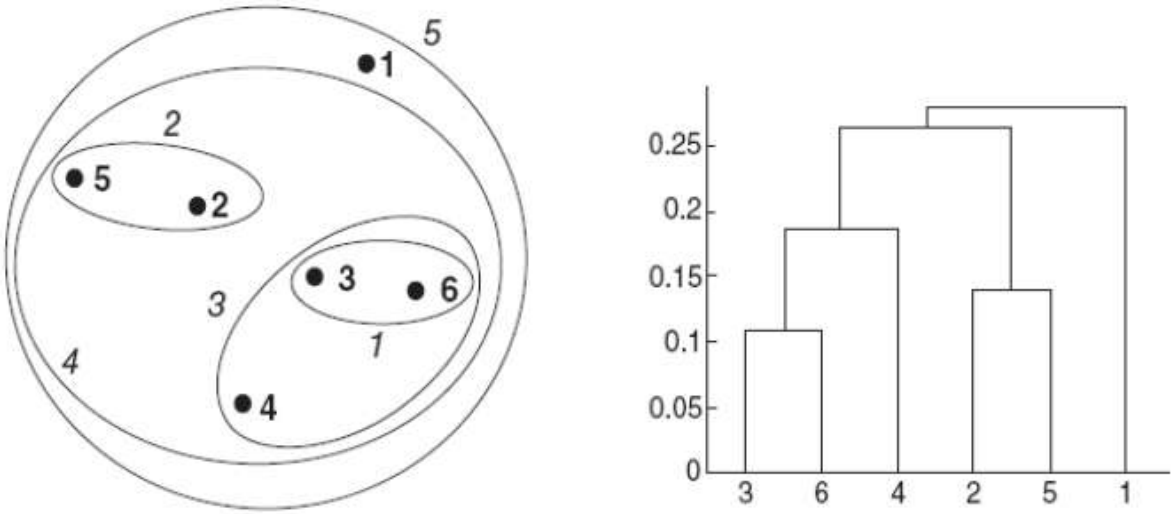


Figure 8.27: Average link clustering of six points.

- Compromise between single and complete links

Ward's Minimum Variance Method

- Similarity of two clusters is based on the increase in *squared error* when two clusters are merged
- **Less susceptible to noise and outliers**
- Biased towards *globular clusters*.
- Hierarchical analogue of the K-Means; it can be used **to initialize the K-Means**. (Note that the K-Means works with a global objective function.)

8.4. DBSCAN: Density-based Clustering

In **density-based clustering**:

- Clusters are defined as **areas of higher density** than the remainder of the data set. (**core points**)
- Objects in sparse areas are usually considered to be **noise** and **border points**.
- The most popular density-based clustering method is
  - **DBSCAN**[a] (Ester, Kriegel, Sander, & Xu, 1996) [18].

[a] Density-Based Spatial Clustering of Applications with Noise (DBSCAN).

DBSCAN

- Given a set of points, it groups points that are closely packed together (points with many nearby neighbors),
  - **marking as outliers** points that lie alone in low-density regions.
- It is **one of the most common clustering algorithms** and also most cited in scientific literature. (Citation #: 28,008, as of Apr. 15, 2023)
  - In 2014, the algorithm was awarded **the test of time award**[a] at the leading data mining conference, KDD 2014: https://www.kdd.org/kdd2014/.

[a] The test of time award is an award given to algorithms which have received substantial attention in theory and practice.

Preliminary for DBSCAN

- Consider a dataset to be clustered.
- Let ε be a parameter specifying the **radius** of a neighborhood with respect to some point.
- In DBSCAN clustering, the points are classified as **core points**, **reachable points**, and **outliers**, as follows:
  - **A point** p is a **core point** if at least m (=`minPts`) points are within distance ε of it (including p itself).
  - **A point q** is **directly reachable from p** if point q is within distance ε from the core point p.
    (Points are only said to be directly reachable from core points.)
  - A point q is **reachable from p** if there is a path $p_1, \cdots, p_n$ with $p_1 = p$ and $p_n = q$, where each p_{i+1} is directly reachable from p_i.
    (Note that this implies that all points on the path must be core points, with the possible exception of q.)
  - All points not reachable from any other points are **outliers** or **noise points**.
- Now, **a core point forms a cluster** together with all points (core or non-core) that are **reachable** from it.
  - Each cluster contains **at least one core point**;
    non-core points can be part of a cluster, but they form its **"edge"**, since they cannot be used to reach more points.
  - A non-core reachable point is also called a **border point**.

User parameters:

- ε: the radius of a neighborhood
- `minPts`: the minimum number of points in the ε-neighborhood

Illustration of the DBSCAN

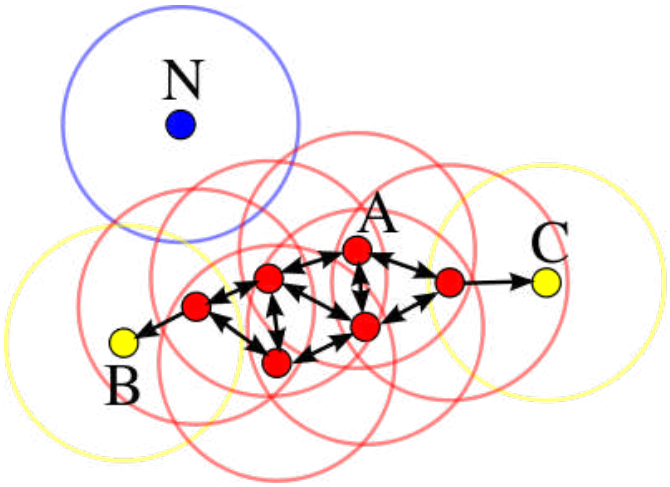


Figure 8.28: Illustration of the DBSCAN, with m (= minPts) = 4.

- Point A and 5 other red points are **core points**. They are all reachable from one another, so they form **a single cluster**.
- Points B and C are not core points, but are **reachable from** A (via other core points) and **thus belong to the cluster** as well.
- Point N is a **noise point** that is neither a core point nor directly-reachable.

Note: **Reachability is not a symmetric relation** since, by definition, no point may be reachable from a non-core point, regardless of distance. (A non-core point may be reachable, but nothing can be reached from it.)

Definition 8.3. *Two points p and q are **density-connected** if there is a point c such that both p and q are reachable from c. Density-connectedness is symmetric.*

A DBSCAN cluster satisfies two properties:

1. All points within the cluster are **mutually density-connected**.
2. If a point is **density-reachable** from any point of the cluster, then it is part of the cluster as well.

DBSCAN: Pseudocode

DBSCAN

```
1  DBSCAN(D, eps, MinPts)
2      C=0                                              # Cluster counter
3      for each unvisited point P in dataset D
4          mark P as visited
5          NP = regionQuery(P, eps)                     # Find neighbors of P
6          if size(NP) < MinPts
7              mark P as NOISE
8          else
9              C = C + 1
10             expandCluster(P, NP, C, eps, MinPts)
11
12 expandCluster(P, NP, C, eps, MinPts)
13     add P to cluster C
14     for each point Q in NP
15         if Q is not visited
16             mark Q as visited
17             NQ = regionQuery(Q, eps)
18             if size(NQ) >= MinPts
19                 NP = NP joined with NQ
20         if Q is not yet member of any cluster
21             add Q to cluster C
```

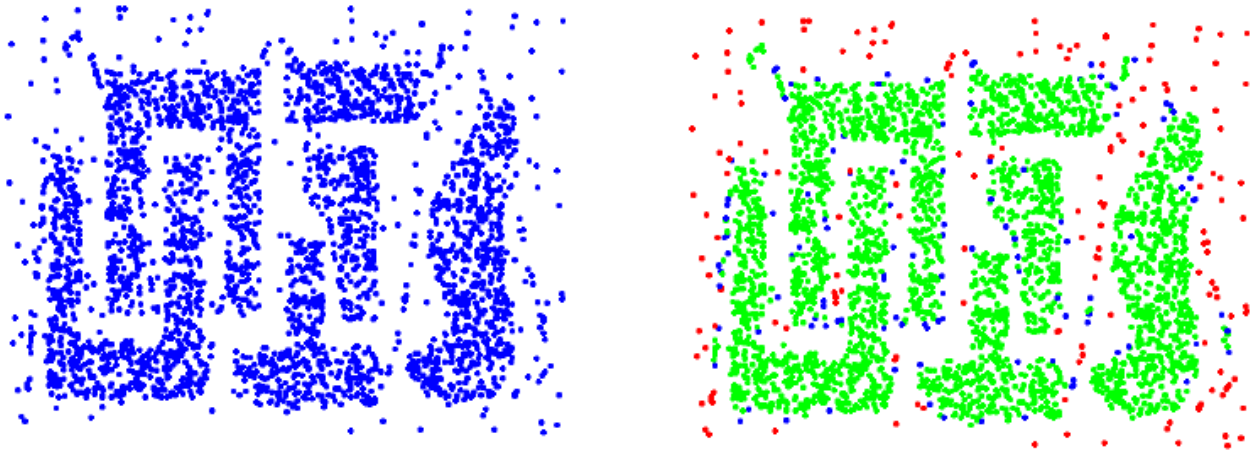


Figure 8.29: Original points (left) and point types of the DBSCAN clustering with `eps=10` and `MinPts=4` (right): core (green), border (blue), and noise (red).

Note: **In Pseudocode**: Line 7 may classify a border point as noise, which would be corrected by Lines 20-21.

Properties of DBSCAN Clustering

- Resistant to Noise
- Can handle clusters of different shapes and sizes
- `Eps` and `MinPts` depend on each other and **can be hard to specify**

When the DBSCAN does NOT work well

- Varying densities
- High-dimensional data

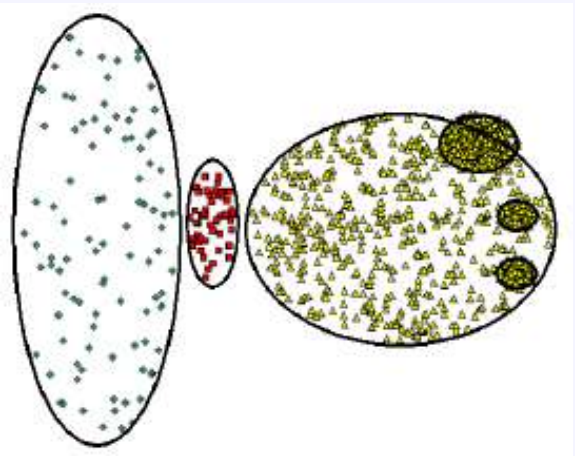


Figure 8.30: Original points.

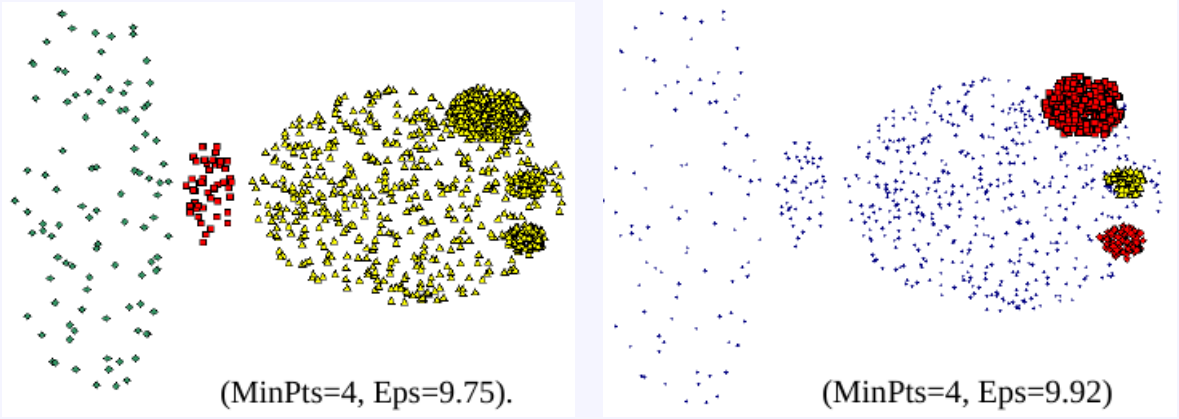


Figure 8.31: The DBSCAN clustering. For both cases, it results in 3 clusters.

Overall, DBSCAN is a great density-based clustering algorithm.

8.5. Cluster Validation

8.5.1. Basics of cluster validation

- For supervised classification (= using class label), we have a variety of measures to evaluate how good our model is.
  - Accuracy, precision, recall
- For cluster analysis (= unsupervised), the analogous question is:

  How to evaluate the *"goodness"* of the resulting clusters?

- But "clusters are in the eye of the beholder"!
- Yet, we want to evaluate them. Why?
  - To avoid finding patterns in **noise**
  - To compare **clustering algorithms**
  - To compare **two sets of clusters**
  - To compare **two clusters**

Aspects of Cluster Validation

1. Understanding the **clustering tendency** of a set of data, (i.e., distinguishing **non-random structures** from all the retrieved).
2. **Validation Methods?**
   - ***External validation***: Compare the results of a cluster analysis to externally known class labels (ground truth).
   - ***Internal validation***: Evaluating how well the results of a cluster analysis fit the data without reference to external information – **use only the data**.
3. **Compare clusterings** to determine which is better.
4. Determining the **"correct" number of clusters**.

For 2 and 3, we can further distinguish whether we want to evaluate the entire clustering or just individual clusters.

Definition 8.4. Precision and Recall
Precision is the fraction of relevant instances among all the retrieved, while **recall** (a.k.a. **sensitivity**) is the fraction of relevant instances that were retrieved.

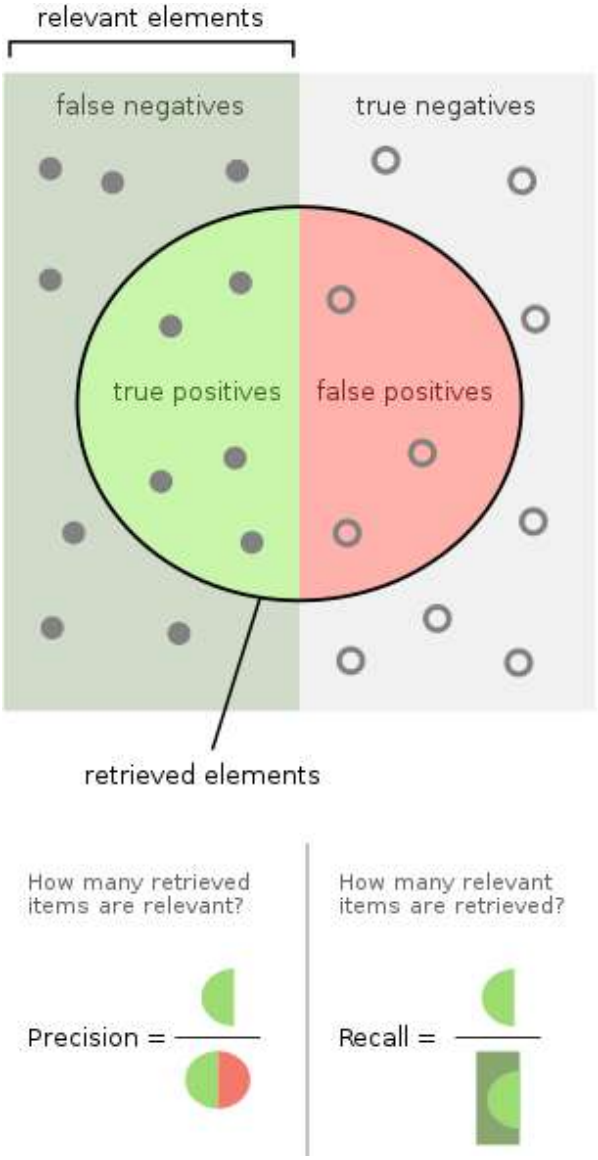


Figure 8.32: Illustration of precision and recall, Wikipedia.

Note: Therefore, both **precision** and **recall** are about **relevance of the retrieval**, measured respectively from **all the retrieved instances** and **all the relevant instances in the database**.

Measures of Cluster Validity

Numerical measures for judging various aspects of cluster validity are classified into the following three types.

- **External Measures**: Used to measure the extent to which cluster labels match **externally supplied class labels**.
  - Entropy, Purity, Rand index
  - Precision, Recall
- **Internal Measures**: Used to measure the **goodness** of a clustering structure **without respect to external information**.
  - Correlation, Similarity matrix
  - Sum of Squared Error (SSE), Silhouette coefficient
- **Relative Measures**:
  - Used to compare 2 different clusterings or clusters.
  - Often an external or internal measure is used for this function, e.g., SSE or entropy

Definition 8.5. The **correlation** coefficient $\rho_{X,Y}$ between two random variables X and Y with expected values μ_X and μ_Y and standard deviations σ_X and σ_Y is defined as

$$\rho_{X,Y} = \operatorname{corr}(X,Y) = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{\mathbb{E}[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \in [-1,1]. \tag{8.7}$$

If X and Y are independent, $\rho_{X,Y} = 0$. (The reverse may not be true.)

Note: **The correlation between two vectors $\mathbf{u}$ and $\mathbf{v}$** is the cosine of the angle between the two vectors.

$$\operatorname{corr}(\mathbf{u},\mathbf{v}) = \frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{u}\|\,\|\mathbf{v}\|}. \tag{8.8}$$

The correlation between two matrices can be defined similarly, by considering the matrices as vectors.

Measuring Cluster Validity Via Correlation

- Two matrices
  - **Proximity matrix**[a] ($P \in \mathbb{R}^{N \times N}$)
  - **Incidence matrix** ($I \in \mathbb{R}^{N \times N}$)
    - One row and one column for each data point
    - An entry is 1 if the associated pair of points belong to the same cluster
    - An entry is 0 if the associated pair of points belongs to different clusters
- **Compute the correlation between the two matrices**
  - Since the matrices are symmetric, only the correlation between $N(N-1)/2$ entries needs to be calculated.
- **High correlation** indicates that points that belong to the same cluster are close to each other.

  Example: For K-Means clusterings of two data sets, the correlation coefficient are:

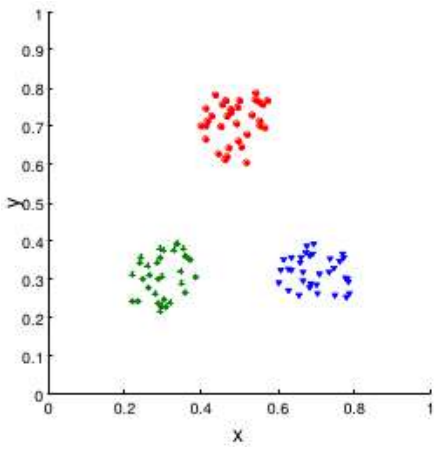


Figure 8.33: $\rho_{P,I} = -0.924$.

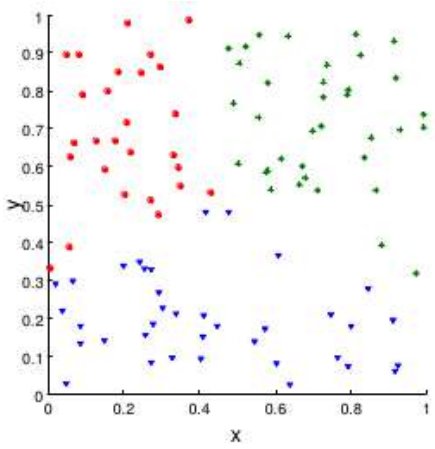


Figure 8.34: $\rho_{P,I} = -0.581$.

- **Not a good measure for some density- or contiguity-based clusters** (e.g., single link HC).

[a] A **proximity matrix** is a square matrix in which the entry in cell (i, j) is some measure of the similarity (or distance) between the items to which row i and column j correspond.

Using Similarity Matrix for Cluster Validation

Order the **similarity matrix** with respect to cluster labels and inspect visually.

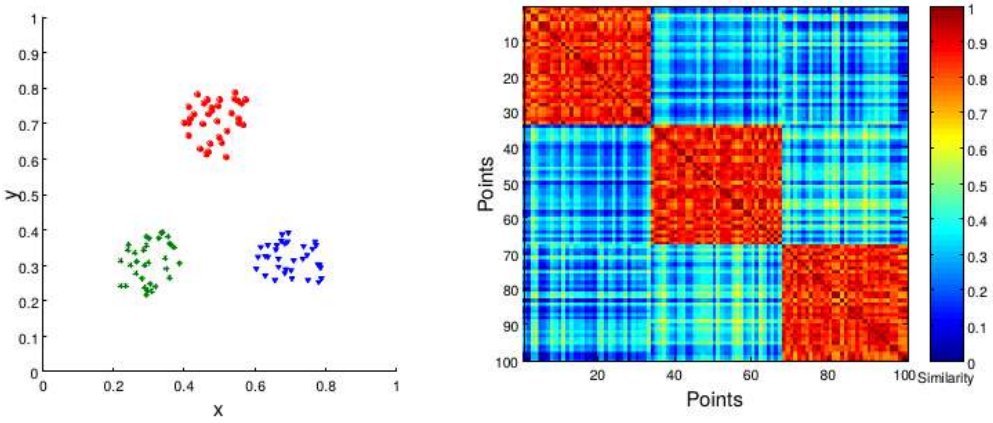


Figure 8.35: **Clusters are so crisp!**

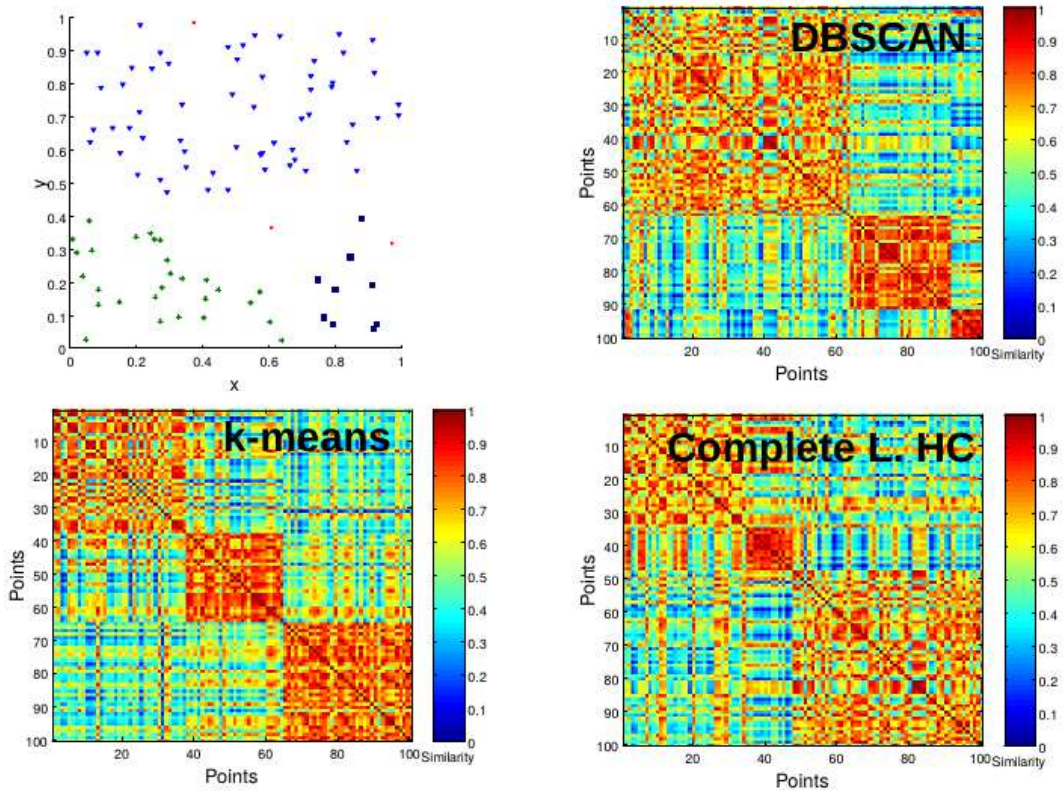


Figure 8.36: Clusters in random data are not so crisp.

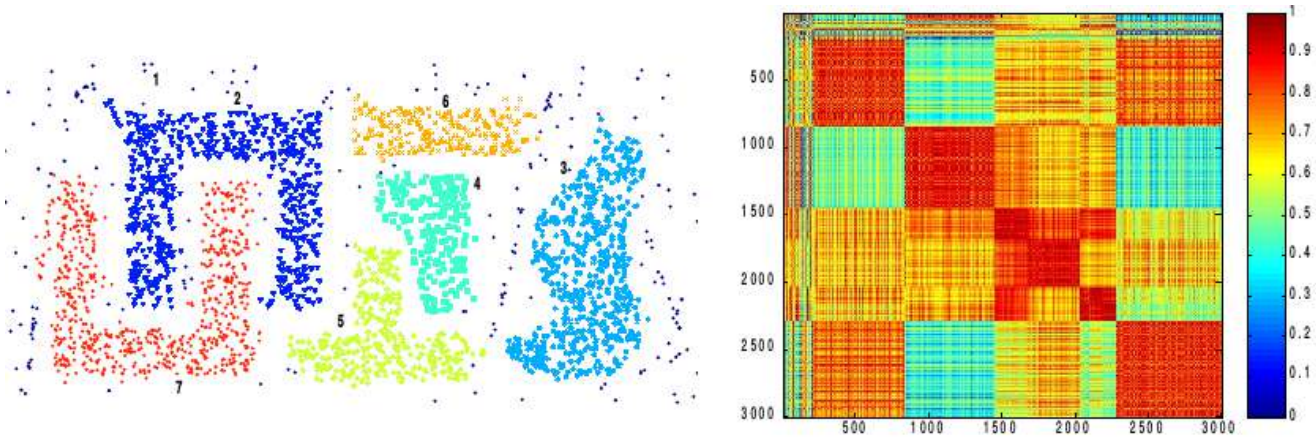


Figure 8.37: Similarity matrix for cluster validation, for DBSCAN.

8.5.2. Internal and external measures of cluster validity

Internal Measures

- **(Average) SSE** is good for comparing two clusterings or two clusters.

$$SSE = \sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2. \tag{8.9}$$

- It can also be used **to estimate the number of clusters**

10 clusters

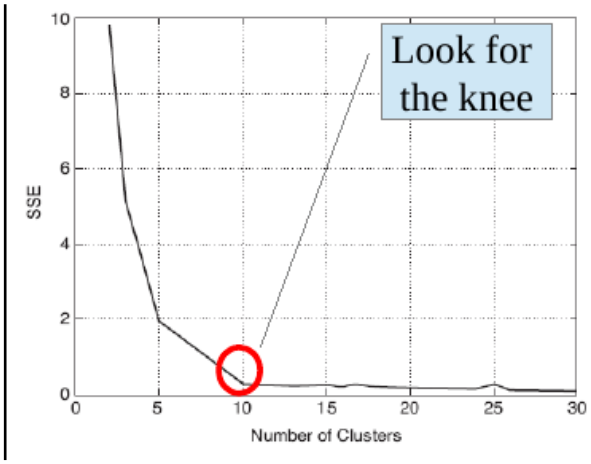


Figure 8.38: Estimation for the number of clusters.

Cohesion and Separation

- **Cluster cohesion**: Measure how closely related are objects in a cluster
  - Example: Within-cluster sum of squares (WSS=SSE)

$$WSS = \sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2. \tag{8.10}$$

- **Cluster separation**: Measures how distinct or well-separated a cluster is from other clusters
  - Example: Between-cluster sum of squares (BSS)

$$BSS = \sum_{i=1}^{K} |C_i| \, ||\boldsymbol{\mu} - \boldsymbol{\mu}_i||^2. \tag{8.11}$$

- **Total sum of squares**: $TSS = WSS + BSS$
  - TSS is a constant for a given data set (independently of the number of clusters)
  - Example: a cluster $\{1, 2, 4, 5\}$ can be separated into two clusters $\{1, 2\} \cup \{4, 5\}$. It is easy to check the following.
    - 1 cluster: $TSS = WSS + BSS = 10 + 0 = 10$.
    - 2 clusters: $TSS = WSS + BSS = \ 1 + 9 = 10$.

Silhouette Coefficient

- **Silhouette coefficient** combines ideas of **both cohesion and separation**, **but for individual points**. For an individual point i:
  - Calculate $a(i)$ = average distance of i to all other points in its cluster
  - Calculate $b(i)$ = min {average distance of i to points in another cluster}
  - The silhouette coefficient for the point i is then given by

$$s(i) = 1 - a(i)/b(i). \tag{8.12}$$

  - Typically, $s(i) \in [0, 1]$.
  - The closer to 1, the better.
- We can calculate **the average silhouette width** for a cluster or a clustering

Selecting K with Silhouette Analysis on K-Means Clustering

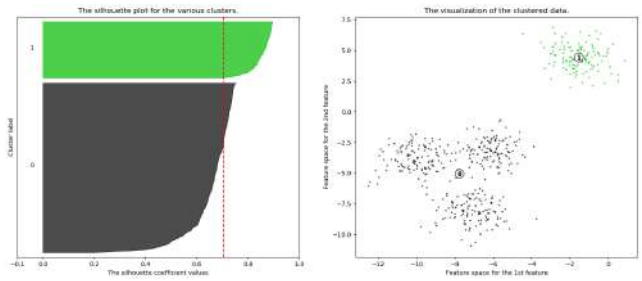


Figure 8.39: `n_clusters = 2`; `average silhouette score = 0.705`.

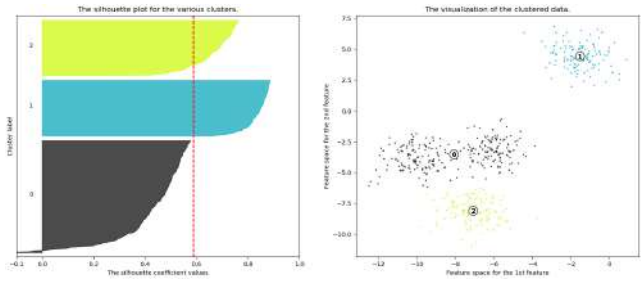


Figure 8.40: `n_clusters = 3`; `average silhouette score = 0.588`.

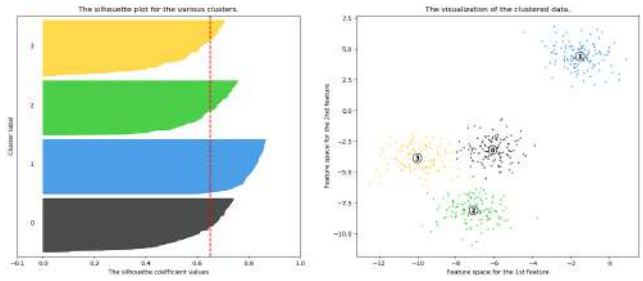


Figure 8.41: `n_clusters = 4`; `average silhouette score = 0.651`.

https://scikit-learn.org

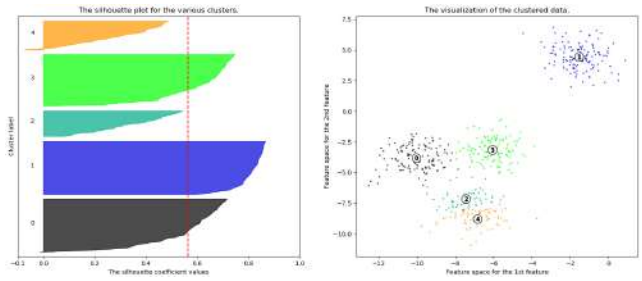


Figure 8.42: `n_clusters = 5`; `average silhouette score = 0.564`.

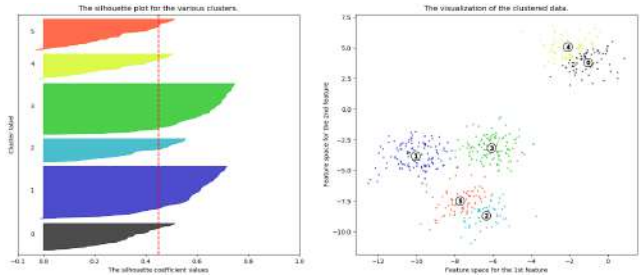


Figure 8.43: `n_clusters = 6`; `average silhouette score = 0.450`.

- The silhouette plot shows that (`n_clusters` **= 3, 5, and 6**) are **bad picks** for the data, due to
  - the presence of clusters with below average silhouette scores
  - wide fluctuations in the size of the silhouette plots
- Silhouette analysis is ambivalent in deciding between **2 and 4**.
- When `n_clusters` **= 4**, all the silhouette subplots are more or less of similar thickness.

Another way of Picking K with Silhouette Analysis

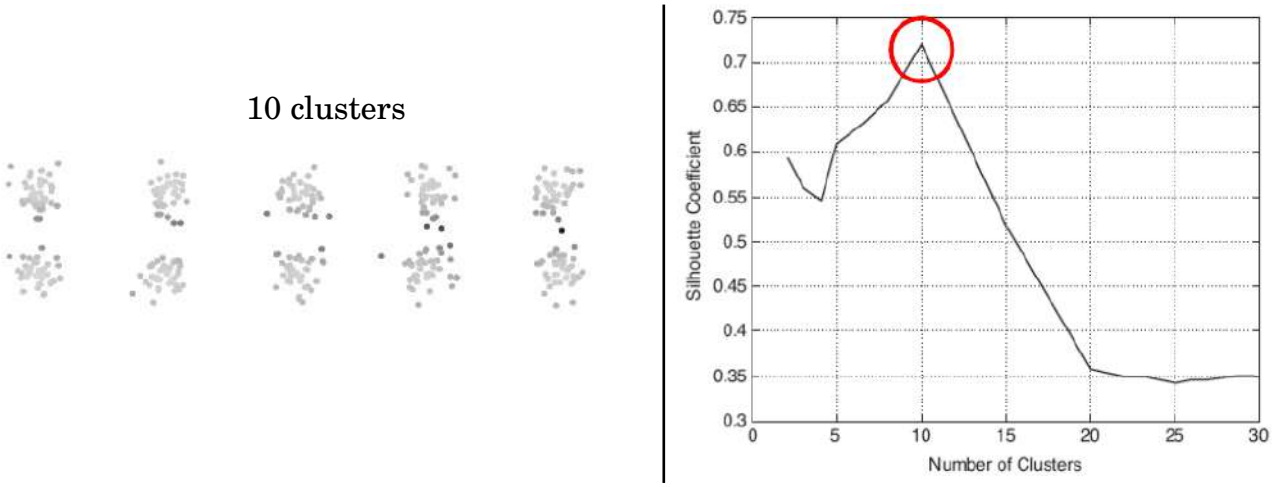


Figure 8.44: Average silhouette coefficient vs. number of clusters.

External Measures of Cluster Validity

- **Entropy**[a]
  - For cluster j, let p_{ij} be the probability that a member of cluster j belongs to class i, defined as
$$p_{ij} = n_{ij}/N_j, \tag{8.13}$$
where N_j is the number of points in cluster j and n_{ij} is the number of points of class i in cluster j.
  - **The entropy of each cluster** j is defined as
$$e_j = -\sum_{i=1}^{L} p_{ij} \log_2 p_{ij}, \tag{8.14}$$
where L is the number of classes and
  - The total entropy is calculated as the sum of entropies of each cluster weighted by the size of each cluster: for $N = \sum_{j=1}^{K} N_j$,
$$e = \frac{1}{N} \sum_{j=1}^{K} N_j \, e_j. \tag{8.15}$$
- **Purity**
  - The purity of cluster j is given by
$$\text{purity}_j = \max_i p_{ij}. \tag{8.16}$$
  - The overall purity of a clustering is
$$\text{purity} = \frac{1}{N} \sum_{j=1}^{K} N_j \, \text{purity}_j. \tag{8.17}$$

[a] The concept of **entropy** was introduced earlier in § 5.4.1. *Decision tree objectives*, when we defined impurity measures. See (5.68) on p. 132.

Final Comment on Cluster Validity

The following is a claim in an old book by (Jain & Dubes, 1988) [35].
However, today, the claim is yet true.

> "The validation of clustering structures is the most difficult and frustrating part of cluster analysis. Without a strong effort in this direction, cluster analysis will remain a black art accessible only to those true believers who have experience and great courage."

8.6. Self-Organizing Maps

8.6.1. Basics of the SOM

- **Self-Organizing Map** (**SOM**) refers to a process in which *the internal organization increases automatically* without being guided or managed by an outside source.
  - This process is due to **local interaction** with **simple rules**.
  - Local interaction gives rise to a **global structure**.
- **Why SOM?**
  A high-dimensional dataset is represented as an one/two-dimensional discretized pattern using **self-organizing maps** or **Kohonen maps**.
- **Advantage of SOM?**
  The primary benefit of employing an SOM is that **the data is simple to read and comprehend**. Grid clustering and the decrease of dimensionality make it simple to spot patterns in the data.

We can interpret emerging **global structures** as **learned structures**, which in turn appear as **clusters** of similar objects.

Note: The SOM acts as a **unsupervised clustering algorithm** and a powerful **visualization tool** as well.

- It considers a **neighborhood structure** among the clusters.

⊕ The SOM is **widely used** in many application domains, such as economy, industry, management, sociology, geography, text mining, etc..

⊕ **Many variants** have been suggested to adapt the SOM to the processing of complex data, such as time series, categorical data, nominal data, dissimilarity or kernel data.

⊖ However, the SOM has suffered from **a lack of rigorous results** on its **convergence** and **stability**.

Game of Life: – Most famous example of self-organization.

Simple local rules (en.wikipedia.org/wiki/Conway's_Game_of_Life): Suppose that every cell interacts with its *eight* neighbors.

- **Any live cell with fewer than two live neighbors** dies, as if caused by under-population.
- **Any live cell with two or three live neighbors** lives on to the next generation.
- **Any live cell with more than three live neighbors** dies, as if by overcrowding.
- **Any dead cell with exactly three live neighbors** becomes a live cell, as if by reproduction.

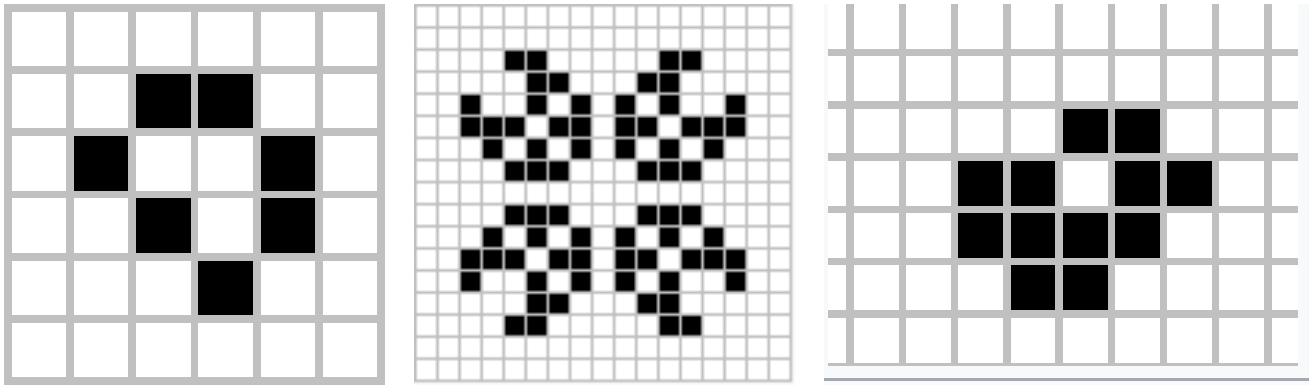


Figure 8.45: Still life, oscillator, and spaceship.

SOM Architecture

- A **feed-forward neural network** architecture based on **competitive learning**[a], invented by **Teuvo Kohonen** in 1982 [39].
- Neurobiological studies indicate that different sensory inputs (motor[b], visual, auditory, etc.) are mapped onto corresponding areas of the cerebral cortex in an **orderly fashion**.
  - Our interest is in building **artificial topographic maps** that learn through self-organization in a neurobiologically inspired manner.

[a] One particularly interesting class of unsupervised system is based on **competitive learning**, in which the output neurons compete amongst themselves to be activated, with the result that only one is activated at any one time. This activated neuron is called a **winner-takes-all neuron** or simply the **winning neuron**. Such competition can be induced/implemented by having **lateral inhibition connections** (negative feedback paths) between the neurons. The result is that the neurons are forced to organize themselves.

[b] **Motor output** is a response to the stimuli received by the nervous system.

- **The principal goal of an SOM** is to **transform** an incoming signal pattern of arbitrary dimension **into a one/two-dimensional discrete map**, and to perform this transformation adaptively in a topologically ordered fashion.
  - We therefore set up our SOM by placing neurons at the nodes of a one/two-dimensional lattice.
  - Higher dimensional maps are also possible, but not so common.
- **The neurons become selectively tuned**, and the locations of the neurons so tuned (i.e. the winning neurons) become ordered, and a **meaningful coordinate system** for the input features is created on the lattice.
  - The SOM thus forms the required topographic map of the input patterns.
- We can view this as **a non-linear generalization of principal component analysis (PCA)**.

Versions of the SOM

- **Basic version**: a stochastic process
- **Deterministic version**:
  - For **industrial applications**, it can be more convenient to use a deterministic version of the SOM, in order to get **the same results** at each run of the algorithm when the initial conditions and the data remain unchanged **(repeatable!)**.
  - To address this issue, T. Kohonen has introduced the batch SOM in 1995 [40].

Remark 8.6. The following are quite deeply related to each other.

(a) **Repeatability**
(b) **Optimality**
(c) **Convergence**
(d) **Interpretability**

Indeterministic Issue & Deterministic Clustering

Clustering algorithms are to partition objects into groups based on their similarity.

- Many clustering algorithms face **indeterministic issue**.
  - For example, the standard K-means algorithm randomly selects its initial centroids, which causes to produce different results in each run.
- There have been several studies on how to achieve **deterministic clustering**.
  (a) **Multiple runs**
  (b) **Initialization**, using hierarchical clustering approaches and PCA
  (c) **Elimination of randomness** (Zhang *et al.*, 2018) [83]

8.6.2. Kohonen SOM networks

We will see some details of the **Kohonen SOM network** or **Kohonen network**.

- The Kohonen SOM network has a **feed-forward structure** with a **single computational layer** arranged in rows and columns.
- Each neuron is **fully connected** to all the source nodes in the input layer

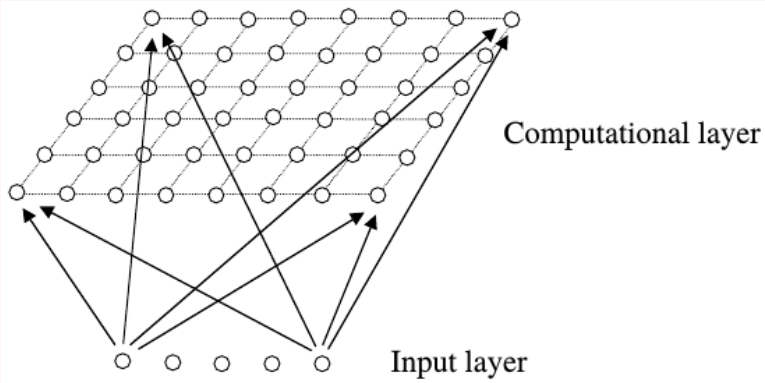


Figure 8.46: Kohonen network.

Data Types for the Kohonen SOM

Originally, the SOM algorithm was defined for data described by numerical vectors which belong to a subset X of $\mathbb{R}^d$.

- **Continuous setting**: the input space $X \subset \mathbb{R}^d$ is modeled by a probability distribution with a density function f,
- **Discrete setting**: the input space X comprises N data points

$$\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N \in \mathbb{R}^d.$$

Here the discrete setting means a finite subset of the input space.

Components of Self-Organization

Preliminary **8.7.** The self-organization process involves four major components:

1. **Initialization**: **All the connection weights** are initialized with small random values.
2. **Competition**: For each input pattern, the neurons compute their respective values of a **discriminant function** which provides the basis for competition.
   - The particular neuron with **the smallest value of the discriminant function** is declared **the winner**.
3. **Cooperation**: **The winning neuron determines the spatial location of a topological neighborhood** of excited neurons, thereby providing the basis for cooperation among neighboring neurons.
   (smoothing the neighborhood of the winning neuron)
4. **Adaptation**: The excited neurons decrease their individual values of the discriminant function in relation to the input pattern through suitable adjustment of the associated connection weights, such that the response of the winning neuron to the subsequent application of a similar input pattern is enhanced.
   (making the winning neuron look more like the observation)

Remark **8.8.** The SOM is an unsupervised system based on **competitive learning**.

- The output neurons compete amongst themselves to be activated, with the result that only one is activated at any one time.
- This activated neuron is called a **winner-takes-all neuron** or simply the **winning neuron**.
- Such competition can be implemented by having lateral inhibition connections (**negative feedback paths**) between the neurons.
- **The result**: The neurons are forced to organize themselves.

The Competitive Process

- **The input space** is d-dimensional (i.e. there are d input units).
- **The connection weights** between **the d input units** and **the kth output neuron** (in the computational layer) can be written

$$\mathbf{w}_k = \begin{bmatrix} w_{1k} \\ w_{2k} \\ \vdots \\ w_{dk} \end{bmatrix}, \quad k = 1, \cdots, K, \tag{8.18}$$

where K is the total number of neurons in the computational layer.

Definition 8.9. We can define the **discriminant function** to be the squared Euclidean distance between the input vector $\mathbf{x}$ and the weight vector $\mathbf{w}_k$ for each neuron k:

$$d_k(\mathbf{x}) = ||\mathbf{x} - \mathbf{w}_k||^2 = \sum_{i=1}^{d} (x_i - w_{ik})^2. \tag{8.19}$$

Thus, the neuron whose weight vector comes closest to the input vector (i.e. is most similar to it) is declared the winner; see Preliminary 8.7, item 2.

The Cooperative Process

In neurobiological studies, it is found that there is **lateral interaction** between a set of excited neurons.

- When one neuron fires, its closest neighbors tend to get excited more than those further away.
- There is a **topological neighbourhood** that decays with distance.

Definition 8.10. **Neighbourhood Function**

- Let us take K units on a regular lattice (string-like for 1D, or grid-like for 2D).
- If $\mathcal{K} = \{1, 2, \cdots, K\}$ and t is the time, a **neighborhood function $h(t)$** is defined on $\mathcal{K} \times \mathcal{K}$. It has to satisfy the following properties:
  (a) h is symmetric with $h_{kk} = 1$,
  (b) $h_{k\ell}$ depends only on the distance $\text{dist}(k, \ell)$ between units k and ℓ on the lattice, and
  (c) h decreases with increasing distance.
- **Several choices are possible for h.**
  - The most classical is the **step function**; equal to 1 if the distance between k and ℓ is less than a specific radius (this radius can decrease with time), and 0 otherwise.
  - Another very classical choice is a **Gaussian-shaped function**

$$h_{k\ell}(t) = \exp\Big(-\frac{\text{dist}^2(k, \ell)}{2\sigma^2(t)}\Big), \tag{8.20}$$

where $\sigma^2(t)$ **can decrease over time** to reduce the intensity and the scope of the neighborhood relations. A popular time dependence is an exponential decay:

$$\sigma(t) = \sigma_0 \exp(-t/\tau_\sigma). \tag{8.21}$$

The Adaptive Process

The SOM must involve an **adaptive (learning) process** by which

- the outputs become self-organized and
- the **feature map** between inputs and outputs is formed.

Learning Rule in the Adaptive Process

- The point of the **topographic neighborhood** is twofold:
  - The winning neuron gets its weights updated.
  - Its neighbors will have their weights updated as well, although by not as much as the winner itself.
- An appropriate **weight update rule** is formulated as

$$\Delta \mathbf{w}_k = \eta(t) \cdot h_{I(\mathbf{x}),k}(t) \cdot (\mathbf{x} - \mathbf{w}_k), \quad \forall\, k \in \mathcal{K}, \tag{8.22}$$

  where $I(\mathbf{x})$ is the index of the winning neuron and $\eta(t)$ is a **learning rate** ($0 < \eta(t) < 1$, constant or decreasing).
- **The effect of each weight update** is to move the weight vectors $\mathbf{w}_k$ of the winning neuron and its neighbors towards the input vector $\mathbf{x}$.
  - Repeated presentations of the training data thus leads to topological ordering.

Remark 8.11. The learning rule (8.22) has several properties:

- Maximal at the winning neuron.
- Symmetric about that neuron.
- Decreases monotonically to zero as the distance goes to infinity.
- Translation invariant (i.e., independent of the location of the winning neuron).

8.6.3. The SOM algorithm and its interpretation

The stages of the SOM can be summarized as follows.

Algorithm 8.12. The Stochastic SOM

- **Initialization**: A connection weight $\mathbf{w}_k \in \mathbb{R}^d$ is attached to each unit k, whose initial values are chosen at random and denoted by

$$W(0) = [\mathbf{w}_1(0), \mathbf{w}_2(0), \cdots, \mathbf{w}_K(0)].$$

- **For** $t = 0, 1, 2, \cdots$

  (a) **Sampling**: A data point $\mathbf{x}$ is **randomly drawn** (according to the density function f or from the finite set X)

  (b) **Matching**: The **best matching unit** is defined by

$$I(\mathbf{x}) = \arg\min_{k \in \mathcal{K}} ||\mathbf{x} - \mathbf{w}_k(t)||^2 \tag{8.23}$$

  (c) **Updating**: All the weights are updated via

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \Delta\mathbf{w}_k, \qquad \forall\, k \in \mathcal{K}, \tag{8.24}$$

  where, as defined in (8.22),

$$\Delta\mathbf{w}_k = \eta(t) \cdot h_{I(\mathbf{x}),k}(t) \cdot (\mathbf{x} - \mathbf{w}_k).$$

  (d) **Continuation**: Keep returning to the **sampling** step until the feature map stops changing.

Results of the SOM

- After learning, cluster C_k can be defined as the set of inputs closer to $\mathbf{w}_k$ than to any other one.
- The **Kohonen map** is the representation of
  - the weights or
  - the cluster contents,

  displayed according to the **neighborhood structure**.

Example: Data approximation

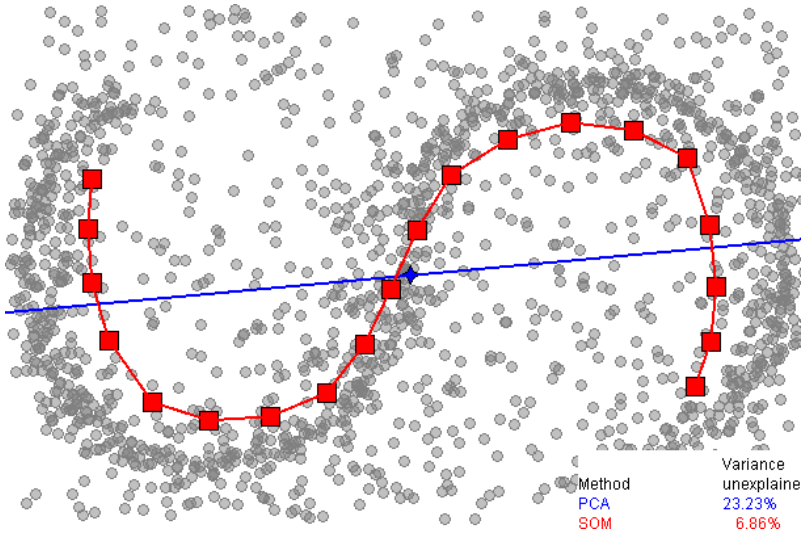


Figure 8.47: **Data approximation**: One-dimensional **SOM** vs. **PCA**.
SOM is a red broken line with squares, 20 nodes. The first principal component is presented by a blue line. Data points are the small gray circles. The fraction of **variance unexplained** in this example is **6.86% for SOM** and **23.23% for PCA**.
("Self-organizing map", Wikipedia)

Properties of the Kohonen Maps

- **The quantization property**: the weights represent the data space as accurately as possible, as do other quantization algorithms.
  - To get a better quantization, the learning rate $\eta(t)$ decreases with time as well as the scope of the neighborhood function h.
- **The self-organization property**, that means that the weights preserve the topology of the data:
  - **close inputs** belong to the **same cluster** (as do any clustering algorithms) or to **neighboring clusters**.

Theoretical Issues

- The algorithm is easy to define and to use, and a lot of practical studies confirm that it works.
  - However, the theoretical study of its convergence when t tends to ∞ remains without complete proof and provides open problems.
  - The main question is to know if the solution obtained from a finite sample converges to the true solution that might be obtained from the true data distribution.
- When t tends to ∞, the $\mathbb{R}^d$-valued stochastic processes $[\mathbf{w}_k(t)]_{k=1,2,\cdots,K}$ can present oscillations, explosion to infinity, convergence in distribution to an equilibrium process, convergence in distribution or almost sure to a finite set of points in $\mathbb{R}^d$, etc.. Some of the open questions are:
  - Is the algorithm convergent in distribution or almost surely, when t tends to ∞?
  - What happens when $\eta(t)$ is constant? (when it decreases?)
  - If a limit state exists, is it stable?
  - How to characterize the organization?

In Practice: Ordering and Convergence

Note: The SOM algorithm may start from an initial state of complete disorder, and it will gradually lead to an organized representation of activation patterns drawn from the input space.

There are two identifiable phases of **the adaptive process**:

1. **Ordering or Self-organizing phase** – during which the topological ordering of the weight vectors takes place.
   - Typically this will take as many as 1000 iterations of the SOM algorithm.
   - Careful consideration needs to be given to the choice of neighbourhood and learning rate parameters.
2. **Convergence phase** – during which the feature map is fine tuned and comes to provide an accurate statistical quantification of the input space.
   - Typically the number of iterations in this phase will be at least **500 times the number of neurons in the network**.
   - Again, the parameters must be chosen carefully.

Exercises for Chapter 8

8.1. We will experiment the K-Means algorithm following the first section of Chapter 11, *Python Machine Learning, 3rd Ed.*, in a little bit different fashion.

(a) Make a dataset of 4 clusters (modifying the code on pp. 354–355).

(b) For $K = 1, 2, \cdots, 10$, run the K-Means clustering algorithm with the initialization `init='k-means++'`.

(c) For each K, compute the within-cluster SSE (distortion) for an **elbow analysis** to select an appropriate K. **Note**: Rather than using `inertia_` attribute, **implement a function** for the computation of distortion.

(d) Produce **silhouette plots** for $K = 3, 4, 5, 6$.

8.2. Now, let's experiment DBSCAN, following *Python Machine Learning, 3rd Ed.*, pp. 376–381.

(a) Produce a dataset having **three** half-moon-shaped structures each of which consists of 100 samples.

(b) Compare performances of K-Means, AGNES, and DBSCAN.
(Set `n_clusters=3` for K-Means and AGNES.)

(c) For K-Means and AGNES, what if you choose `n_clusters` much larger than 3 (for example, 9, 12, 15)?

(d) Again, for K-Means and AGNES, perform an **elbow analysis** to select an appropriate K.

CHAPTER 9

Neural Networks and Deep Learning

Deep learning is a family of machine learning methods based on **learning data representations (features)**, as opposed to task-specific algorithms. Learning can be supervised, semi-supervised, or unsupervised [3, 5, 69].

Contents of Chapter 9

9.1. Basics for Deep Learning 270
9.2. Neural Networks 274
9.3. Back-Propagation 286
9.4. Deep Learning: Convolutional Neural Networks 293
Exercises for Chapter 9 303

9.1. Basics for Deep Learning

Conventional Machine Learning

- Limited in their ability to process data in their raw form
- **Feature!!**
  - **Coming up with features**:
    Difficult, time-consuming, requiring expert knowledge.
  - **Tuning the features**: We spend a lot of time, before and during learning.

$\longrightarrow$ | Feature Representation | $\longrightarrow$ | Learning Algorithm |

Examples of features: Histogram of oriented gradients (HOG), the **scale-invariant feature transform** (**SIFT**) (Lowe, 1999) [50], etc.

Representation Learning

- **Discover representations, automatically**
  $\Rightarrow$ The machine is fed with **raw data**
- **Deep Learning** methods are representation-learning methods with multiple levels of **representation/abstraction**
  - Simple non-linear modules $\Rightarrow$ higher and abstract representation
  - With the composition of enough such transformations, very complex functions can be learned.
- **Key Aspects**
  - **Layers of features** are not designed by human engineers.
  - **Learn features** from data using a general-purpose learning procedure.

Advances in Deep Learning

- Image recognition [23, 29, 41]
- Speech recognition [29, 66]
- Natural language understanding [23, 73, 81]
  - Machine translation
  - Image 2 text
  - Sentiment analysis
  - **Question-answering (QA) machine**:
    IBM's Watson, 2011, defeated legendary Jeopardy champions Brad Rutter and Ken Jennings, winning the first place prize of $1 million
- Many other domains
  - Predicting the activity of potential drug molecules
  - Analyzing particle accelerator data
  - Reconstructing brain circuits
  - Predicting the effects of mutations in non-coding DNA on gene expression and disease
- **Image-based Classifications**: Deep learning has provided breakthrough results in **speech recognition** and **image classification**.

Why/What about Deep Learning?

- **Why is it generally better than other methods** on image, speech, and certain other types of data? Short answers:
  - Deep learning means using a **neural network** with **several layers of nodes** between input and output
  - The series of layers between input & output do **feature identification and processing** in a series of stages, just as our brains seem to do.
- Multi-layer neural networks have been more than 30 years (Rina Dechter, 1986) [14]. **What is actually new?**
  - We have always had good algorithms for learning the weights in networks **with 1 hidden layer**.

    But these algorithms are not good at learning the weights for networks with more hidden layers
  - **The New** are: **methods for training many-layer networks**

Terms: AI vs. ML vs. Deep Learning

- **Artificial intelligence** (AI): Intelligence exhibited by machines
- **Machine learning** (ML): An approach to achieve AI
- **Deep learning** (DL): A technique for implementing ML
  - Feature/Representation-learning
  - Multi-layer neural networks (NN)
  - Back-propagation

    (*In the 1980s and 1990s, researchers did not have much luck, except for a few special architectures*.)
  - **New ideas** enable learning in deep NNs, since 2006

Back-propagation to Train Multi-layer Architectures

- Nothing more than a **practical application of the chain rule**, for derivatives
- Forsaken because poor **local minima**
- Revived around 2006 by unsupervised learning procedures with unlabeled data
  - **CIFAR** (Canadian Institute for Advanced Research): [4, 30, 31, 46]
  - Recognizing handwritten digits or detecting pedestrians
  - Speech recognition by **GPUs**, with 10 or 20 times faster (Raina *et al.*, 2009) [60] and (Bengio, 2013) [2].
  - Local minima become rarely a problem.
- **Convolutional neural network (CNN)**
  - Widely adopted by computer-vision community (LeCun *et al.*, 1989) [45]
- **Activations**
  - Non-linear functions: $\max(z, 0)$ (ReLU), $\tanh(z)$, $1/(1 + e^{-z})$

Machine Learning Challenges We've Yet to Overcome

- **Interpretability**: Although ML has come very far, researchers still **don't know exactly how deep training nets work**.
  - If we don't know how training nets actually work,
    $\Rightarrow$ **how do we make any real progress?**
- **One-Shot Learning**: We still haven't been able to achieve one-shot learning. ***Traditional networks*** **need a huge amount of data**, and are often in the form of **extensive iterative training**.
  - Instead, we should find a way to enable neural networks to learn, using just a few examples.
  - Current neural networks are **gradient-and-iteration**-based;
    $\Rightarrow$ **can we modify/replace it?**

9.2. Neural Networks

Recall: In 1957, Frank Rosenblatt invented the **perceptron** algorithm:

- For input values: $\mathbf{x} = (x_1, x_2, \cdots, x_d)^T$,
- Learn weight vector: $\mathbf{w} = (w_1, w_2, \cdots, w_d)^T$
- Get the net input $z = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d = \mathbf{w} \cdot \mathbf{x}$
- Classify, using the activation function

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise}, \end{cases} \qquad z = \mathbf{w} \cdot \mathbf{x}, \tag{9.1}$$

or, equivalently,

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise}, \end{cases} \qquad z = b + \mathbf{w} \cdot \mathbf{x}, \tag{9.2}$$

where $b = -\theta$ is the **bias**. (See (3.2) and (3.3), p. 46.)

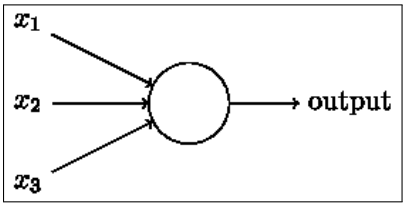


Figure 9.1: Perceptron: The simplest artificial neuron.

Perceptron is the simplest artificial neuron:

- It **makes decisions** by **weighting up evidence**.
- However, it is **not a complete model** for decision-making!

Complex Network of Perceptrons

- **Perceptron as a building block**:
  - What the example illustrates is **how a perceptron can weigh up** different kinds of evidence in order to make decisions.

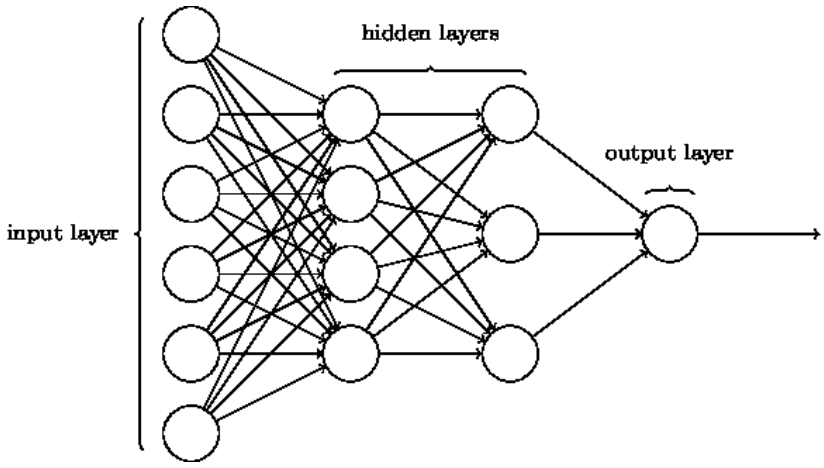


Figure 9.2: A complex network of perceptrons.

- It should seem plausible that **a complex network of perceptrons** could make quite subtle decisions.

An Issue on Perceptron Networks

- **Thresholding**. A small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to **completely flip**, say from -1 to 1.
  - That flip may **then cause the behavior of the rest of the network to completely change** in some very complicated way.
- We can overcome this problem by introducing a new type of artificial neuron, e.g., a **sigmoid neuron**.

9.2.1. Sigmoid neural networks

Recall: The **logistic sigmoid function** is defined as

$$\sigma(z) = \frac{1}{1+e^{-z}}. \tag{9.3}$$

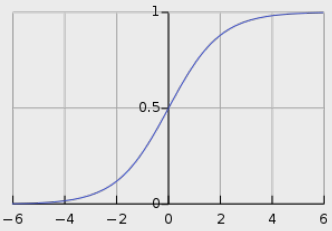


Figure 9.3: The standard logistic sigmoid function $\sigma(z) = 1/(1+e^{-z})$.

Sigmoid Neural Networks

- They are built with **sigmoid neurons**.
- The output of a sigmoid neuron with inputs x, weights w, and bias b is

$$\sigma(z) = \frac{1}{1+\exp(-b-\mathbf{w}\cdot\mathbf{x})}, \tag{9.4}$$

  which we considered as the **logistic regression** model in Section 5.2.
- Advantages of the sigmoid activation:
  - It allows **calculus** to design learning rules. ($\sigma' = \sigma(1-\sigma)$)
  - **Small changes in weights and bias** produce a **corresponding small change** in the output.

$$\Delta\text{output} \approx \sum_j \frac{\partial\,\text{output}}{\partial w_j}\Delta w_j + \frac{\partial\,\text{output}}{\partial b}\Delta b. \tag{9.5}$$

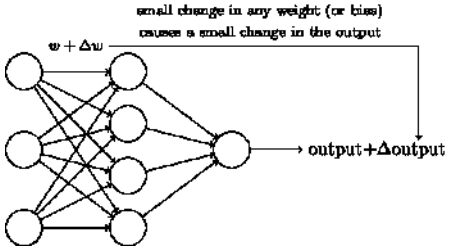


Figure 9.4: Δoutput is a linear combination of Δw_j and Δb.

The Architecture of (Smooth) Neural Networks

- The leftmost layer is called the **input layer**, and the neurons within the layer are called **input neurons**.
- The rightmost layer is the **output layer**.
- The middle layers are called **hidden layers**.

- **The design of the input and output layers in a network** is **often straightforward**. For example, for the classification of handwritten digits:
  - If the images are in 28×28 grayscale pixels, then we'd have $784 (= 28 \times 28)$ input neurons.
  - It is heuristic to set 10 neurons in the output layer. (rather than 4, where $2^4 = 16 \geq 10$)
- There can be **quite an art to the design of the hidden layers**.
  - In particular, **it is not possible to sum up the design process** for the hidden layers **with a few simple rules of thumb**.
  - Instead, neural networks researchers have developed **many design heuristics** for the hidden layers, which help people get the behavior they want out of their nets.
  - For example, **such heuristics** can be used to help determine how to trade off **the number of hidden layers** against **the accuracy and the time** required to train the network.

9.2.2. A simple network to classify handwritten digits

- The problem of recognizing handwritten digits has two components: **segmentation** and **classification**.

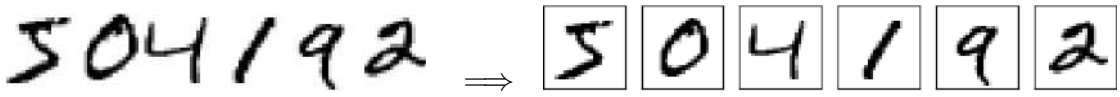


Figure 9.5: Segmentation.

- We'll focus on algorithmic components for the classification of individual digits.

MNIST Dataset:

A modified subset of two datasets collected by NIST (US National Institute of Standards and Technology):

- The first part contains 60,000 images (for training)
- The second part is 10,000 images (for test)

Each image is in 28×28 **grayscale pixels**.

A Simple Feed-forward Network

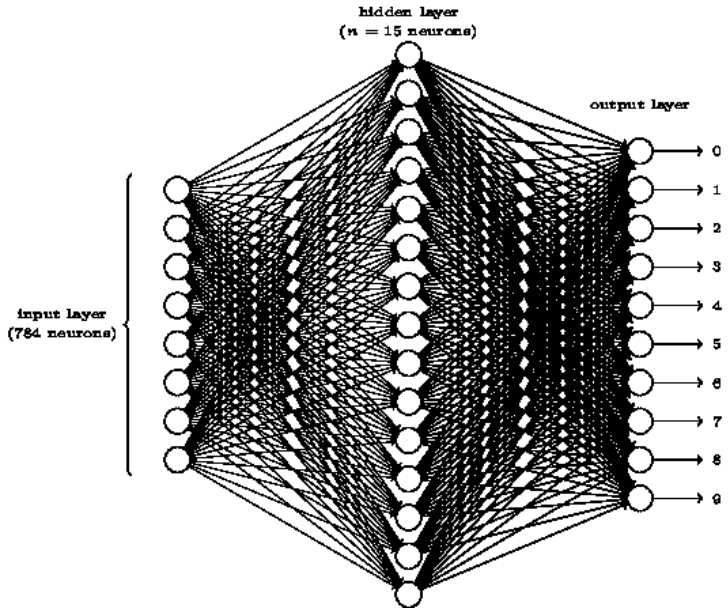
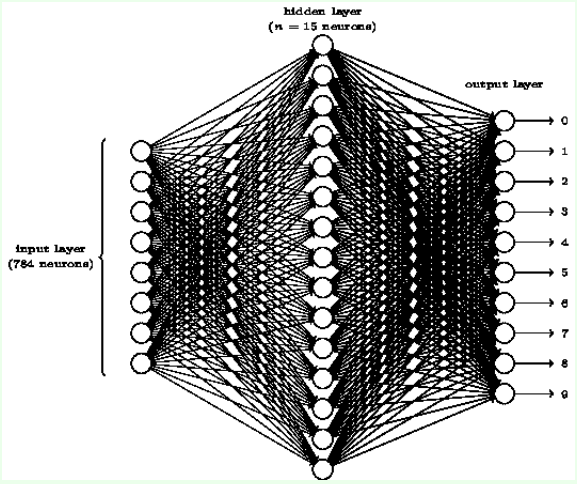


Figure 9.6: A sigmoid network having a hidden layer.

What the Neural Network Will Do

- Let's concentrate on **the first output neuron**, the one that is trying to decide whether or not the input digit is a **0**.
- It does this by **weighing up evidence** from the hidden layer of neurons.



- **What are those hidden neurons doing?**
  - Let's suppose **for the sake of argument** that **the first neuron** in the hidden layer may detect whether or not an image like the following is present

    It can do this by **heavily weighting input pixels** which overlap with the image, and only lightly weighting the other inputs.
  - Similarly, let's suppose that **the second, third, and fourth neurons** in the hidden layer detect whether or not the following images are present

  - As you may have guessed, these four images together make up the 0 image that we saw in the line of digits shown in Figure 9.5:

  - So **if all four of these hidden neurons are firing**, then we can conclude that the digit is a 0.

Learning with Gradient Descent

- **Dataset**: $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}$, $i = 1, 2, \cdots, N$
  - $\mathbf{y}^{(i)}$? For example, if an image $\mathbf{x}^{(i)}$ depicts a 2, then
$$\mathbf{y}^{(i)} = (0, 0, 1, 0, 0, 0, 0, 0, 0, 0)^T.$$
- **Cost function**
$$C(\boldsymbol{W}, B) = \frac{1}{2N} \sum_i ||\mathbf{y}^{(i)} - \mathbf{a}(\mathbf{x}^{(i)})||^2, \tag{9.6}$$
where $\boldsymbol{W}$ denotes the collection of all weights in the network, B all the biases, and $\mathbf{a}(\mathbf{x}^{(i)})$ is the vector of outputs from the network when $\mathbf{x}^{(i)}$ is input.
- **Gradient descent method**
$$\begin{bmatrix} \boldsymbol{W} \\ B \end{bmatrix} \leftarrow \begin{bmatrix} \boldsymbol{W} \\ B \end{bmatrix} + \begin{bmatrix} \Delta\boldsymbol{W} \\ \Delta B \end{bmatrix}, \tag{9.7}$$
where
$$\begin{bmatrix} \Delta\boldsymbol{W} \\ \Delta B \end{bmatrix} = -\eta \begin{bmatrix} \nabla_{\boldsymbol{W}} C \\ \nabla_B C \end{bmatrix}.$$

Note: To compute the gradient ∇C, we need to compute the gradients $\nabla C_{\mathbf{x}^{(i)}}$ separately for each training input, $\mathbf{x}^{(i)}$, and then average them:
$$\nabla C = \frac{1}{N} \sum_i \nabla C_{\mathbf{x}^{(i)}}. \tag{9.8}$$

- Unfortunately, when the number of training inputs is very large, it can take a long time, and learning thus occurs slowly.
- An idea called **stochastic gradient descent** can be used to speed up learning.

Stochastic Gradient Descent

The idea is to estimate the gradient ∇C by computing $\nabla C_{\mathbf{x}^{(i)}}$ for a **small sample of randomly chosen training inputs**. By averaging over this small sample, it turns out that we can quickly get a good estimate of the true gradient ∇C; this helps speed up gradient descent, and thus learning.

- Pick out a small number of randomly chosen training inputs ($m \ll N$):

$$\widetilde{\mathbf{x}}^{(1)}, \widetilde{\mathbf{x}}^{(2)}, \cdots, \widetilde{\mathbf{x}}^{(m)},$$

  which we refer to as a **mini-batch**.
- Average $\nabla C_{\widetilde{\mathbf{x}}^{(k)}}$ to approximate the gradient ∇C. That is,

$$\frac{1}{m}\sum_{k=1}^{m} \nabla C_{\widetilde{\mathbf{x}}^{(k)}} \approx \nabla C \overset{\text{def}}{=\!=} \frac{1}{N}\sum_{i} \nabla C_{\mathbf{x}^{(i)}}. \tag{9.9}$$

- For classification of handwritten digits for the MNIST dataset, you may choose: `batch_size = 10`.

Note: In practice, you can implement the stochastic gradient descent as follows. **For an epoch**,

- Shuffle the dataset
- For each m samples (selected from the beginning), update $(\boldsymbol{W}, B)$ using the approximate gradient (9.9).

Implementing a Network to Classify Digits [56]

network.py

```
"""
network.py       (by Michael Nielsen)
~~~~~~~~~~
A module to implement the stochastic gradient descent learning
algorithm for a feedforward neural network.  Gradients are calculated
using backpropagation. """
#### Libraries
# Standard library
import random
# Third-party libraries
import numpy as np

class Network(object):
    def __init__(self, sizes):
        """The list ``sizes`` contains the number of neurons in the
        respective layers of the network.  For example, if the list
        was [2, 3, 1] then it would be a three-layer network, with the
        first layer containing 2 neurons, the second layer 3 neurons,
        and the third layer 1 neuron. """

        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The ``training_data`` is a list of tuples
        ``(x, y)`` representing the training inputs and the desired
        outputs.  """

        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
```

```
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)

    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] #list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())

```

```
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def evaluate(self, test_data):
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives \partial C_x /
        \partial a for the output activations."""
        return (output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))
```

The code is executed using

Run_network.py

```
import mnist_loader
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()

import network
n_neurons = 20
net = network.Network([784 , n_neurons, 10])

n_epochs, batch_size, eta = 30, 10, 3.0
net.SGD(training_data , n_epochs, batch_size, eta, test_data = test_data)
```

```
len(training_data)=50000, len(validation_data)=10000, len(test_data)=10000
```

Validation Accuracy

Validation Accuracy

```
Epoch 0: 9006 / 10000
Epoch 1: 9128 / 10000
Epoch 2: 9202 / 10000
Epoch 3: 9188 / 10000
Epoch 4: 9249 / 10000
 ...
Epoch 25: 9356 / 10000
Epoch 26: 9388 / 10000
Epoch 27: 9407 / 10000
Epoch 28: 9410 / 10000
Epoch 29: 9428 / 10000
```

Accuracy Comparisons

- scikit-learn's SVM classifier using the default settings: 9435/10000
- A well-tuned SVM: ≈98.5%
- Well-designed (convolutional) NN: 9979/10000 **(only 21 missed!)**

Note: For **well-designed neural networks**, the performance is close to **human-equivalent**, and is **arguably better**, since quite a few of the MNIST images are difficult even for humans to recognize with confidence, e.g.,

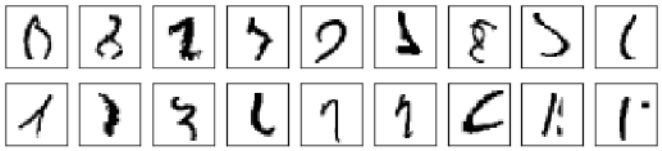


Figure 9.7: MNIST images difficult even for humans to recognize.

Moral of the Neural Networks

- Let all the complexity be learned, automatically, from data
- Simple algorithms can perform well for some problems:

(sophisticated algorithm) $\leq$ **(simple learning algorithm + good training data)**

9.3. Back-Propagation

- **In the previous section**, we saw an example of neural networks that could learn their weights and biases using the stochastic gradient descent algorithm.
- **In this section**, we will see **how to compute the gradient**, more precisely, **the derivatives of the cost function** with respect to weights and biases **in all layers**.
- ***The back-propagation is a practical application of the chain rule*** for the computation of derivatives.
- The back-propagation algorithm was originally introduced in the 1970s, but its importance was not fully appreciated until a famous 1986 paper by Rumelhart-Hinton-Williams [65], in *Nature*.

9.3.1. Notations

- Let's begin with notations which let us refer to weights, biases, and activations in the network in an unambiguous way.

w^ℓ_{jk}:	the **weight** for the connection from the k-th neuron in the $(\ell-1)$-th layer to the j-th neuron in the ℓ-th layer
b^ℓ_j:	the **bias** of the j-th neuron in the ℓ-th layer
a^ℓ_j:	the **activation** of the j-th neuron in the ℓ-th layer

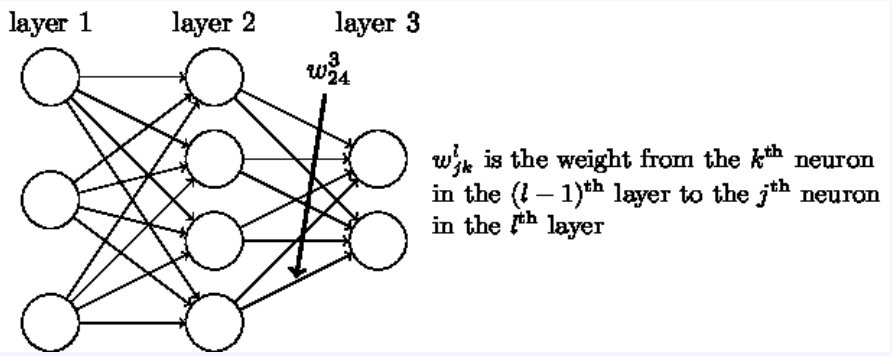


Figure 9.8: The weight w^ℓ_{jk}.

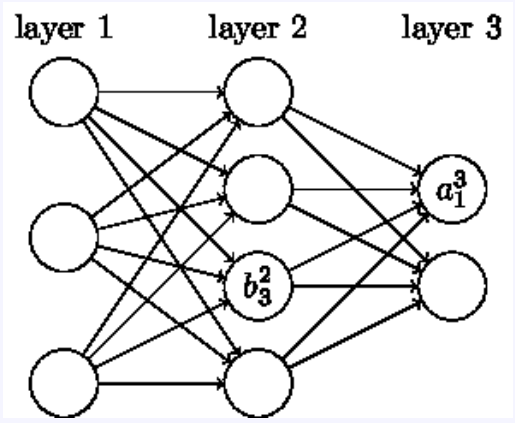


Figure 9.9: The bias b_j^ℓ and activation a_j^ℓ.

- With these notations, the activation a_j^ℓ reads

$$a_j^\ell = \sigma\Big(\sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell\Big), \tag{9.10}$$

  where the sum is over all neurons k in the $(\ell-1)$-th layer. Denote the **weighted input** by

$$z_j^\ell := \sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell. \tag{9.11}$$

- Now, define

$$\begin{array}{lcl} W^\ell = [w_{jk}^\ell] & : & \text{the weight matrix for layer } \ell \\ \mathbf{b}^\ell = [b_j^\ell] & : & \text{the bias vector for layer } \ell \\ \mathbf{z}^\ell = [z_j^\ell] & : & \text{the weighted input vector for layer } \ell \\ \mathbf{a}^\ell = [a_j^\ell] & : & \text{the activation vector for layer } \ell \end{array} \tag{9.12}$$

- Then, (9.10) can be rewritten (in a vector form) as

$$\mathbf{a}^\ell = \sigma(\mathbf{z}^\ell) = \sigma(W^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell). \tag{9.13}$$

9.3.2. The cost function

The Cost Function: With the notations, the quadratic cost function (9.6) has the form

$$C = \frac{1}{2N} \sum_{\mathbf{x}} ||\mathbf{y}(\mathbf{x}) - \mathbf{a}^L(\mathbf{x})||^2, \tag{9.14}$$

where N is the total number of training examples, $\mathbf{y}(\mathbf{x})$ is the corresponding desired output for the training example $\mathbf{x}$, and L denotes the number of layers in the network.

Two Assumptions for the Cost Function

1. The cost function can be written as an average

$$C = \frac{1}{N} \sum_{\mathbf{x}} C_{\mathbf{x}}, \tag{9.15}$$

   over cost functions $C_{\mathbf{x}}$ for individual training examples $\mathbf{x}$.
2. The cost function can be written as a function of the outputs from the neural network ($\mathbf{a}^L$).

Remark 9.1. Thus the cost function in (9.14) satisfies the assumptions, with

$$C_{\mathbf{x}} = \frac{1}{2}||\mathbf{y}(\mathbf{x}) - \mathbf{a}^L(\mathbf{x})||^2 = \frac{1}{2} \sum_j (y_j(\mathbf{x}) - a_j^L(\mathbf{x}))^2. \tag{9.16}$$

- The reason we need the first assumption is because what the back-propagation actually lets us do is **compute the partial derivatives $\partial C_{\mathbf{x}}/\partial w_{jk}^{\ell}$ and $\partial C_{\mathbf{x}}/\partial b_j^{\ell}$ for a single training example**.
  - We then **can recover $\partial C/\partial w_{jk}^{\ell}$ and $\partial C/\partial b_j^{\ell}$ by averaging over training examples**.
- With the assumptions in mind, we may **focus on computing the partial derivatives for a single example**.

9.3.3. The four fundamental equations behind the back-propagation

The back-propagation is about understanding *how* **changing the weights and biases in a network changes the cost function**, which means computing the partial derivatives $\partial C/\partial w_{jk}^{\ell}$ and $\partial C/\partial b_j^{\ell}$.

Definition 9.2. *Define the **learning error** (or, **error**) of neuron j in layer ℓ by*

$$\delta_j^{\ell} \stackrel{\text{def}}{=\!=} \frac{\partial C}{\partial z_j^{\ell}}. \tag{9.17}$$

The back-propagation will give us a way of computing $\boldsymbol{\delta}^{\ell} = [\delta_j^{\ell}]$ for every layer ℓ, and then relating those errors to the quantities of real interest, $\partial C/\partial w_{jk}^{\ell}$ and $\partial C/\partial b_j^{\ell}$.

Theorem 9.3. *Suppose that the cost function C satisfies the two assumptions in Section 9.3.2 so that it represents the cost for a single training example. Assume the network contains L layers, of which the feed-forward model is given as in (9.10):*

$$a_j^{\ell} = \sigma(z_j^{\ell}), \quad z_j^{\ell} = \sum_k w_{jk}^{\ell} a_k^{\ell-1} + b_j^{\ell}; \quad \ell = 2, 3, \cdots, L.$$

Then,

$$\begin{aligned}
&\text{(a)} \quad \delta_j^L = \frac{\partial C}{\partial a_j^L}\, \sigma'(z_j^L), \\
&\text{(b)} \quad \delta_j^{\ell} = \sum_k w_{kj}^{\ell+1} \delta_k^{\ell+1} \sigma'(z_j^{\ell}), \quad \ell = L-1, \cdots, 2, \\
&\text{(c)} \quad \frac{\partial C}{\partial b_j^{\ell}} = \delta_j^{\ell}, \qquad\qquad \ell = 2, \cdots, L, \\
&\text{(d)} \quad \frac{\partial C}{\partial w_{jk}^{\ell}} = a_k^{\ell-1} \delta_j^{\ell}, \qquad\qquad \ell = 2, \cdots, L.
\end{aligned} \tag{9.18}$$

Proof. Here, we will prove (b) only; see Exercise 1 for the others. Using the definition (9.17) and the chain rule, we have

$$\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell} = \sum_k \frac{\partial C}{\partial z_k^{\ell+1}} \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_k \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} \delta_k^{\ell+1} \tag{9.19}$$

Note

$$z_k^{\ell+1} = \sum_i w_{ki}^{\ell+1} a_i^\ell + b_k^{\ell+1} = \sum_i w_{ki}^{\ell+1} \sigma(z_i^\ell) + b_k^{\ell+1}. \tag{9.20}$$

Differentiating it, we obtain

$$\frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_i w_{ki}^{\ell+1} \frac{\partial \sigma(z_i^\ell)}{\partial z_j^\ell} = w_{kj}^{\ell+1} \sigma'(z_j^\ell). \tag{9.21}$$

Substituting back into (9.19), we complete the proof. □

The Hadamard product / Schur product

Definition 9.4. A frequently used algebraic operation is the **element-wise product** of two vectors/matrices, which is called the **Hadamard product** or the **Schur product**, and defined as

$$(\mathbf{c} \odot \mathbf{d})_j = c_j \, d_j. \tag{9.22}$$

- For example,

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 3 \\ 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}. \tag{9.23}$$

- In `Numpy`, `A*B` denotes the Hadamard product of `A` and `B`, while `A.dot(B)` or `A@B` produces the regular matrix-matrix multiplication.

Remark 9.5. The four fundamental equations (9.18) can be written in a vector form as

$$\begin{aligned}
&\text{(a)} \quad \boldsymbol{\delta}^L = \nabla_{\mathbf{a}^L} C \odot \sigma'(\mathbf{z}^L), \\
&\text{(b)} \quad \boldsymbol{\delta}^\ell = ((W^{\ell+1})^T \boldsymbol{\delta}^{\ell+1}) \odot \sigma'(\mathbf{z}^\ell), \quad \ell = L-1, \cdots, 2, \\
&\text{(c)} \quad \nabla_{\mathbf{b}^\ell} C = \boldsymbol{\delta}^\ell, \qquad\qquad\qquad\quad \ell = 2, \cdots, L, \\
&\text{(d)} \quad \nabla_{W^\ell} C = \boldsymbol{\delta}^\ell (\mathbf{a}^{\ell-1})^T, \qquad\quad \ell = 2, \cdots, L.
\end{aligned} \tag{9.24}$$