
IT496: Introduction to Data Mining



Lecture - 01

Introduction

[Definition, Tasks, and Case Studies]

Arpit Rana
25th July 2023

Course Logistics

Syllabus and Evaluation Scheme

Course Logistics

Instructor	Arpit Rana Room-3105, Faculty Block-3 Email: arpit_rana@daiict.ac.in
TA Contact Info	Himanshu Beniwal (himanshubeniwal@iitgn.ac.in) - Head TA Dhara Shah (202211008@daiict.ac.in) Naiya Patel (202211075@daiict.ac.in)
Prerequisites	Programming in Python, Linear Algebra, Probability and Statistics
Eligibility	<ul style="list-style-type: none">• B.Tech. V / VII Semester (ICT/MnC)• M.Tech. ICT - I Semester (ML/SS specialization)• M.Sc. DS - III Semester

Course Logistics

Credit Weighting	3-0-2-4 (L-T-P-Cr)
Lectures [CEP-108]	Tuesday: 10:00 – 11:00, Thursday, Friday: 11:00 – 12:00
Lab/Tutorial [M.Sc. DS Lab]	Friday, 14:00 – 16:00
Private Study	At least 5 hrs per week
Potential Outcome	<ul style="list-style-type: none">• Learn how to solve Data-driven Decision-Making Problems;• Learn how to work on structured and unstructured (e.g., text, image, sequential) data;• Targeted Jobs: <i>Data Analyst, Data Engineer, Data Scientist, ML Engineer, Research Engineer</i>

Course Logistics

Assessment	<ul style="list-style-type: none">● Surprise Quizzes: 25%● End Term: 25%● Course Projects: 40% (10% + 15% + 15%)● Case Study: 10% <p><u>Extra Credits:</u> ML Challenges, Participate on Course Stream</p>
How to Fail	Skip lectures; avoid private study; cram just before the exam; expect the exam to be a memory test; copy project assignments; be inactive on the course stream
How to Pass	Attend lectures; summarize the notes; expect a problem-solving exam; do your project yourself; <u>be active and accurate in the class and on the course stream</u>

Preliminary Schedule

Week	Lecture	Lab	Due ¹
Week-1 [24 July 2023]	Introduction	- No lab -	-
Week-2 [31 July 2023]	Statistics for Data Mining	- No lab -	-
Week-3 [07 Aug 2023]	Data Preprocessing	End-to-End ML Project in Python	-
Week-4 [14 Aug 2023]	Fundamentals of Predictive Analytics - I <i>Holidays: 15 Aug (Tues), 16 Aug (Wed)</i>	CP - 1	Sunday, 10 Sept. 2023
Week-5 [21 Aug 2023]	Fundamentals of Predictive Analytics – II		
Week-6 [28 Aug 2023]	Regression Techniques <i>Holidays: 30 Aug (Wed) 28 Aug (Mon) to be treated as Tues</i>		
Week-7 [04 Sept 2023]	Dimensionality Reduction <i>Holidays: 07 Sept (Thurs)</i>		
Week-8 [11 Sept 2023]	First In-Semester Exam Week	Evaluation: CP - 1	

Preliminary Schedule

Week	Lecture	Lab	Due ¹
Week-9 [18 Sept 2023]	Eager Classifiers – I: Support Vector Machines and Decision Trees <i>Holidays: 19 Sept (Tues)</i>		
Week-10 [25 Sept 2023]	Eager Classifiers – II: Neural Networks <i>Holidays: 28 Sept (Thurs) 29 Sept (Fri) to be treated as Thurs</i>	CP - 2	Sunday, 15 Oct 2023
Week-11 [02 Oct 2023]	Eager Classifiers – III: Neural Networks Contd. <i>Holidays: 02 Oct (Mon)</i>		
Week-12 [09 Oct 2023]	Lazy Classifiers and Ensemble Techniques		
Week-13 [16 Oct 2023]	Second In-Semester Exam Week	Evaluation: CP - 2	

Preliminary Schedule

Week	Lecture	Lab	Due ¹
Week-14 [23 Oct 2023]	Cluster Analysis – I Holidays: 24 Oct (Tues)		
Week-15 [30 Oct 2023]	Cluster Analysis – II Holidays: 31 Oct (Tues) 03 Nov (Fri) to be treated as Tues	CP - 3	Sunday, 19 Nov 2023
Week-16 [6 Nov 2023]	Outlier Analysis		
Week-17 [13 Nov 2023]	In-Semester Break		
Week-18 [20 Nov 2023]	Association Rule Mining	Evaluation: CP - 3	
Week-19 [27 Nov 2023]	End-semester Examination		

1 – Course Projects (CPs) are due at 11:59 PM on the due date listed.

Preliminary Schedule

Lab Projects and Case Study:

- Projects will be allocated to groups, with **each group consisting of four members**.
- The assigned group will be responsible for delivering a **case study** on an AI-focused startup that addresses issues of the nation's priority and global interest.
- Further instructions related to the labs and case study presentations will be provided later on the classroom portal.

Introduction

Definition and Tasks

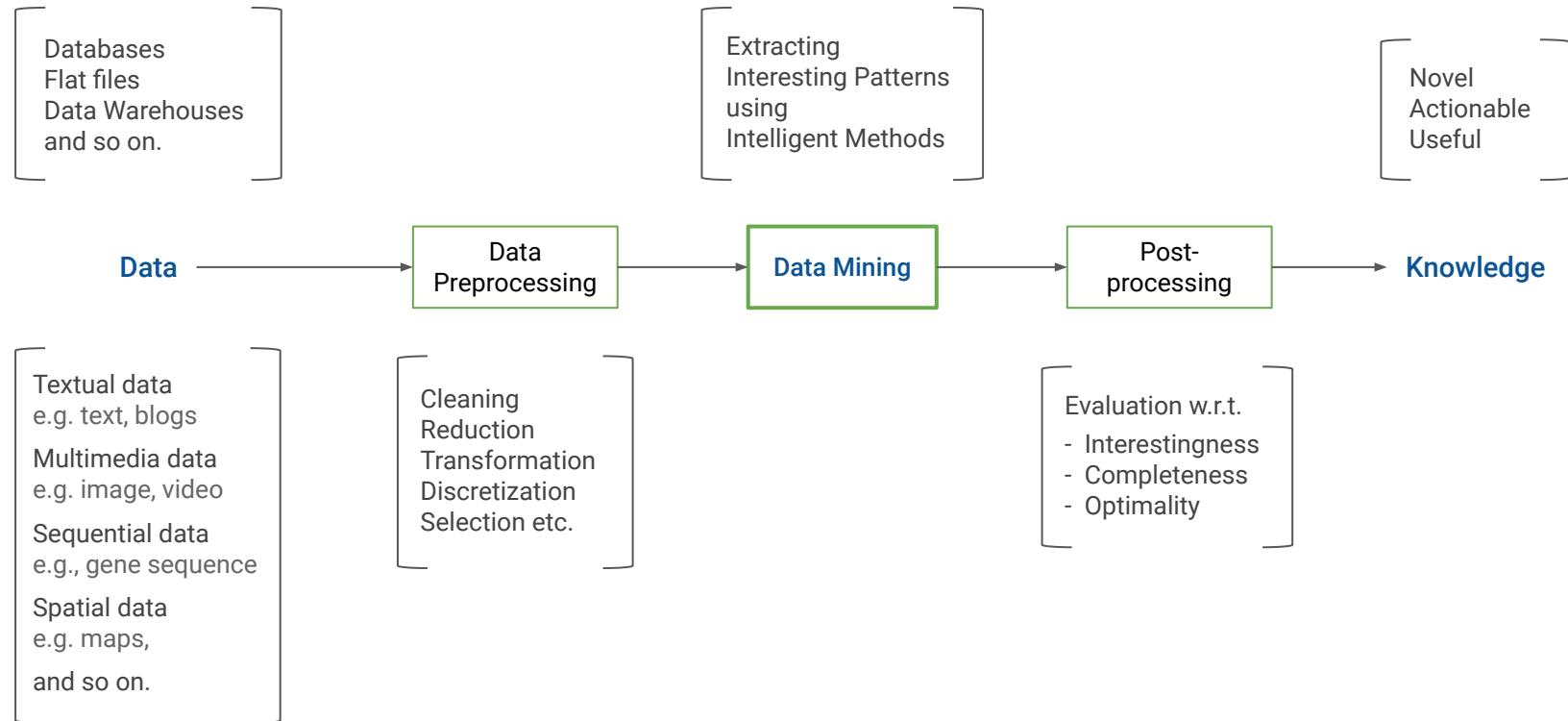
What is Data (Knowledge) Mining?

The process of automatically (or semi-automatically) discovering *interesting patterns* from large amounts of data.

- Implicit (somewhat hidden),
- Non-trivial (not obvious),
- Previously unknown (novel), and
- Potentially useful (for consumers / sellers / stakeholders)



Data (Knowledge) Mining: Knowledge Discovery from Data

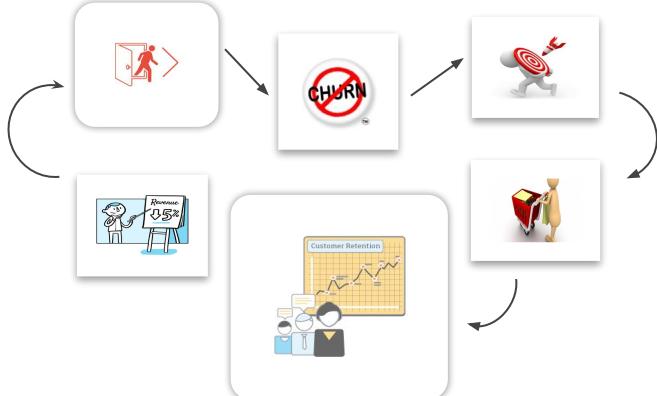


Data Mining vs. Machine Learning

The process of automatically (or semi-automatically) discovering *interesting patterns* from large amounts of data.

It uses methods at the intersection of *machine learning*, statistics, and *database systems*.

E.g., customer churn



Machine learning (ML) is focused on understanding and building methods that 'learn'.

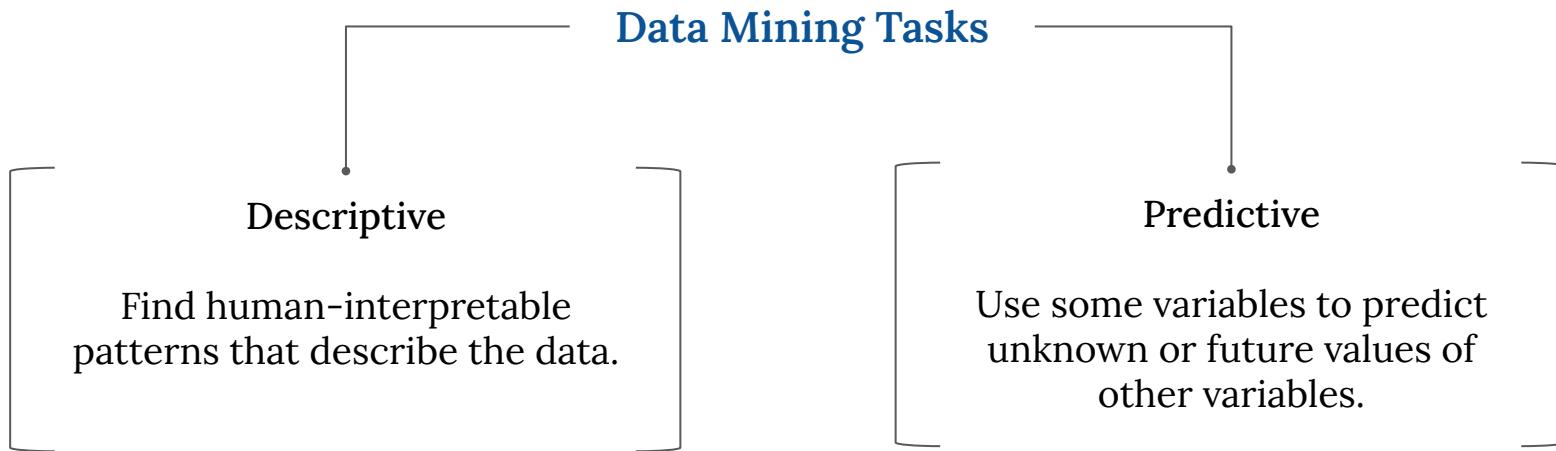
It leverages data to improve performance on some set of tasks.

E.g.: A spam filter (an ML program)



Data Mining Tasks

The actual data mining task is the semi-automatic or automatic analysis of large quantities of data to extract interesting patterns.



Descriptive Tasks

- **Cluster Analysis** (groups of data records),
 - Market Segmentation
 - Document clustering
- **Anomaly Detection | Outlier Analysis** (unusual records), and
 - Credit card fraud detection
 - Stock market manipulation detection
- **Association Rule Mining**, Sequential pattern mining (dependencies)
 - Market-basket analysis for sales promotion, shelf and inventory management
 - Medical informatics to find combination of patient symptoms and test results associated with certain diseases

Predictive Tasks

- **Classification** (predicting the class of a record)
 - Categorizing news stories as finance, weather, entertainment, sports, etc
 - Classifying land covers (water bodies, urban areas, forests, etc.) using satellite image data
- **Regression** (predicting the value of a variable of a record)
 - Predicting sales amounts of new product based on advertising expenditure.
 - Predicting wind velocities as a function of temperature, humidity, air pressure, etc.
 - Time series prediction of stock market indices.

Business Case Studies

Fakespot, GoKwik, and Intello Labs

Case Study - I: Fakespot

Problem Identified

- Nearly 93% consumers read reviews before any kind of purchasing decision.
- Out of these, around 91% of 18-34 year olds trust reviews as much as a recommendation from a friend!
- Over 30% of reviews are found to be *fake*.

Target Audience

All e-commerce businesses that allow users to write reviews.

Data-driven Solution

Fakespot reports provide an Adjusted Rating that weighs reviews based on authenticity and then recalculates it.



Courtesy: Fakespot

Case Study - II: GoKwik

Problem Identified

- In e-commerce, more than 30% of orders are returned to origin (RTO, i.e. shipped back to the warehouse) in India.

Target Audience

All e-commerce businesses

Data-driven Solution

- Mostly, CoD orders are converted to RTO.
- So, analyzing customer behavioural patterns and disable CoD option for those showing high-risk RTO behaviour.



Courtesy: Gokwik

Case Study - III: Intello Labs

Problem Identified

- One-third of the food produced in the world for human consumption every year gets lost or wasted.
- Mainly (in some countries) at the early stages of the food value chain.

Target Audience

From growers to packers, from exporters to food services

Data-driven Solution

Smart, scalable solutions to *digitize food quality*, achieve fair pricing and reduce food wastage.



Best delivered today

Best delivered in 2 days

Best delivered in 5 days

Using AI, ML, and Computer Vision technology

Next lecture

Statistics for Data Mining

27th July 2023

IT496: Introduction to Data Mining



Lecture - 02

Statistics for Data Mining - I

[Attribute Types and Measures of Central Tendency]

Arpit Rana
27th July 2023

Attribute Types

Terminology of Structured Data

House Rent Prediction Dataset (from Magicbricks, India)

	Posted On	BHK	Rent	Size	Floor	Area Type	Area Locality	City	Furnishing Status	Tenant Preferred	Bathroom	Point of Contact
311	2022-06-03	1	9000	450	Ground out of 3	Carpet Area	Salt Lake City Sector 5	Kolkata	Unfurnished	Bachelors/Family	1	Contact Agent
3869	2022-05-20	3	19500	1270	1 out of 2	Super Area	Madipakkam	Chennai	Semi-Furnished	Bachelors	2	Contact Owner
1368	2022-06-21	1	20000	310	Ground out of 7	Carpet Area	Malad West	Mumbai	Unfurnished	Bachelors	1	Contact Agent
1528	2022-06-13	2	16000	600	1 out of 2	Carpet Area	Girinagar	Bangalore	Unfurnished	Bachelors	2	Contact Owner
309	2022-06-25	3	13000	950	Ground out of 2	Carpet Area	Rabindrapally, Garia	Kolkata	Unfurnished	Bachelors/Family	2	Contact Owner

Dataset Glossary (Column-Wise): 4746 Records

- **BHK:** Number of Bedrooms, Hall, Kitchen.
- **Floor:** Ground out of 2, 3 out of 5, etc.
- **Size:** Size of property in Square Feet.
- **Area Type:** Super Area/Carpet Area/Built Area.
- **Furnishing Status:** Furnished/Semi-Furnished/Unfurnished.
- **Bathroom:** Number of Bathrooms.
- **Area Locality:** Locality of the property
- **City:** City where the property is Located.
- **Tenant Preferred:** Family/Bachelor
- **Point of Contact:** Agent / Owner
- **Rent:** Price of the property

Data Objects

A data object represents an *entity* (i.e. a row in a database).

- Customers, store items in a sales database
- Professors, students, and courses in a university database
- Patients in a medical database

Data objects are also referred to as *samples*, *examples*, *instances*, *data points*, *domain points* or *simply objects*.

	Posted On	BHK	Rent	Size	Floor	Area Type	Area Locality	City	Furnishing Status	Tenant Preferred	Bathroom	Point of Contact
311	2022-06-03	1	9000	450	Ground out of 3	Carpet Area	Salt Lake City Sector 5	Kolkata	Unfurnished	Bachelors/Family	1	Contact Agent
3869	2022-05-20	3	19500	1270	1 out of 2	Super Area	Madipakkam	Chennai	Semi-Furnished	Bachelors	2	Contact Owner
1368	2022-06-21	1	20000	310	Ground out of 7	Carpet Area	Malad West	Mumbai	Unfurnished	Bachelors	1	Contact Agent
1528	2022-06-13	2	16000	600	1 out of 2	Carpet Area	Girinagar	Bangalore	Unfurnished	Bachelors	2	Contact Owner
309	2022-06-25	3	13000	950	Ground out of 2	Carpet Area	Rabindrapally, Garia	Kolkata	Unfurnished	Bachelors/Family	2	Contact Owner

Attributes

An attribute is a data field *representing a characteristic or a feature* of a data object.

- A customer object → customer_ID, name, address, age, and gender
- A course object → course_ID, credit_structure, slot, and instructor

Attributes are also referred to as *dimensions, features, and variables*.

- A set of attributes used to describe a given object is called an *attribute vector*.

	Posted On	BHK	Rent	Size	Floor	Area Type	Area Locality	City	Furnishing Status	Tenant Preferred	Bathroom	Point of Contact
311	2022-06-03	1	9000	450	Ground out of 3	Carpet Area	Salt Lake City Sector 5	Kolkata	Unfurnished	Bachelors/Family	1	Contact Agent
3869	2022-05-20	3	19500	1270	1 out of 2	Super Area	Madipakkam	Chennai	Semi-Furnished	Bachelors	2	Contact Owner
1368	2022-06-21	1	20000	310	Ground out of 7	Carpet Area	Malad West	Mumbai	Unfurnished	Bachelors	1	Contact Agent
1528	2022-06-13	2	16000	600	1 out of 2	Carpet Area	Girinagar	Bangalore	Unfurnished	Bachelors	2	Contact Owner
309	2022-06-25	3	13000	950	Ground out of 2	Carpet Area	Rabindrapally, Garia	Kolkata	Unfurnished	Bachelors/Family	2	Contact Owner

Attribute Types

The type of an attribute is determined by the set of possible values it can have. They can be categorized in two ways:

Attribute Types

Discrete

An attribute may have a finite (e.g., *hair_color*) or countably infinite (e.g., *zip_code*) set of values.

Continuous

If an attribute is not discrete, it is continuous; typically represented as floating-point values (e.g., *rent*).

Qualitative

An attribute does not have an actual size or quantity; however, typically have words representing categories (e.g., *furnishing_status*).

Quantitative

These provide quantitative measurement of an object (e.g., *size*).

Nominal Attributes (a.k.a. categorical) | Qualitative | Discrete

Its values can be *symbols or names of things* representing a category, code, or state.

- *hair_color* (black, brown, blond, etc.)
- *marital_status* (single, married, divorced, and widowed)
- *occupation* (doctor, programmer, teacher, etc.)

	Posted On	BHK	Rent	Size	Floor	Area Type	Area Locality	City	Furnishing Status	Tenant Preferred	Bathroom	Point of Contact
311	2022-06-03	1	9000	450	Ground out of 3	Carpet Area	Salt Lake City Sector 5	Kolkata	Unfurnished	Bachelors/Family	1	Contact Agent
3869	2022-05-20	3	19500	1270	1 out of 2	Super Area	Madipakkam	Chennai	Semi-Furnished	Bachelors	2	Contact Owner
1368	2022-06-21	1	20000	310	Ground out of 7	Carpet Area	Malad West	Mumbai	Unfurnished	Bachelors	1	Contact Agent
1528	2022-06-13	2	16000	600	1 out of 2	Carpet Area	Girinagar	Bangalore	Unfurnished	Bachelors	2	Contact Owner
309	2022-06-25	3	13000	950	Ground out of 2	Carpet Area	Rabindrapally, Garia	Kolkata	Unfurnished	Bachelors/Family	2	Contact Owner

Binary Attributes | Qualitative | Discrete

These are nominal attributes with *only two categories or states* (0 or 1 / true or false).

- *gender* (make or female | *symmetric* i.e. both the outcomes are equally important)
- *medical_test* (positive or negative | *asymmetric*)

	Posted On	BHK	Rent	Size	Floor	Area Type	Area Locality	City	Furnishing Status	Tenant Preferred	Bathroom	Point of Contact
311	2022-06-03	1	9000	450	Ground out of 3	Carpet Area	Salt Lake City Sector 5	Kolkata	Unfurnished	Bachelors/Family	1	Contact Agent
3869	2022-05-20	3	19500	1270	1 out of 2	Super Area	Madipakkam	Chennai	Semi-Furnished	Bachelors	2	Contact Owner
1368	2022-06-21	1	20000	310	Ground out of 7	Carpet Area	Malad West	Mumbai	Unfurnished	Bachelors	1	Contact Agent
1528	2022-06-13	2	16000	600	1 out of 2	Carpet Area	Girinagar	Bangalore	Unfurnished	Bachelors	2	Contact Owner
309	2022-06-25	3	13000	950	Ground out of 2	Carpet Area	Rabindrapally, Garia	Kolkata	Unfurnished	Bachelors/Family	2	Contact Owner

Ordinal Attributes | Qualitative | Discrete

These are nominal attributes with values that have a *meaningful order or ranking among them*, however, the magnitude between successive values is not known.

- Professional ranks (assistant, associate, and full professor)
- Likert scale (Below average, average, Above average)
- A finite number of ordered categories, e.g., Age ≤ 45 , $45 < \text{Age} \leq 60$, $\text{Age} > 60$

	Posted On	BHK	Rent	Size	Floor	Area Type	Area Locality	City	Furnishing Status	Tenant Preferred	Bathroom	Point of Contact
311	2022-06-03	1	9000	450	Ground out of 3	Carpet Area	Salt Lake City Sector 5	Kolkata	Unfurnished	Bachelors/Family	1	Contact Agent
3869	2022-05-20	3	19500	1270	1 out of 2	Super Area	Madipakkam	Chennai	Semi-Furnished	Bachelors	2	Contact Owner
1368	2022-06-21	1	20000	310	Ground out of 7	Carpet Area	Malad West	Mumbai	Unfurnished	Bachelors	1	Contact Agent
1528	2022-06-13	2	16000	600	1 out of 2	Carpet Area	Girinagar	Bangalore	Unfurnished	Bachelors	2	Contact Owner
309	2022-06-25	3	13000	950	Ground out of 2	Carpet Area	Rabindrapally, Garia	Kolkata	Unfurnished	Bachelors/Family	2	Contact Owner

Numeric Attributes | Quantitative | Continuous/Discrete

These are measurable quantity *represented in integers or real values*. These attributes can be *interval-scaled* or *ratio-scaled*.

Numeric Attributes

Interval-scaled

Interval scales hold no true zero and can represent values below zero.

- Temperature in Celsius and Fahrenheit
- Calendar dates

Ratio-scaled

Ratio variables have inherent zero-point. They never fall below zero.

- Temperature in Kelvin (0 K = -273.15 C, i.e. matter's particles have zero kinetic energy)
- Weight, Height, Year of Experience, Word count, etc.

Numeric Attributes | Quantitative | Continuous/Discrete

Find out *interval-scaled* or *ratio-scaled* attributes from the database below.

	Posted On	BHK	Rent	Size	Floor	Area Type	Area Locality	City	Furnishing Status	Tenant Preferred	Bathroom	Point of Contact
311	2022-06-03	1	9000	450	Ground out of 3	Carpet Area	Salt Lake City Sector 5	Kolkata	Unfurnished	Bachelors/Family	1	Contact Agent
3869	2022-05-20	3	19500	1270	1 out of 2	Super Area	Madipakkam	Chennai	Semi-Furnished	Bachelors	2	Contact Owner
1368	2022-06-21	1	20000	310	Ground out of 7	Carpet Area	Malad West	Mumbai	Unfurnished	Bachelors	1	Contact Agent
1528	2022-06-13	2	16000	600	1 out of 2	Carpet Area	Girinagar	Bangalore	Unfurnished	Bachelors	2	Contact Owner
309	2022-06-25	3	13000	950	Ground out of 2	Carpet Area	Rabindrapally, Garia	Kolkata	Unfurnished	Bachelors/Family	2	Contact Owner

Measures of Central Tendency

...where would the most of the values fall. . .

Mean (*the average value*)

Let x_1, x_2, \dots, x_N be the set of N observed values or observations for an attribute X.

$$\bar{x} = \frac{\sum_{i=1}^N x_i}{N} = \frac{x_1 + x_2 + \dots + x_N}{N}.$$

Suppose that each value x_i is associated with a weight w_i for $i = 1, \dots, N$.

$$\bar{x} = \frac{\sum_{i=1}^N w_i x_i}{\sum_{i=1}^N w_i} = \frac{w_1 x_1 + w_2 x_2 + \dots + w_N x_N}{w_1 + w_2 + \dots + w_N}.$$

Highly sensitive to extreme values (e.g. outliers)!

Trimmed Mean:
After chopping off extreme values

Too much chopping may lead to LoI.

Median (*the middle value*)

Let x_1, x_2, \dots, x_N be the set of N observed values or observations for an attribute X.

- Sort x_1, x_2, \dots, x_N in an increasing order.
 - If N is *odd*, select the middle value.
 - If N is *even* and type of X is *numeric*, take the average of the two middlemost values.
 - If N is *even* and type of X is *ordinal*, the two middlemost values and any value in between.

For *skewed* data, median is a better measure of central tendency.

Mode (*the most frequent value*)

- A distribution may have more than one most frequent values (modes), are called **multimodal**.
 - The *bimodal* and *trimodal* distributions are special cases of multimodal.
- It can be determined for both *qualitative* and *quantitative* attributes.

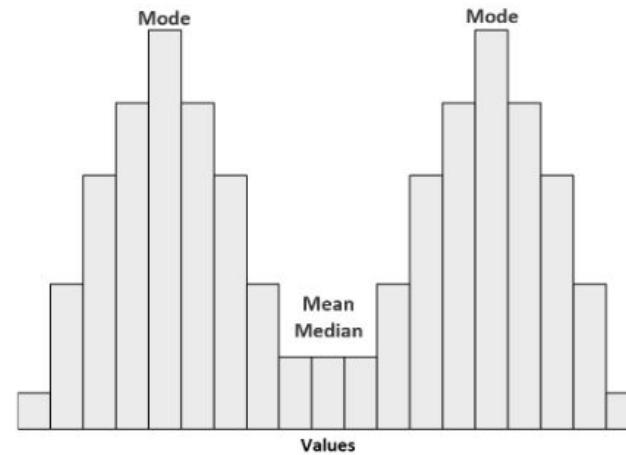
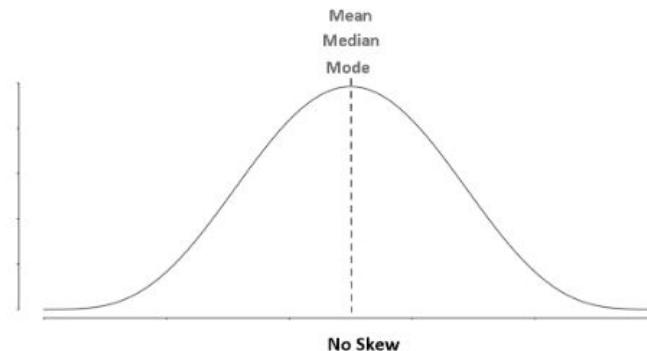
Please Indicate How You Feel About Capital Punishment?					
Frequency of Responses	Strongly oppose	Somewhat oppose	Neither	Somewhat support	Strongly support
	42	6	3	4	45

Example of Bimodal Distribution
– controversial questions tend to polarize the public.

Measures of Central Tendency

Symmetric Data

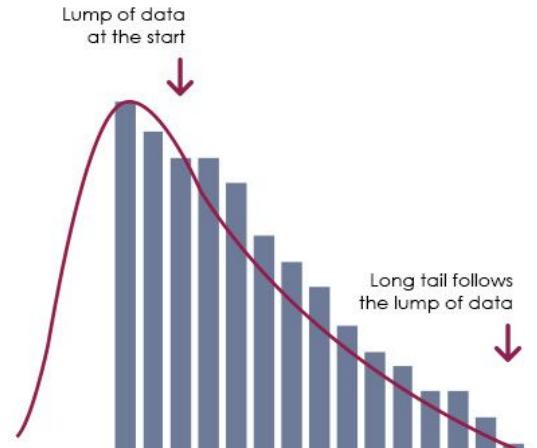
- Values are spreaded symmetrically about its mean, i.e. Skewness = 0
- For *unimodal* distribution,
Mean = Mode = Median
- For bimodal distribution,
Mean = Median, and
there will be two Modes.



Measures of Central Tendency

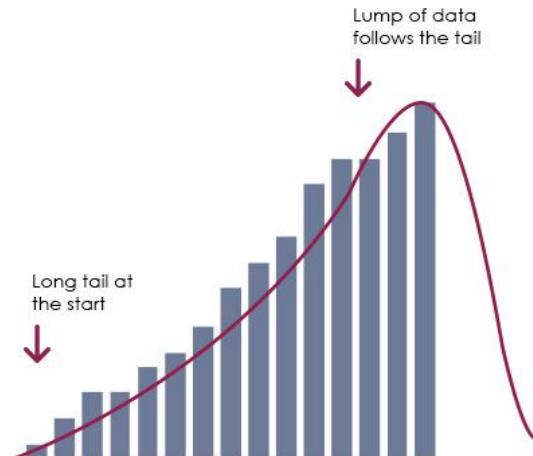
Skewed data

- If the values of a (random) variable are spreaded *asymmetrically about its mean*.



Positive skew
(right-skewed distribution)

Mode < Median < Mean



Negative skew
(left-skewed distribution)

Mode > Median > Mean

Long tail pulls
the Mean
towards its end.

Exercises

1. Does all data have a median, mode and mean?
2. In a normally distributed data set, which is greatest: mode, median or mean?
3. For any data set, which measures of central tendency have only one value?
4. Fill the entries in the following table.

Attribute Type	Measure(s) Defined	Best Measure	Why is it the best?
Nominal			
Ordinal			
Numeric (symmetric)			
Numeric (skewed)			

Next lecture

Statistics for Data Mining

28th July 2023

IT496: Introduction to Data Mining



Lecture - 03

Statistics for Data Mining - II

[Measures of Dispersion]

Arpit Rana
28th July 2023

Measures of Spread (Dispersion)

. . . about the variety of the values. . .

[Disclaimer](#): All images incorporated within the presentation slides have been sourced from a diverse array of online platforms.

Measures of Spread (Dispersion)

A measure of spread is used to describe the *variability* in a sample or population.

Why is it important?

- Mostly used in conjunction with a measure of central tendency.
- It gives us an idea of how well the measure of central tendency represents the data.

Example

- If the spread of values in the dataset is large, the mean is not as representative of the data as if the spread of data is small.
- This is because a large spread indicates that there are probably large differences between individual scores.

Range

The range of the set is the *difference between the largest and smallest values.*

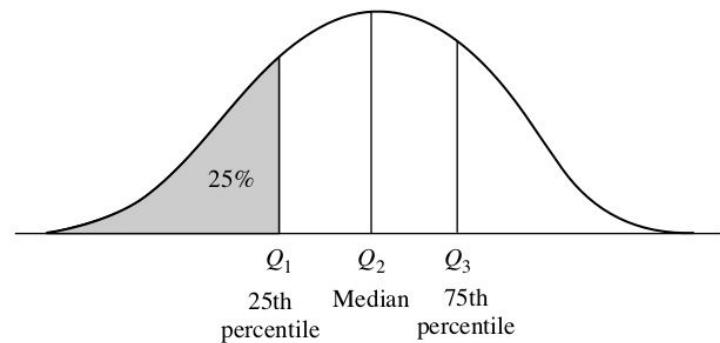
Let x_1, x_2, \dots, x_N be the set of N observed values or observations for a numeric attribute X.

$$\text{Range} = \max(X) - \min(X)$$

Quantiles

The data points that *split the data distribution into equal-size consecutive sets* are called *quantiles*.

- The 2-quantile is the data point dividing the lower and upper halves of the data distribution. It corresponds to the *median*.
- The 4-quantiles are the three data points that split the data distribution into four equal parts. They are referred to as *quartiles*.
- The 100-quantiles divide the data distribution into 100 equal-sized consecutive sets. They are referred to as *percentiles*.



Quantiles

In general

- Suppose that x_1, x_2, \dots, x_N be the set of N observed values or observations for a numeric attribute X sorted in increasing order.
- The k^{th} q -quantile for a given data distribution is the value x such that at most k/q of the data values are less than x and at most $(q - k)/q$ of the data values are more than x .
- Here k is an integer such that $0 < k < q$. There are $q-1$ q -quantiles.

Interquartile Range (IQR)

The distance between the *first* and *third* quartiles gives the range covered by the middle half of the data.

This distance is called the interquartile range (IQR).

$$\text{IQR} = Q3 - Q1$$

- A common rule of thumb for identifying suspected outliers is to figure out values falling at least $1.5 \times \text{IQR}$ above the third quartile or below the first quartile.

Why is the scale **1.5** and not any other number?

Five-Number Summary

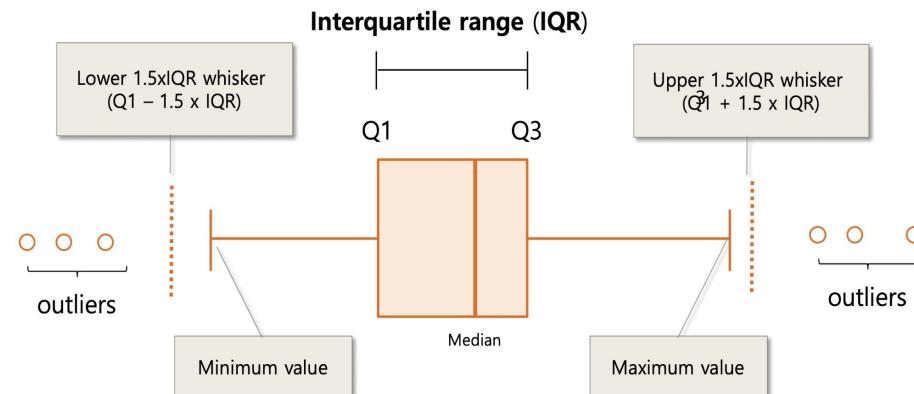
The median (Q2) and the quartiles Q1 and Q3 together contain no information about the tailing ends of the data.

The *five-number summary* of a distribution includes the *smallest* and *largest* individual observations, and is written in the order of *Minimum, Q1 , Median, Q3 , Maximum*.

Boxplot

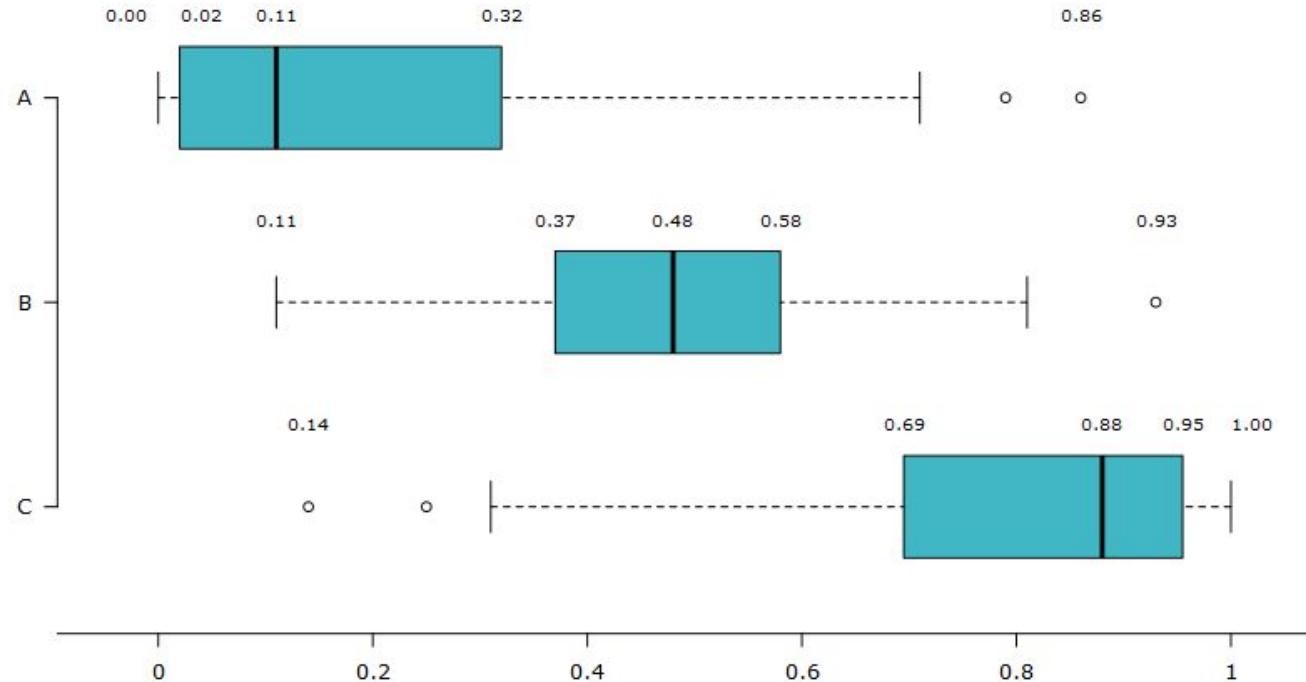
Boxplots are a popular way of visualizing a distribution. A boxplot incorporates the **five-number summary** as follows:

- Typically, the ends of the box are at the quartiles so that the box length is the IQR.
- The median is marked by a line within the box.
- Two lines (called whiskers) outside the box extend to the smallest (Minimum) and the largest (Maximum) observations.



Boxplot

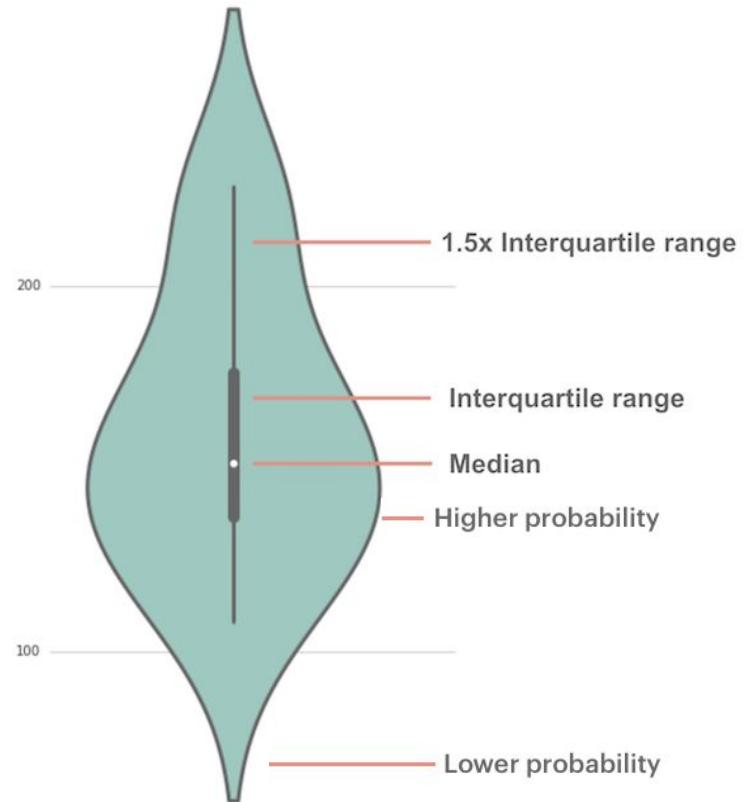
Skewed data using boxplot can be represented as below.



Violin Plot

Violin plots depict summary statistics and the density of each variable.

- the *white dot* represents the median
- the *thick gray bar* in the center represents the interquartile range
- the *thin gray line* represents the rest of the distribution, except for points that are determined to be “outliers”.



Variance and Standard Deviation

- Quantiles do not take into account every score in the data.
- To get a more representative idea of spread we need to take into account the actual values of each score in a dataset.

Variance and Standard Deviation

Variance and standard deviation indicate how spread out a data distribution is.

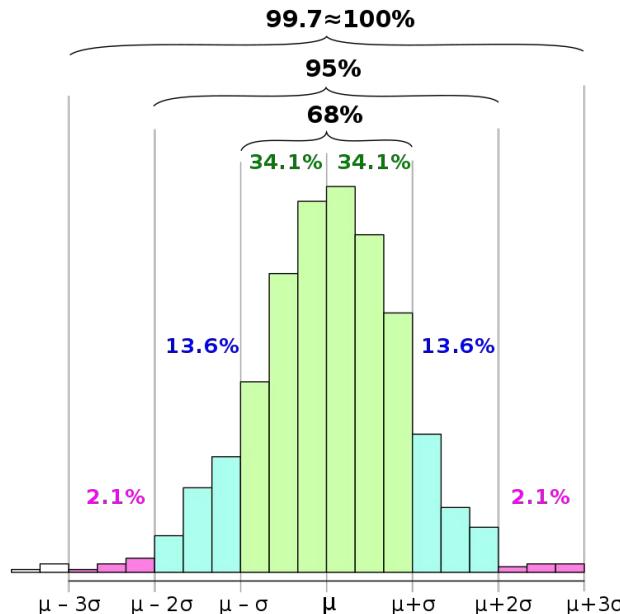
Let x_1, x_2, \dots, x_N be the set of N observed values or observations for a numeric attribute X.

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2$$

The standard deviation (σ) of the observations is the square root of the variance (σ^2).

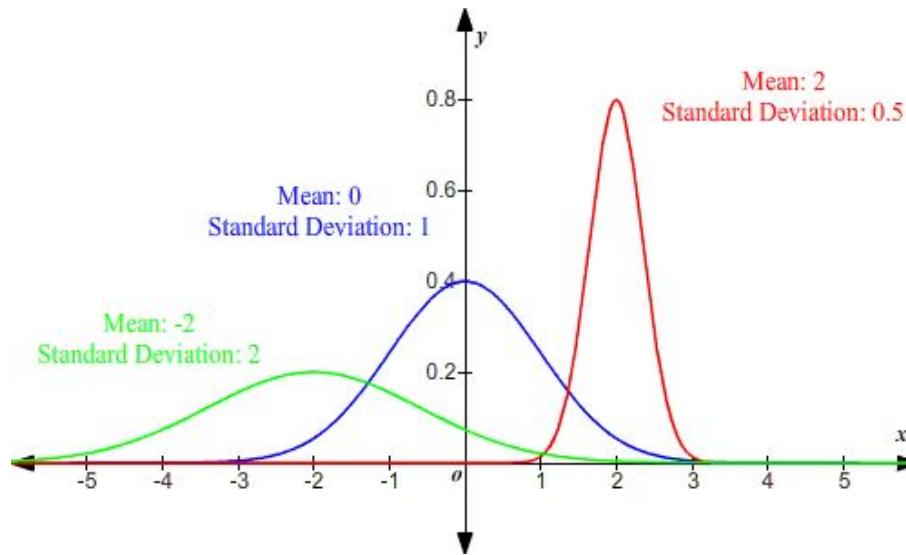
Properties on Standard Deviation

- σ measures spread about the mean and should be considered only when the mean is chosen as the measure of central tendency.
- $\sigma = 0$ only when there is no spread, that is, when all observations have the same value. Otherwise, $\sigma > 0$.



Most observations lie within a few standard deviations around the mean.

Properties on Standard Deviation



- A low standard deviation means that the data observations tend to be very close to the mean,
- while a high standard deviation indicates that the data are spread out over a large range of values.

Next lecture

Measures of Proximity

1st August 2023

IT496: Introduction to Data Mining



Lecture 04 - 06

Statistics for Data Mining - III

[Measures of Proximity]

Arpit Rana
1st / 3rd / 4th August 2023

Measures of Similarity and Dissimilarity

...how alike and unlike the data objects are in comparison to one-another...

[Disclaimer](#): Most images incorporated within the presentation slides
have been sourced from Introduction to Data Mining by Peng-Ning Tan..

Definitions

Measures of Proximity¹ (between two objects²)

Similarity

A numerical measure of the degree to which the two objects are alike.

- They are usually non-negative ($>= 0$) and are often between 0 (unlike) and 1 (alike).

Dissimilarity

A numerical measure of the degree to which the two objects are different.

- They usually fall in the interval [0, 1], but it is also common for them to range between $[0, \infty)$.

1 - For convenience, the term proximity is used to refer to either similarity or dissimilarity.

2 - The proximity between two objects is a function of the proximity between the corresponding attributes of the two objects,

Motivation

They are used by a number of data mining techniques,

- such as clustering, nearest neighbor classification, and anomaly detection.
- some approaches transforms the data to a similarity (dissimilarity) space and then performs the analysis, e.g., *kernel methods*.

Facts

We will observe the following -

- *Jaccard* and *Cosine* similarities are used on *sparse* data, e.,g., documents, user ratings
- *Euclidean* distance and *Correlation* are used on *dense* data, e.g., time-series, multidimensional data
- *Correlation* captures the linear relationship while mutual information detects non-linear relationships between the two variables.

Transformations

Transformations are often applied -

- to convert a similarity to a dissimilarity, or vice versa, or
- to transform a proximity measure to fall within a particular range, such as [0,1].

Linear Transformations

- It preserves the relative distances between points.

- Min-max transformation, $d' = \frac{d - d_{\min}}{d_{\max} - d_{\min}}$

- If the similarity falls in the interval [0,1], then the dissimilarity can be defined as

$$d = 1 - s$$

- Other transformations (*any monotonically decreasing function*):

Linear Transformations

- A few examples of monotonically decreasing function of the distance d that returns similarity within the range of $[0, 1]$:

d	$s = \frac{1}{d + 1}$	$s = e^{-d}$	$s = 1 - \frac{d - d_{\min}}{d_{\max} - d_{\min}}$
0	1	1.00	1.00
1	0.5	0.37	0.99
10	0.09	0.00	0.90
100	0.01	0.00	0.00

Objects with Single Attribute

Proximity

We first discuss proximity between objects having a single attribute.

Attribute Type	Dissimilarity	Similarity
Nominal	$d = \begin{cases} 0, & x = y \\ 1, & x \neq y \end{cases}$	$s = 1 - d = \begin{cases} 1, & x = y \\ 0, & x \neq y \end{cases}$
Ordinal	$d = \frac{ x - y }{n - 1}$ (values mapped to integers 0 to $n - 1$, where n is the number of values)	$s = 1 - d$
Numeric (interval or ratio)	$d = x - y $	$s = \frac{1}{d + 1}; \quad s = 1 - \frac{d - d_{\min}}{d_{\max} - d_{\min}}$ $s = e^{-d}; \quad s = -d$

A Scenario on an Ordinal Attribute

Consider an attribute that measures the quality of a product, e.g., a candy bar, on the scale $\{\text{poor}, \text{fair}, \text{OK}, \text{good}, \text{wonderful}\}$.

- A product, x_1 , which is rated *wonderful*, would be closer to a product x_2 , which is rated *good*, than it would be to a product x_3 , which is rated *OK*.
- To make this observation quantitative, the values of the ordinal attribute are often mapped to successive integers, beginning at 0 or 1, e.g., $\{\text{poor}=0, \text{fair}=1, \text{OK}=2, \text{good}=3, \text{wonderful}=4\}$

Then, $d(x_1, x_2) = 3 - 2 = 1$ or, if we want the dissimilarity to fall between 0 and 1,

$$d(x_1, x_2) = (3-2) / 4 = 0.25$$

Ques. Is the difference between the values ‘fair’ and ‘good’ really the same as that between the values ‘OK’ and ‘wonderful’ ?

Dissimilarities between Data Objects

... that involve multiple attributes. . .

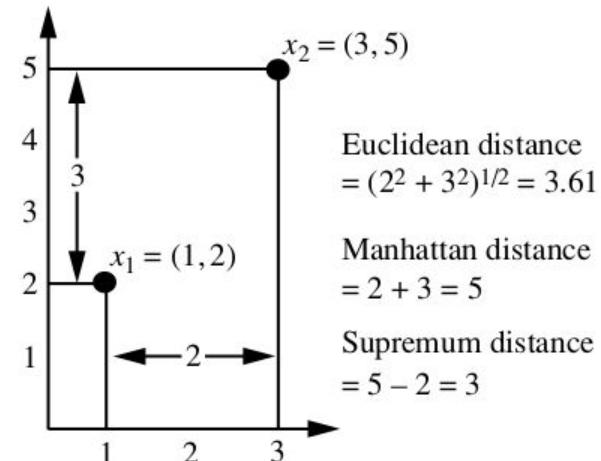
Distances

If $d(x, y)$ is the distance between two points (in our case, data objects), x and y ,

$$d(x, y) = \left(\sum_{k=1}^p |x_k - y_k|^r \right)^{1/r}$$

This is Minkowski distance metric. Where, r is a parameter.

- $r = 1$, Manhattan distance or L_1 norm
(e.g. Hamming distance for binary attributes)
- $r = 2$, Euclidean distance or L_2 norm
- $r = \infty$, Supremum distance or L_{\max} or L_∞ norm
(maximum difference between any attribute of the objects)



Distance as a Metric

If $d(x, y)$ is the distance between two points (in our case, data objects), x and y , then the following properties hold.

1. Positivity

- (a) $d(\mathbf{x}, \mathbf{y}) \geq 0$ for all \mathbf{x} and \mathbf{y} ,
- (b) $d(\mathbf{x}, \mathbf{y}) = 0$ only if $\mathbf{x} = \mathbf{y}$.

2. Symmetry

$$d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x}) \text{ for all } \mathbf{x} \text{ and } \mathbf{y}.$$

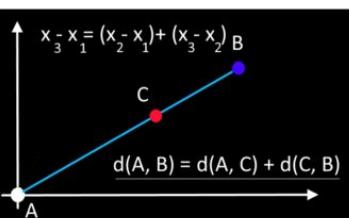
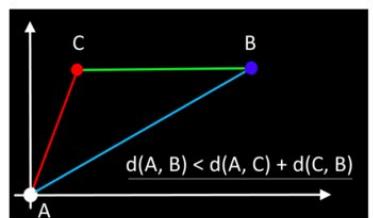
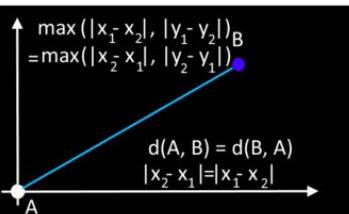
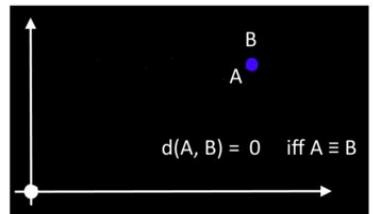
3. Triangle Inequality

$$d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z}) \text{ for all points } \mathbf{x}, \mathbf{y}, \text{ and } \mathbf{z}.$$

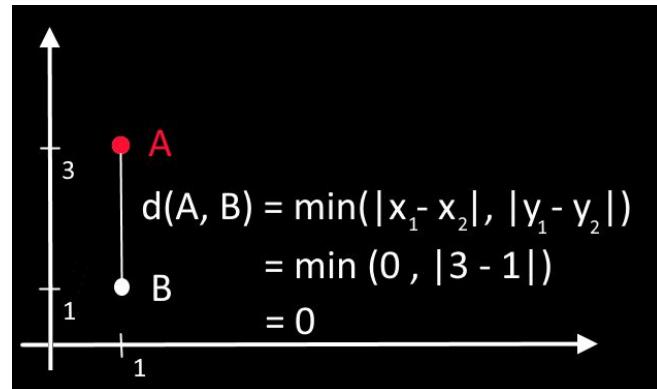
Measures that satisfy all three properties are known as **metrics**.

Distance as a Metric: Examples

Is supreme distance a metric?



What about a *minimum difference* in any attribute of the two objects (i.e. $r = -\infty$) ?



A counterexample where positivity condition violates

Ques. If A and B are two sets, then which of the following is a metric?

- $d(A, B) = |A - B|$
- $d(A, B) = |A \ominus B|$

Similarities between Data Objects

... that involve multiple attributes. . .

Similarities between Data Objects

For similarities, the triangle inequality (or the analogous property) typically does not hold, but symmetry and positivity typically do.

If $s(x, y)$ is the similarity between two points (in our case, data objects), x and y , then the following properties hold.

1. $s(\mathbf{x}, \mathbf{y}) = 1$ only if $\mathbf{x} = \mathbf{y}$. ($0 \leq s \leq 1$)
2. $s(\mathbf{x}, \mathbf{y}) = s(\mathbf{y}, \mathbf{x})$ for all \mathbf{x} and \mathbf{y} . (Symmetry)

Simple Matching Coefficient (SMC)

Let x and y be two objects that consist of n binary attributes. The comparison of two such objects, i.e., two binary vectors, leads to the following four quantities (frequencies).

f_{00} , the number of attributes where $x = 0$ and $y = 0$

f_{01} , the number of attributes where $x = 0$ and $y = 1$

f_{10} , the number of attributes where $x = 1$ and $y = 0$

f_{11} , the number of attributes where $x = 1$ and $y = 1$

$$SMC(x, y) = \frac{f_{00} + f_{11}}{f_{00} + f_{01} + f_{10} + f_{11}}$$

In case of transaction data or documents (i.e. where binary representation is sparse), SMC is not a correct measure.

Jaccard Coefficient

Let x and y be two objects that consist of n binary attributes. The Jaccard coefficient only captures the asymmetric binary attributes.

$$J(x, y) = \frac{f_{11}}{f_{01} + f_{10} + f_{11}}$$

Jaccard Coefficient

$$\mathbf{x} = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

$$\mathbf{y} = (0, 0, 0, 0, 0, 0, 1, 0, 0, 1)$$

$f_{01} = 2$ the number of attributes where \mathbf{x} was 0 and \mathbf{y} was 1

$f_{10} = 1$ the number of attributes where \mathbf{x} was 1 and \mathbf{y} was 0

$f_{00} = 7$ the number of attributes where \mathbf{x} was 0 and \mathbf{y} was 0

$f_{11} = 0$ the number of attributes where \mathbf{x} was 1 and \mathbf{y} was 1

$$SMC = \frac{f_{11} + f_{00}}{f_{01} + f_{10} + f_{11} + f_{00}} = \frac{0 + 7}{2 + 1 + 0 + 7} = 0.7$$

$$J = \frac{f_{11}}{f_{01} + f_{10} + f_{11}} = \frac{0}{2 + 1 + 0} = 0$$

In case of documents (i.e. non-binary sparse representation), Jaccard is not the right choice.

Cosine Similarity

Let \mathbf{x} and \mathbf{y} be two objects (e.g. documents) that consist of n non-binary attributes. The cosine similarity between the two vectors is defined as below.

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\| \|\mathbf{y}\|} = \frac{\mathbf{x}' \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|},$$

Where,

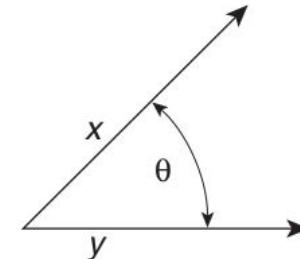
$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{k=1}^n x_k y_k = \mathbf{x}' \mathbf{y},$$

$$\|\mathbf{x}\| = \sqrt{\sum_{k=1}^n x_k^2} = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle} = \sqrt{\mathbf{x}' \mathbf{x}}.$$

Cosine Similarity

It is a measure of the (cosine of the) angle between x and y .

$$\cos(\mathbf{x}, \mathbf{y}) = \left\langle \frac{\mathbf{x}}{\|\mathbf{x}\|}, \frac{\mathbf{y}}{\|\mathbf{y}\|} \right\rangle = \langle \mathbf{x}', \mathbf{y}' \rangle,$$



Dividing x and y by their lengths normalizes them to have a length of 1.

- Thus, if the cosine similarity is 1, the angle between x and y is 0° , and x and y are the same except for length.
- If the cosine similarity is 0, then the angle between x and y is 90° , and they do not share any terms (words).

Cosine Similarity

$$\cos(\mathbf{x}, \mathbf{y}) = \left\langle \frac{\mathbf{x}}{\|\mathbf{x}\|}, \frac{\mathbf{y}}{\|\mathbf{y}\|} \right\rangle = \langle \mathbf{x}', \mathbf{y}' \rangle,$$

$$\mathbf{x} = (3, 2, 0, 5, 0, 0, 0, 2, 0, 0)$$

$$\mathbf{y} = (1, 0, 0, 0, 0, 0, 0, 1, 0, 2)$$

$$\langle \mathbf{x}, \mathbf{y} \rangle = 3 \times 1 + 2 \times 0 + 0 \times 0 + 5 \times 0 + 0 \times 0 + 0 \times 0 + 0 \times 0 + 2 \times 1 + 0 \times 0 + 0 \times 2 = 5$$

$$\|\mathbf{x}\| = \sqrt{3 \times 3 + 2 \times 2 + 0 \times 0 + 5 \times 5 + 0 \times 0 + 0 \times 0 + 0 \times 0 + 2 \times 2 + 0 \times 0 + 0 \times 0} = 6.48$$

$$\|\mathbf{y}\| = \sqrt{1 \times 1 + 0 \times 0 + 1 \times 1 + 0 \times 0 + 2 \times 2} = 2.45$$

$$\cos(\mathbf{x}, \mathbf{y}) = \mathbf{0.31}$$

The inner product depends only on components that are non-zero in both vectors (i.e. asymmetric attributes).

Correlation

It measures the strength and direction of a *linear and monotonic relationship* between two sets of values (or attributes) that are observed together (i.e., paired values).

Pearson correlation between two variables x and y is defined as below.

$$\text{corr}(\mathbf{x}, \mathbf{y}) = \frac{\text{covariance}(\mathbf{x}, \mathbf{y})}{\text{standard_deviation}(\mathbf{x}) \times \text{standard_deviation}(\mathbf{y})} = \frac{s_{xy}}{s_x s_y},$$

$$\text{covariance}(\mathbf{x}, \mathbf{y}) = s_{xy} = \frac{1}{n-1} \sum_{k=1}^n (x_k - \bar{x})(y_k - \bar{y})$$

$$\text{standard_deviation}(\mathbf{x}) = s_x = \sqrt{\frac{1}{n-1} \sum_{k=1}^n (x_k - \bar{x})^2}$$

$$\text{standard_deviation}(\mathbf{y}) = s_y = \sqrt{\frac{1}{n-1} \sum_{k=1}^n (y_k - \bar{y})^2}$$

Correlation

Correlation is always in the range -1 to 1 .

- A correlation of 1 (-1) means that x and y have a perfect positive (negative) linear relationship;
i.e., $x = ay + b$, where a and b are constants.

$$\mathbf{x} = (-3, 6, 0, 3, -6)$$

$$\mathbf{y} = (1, -2, 0, -1, 2)$$

$$\text{corr}(\mathbf{x}, \mathbf{y}) = -1 \quad x_k = -3y_k$$

$$\mathbf{x} = (3, 6, 0, 3, 6)$$

$$\mathbf{y} = (1, 2, 0, 1, 2)$$

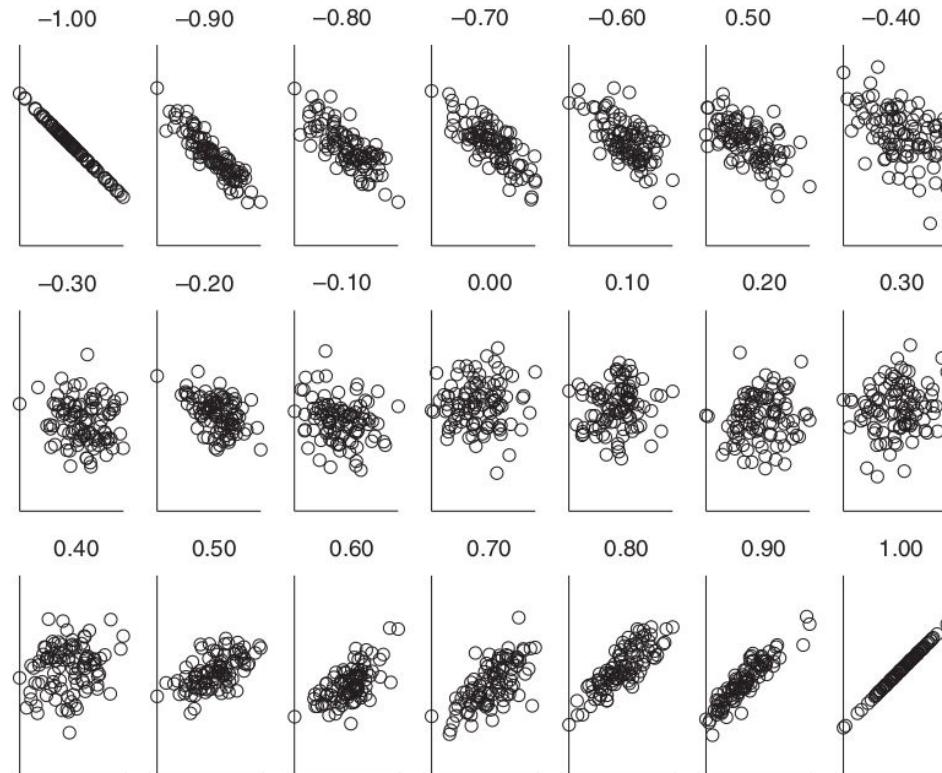
$$\text{corr}(\mathbf{x}, \mathbf{y}) = 1 \quad x_k = 3y_k$$

- If the correlation is 0 , then there is *no linear relationship* between the two sets of values.
However, nonlinear relationships can still exist.
 - For example, $y = x^2$, but their correlation is 0 .

$$\mathbf{x} = (-3, -2, -1, 0, 1, 2, 3)$$

$$\mathbf{y} = (9, 4, 1, 0, 1, 4, 9)$$

Correlation



Correlation vs. Covariance

Covariance signifies the *direction of the linear relationship* between the two variables. i.e.,

- If the variables are *directly proportional* or *inversely proportional* to each other.
(Increasing the value of one variable might have a positive or a negative impact on the value of the other variable).

Correlation explains the change in one variable leads the amount of proportion change in the second variable.

- It measures both the strength and the direction of the relationship between two variables.

Invariance to Transformation

Invariance to Transformations

A proximity measure is considered to be *invariant* to a data transformation if its value remains unchanged even after performing the transformation.

Property	Cosine	Correlation	Minkowski Distance
Invariant to scaling (multiplication)	Yes	Yes	No
Invariant to translation (addition)	No	Yes	No

Invariance to Transformations

Consider the following two vectors \mathbf{x} and \mathbf{y} with seven numeric attributes.

$$\mathbf{x} = (1, 2, 4, 3, 0, 0, 0)$$

$$\mathbf{y} = (1, 2, 3, 4, 0, 0, 0)$$

$$\mathbf{y_s} = 2 \times \mathbf{y} = (2, 4, 6, 8, 0, 0, 0)$$

$$\mathbf{y_t} = \mathbf{y} + 5 = (6, 7, 8, 9, 5, 5, 5)$$

Measure	(\mathbf{x}, \mathbf{y})	$(\mathbf{x}, \mathbf{y_s})$	$(\mathbf{x}, \mathbf{y_t})$
Cosine	0.9667	0.9667	0.7940
Correlation	0.9429	0.9429	0.9429
Euclidean Distance	1.4142	5.8310	14.2127

Invariance to Transformations

Consider the document space -

- x and y both are document vectors representing term frequencies
- y_s denotes the scaled version of y with the same term distribution; i.e., just a larger document
- y_t denotes a different document with large number of words with non-zero frequency that do not occur in y .

So, which similarity measure will be the ideal choice?

Invariance to Transformations

Consider that

- x represents a location's temperature measured on the Celsius scale for seven days.
- Let y , y_s , and y_t be the temperatures measured on those days at a different location, but using three different measurement scales.

So, which similarity measure will be the ideal choice?

Hint: Different units of temperature have different offsets (e.g., Celsius and Kelvin) and different scaling factors (e.g., Celsius and Fahrenheit).

Invariance to Transformations

Consider a scenario where

- x represents the amount of precipitation (in cm) measured at seven locations.
- Let y , y_s , and y_t be estimates of the precipitation at these locations, which are predicted using three different models.
- We would like to choose a model that accurately reconstructs the measurements in x without making any error.

So, which proximity measure will be the ideal choice?

Mutual Information (MI)

Given that the values come in pairs, MI shows how much information one set of values provides about another.

It is used when a *nonlinear relationship* is suspected between the pairs of values.

- If the two sets of values are independent, then their MI is 0.
- If the two sets of values are completely dependent, then they have maximum MI.
- MI does not have a maximum value, but can be normalized to $[0, 1]$.

Mutual Information (MI)

Let X can take m distinct values, u_1, u_2, \dots, u_m and Y can take n distinct values, v_1, v_2, \dots, v_n .

Then their individual and joint entropy can be defined in terms of the probabilities of each value and pair of values as follows:

$$H(X) = - \sum_{j=1}^m P(X = u_j) \log_2 P(X = u_j)$$

$$H(Y) = - \sum_{k=1}^n P(Y = v_k) \log_2 P(Y = v_k)$$

$$H(X, Y) = - \sum_{j=1}^m \sum_{k=1}^n P(X = u_j, Y = v_k) \log_2 P(X = u_j, Y = v_k)$$

The mutual information of X and Y can now be defined straightforwardly:

$$I(X, Y) = H(X) + H(Y) - H(X, Y)$$

One way to normalize MI is to divide it by $\log_2(\min(m, n))$.

Mutual Information (MI): Example

Suppose $y = x^2$, and their correlation is 0.

$$\mathbf{x} = (-3, -2, -1, 0, 1, 2, 3)$$

$$\mathbf{y} = (9, 4, 1, 0, 1, 4, 9)$$

$$\begin{aligned} I(X, Y) &= H(X) + H(Y) - H(X, Y) \\ &= (2.8074 + 1.9502) - 2.8074 \\ &= 1.9502 \end{aligned}$$

On normalizing the value,,

$$= 1.9502 / \log_2(4) = 0.9751$$

x_j	$P(\mathbf{x} = x_j)$	$-P(\mathbf{x} = x_j) \log_2 P(\mathbf{x} = x_j)$
-3	1/7	0.4011
-2	1/7	0.4011
-1	1/7	0.4011
0	1/7	0.4011
1	1/7	0.4011
2	1/7	0.4011
3	1/7	0.4011
$H(\mathbf{x})$		2.8074

y_k	$P(\mathbf{y} = y_k)$	$-P(\mathbf{y} = y_k) \log_2 (P(\mathbf{y} = y_k))$
9	2/7	0.5164
4	2/7	0.5164
1	2/7	0.5164
0	1/7	0.4011
$H(\mathbf{y})$		1.9502

x_j	y_k	$P(\mathbf{x} = x_j, \mathbf{y} = y_k)$	$-P(\mathbf{x} = x_j, \mathbf{y} = y_k) \log_2 P(\mathbf{x} = x_j, \mathbf{y} = y_k)$
-3	9	1/7	0.4011
-2	4	1/7	0.4011
-1	1	1/7	0.4011
0	0	1/7	0.4011
1	1	1/7	0.4011
2	4	1/7	0.4011
3	9	1/7	0.4011
$H(\mathbf{x}, \mathbf{y})$			2.8074

Issues related to Proximity Measures

There are a few important issues in proximity calculation:

- how to handle the case in which attributes have different scales and/or are correlated,
- how to calculate proximity between objects that are composed of different types of attributes, e.g., quantitative and qualitative, and
- how to handle proximity calculations when attributes have different weights; i.e., when not all attributes contribute equally to the proximity of objects.

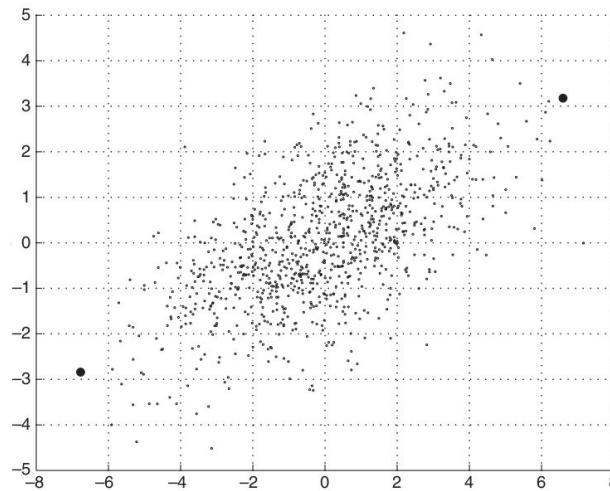
Issues related to Proximity Measures

- how to handle the case in which attributes have different scales and/or are correlated,
 - If the attributes are relatively uncorrelated, but have different ranges, then standardizing the variables is sufficient.
 - If the attributes are correlated and have different ranges of values, the *Mahalanobis distance* is useful.

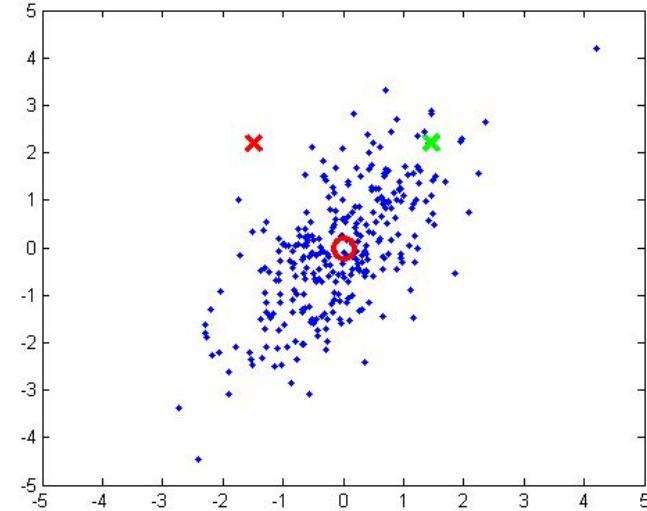
$$\text{Mahalanobis}(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})' \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \mathbf{y})},$$

Here, $\boldsymbol{\Sigma}^{-1}$ is the inverse of the covariance matrix of the data.

Issues related to Proximity Measures



$$d_{\text{eucl}} = 14.7; \quad d_{\text{mahl}} = 6$$



$$d_{\text{eucl}}^r = d_{\text{eucl}}^g; \quad d_{\text{mahl}}^r = 4.12, \quad d_{\text{mahl}}^g = 2$$

We can also use PCA to remove correlation among attributes. It transforms the data into orthogonal principal components. We will see it while studying dimensionality reduction.

Issues related to Proximity Measures

- how to calculate proximity between objects that are composed of different types of attributes, e.g., quantitative and qualitative,

1: For the k^{th} attribute, compute a similarity, $s_k(\mathbf{x}, \mathbf{y})$, in the range [0, 1].

2: Define an indicator variable, δ_k , for the k^{th} attribute as follows:

$$\delta_k = \begin{cases} 0 & \text{if the } k^{th} \text{ attribute is an asymmetric attribute and} \\ & \text{both objects have a value of 0, or if one of the objects} \\ & \text{has a missing value for the } k^{th} \text{ attribute} \\ 1 & \text{otherwise} \end{cases}$$

3: Compute the overall similarity between the two objects using the following formula:

$$\text{similarity}(\mathbf{x}, \mathbf{y}) = \frac{\sum_{k=1}^n \delta_k s_k(\mathbf{x}, \mathbf{y})}{\sum_{k=1}^n \delta_k}$$

Issues related to Proximity Measures

- how to handle proximity calculations when attributes have different weights; i.e., when not all attributes contribute equally to the proximity of objects.

$$\text{similarity}(\mathbf{x}, \mathbf{y}) = \frac{\sum_{k=1}^n w_k \delta_k s_k(\mathbf{x}, \mathbf{y})}{\sum_{k=1}^n w_k \delta_k}.$$

Next lecture

Data Preprocessing

8th August 2023



IT496: Introduction to Data Mining



Lecture 07

Fundamentals of Predictive Analytics
[Representation, Evaluation, and Optimization]

Arpit Rana
8th August 2023

Data Mining Tasks

[Disclaimer](#): Most images incorporated within the presentation slides have been sourced from different sources on the web and ML books.

Data Mining Tasks

Data Mining Tasks

The actual data mining task is the semi-automatic or automatic analysis of large quantities of data to extract interesting patterns.

Descriptive

Find human-interpretable patterns that describe the data.

- Cluster Analysis
- Outlier Analysis
- Association Rule Mining
- Sequence Pattern Mining

Predictive

Use some variables to predict future or unknown values of other variables.

- Regression
- Classification

In Machine Learning terminology, these tasks are categorised as “**Unsupervised Learning**”.

In Machine Learning terminology, these tasks are categorised as “**Supervised Learning**”.

Data Mining Tasks

Data Mining Tasks

The actual data mining task is the semi-automatic or automatic analysis of large quantities of data to extract interesting patterns.

Descriptive

Find human-interpretable patterns that describe the data.

- Cluster Analysis
- Outlier Analysis
- Association Rule Mining
- Sequence Pattern Mining

Predictive

Use some variables to predict future or unknown values of other variables.

- Regression
- Classification

In Machine Learning terminology, these tasks are categorised as “**Unsupervised Learning**”.

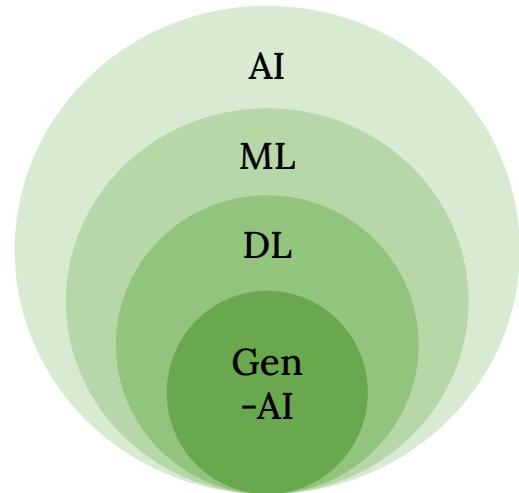
In Machine Learning terminology, these tasks are categorised as “**Supervised Learning**”.

Machine Learning: Definition

Machine Learning is

- the science (and art) of programming computers
- so they can learn from data.

– Aurelien Geron, Google



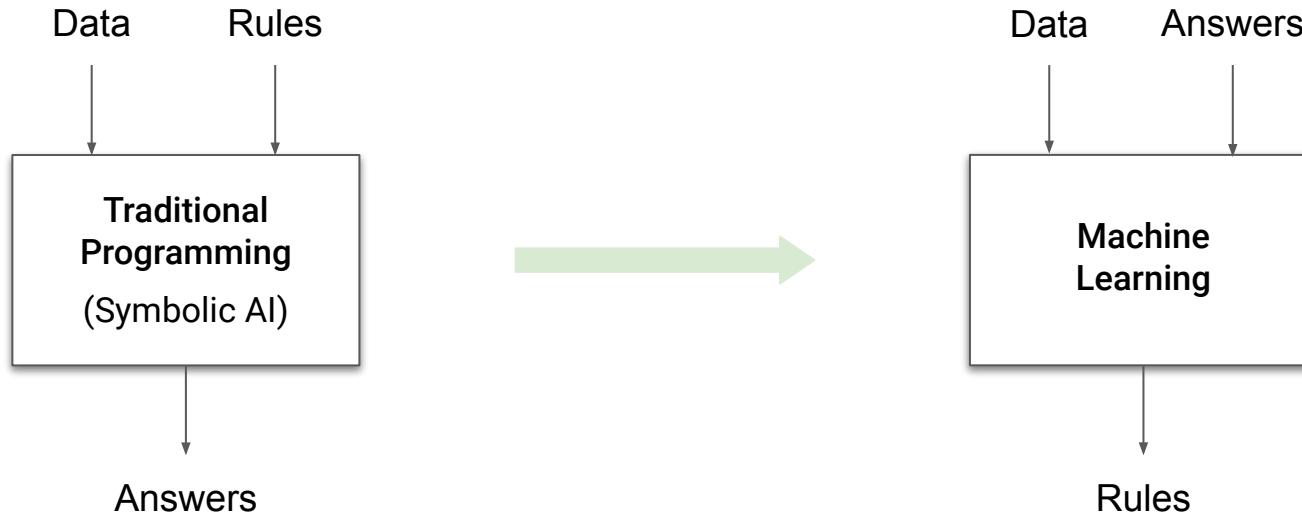
Machine Learning: Example

A Spam Filter,

- a Machine Learning Program, given
 - examples of “spam” emails (e.g. flagged by users), and
 - examples of “ham” (i.e. regular) emails
- can learn to flag spam



Machine Learning: A New Programming Paradigm

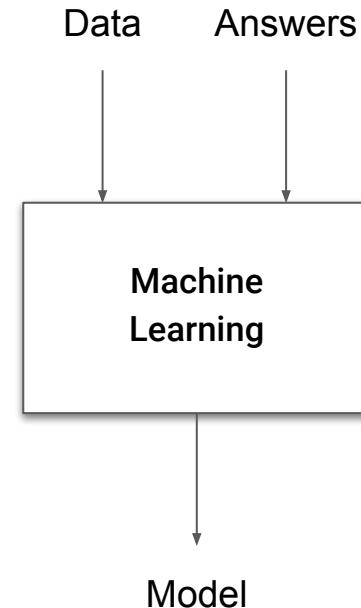


- A long list of complex (hard coded) rules
- Keep writing new rules as the new phrases are introduced by spammers
- Automatically learns which words or phrases are good predictors of spam

Machine Learning: Definition Revisited

Machine Learning is the training of a model from data that generalises a decision against a performance measure.

- Training a model suggests training examples.
- A model suggests state acquired through experience.
- Generalises a decision suggests the capability to make a decision based on inputs and anticipating unseen inputs in the future for which a decision will be required.
- against a performance measure suggests a targeted need and directed quality to the model being prepared.



Learning = Representation + Evaluation + Optimization

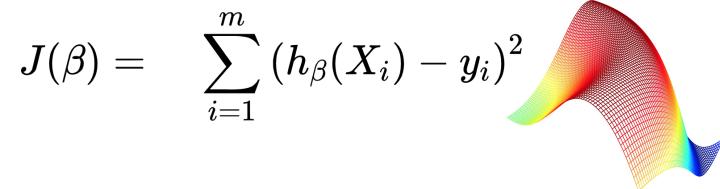
Representation

Choosing a representation of the learner: the *hypotheses space* or the *model class* – the set of models that it can possibly learn.

$$h_{\beta}(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_m X_m$$
$$= \sum_{i=1}^m \beta_i X_i$$

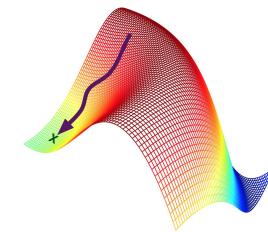

Evaluation

Choosing an evaluation function (also called objective function, utility function, loss function, or scoring function) is needed to distinguish good classifiers from bad ones.



Optimization

Choosing a method to search among the models in the hypothesis space for the highest-scoring one.

$$\min J(\beta)$$


Learning = Representation + Evaluation + Optimization

Representation	Evaluation	Optimization
Instances	Accuracy/Error rate ✓	Combinatorial optimization
K-nearest neighbor ✓	Precision and recall ✓	Greedy search ✓
Support vector machines ✓	Squared error ✓	Beam search ✓
Hyperplanes	Likelihood	Branch-and-bound
Naive Bayes	Posterior probability	Continuous optimization
Logistic regression ✓	Information gain ✓	Unconstrained
Decision trees ✓	K-L divergence	Gradient descent ✓
Sets of rules	Cost/Utility ✓	Conjugate gradient
Propositional rules	Margin ✓	Quasi-Newton methods
Logic programs		Constrained
Neural networks ✓		Linear programming
Graphical models		Quadratic programming ✓
Bayesian networks		
Conditional random fields		

Supervised Learning

Problem Settings and Examples

Supervised Learning: A Formal Model

The learner's input:

- **Domain set**

An arbitrary set (instance space), X , the set of objects (a.k.a. instances, domain points) we may wish to label.

- **Label set**

A set of possible labels, Y . e.g., $\{0, 1\}$, $\{-1, 1\}$.

- **Training data**

$S = ((x_1, y_1) \dots (x_m, y_m))$ is finite sequence of pairs in $X \times Y$, i.e., a sequence of labeled domain points.

The learner's output:

- A prediction rule, $h : X \rightarrow Y$, also called a *predictor*, a *hypothesis*, or a *classifier*.

- The learner returns h upon receiving the training sequence S .

- It can be used to predict the label of new domain points (*like the past ones*).

Supervised Learning: A Formal Model

Data-generation Model:

- Let D be a probability distribution over $X \times Y$, i.e., D is joint probability distribution over domain points and labels.
 - A distribution D_x over unlabeled domain points (sometimes called *marginal distribution*),
 - A conditional probability over labels for each domain point, $D(y | x)$.

Independent and Identically Distributed (I.I.D.) Assumption

- Each domain point x has the same prior probability distribution (to be sampled):

$$P(x_i) = P(x_{i+1}) = P(x_{i+2}) = \dots,$$

and is independent of the previous examples:

$$P(x_i) = P(x_i | x_{i-1}, x_{i-2}, \dots).$$

Supervised Learning: A Formal Model

More formally, the task of supervised learning can be defined as -

Given a training set (S) of m example input-output pairs,

$$S = (X, y)$$

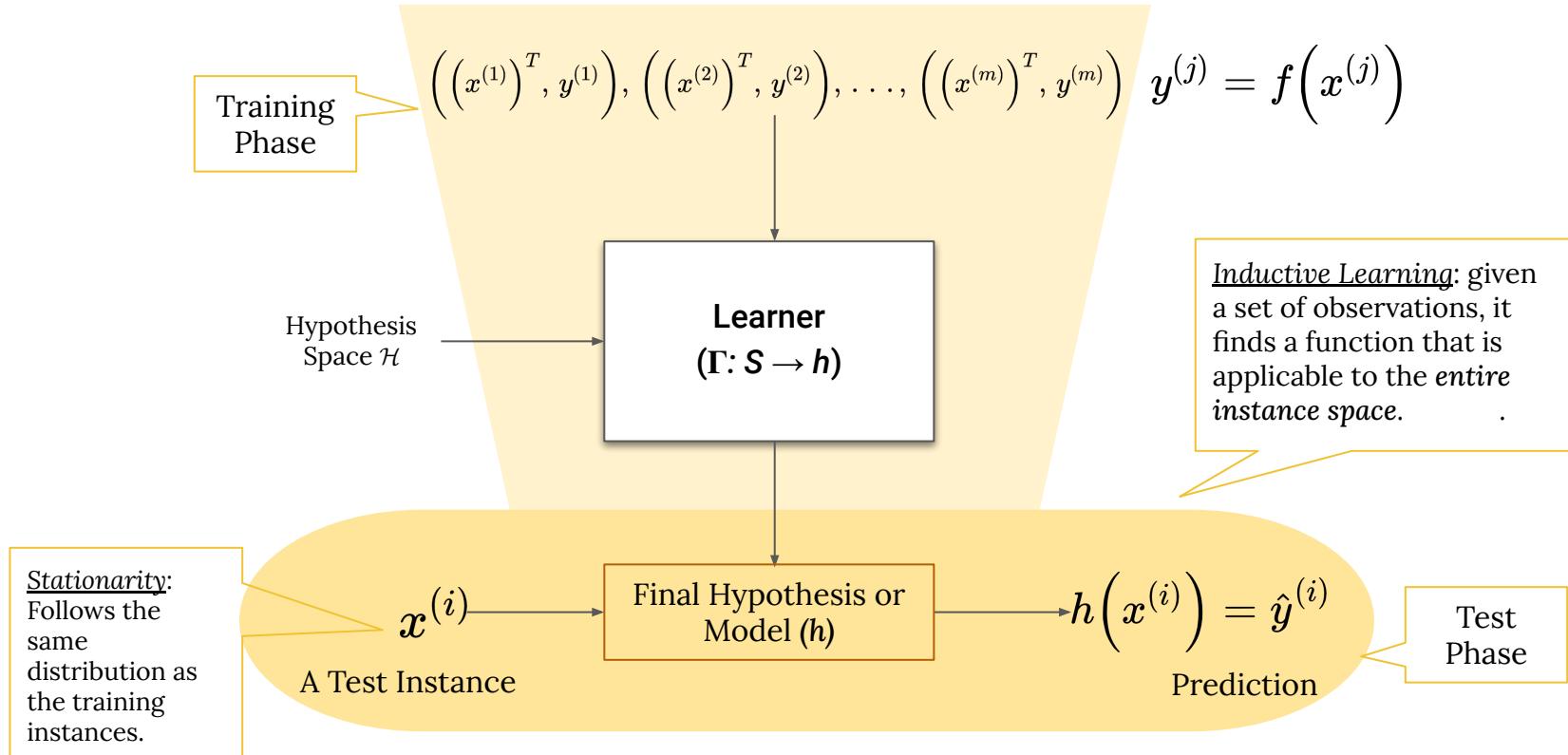
$$X = \begin{pmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ \vdots \\ (x^{(m-1)})^T \\ (x^{(m)})^T \end{pmatrix}, y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ \vdots \\ y^{(m-1)} \\ y^{(m)} \end{pmatrix}$$

We call the output $y^{(i)}$ the ground truth – the true answer we are asking our model to predict.

$$\left((x^{(1)})^T, y^{(1)} \right), \left((x^{(2)})^T, y^{(2)} \right), \dots, \left((x^{(m)})^T, y^{(m)} \right)$$

where each pair was generated by an unknown function $y = f(x)$,
discover a function h that approximates the true function f .

Supervised Learning Process



Next lecture

Choosing a Hypothesis Space

10th August 2023

IT496: Introduction to Data Mining



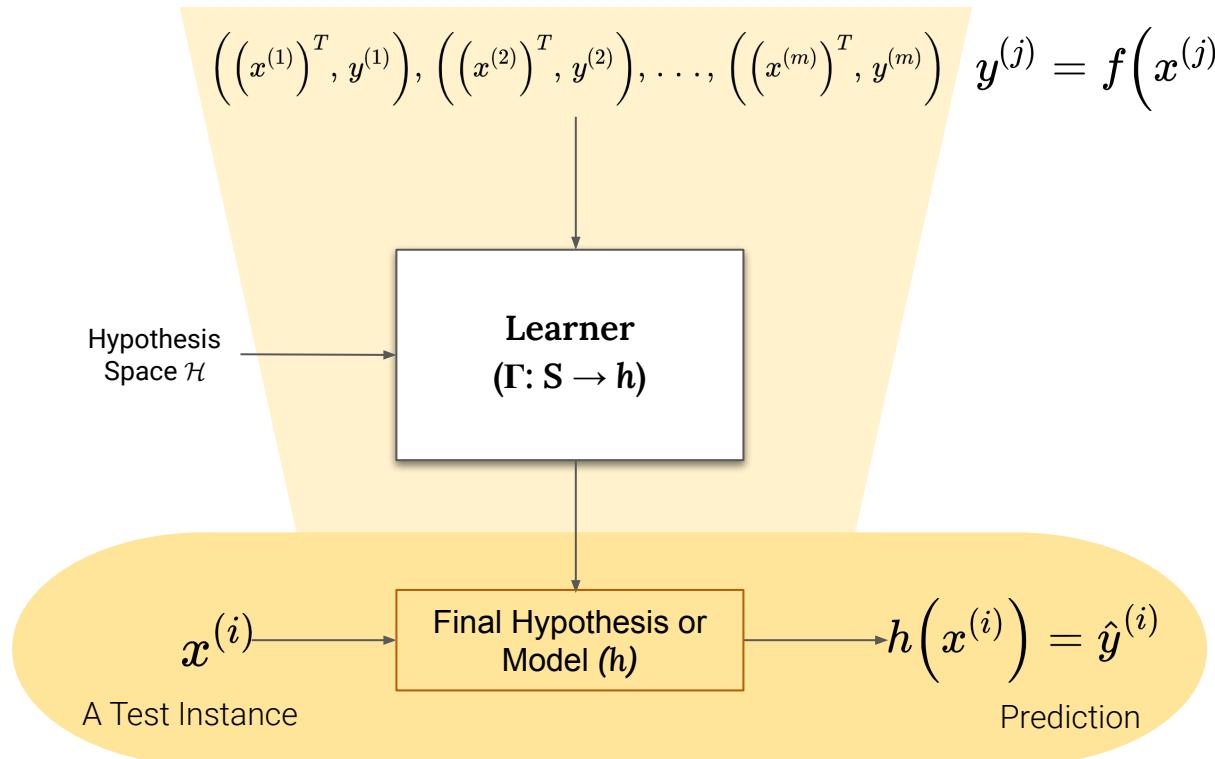
Lecture 08

Choosing a Hypothesis Space

[Inductive Bias, Bias-Variance Trade-off, Model Complexity and Expressiveness Trade-off]

Arpit Rana
11th August 2023

Supervised Learning Process



Supervised Learning: Example

Problem: whether to wait for a table at a restaurant.

Example	Input Attributes										Output
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	
x_1	Yes	No	No	Yes	Some	\$\$\$	No	Yes	French	0–10	$y_1 = Yes$
x_2	Yes	No	No	Yes	Full	\$	No	No	Thai	30–60	$y_2 = No$
x_3	No	Yes	No	No	Some	\$	No	No	Burger	0–10	$y_3 = Yes$
x_4	Yes	No	Yes	Yes	Full	\$	Yes	No	Thai	10–30	$y_4 = Yes$
x_5	Yes	No	Yes	No	Full	\$\$\$	No	Yes	French	>60	$y_5 = No$
x_6	No	Yes	No	Yes	Some	\$\$	Yes	Yes	Italian	0–10	$y_6 = Yes$
x_7	No	Yes	No	No	None	\$	Yes	No	Burger	0–10	$y_7 = No$
x_8	No	No	No	Yes	Some	\$\$	Yes	Yes	Thai	0–10	$y_8 = Yes$
x_9	No	Yes	Yes	No	Full	\$	Yes	No	Burger	>60	$y_9 = No$
x_{10}	Yes	Yes	Yes	Yes	Full	\$\$\$	No	Yes	Italian	10–30	$y_{10} = No$
x_{11}	No	No	No	No	None	\$	No	No	Thai	0–10	$y_{11} = No$
x_{12}	Yes	Yes	Yes	Yes	Full	\$	No	No	Burger	30–60	$y_{12} = Yes$

- Alternate: whether there is a suitable alternative restaurant nearby.
- Bar: whether the restaurant has a comfortable bar area to wait in.
- Fri/Sat: true on Fridays and Saturdays.
- Hungry: whether we are hungry right now.
- Patrons: how many people are in the restaurant (values are None, Some, and Full).
- Price: the restaurant's price range (\$, \$\$, \$\$\$).
- Raining: whether it is raining outside.
- Reservation: whether we made a reservation.
- Type: the kind of restaurant (French, Italian, Thai, or Burger).
- WaitEstimate: host's wait estimate: 0–10, 10–30, 30–60, or >60 minutes.

Supervised Learning: Example

Problem: whether to wait for a table at a restaurant.

Example	Input Attributes										Output WillWait	
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est		
Training Data	x_1	Yes	No	No	Yes	Some	\$\$\$	No	Yes	French	0–10	$y_1 = \text{Yes}$
	x_2	Yes	No	No	Yes	Full	\$	No	No	Thai	30–60	$y_2 = \text{No}$
	x_3	No	Yes	No	No	Some	\$	No	No	Burger	0–10	$y_3 = \text{Yes}$
	x_4	Yes	No	Yes	Yes	Full	\$	Yes	No	Thai	10–30	$y_4 = \text{Yes}$
	x_5	Yes	No	Yes	No	Full	\$\$\$	No	Yes	French	>60	$y_5 = \text{No}$
	x_6	No	Yes	No	Yes	Some	\$\$	Yes	Yes	Italian	0–10	$y_6 = \text{Yes}$
	x_7	No	Yes	No	No	None	\$	Yes	No	Burger	0–10	$y_7 = \text{No}$
	x_8	No	No	No	Yes	Some	\$\$	Yes	Yes	Thai	0–10	$y_8 = \text{Yes}$
	x_9	No	Yes	Yes	No	Full	\$	Yes	No	Burger	>60	$y_9 = \text{No}$
	x_{10}	Yes	Yes	Yes	Yes	Full	\$\$\$	No	Yes	Italian	10–30	$y_{10} = \text{No}$
	x_{11}	No	No	No	No	None	\$	No	No	Thai	0–10	$y_{11} = \text{No}$
	x_{12}	Yes	Yes	Yes	Yes	Full	\$	No	No	Burger	30–60	$y_{12} = \text{Yes}$

$$y = f(x)$$

Unknown
Target
function f

Instances

Instance Space (X) $2 \times 2 \times 2 \times 2 \times 2 \times 3 \times 3 \times 3 \times 2 \times 2 \times 4 \times 4 = 9216$

Size of Hypothesis Space ($|\mathcal{H}|$)
of Boolean Functions

$$= 2^{9216}$$

Hypothesis Space vs. Hypothesis

What do we mean by a Hypothesis Space (a.k.a. Model Class) and a hypothesis?

There are three different levels of specificity for using the term Hypothesis or Model:

- a broad hypothesis space (like “polynomials”),
- a hypothesis space with hyperparameters filled in (like “degree-2 polynomials”), and
- a specific hypothesis with all parameters filled in (like $5x^2 + 3x - 2$).

Hypothesis Space vs. Hypothesis

What do we mean by a Hypothesis Space (a.k.a. Model Class) and a hypothesis?

There are three different levels of specificity for using the term Hypothesis or Model:

$$\begin{aligned}y &= f(x) = bx + a \\&= f(x) = e^{-bx} \\&= f(x) = \sin(bx) \\&= f(x) = bx^2 \\&= f(x) = \sqrt{bx + a}\end{aligned}$$

Polynomials
→

$$\begin{aligned}y &= f(x) = bx + a \\&= f(x) = bx^2 \\&= f(x) = \sqrt{bx + a}\end{aligned}$$

Hyperparameter:
degree=1
→

$$y = f(x) = bx + a$$

Parameters:
a=2, b=3
↓

$$f(x) = 3x + 2$$

Hypothesis Space vs. Hypothesis

How do we choose a good Hypothesis Space or Model Class?

$$y = f(x) = bx + a$$

$$= f(x) = e^{-bx}$$

$$= f(x) = \sin(bx)$$

$$= f(x) = bx^2$$

$$= f(x) = \sqrt{bx + a}$$

Polynomials
→

$$y = f(x) = bx + a$$

$$= f(x) = bx^2$$

$$= f(x) = \sqrt{bx + a}$$

Hyperparameter:
degree=1
→

$$y = f(x) = bx + a$$

Parameters:
 $a=2, b=3$

$$f(x) = 3x + 2$$

Hypothesis Space / Representation / Model Class Selection
(popularly known as **Model Selection**)

Optimization
or Training

Choosing the Hypothesis Space

Hypothesis Space Selection is Subjective

Most probable hypothesis given the data -

$$h^* = \arg \max_{h \in \mathcal{H}} P(h | S) \quad \equiv \quad h^* = \arg \max_{h \in \mathcal{H}} P(S | h) P(h)$$

- We can say that the prior probability $P(h)$ is high for a smooth degree-1 or -2 polynomial and lower for a degree-12 polynomial with large, sharp spikes.

Hypothesis Space Selection is Subjective

The observed dataset S alone does not allow us to make conclusions about unseen instances.
We need to make some assumptions!

- These assumptions induce the *bias* (a.k.a. *inductive* or *learning bias*) of a learning algorithm.
- Two ways to induce bias:
 - *Restriction*: Limit the hypothesis space (e.g., degree-2 polynomials)
 - *Preference*: Impose ordering on hypothesis space (e.g., prefer simpler than complex)

Hypothesis Space Selection is not only subjective but is empirical also.

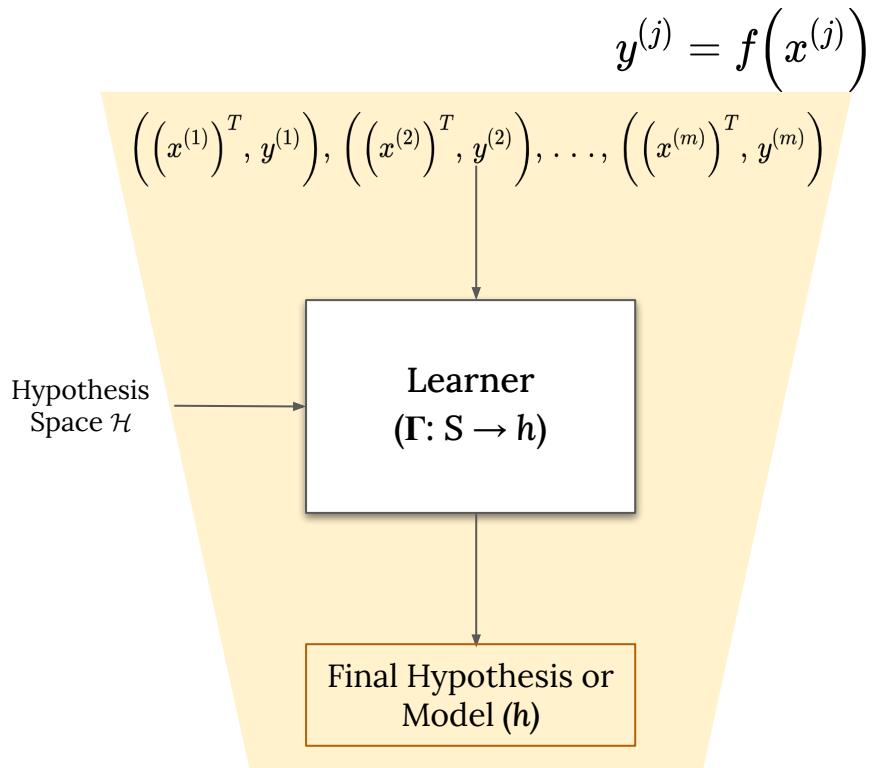
- Part of hypothesis space selection is qualitative and subjective:
We might select polynomials rather than decision trees based on something that we know about the problem,
and
- part is quantitative and empirical:
Within the class of polynomials, we might select Degree = 2, because that value performs best on the validation data set.

Experimental Evaluation of Learning Algorithms

The overall objective of the Learning Algorithm is to find a *hypothesis* that -

- is consistent (i.e., fits the training data), but more importantly,
- generalizes well for previously unseen data.

Experimental Evaluation defines ways to Measure the Generalizability of a Learning Algorithm.



Experimental Evaluation of Learning Algorithms

Sample Error

The *sample error* of hypothesis h with respect to the target function f and data sample S is:

$$\text{error}_S(h) = \frac{1}{n} \sum_{x \in S} \delta(h(x), f(x))$$

It is impossible to assess *true error*, so we try to estimate it using *sample error*.



True Error

The *true error* of hypothesis h with respect to the target function f and the distribution D is the probability that h will misclassify an instance drawn at random according to D :

$$\text{error}_D(h) = P_{x \in D}[h(x) \neq f(x)]$$

Generalization Error

Generalization error (a.k.a. *out-of-sample error*) is a measure of how accurately an algorithm is able to predict outcome values for *previously unseen data*.

$$\text{error}(h(x), f(x)) = \boxed{\text{var}(x)} + \boxed{\text{bias}(x)^2} + \boxed{\epsilon^2}$$



Variance

Due to the model's sensitivity to small variations in the training data.

It leads to **overfitting!**



Bias

Due to Wrong Assumptions. Restrictions imposed by -

The Representation Function (i.e., Hypothesis space, such as, linear or quadratic)

The Search Algorithm (e.g., Grid search or Beam search)

It leads to **underfitting!**



Irreducible Error

Due to the noisiness of the data itself.

The only way to handle it is to clean up the data properly, detect and remove outliers.

Choosing a Hypothesis Space - I

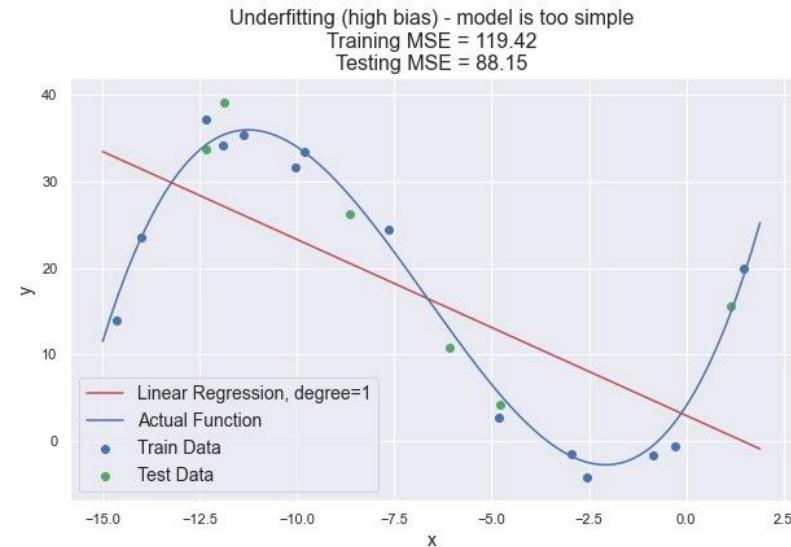
One way to analyze hypothesis spaces is by

- the bias they impose (regardless of the training data set), and
- the variance they produce (from one training set to another).

Bias

The tendency of a predictive hypothesis to deviate from the expected value when averaged over different training sets.

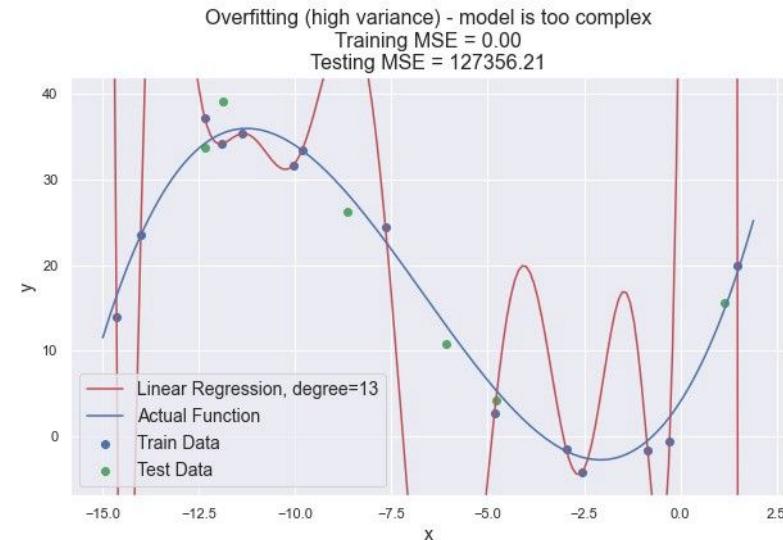
- Bias often results from restrictions imposed by the hypothesis space.
- We say that a hypothesis is underfitting when it fails to find a pattern in the data.



Variance

The amount of change in the hypothesis due to fluctuation in the training data.

- We say a function is overfitting the data when it pays too much attention to the particular data set it is trained on.
- It causes the hypothesis to perform poorly on unseen data.



Bias-Variance Trade-off

- **High Variance-High Bias**

The model is inconsistent and also inaccurate on average

- **Low Variance-High Bias**

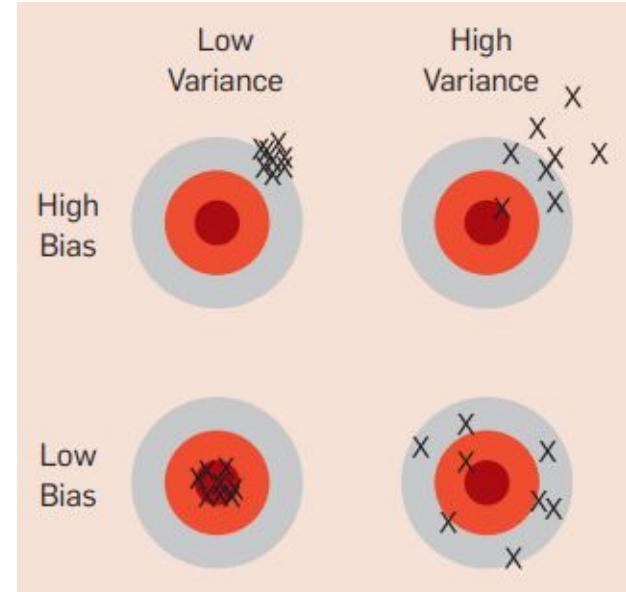
Models are consistent but low on average

- **High Variance-Low Bias**

Somewhat accurate but inconsistent on average

- **Low Variance-Low Bias**

Model is consistent and accurate on average



Analogy with throwing darts at a board.

Choosing a Hypothesis Space - II

Another way to analyze hypothesis spaces is by

- the *expressiveness* (i.e., ability of a model to represent a wide variety of functions or patterns) of a hypothesis space, and
 - Can be measured by the size of the hypothesis space
- the *model complexity* (i.e., how intricate the relationships a model can capture) of a hypothesis space.
 - Can be estimated by the number of parameters of a hypothesis

Note-1: Sometimes the term *model capacity* is used to refer to model complexity and expressiveness together.

Note-2: In general, the required amount of training data depends on the model complexity, representativeness of the training sample, and the acceptable error margin.

Choosing a Hypothesis Space - II

There is a tradeoff between the expressiveness of a hypothesis space and the computational complexity of finding a good hypothesis within that space.

- Fitting a straight line to data is an easy computation; fitting high-degree polynomials is somewhat harder; and fitting unusual-looking functions may be undecidable.
- After learning h , computing $h(x)$ when h is a linear function is guaranteed to be fast, while computing an arbitrarily complex function may not even guaranteed to terminate.

For example:

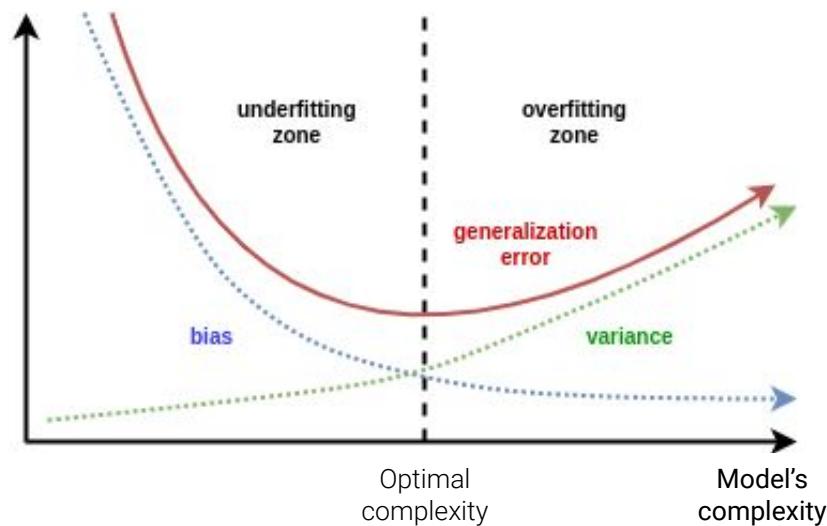
- In Deep Learning, representations are not simple but the $h(x)$ computation still takes only a *bounded number* of steps to compute with appropriate hardware.

Bias-Variance vs. Model's Complexity

The relationship between bias and variance is closely related to the machine learning concepts of overfitting, underfitting, and model's complexity.

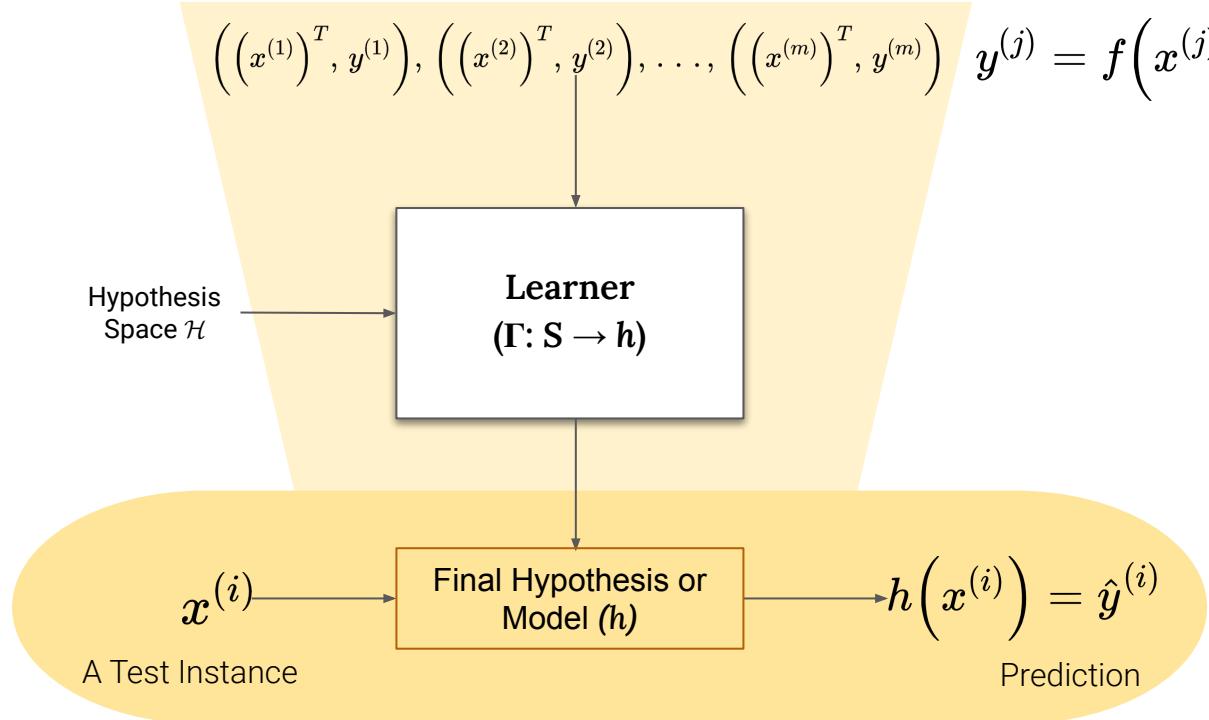
- Increasing a model's complexity typically increases its variance and reduces its bias.
- Reducing a model's complexity increases its bias and reduces its variance.

This is why it is called a *tradeoff*.



Learning as a Search

Given a *hypothesis space*, *data*, and a *bias*, the problem of learning can be reduced to one of search.



Next lecture

Evaluation

18th August 2023



IT496: Introduction to Data Mining



Lecture 09

Evaluation - I

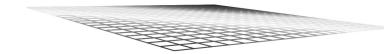
[Schemes for Data Split and Handling Bias-Variance]

Arpit Rana
17th August 2023

Learning = Representation + Evaluation + Optimization

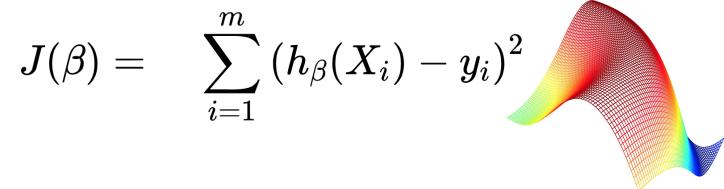
Representation ✓

Choosing a representation of the learner: the *hypotheses space* or the *model class* – the set of models that it can possibly learn.

$$h_{\beta}(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_m X_m$$
$$= \sum_{i=1}^m \beta_i X_i$$


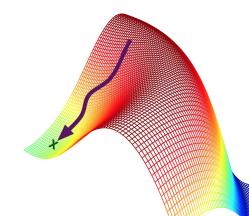
Evaluation

Choosing an evaluation function (also called objective function, utility function, loss function, or scoring function) is needed to distinguish good classifiers from bad ones.



Optimization

Choosing a method to search among the models in the hypothesis space for the highest-scoring one.

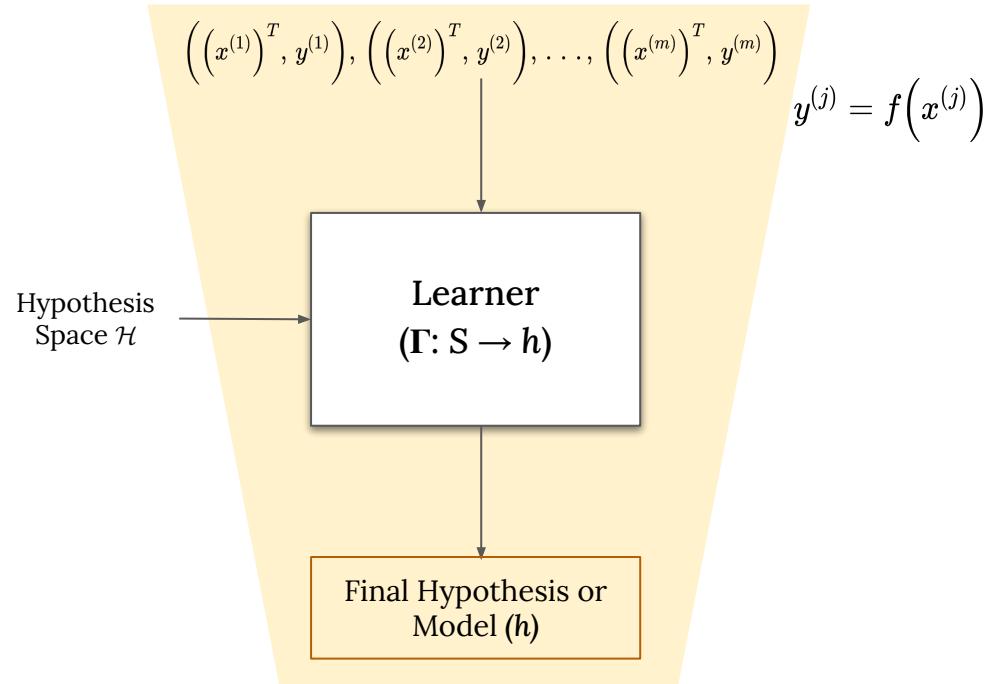
$$\min J(\beta)$$


Experimental Evaluation of Learning Algorithms

The overall objective of the Learning Algorithm is to find a hypothesis that -

- is consistent (i.e., fits the training data), but more importantly,
- generalizes well for previously unseen data.

Experimental Evaluation defines ways to Measure the Generalizability of a Learning Algorithm.



Given a *representation*, *data*, and a *bias*, the learning algorithm returns a final hypothesis.

Experimental Evaluation of Learning Algorithms

Sample Error

The *sample error* of hypothesis h with respect to the target function f and data sample S is:

$$\text{error}_S(h) = \frac{1}{n} \sum_{x \in S} \delta(h(x), f(x))$$

It is impossible to assess *true error*, so we try to estimate it using *sample error*.



True Error

The *true error* of hypothesis h with respect to the target function f and the distribution D is the probability that h will misclassify an instance drawn at random according to D :

$$\text{error}_D(h) = P_{x \in D}[h(x) \neq f(x)]$$

Generalizing to Unseen Data

The error on the training set is called the *training error* (a.k.a. *resubstitution error* and *in-sample error*).

- The training error is not, in general a good indicator of performance on unseen data. It's often too optimistic.
- Why?

Generalizing to Unseen Data

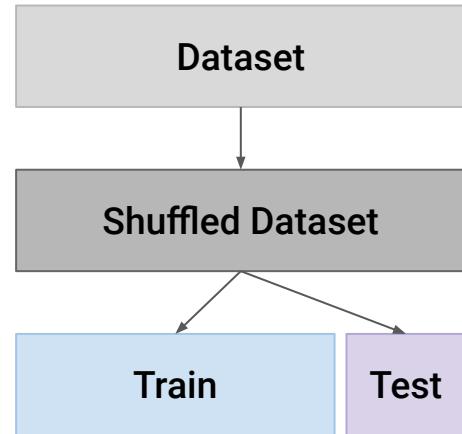
To predict future performance, we need to measure error on an *independent dataset*:

- We want a dataset that has played no part in creating the model.
- This second dataset is called the test set.
- The error on the test set is called the *test error* (a.k.a. *out-of-sample error* and *extra-sample error*).

Given a sample data S, there are methodologies to better approximate the true error of the model.

Holdout Method

- Shuffle the dataset and partition it into two disjoint sets:
 - **training set** (e.g., 80% of the full dataset); and
 - **test set** (the rest of the full dataset).
- Train the estimator on the training set.
- Test the model (evaluate the predictions) on the test set.



It is essential that the test set is not used in any way to create the model. Don't even look at it!

- 'Cheating' is called *leakage*.
- 'Cheating' is one cause of *overfitting*

Holdout Method: Class Exercise

Standardization, as we know, is about scaling the data. It requires calculation of the *mean* and *standard deviation*.

When should the mean and standard deviation be calculated? And Why?

- (a) before splitting, on the entire dataset, or
- (b) after splitting, on just the training set, or
- (c) after splitting, on just the test set, or
- (d) after splitting, on the training and test sets separately,

What to do when the model is deployed?

Facts about Holdout Method

- The disadvantages of this method are:
 - Results can vary quite a lot across different runs.
 - Informally, you might get lucky – or unlucky
i.e., in any one split, the data used for training or testing might not be *representative*.
- We are training on only a subset of the available dataset, perhaps as little as 50% of it.
From so little data, we may learn a worse model and so our error measurement may be pessimistic.
- In practice, we only use the holdout method when we have a very large dataset. The size of the dataset mitigates the above problems.
- When we have a smaller dataset, we use a *resampling* method:
 - The examples get re-used for training and testing.

K-fold Cross-Validation Method

The most-used *resampling* method is k -fold cross-validation:

- Shuffle the dataset and partition it into k disjoint subsets of equal size.
 - Each of the partitions is called a *fold*.
 - Typically, $k=10$, so you have 10 folds.
- You take each fold in turn and use it as the test set, training the learner on the remaining folds.
- Clearly, you can do this k times, so that each fold gets 'a turn' at being the test set.
 - By this method, each example is used exactly once for testing, and $k-1$ times for training.

K-fold Cross-Validation: Pseudocode

- Shuffle the dataset D and partition it into k disjoint equal-sized subsets, D_1, \dots, D_k
- for $i = 1$ to k :
 - train on $D \setminus D_i$
 - make predictions for D_i
 - measure error (e.g. MAE)
- Report the mean of the errors

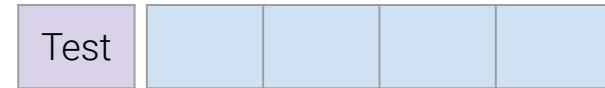
$k=5$ folds



.

.

.



Facts about K-fold Cross-Validation

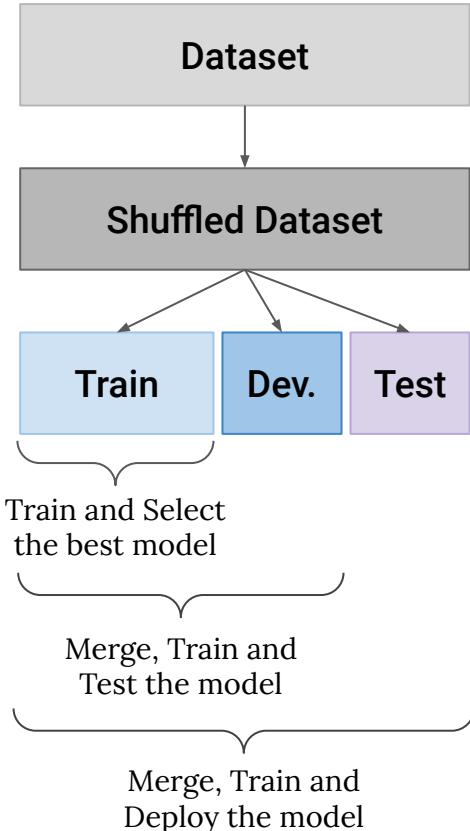
- The disadvantages of this method are:
 - The number of folds is constrained by the size of the dataset and the desire sometimes on the part of statisticians to have folds of at least 30 examples.
 - It can be costly to train the learning algorithm k times.
 - There may still be some variability in the results due to 'lucky'/'unlucky' splits.
- The extreme is $k = n$, also known as *leave-one-out cross-validation* or LOOCV.

Nested K-fold Cross-Validation Method

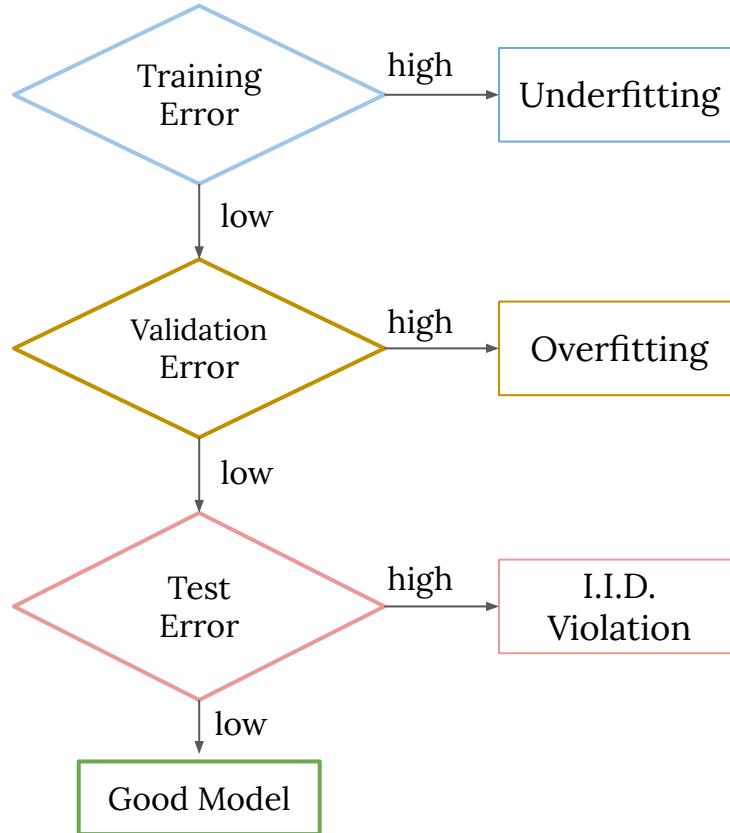
In case of *hyperparameter* (parameters of the model class, not of the individual model) or *parameter tuning*, we partition the whole dataset into three disjoint sets:

- A training set to train candidate models.
- A validation set, (a.k.a. a *development set* or *dev set*) to evaluate the candidate models and choose the best one.
- A test set to do a final unbiased evaluation of the best model.

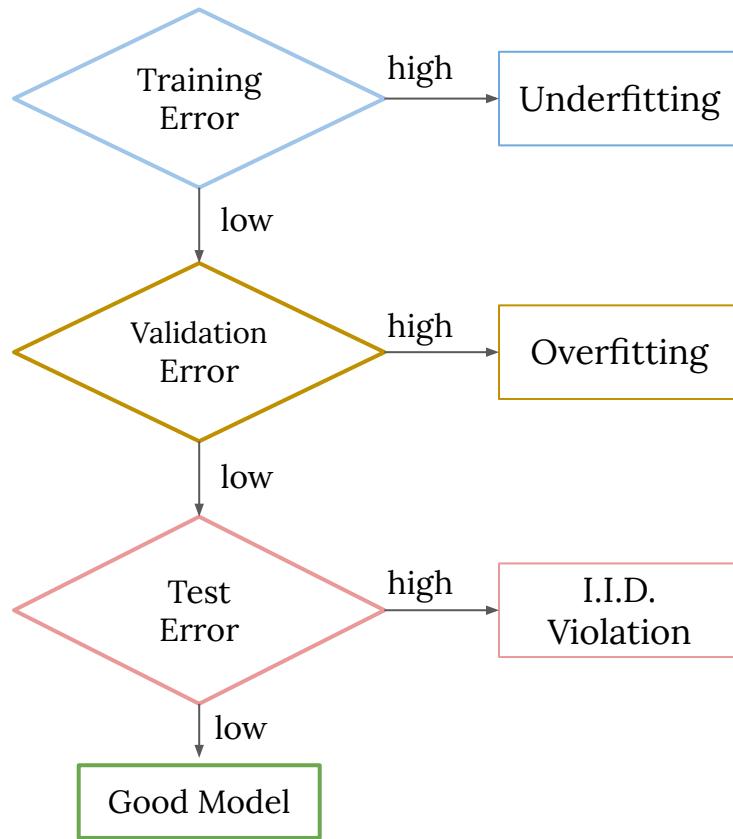
K-fold Cross-Validation can be applied to validation set (inner CV) and test set (outer CV) in a nested way.



Model's Performance



Model's Performance



Underfitting	Overfitting
Need More Complex Model	Need Simpler Model
Need Less Regularization	Need More Regularization
Need More Features	Remove Extra Features
More Data Doesn't Work	Need More Data

Next lecture

Evaluation - II

18th August 2023

IT496: Introduction to Data Mining



Lecture 10

Evaluation - II

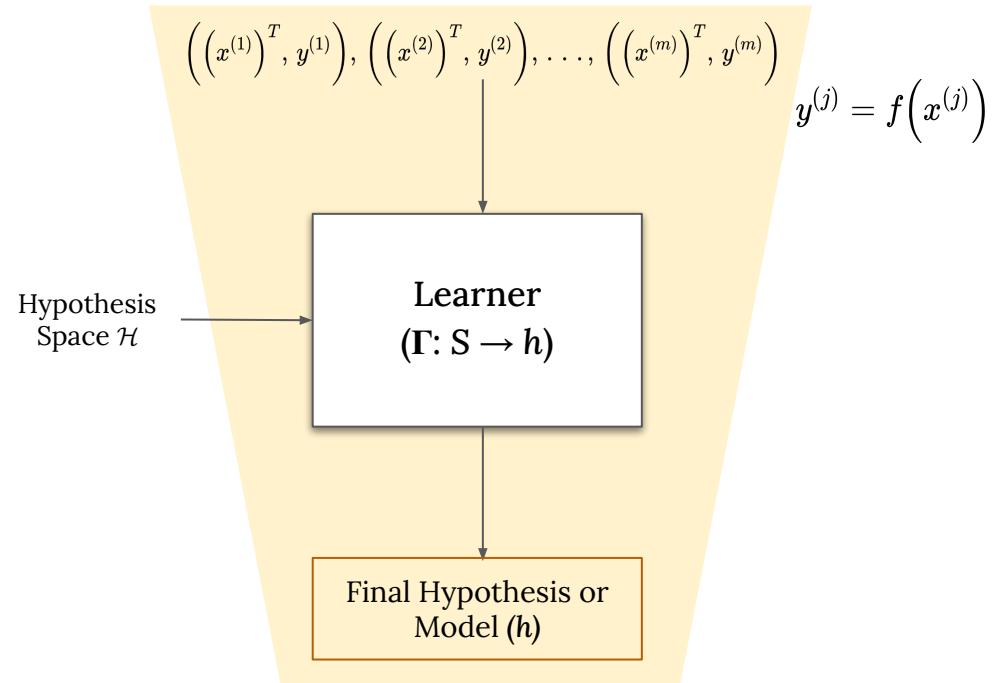
[Evaluation Metrics]

Arpit Rana
18th August 2023

Experimental Evaluation of Learning Algorithms

Given a *representation*, *data*, and a *bias*, the learning algorithm returns a Final Hypothesis (h).

How to Check the Performance of Learning Algorithms?



Evaluation Metrics

Common Measures

Experimental Evaluation of Learning Algorithms

Typical Experimental Evaluation Metrics

- Error
- Accuracy
- Precision/ Recall

Measures for Regression Problems

- Mean Absolute Error

$$MAE = \frac{1}{n} \sum_{x \in S} |h(x) - y|$$

- Squared Error

$$MSE = \frac{1}{n} \sum_{x \in S} (h(x) - y)^2$$

$$RMSE = \sqrt{\frac{1}{n} \sum_{x \in S} (h(x) - y)^2}$$

Which one is better and why?

- Non-differentiability
- Robustness (sensitivity to outliers)
- Unit changes in MSE

Measures for Regression Problems

- Misclassification Rate (a.k.a. Error Rate)

$$MR = \frac{1}{n} \sum_{x \in S} \delta(h(x), y)$$

Where,

$$\delta(h(x), y) = \begin{cases} 1 & h(x) \neq y \\ 0 & otherwise \end{cases}$$

Measures for Classification Problems

		True Class (Actual)		Total	
Confusion Matrix	Hypothesized Class (Predicted)	Positive	Negative		
		Total	P	N	P + N
Positive	True Positive (TP)	False Positive (FP)		P'	
Negative	False Negative (FN)	True Negative (TN)		N'	

$$\text{accuracy} = \frac{TP + TN}{P + N}$$

$$\begin{aligned}\text{error rate} &= 1 - \text{accuracy} \\ &= \frac{FP + FN}{P + N}\end{aligned}$$

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{TP}{P'}$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{TP}{P}$$

$$\text{Specificity} = \frac{TN}{FP + TN} = \frac{TN}{N}$$

Measures for Classification Problems

		True Class (Actual)		Total
Confusion Matrix	Hypothesized Class (Predicted)	Positive	Negative	
		Total	P'	
Positive	True Positive (TP)	False Positive (FP)		
Negative	False Negative (FN)	True Negative (TN)		
Total	P	N	P + N	

F measure: weighted harmonic mean of precision and recall.

$$\begin{aligned}
 F &= \frac{1}{\alpha \cdot \frac{1}{\text{Precision}} + (1 - \alpha) \cdot \frac{1}{\text{Recall}}} \\
 &= \frac{(\beta^2 + 1) \text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}}
 \end{aligned}$$

$$\text{where, } \beta^2 = \frac{1 - \alpha}{\alpha}$$

$$\alpha \in [0, 1] \text{ and } \beta \in [0, \infty]$$

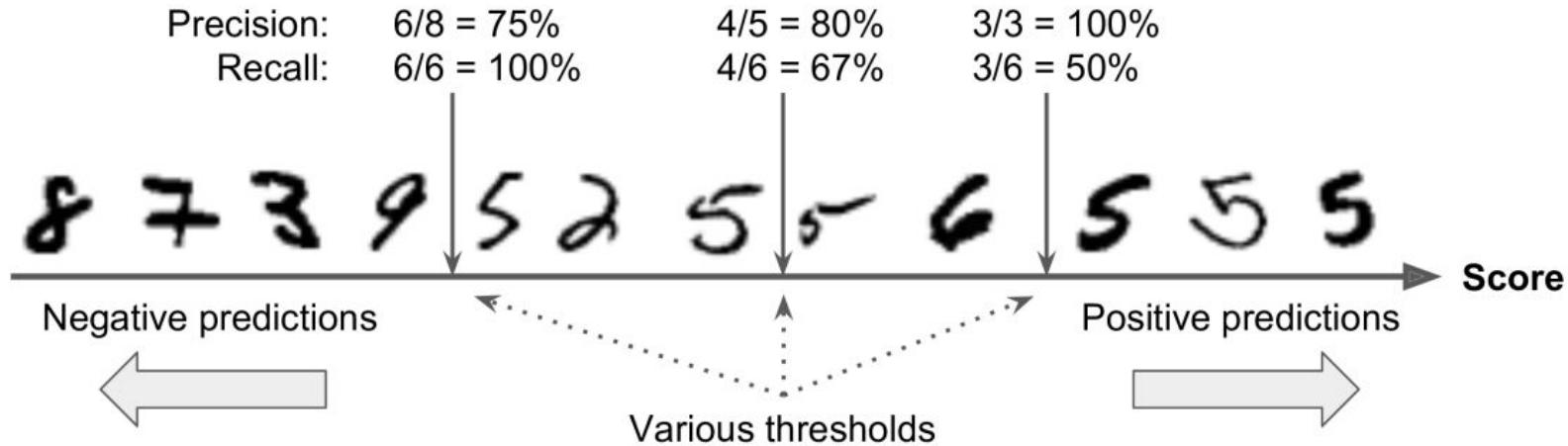
For $\alpha = \frac{1}{2}$, $\beta = 1$, F measure will be balanced and is known as F_1 measure.

Measures for Classification Problems

What metric would you use to measure the performance of the following classifiers.

- A classifier to detect videos that are safe for kids.
- A classifier to detect shoplifters in surveillance images.

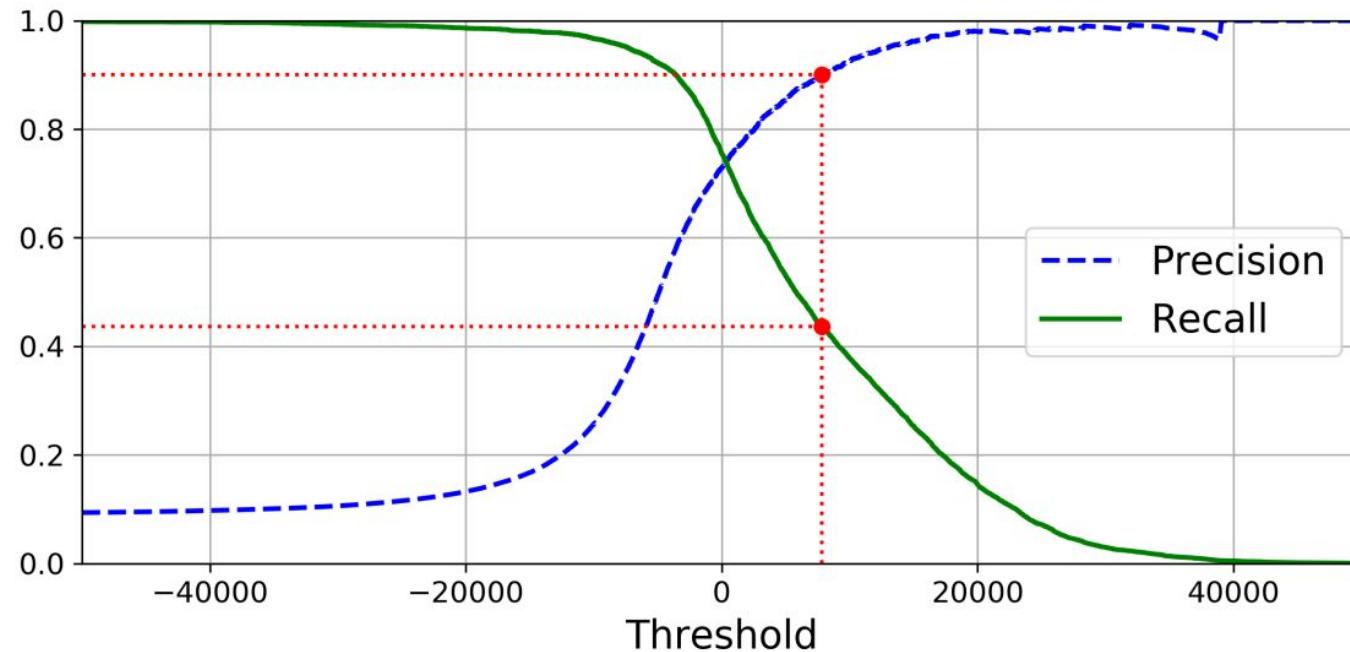
Precision/Recall Trade-off



- Images are ranked by their classifier (*whether the image is 5 or not*) score.
- Those above the chosen decision threshold are considered positive.
- The higher the threshold, the lower the recall, but (in general) the higher the precision.

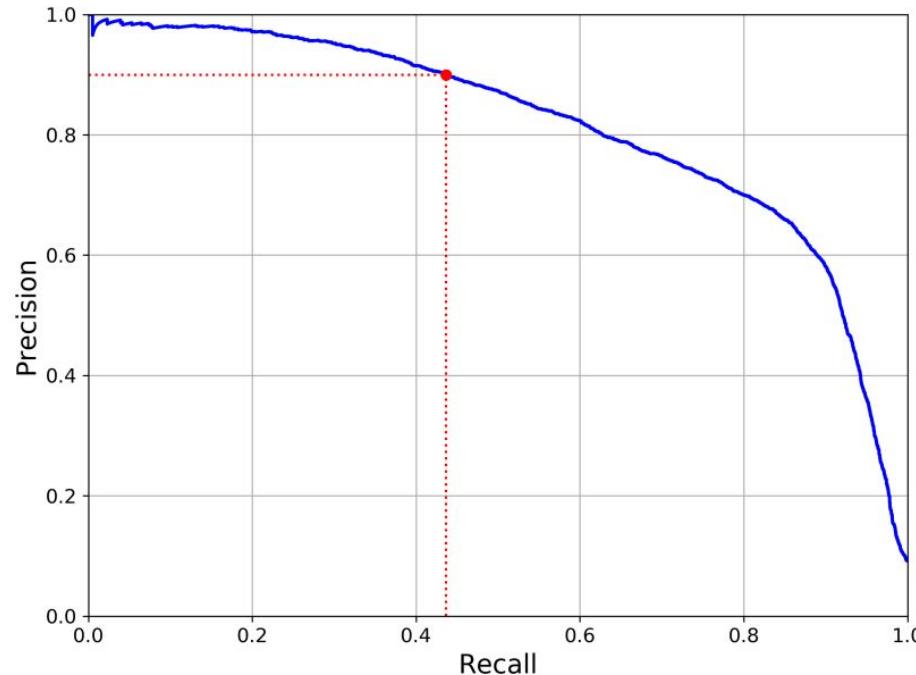
Precision/Recall Trade-off

How do you decide which threshold to use?



Precision/Recall Trade-off

How do you decide which threshold to use?



Precision/Recall Trade-off

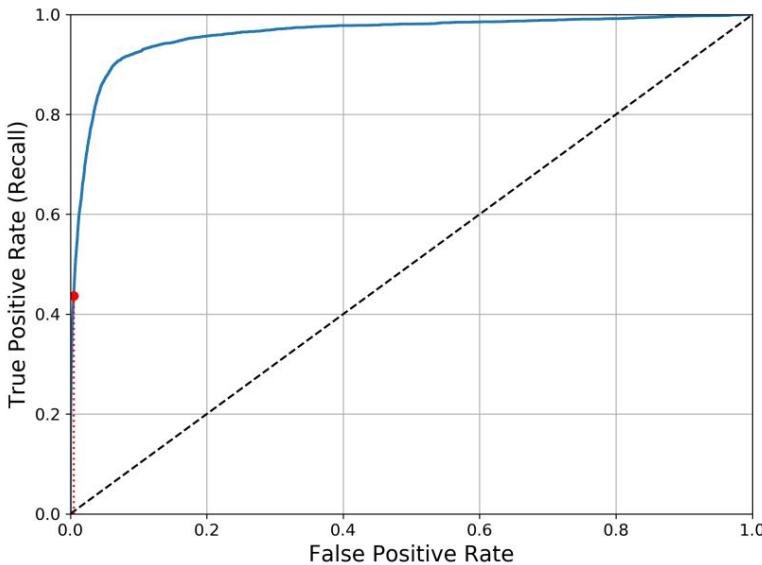
- A high-precision classifier is not very useful if its recall is too low!
- If someone says “let’s reach 99% precision,” you should ask, “at what recall?”

To take recall into consideration, we use other measures.

The ROC Curve

- The receiver operating characteristic (ROC) curve is another common tool used with binary classifiers.
- It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the *true positive rate* (TPR, another name for *recall* or *sensitivity*) against the *false positive rate* (FPR, $1-\text{specificity}$).

The ROC Curve



- Once again there is a tradeoff: the higher the recall (TPR), the more false positives (FPR) the classifier produces.
- The dotted line represents the ROC curve of a purely random classifier;
- A good classifier stays as far away from that line as possible (toward the top-left corner).

AUC: Area Under the (ROC) Curve

- One way to compare classifiers is to measure the area under the curve (AUC).
- A perfect classifier will have a ROC AUC equal to 1, whereas a purely *random* classifier will have a ROC AUC equal to 0.5.

Note: As a rule of thumb, you should prefer the PR (precision-recall) curve whenever the positive class is rare or when you care more about the false positives than the false negatives, and the ROC curve otherwise.

Next lecture

Loss Functions

22nd August 2023

IT496: Introduction to Data Mining



Lecture 11

Evaluation - III
[Loss Functions]

Arpit Rana
28th August 2023

Loss Functions

Commonly used Loss Functions

Error Rate to Loss Function

So far, we assumed that the *optimal fit* is the hypothesis that minimizes the *error rate*: the proportion of times that $h(x) \neq y$ for an (x, y) example.

However, different errors may have different costs.

Example:

It is worse to classify *non-spam* as *spam* than to classify *spam* as *non-spam*.

- *error rate* = 1%, mostly classifying *spam* as *non-spam*,
- *error rate* = 0.5%, mostly classifying *non-spam* as *spam*.

Loss Function: Definition

The loss function $L(x, y, \hat{y})$ is defined as the amount of utility lost by predicting $h(x) = \hat{y}$ when the correct answer is $f(x) = y$:

$$\begin{aligned} L(x, y, \hat{y}) &= \text{Utility(result of using } y \text{ given an input } x) - \text{Utility(result of using } \hat{y} \text{ given an input } x) \\ &= \text{Utility}(f(x)) - \text{Utility}(h(x)) \end{aligned}$$

- Often a simplified version is used, $L(y, \hat{y})$, that is independent of x .

For example,

$$L(\text{spam}, \text{non-spam}) = 1, L(\text{non-spam}, \text{spam}) = 10$$

We choose the learner that *minimizes* the *expected loss* for *all input-output pairs* it will see.

Generalization Loss

Let \mathcal{E} be the set of all possible input–output examples follow a prior probability distribution $P(X, Y)$.

Then the expected *generalization loss* for a hypothesis h (with respect to loss function L) is–

$$GenLoss_L(h) = \sum_{(x,y) \in \xi} L(y, h(x))P(x, y).$$

The best hypothesis h^* , is the one with the minimum expected generalization loss:

$$\dot{h} = \arg \min_{h \in \mathcal{H}} GenLoss_L(h)$$

Empirical Loss

Because $P(X, Y)$ is not known in most cases, the learner can only estimate generalization loss with *empirical loss* on a set of examples D of size N .

$$EmpLoss_{L,D}(h) = \sum_{(x,y) \in D} L(y, h(x)) \frac{1}{N}.$$

The estimated best hypothesis \hat{h}^* , is the one with the minimum expected empirical loss:

$$\hat{h}^* = \arg \min_{h \in \mathcal{H}} EmpLoss_{L,D}(h)$$

There are four reasons why \hat{h}^* may differ from the true function, f : *unrealizability, variance, noise, and computational complexity*.

Commonly Used Loss Functions

There's no one-size-fits-all loss function to algorithms in machine learning.

- There are various factors involved in choosing a loss function for specific problem, e.g.,
 - type of machine learning algorithm chosen,
 - ease of calculating the derivatives, and
 - to some degree the percentage of outliers in the data set.

Broadly, loss functions can be classified into two major categories – *Regression losses* and *Classification losses*.

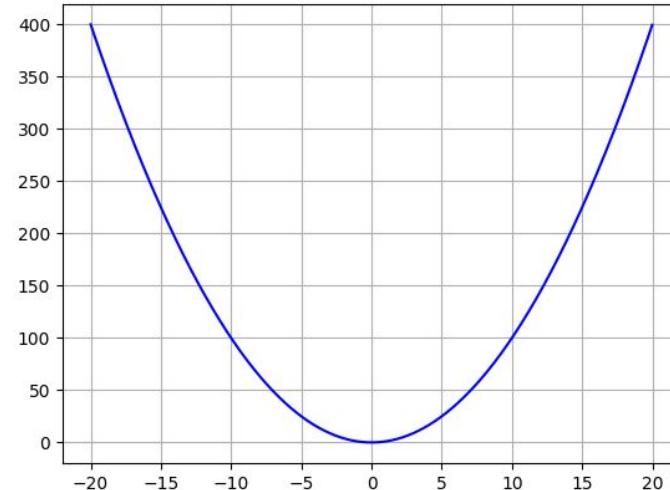
Regression Losses

Mean Squared Error (L2 loss)

This is almost every data scientist's preference when it comes to loss functions for regression. Because most variables can be modeled into a Gaussian distribution.

This is differentiable.

$$MSE = \frac{1}{n} \sum_{x \in S} (h(x) - y)^2$$



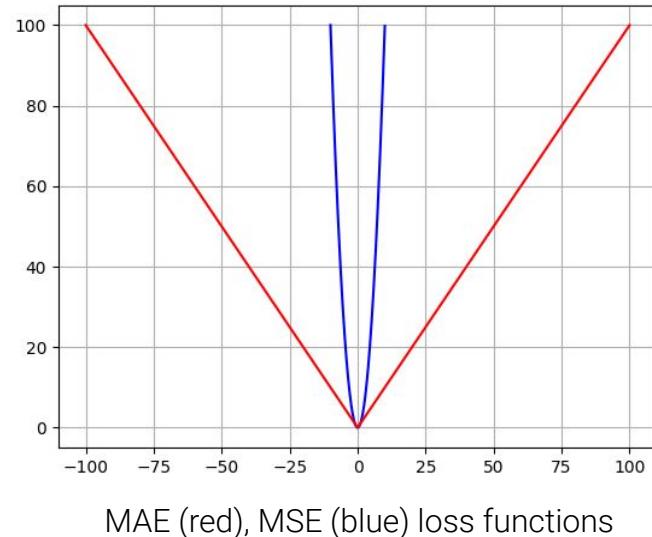
Mean Absolute Error (L1 loss)

This is one of the most simple yet *robust* loss functions used for regression models.

Regression problems may have variables that are not strictly *Gaussian* in nature due to the presence of outliers (values that are very different from the rest of the data).

L1 loss is an ideal option in such cases.

$$MAE = \frac{1}{n} \sum_{x \in S} |h(x) - y|$$

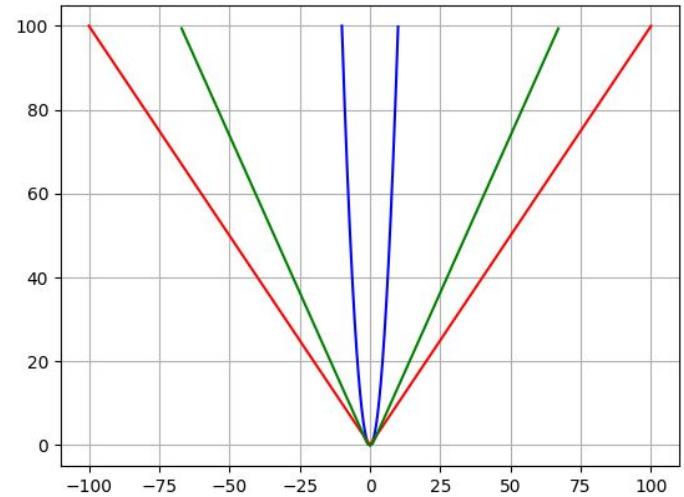


Huber Loss

Now we know that the MSE is great for outliers while the MAE is great for ignoring them. But what about something in the middle?

The Huber Loss offers the best of both worlds by balancing the MSE and MAE together. We can define it using the following piecewise function:

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \leq \delta, \\ \delta|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$



MAE (red), MSE (blue), and Huber (green) Loss functions

Classification Losses

Entropy

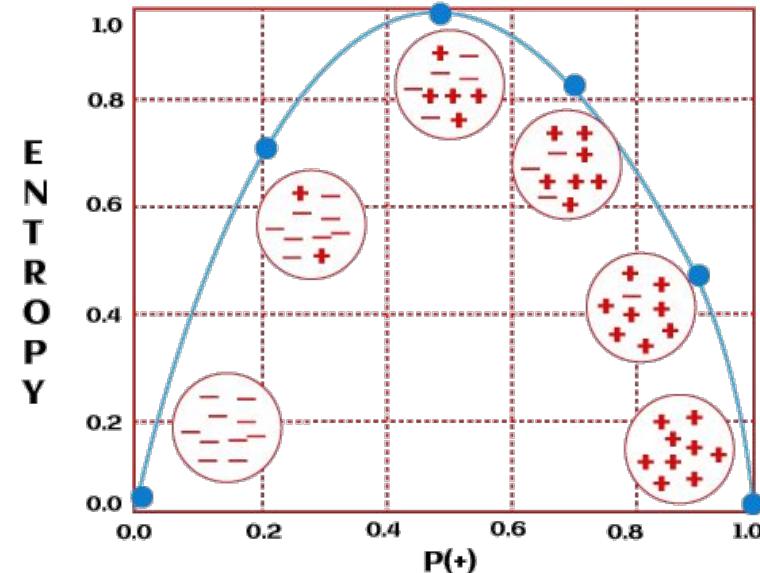
Entropy is the measure of uncertainty of a variable: the more uncertain it is, the higher the entropy is, the higher the surprise is, and the lower the probability is.

$$\text{Entropy} = - \sum_x p(x) \log_e (p(x))$$

Here,

x is a class

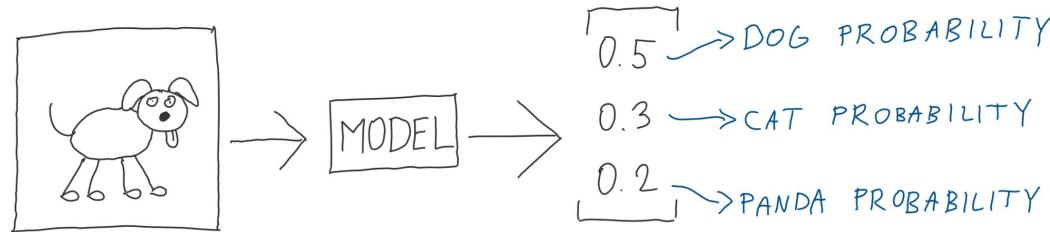
$p(x)$ is the probability of class x in the dataset



Cross Entropy Loss Function

Which class is on the image – dog, cat, or panda? It can only be one of them (multi-class classification).

Let's have an image of a dog.



The prediction is a *probability vector*, meaning it represents predicted probabilities of all classes, summing up to 1.

Cross Entropy Loss Function

- Cross-Entropy

The general formula, used for calculating loss among two probability vectors: the target and the prediction.

The more we are away from our target, the more the error grows – similar idea to the square error.

$$L = - \sum_x p(x) \log (q(x))$$

Here,

x is a class

$p(x)$ is the probability of class x in target

$q(x)$ is the probability of class x in prediction

Cross Entropy Loss Function

The target for multi-class classification is a *one-hot vector*, meaning it has 1 on a single position and 0's everywhere else.

TARGET	PREDICTION
1	0.5
0	0.3
0	0.2

We will start by calculating the loss for each class separately and then summing them. The loss for each separate class is computed like this:

$$\text{Loss for class } X = - \underbrace{P(X)}_{\substack{\text{probability} \\ \text{of class } X \\ \text{in TARGET}}} \cdot \log \underbrace{q(X)}_{\substack{\text{probability} \\ \text{of class } X \\ \text{in PREDICTION}}}$$

Cross Entropy Loss Function

$$\begin{aligned}\text{Loss For CAT} &= -p(\text{CAT}) \cdot \log q(\text{CAT}) \\ &= -0 \cdot \log q(\text{CAT}) \\ &= 0\end{aligned}$$

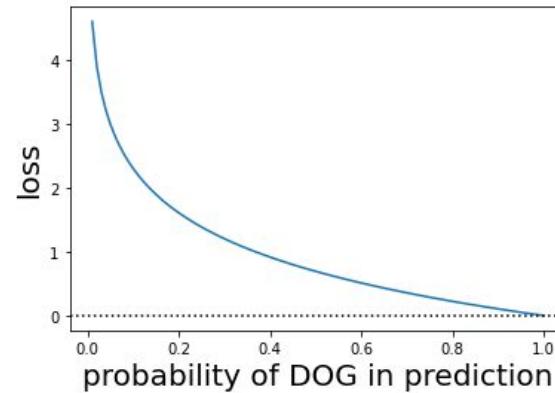
$$\begin{aligned}\text{Loss For PANDA} &= -p(\text{PANDA}) \cdot \log q(\text{PANDA}) \\ &= -0 \cdot \log q(\text{PANDA}) \\ &= 0\end{aligned}$$

$$\begin{aligned}\text{Loss For DOG} &= -p(\text{DOG}) \cdot \log q(\text{DOG}) \\ &= -1 \cdot \log 0.5 \\ &= 0.693...\end{aligned}$$

$$\begin{aligned}\text{Cross-entropy} &= \text{Loss For DOG} + \text{Loss For CAT} + \text{Loss For PANDA} \\ &= 0.693 + 0 + 0 \\ &= 0.693\end{aligned}$$

Cross Entropy Loss Function

Let's see how would the loss behave if the predicted probability was different:



This shows a similar concept to square error – the further away we are from the target, the faster the error grows.

If we want the loss for the whole dataset, we would just sum up the losses of the individual images.

Cross Entropy Loss

- Categorical Cross-Entropy (Multi-class Classification)

If our target is a one-hot vector, we can indeed forget targets and predictions for all the other classes and compute only the loss for the hot class.

This is the negative natural logarithm of our prediction.

$$L = -\log (q(x))$$

x is the class that
is 1 in our target.

This is called *categorical cross-entropy* – a special case of cross-entropy, where our target is a one-hot vector.

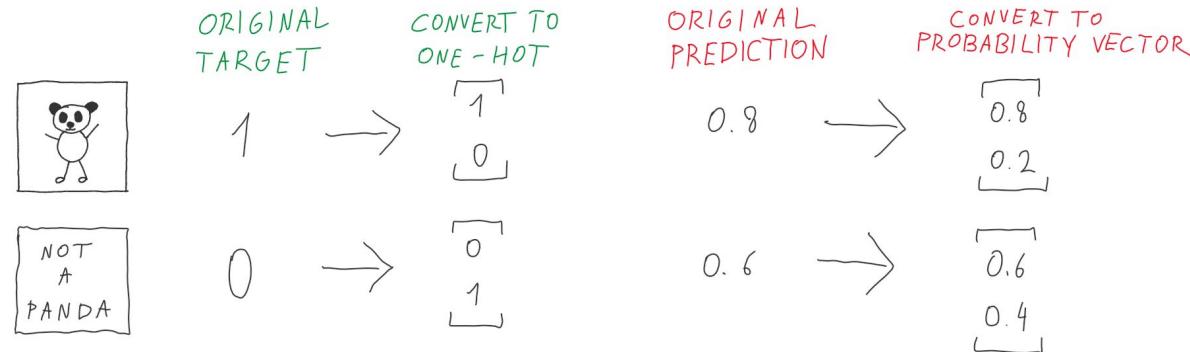
Cross Entropy Loss

- Binary Cross-Entropy (for Binary Classification)

We use binary cross-entropy – a specific case of cross-entropy where target is 0 or 1.

It can be computed with the cross-entropy formula if we convert the target to a one-hot vector like [0,1] or [1,0] and the predictions respectively.

Suppose we want to predict whether the image contains a panda or not.



Cross Entropy Loss

- Binary Cross-Entropy (for Binary Classification)

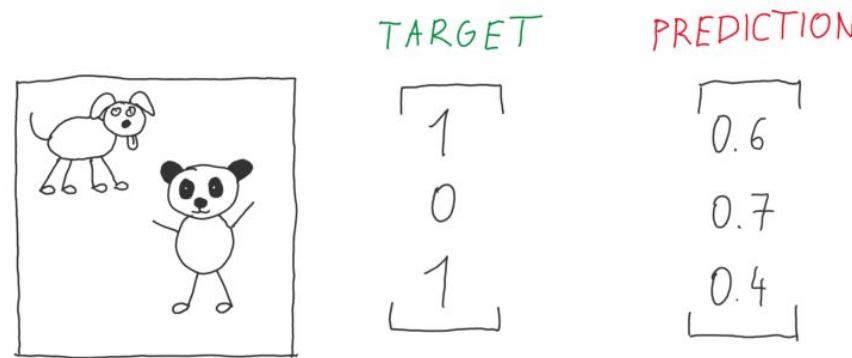
We can compute it even without this conversion, with the simplified formula.

$$L = -(p(x) \log (q(x)) + (1 - p(x)) \log (1 - q(x)))$$

Cross Entropy Loss

- Multi-label Classification (Multiple Binary Class Cross-Entropy)

Our target can represent multiple (or even zero) classes at once.



Notice our target and prediction are not a probability vector. It's possible that there are all classes in the image, as well as none of them.

Cross Entropy Loss

- Multi-label Classification (Multiple Binary Class Cross-Entropy)

We can look at this problem as multiple binary classification subtasks.

$$\begin{aligned}\text{Binary Cross-entropy}_{\text{DOG}} &= - \left(p(x) \cdot \log q(x) + (1-p(x)) \cdot \log (1-q(x)) \right) \\ &= - \left(1 \cdot \log 0.6 + (1-1) \cdot \log (1-0.6) \right) \\ &= - \left(\log 0.6 + 0 \right) \\ &\approx 0.510...\end{aligned}$$

$$\begin{aligned}\text{Binary Cross-entropy}_{\text{PANDA}} &= - \left(p(x) \cdot \log q(x) + (1-p(x)) \cdot \log (1-q(x)) \right) \\ &= - \left(1 \cdot \log 0.4 + (1-1) \cdot \log (1-0.4) \right) \\ &= - \left(\log 0.4 + 0 \right) \\ &\approx 0.916...\end{aligned}$$

$$\begin{aligned}\text{Binary Cross-entropy}_{\text{CAT}} &= - \left(p(x) \cdot \log q(x) + (1-p(x)) \cdot \log (1-q(x)) \right) \\ &= - \left(0 \cdot \log 0.7 + (1-0) \cdot \log (1-0.7) \right) \\ &= - \left(0 + \log 0.3 \right) \\ &\approx 1.20...\end{aligned}$$

$$\begin{aligned}\text{Total Loss} &= \text{Binary Cross-entropy}_{\text{DOG}} \\ &\quad + \text{Binary Cross-entropy}_{\text{CAT}} \\ &\quad + \text{Binary Cross-entropy}_{\text{PANDA}} \\ &\approx 2.631...\end{aligned}$$

Cross Entropy Loss

- Multi-label Classification (Multiple Binary Class Cross-Entropy)

Our target can represent multiple (or even zero) classes at once.

We compute the binary cross-entropy for each class separately and then sum them up for the complete loss.

$$L = \sum_x L_{binary}(x)$$

Here, $L_{binary}(x)$ is the binary cross-entropy loss for class x.

Other Classification Losses

There are other commonly used loss functions available for classification problems.

- Hinge Loss: Specifically used in SVM
- Kullback Leibler Divergence Loss (KL Loss): Most commonly used for Neural Networks

$$\begin{aligned} D_{KL}(P||Q) &= \sum_x P(x) \log \frac{P(x)}{Q(x)} \\ &= - \sum_x P(x) \log \frac{Q(x)}{P(x)} \\ &= - \left(\sum_x P(x) \log Q(x) - \sum_x P(x) \log P(x) \right) \\ &= H(P, Q) - H(P) \end{aligned}$$

Next lecture

Optimization

29th August 2023



IT496: Introduction to Data Mining



Lecture 12

Optimization - I

[Search Methods]

Arpit Rana
29th August 2023

Components of Supervised Learning

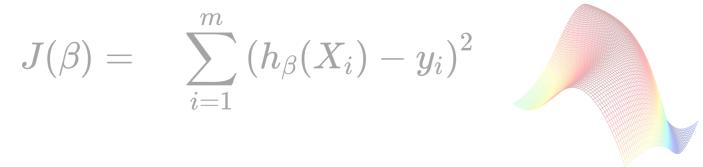
Representation ✓

choosing the set of functions (*hypotheses space or the model class*) that can be learned.

$$h_{\beta}(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_m X_m \\ = \sum_{i=1}^m \beta_i X_i$$

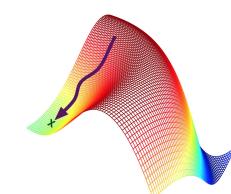

Evaluation ✓

An evaluation function (also called *objective function* or *scoring function*) is needed to distinguish good hypotheses from bad ones.



Optimization

We need a method to search the hypothesis space for the highest-scoring one.

$$\arg \min J(\beta)$$


Hyperparameters

Hyperparameters are parameters of the model class, not of the individual model.

- We define them before training to control the learning process.
- The *learner* algorithm does not learn these (can't be estimated from data).

Example:

```
from sklearn.ensemble import RandomForestClassifier

# Instantiate the model
rf_model = RandomForestClassifier()

# Print hyperparameters
rf_model.get_params
```

```
RandomForestClassifier(
    bootstrap=True,
    ccp_alpha=0.0, class_weight=None,
    criterion='gini',
    max_depth=None,
    max_features='auto',
    max_leaf_nodes=None,
    max_samples=None,
    min_impurity_decrease=0.0,
    min_impurity_split=None,
    min_samples_leaf=1,
    min_samples_split=2,
    min_weight_fraction_leaf=0.0,
    n_estimators=100,
    n_jobs=None, oob_score=False,
    random_state=None,
    verbose=0, warm_start=False)
```

Hyperparameters

Hyperparameters are parameters of the model class, not of the individual model.

- We define them before training to control the learning process.
- The *learner* algorithm does not learn these (can't be estimated from data).

Example:

```
from sklearn.ensemble import RandomForestClassifier

# Instantiate the model
rf_model = RandomForestClassifier()

# Print hyperparameters
rf_model.get_params
```

Some hyperparameters are more important than others.

```
RandomForestClassifier(
    bootstrap=True,
    ccp_alpha=0.0, class_weight=None,
    criterion='gini',
    max_depth=None,
    max_features='auto',
    max_leaf_nodes=None,
    max_samples=None,
    min_impurity_decrease=0.0,
    min_impurity_split=None,
    min_samples_leaf=1,
    min_samples_split=2,
    min_weight_fraction_leaf=0.0,
    n_estimators=100,
    n_jobs=None, oob_score=False,
    random_state=None,
    verbose=0, warm_start=False)
```

Parameters

(Model) Parameters are components of the model whose value can be estimated from data.

- We do not set these manually (we can't in fact!)
- The algorithm will discover these for us (learned during the training).

Example:

```
from sklearn import LogisticRegression

# Instantiate the model
log_reg_clf = LogisticRegression()

# Train the model
log_reg_clf.fit(X_train, y_train)

print(log_reg_clf.coef_)
```

```
array([[-2.88651273e-06,
       -8.23168511e-03,
       7.50857018e-04,
       3.94375060e-04,
       3.79423562e-04,
       4.34612046e-04,
       4.37561467e-04,
       4.12107102e-04,
       -6.41089138e-06,
       -4.39364494e-06, ... ]])
```

- For decision tree or random forest, split column (the attribute chosen for split) and split column value (the value of that attribute chosen for split) are examples of parameters that are learned while training.

Hyperparameter Tuning

Hyperparameter Tuning is choosing the best combination of hyperparameters. But, they can't be estimated from the data.

The following methods are commonly used for tuning the hyperparameters.

- Manual Search
- Grid Search
- Random Search
- Coarse to Fine Search
- Bayesian Search
- Genetic Algorithm

Manual Search/Hand Tuning

In this search, the user himself manually tweak the hyperparameter combinations until the model gets the optimal performance.

- Guess some parameter values based on past experience,
- Train a model, measure its performance on the validation data,
- Analyze the results, and use your intuition to suggest new parameter values.
- Repeat until you have satisfactory performance (or you run out of time, computing budget, or patience).

Pros

- For a skilled practitioner, this can help to reduce computational time

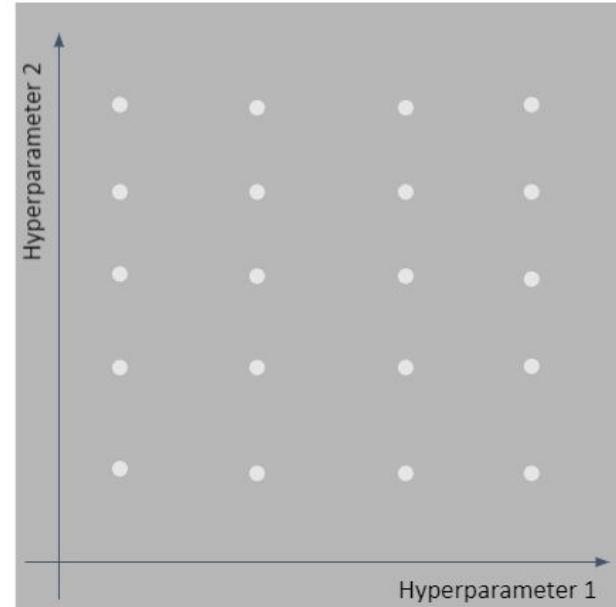
Cons

- Hard to guess even though you really understand the algorithm
- Time-consuming

Grid Search

If there are only a few hyperparameters, each with a small number of possible values, then a more systematic approach called *grid search* is appropriate.

- Try all combinations of values and see which performs best on the validation data.
- Different combinations can be run in parallel on different machines, so if you have sufficient computing resources, this need not be slow.
- Although in some cases model selection has been known to suck up resources on thousand-computer clusters for days at a time.
- if two hyperparameters are independent of each other, they can be optimized separately.



Grid Search

We simply run a random forest classifier with default values and get the predictions for the test set.

Example:

```
# Instantiate and fit random forest classifier
rf_model = RandomForestClassifier()
rf_model.fit(X_train, y_train)

# Predict on the test set and call accuracy
y_pred = rf_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print(accuracy)
```

0.81

Grid Search

Grid Search starts with defining a *search space grid*.

The grid consists of selected hyperparameter names and values, and grid search exhaustively searches the best combination of these given values.

Example:

```
# Define the parameter grid
param_grid = {
    'n_estimators': [50, 100, 200, 300],
    'min_samples_leaf': [1, 5, 10],
    'max_depth': [2, 4, 6, 8, 10],
    'max_features': ['auto', 'sqrt'],
    'bootstrap': [True, False]}

# Instantiate GridSearchCV
model_gridsearch = GridSearchCV(
    estimator=rf_model,
    param_grid=param_grid,
    scoring='accuracy',
    n_jobs=4,
    cv=5,
    refit=True,
    return_train_score=True)
```

Grid search will have to run and compare 240 models ($=4*5*3*2*2$).

For 5-fold cross-validation, grid search will have to evaluate 1200 ($=240*5$) model performances.

Grid Search

```
# Record the current time
start = time()

# Fit the selected model
model_gridsearch.fit(X_train, y_train)

# Print the time spend and number of models ran
print("GridSearchCV took %.2f seconds for %d candidate parameter settings." % ((time() -
start), len(model_gridsearch.cv_results_['params'])))

# Predict on the test set and call accuracy
y_pred_grid = model_gridsearch.predict(X_test)
accuracy_grid = accuracy_score(y_test, y_pred_grid)
```

GridSearchCV took 247.79 seconds for 240 candidate parameter settings.

0.88

Random Search

In random search, we define distributions for each hyperparameter which can be defined *uniformly* or with a *sampling method*.

For example, if there are 500 values in the distribution and if we input `n_iter=50` then random search will randomly sample 50 values to test.

Since random search does not try every hyperparameter combination, it does not necessarily return the best performing values, but it returns a relatively good performing model in a significantly shorter time.

Random Search

In random search, we define distributions for each hyperparameter which can be defined *uniformly* or with a *sampling method*.

```
# specify distributions to sample from
param_dist = {
    'n_estimators': list(range(50, 300, 10)),
    'min_samples_leaf': list(range(1, 50)),
    'max_depth': list(range(2, 20)),
    'max_features': ['auto', 'sqrt'],
    'bootstrap': [True, False]}

# specify number of search iterations
n_iter = 50

# Instantiate RandomSearchCV
model_random_search = RandomizedSearchCV(
    estimator=rf_model,
    param_distributions=param_dist,
    n_iter=n_iter)
```

Random Search

```
# Record the current time
start = time()

# Fit the selected model
model_random_search.fit(X_train, y_train)

# Print the time spend and number of models ran
print("RandomizedSearchCV took %.2f seconds for %d candidate parameter settings." %
((time() - start), len(model_random_search.cv_results_['params'])))

# Predict on the test set and call accuracy
y_pred_random = model_random_search.predict(X_test)
accuracy_random = accuracy_score(y_test, y_pred_random)
```

RandomizedSearchCV took 64.17 seconds for 50 candidate parameter settings.

0.86

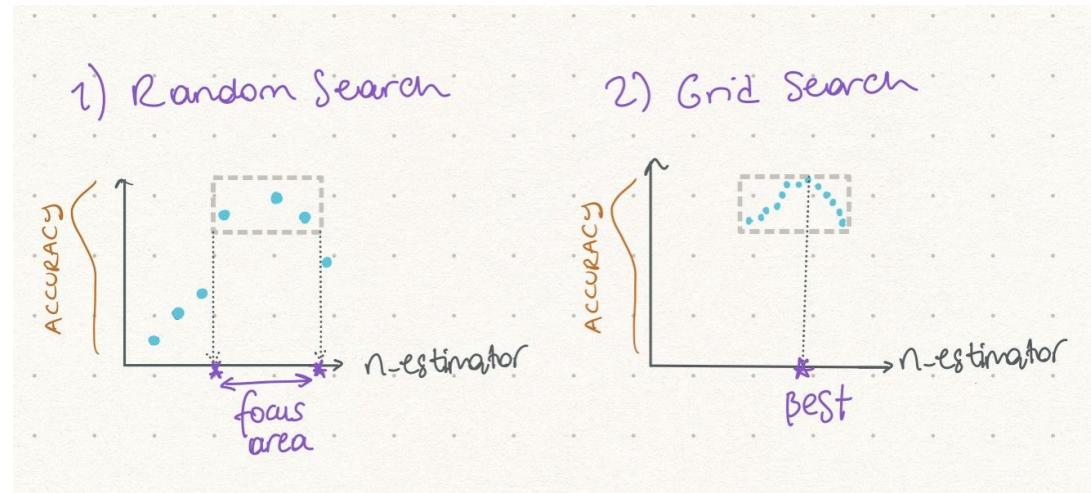
Coarse to Fine Search

For Grid Search, an increased number of hyperparameters easily becomes a bottleneck. To prevent this inefficiency, we can combine grid search with random search.

- In coarse-to-fine tuning, we start with a random search to find the promising value ranges for each hyperparameter.
- After getting focus area for each hyperparameter using random search, we can define the grid accordingly for grid search to find the best values amongst them.
- For example, if the random search returns high performance for n_estimators between 150 and 200, this is the range we want grid search to focus on.

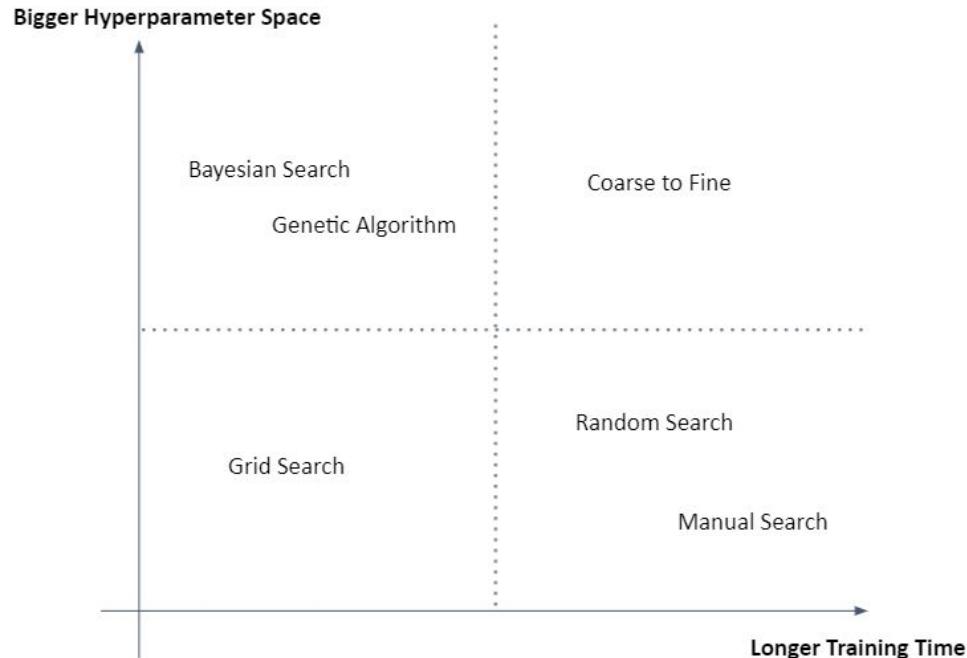
Coarse to Fine Search

For Grid Search, an increased number of hyperparameters easily becomes a bottleneck. To prevent this inefficiency, we can combine grid search with random search.



When to Use What

When we are training a deep neural network with huge hyperparameter space, it is preferable to use a Manual Search or Random Search method rather than using the Grid Search method.



Next lecture

Optimization - II

11th September 2023

IT496: Introduction to Data Mining



Lecture 13

Optimization - II

[Gradient Descent Algorithm]

Arpit Rana
12th September 2023

Components of Supervised Learning

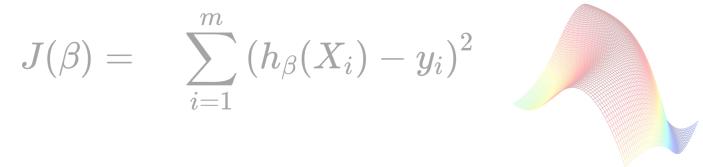
Representation ✓

choosing the set of functions (*hypotheses space or the model class*) that can be learned.

$$h_{\beta}(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_m X_m \\ = \sum_{i=1}^m \beta_i X_i$$

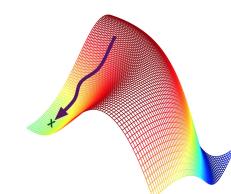

Evaluation ✓

An evaluation function (also called *objective function* or *scoring function*) is needed to distinguish good hypotheses from bad ones.



Optimization

We need a method to search the hypothesis space for the highest-scoring one.

$$\arg \min J(\beta)$$


Overall Objective

Let \mathcal{E} be the set of all possible input-output examples that follow a prior probability distribution $P(X, Y)$.

Then the expected *generalization loss* for a hypothesis h (with respect to loss function L) is-

$$GenLoss_L(h) = \sum_{(x,y) \in \xi} L(y, h(x))P(x, y).$$

The best hypothesis h^* , is the one with the minimum expected generalization loss:

$$h^* = \arg \min_{h \in \mathcal{H}} GenLoss_L(h)$$

Overall Objective

Because $P(X, Y)$ is not known in most cases, the learner can only estimate generalization loss with *empirical loss* on a set of examples D of size N .

$$EmpLoss_{L,D}(h) = \sum_{(x,y) \in D} L(y, h(x)) \frac{1}{N}.$$

The estimated best hypothesis h^* , is the one with the minimum expected empirical loss:

$$\hat{h}^* = \arg \min_{h \in \mathcal{H}} EmpLoss_{L,D}(h)$$

Training

Training finds the best hypothesis within the hypothesis space.

$$y = f(x) = bx + a$$

Parameters:
 $a=2, b=3$



$$f(x) = 3x + 2$$

One common way to find the final hypothesis is by minimizing a loss function using Gradient Descent algorithm.

Gradient Descent

Gradient Descent is a generic method to tweak parameters iteratively in order to minimize a cost (a.k.a. loss) function.

- It is a search in the model's parameter space for values of the parameters that minimize the *loss function*.

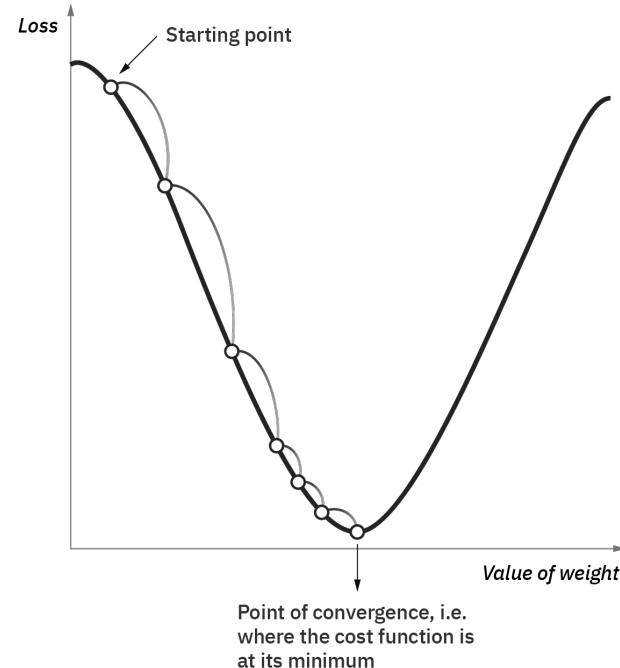
Gradient Descent

Conceptually:

- It starts with an initial guess for the values of the parameters (called *random initialization*).
- Then repeatedly:
 - It updates the parameter values — hopefully to reduce the loss.

Ideally, it keeps doing this until convergence — changes to the parameter values do not result in lower loss.

The key to this algorithm is how to update the parameter values.



Gradient Descent: The Update Rule

To update the parameter values to reduce the loss:

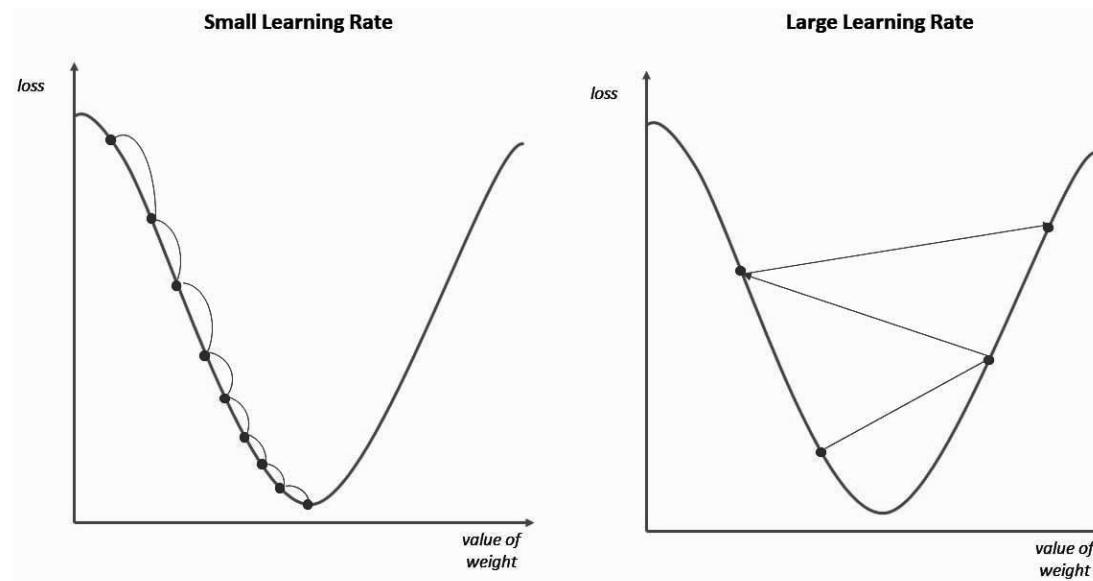
- Compute the *gradient vector*.
 - But this points 'uphill' and we want to go 'downhill'.
 - And we want to make 'baby steps' (see later), so we use a *learning rate*, α which is between 0 and 1.
- So subtract α times the gradient vector from \mathbf{w}

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$$

Gradient Descent: The Learning Rate

The size of the steps is determined by the *learning rate*.

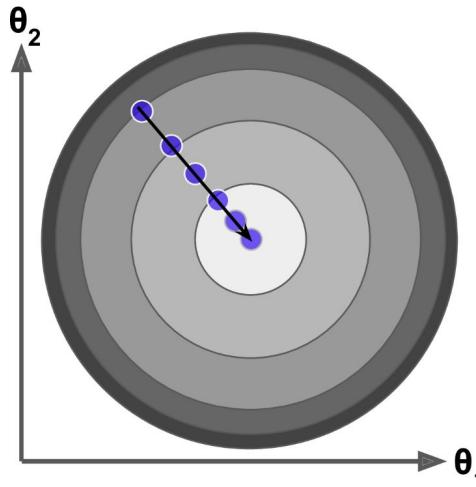
- If the learning rate is too small, it will take many updates until convergence:
- If the learning rate is too big, the algorithm might jump across the valley (overshoot) – it may even end up with higher loss than before, making the next step bigger.



Gradient Descent: Sensitive to Scaling of Features

For Gradient Descent, we do need to scale the features.

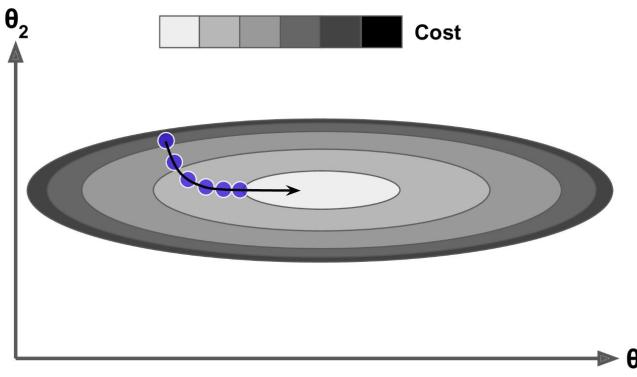
- If features have different ranges, it affects the shape of the 'bowl'.
 - E.g. features 1 and 2 have similar ranges of values – a 'bowl':
- The algorithm goes straight towards the minimum.



Gradient Descent: Sensitive to Scaling of Features

E.g. feature 1 has smaller values than feature 2 – an elongated 'bowl':

- Since feature 1 has smaller values, it takes a larger change in θ_1 to affect the loss function, which is why it is elongated.
- It takes more steps to get to the minimum – steeply down but not really towards the goal, followed by a long march down a nearly flat valley.
- It makes it more difficult to choose a value for the learning rate that avoids *divergence*: a value that suits one feature may not suit another.



Gradient Descent Algorithm Pseudocode

w \leftarrow any point in the parameter space

while not converged **do**

for each w_i **in** **w do**

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$$

Gradient Descent in Action

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$$

For univariate regression, the squared-error loss is quadratic, so the partial derivative will be linear.

$$\begin{aligned}\frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(x))^2 = 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(x)) \\ &= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i} (y - (w_1 x + w_0)).\end{aligned}$$

Gradient Descent in Action

Applying this to both w_0 and w_1 we get:

$$\frac{\partial}{\partial w_0} Loss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)); \quad \frac{\partial}{\partial w_1} Loss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)) \times x.$$

$$w_0 \leftarrow w_0 + \alpha (y - h_{\mathbf{w}}(x)); \quad w_1 \leftarrow w_1 + \alpha (y - h_{\mathbf{w}}(x)) \times x.$$

Here, loss covers only one training example.

Batch Gradient Descent

For N training examples, we want to minimize the sum of the individual losses for each example.

- The derivative of a sum is the sum of the derivatives, so we have:

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)); \quad w_1 \leftarrow w_1 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)) \times x_j.$$

- We have to sum over all N training examples for every step, and there may be many steps.
- A step that covers all the training examples is called an epoch.

These updates constitute the *batch gradient descent learning rule for univariate linear regression*.

Gradient Descent: Multivariate Linear Regression

We can easily extend to multivariable linear regression problems, in which each example x_j is an n -element vector

$$h_{\mathbf{w}}(\mathbf{x}_j) = w_0 + w_1 x_{j,1} + \cdots + w_n x_{j,n} = w_0 + \sum_i w_i x_{j,i}.$$

Vectorized form -

$$h_{\mathbf{w}}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^\top \mathbf{x}_j = \sum_i w_i x_{j,i}.$$

The best vector of weights, \mathbf{w}^* , minimizes squared-error loss over the examples:

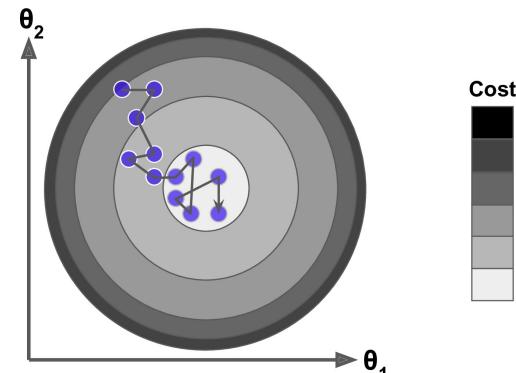
$$\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j).$$

$$w_i \leftarrow w_i + \alpha \sum_j (y_j - h_{\mathbf{w}}(\mathbf{x}_j)) \times x_{j,i}.$$

Stochastic Gradient Descent

As we saw, in each iteration, Batch Gradient Descent does a calculation on the entire training set, which, for large training sets, may be slow.

- Stochastic Gradient Descent (SGD), on each iteration, picks just one training example x_i at random and computes the gradients on just that one example.
- This gives huge speed-up.
 - It enables us to train on huge training sets since only one example needs to be in memory in each iteration.
 - But, because it is stochastic (the randomness), the loss will not necessarily decrease on each iteration:
- On average, the loss decreases, but in any one iteration, loss may go up or down.
- Eventually, it will get close to the minimum, but not necessarily optimal.



Stochastic Gradient Descent

Simulated Annealing

- As we discussed, SGD does not settle at the minimum.
- One solution is to gradually reduce the learning rate:
 - Updates start out 'large' so you make progress.
 - But, over time, updates get smaller, allowing SGD to settle at or near the global minimum.
- The function that determines how to reduce the learning rate is called the learning schedule.
 - Reduce it too quickly and you may not converge on or near to the global minimum.
 - Reduce it too slowly and you may still bounce around a lot and, if stopped after too few iterations, may end up with a suboptimal solution.

Mini-Batch Gradient Descent

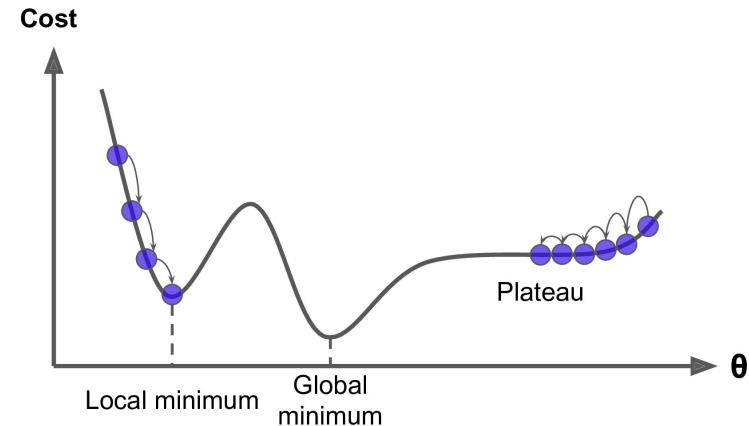
Batch Gradient Descent computes gradients from the full training set and Stochastic Gradient Descent computes gradients from just one example.

- Mini-Batch Gradient Descent lies between the two:
 - It computes gradients from a small randomly-selected subset of the training set, called a mini-batch.
- Since it lies between the two:
 - It may bounce less and get closer to the global minimum than SGD...
 - ...although both of them can reach the global minimum with a good learning schedule.
 - Its time and memory costs lie between the two.

Non-Convex Loss Functions

Gradient Descent is a generic method: you can use it to find the minima of other loss functions.

- Not all loss functions are convex, which can cause problems for Gradient Descent:
- The algorithm might converge to a local minimum, instead of the global minimum.
- It may take a long time to cross a plateau.



What do we do about this?

Non-Convex Loss Functions

What do we do about this?

- One thing is to prefer Stochastic Gradient Descent (or Mini-Batch Gradient Descent): because of the way they 'bounce around', they might even escape a local minimum, and might even get to the global minimum.
- In this context, simulated annealing is also useful: updates start out 'large' allowing these algorithms to make progress and even escape local minima; but, over time, updates get smaller, allowing these algorithms to settle at or near the global minimum.
- But, if using simulated annealing, if you reduce the learning rate too quickly, you may still get stuck in a local minimum.

Next lecture

Linear Regression

14th September 2023

IT496: Introduction to Data Mining



Lecture 14

Linear Regression

[Univariate and Multivariate Regression]

Arpit Rana
14th September 2023

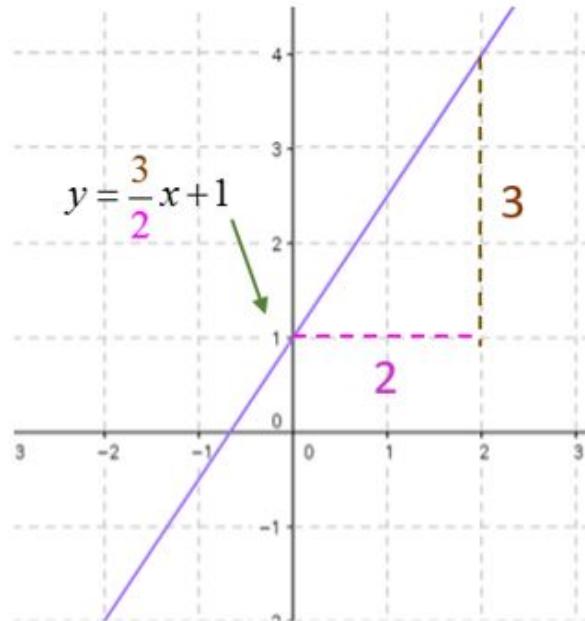
Linear Equations

We know the equation of a straight line:

$$y = mx + c$$

Gradient of the line (can be positive/ negative)

Intercept of the line (can be positive/ negative)



Linear Equations and Vectors

In general:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$$

- β_0, \dots, β_n are numbers, called the coefficients;
- x_1, \dots, x_n are the variables;
- Each of the things being added together is called a term.

Given a linear equation and the values of the variables (x_1, \dots, x_n), we can evaluate the equation, i.e. work out the value of y .

Linear Equations and Vectors

Given a general equation:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$$

- We can gather the variables into a row vector $[x_1 \ x_2 \ \dots \ x_n]$
- We can gather the coefficients (except β_0) into a column vector $\begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$ (of the same dimension, n)
- From an equation $y = 12 + 3x_1 + 4x_2 + 5x_3$ we get $x = [x_1 \ x_2 \ x_3]$ and $\beta = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}$

Linear Equations and Vectors

Given a general equation

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$$

- Hence, the above equation can be written as

$$y = \beta_0 + \sum_{i=1}^n x_i \beta_i$$

- It can also, equivalently, be written in this form

$$y = \beta_0 + x\beta$$

Hence, to evaluate a linear equation, simply multiply the two vectors and add β_0

Linear Equations and Vectors

Given a general equation

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$$

- Hence, the above equation can be written as

$$y = \beta_0 + \sum_{i=1}^n x_i \beta_i$$

- In a more simplified manner

$$y = \sum_{i=0}^n x_i \beta_i = x\beta \quad (\text{here, } x_0 = 1)$$

This is **vectorization form**: concise and fast code!

House Rent Prediction Dataset (Magicbricks, India)

	Posted On	BHK	Rent	Size	Floor	Area Type	Area Locality	City	Furnishing Status	Tenant Preferred	Bathroom	Point of Contact
311	2022-06-03	1	9000	450	Ground out of 3	Carpet Area	Salt Lake City Sector 5	Kolkata	Unfurnished	Bachelors/Family	1	Contact Agent
3869	2022-05-20	3	19500	1270	1 out of 2	Super Area	Madipakkam	Chennai	Semi-Furnished	Bachelors	2	Contact Owner
1368	2022-06-21	1	20000	310	Ground out of 7	Carpet Area	Malad West	Mumbai	Unfurnished	Bachelors	1	Contact Agent
1528	2022-06-13	2	16000	600	1 out of 2	Carpet Area	Girinagar	Bangalore	Unfurnished	Bachelors	2	Contact Owner
309	2022-06-25	3	13000	950	Ground out of 2	Carpet Area	Rabindrapally, Garia	Kolkata	Unfurnished	Bachelors/Family	2	Contact Owner

Dataset Glossary (Column-Wise): 4746 Records

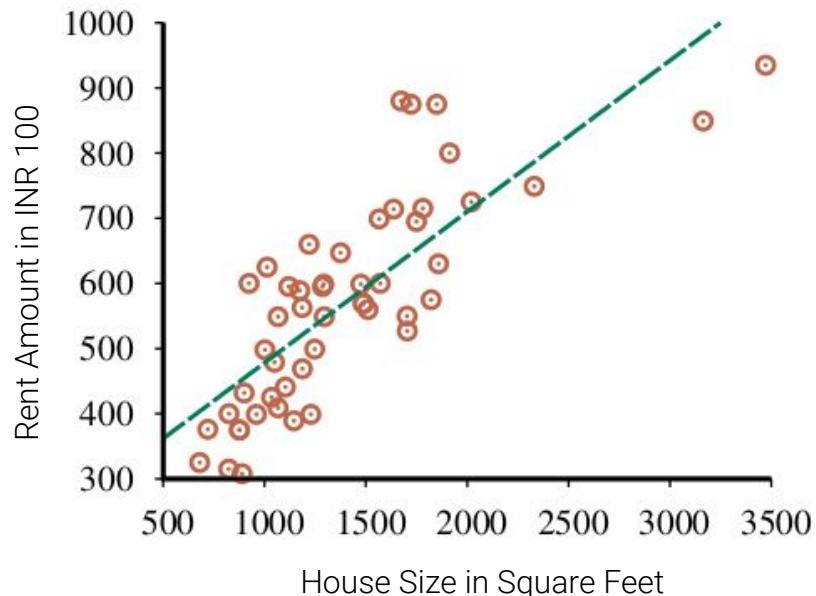
- **BHK:** Number of Bedrooms, Hall, Kitchen.
- **Floor:** Ground out of 2, 3 out of 5, etc.
- **Size:** Size of property in Square Feet.
- **Area Type:** Super Area/Carpet Area/Built Area.
- **Furnishing Status:** Furnished/Semi-Furnished/Unfurnished.
- **Bathroom:** Number of Bathrooms.
- **Area Locality:** Locality of the property
- **City:** City where the property is Located.
- **Tenant Preferred:** Family/Bachelor
- **Point of Contact:** Agent / Owner
- **Rent:** Price of the property

Univariate Linear Regression (with one variable)

The goal of our learning algorithm is to fit a linear model to this data:

$$\hat{y} = \beta_0 + \beta_1 \times \text{size}$$

In other words, our goal is to choose values for β_0 and β_1

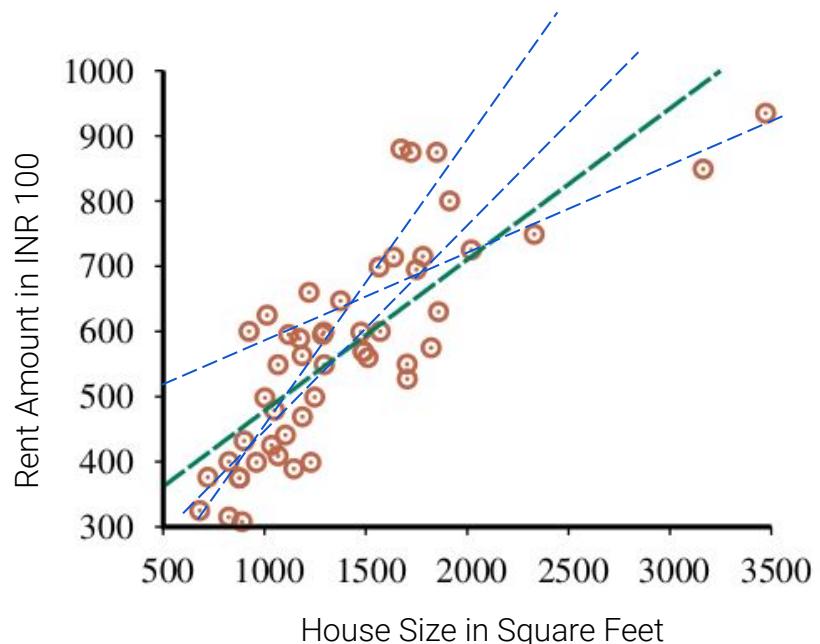


Univariate Linear Regression (with one variable)

But there is an infinite set of linear models the algorithm can choose from:

- an infinite number of straight lines it can draw;
- or, equivalently, an infinite set of values from which it can pick β_0 and β_1

We want it to choose the one that best fits the data.



Univariate Linear Regression (with one variable)

The algorithm needs a function that measures how well a model (hypothesis) fits the data.

- This is called its loss function, designated as J .
- The function takes in a particular hypothesis h_β and gives it a score.
 - Low numbers are better!
- For each x in the training set, it will compare $h_\beta(x)$, which is the prediction that h_β makes on x , with the actual value y .

The loss function most usually used for linear regression is the *mean squared error*, i.e.:

$$\begin{aligned} J(X, y, \beta) &= \frac{1}{2m} \sum_{i=1}^m \left(h_\beta(x^{(i)}) - y^{(i)} \right)^2 \\ &= \frac{1}{2} \text{mean}(X\beta - y)^2 \end{aligned}$$

This is often referred to as *ordinary least-squares* regression (OLS).

Univariate Linear Regression (with one variable)

The goal of our learning algorithm is to fit a linear model to this data:

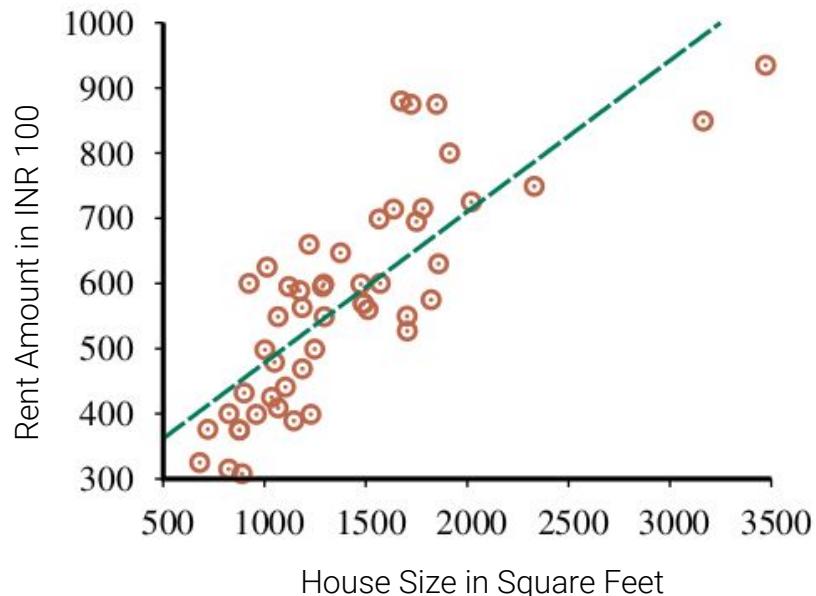
$$\hat{y} = \beta_0 + \beta_1 \times \text{size}$$

We found the solution is -

$$\beta_0 = 0.232, \text{ and } \beta_1 = 246, \text{ i.e.}$$

the line $y = 0.232x + 246$

minimizes the loss function.



Multivariate Linear Regression (with more than one variable)

We can simply generalize the idea to multiple variables:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$$

Equivalently -

$$y = \sum_{i=0}^n x_i \beta_i = x\beta$$

The only differences when we move to more than one feature:

- We can't plot so easily.
- The model is a plane when there are two features.
- The model is a hyperplane when there are more than two features.

Finding OLS Models

- We've been trying out different values for β , looking for the model with lowest mean squared error...

- ...by trial and error!

In practice, it is not done by trial-and-error.

- There are two main methods:

- The **Normal Equation** (`LinearRegression` class in scikit-learn)
 - Various forms of **Gradient Descent** (`SGDRegressor` class in scikit-learn)

The Normal Equation

- The normal equation solves for β

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

i.e., the normal equation gives us the parameters that minimize the loss function.

- Where does it come from?

- Take the gradient of the loss function: $\frac{1}{m} X^T (X\beta - y)$

- Set it to zero: $\frac{1}{m} X^T (X\beta - y) = 0$ (in fact, a $(n+1)$ -dimensional vector of zeros).

- Then do some algebraic manipulation to get β on the left-hand side, that's it.

The Normal Equation: Partial Derivatives

We need the gradient of the loss function with regards to each β_j

- In other words, how much the loss will change if we change β_j a little.
- With respect to a particular β_j , it is called the partial derivative
- The partial derivatives of $J(X, y, \beta)$ with respect to β_j , are

$$\frac{\partial J(X, y, \beta)}{\partial \beta_j} = \frac{1}{m} \sum_{i=1}^m (x^{(i)} \beta_j - y^{(i)}) \times x_j^{(i)}$$

The Normal Equation: Partial Derivatives

- The gradient vector, $\nabla_{\beta}J(X, y, \beta)$ is a vector of each partial derivative:

$$\nabla_{\beta}J(X, y, \beta) = \begin{bmatrix} \frac{\partial J(X, y, \beta)}{\partial \beta_0} \\ \frac{\partial J(X, y, \beta)}{\partial \beta_1} \\ \vdots \\ \vdots \\ \frac{\partial J(X, y, \beta)}{\partial \beta_n} \end{bmatrix} = \begin{bmatrix} \frac{1}{m} \sum_{i=1}^m x^{(i)} \beta_j - y^{(i)} \times x_0^{(i)} \\ \frac{1}{m} \sum_{i=1}^m x^{(i)} \beta_j - y^{(i)} \times x_1^{(i)} \\ \vdots \\ \vdots \\ \frac{1}{m} \sum_{i=1}^m x^{(i)} \beta_j - y^{(i)} \times x_n^{(i)} \end{bmatrix}$$

- Also, there is a vectorized way to compute it:

$$\nabla_{\beta}J(X, y, \beta) = \frac{1}{m} X^T (X\beta - y)$$

Multivariate Linear Regression (with more than one variable)

Python Implementation

The `fit` method of scikit-learn's `LinearRegression` class does what we have described:

- It inserts the extra column of 1s.
- It calculates β using the Normal Equation.

There's a problem:

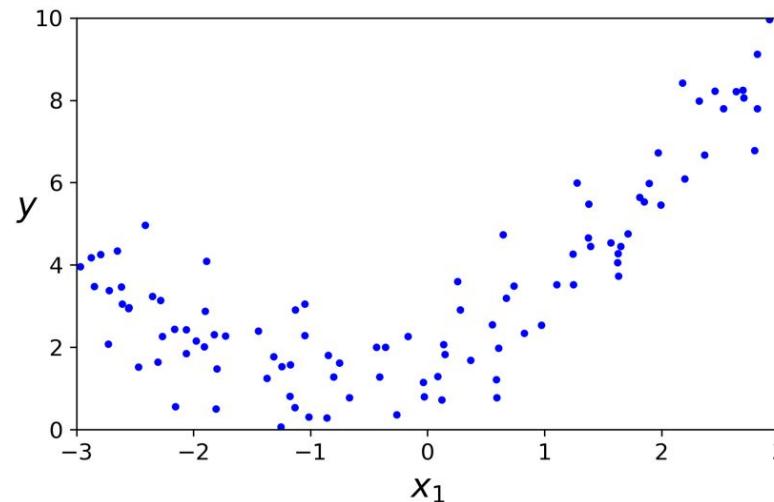
- The normal equation requires that $X^T X$ has an inverse.
- But it might not (might be a singular matrix).
- Therefore, we can use the pseudo-inverse instead. (in place of `numpy.linalg.inv()` , we can use `numpy.linalg.pinv()`).

Non-linearity

What if the true relationship between the features and the target values is non-linear?

A linear model may not be good enough:

- The test error may be too high
- Even the training error may be too high!



Polynomial Regression

What we need is a more complex model.

- Roughly, a model is more complex if it has more parameters.
 - E.g. quadratic functions for a dataset with just three features (x_1 , x_2 and x_3)

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_1^2 + \beta_5 x_2^2 + \beta_6 x_3^2 + \beta_7 x_1 x_2 + \beta_8 x_2 x_3 + \beta_9 x_3 x_1$$

Polynomial Regression

There are learning algorithms for non-linear models (including neural networks, see later).

But there's a really neat trick for using a linear model to get some of the same effect!

- We add extra features to our dataset.
 - Some of the new features will be powers of the original features,
e.g. a new feature $x_4 = x_1^2$
 - Others will be products of the original features,
e.g. a new feature $x_7 = x_1x_2$ (these are often called interaction features).
- Then learn a linear model on the new dataset.

This technique is called Polynomial Regression.

Polynomial Regression

Example:

- We want to learn models of this form:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_1^2 + \beta_5 x_2^2 + \beta_6 x_3^2 + \beta_7 x_1 x_2 + \beta_8 x_2 x_3 + \beta_9 x_3 x_1$$

- But instead we learn linear models of this form:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5 + \beta_6 x_6 + \beta_7 x_7 + \beta_8 x_8 + \beta_9 x_9$$

- but where

$$x_4 = x_1^2$$

$$x_5 = x_2^2$$

$$x_6 = x_3^2$$

$$x_7 = x_1 x_2$$

$$x_8 = x_2 x_3$$

$$x_9 = x_3 x_1$$

Polynomial Regression

Polynomial Regression in scikit-learn

- There's a class called `PolynomialFeatures`
- We use it as follows

```
>>> from sklearn.preprocessing import PolynomialFeatures  
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)  
>>> X_poly = poly_features.fit_transform(X)
```

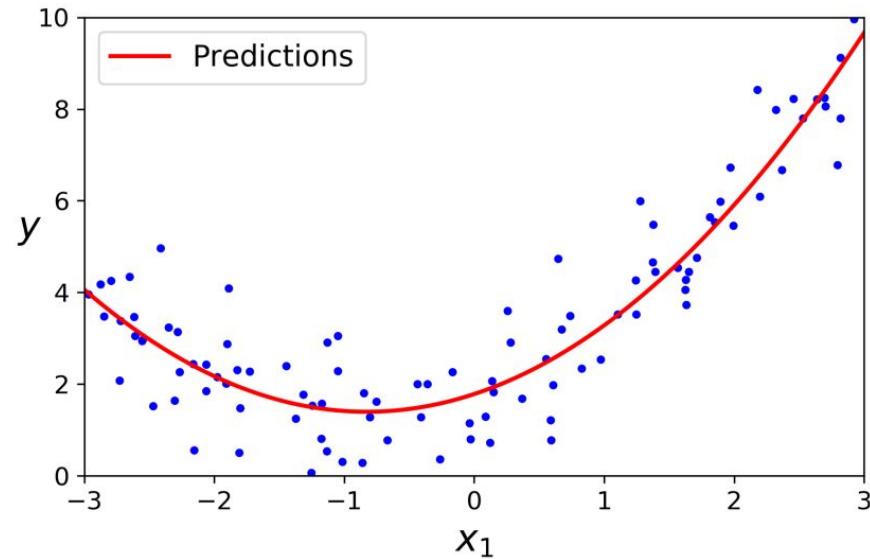
- `PolynomialFeatures(degree=d, include_bias=False)` transforms a dataset that had n features into one that has $(n+d)!/n!d!$ features.

Polynomial Regression

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

$$\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$$

$$y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise.}$$



Next lecture

Regularization

15th September 2023



IT496: Introduction to Data Mining



Lecture 15

Variants of Regression

[Regularization]

Arpit Rana
15th September 2023

Regularized Linear Models

You are building a predictor but its performance is not good enough. What should you do?

Some of the options include:

- gather more training examples; may not help. . .
- remove noise in the training examples; may in some cases worsen the performance. . .
- add more features or remove features;
- change model: move to a more complex model or maybe to a less complex model;
- stick with your existing model but add constraints to it to reduce its complexity or remove constraints to increase its complexity.

...it all depends on what is causing the poor performance (underfitting or overfitting).

Underfitting

If your model underfits:

- gathering more training examples will not help.

Your main options are:

- change model: move to a more complex model;
- collect data for additional features that you hope will be more predictive;
- create new features which you hope will be predictive (see examples of feature engineering in the LA-01 Lab colab file);
- stick with your existing model but remove constraints (if you can) to increase its complexity.

Overfitting

If your model overfits, your main options are:

- gather more training examples;
- remove noise in the training examples;
- change model: move to a less complex model;
- simplify by reducing the number of features;
- stick with your existing model but add constraints (if you can) to reduce its complexity.

Regularization

If your model underfits, we saw that one option is:

- stick with your existing model but remove constraints (if you can) to increase its complexity.

If your model overfits, we saw that one option is:

- stick with your existing model but add constraints (if you can) to reduce its complexity.

Constraining a model to make it less complex and reduce the risk of overfitting is called regularization.

Regularization is a general concept but we will explain it in the case of linear regression in the rest of this lecture.

Regularization for Linear Regression

Linear models are among the least complex models.

- Hence, we normally associate them with underfitting.

But, even linear regression (multivariate) might overfit the training data. If you are overfitting, you must reduce the degrees of freedom.

- One way is to discard some features.
 - Then you have fewer coefficients (β) that you can modify.
- Another way is to constrain the range of values that the coefficients can take:
 - E.g. force the learning algorithm to only choose small values (close to zero). This makes the distribution of the values of the coefficients more regular.

Regularization for Linear Regression

Recall that OLS linear regression finds coefficients that minimize

$$\begin{aligned} J(X, y, \beta) &= \frac{1}{2m} \sum_{i=1}^m \left(h_{\beta}(x^{(i)}) - y^{(i)} \right)^2 \\ &= \frac{1}{2} \text{mean}(X\beta - y)^2 \end{aligned}$$

Regularization imposes a penalty on the size of the coefficients.

This is how we regularize linear regression.

- In effect, it penalizes hypotheses that fit the data too well.

Lasso Regression: Using the L₁-norm

'Lasso' stands for 'least absolute shrinkage and selection operator' – but this doesn't matter!

- We penalize by the L₁-norm of β , which is simply the sum of their absolute values, i.e.

$$\sum_{j=1}^n |\beta_j|$$

Minor point: we don't penalize β_0 , which is why j starts at 1.

So Lasso Regression finds the β that minimizes

$$J(X, y, \beta) = \frac{1}{2m} \sum_{i=1}^m \left(h_\beta(x^{(i)}) - y^{(i)} \right)^2 + \lambda \sum_{j=1}^n |\beta_j|$$

λ is called the 'regularization parameter'.

Lasso Regression: Using the L₁-norm

Regularization Parameter (λ , in scikit-learn it is `alpha`)

It controls how much penalization we want and this determines the balance between the two parts of the modified loss function: *fitting the data versus shrinking the parameters*.

- As $\lambda \rightarrow 0$, Lasso Regression gets closer to being OLS Linear Regression.
- When $\lambda = 0$, Lasso Regression is the same as OLS Linear Regression.
- When $\lambda \rightarrow \infty$, penalties are so great that all the coefficients will tend to zero: the only way to minimize the loss function will be to make the coefficients as small as possible. It's likely that in this case we will underfit the data.

So, for regularization to work well, we must choose the value of λ carefully.

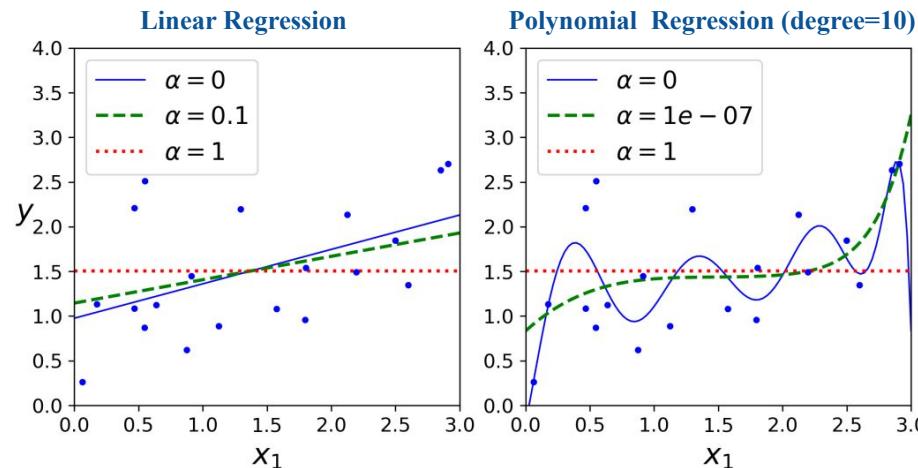
So what kind of thing is λ ?

Lasso Regression: Using the L_1 -norm

Regularization Parameter (λ)

An important observation about Lasso Regression:

- As λ grows, some of the β will be driven to zero.
- This means that the model that it learns treats some features as irrelevant.
- Hence, it performs some feature selection too.



Lasso Regression: Using the L₁-norm

Python Implementation

- scikit-learn has a class for this, called `Lasso`.
- Or you can use `SGDRegressor` with `penalty="l1"`.
- They both refer λ to as `alpha`!
- Scaling of feature values is usually advised.
- In case of polynomial regression, the data is first expanded using `PolynomialFeatures(degree=3)`, then it is scaled using `StandardScaler`, and finally the regularization is applied.

```
>>> from sklearn.linear_model import Lasso  
>>> lasso_reg = Lasso(alpha=0.1)  
>>> lasso_reg.fit(X, y)  
>>> lasso_reg.predict([[1.5]])  
array([1.53788174])
```

We could instead use
`SGDRegressor(penalty="l1")`

Ridge Regression: Using the L₂-norm

We penalize by the L₂-norm of β , which is simply the sum of the squares of the coefficients, i.e.

$$\sum_{j=1}^n \beta_j^2$$

Minor point: we don't penalize β_0 , which is why j starts at 1.

Note that the L₂-norm is the square root of the sum of squares.

So Ridge Regression finds the β that minimizes

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

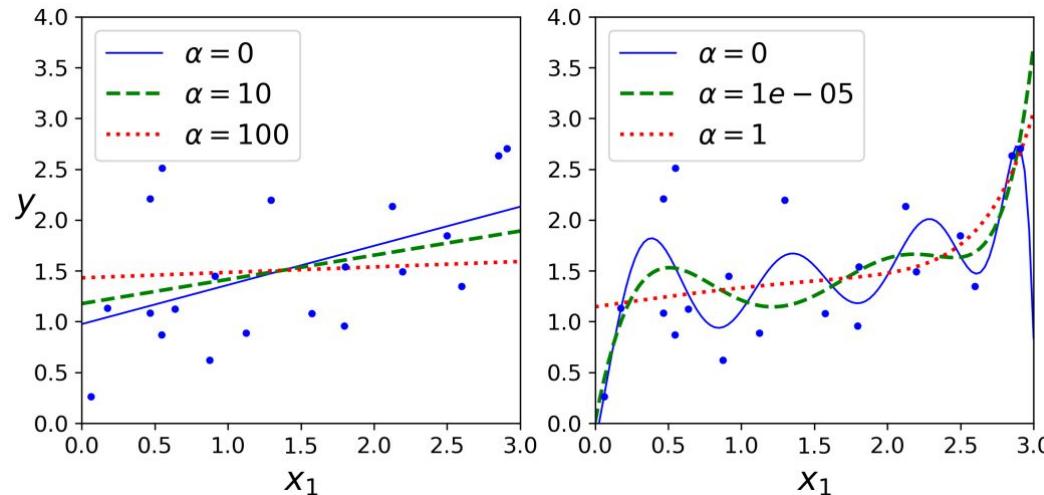
λ is called the 'regularization parameter'.

Ridge Regression: Using the L₂-norm

We penalize by the L₂-norm of β , which is simply the sum of the squares of the coefficients, i.e.

$$\sum_{j=1}^n \beta_j^2$$

Minor point: we don't penalize β_0 , which is why j starts at 1.



Ridge Regression: Using the L₂-norm

Implementing Ridge Regression

There is an equivalent to the Normal Equation (solved, e.g., by Cholesky decomposition).

- Take the gradient, set it equal to zero, and solve for β (details unimportant)

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

- In the above equation, we chose not to penalize β_0 , we want a zero in the top left, so I is not really the identity matrix.
- Also, you don't need to implement this with the pseudo-inverse. It's possible to prove that, provided $\lambda > 0$, then $(X^T X + \lambda I)$ will be invertible.

Ridge Regression: Using the L₂-norm

Implementing Ridge Regression

Alternatively, use Gradient Descent:

- The update rule for β_j for all j except $j=0$ becomes:

$$\begin{aligned}\beta_j &= \beta_j - \alpha \left(\frac{1}{m} \sum_{i=1}^m (h_\beta(x^{(i)}) - y^{(i)}) \times x_j^{(i)} + \frac{\lambda}{m} \beta_j \right) \\ &= \beta_j \left(1 - \alpha \frac{\lambda}{m} \right) - \alpha \frac{1}{m} \sum_{i=1}^m (h_\beta(x^{(i)}) - y^{(i)}) \times x_j^{(i)}\end{aligned}$$

This helps to show why this shrinks β_j .

Ridge Regression: Using the L₂-norm

Python Implementation

- In scikit-learn, there is a special class for this, called `Ridge`.
- You can set its `solver` parameter to choose different methods, or leave it as default `auto`.
- Or you can use `SGDRegressor` with `penalty="l2"`.
- Scaling of feature values is usually advised.

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([1.55071465])
```

Alternatively, using `SGDRegressor`

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

Ridge Regression: Using the L₂-norm

Both Lasso and Ridge Regression shrink the values of the coefficients.

- But, as we mentioned, Lasso Regression may additionally result in coefficients being set to zero.
- This does not happen with Ridge Regression.

Roughly speaking, Lasso Regression shrinks the coefficients by approximately the same constant amount (unless they are so small that they get shrunk to zero),

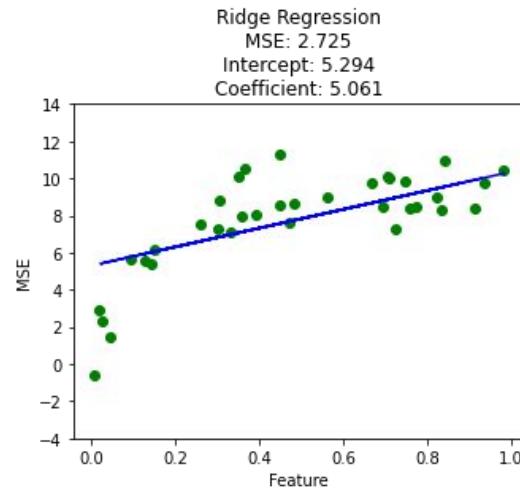
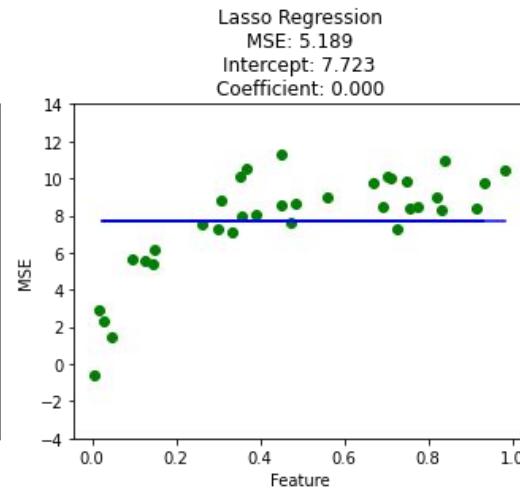
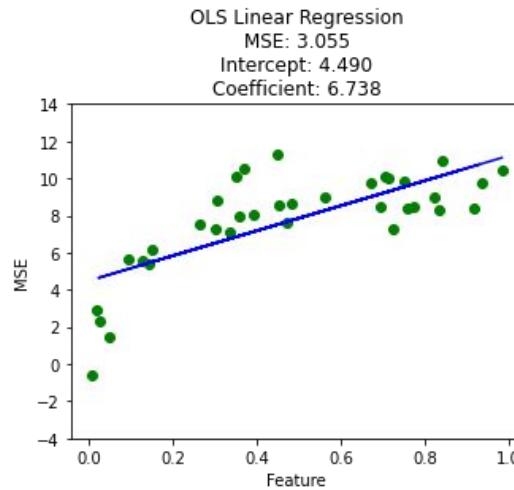
whereas,

Ridge Regression shrinks the coefficients by approximately the same proportion.

Lasso and Ridge Regression

Illustrating the Effects of Lasso and Ridge Regression

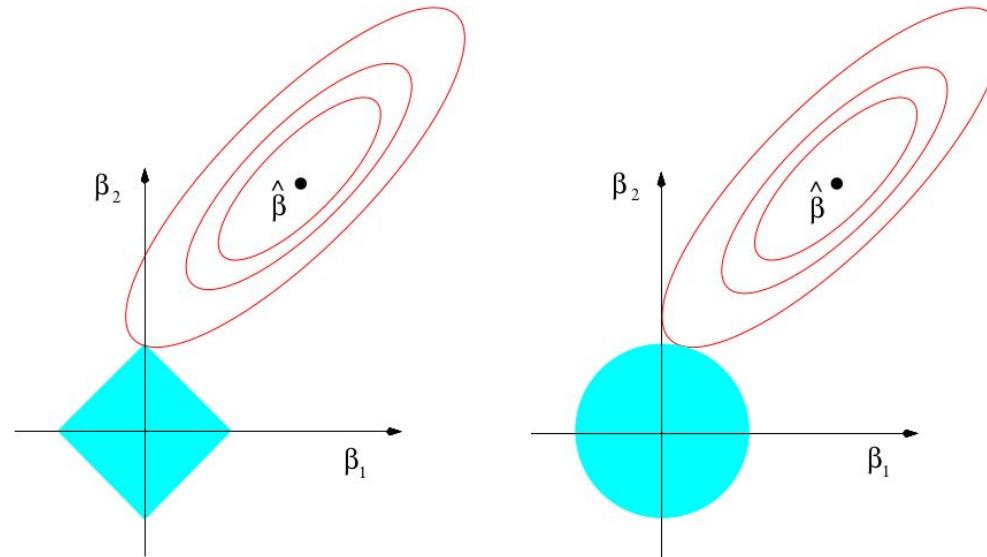
- We generate a random, non-linear dataset.
- Then we fit an unregularized linear model and two regularized models (Lasso and Ridge).



Lasso and Ridge Regression

Illustrating the Effects of Lasso and Ridge Regression

Shown are contours of the error and constraint functions. The solid areas are the constraint regions $|\beta_1| + |\beta_2| \leq t$ and $\beta_1^2 + \beta_2^2 \leq t^2$, respectively, while the red ellipses are the contours of the least squares error function.

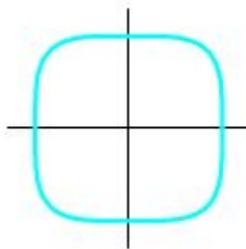


Lasso and Ridge Regression

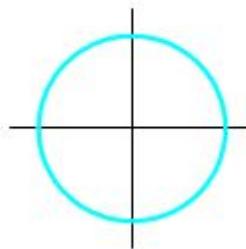
Generalizing Regularization

$$J(X, y, \beta) = \frac{1}{2m} \sum_{i=1}^m \left(h_{\beta}(x^{(i)}) - y^{(i)} \right)^2 + \lambda \sum_{j=1}^n |\beta_j|^q$$

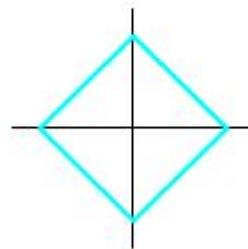
$q = 4$



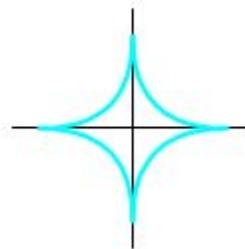
$q = 2$



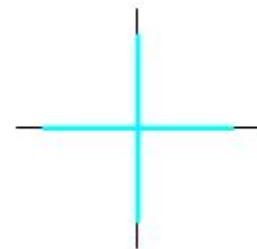
$q = 1$



$q = 0.5$



$q = 0.1$



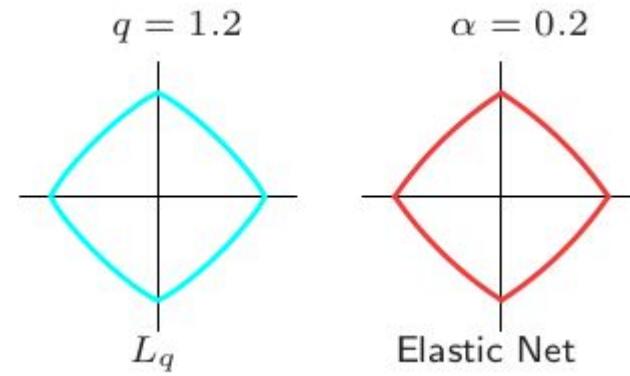
Elastic Net Regression

Combining Lasso and Ridge Regression

For completeness, we mention Elastic Net, which combines Lasso and Ridge regularization, with yet another hyperparameter to control the balance between the two.

$$\lambda \sum_{j=1}^n (\alpha \beta_j^2 + (1 - \alpha) |\beta_j|)$$

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.54333232])
```



Next lecture

Logistic Regression

19th September 2023

IT496: Introduction to Data Mining



Lecture 16

Logistic Regression

Arpit Rana
21st September 2023

Classification

We want to create programs that make predictions.

- We have been studying *regression*: regressors are programs that predict numeric target values.
- We turn now to *classification*: classifiers predict an object's class from a finite set of classes.

Example

Given a vector of feature values that describe an email, predict whether the email is *spam* or *ham*.

Notation

Our notation will be the same that we used for regression:

- \mathbf{x} for an object, \mathbf{y} for the actual class label, $\hat{\mathbf{y}}$ for the predicted class label.
- We assume we have a finite set of labels, \mathbf{C} , one per class.
 - Given an object \mathbf{x} , our task is to assign one of the labels $\hat{\mathbf{y}} \in \mathbf{C}$ to the object.
- We will often use integers for the labels.
 - E.g. given an email, a spam filter predicts $\hat{\mathbf{y}} \in \{0, 1\}$, where 0 means *ham* and 1 means *spam*.
 - But a classifier should not treat these as continuous, e.g. it should never output 0.5.

Notation

Furthermore, where there are more than two labels, we should not assume a relationship between the labels.

- Suppose there are three classes $\{1, 2, 3\}$.
- Suppose we are classifying object \mathbf{x} and we happen to know that its actual class label is $y=3$.
 - One classifier predicts $\hat{y}=1$.
 - Another classifier predicts $\hat{y}=2$.
- Which classifier has done better?

A Variation of Classification

Given an object \mathbf{x} , a classifier outputs a label, $\hat{\mathbf{y}} \in \mathbf{C}$.

- Instead, a classifier could output a probability distribution over the labels \mathbf{C} .
- For Example,
 - Given an email \mathbf{x} , a spam filter might output $\langle 0.2, 0.8 \rangle$ meaning $P(y = \text{ham} | x) = 0.2$ and $P(y = \text{spam} | x) = 0.8$
 - The probabilities must sum to 1.
- We can convert such a classifier into a more traditional one by taking the probability distribution and selecting the class with the highest probability:

$$\arg \max_{\hat{y} \in C} P(\hat{y} | x)$$

Types of Classification

We distinguish three types of classification:

- **Binary classification**, in which there are just two classes, i.e. $|C|=2$,
e.g. fail/pass, ham/spam, benign/malignant.
- **Multiclass classification**, where there are more than two classes, i.e. $|C|>2$,
e.g. let's say that a post to a forum or discussion board can be *a question, an answer, a clarification or an irrelevance*.
- **Multilabel classification**, where the classifier can assign \mathbf{x} to more than one class. I.e. it outputs a set of labels, $\hat{\mathbf{y}} \subseteq \mathbf{C}$.
 - E.g. consider a movie classifier where the classes are genres, e.g. $\mathbf{C} = \{\text{comedy}, \text{action}, \text{horror}, \text{documentary}, \text{romance}, \text{musical}\}$.
 - The classifier's output for *The Blues Brothers* should be $\{\text{comedy}, \text{action}, \text{musical}\}$.

Do not confuse this with multiclass classification.

Types of Classification

In fact, there are even more types of classification, but we will not be studying them further:

- **Ordered classification**, there is an ordering defined on the classes.
 - The ordering matters in measuring the performance of the classifier.
- E.g. consider a classifier that predicts a student's overall grade, i.e. {A, B, C, D}.
 - Suppose for student x , the actual class $y=A$.
 - One classifier predicts $\hat{y}=B$.
 - Another classifier predicts $\hat{y}=C$.
 - Which classifier has done better?

Binary Classification

In binary classification, there are two classes.

- It is common to refer to one class (the one labelled 0) as the negative class and the other (the one labelled 1) as the positive class.
- It doesn't really matter which is which.
 - But, usually, we treat the class we are trying to identify, or the class that requires special action, as the positive class.
 - E.g. in spam filtering, ham is the negative class; spam is the positive class.
 - What about *tumour* classification?
- This terminology is extended to other things too, e.g. we can refer to negative examples and positive examples.

Class Exercise

Consider:

- Predicting tomorrow's rainfall.
- Predicting whether Ireland will have a white Christmas.
- Predicting the sentiment of a tweet (negative, neutral or positive).
- Predicting a person's opinion of a movie on a rating scale of 1 star (rotten) to 5 stars (fab).

Answer the following:

- Which are regression and which classification?
- If classification, which are binary and which are multiclass?
- If binary, which is the positive class and which the negative?

Logistic Regression

The Model

Logistic Regression

We can use *model-based learning* for classification.

- There are all sorts of model, but the simplest again is a linear model.
- For regression, we wanted to find the line/plane/hyperplane that best fits the training examples.
- For classification, we want to find the line/plane/hyperplane that best separates training examples of different classes.

For two features, if it is possible to find a line that separates the data (only positive examples on one side, only negative examples on the other), we say the dataset is linearly separable.

This generalizes from straight lines to planes and hyperplanes in the case of more features.

Logistic Regression

Despite its name, logistic regression is used for classification.

- At heart, it predicts a number (and turns it into a probability), and perhaps this is why its name mentions regression.
- At heart, it builds linear models.

Logistic Regression for Binary Classification

Let's start with logistic regression for binary classification.

- In this case, logistic regression predicts the probability that \mathbf{x} belongs to the positive class.
- This is what logistic regression does:

$$\hat{y} = \begin{cases} 0 & \text{if } P(\hat{y} = 1 | x) < 0.5 \\ 1 & \text{if } P(\hat{y} = 1 | x) \geq 0.5 \end{cases}$$

Thresholding: so it outputs one of the two class labels

Where, $P(\hat{y} = 1 | x) = \sigma(x\beta)$

Where, $\sigma(z) = \frac{1}{1 + e^{-z}}$

and $\mathbf{x}\beta$ is familiar from linear regression (and assumes that \mathbf{x} has an extra 'feature', $\mathbf{x}_0 = 1$)

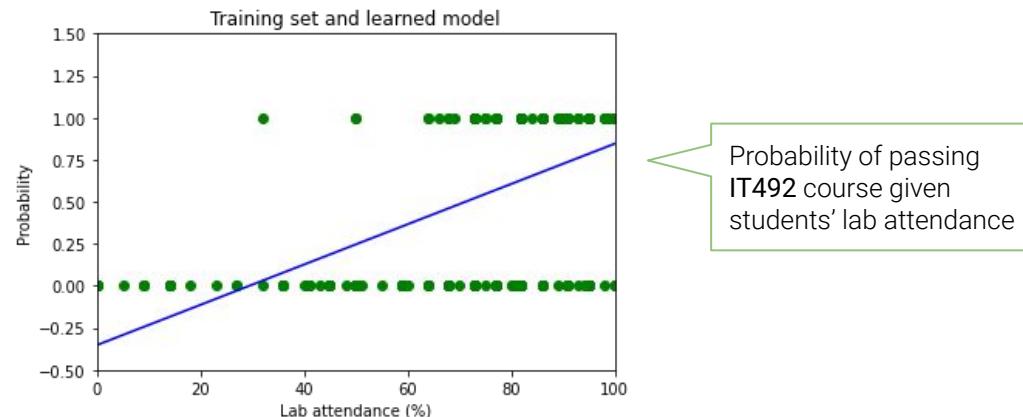
Why Not Just Linear Regression

In Linear Regression, each hypothesis h_β was of the form

$$\begin{aligned} h_\beta(x) &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n \\ &= \mathbf{x}\beta \end{aligned}$$

where \mathbf{x} is a row vector with $(n+1)$ elements (with $x_0 = 1$), and β is a (column) vector of coefficients.

Why can't we just use this directly to predict probabilities?



Why Not Just Linear Regression

The Logistic Function

To 'squash' the values of $\mathbf{x}\beta$ to $[0, 1]$ so we can treat them as probabilities

$$h_{\beta}(x) = \sigma(x\beta)$$

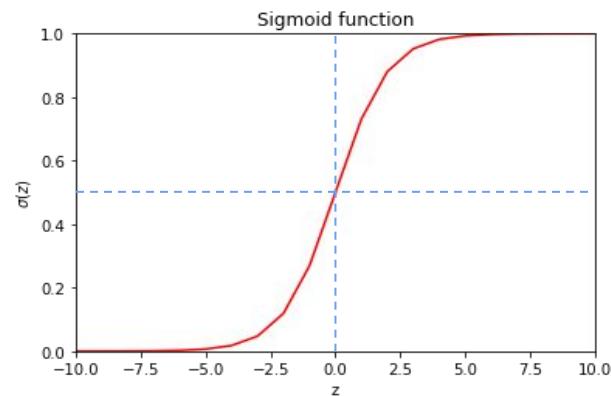
where σ is the logistic function
(also called the 'logit'):

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The logistic function is also called the sigmoid function
(which is what we will call it) because it is S-shaped:

A minor point: the sigmoid function asymptotically approaches 0 and 1.

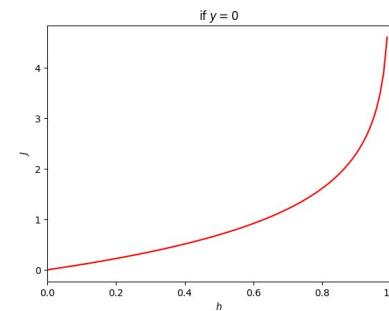
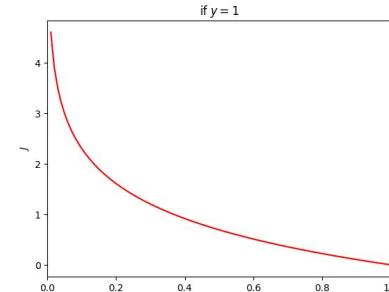
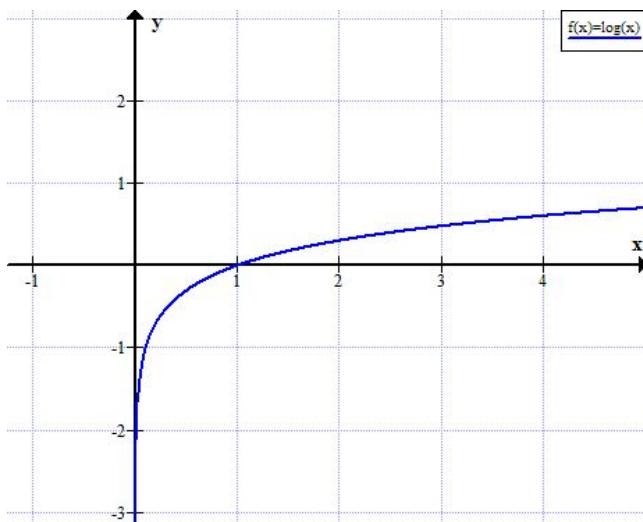
- So, in fact its values are in $(0, 1)$ and not $[0, 1]$.



Logistic Regression: Loss Function

For a given example \mathbf{x} , we propose the binary cross-entropy loss as the loss function -

$$J(x, y, \beta) = -(y \log(h_\beta(x)) + (1 - y) \log(1 - h_\beta(x)))$$



Logistic Regression: Loss Function

The overall loss function, J , is simply the average of this over all the training examples

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

This loss function is sometimes called the log loss function.

Logistic Regression: Gradient Descent

So how do we find the hypothesis that minimizes this log loss function?

- Happily, this function is convex.
- But there is no equivalent to the Normal Equation, so we must use Gradient Descent.
- Not that it matters, but here is the partial derivative of its loss function with respect to β_j

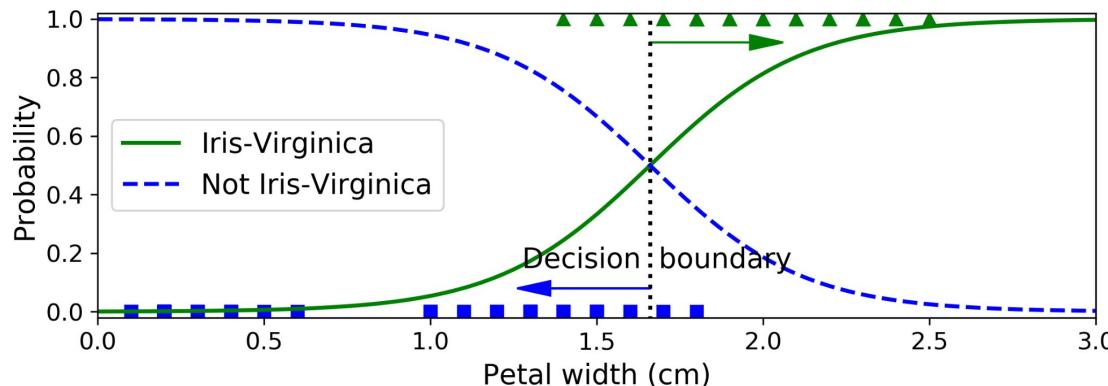
$$\frac{\partial J(X, y, \beta)}{\partial \beta_j} = \frac{1}{m} \sum_{i=1}^m (x^{(i)}\beta_j - y^{(i)}) \times x_j^{(i)}$$

Since we'll be using Gradient Descent, we must remember to scale our data.

Logistic Regression: Decision Boundary

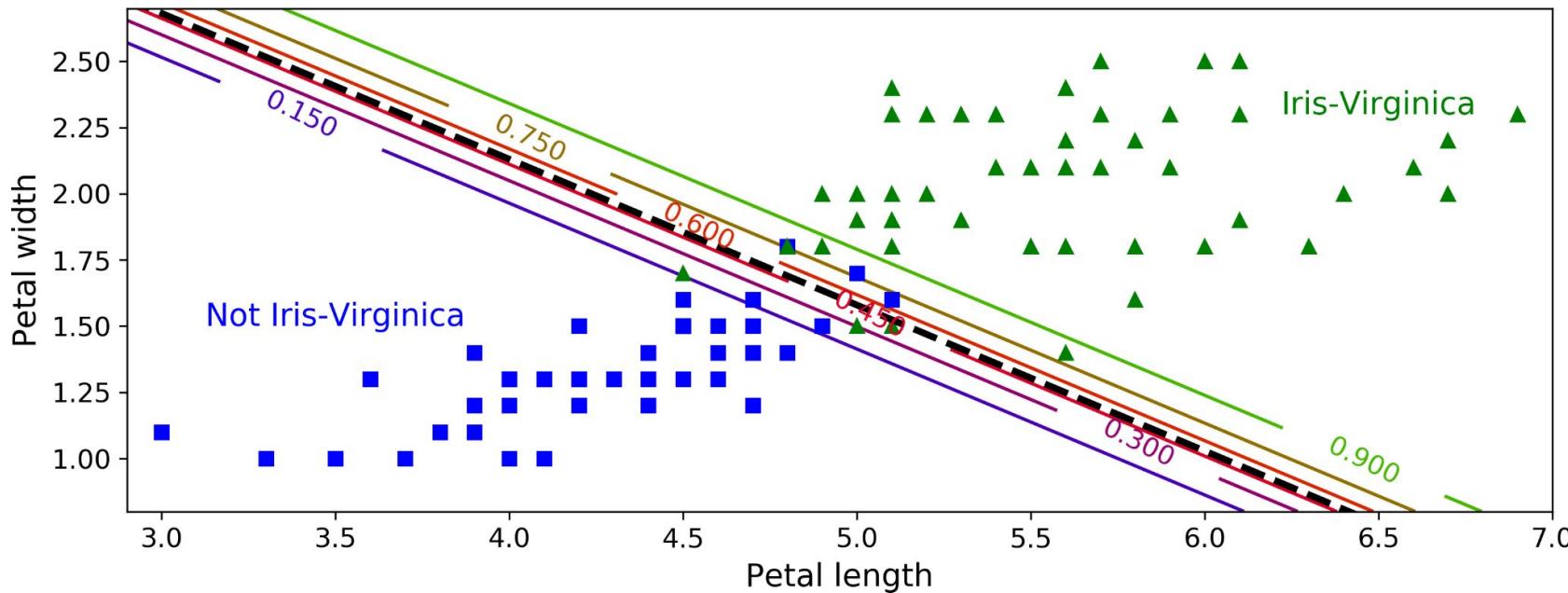
The examples for which logistic regression predicts probabilities of 0.5, $P(\hat{y} = 1 | x) = 0.5$ lie on what is called the *decision boundary*.

- If you look at the graph of the sigmoid function, its output is 0.5 when its input (z) is zero. It follows that the decision boundary are examples where $\mathbf{x}\beta = 0$.
- For example, we use the iris dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: Iris-Setosa, Iris-Versicolor, and Iris-Virginica.
- Let's try to build a classifier to detect the Iris-Virginica type based only on the petal width feature.



Logistic Regression: Decision Boundary

- Let's try to build a classifier to detect the Iris-Virginica type using the same dataset but this time displaying two features: *petal width* and *length*.



Logistic Regression in Scikit-learn

FYR:

- If you want fine-grained control over the learning rate and so on, then scikit-learn offers you the `SGDClassifier` class.
- But most people use the `LogisticRegression` class, which sits on top of the `SGDClassifier` class.
- If you want to use regularization with Logistic Regression, then there is a separate scikit-learn class for ridge classification (`RidgeClassifier` but none for lasso).
- But you can instead use `LogisticRegression`, which has an argument called `penalty`, whose possible values include "l1" and "l2". The amount of regularization is usually controlled by a hyperparameter called `alpha` in the Lasso and Ridge classes.
- For `LogisticRegression`, this hyperparameter is called `c` and `c` is the inverse of `alpha`, so small values means more regularization!

Next lecture

Multinomial Logistic Regression

22nd September 2023



IT496: Introduction to Data Mining



Lecture 17

Multinomial Logistic Regression

Arpit Rana

22nd September 2023

Multiclass Logistic Regression

Suppose there are more than two classes. Logistic Regression is one of the few classifiers that can directly handle multiclass classification.

There are three solutions:

- One-versus-Rest
- One-versus-One
- Multinomial Logistic Regression

One-vs-Rest (One-vs-All)

One-versus-Rest (also called 'one-versus-all') involves training $|C|$ binary classifiers, one per class

- for each class $c \in C$
 - create a copy of the training set in which you replace examples (\mathbf{x}, c) by $(\mathbf{x}, 1)$ and examples (\mathbf{x}, c') where $c' \neq c$ by $(\mathbf{x}, 0)$.
 - train a binary classifier h^c on this modified training set
- After all these classifiers have been trained, to classify \mathbf{x} , we run all the classifiers h^c for each $c \in C$
 - The predicted class of \mathbf{x} is the class c whose classifier h^c predicts 1 with the highest probability.

How many classifiers would we end up building for a dataset where there are three classes?

One-vs-One (Pairwise Classification)

In One-versus-One (also called 'pairwise classification'),

- we build a classifier for every pair of classes using only the training data for those two classes.
- after all these classifiers have been trained, when we want to classify \mathbf{x} , we run all the classifiers and, for each class c , we count how many of the classifiers predict that class.
 - The predicted class of \mathbf{x} is the one that is predicted most often.

One-versus-One's advantage over One-versus-Rest is that the individual classifiers do not need to be classifiers that produce probabilities.

Its disadvantage is the number of individual classifiers it must train:

- How many for a dataset that has three classes?
- In terms of $|C|$, how many in general?

Multinomial Logistic Regression

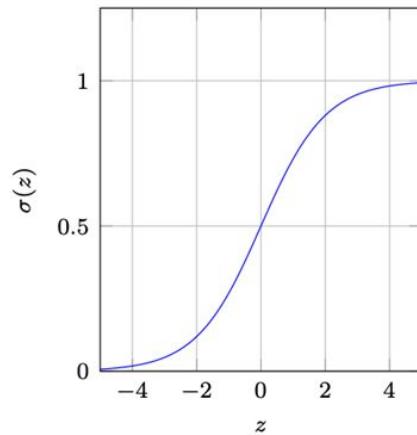
We must modify how the classifier works and the loss function for training.

- Now, instead of one vector of coefficients, β , the classifier has one per class, β_c for each $c \in \mathbf{C}$.
- Hence, instead of computing one value, $\mathbf{x}\beta$, it computes one per class, $\mathbf{x}\beta_c$ for each $c \in \mathbf{C}$.
- Now, 'squashing' is more complicated:
 - Not only must each of these values be squashed to $[0, 1]$;
 - but they must also sum to 1.
- So we do not use the *sigmoid function*.

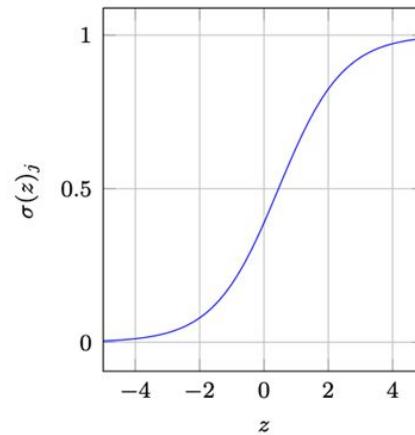
Multinomial Logistic Regression

We use the softmax function (but still often designated by σ)

$$P(\hat{y} = c \mid x) = \sigma(x\beta_c) = \frac{e^{x\beta_c}}{\sum_{c \in C} e^{x\beta_c}}$$



Sigmoid Function



Softmax Function

Multinomial Logistic Regression

We use the softmax function (but still often designated by σ)

$$P(\hat{y} = c \mid x) = \sigma(x\beta_c) = \frac{e^{x\beta_c}}{\sum_{c \in C} e^{x\beta_c}}$$

- If we put all the different β_c into a single matrix B , then we can have a vectorized implementation of this.
- Finally, there is no thresholding this time: the classifier simply predicts the class with the highest estimated probability as below.

$$\arg \max_{c \in C} (\sigma(x\beta_c))$$

Tip: To work out the probabilities, you need the softmax formula given above. But if all you want to know the winner, then all you need is $\arg \max_{c \in C} (x\beta_c)$ because the rest of the softmax formula makes no difference to the ordering.

Loss Function

So how does logistic regression learn all these different β_c ?

- We need a new loss function: the cross-entropy loss function (or sometimes the categorical cross-entropy function)

$$J(X, y, B) = -\frac{1}{m} \sum_{i=1}^m \sum_{c \in C} I(y^{(i)} = c) \log \left(\sigma(x^{(i)} \beta_c) \right)$$

where $I(p)$ is the indicator function that outputs 1 if predicate p is true and zero otherwise.

- The easiest way to get some grasp of this is to realise that when there are just two classes, it is equivalent to the loss function we used earlier.

Multiclass Logistic Regression

Which one is better?

- Sometimes, even when you have a classifier, such as Logistic Regression, that can directly handle multiclass classification, using it in a One-versus-One fashion can be more accurate! (Do you have any ideas why?)
- But one-versus-rest and one-versus-one have higher training costs.

Multiclass Logistic Regression in Scikit-learn

FYR:

- If there are more than two classes, scikit-learn will handle them without you needing to do anything.
- For most of scikit-learn's binary classifiers, it will use one-versus-rest.
- In the case of scikit-learn's `LogisticRegression` class, it will use Multinomial Logistic Regression – the method explained in the earlier slides.
- For scikit-learn's binary classifiers, if you want to use one-versus-rest, then set `multi-class="ovr"` (which is often the default, but is not the default in the case of `LogisticRegression`).
- There are also classes `OneVsRestClassifier` and `OneVsOneClassifier`.

Instance-based Learning

kNN Regressor/Classifier

Instance-based Learning

Instance-based learners learn by heart: they simply store the examples in the labeled dataset.

- The way they generalize is using *similarity* (or *distance*):
 - given an unseen example \mathbf{x} ,
 - they predict \hat{y}
 - from the y -values of examples in the dataset that are similar to \mathbf{x} .

Let's look at two concrete examples of this: *nearest-neighbour regression* and *k-nearest-neighbours regression*.

Nearest Neighbour Regression

To predict the target value \hat{y} for unseen example \mathbf{x} ,

- we find the example (\mathbf{x}', y') in the labeled dataset whose distance from \mathbf{x} is smallest; and
- we use y' as our prediction.

We also refer to this as a *1-nearest-neighbour regressor* or just 1NN or kNN for $k=1$.

Nearest Neighbour Regression

The problems with 1NN is that it can be incorrectly influenced by noisy examples:

- If there are examples in the labeled dataset where we have *incorrectly recorded the feature values*, then we may not find the best example from which to make our prediction.
- If there are examples in the labeled dataset where we have *incorrectly recorded the target value*, then these will result in incorrect predictions.

k-Nearest-Neighbours Regression

To reduce the influence of noisy examples, we use more than one neighbour:

- we find k examples whose distance from unseen example \mathbf{x} is smallest; and
- we use the mean of their y -values as our prediction.

We abbreviate the name of this to kNN, e.g. 3NN is where we use 3 nearest-neighbours.

k-Nearest-Neighbours Regression

There are many variants of kNN.

- A common one, for example, is to use a weighted average of the neighbour's target values.
- The weights could be the inverse of the distances so that more similar examples count for more.

We don't need to implement them for ourselves: scikit-learn has a class `KNeighborsRegressor`.

k-Nearest-Neighbours Classifier

We can use instance-based learning for classification.

As before, we find the k -nearest-neighbours.

- For *regression*, we took the mean of the neighbours' y-values.
- For *classification*, we take a vote: the class with the majority vote wins.
- E.g. to classify Craig, we find 3 similar students. If two of them passed, we predict Craig will pass. Otherwise, we predict Craig will fail.

Why for kNN classification, do we often choose k to be an odd number?

k-Nearest-Neighbours Classifier

There are lots of variants of this,

- e.g. we can have weighted majority vote, where the closer a neighbour is, the greater the weight of its vote.

scikit-learn has a class that implements kNN classifier: `KNeighborsClassifier`

Next lecture

Support Vector Machines

25th September 2023

IT496: Introduction to Data Mining



Lecture 18

Support Vector Machines

(Slides are created from the book Hands-on ML by Aurelien Geron)

Arpit Rana
26th September 2023

Support Vector Machines

A Support Vector Machine (SVM) is a very powerful and versatile Machine Learning model, capable of performing

- linear or nonlinear classification,
- regression, and
- even outlier detection.

SVMs are particularly well suited for classification of complex but small- or medium-sized datasets.

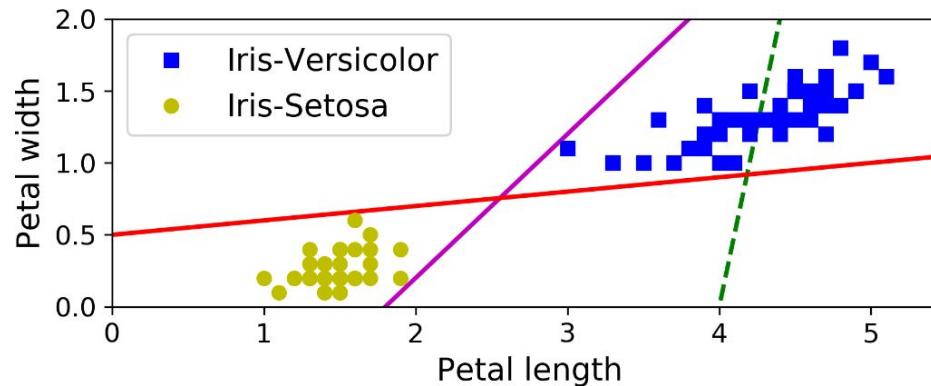
Iris Dataset

Iris flower image dataset that contains the *sepal* and *petal* length and *width* of 150 iris flowers of three different species: Iris-Setosa, Iris-Versicolor, and Iris-Virginica.



Classifying Iris Dataset

The two classes can clearly be separated easily with a straight line (they are *linearly separable*).

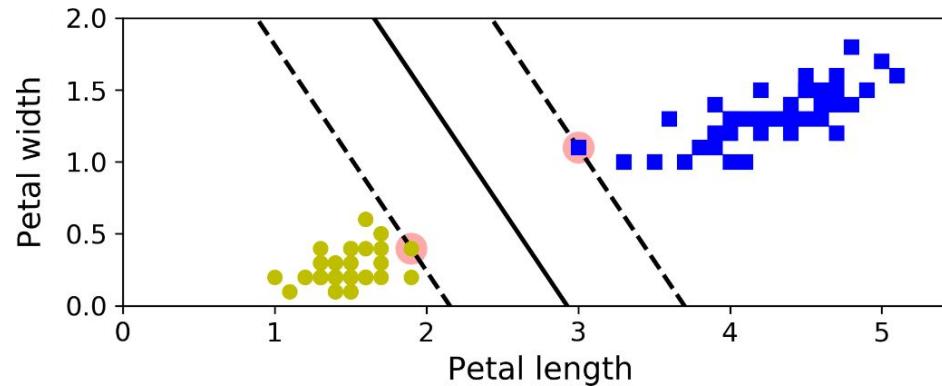


- The model whose decision boundary is represented by the green dashed line is so bad that it does not even separate the classes properly.
- The other two models work perfectly on this training set, but their decision boundaries come so close to the instances that these models will probably not perform as well on new instances.

Classifying Iris Dataset

The decision boundary of an SVM classifier not only separates the two classes but also stays as far away from the closest training instances as possible.

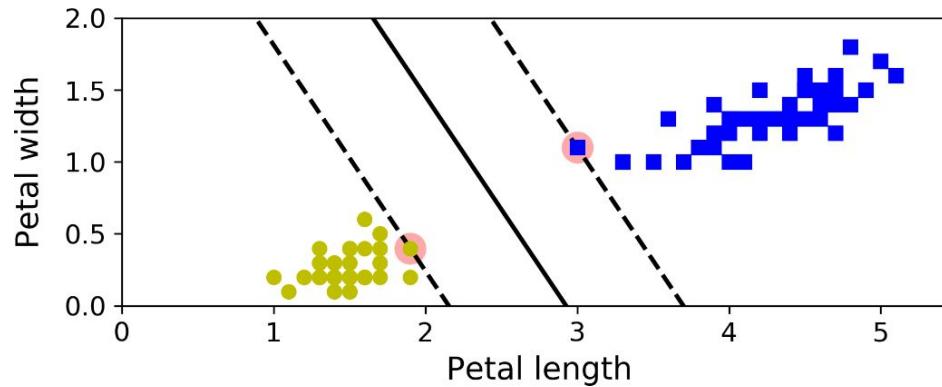
You can think of an SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes.



This is called *large (maximum) margin classification*.

Classifying Iris Dataset

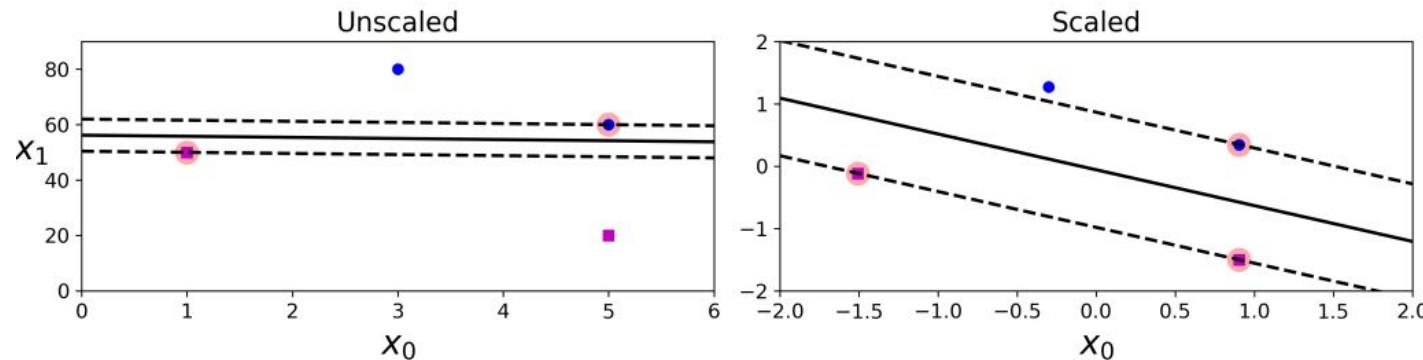
- Adding more training instances “off the street” will not affect the decision boundary at all.
- It is fully determined (or “supported”) by the instances located on the edge of the street. These instances are called the **support vectors**.



Classifying Iris Dataset

SVMs are sensitive to the feature scales (see in the figure below) -

- the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal.
- After feature scaling (e.g., using scikit-learn's `StandardScaler`), the decision boundary looks much better (on the right plot).

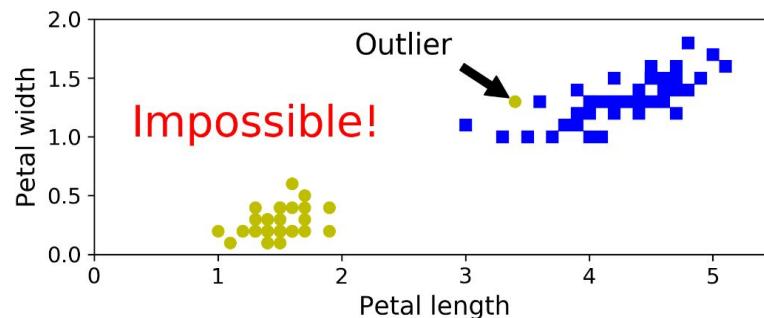


Hard Margin Classification

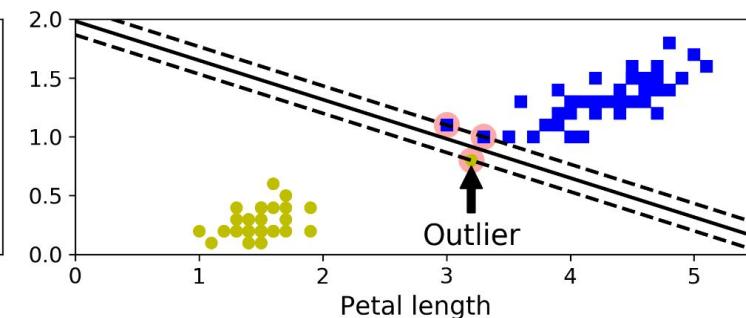
If we strictly impose that all instances be off the street and on the right side, this is called *hard margin classification*.

There are two main issues with hard margin classification.

- It only works if the data is linearly separable, and
- It is quite sensitive to outliers.



With just one additional outlier, it is impossible to find a hard margin



It will probably not generalize as well.

Soft Margin Classification

To avoid these issues it is preferable to use a more flexible model. The objective is to find a good balance between -

- keeping the street as large as possible and
- limiting the margin violations (i.e., instances that end up in the middle of the street or even on the wrong side).

This is called *soft margin classification*.

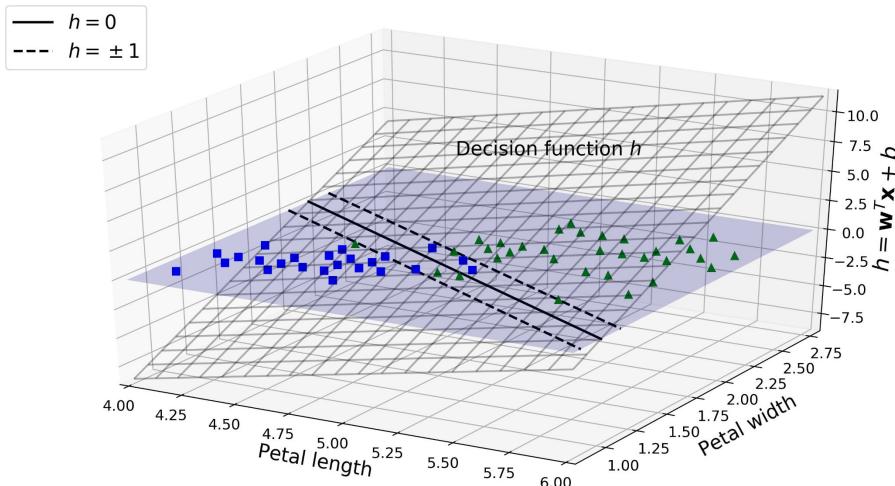
Decision Function and Prediction

The linear SVM classifier model predicts the class of a new instance x by simply computing the decision function -

$$w^T x + b = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b$$

If the result is positive, the predicted class \hat{y} is the *positive class* (1), or else it is the *negative class* (0)

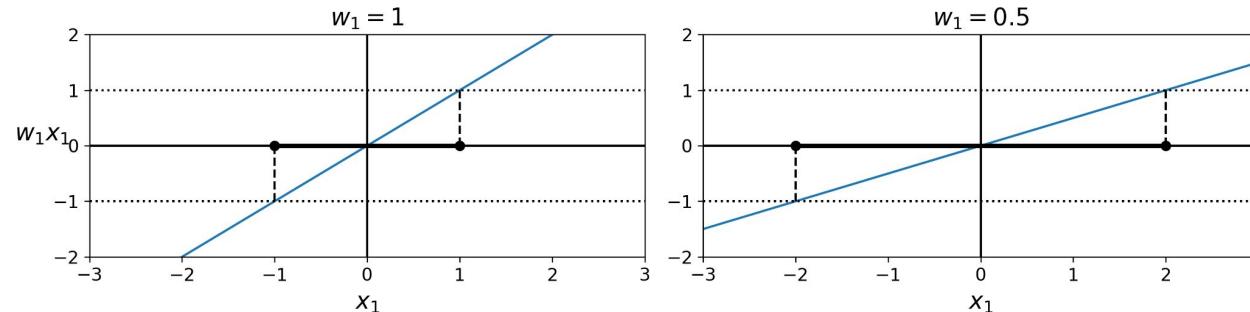
$$\hat{y} = \begin{cases} +1 & w^T x + b \geq 0, \\ -1 & w^T x + b < 0 \end{cases}$$



Training Objective

Training a linear SVM classifier means finding the value of w and b that make this margin as wide as possible while avoiding margin violations (hard margin) or limiting them (soft margin).

- Consider the slope of the decision function: it is equal to the norm of the weight vector, $\| w \|$.
- If we divide this slope by 2, the points where the decision function is equal to ± 1 are going to be twice as far away from the decision boundary.



A smaller weight vector results in a larger margin

Hard Margin SVM

So, we want to minimize $\| \mathbf{w} \|$ to get a large margin.

- If we also want to avoid any margin violation (hard margin), then we need the decision function to be greater than 1 for all positive training instances, and lower than -1 for negative training instances.
- We define $t^{(i)} = -1$ for negative instances (if $y^{(i)} = 0$) and $t^{(i)} = 1$ for positive instances (if $y^{(i)} = 1$), then we can express this constraint as $t^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1$ for all instances.

$$\begin{aligned} & \underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^T \mathbf{w} \\ & \text{subject to} \quad t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

Hard Margin Linear SVM Classifier Objective

Soft Margin SVM

To get the soft margin objective, we need to introduce a *slack variable* $\zeta^{(i)} \geq 0$ for each instance: $\zeta^{(i)}$ measures how much the i^{th} instance is allowed to violate the margin.

We now have two conflicting objectives:

- making the slack variables as small as possible to reduce the margin violations, and
- making $1/2 \mathbf{w}^T \mathbf{w}$ as small as possible to increase the margin.

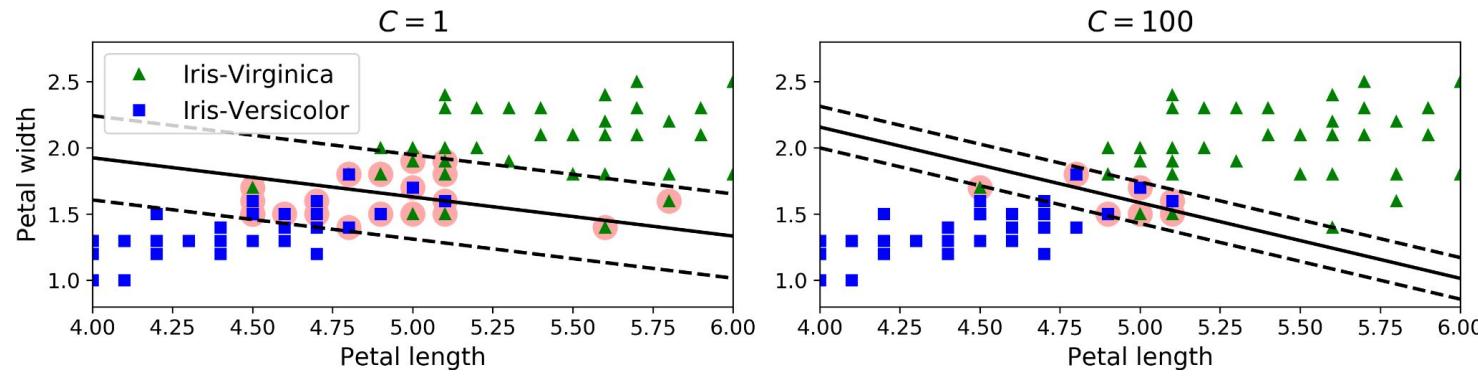
$$\begin{aligned} & \underset{\mathbf{w}, b, \zeta}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ & \text{subject to} \quad t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

Soft Margin Linear SVM Classifier Objective

Soft Margin SVM

Here, C is a hyperparameter that defines the trade-off between the two objectives

- In Scikit-learn's SVM classes, you can control this balance using the C hyperparameter: a smaller C value leads to a wider street but more margin violations.
- If the SVM model is overfitting, you can try regularizing it by reducing C.



Large margin (left) versus fewer margin violations (right)

SVM in Python

- Unlike Logistic Regression classifiers, SVM classifiers do not output probabilities for each class.

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris-Virginica

svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1, loss="hinge")),
])
svm_clf.fit(X, y)

>>> svm_clf.predict([[5.5, 1.7]])
array([1.])
```

SVM in Python

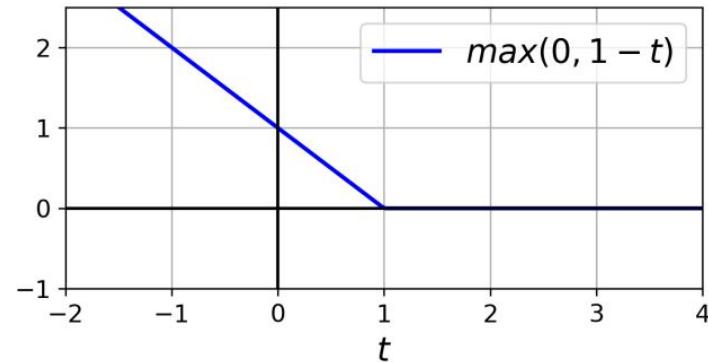
- Alternatively, we could use the SVC class, using `SVC(kernel="linear", C=1)`, but it is much slower, especially with large training sets.
- Another option is to use the SGDClassifier class, with `SGDClassifier(loss="hinge", alpha=1 / (m*C))`.
 - It does not converge as fast as the LinearSVC class, but it can be useful to handle huge datasets that do not fit in memory (out-of-core training), or to handle online classification tasks.

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
LinearSVC	$O(m \times n)$	No	Yes	No
SGDClassifier	$O(m \times n)$	Yes	Yes	No
SVC	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

Hinge Loss

The function $\max(0, 1 - t)$ is called the *hinge loss* function.

- It is equal to 0 when $t \geq 1$. Its derivative (slope) is equal to -1 if $t < 1$ and 0 if $t > 1$.
- It is not differentiable at $t = 1$, but you can still use Gradient Descent using any subderivative at $t = 1$ (i.e., any value between -1 and 0).



Online SVM

Online SVM classifiers learn incrementally, typically as new instances arrive.

- For linear SVM classifiers, one method is to use Gradient Descent (e.g., using `SGDClassifier`) to minimize the cost function -

$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \max(0, 1 - t^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b))$$

- The first sum in the cost function will push the model to have a small weight vector w , leading to a larger margin.
- The second sum computes the total of all margin violations.
- An instance's margin violation is equal to 0 if it is located off the street and on the correct side, or else it is proportional to the distance to the correct side of the street.

Next lecture

Non-linear SVM
28th September 2023

IT496: Introduction to Data Mining



Lecture 19

Non-linear Support Vector Machines

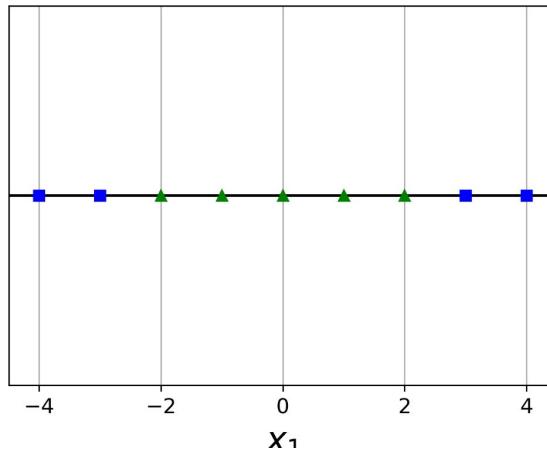
(Slides are created from the book Hands-on ML by Aurelien Geron)

Arpit Rana
3rd October 2023

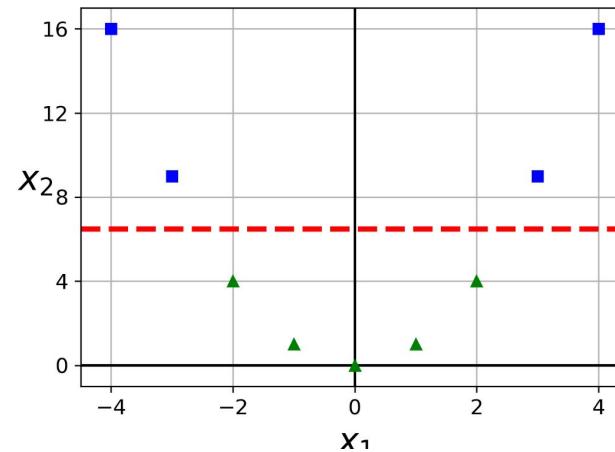
SVM with Polynomial Features

Many datasets are not even close to being linearly separable.

- One approach to handling nonlinear datasets is to add more features, such as polynomial features (as we did in polynomial regression);
- In some cases this can result in a linearly separable dataset.



Not linearly separable with one feature

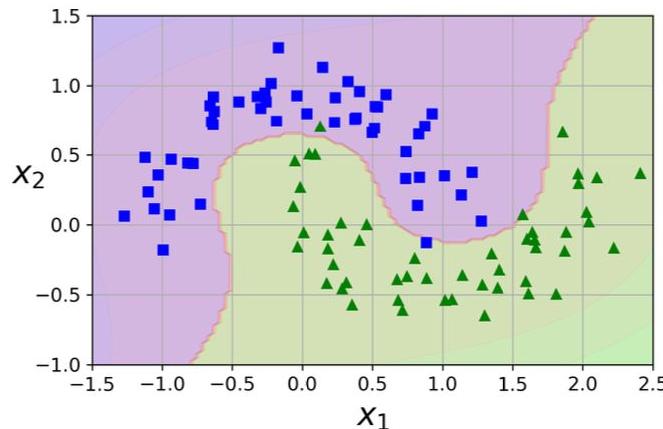


After adding second feature $x_2 = x_1^2$

SVM with Polynomial Features

Adding polynomial features is simple to implement and can work great with all sorts of Machine Learning algorithms (not just SVMs).

For example,



```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
])

polynomial_svm_clf.fit(X, y)
```

SVM with Polynomial Features

However,

- at a *low polynomial degree* it cannot deal with very complex datasets, and
- with a *high polynomial degree* it creates a huge number of features, making the model too slow.
 - **PolynomialFeatures (degree=d)** transforms a dataset that had n features into one that has $(n+d)! / n!d!$ features.

Non-linear SVM

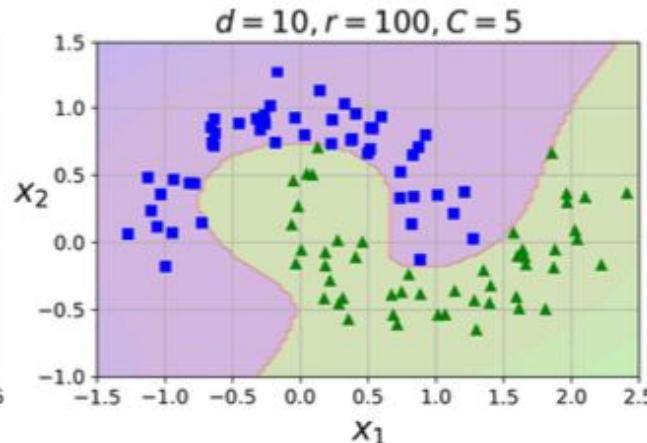
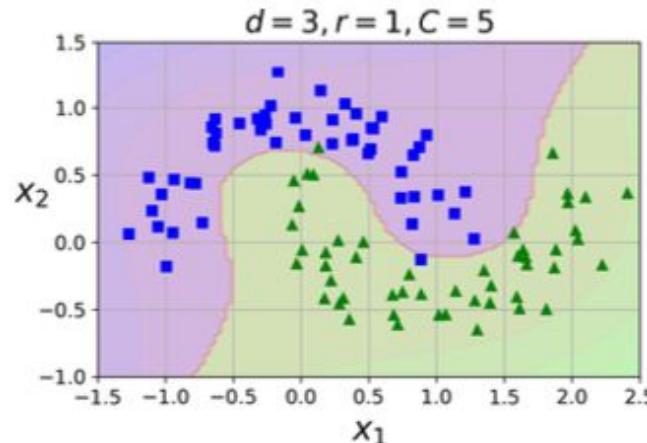
When using SVMs you can apply a mathematical technique called the *kernel trick* (it is explained in further slides).

- It makes it possible to get the same result as if you added many polynomial features, even with very high-degree polynomials, without actually having to add them.
- This trick is implemented by the SVC class.

Non-linear SVM

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
])
poly_kernel_svm_clf.fit(X, y)
```

The hyperparameter `coef0` controls how much the model is influenced by high-degree polynomials versus low-degree polynomials



Understanding the Kernel Trick

Suppose you want to apply a 2nd-degree polynomial transformation to a two-dimensional training set, then train a linear SVM classifier on the transformed training set.

The 2nd-degree polynomial mapping function ϕ that you want to apply.

$$\phi(\mathbf{x}) = \phi\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

Notice that the transformed vector is three-dimensional instead of two-dimensional.

if we apply this 2nd-degree polynomial mapping and then compute the dot product of the transformed vectors:

$$\begin{aligned}\phi(\mathbf{a})^T \phi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2\end{aligned}$$

$$= (a_1b_1 + a_2b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^T \mathbf{b})^2$$

The dot product of the transformed vectors is equal to the square of the dot product of the original vectors:

$$\phi(\mathbf{a})^T \phi(\mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$$

Understanding the Kernel Trick

So, you don't actually need to transform the training instances at all: just replace the dot product by its square.

$$\langle \phi(a), \phi(b) \rangle = \phi(a)^T \phi(b) = (a^T b)^2$$

- The result will be strictly the same as if you went through the trouble of actually transforming the training set, and then fitting a linear SVM algorithm.
- This trick makes the whole process much more computationally efficient. This is the essence of the ***kernel trick***.

Kernel Function

A **kernel** is a function capable of computing the dot product $\langle \phi(a), \phi(b) \rangle$ based only on the original vectors \mathbf{a} and \mathbf{b} , without having to compute (or even to know about) the transformation ϕ .

$$K(a, b) = \langle \phi(a), \phi(b) \rangle = f(a, b)$$

where $f(\cdot)$ is a function that follows certain conditions as stated by the Mercer's Theorem.

Some commonly used kernel functions are as follows.

Linear: $K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \mathbf{b}$

Polynomial: $K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \mathbf{b} + r)^d$

Gaussian RBF: $K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \| \mathbf{a} - \mathbf{b} \|^2)$

Sigmoid: $K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^T \mathbf{b} + r)$

Kernel Function

When to use What

- As a rule of thumb, you should always try the linear kernel first (remember that `LinearSVC` is much faster than `SVC(kernel="linear")`), especially if the training set is very large or if it has plenty of features.
- If the training set is not too large, you should try the Gaussian RBF kernel as well; it works well in most cases.
- Then, if you have spare time and computing power, you can also experiment with a few other kernels using *cross-validation* and *grid search*.

Learning Non-linear SVM

Using a suitable function, $\phi(\cdot)$, we can transform any data instance \mathbf{x} to $\phi(\mathbf{x})$.

- The linear hyperplane in the transformed space can be expressed as $\mathbf{w}^T \phi(\mathbf{x}) + b = 0$.
- To learn the optimal separating hyperplane, we can substitute $\phi(\mathbf{x})$ for \mathbf{x} in the formulation of SVM to obtain the following hard margin optimization (primal) problem:

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^T \mathbf{w}$$

$$\text{subject to} \quad t^{(i)} \left(\mathbf{w}^T \cancel{\mathbf{x}^{(i)}} + b \right) \geq 1 \quad \text{for } i = 1, 2, \dots, m$$
$$\phi(\mathbf{x}^{(i)})$$

Primal to Dual Problem

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^T \mathbf{w}$$

$$\text{subject to} \quad t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \dots, m$$

$$L = \frac{1}{2} \mathbf{w}^T \mathbf{w} - \sum_i \alpha^{(i)} \left[t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) - 1 \right]$$

w is the linear sum
of the samples x.

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_i \alpha^{(i)} t^{(i)} \mathbf{x}^{(i)} = \mathbf{0} \implies \mathbf{w} = \sum_i \alpha^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\frac{\partial L}{\partial b} = - \sum_i \alpha^{(i)} t^{(i)} = 0 \implies \sum_i \alpha^{(i)} t^{(i)} = 0$$

Replacing w in the
above expression of L

$$L = \frac{1}{2} \sum_i \sum_j \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \boxed{\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}} - \sum_i \alpha^{(i)}$$

Learning Non-linear SVM

It is possible to express a different but closely related problem, called its *dual problem*.

$$\begin{aligned} & \text{minimize}_{\alpha} \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \cancel{\mathbf{x}^{(i)T} \mathbf{x}^{(j)}} - \sum_{i=1}^m \alpha^{(i)} \\ & \text{subject to } \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

- The solution to the dual problem typically gives a lower bound to the solution of the primal problem,
- however, under some conditions it can even have the same solutions as the primal problem.
 - In the primal, the objective function is convex, and the inequality constraints are continuously differentiable and convex functions, thus, meet those conditions.

Learning Non-linear SVM

$$\begin{aligned} & \underset{\alpha}{\text{minimize}} \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \cancel{\mathbf{x}^{(i)T} \mathbf{x}^{(j)}} - \sum_{i=1}^m \alpha^{(i)} \\ & \text{subject to } \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

- Once you find the vector $\hat{\alpha}$ that minimizes this equation (using a QP solver), you can compute \hat{w} and \hat{b} that minimize the primal problem by using the following.

$$\hat{\mathbf{w}} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \hat{\mathbf{w}}^T \mathbf{x}^{(i)} \right)$$

- The dual problem is faster to solve than the primal when *the number of training instances is smaller than the number of features*.

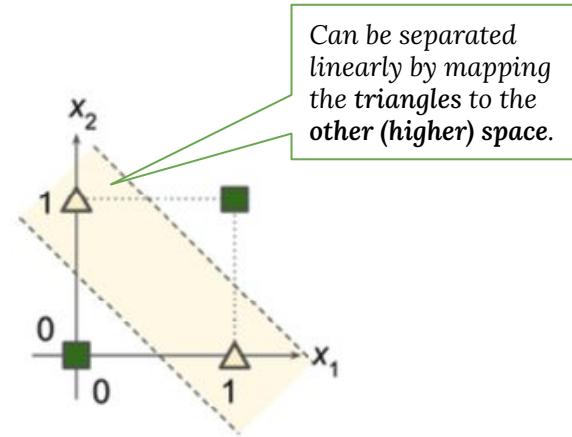
Non-linear SVM

- Finally, the decision rule will look like the following.

$$\hat{y} = \begin{cases} +1 & w^T x + b \geq 0, \\ -1 & w^T x + b < 0 \end{cases}$$

$$\phi(\mathbf{x}^{(i)})^\top \phi(\mathbf{x}^{(j)}) = K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$$

$$\sum_i \alpha^{(i)} t^{(i)} \boxed{x^{(i)} \cdot x^{(j)}} + b \geq 0$$



- Dual makes the *kernel trick* possible, while the primal does not.

Next lecture

Decision Trees

5th October 2023

IT496: Introduction to Data Mining



Lecture 19

Decision Tree Classifier

(Slides are created from the book Hands-on ML by Aurelien Geron)

Arpit Rana
5th October 2023

Decision Trees

A decision tree is a representation of a function that maps a *vector of attribute values* to a *single output value—a “decision.”*

- Decision Trees can be used for *regression* and *classification*.
 - for binary or multiclass classification (so you don't need one-versus-rest or one-versus-all)
- They are *more complex than linear models* and so can better fit complex datasets.
- Many people claim that they produce *interpretable* models.
- *Random Forests* are another popular model in Machine Learning, and they contain Decision Trees.

Decision Trees

A decision tree reaches its decision by performing a sequence of tests, starting at the root and following the appropriate branch until a leaf is reached.

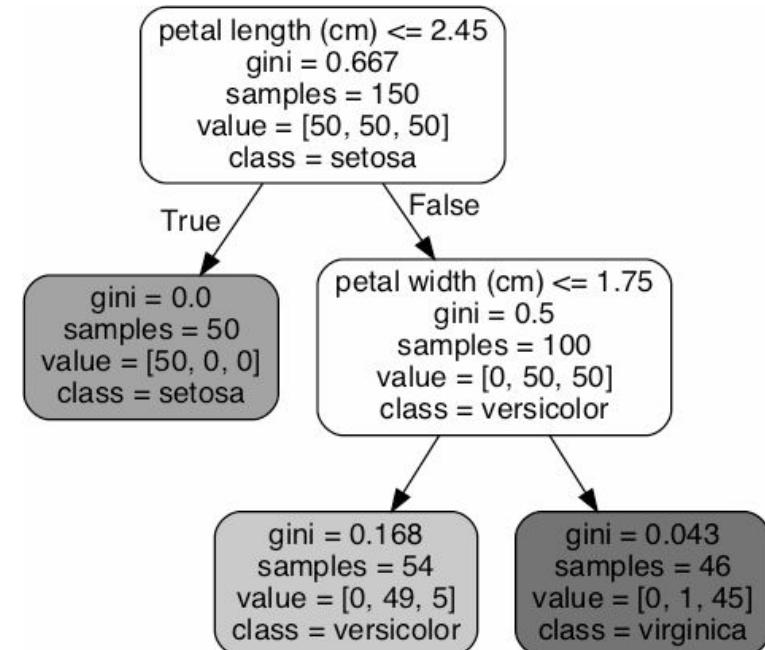
- Each *internal node* in the tree corresponds to a test of the value of one of the input attributes,
- the *branches* from the node are labeled with the possible values of the attribute, and
- the *leaf nodes* specify what value is to be returned by the function.

Decision Tree on the Iris Dataset

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

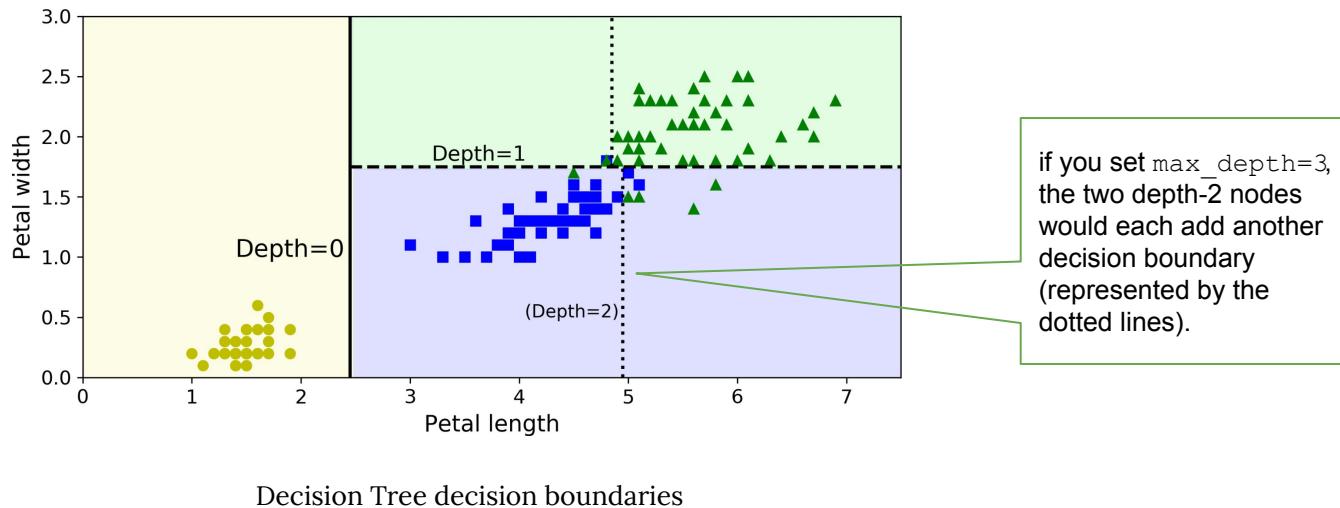
iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```



Making Predictions

- What about the flower with Petal length = 5 and Petal width = 1.5?
 - Start at the root: is the Petal length ≤ 2.45 cm?
 - No, so move right: is the Petal width ≤ 1.75 cm?
 - Yes, so move left: it is an *Iris-Versicolor*.



Reading the Nodes of the Tree

- `samples`: how many training examples the node applies to.
 - E.g. 100 training examples have a petal length > 2.45 cm.
- `value`: how many training examples of each class this node applies to.
 - E.g. [0, 1, 45] means 0 Iris-Sentosa, 1 Iris-Versicolor, and 45 Iris-Virginica.
- `gini`: is the *Gini impurity* of a node (with values in [0.0, 1.0]):
 - it measures how often an example would be incorrectly labeled if it was randomly labeled according to the distribution of labels for this node;
 - e.g. gini of 0 means no impurity: all examples that this node applies to belong to the same class;

Reading the Nodes of the Tree

The *Gini impurity/ Gini index* measures the impurity of X, a data partition or set of training tuples at a node, as

$$Gini(X) = 1 - \sum_{i=1}^n p_i^2$$

where p_i is the probability that a tuple in X belongs to the class C_i and is estimated by $|C_{i,X}|/|X|$. The sum is computed over n classes.

- For example, in the tree shown previously, the depth-2 left node has a gini score equal to $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$.

Learning a Decision Tree

- scikit-learn uses the CART Algorithm (Classification and Regression Tree).
 - It only *produces binary trees* (hence yes/no questions in non-leaf nodes);
 - it is *recursive* (repeats until it reaches some stopping criterion); and
 - it is *greedy* (It does not check whether or not the split will lead to the lowest possible impurity several levels down).
- There are other algorithms:
 - ID3, which can produce non-binary trees, and
 - C4.5, which is like CART but uses *entropy* in place of *Gini*, and
 - Others including ones that can directly handle nominal-valued features and missing values, which we will not study.

The CART Training Algorithm

- For each feature x_i and each value v , split the dataset into two:
 - X_{left} are the examples in X for which $x_i \leq v$
 - X_{right} are the examples in X for which $x_i > v$

and calculate the **CART's loss function**:
$$\frac{|X_{left}|}{|X|} Gini(X_{left}) + \frac{|X_{right}|}{|X|} Gini(X_{right})$$

- From the above, choose the feature x_i and a value v with the lowest loss
- If a stopping criterion has been reached (e.g. maximum depth or if no split reduces impurity) then:
 - return
- Else:
 - Recursively call CART on X_{left}
 - Recursively call CART on X_{right}

Regularization Hyperparameters

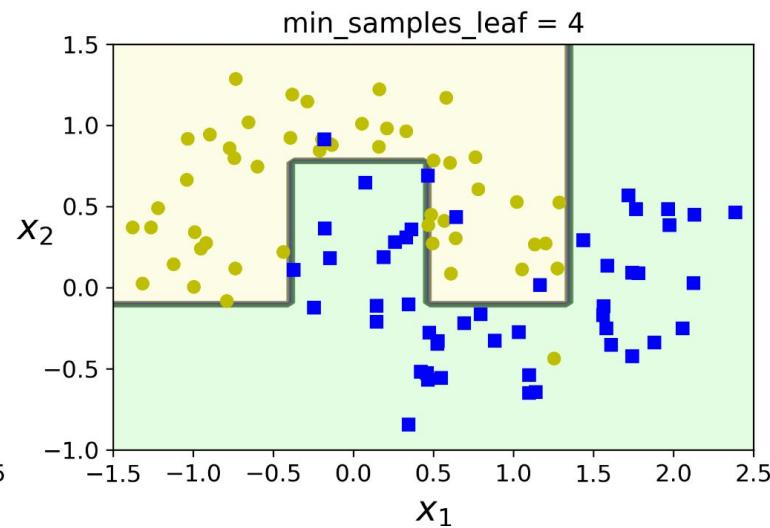
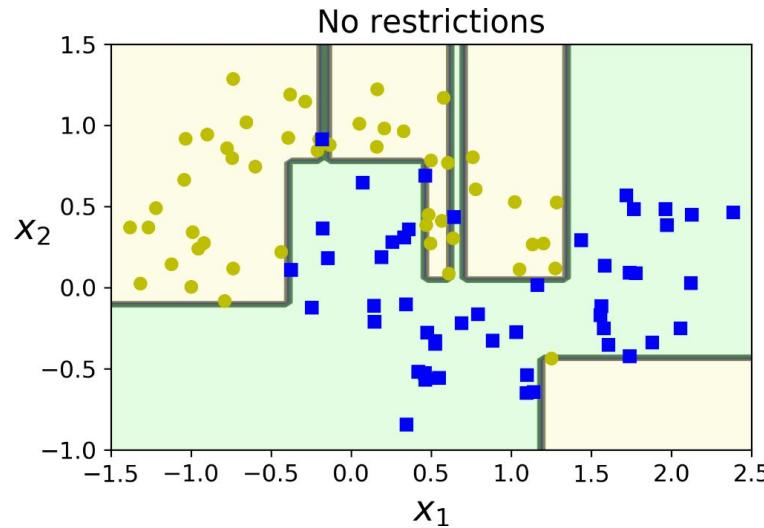
- `max_depth` is a hyperparameter.
 - Increasing `max_depth` increases model's complexity and may result in overfitting.
 - If there is no constraint on tree depth, then branches will be grown until all leaves are pure.
- There are other hyperparameters:
 - `min_samples_split`, the minimum number of samples a node must have before it can be split,
 - `min_samples_leaf`, the minimum number of samples a leaf node must have,
 - `max_leaf_nodes`, maximum number of leaf nodes, and
 - `max_features`, maximum number of features that are evaluated for splitting at each node.

Increasing `min_*` or reducing `max_*` hyperparameters will regularize the model.

Regularization Hyperparameters

Regularization using `min_samples_leaf`

It is quite obvious that the model on the left is overfitting, and the model on the right will probably generalize better.



Computational Complexity

- Learning is $O(mn \log_2 m)$ for the basic CART algorithm above.
 - Of course, hyperparameters such as a maximum depth can speed-up learning.
- Prediction is roughly $O(\log_2 m)$ (which is the depth of tree, assuming the tree is balanced, which it often, approximately, is). This is fast!

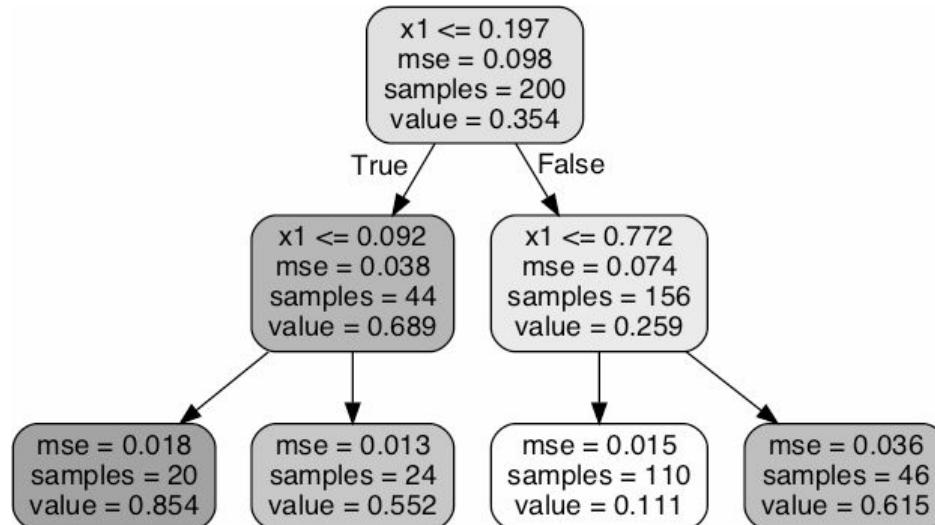
Regression using a Decision Tree

- What does the Decision Tree Regressor predict?
 - Start at the root and follow the decisions down to a leaf.
 - At any node, `value` is the prediction and is simply the mean target values of the training examples that the node applies to.
- For regression, the only difference is the loss function: in place of *Gini* it uses the *mean squared error* between the y -values of the training examples and their mean.

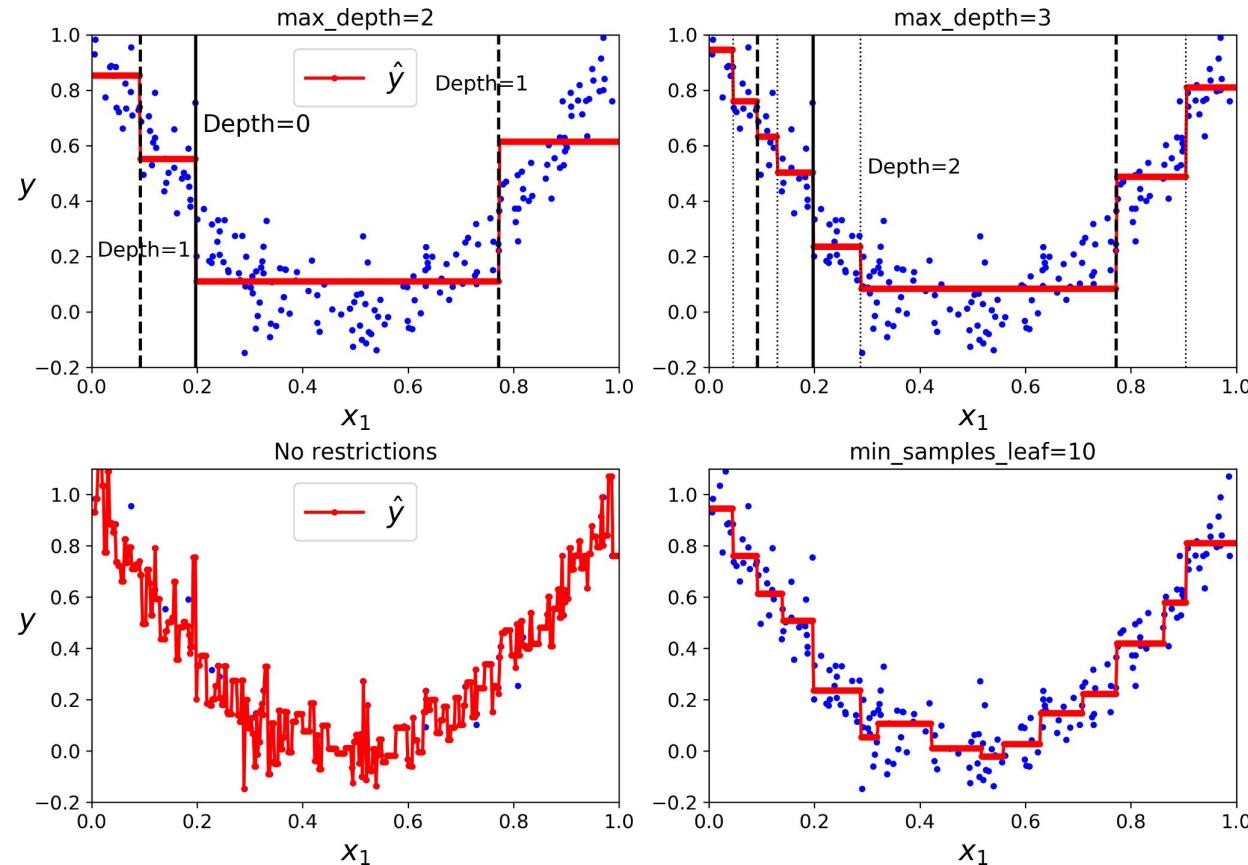
$$\frac{|X_{left}|}{|X|} MSE(X_{left}) + \frac{|X_{right}|}{|X|} MSE(X_{right})$$

Regression using a Decision Tree

```
from sklearn.tree import DecisionTreeRegressor  
  
tree_reg = DecisionTreeRegressor(max_depth=2)  
tree_reg.fit(X, y)
```



Regression using a Decision Tree



Decision Tree: Pros and Cons

- The model is *interpretable*:
 - we can display the tree and people can see what has been learned.
- An individual prediction is *explainable*:
 - we can display the path through the tree.
- Note we do not need to scale the data before using this algorithm.
- They are very sensitive to small variations in the training data (a.k.a. *instability*). Use of PCA or Random Forests can limit this issue.

Parametric vs. Non-parametric Learning

- Parametric learning: the number of parameters is known prior to training and is not affected by m the number of examples in the training set.
 - E.g. Linear Regression the number of parameters is $n + 1$
 - E.g. Polynomial Regression the number of parameters is $\frac{(n + d)!}{n!d!}$
- Non-parametric learning: the number of parameters is not known in advance and may grow with the size of the training set.
 - E.g. For Decision Trees, in some sense, the nodes are the parameters: the structure of the model (tree) may grow to accommodate the complexity of the training data.
 - E.g. For kNN, in some sense, the neighbours are the parameters: the more training examples there are, the more different possible sets of neighbours there are.

Parametric vs. Non-parametric Learning

This does not mean that non-parametric models will have lower validation error.

- Unconstrained, they are prone to overfitting.
- So we want to impose some constraints on the CART algorithm to restrict the shape of the Decision Tree (such as maximum depth and others).
- Similarly, we avoid small values for k in k NN.

Next lecture

Dimensionality Reduction

6th October 2023



IT496: Introduction to Data Mining



Lecture 21

Dimensionality Reduction

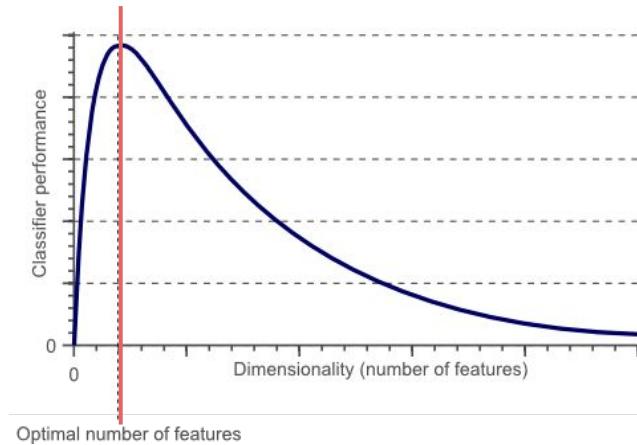
(Slides are created from the lecture notes of ML class at UCD, Ireland)

Arpit Rana
6th October 2023

Motivation

Curse of dimensionality is the phenomenon whereby many machine learning algorithms can perform poorly on high-dimensional data.

- Intuitively, adding more features (dimensions) to a dataset should provide more information about each example, making prediction easier.
- In reality, we often reach a point where adding more features no longer helps, or can even reduce predictive power.



In practice, to build a good predictive model, the number of examples required per feature increases exponentially with number of features.

Motivation

There are often other reasons why we might want to reduce the number of features used to represent data:

- less training time and less computational resources
- extremely useful for data visualization
- one way to avoid the problem of overfitting
- takes care of multicollinearity
- filters out some noise and unnecessary details in the data and increases the overall performance

Understanding which features in our data are informative and which are not is an important knowledge discovery task.

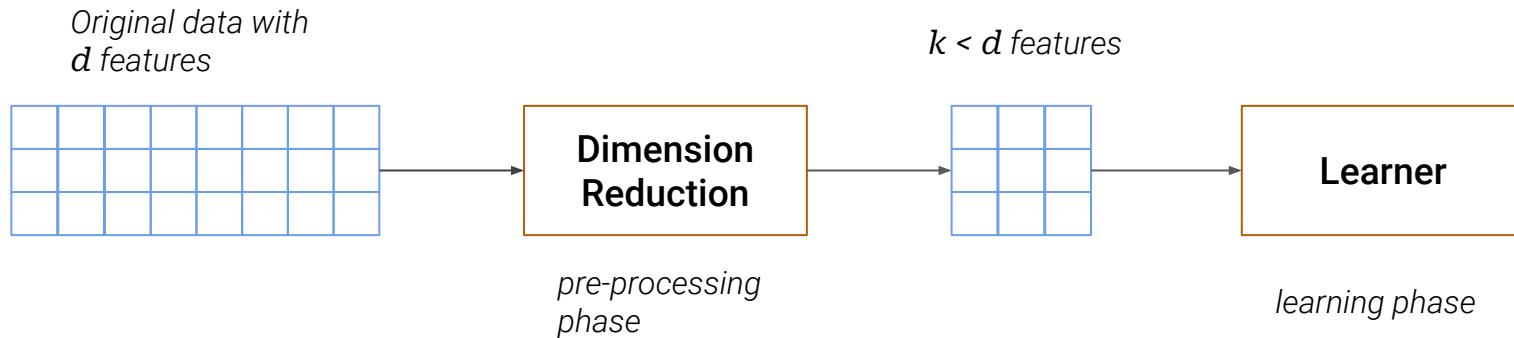
Dimensionality Reduction: Formal Definition

Dimension (or Dimensionality) reduction is -

- the transformation of data from a high-dimensional space into a low-dimensional space
- while preserving as much of the variation in the original dataset as possible.

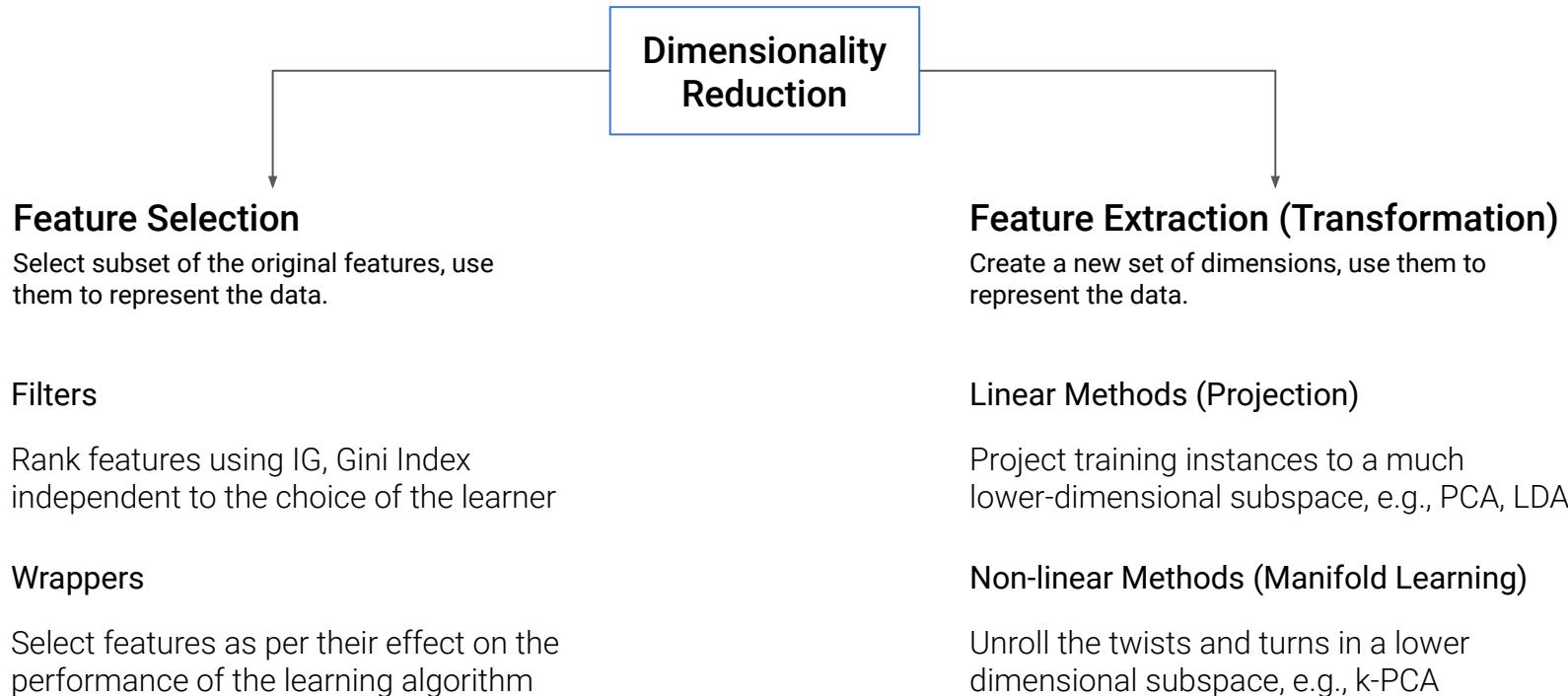
Dimensionality Reduction

- Basic idea is to try to beat the curse of dimensionality:
 - Apply pre-processing techniques to reduce the number of features used to represent a dataset.
 - Then build a model on the smaller feature set.

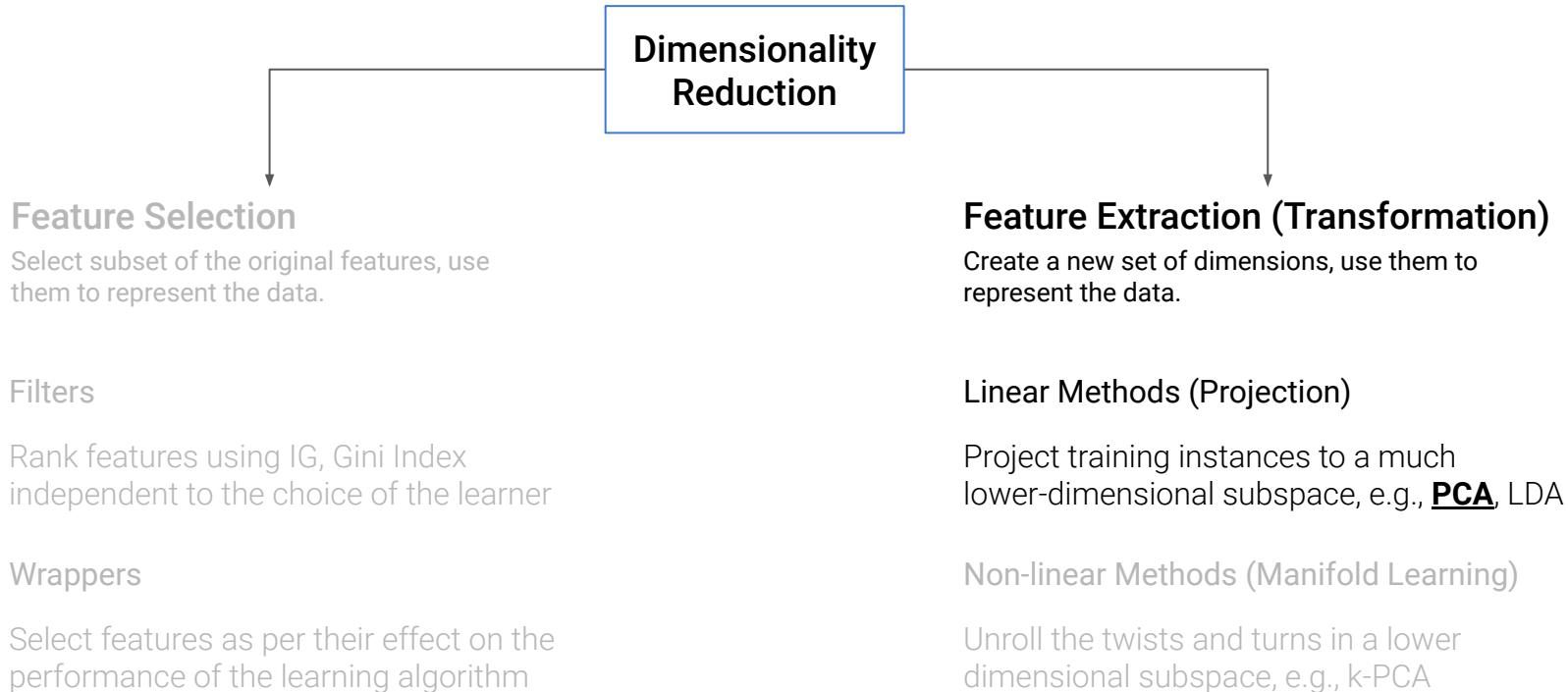


- In many (but not all) cases, the additional information that is lost by removing some features is (more than) compensated by higher classifier accuracy in the lower dimensional space.

Dimensionality Reduction: Strategies



Dimensionality Reduction: Strategies

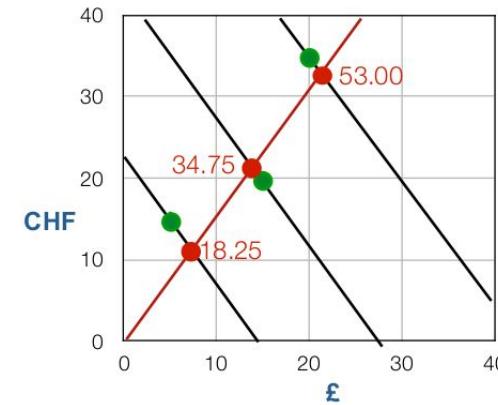


Linear Transformations

In a linear transformation, we map data to new variables, which are linear functions of the original variables.

- Example: Bill owes Mary £ 5 and CHF 15 after a holiday.
 - Current exchange rates: € : £ \rightarrow 1.25 : 1, € : CHF \rightarrow 0.8:1
- Based on rates, Bill owes € $(1.25 \times 5 + 0.8 \times 15) = € 18.25$
- We can view this as a linear transformation...

$$\begin{array}{c|c} \text{£} & \text{CHF} \\ \hline 5 & 15 \\ \hline 15 & 20 \\ \hline 20 & 35 \end{array} \times \begin{array}{c|c} & \\ \hline 1.25 & \\ \hline 0.8 & \end{array} = \begin{array}{c|c} \text{€} & \\ \hline 18.25 & \\ \hline 34.75 & \\ \hline 53.00 & \end{array}$$



Principal Component Analysis (PCA)

Basic Idea

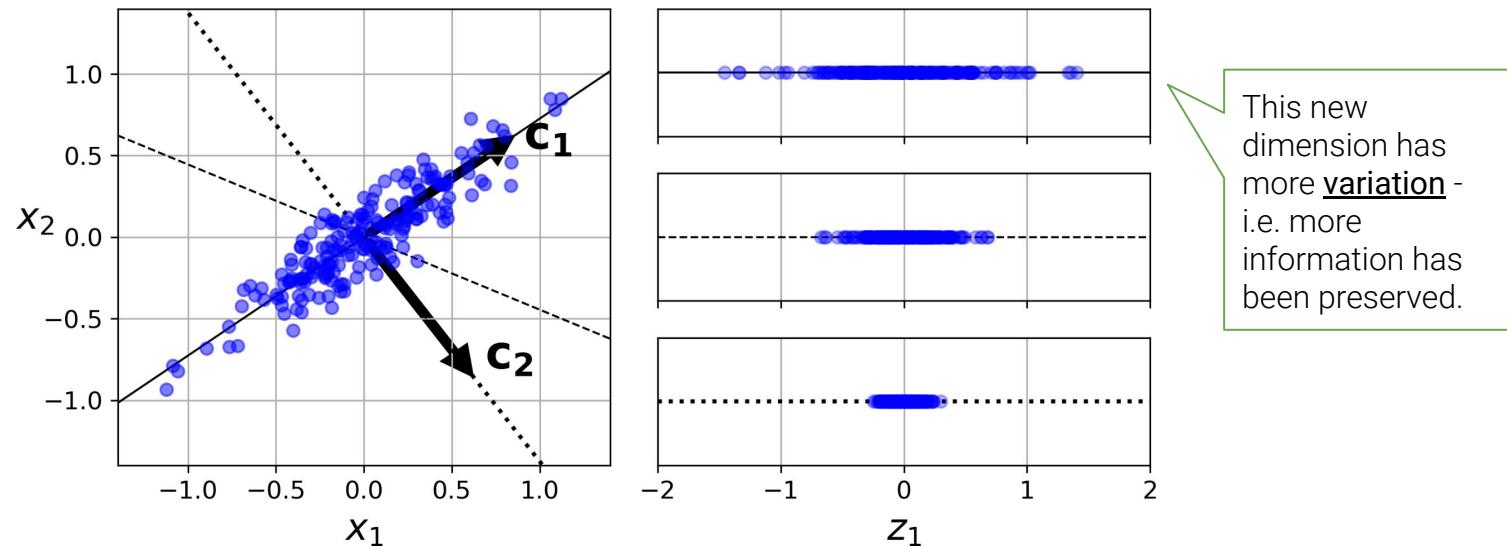
- *Projection methods* map the original d -dimensional space to a new ($k < d$)-dimensional space, with the minimum loss of information.
 - PCA identifies the hyperplane that lies closest to the data, and
 - then it projects the data onto it.

How to choose the hyperplane?

PCA: Preserve the Variance

“Good” spaces for projections are characterised by -

- preserving most of the useful information in the data - the **variation** in the data.
- i.e., minimize the mean squared distance between the original dataset and its projection.



Principal Components

PCA identifies the axis that accounts for the largest amount of variance in the training set.

- It finds a second axis, orthogonal to the first one, that accounts for the largest amount of *remaining variance*.
- Then, a third axis, orthogonal to both previous axes, and a fourth, a fifth, and so on—as many axes as the number of dimensions in the dataset.

The unit vector that defines the i^{th} axis is called the i^{th} principal component (PC).

Principal Components

Singular Value Decomposition (SVD)

- that can decompose the training set matrix X into the matrix multiplication of three matrices $U \Sigma V^T$, where V contains **unit vectors** that define all the principal components.

$$V = \begin{pmatrix} | & | & & | \\ c_1 & c_2 & \cdots & c_n \\ | & | & & | \end{pmatrix}$$

Projecting down to d dimension

$$X_{d\text{-proj}} = XW_d$$

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

PCA using Scikit-learn

- Scikit-Learn's PCA class implements PCA using SVD decomposition just like we did before.
- The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
from sklearn.decomposition import PCA  
  
pca = PCA(n_components = 2)  
X2D = pca.fit_transform(X)
```

- We can access the principal components using the `components_` variable (note that it contains the PCs as row vectors, i.e., transpose of W_d).
 - The first principal component is equal to `pca.components_.T[:, 0]`

Explained Variance Ratio

- Another very useful piece of information is the *explained variance ratio* of each principal component, available via the `explained_variance_ratio_` variable.
- It indicates the proportion of the dataset's variance that lies along the axis of each principal component.

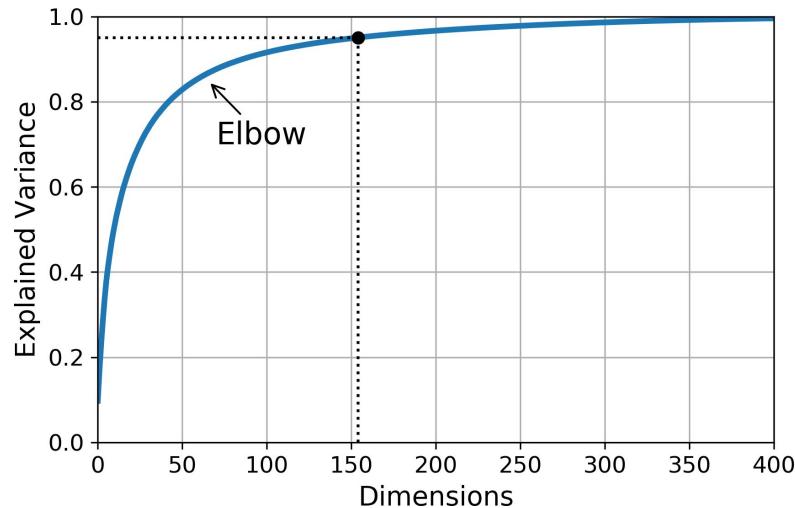
```
>>> pca.explained_variance_ratio_
array([0.84248607, 0.14631839])
```

- This tells us that 84.2% of the dataset's variance lies along the first axis, and 14.6% lies along the second axis.

Choosing the Right Number of Dimensions

It is generally preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%).

```
pca = PCA(n_components=0.95)  
X_reduced = pca.fit_transform(X_train)
```



PCA for Compression

It is also possible to *decompress* the reduced dataset back to its original dimensions by applying the *inverse transformation* of the PCA projection.

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}} \mathbf{W}_d^T$$

- This won't give you back the exact original data, since the projection lost a bit of information (e.g., within the 5% variance that was dropped), but it will likely be quite close to the original data.
- The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the *reconstruction error*.

$$Error_{\text{reconstruction}} = MSE(X, X_{\text{recovered}})$$

PCA for Compression

Applying PCA to the MNIST dataset while preserving 95% of its variance: in place of 784 (28 x 28 shape images) original features, just using 154 (nearly 20% of the original).

```
pca = PCA(n_components = 154)  
X_reduced = pca.fit_transform(X_train)  
X_recovered = pca.inverse_transform(X_reduced)
```

Original	Compressed
0 0 1 7 3	0 0 1 7 3
5 9 1 3 2	5 9 1 3 2
5 8 5 8 8	5 8 5 8 8
2 6 4 0 3	2 6 4 0 3
3 6 0 3 1	3 6 0 3 1

Conclusions

- Each eigenvector has a direction - i.e. it is a dimension.
- Eigenvectors of symmetric matrices are orthogonal to each other - i.e. they point in completely different directions.
- Each eigenvalue is a number indicating how much variance there is in the data in that direction.
- Principal Components (PCs): New dimensions constructed as linear combinations of the original features, which are uncorrelated with one another. Constructed from eigenvectors.
- The first PC accounts for the most variability in the data. The next PC has the highest variance possible under the constraint that it is uncorrelated with the first PC, and so on...

Next lecture

Neural Networks

10th October 2023

IT496: Introduction to Data Mining



Ensemble Methods (Bagging, Boosting, Stacking ..)

Himanshu Beniwal

06th Oct 2023

Ensemble Methods

- **Bagging:** It is a homogeneous weak learners' model that learns from each other independently in parallel and combines them for determining the model average.
- **Boosting:** It is also a homogeneous weak learners' model but works differently from Bagging. In this model, learners learn sequentially and adaptively to improve model predictions of a learning algorithm.

Bagging (Random Forest)

- It is an ensemble machine learning algorithm called **Bootstrap Aggregation** or **bagging**.
- The **bootstrap** is a powerful statistical method for estimating a quantity from a data sample. This is easiest to understand if the quantity is a descriptive statistic such as a mean or a standard deviation.
- Example

Sample of 100 values (x) and we'd like to get an estimate of the mean of the sample.

$$\text{mean}(x) = 1/100 * \text{sum}(x)$$

Bagging (Random Forest)

- It is an ensemble machine learning algorithm called **Bootstrap Aggregation** or **bagging**.
- The **bootstrap** is a powerful statistical method for estimating a quantity from a data sample. This is easiest to understand if the quantity is a descriptive statistic such as a mean or a standard deviation.
- Example

Sample of 100 values (x) and we'd like to get an estimate of the mean of the sample.

$$\text{mean}(x) = 1/100 * \text{sum}(x)$$

[bootstrap]

- Create many (e.g. 1000) random sub-samples of our dataset with replacement (meaning we can select the same value multiple times).
- Calculate the mean of each sub-sample.
- Calculate the average of all of our collected means and use that as our estimated mean for the data.

For example, let's say we used 3 resamples and got the mean values 2.3, 4.5 and 3.3. Taking the average of these we could take the estimated mean of the data to be 3.367.

Bagging

- Ensemble meta-algorithm to improve the stability and accuracy used in statistical classification and regression.
- It decreases the variance and helps to avoid overfitting.
- It is usually applied to decision tree methods.
- *Bagging is a special case of the model averaging approach.*

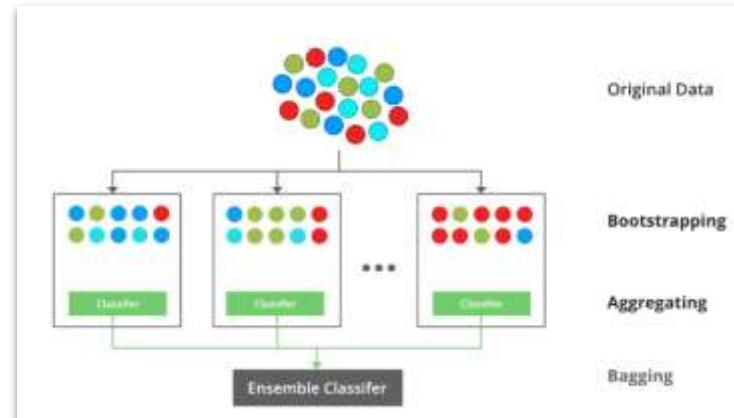
Bagging

- Ensemble meta-algorithm to improve the stability and accuracy used in statistical classification and regression.
- It decreases the variance and helps to avoid overfitting.
- It is usually applied to decision tree methods.
- *Bagging is a special case of the model averaging approach.*

Suppose a set D of d tuples, at each iteration i , a training set D_i of d tuples is selected via row sampling with a replacement method (i.e., there can be repetitive elements from different d tuples) from D (i.e., bootstrap). Then a classifier model M_i is learned for each training set $D < i$. Each classifier M_i returns its class prediction. The bagged classifier M^* counts the votes and assigns the class with the most votes to X (unknown sample).

Bagging (Implementation)

- **Step 1:** Multiple subsets are created from the original data set with equal tuples, selecting observations with replacement.
- **Step 2:** A base model is created on each of these subsets.
- **Step 3:** Each model is learned in parallel with each training set and independent of each other.
- **Step 4:** The final predictions are determined by combining the predictions from all the models.



Source: [Geeksforgeeks](#)

Bagging (Implementation)

```
class BaggingClassifier:  
    def __init__(self, base_classifier, n_estimators):  
        self.base_classifier = base_classifier  
        self.n_estimators = n_estimators  
        self.classifiers = []  
  
    def fit(self, X, y):  
        for _ in range(self.n_estimators):  
            # Bootstrap Sampling with Replacement  
            indices = np.random.choice(len(X), len(X), replace=True)  
            X_sampled = X[indices]  
            y_sampled = y[indices]  
  
            # Create a new base classifier and train it on the sampled data  
            classifier = self.base_classifier.__class__()  
            classifier.fit(X_sampled, y_sampled)  
  
            # Store the trained classifier in the list of classifiers  
            self.classifiers.append(classifier)  
        return self.classifiers  
  
    def predict(self, X):  
        # Make predictions using all the base classifiers  
        predictions = [classifier.predict(X) for classifier in self.classifiers]  
        # Aggregate predictions using majority voting  
        majority_votes = np.apply_along_axis(lambda x: np.bincount(x).argmax(), axis=0, arr=predictions)  
  
        return majority_votes
```

```
# Import the necessary libraries  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.datasets import load_digits  
from sklearn.model_selection import train_test_split  
from sklearn.metrics import accuracy_score  
  
# Load the dataset  
digit = load_digits()  
X, y = digit.data, digit.target  
  
# Split the data into training and testing sets  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)  
  
# Create the base classifier  
dc = DecisionTreeClassifier()  
model = BaggingClassifier(base_classifier=dc, n_estimators=10)  
model.fit(X_train, y_train)  
  
# Make predictions on the test set  
y_pred = model.predict(X_test)  
  
# Calculate accuracy  
accuracy = accuracy_score(y_test, y_pred)  
print("Accuracy:", accuracy)
```

Bagging (Advantages)

- **Improved Predictive Performance:** Often outperforms single classifiers by reducing overfitting and increasing predictive accuracy. By combining multiple base models, it can better generalize to unseen data.

Bagging (Advantages)

- **Improved Predictive Performance:** Often outperforms single classifiers by reducing overfitting and increasing predictive accuracy. By combining multiple base models, it can better generalize to unseen data.
- **Robustness:** Bagging reduces the impact of outliers and noise in the data by aggregating predictions from multiple models. This enhances the overall stability and robustness of the model.

Bagging (Advantages)

- **Improved Predictive Performance:** Often outperforms single classifiers by reducing overfitting and increasing predictive accuracy. By combining multiple base models, it can better generalize to unseen data.
- **Robustness:** Bagging reduces the impact of outliers and noise in the data by aggregating predictions from multiple models. This enhances the overall stability and robustness of the model.
- **Reduced Variance:** Since each base model is trained on different subsets of the data, the aggregated model's variance is significantly reduced compared to an individual model.

Bagging (Advantages)

- **Improved Predictive Performance:** Often outperforms single classifiers by reducing overfitting and increasing predictive accuracy. By combining multiple base models, it can better generalize to unseen data.
- **Robustness:** Bagging reduces the impact of outliers and noise in the data by aggregating predictions from multiple models. This enhances the overall stability and robustness of the model.
- **Reduced Variance:** Since each base model is trained on different subsets of the data, the aggregated model's variance is significantly reduced compared to an individual model.
- **Parallelization:** Bagging allows for parallel processing, as each base model can be trained independently. This makes it computationally efficient, especially for large datasets.

Bagging (Advantages)

- **Improved Predictive Performance:** Often outperforms single classifiers by reducing overfitting and increasing predictive accuracy. By combining multiple base models, it can better generalize to unseen data.
- **Robustness:** Bagging reduces the impact of outliers and noise in the data by aggregating predictions from multiple models. This enhances the overall stability and robustness of the model.
- **Reduced Variance:** Since each base model is trained on different subsets of the data, the aggregated model's variance is significantly reduced compared to an individual model.
- **Parallelization:** Bagging allows for parallel processing, as each base model can be trained independently. This makes it computationally efficient, especially for large datasets.
- **Flexibility:** Bagging Classifier is a versatile technique that can be applied to a wide range of machine learning algorithms, including decision trees, random forests, and support vector machines.

Bagging (Applications)

- **Fraud Detection:** Bagging Classifier can be used to detect fraudulent transactions by aggregating predictions from multiple fraud detection models.

Bagging (Applications)

- **Fraud Detection:** Bagging Classifier can be used to detect fraudulent transactions by aggregating predictions from multiple fraud detection models.
- **Spam filtering:** Bagging classifier can be used to filter spam emails by aggregating predictions from multiple spam filters trained on different subsets of the spam emails.

Bagging (Applications)

- **Fraud Detection:** Bagging Classifier can be used to detect fraudulent transactions by aggregating predictions from multiple fraud detection models.
- **Spam filtering:** Bagging classifier can be used to filter spam emails by aggregating predictions from multiple spam filters trained on different subsets of the spam emails.
- **Credit scoring:** Bagging classifier can be used to improve the accuracy of credit scoring models by combining the predictions of multiple models trained on different subsets of the credit data.

Bagging (Applications)

- **Fraud Detection:** Bagging Classifier can be used to detect fraudulent transactions by aggregating predictions from multiple fraud detection models.
- **Spam filtering:** Bagging classifier can be used to filter spam emails by aggregating predictions from multiple spam filters trained on different subsets of the spam emails.
- **Credit scoring:** Bagging classifier can be used to improve the accuracy of credit scoring models by combining the predictions of multiple models trained on different subsets of the credit data.
- **Image Classification:** Bagging classifier can be used to improve the accuracy of image classification tasks by combining the predictions of multiple classifiers trained on different subsets of the training images.

Bagging (Applications)

- **Fraud Detection:** Bagging Classifier can be used to detect fraudulent transactions by aggregating predictions from multiple fraud detection models.
 - **Spam filtering:** Bagging classifier can be used to filter spam emails by aggregating predictions from multiple spam filters trained on different subsets of the spam emails.
 - **Credit scoring:** Bagging classifier can be used to improve the accuracy of credit scoring models by combining the predictions of multiple models trained on different subsets of the credit data.
 - **Image Classification:** Bagging classifier can be used to improve the accuracy of image classification tasks by combining the predictions of multiple classifiers trained on different subsets of the training images.
 - **Natural language processing:** In NLP tasks, the bagging classifier can combine predictions from multiple language models to achieve better text classification results.
-

2. Boosting

- Attempts to build a strong classifier from the number of weak classifiers. It is done by building a model by using weak models in series.
- Firstly, a model is built from the training data. Then the second model is built which tries to correct the errors present in the first model.
- This procedure is continued and models are added until either the complete training data set is predicted correctly or the maximum number of models is added.

2. Boosting (Similarities)

- Both are ensemble methods to get N learners from 1 learner.
- Both generate several training data sets by random sampling.
- Both make the final decision by averaging the N learners (or taking the majority of them i.e Majority Voting).
- Both are good at reducing variance and provide higher stability.

2. Boosting (Differences)

S.NO	Bagging	Boosting
1.	The simplest way of combining predictions that belong to the same type.	A way of combining predictions that belong to the different types.
2.	Aim to decrease variance, not bias.	Aim to decrease bias, not variance.
3.	Each model receives equal weight.	Models are weighted according to their performance.
4.	Each model is built independently.	New models are influenced by the performance of previously built models.
5.	Different training data subsets are selected using row sampling with replacement and random sampling methods from the entire training dataset.	Every new subset contains the elements that were misclassified by previous models.
6.	Bagging tries to solve the over-fitting problem.	Boosting tries to reduce bias.
7.	If the classifier is unstable (high variance), then apply bagging.	If the classifier is stable and simple (high bias) then apply boosting.
8.	In this base classifiers are trained parallelly.	In this base classifiers are trained sequentially.
9	Example: The Random forest model uses Bagging.	Example: The AdaBoost uses Boosting techniques

IT496: Introduction to Data Mining

That's all! :)

[Colab link](#) ←

IT496: Introduction to Data Mining



Lecture 24

Introduction to Neural Networks

(Slides are created from the lecture notes of Dr. Derek Bridge, UCC, Ireland)

Arpit Rana
10th October 2023

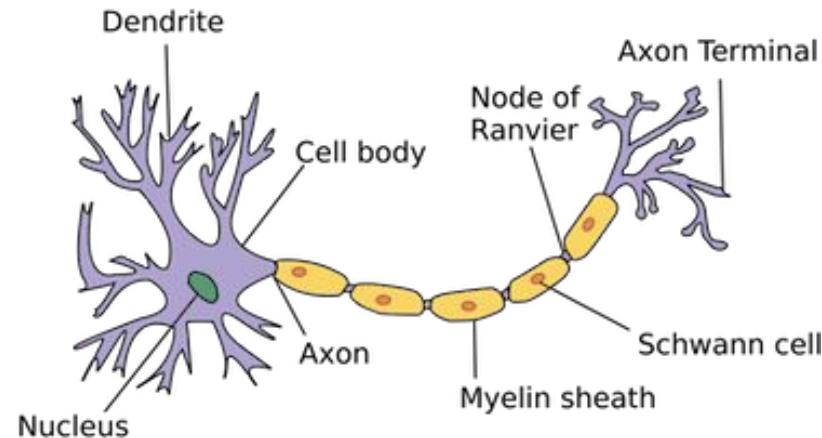
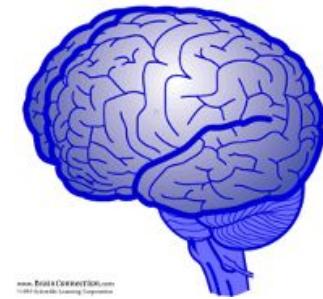
Introduction

Neural Networks are loosely inspired by what we know about our brains:

- Networks of neurons.
- However, they are not models of our brains.
 - E.g. there is no evidence that the brain uses the learning algorithm that is used by neural networks.

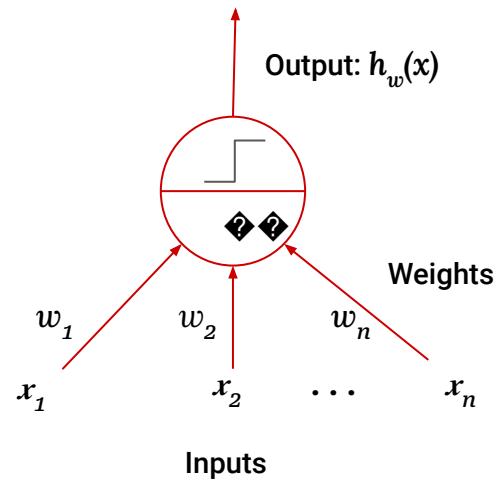
Biological Neurons

- Our brain is a network of about 10^{11} neurons, each connected to about 10^4 others
- Sufficient electrical activity on a neuron's dendrites causes an electrical pulse to be sent down the axon, where it may activate other neurons.



Artificial Neuron

- A simple artificial neuron has n real-valued inputs, x_1, \dots, x_n .
- The connections have real-valued weights, w_1, \dots, w_n .
- The neuron also has a number b called the bias.



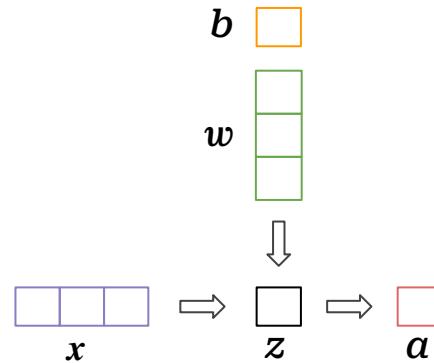
Artificial Neuron

- The neuron computes the weighted sum of its inputs and adds b :

$$z = b + w_1x_1 + w_2x_2 + \dots + w_nx_n$$

or if x is a row vector of the inputs and w is a (column) vector of the weights

$$z = b + xw$$



Although artificial neurons are inspired by real neurons, really all we're doing is the dot product of two vectors, followed by element-wise application of the activation function.

Artificial Neuron

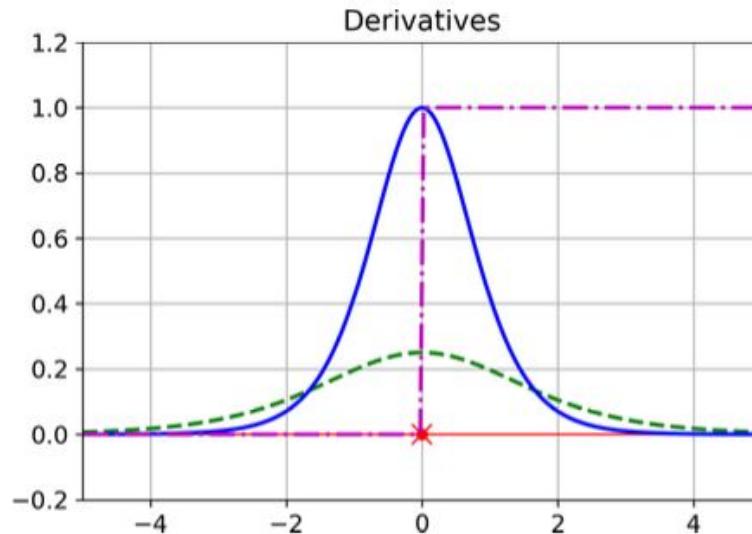
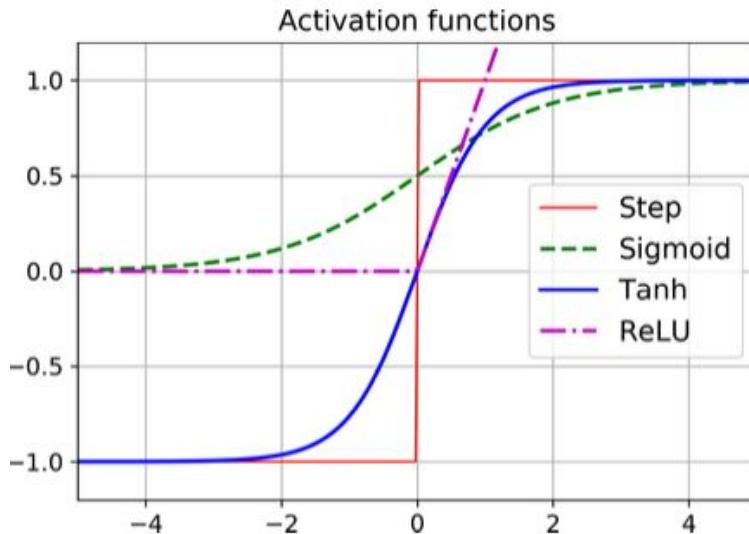
Many activation functions have been proposed, including:

- **linear** activation function: $g(z) = z$
- **step** activation function: $g(z) = \begin{cases} 0 & z < 0, \\ 1 & z \geq 0 \end{cases}$
- **sigmoid** activation function: $g(z) = \frac{1}{1 + e^{-z}}$
- **tanh** activation function (tanh is the hyperbolic tangent): $g(z) = \tanh z$
- **ReLU** activation function (ReLU stands for Rectified Linear Unit): $g(z) = \max(0, z)$

Apart from the **linear** activation function, these activation functions are **non-linear**, which is important to the power of neural networks.

Artificial Neuron

Activation functions and their derivatives



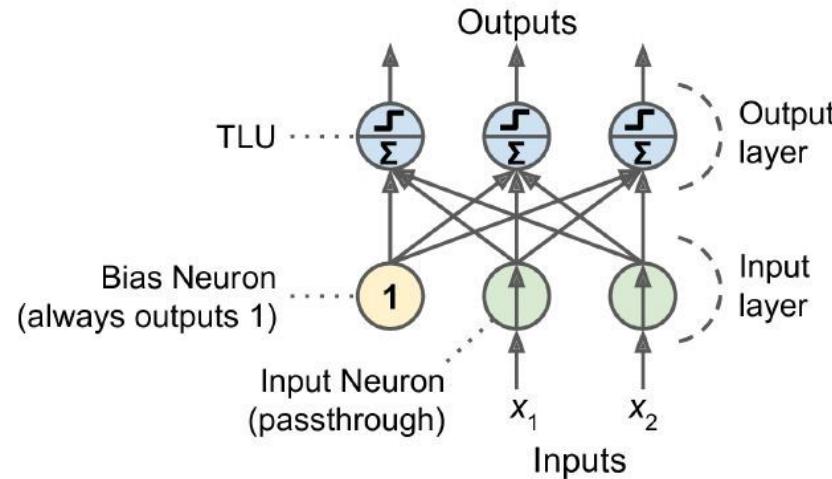
Relationship with Linear Models

- A single artificial neuron that uses the linear activation function gives us the same linear models that we had in Linear Regression.
 - If we find the values of the weights and bias using MSE as our loss function, then we will be doing OLS regression.
- A single artificial neuron that uses the sigmoid activation function gives us the same models that we had when using Logistic Regression for binary classification.
 - We can set the weights using the binary cross-entropy function as our loss function.

Layers of Neurons

We don't usually have just one neuron. We have a layer, containing several neurons.

- For now let's consider what is called a dense layer (also a *fully-connected layer*): every input is connected to *every neuron in the layer*.

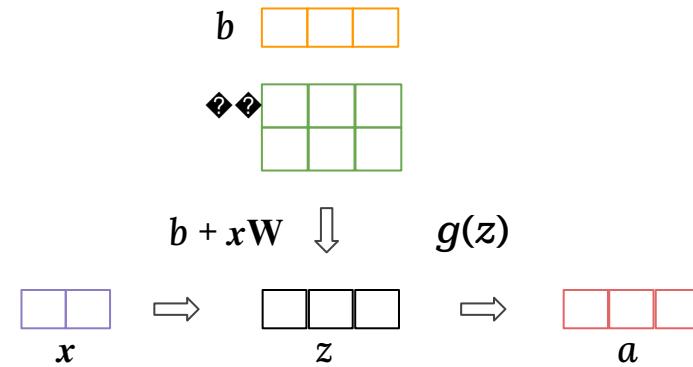
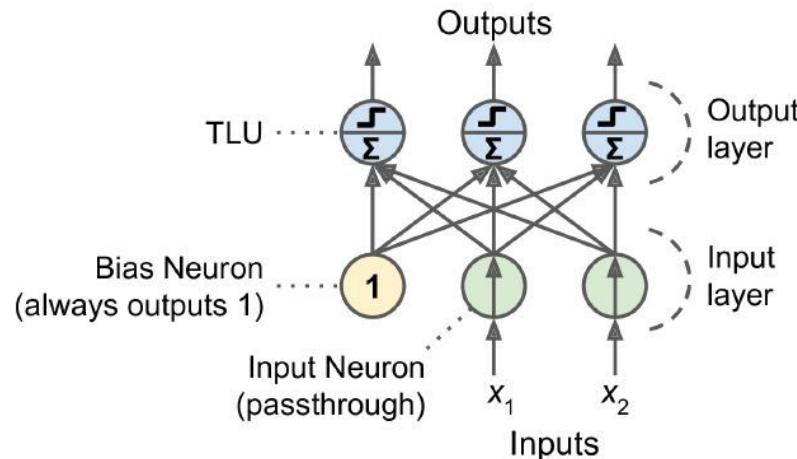


- So now we have more than one output, one per neuron, each calculated as before.

Matrix Multiplication

Suppose there are m inputs and p neurons in a layer. We can put all the weights into a $m \times p$ matrix:

$$\underline{m = 2, p = 3}$$

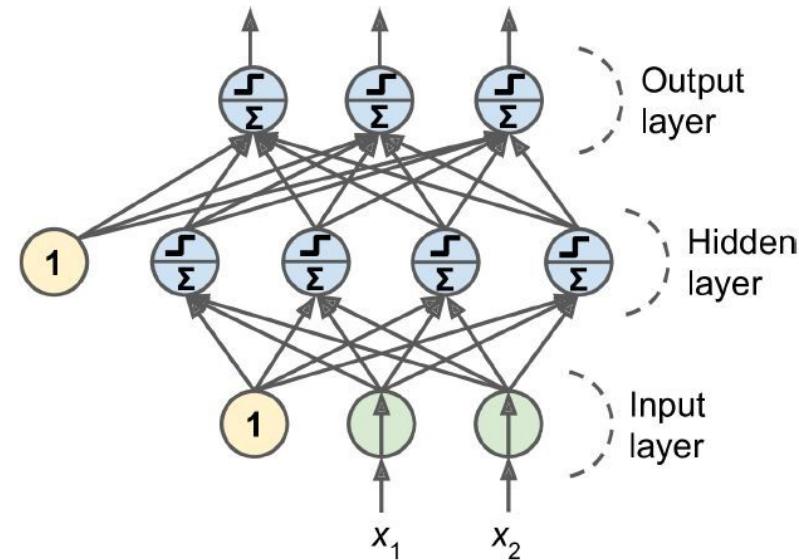


Multi-layer Neural Network

Let's assume we have multiple layers and they are also *dense* layers. These neural networks contain:

- an *input layer* (although this is not a layer of neurons);
- one or more *hidden layers*;
- an *output layer*.

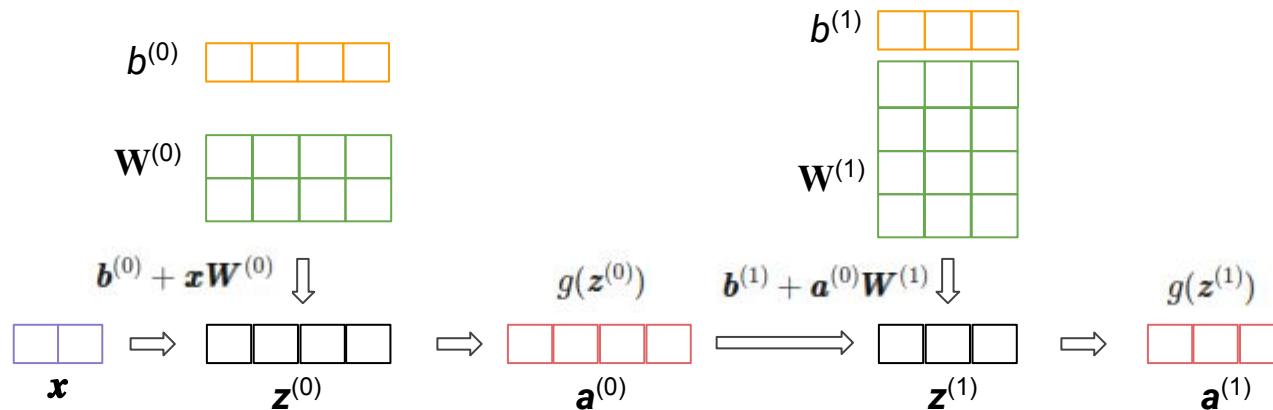
Every neuron has a *bias*.



- The network shown in the diagram is a *layered, dense, feedforward* network.
- The depth of a MLNN is simply the number of layers of neurons.

Matrix Multiplication Again

Similarly, we can obtain output for the second layer, and so on.



Matrix Multiplication Again

- When we make predictions for unseen examples, we often want predictions, not for a single object x , but for a set of objects X .
 - This is also true during training, in the case of *Batch Gradient Descent* and *Mini-Batch Gradient Descent*.

$$\mathbf{Z}^{(0)} = \mathbf{b}^{(0)} + \mathbf{X}\mathbf{W}^{(0)} \text{ and } \mathbf{A}^{(0)} = g(\mathbf{Z}^{(0)})$$

$$\mathbf{Z}^{(1)} = \mathbf{b}^{(1)} + \mathbf{Z}^{(0)}\mathbf{W}^{(1)} \text{ and } \mathbf{A}^{(1)} = g(\mathbf{Z}^{(1)})$$

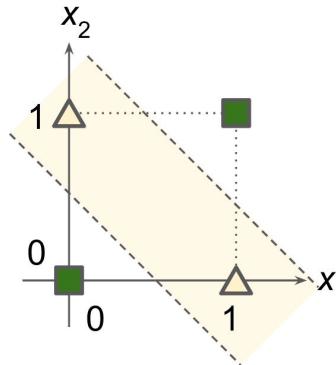
Matrix Multiplication Again

- This is all that a neural network consists of! They are just collections of:
 - matrix multiplications; and
 - element-wise activation functions.
- In general, they are collections of:
 - affine transformations (matrix multiplication being one example of an affine transformation, which are linear operations); and
 - element-wise functions (activation functions being one example).
- Looking at neural networks in this way also helps us realise that a neural network simply defines a function as a composite of other functions.
- In the example above, the whole network computes the following:

$$g^{(1)}(g^{(0)}(\mathbf{X}\mathbf{W}^{(0)} + \mathbf{b}^{(0)})\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$$

Why Do We Need More Layers?

- A single neuron (or layer of neurons) gives us linear models.
- With linear models, there are problems we cannot solve, e.g., we cannot build a classifier that correctly classifies exclusive-or:

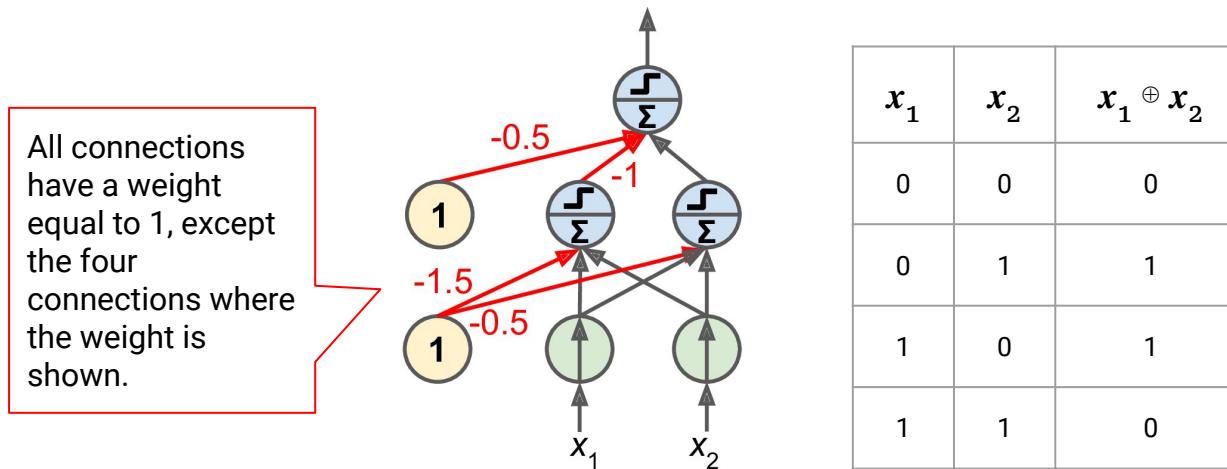


x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Note: A recent paper in Science Magazine claims that a single layer of biological neurons can compute exclusive-or. If true, this confirms what we said earlier: artificial neural networks are inspired by the human brain, but they are not a model of the human brain.

Why Do We Need More Layers?

But, a two-layer network that can correctly classify exclusive-or.



So, with multiple layers of neurons and the non-linearities of their activation functions, we can eliminate these limitations.

Why Do We Need More Layers?

In general, MLNN has the following advantages:

- Other things being equal, each extra hidden layer enlarges the set of hypotheses that the network can represent: increasing complexity.
- In fact, the universal approximation theorem states that a feed-forward network with a finite (but arbitrarily large) single hidden layer can approximate any continuous function (to any desired precision), under mild assumptions on the activation function.

Training a Neural Network

Neural networks learn by modifying the values of the weights and biases.

- It is our job to decide on the neural network architecture (structure).
- It is our job to choose the values of numerous hyperparameters that we will encounter.
 - The hyperparameters of a neural network are the number of layers, number of neurons in each layer, activation function, loss function, optimizer, learning rate, batch size, etc.
- But, we use a dataset and a learning algorithm to find the values of the network's parameters.
 - The parameters of a neural network are its *weights* and *biases*.

Training a Neural Network

- A lot of this is done using *supervised learning*:
 - So we need a *labeled dataset*;
 - a *loss function*; and
 - a learning algorithm known as *backpropagation* (or *backprop*) that uses some variant of *Gradient Descent*.

Next lecture

Neural Network Examples

17th October 2023

IT496: Introduction to Data Mining



Lecture 25

Neural Network Examples Using Keras

(Slides are created from the lecture notes of Dr. Derek Bridge, UCC, Ireland)

Arpit Rana
17th October 2023

Deep Learning

- The word 'deep' in 'deep learning' does not mean profound.
- In deep learning, we have 'lots' of layers – tens or even hundreds.

Representations

One way of thinking about Machine Learning:

- It uses guidance from a feedback signal to automatically find transformations that turn input data into more useful representations.

For example,

- in the case of supervised learning, the feedback comes from the loss function and the algorithm seeks a representation that is closer to the target outputs.

Representations

Deep learning is about jointly finding successive layers of representations, usually in the form of the layers of a neural network.

- The network takes in vectors (examples).
- The first layer in some sense transforms the input vectors into new vectors – a different representation of the inputs examples.
- The second layer transforms again into new vectors – another representation.
- Since each layer produces a new representations, one way of thinking about this is, for the kinds of tasks on which it is successful, deep learning automates feature engineering.

Drivers of Deep Learning

Hardware:

- Faster CPUs but then highly-parallel Graphical Processing Units (GPUs) and now specially-designed Tensor Processing Units (TPUs).

Data:

- Sensors and the Internet have made vast datasets available: text, images, video, ...

Algorithmic advances:

- The core ideas have been around a long time: Perceptrons (1950s), backpropagation (1980s or earlier), convolutional networks (1980s), LSTMs (1990s), ...
- But new ideas from 2010 onwards: better weight initialization, batch normalization, different activation functions, variants of SGD, numerous ways to avoid overfitting, new architectures,...

Freeware:

- Toolkits/APIs; Educational resources.

Money!

Applications of Deep Learning

It is excelling at 'perceptual' tasks, e.g.

- image classification;
- image segmentation;
- speech recognition;
- handwriting transcription.

But it is finding ever wider application:

- video recommendation;
- machine translation;
- text-to-speech;
- question-answering;
- autonomous driving;
- the protein folding problem (AlphaFold);
- superhuman game playing (e.g. AlphaGo).

Implementation

In this lecture:

- We will use layered, dense, feedforward neural networks for regression, binary classification and multi-class classification:
 - We'll use our two small datasets that contain **structured data** (sometimes called **tabular data**): not necessarily ideal for deep learning.
 - We'll see one example that uses images.
- This will illustrate some of the different activation functions we can use:
 - in the output layer: linear, sigmoid or softmax; and
 - in the hidden layers: sigmoid or ReLU.
- This will also introduce the Keras library.

The Keras Library

scikit-learn has very limited support for neural networks.

Tensorflow and PyTorch are the two main libraries that do support tensor computation, neural networks and deep learning in Python:

We will use Keras, which is a high-level API for Tensorflow, first released in 2015 by François Chollet of Google (<https://keras.io>):

- It is very high-level, making it easy to construct networks, fit models and make predictions.
- The downside is it gives less fine-grained control than TensorFlow itself. When fine-grained control is needed, you can mix in TensorFlow functions, methods and classes.
- This seems a suitable trade-off for us: our module is about AI, not the intricacies of TensorFlow.

Keras Concepts

Network Architecture: Number of Hidden Layers

- Neural network with no hidden layers is just a linear model.
- Hidden layers are needed when data is not linearly separable.
 - Try to avoid more than 2 hidden layers otherwise it will increase the model complexity.
 - For very large datasets, gradually ramp up the number of hidden layers until you start overfitting the training set.

Keras Concepts

Network Architecture: Number of Neurons in Hidden Layers

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be $2/3$ the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

Keras Concepts

Layers are the building blocks.

- To begin with, we will use **dense (fully connected)** layers.

The activation functions of hidden layers are open for you to choose, e.g. **sigmoid** or **ReLU**.

- But the *activation functions* of output layers are determined by the task:
 - **Regression:** linear activation function (default);
 - **Binary classification:** sigmoid activation function; and
 - **Multiclass classification:** softmax activation function.

Layers are combined into **networks**:

- Consecutive layers must be compatible: the shape of the input to one layer is the shape of the output of the preceding layer.

Keras Concepts

Once the network is built, we compile it, specifying:

A **loss function**:

- Regression, e.g. mean-squared-error (mse);
- Binary classification, e.g. (binary) cross-entropy (binary_crossentropy);
- Multiclass classification, e.g. (categorical) cross-entropy
(sparse_categorical_crossentropy if the labels are encoded as integer labels or categorical_crossentropy if the integer labels are then also one-hot encoded).

An **optimizer**, such as SGD – but see below.

A **list of metrics** to monitor during training and testing:

- Regression, e.g. mean-absolute-error (mae);
- Classification, e.g. accuracy (acc).

Keras Optimizers

We know about Gradient Descent: Batch, Mini-Batch, Stochastic.

Without going into details, many other variants of Gradient Descent have been devised (e.g. RMSprop, Adam, Nadam, Adagrad ,...):

- some may have better convergence behaviour in the case of local minima;
- some may converge more quickly.

although a disadvantage is that they typically introduce further hyperparameters (e.g. momentum) in addition to learning rate.

Keras Optimizers

We will use RMSprop below.

- Be aware, its default *learning rate* is 0.001. This is usually OK, but in some cases you may need to change it.
- Be aware too that there is an argument called `batch_size`. Assuming we set its value to somewhere between 1 and the size of the training set then we are getting Mini-Batch Gradient Descent.

A Neural Network for Regression

For regression on structured/tabular data, we might use a network with the following architecture:

- **Input layer:** one input per feature.
- **Hidden layers:** one or more hidden layers.
 - Activation function for neurons in hidden layers can be the sigmoid function or ReLU.
- **Output layer:** just one output neuron (assuming we're predicting a single number).
 - Activation function for the output neuron should be the linear function: $g(\mathbf{z}) = \mathbf{z}$

There are also biases in each layer except the output layer – Keras will give us these 'for free'!

Example: Property Rent Prediction

We don't want too many hidden layers, nor too many neurons in each hidden layer. Why?

Let's start with this:

- An input layer with three inputs (BHK, Size, Bathrooms);
- Two hidden layers, with 64 neurons in each, and ReLU activation function;
- An output layer with a single neuron and linear activation function.

We need to scale the features. But, since we are now not using scikit-learn's `ColumnTransformers` to create a preprocessor, we need to take care of the scaling.

Example: Property Rent Prediction

```
from sklearn.model_selection import train_test_split

hr_train_x, hr_test_x, hr_train_y, hr_test_y = train_test_split(df_reduced[['BHK', 'Size', 'Bathroom']].to_numpy(),
                                                               df_reduced[['Rent']].to_numpy(),
                                                               train_size=0.8, shuffle=True, random_state=25
                                                               )
hr_train_x.shape, hr_test_x.shape

((3796, 3), (950, 3))
```

```
from tensorflow import keras

hr_model = keras.models.Sequential([
    keras.layers.InputLayer(input_shape=(3,)),
    keras.layers.Normalization(),
    keras.layers.Dense(32, activation="relu"),
    keras.layers.Dense(32, activation="relu"),
    keras.layers.Dense(1, activation="linear")
])
```

Example: Property Rent Prediction

```
hr_model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=0.001),  
                 loss="mse",  
                 metrics=["mae"]  
)
```

```
history = hr_model.fit(hr_train_x, hr_train_y,  
                       epochs=30,  
                       batch_size=64,  
                       validation_split=0.1,  
                       verbose="auto"  
)
```

```
test_loss, test_accuracy = hr_model.evaluate(hr_test_x, hr_test_y)  
test_accuracy
```

```
30/30 [=====] - 0s 1ms/step - loss: 3025089536.0000 - mae: 27686.8379  
27686.837890625
```

A Neural Network for Binary Classification

For binary classification, we might use a network with the following architecture:

- **Input layer:** one input per feature.
- **Hidden layers:** one or more hidden layers.
 - Activation function for neurons in hidden layers can be sigmoid or ReLU.
- **Output layer:** just one output neuron (for binary classification).
 - Activation function for the output neuron should be the sigmoid function also. Why?

Example: Class Performance Dataset

Let's start with this:

- An input layer with 3 inputs (lec, lab, cao).
- Two hidden layers, with 64 neurons in each, and ReLU activation function.
- An output layer with a single neuron and sigmoid activation function.

We'll scale using a Normalization layer.

Example: Class Performance Dataset

```
inputs = Input(shape=(3,))
x = Normalization()(inputs)
x = Dense(64, activation="relu")(x)
x = Dense(64, activation="relu")(x)
outputs = Dense(1, activation="sigmoid")(x)
cs1109_model = Model(inputs, outputs)

cs1109_model.compile(optimizer=RMSprop(learning_rate=0.001), loss="binary_crossentropy", metrics=["accuracy"])

cs1109_model.fit(train_cs1109_X, train_cs1109_y, epochs=40, batch_size=32, verbose=0)

test_loss, test_acc = cs1109_model.evaluate(test_cs1109_X, test_cs1109_y)
test_acc
```

```
// 0.6666666865348816
```

Feel free to edit the code, e.g. add or remove hidden layers, change the number of neurons in the hidden layers, change ReLU to sigmoid, change from RMSprop to another optimizer, change the learning rate, change the number of epochs, or change the batch size.

A Neural Network for Multiclass Classification

For multi-class classification, we might use a network with the following architecture:

- **Input layer:** one input per feature.
- **Hidden layers:** one or more hidden layers.
 - Activation function for neurons in hidden layers can be sigmoid or ReLU.
- **Output layer:** one output neuron per class.
 - Activation function for the output neurons should be the softmax function.

Example: Iris Dataset

Let's start with this:

- An input layer with 4 inputs (petal width and length, and sepal width and length).
- Two hidden layers, with 64 neurons in each, and ReLU activation function.
- An output layer with three neurons (one for Setosa, Versicolor and Virginica) and softmax activation function.

Example: Iris Dataset

```
iris = load_iris()

iris_train, iris_test, iris_train_y, iris_test_y = train_test_split(iris.data, iris.target, train_size=0.8, random_state=25)

iris_train[0], iris.target_names[iris_train_y[0]]
# iris_train.shape, iris_test.shape

((120, 4), (30, 4))
```

```
iris_model = keras.models.Sequential([
    keras.layers.InputLayer(input_shape=(4,)),
    keras.layers.Normalization(),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(3, activation="softmax")
])
```

Layer (type)	Output Shape	Param #
normalization_4 (Normalization)	(None, 4)	9
dense_12 (Dense)	(None, 64)	320
dense_13 (Dense)	(None, 64)	4160
dense_14 (Dense)	(None, 3)	195

Total params: 4684 (18.30 KB)
Trainable params: 4675 (18.26 KB)
Non-trainable params: 9 (40.00 Byte)

Example: Iris Dataset

```
hidden1 = iris_model.layers[1]
weights, biases = hidden1.get_weights()
weights.shape, biases.shape

((4, 64), (64,))

iris_model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=0.001),
    loss="sparse_categorical_crossentropy",
    metrics="accuracy"
)

history = iris_model.fit(iris_train, iris_train_y,
                        epochs=40,
                        batch_size=32,
                        validation_split=0.1,
                        verbose=0
                      )

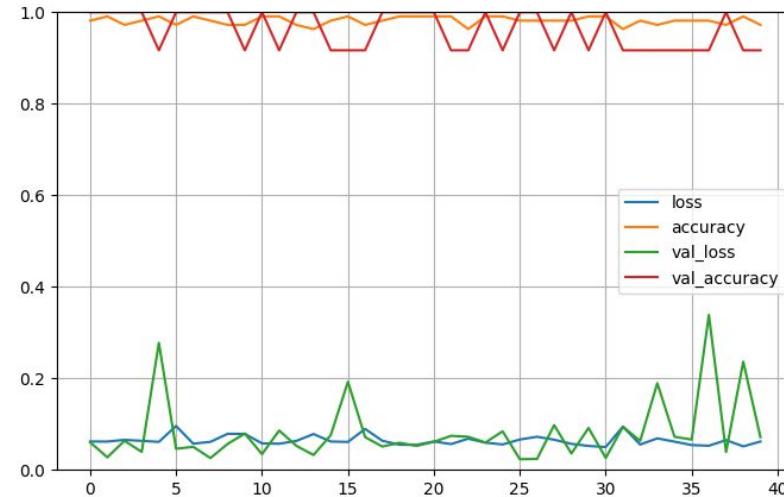
test_loss, test_accuracy = iris_model.evaluate(iris_test, iris_test_y)
test_accuracy

1/1 [=====] - 0s 52ms/step - loss: 0.0959 - accuracy: 0.9333
0.9333333373069763
```

Example: Iris Dataset

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



Example: Iris Dataset

Note the loss function above:

- `sparse_categorical_crossentropy` for multi-class classification when the classes are integers, e.g. 0 = one kind of Iris, 1 = another kind, 2 = a third kind (which is what we have in the Iris dataset).
- `categorical_crossentropy` for multi-class classification when the classes have been one-hot encoded.
- As we've seen, `binary_crossentropy` for binary classification, where the classes will be 0 or 1.

Below, an alternative, is code that illustrates one-hot encoding the target values using the Keras function called `to_categorical`, and then using `categorical_crossentropy` for the loss function.

Example: Iris Dataset

```
train_iris_y = to_categorical(train_iris_y)
test_iris_y = to_categorical(test_iris_y)

inputs = Input(shape=(4,))
x = Normalization()(inputs)
x = Dense(64, activation="relu")(x)
x = Dense(64, activation="relu")(x)
outputs = Dense(3, activation="softmax")(x)
iris_model = Model(inputs, outputs)

iris_model.compile(optimizer=RMSprop(learning_rate=0.001),
                    loss="categorical_crossentropy", metrics=["accuracy"])

iris_model.fit(train_iris_X, train_iris_y, epochs=40, batch_size=32, verbose=0)

test_loss, test_acc = iris_model.evaluate(test_iris_X, test_iris_y)
test_acc

// 0.8999999761581421
```

Example: Iris Dataset

Observations:

- Neural networks are often not the best-performing approaches for structured data.
- And, sure enough, the results here are not great. Of course, there is a lot we can tweak to see if we can improve the results.
- But, instead, let's switch to an image processing example.

Example: Fashion MNIST Dataset

Fashion MNIST is also a classic dataset for multi-class classification.

- The task is classification of fashion items.
 - Features: 28 pixel by 28 pixel grayscale images of fashion items. The values are integers in [0, 255].
 - Classes: ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"].



- Dataset: 70,000 images, so we can safely use holdout, and it is already partitioned:
 - 60,000 training images; 10,000 test images.

Example: Fashion MNIST Dataset

We don't really need scikit-learn pipelines this time.

But we do need to reshape:

- Our training data is in a 3D array of shape (60000, 28, 28).
- We change it to a 2D array of shape (60000, 28 * 28).
 - This 'flattens' the images.
 - When working with images, it is often better not to do this.
- Similarly, the test data.

Example: Fashion MNIST Dataset

We will do a three-layer network:

- One hidden layer with 300 neurons, using the ReLU activation function.
- Second hidden layer with 100 neurons, using the ReLU activation function.
- The output layer will have 10 neurons, one per class, and will use the softmax activation function.

The features (pixel values) are all in the same range $[0, 255]$, so we do not need to standardize using a Normalization layer.

But it is a bad idea to feed into a neural network values that are much larger than the initial weights, so we will rescale to by dividing by 255. We can do this using a **Rescaling** layer.

Example: Fashion MNIST Dataset

```
(X_train_full, Y_train_full), (X_test, Y_test) = keras.datasets.fashion_mnist.load_data()
X_train_full.shape, X_test.shape
((60000, 28, 28), (10000, 28, 28))

class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
class_names[Y_train_full[0]]

'Ankle boot'

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Rescaling(scale=1./255),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Layer (type)	Output Shape	Param #
=====		
flatten_1 (Flatten)	(None, 784)	0
rescaling_4 (Rescaling)	(None, 784)	0
dense_12 (Dense)	(None, 300)	235500
dense_13 (Dense)	(None, 100)	30100
dense_14 (Dense)	(None, 10)	1010
=====		
Total params: 266610 (1.02 MB)		
Trainable params: 266610 (1.02 MB)		
Non-trainable params: 0 (0.00 Byte)		

Example: Fashion MNIST Dataset

```
hidden1 = model.layers[2]
weights, biases = hidden1.get_weights()
weights.shape, biases.shape
```

```
((784, 300), (300,))
```

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="SGD",
              metrics=["accuracy"]
            )
```

```
history = model.fit(X_train_full, Y_train_full,
                     epochs=30,
                     validation_split=0.1  # by default shuffle is set to True
                   )
```

Example: Fashion MNIST Dataset

```
test_loss, test_accuracy = model.evaluate(X_test, Y_test)
test_accuracy

313/313 [=====] - 1s 3ms/step - loss: 0.3565 - accuracy: 0.8846
0.8845999836921692
```

```
y_proba = model.predict(X_test[0:5])
y_proba.round(2)

1/1 [=====] - 0s 34ms/step
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. ],
       [0. , 0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0.81, 0. , 0. , 0. , 0. , 0. , 0.19, 0. , 0. , 0. ]],
      dtype=float32)
```

```
import numpy as np

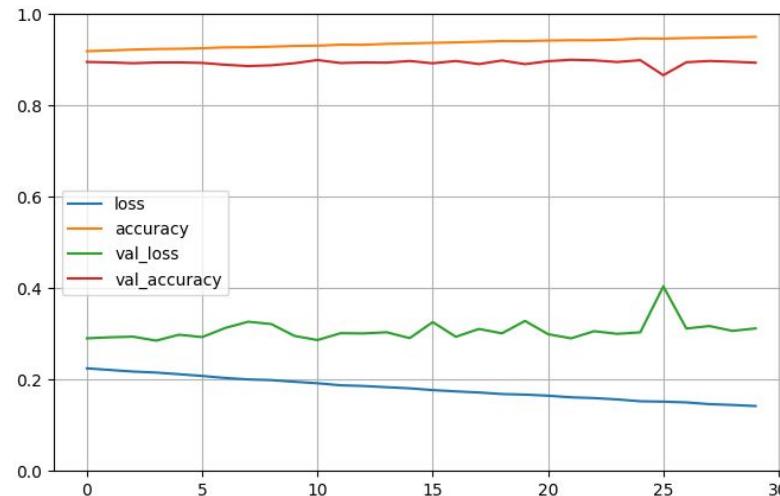
np.array(class_names)[np.argmax(model.predict(X_test[0:3]), axis=-1)]

1/1 [=====] - 0s 21ms/step
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

Example: Fashion MNIST Dataset

```
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



Remarks on Computer Vision Problems

In the 1960s, 70s, 80s and to some extent 90s, the typical pipeline for a computer vision (or image processing) system was as follows:

- There would be a module that would extract features from the images.
 - These features would have been carefully hand-designed.
 - They might include edges detected by some edge detection algorithm, for example. (If you are interested, look up SIFT or SURF or HOG.)
- Then these features would be fed into a typical learning algorithm, e.g. logistic regression.

Remarks on Computer Vision Problems

Notice how different life is now – when using neural networks.

- There's no extraction of hand-crafted features. We feed in the raw pixel values (or, lightly-processed pixel values, e.g. scaled values).
- It is the layers of the neural network that automatically discover the features, and the final layer that makes the classification.
- The dense layers are only one possibility.
 - Computer vision (image processing) more often also uses convolutional layers, pooling layers, batch normalization layers, and so on. We may study them in coming lectures.

Concluding Remarks

- A few decisions are constrained: number of inputs; number of output neurons; activation function of output neurons; and (to some extent) loss function.
- But there are numerous hyperparameters (and even more to come!)
 - Even making a good guess at them is more art than science, although this is changing.
 - On the other hand, grid search or randomized search will make things even slower than they already are – and we still have to specify some sensible values for them to search through.
- There is a considerable risk of overfitting.

Next lecture

Training Deep Neural Network

19th October 2023

IT496: Introduction to Data Mining



Lecture 26

The Backpropagation Algorithm

(Slides are created from the lecture notes of Dr. Derek Bridge, UCC, Ireland)

Arpit Rana
19th October 2023

(Batch) Gradient Descent

Let's start by reminding ourselves of (Batch) Gradient Descent for OLS regression

```
initialize  $\beta$  randomly
repeat until convergence
   $\circ \quad \beta \leftarrow \beta - \frac{\alpha}{m} X^T(X\beta - y)$ 
```

- We see it making predictions $\hat{y} = X\beta$ for all the examples X and comparing them with target values, y .
- And we see it updating all the parameters β by an amount equal to the negative of the gradients, multiplied by the learning rate α .

Why Not Gradient Descent?

For neural networks, however, there is a problem.

- At the output layer, we can straightforwardly compute loss: we can get the network's output (its prediction) and we have the target value, because the training set is a labeled dataset.
- But we cannot straightforwardly compute loss at the hidden layers: we know what outputs their neurons produce but we do not know what they should produce (target values). The labeled dataset doesn't tell us the target values for hidden layer neurons.

The solution is to assume that each neuron in layer l is responsible for some part of the loss of the neurons it is connected to in layer $l + 1$.

We will refer to this amount as the '**error signal**'.

Backpropagation: Basic Idea

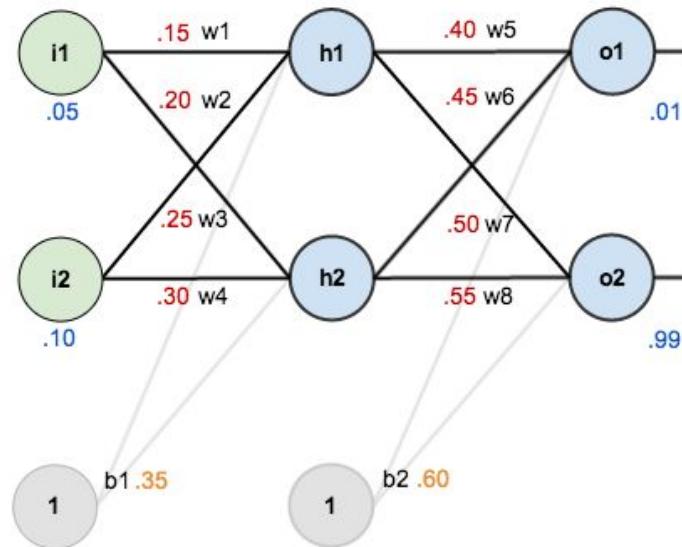
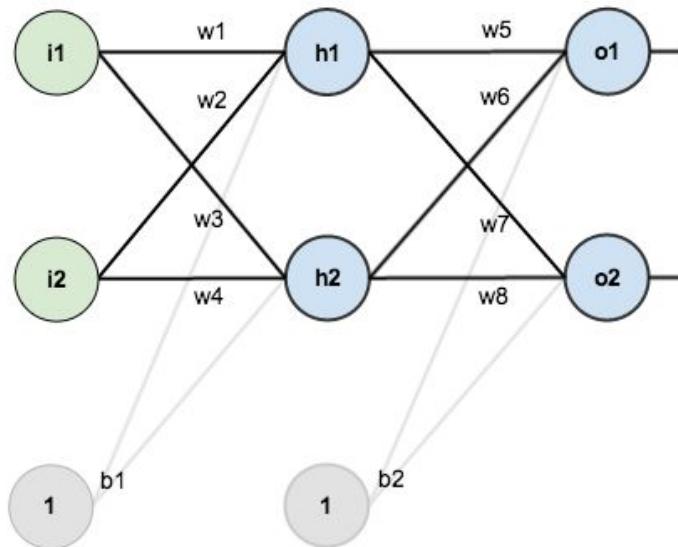
- This leads to the idea of an algorithm that makes two passes through the network:
 - a ***forward pass*** to make predictions: we feed \mathbf{X} into the network and then work forwards through the network, computing activations layer by layer until we reach the output layer;
 - a ***backward pass*** to compute the gradients: we calculate loss at the output layer and then work backwards through the network computing error signals layer by layer until we reach the input layer.
- With these two steps completed, we have all the gradients, so we can update all the weights and biases.

This description helps you see why the calculation of the gradients (and sometimes the entire algorithm) is referred to as **backpropagation** (or just **backprop**).

The Backpropagation Algorithm (High Level)

- **Random initialization:** initialize all weights and biases randomly
- **Forward propagation:** make predictions for all the training examples:
 - Layer by layer from layer 1 to layer L:
 - Calculate the inputs to the units in that layer (weighted sums plus biases)
 - Calculate the outputs of the units in that layer (using activation function)
- **Backpropagation:**
 - Calculate the error signals Δ at layer L
 - Layer by layer in reverse from layer L-1 to layer 1:
 - Calculate the error signals Δ for the units in that layer
- **Update all the weights and biases:** $w_{i,j}^{(l)} = w_{i,j}^{(l)} - \alpha \times a_i \times \Delta_j^{(l)}$
 $b_j^{(l)} = b_j^{(l)} - \alpha \times 1 \times \Delta_j^{(l)}$

The Backpropagation Algorithm (Example)



Here are the initial weights, the biases, and training inputs/outputs:

The Backpropagation Algorithm (Example): Forward Pass

Here's how we calculate the total net input for h_1 :

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

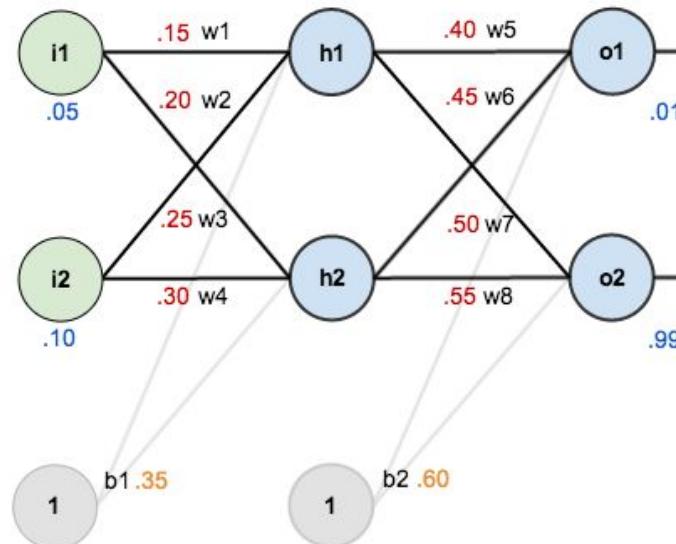
$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

We then squash it using the logistic function to get the output of h_1 :

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

Carrying out the same process for h_2 we get:

$$out_{h2} = 0.596884378$$



The Backpropagation Algorithm (Example): Forward Pass

Similarly, for the output layer neurons, using the output from the hidden layer neurons as inputs. Here's the output for o_1 :

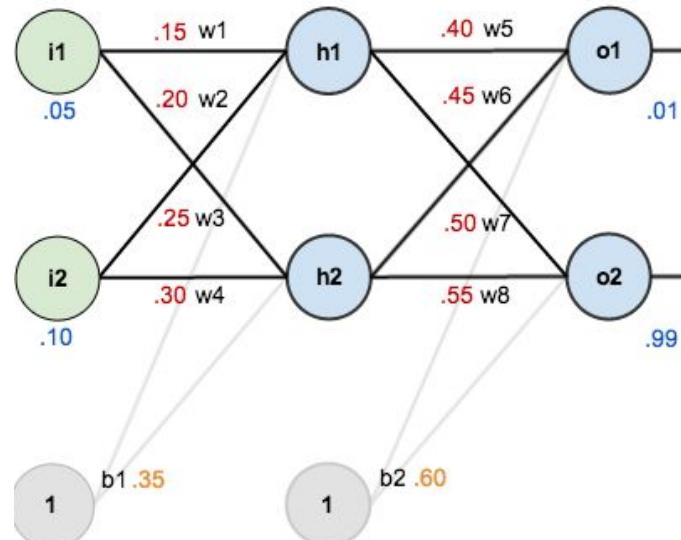
$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\begin{aligned} net_{o1} &= 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 \\ &= 1.105905967 \end{aligned}$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

Carrying out the same process for o_2 we get:

$$out_{o2} = 0.772928465$$



The Backpropagation Algorithm (Example): Forward Pass

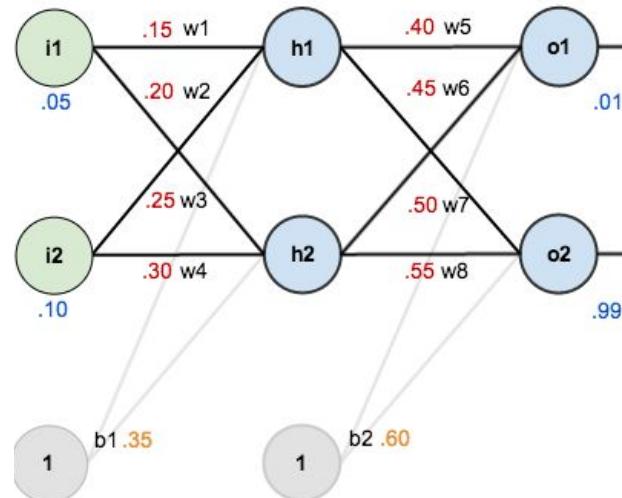
We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

$$E_{o2} = 0.023560026$$

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

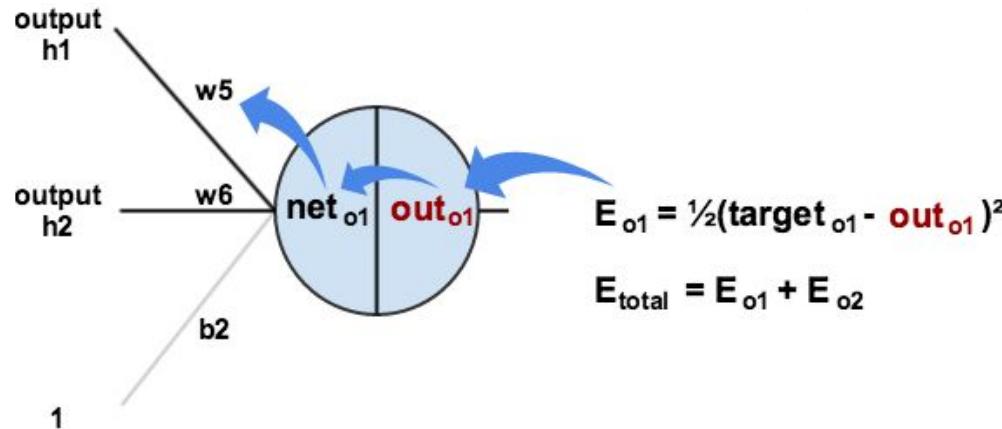


The Backpropagation Algorithm (Example): Backward Pass

Output Layer:

Consider w_5 . We want to know how much a change in w_5 affects the *total error*. By applying the *chain rule*, we know that -

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$



The Backpropagation Algorithm (Example): Backward Pass

First, how much does the total error change with respect to the output?

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

Next, how much does the output of o_1 change with respect to its total *net* input?

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

The Backpropagation Algorithm (Example): Backward Pass

Finally, how much does the total net input of o_1 change with respect to w_5 ?

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

We can repeat this process to get the new weights w_6 , w_7 , and w_8 :

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

So, we update w_5 as per the gradient update rule:

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

What about updating the bias b_2 ?

The Backpropagation Algorithm (Example): Backward Pass

Hidden Layer:

Next, we'll continue the backwards pass by calculating new values for w_1 , w_2 , w_3 , and w_4 .

$$\begin{aligned}\frac{\partial E_{total}}{\partial w_1} &= 0.036350306 * 0.241300709 * 0.05 \\ &= 0.000438568\end{aligned}$$

$$w_1^+ = 0.149780716$$

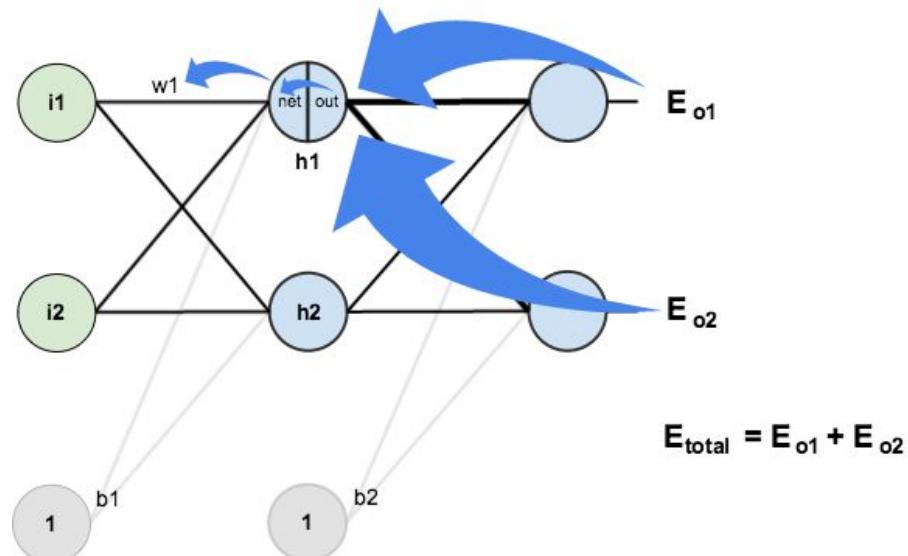
$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



The Backpropagation Algorithm (Example)

- When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109. After this first round of backpropagation, the total error is now down to 0.291027924.
- After repeating this process 10,000 times, for example, the error plummets to 0.0000351085.
- At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

Auto-differentiation (Autodiff)

- We have shown the update rules for a particular network and a particular loss function.
- We would get different update rules if we change:
 - the network, e.g. layers other than dense layers (batch normalization layers, convolutional layers, etc.); and/or
 - the loss function.
- Happily, we don't have to manually find the partial derivatives all over again.
- Neural networks consist of layers of operations, each with simple, known derivatives.

Auto-differentiation (Autodiff)

- Given that the network simply defines a composite function (see previous lecture), the derivatives for the whole network can be obtained automatically by repeated use of the **chain rule**:
 - To differentiate a function of a function, $y = f(g(x))$, let $u = g(x)$ so that $y = f(u)$, then

$$\frac{dy}{dx} = \frac{dy}{du} \times \frac{du}{dx}$$

- Hence, modern frameworks such as TensorFlow can compute gradients automatically in the backpropagation step.
- This is known as *autodiff* (or, for the way it is used by backprop, *reverse mode autodiff*).

The Backpropagation Algorithm: Vectorized Form

- **Random initialization:** initialize all weights randomly
- **Forward propagation:**
 - Calculate and store $\mathbf{Z}^{(1)} = \mathbf{X}\mathbf{W}^{(1)}$
 - Layer by layer from layer $l = 1$ to layer L :
 - Calculate and store $\mathbf{A}^{(l)} = g^{(l)}(\mathbf{Z}^{(l)})$
 - Calculate $\mathbf{Z}^{(l+1)} = \mathbf{A}^{(l)}\mathbf{W}^{(l+1)}$
 - Calculate and store $\mathbf{G}^{(l)} = g'(\mathbf{Z}^{(l)})^T$
- **Backpropagation:**
 - Calculate $\mathbf{D}^{(L)} = (\mathbf{A}^{(L)} - \mathbf{Y})^T$
 - Layer by layer in reverse from layer $l = L - 1$ to layer 1:
 - Calculate and store $\mathbf{D}^{(l)} = \mathbf{G}^{(l)} * \mathbf{W}^{(l)}\mathbf{D}^{(l+1)}$
- **Update all the weights:**
 - $\mathbf{W}^{(1)} = \mathbf{W}^{(1)} - \alpha(\mathbf{D}^{(1)}\mathbf{X})^T$
 - $\mathbf{W}^{(l)} = \mathbf{W}^{(l)} - \alpha(\mathbf{D}^{(l)}\mathbf{A}^{(l-1)})^T$ for all other values of l
- For simplicity, biases are omitted.
- We can see in this more precise version that some of the things that are calculated on the forward pass get stored.
 - These things can be used on the backward pass.
- Similarly, some of the things that are calculated on the backward pass get stored.
 - These things can then be used to update the weights.
- This makes backprop much more efficient than it otherwise would be.

Next lecture

Training Deep Neural Network

20th October 2023

IT496: Introduction to Data Mining



Lecture 27-28

Training DNN: Vanishing Gradient Problem

(Slides are created from the lecture notes of Dr. Derek Bridge, UCC, Ireland)

Arpit Rana
20th October 2023

Problems in Training Deep Neural Network

- ***Vanishing Gradient Problem***

The gradients grow smaller and smaller when flowing backward through the DNN while training.

- ***Need enough (labelled) training data***

Large network demands huge amount of data or it might be too costly to label.

- ***Training may be extremely slow***

Training very large DNN using (batch) gradient descent can be painfully slow.

- ***Overfitting***

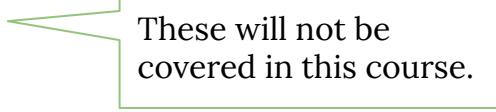
A model with millions of parameters would severely risk overfitting the training data.

Problems and Solutions in Training Deep Neural Network

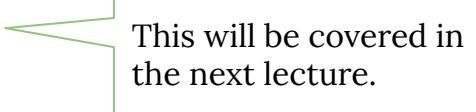
- ***Vanishing Gradient Problem***
 - Better Initialization
 - Non-saturating Activation Functions
 - Batch Normalization
- ***Need enough (labelled) training data***
 - Reusing pretrained layers (transfer learning)
 - Unsupervised pre-training
 - Pre-training on an auxiliary task
- ***Training may be extremely slow***
 - Using faster optimizers
 - Learning rate scheduling
- ***Overfitting***
 - Reducing the network size
 - Weight regularization
 - Dropout
 - Early stopping

Problems and Solutions in Training Deep Neural Network

- **Vanishing Gradient Problem**
 - Better Initialization
 - Non-saturating Activation Functions
 - Batch Normalization
- **Need enough (labelled) training data**
 - Reusing pretrained layers (transfer learning)
 - Unsupervised pre-training
 - Pre-training on an auxiliary task
- **Training may be extremely slow**
 - Using faster optimizers
 - Learning rate scheduling
- **Overfitting**
 - Reducing the network size
 - Weight regularization
 - Dropout
 - Early stopping



These will not be covered in this course.



This will be covered in the next lecture.

Vanishing Gradient Problem

Activation Function: sigmoid (logistic function)

Weight Initialization Technique: random initialization (using a normal distribution with a mean 0 and a standard deviation of 1).

X. Glorot and Y. Bengio found that this combination leads to the following.

- The variance of the outputs of each layer is much greater than the variance of its inputs.
- The variance keeps increasing after each layer until the activation function saturates at the top layers.

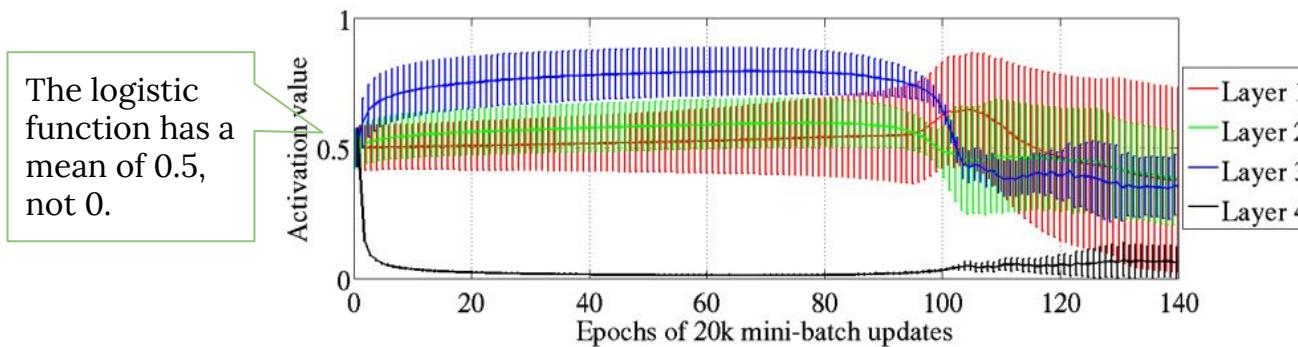
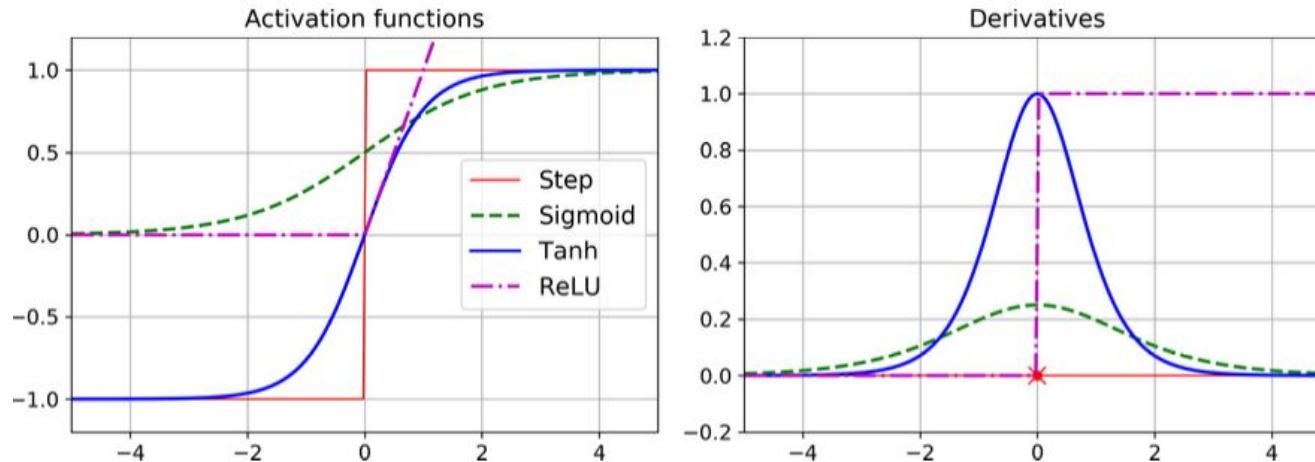


Image Source: "Understanding the Difficulty of Training Deep Feedforward Neural Networks," X. Glorot, Y. Bengio (2010).

Vanishing Gradient Problem

Looking at the logistic function and its derivative plots below.

- When inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0.



- Thus, during backpropagation, there's not much gradient to propagate back to earlier layers (and what little gradient there is, gets diluted as it propagates back).

Vanishing Gradient Problem

- Each weight is updated by an amount proportional to the gradient of the loss function with respect to that weight.
 - But if the gradient is very small, the weight doesn't change much.
 - This may prevent the network from converging to a good approximation of the target function.
 - We can now see that this is worse for deeper networks.
 - The error signal becomes ever smaller as it is back propagated.
- We look at the following solutions:
 - Better weight initialization;
 - Non-saturating activation functions;
 - Batch normalization.

Weight Initialization

Glorot and Bengio (2010) argued that -

- the variance of the outputs of each layer should be equal to the variance of its inputs (going forward), and
- the gradients should have equal variance before and after flowing through a layer (going backward).

Both the conditions are not guaranteed unless the number of input units to a layer (fan_{in}) equals the number of neurons, i.e., number of output units (fan_{out})

Weight Initialization

- Glorot (Xavier) Initialization: connection weights must be initialized randomly while using the logistic activation function -
 - Normal distribution with mean 0 and variance $\sigma^2 = 1 / fan_{avg}$
 - Or a uniform distribution between $-r$ and $+r$, with $r = \sqrt{3 * fan_{avg}}$
where, $fan_{avg} = (fan_{in} + fan_{out}) / 2$
- Other similar strategies for different activation functions were proposed -

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Weight Initialization in Keras

- By default, Keras uses Glorot initialization with a *uniform* distribution.
- You can change this to He initialization by setting `kernel_initializer="he_uniform"` or `kernel_initializer="he_normal"` when creating a layer, like this:

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Non-Saturating Activation Functions

Certain activation functions, including the sigmoid function, are one cause of the vanishing gradient problem.

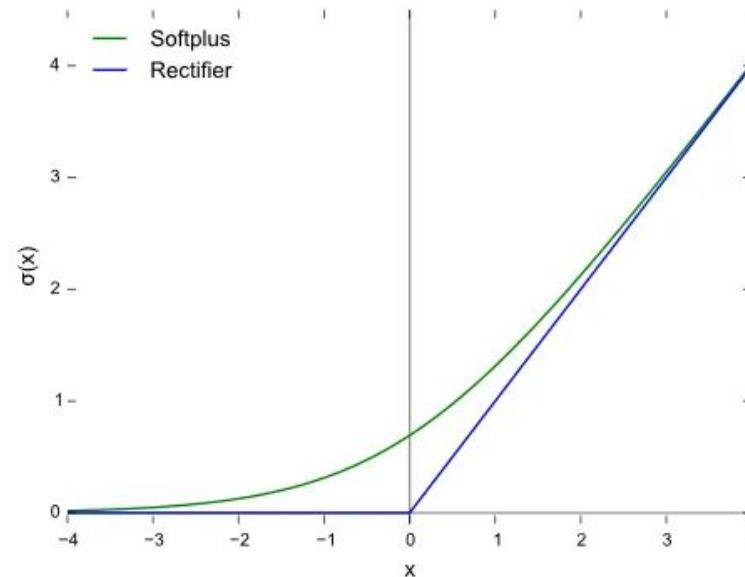
- When the input to this function becomes large (positive or negative), the function saturates (i.e. becomes very flat).
 - Even when the gradient is at its greatest (when input z is 0 and $\sigma(z) = 0.5$), it is only 0.25 (gradient = $\sigma(z)(1 - \sigma(z))$).
 - So in the back propagation, gradients always diminish by a quarter or more.
- This is why we rarely use the sigmoid function as the activation function in the hidden layers of deep networks.

Non-Saturating Activation Functions

- Lots of alternatives have been proposed, including the ReLU (*rectified linear unit*) activation function,
- ReLU does not saturate for positive values and it is quite fast to compute.

$$ReLU(z) = \max(0, z)$$

$$Softplus(z) = \log(1 + e^z)$$



ReLU and its smooth variant, *softplus*.

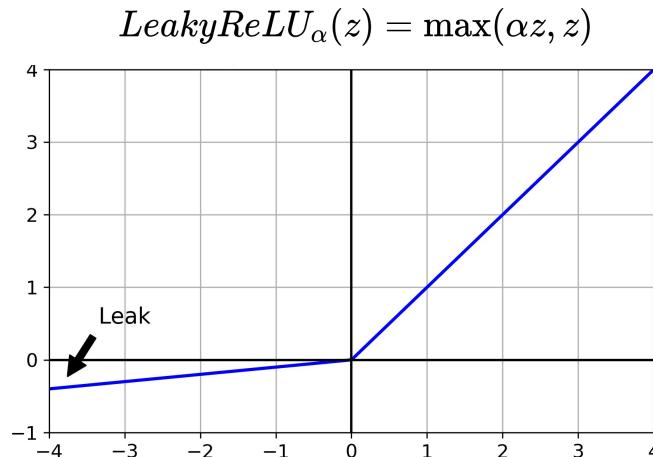
Non-Saturating Activation Functions

- Neurons that use the ReLU activation function have obvious problems too:
 - If their input (weighted sum) is negative, the output is zero; and the gradient is zero; and
 - if this is true for all examples in the training set then, in effect, the neuron dies. This is known as “*dying ReLU*” problem.
 - The gradient changes abruptly at $z = 0$, which can make Gradient Descent bounce around.
- Despite its problems, it remains a popular choice.

Non-Saturating Activation Functions

Leaky ReLU have been proposed having at least some non-zero gradient for negative inputs (alternatives to ReLU are slower to compute and they introduce further hyperparameters.)

The hyperparameter α defines how much the function “leaks”: it is the slope of the function for $z < 0$, and is typically set to 0.01.



```
layers.Dense(7 * 7 * 128),  
layers.LeakyReLU(alpha=0.2),
```

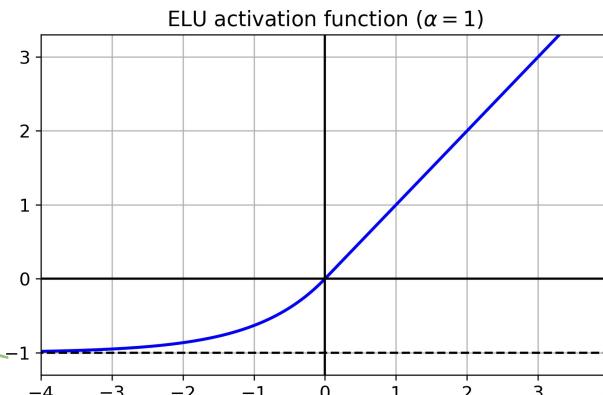
Instead of being a hyperparameter, α becomes a parameter that can be modified by backpropagation like any other parameter, this version of ReLU is known as *Parametric Leaky ReLU (PReLU)*.

Non-Saturating Activation Functions

ELU (Exponential Linear Unit)

- It helps alleviate the vanishing gradient problem.
- It has a non-zero gradient for $z < 0$, which avoids the dead neurons problem.
- If $\alpha = 1$, then the function is smooth everywhere, which helps speed up gradient descent (does not bounce much left and right of $z = 0$.)

$$ELU_{\alpha}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$



The hyperparameter α defines the value that the ELU function approaches when z is a large negative number.

Non-Saturating Activation Functions

SELU (Scaled Exponential Linear Unit)

- If we build a neural network with a stack of dense layers, and all the hidden layers use SELU activation, then the network will “self-normalize” (the output of each layer will tend to preserve mean 0 and standard deviation 1 during training).
 - The input features must be standardized (mean 0 and standard deviation 1).
 - Every hidden layer weights must also be initialized using LeCun normal initialization. In Keras, this means setting `kernel_initializer="lecun_normal"`.

$$SELU(z) = \begin{cases} \lambda\alpha(\exp(z) - 1) & \text{if } z < 0 \\ \lambda z & \text{if } z \geq 0 \end{cases} \quad [\alpha (>1) \text{ and } \lambda (>1) \text{ are predefined constants.}]$$

Non-Saturating Activation Functions

Which activation function should we use for the hidden layers of our deep neural network?

- In general, **SELU > ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic**
- If the network's architecture prevents it from self-normalizing, then ELU may perform better than SELU (since SELU is not smooth at $z = 0$).
- If you care a lot about runtime latency, then you may prefer leaky ReLU. If you don't want to tweak yet another hyperparameter, you may just use the default α values used by Keras (e.g., 0.3 for the leaky ReLU).
- If you have spare time and computing power, you can use cross-validation to evaluate other activation functions, in particular PReLU if you have a huge training set.

Batch Normalization

We previously studied the usefulness of feature scaling when doing Gradient Descent for, e.g., linear regression.

... and we've been doing this to the features in our neural networks too.

- But, if this is a good idea for the inputs to the first hidden layer, why not use the same idea for the inputs to subsequent layers?
- In other words, we normalize the activations (outputs) of layer l prior to them being used as inputs to layer $l + 1$.
- This will control the distribution of the values throughout the training process.
- This, in essence, is the idea of **batch normalization**.

Batch Normalization

In summary, for a given layer, it standardizes the outputs of the neurons (subtract the mean, divide by the standard deviation), **then it scales the result and adds an offset**.

Batch Normalization Algorithm:

$$1. \quad \boldsymbol{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

$$2. \quad \sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \boldsymbol{\mu}_B)^2$$

$$3. \quad \hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$4. \quad \mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$$

During training, four parameter vectors are learned in each batch-normalized layer:

- γ (the output scale vector) and β (the output offset vector) are learned through regular backpropagation, and
- μ (the final input mean vector), and σ (the final input standard deviation vector) are estimated using an exponential moving average.

What about the test time?

Batch Normalization

- Batch Normalization reduces the vanishing gradients problem so much, we can even use saturating activation functions.
- Training becomes less sensitive to the method used for randomly initializing weights.
- Much larger learning rates work (faster convergence) with less risk of divergence.
- It acts like a regularizer.

Batch Normalization in Keras

- Just add another layer!
- For example,
 - `x = Dense(512, activation="relu") (x)`
 - `x = BatchNormalization() (x)`
- This is how we will do batch normalization in this course.
- Ignore: in fact, there is some debate about whether we should batch normalize the activations of the layer (as above) or the weighted sum, before applying the activation function (as below):
 - `x = Dense(512, activation="linear", use_bias=False) (x)`
 - `x = BatchNormalization() (x)`
 - `x = Activation("relu") (x)`
- We will stick with the former, which is more concise.

Batch Normalization in Keras

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

Layer (type)	Output Shape	Param #
=====		
flatten_3 (Flatten)	(None, 784)	0
batch_normalization_v2 (Batch Normalization)	(None, 784)	3136
dense_50 (Dense)	(None, 300)	235500
batch_normalization_v2_1 (Batch Normalization)	(None, 300)	1200
dense_51 (Dense)	(None, 100)	30100
batch_normalization_v2_2 (Batch Normalization)	(None, 100)	400
dense_52 (Dense)	(None, 10)	1010
=====		
Total params: 271,346		
Trainable params: 268,978		
Non-trainable params: 2,368		

Exploring Vanishing Gradient with MNIST

```
def build_mnist_network(activation, initializer, use_batch_norm):
    inputs = Input(shape=(28 * 28,))
    x = Rescaling(scale=1./255)(inputs)
    x = Dense(512, activation=activation, kernel_initializer=initializer)(x)
    if use_batch_norm:
        x = BatchNormalization()(x)
    outputs = Dense(10, activation="softmax", kernel_initializer=initializer)(x)
    model = Model(inputs, outputs)
    model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return model

networks = [
    build_mnist_network("sigmoid", "random_normal", False),
    build_mnist_network("sigmoid", "random_normal", True),
    build_mnist_network("sigmoid", "glorot_uniform", False),
    build_mnist_network("sigmoid", "glorot_uniform", True),
    build_mnist_network("relu", "random_normal", False),
    build_mnist_network("relu", "random_normal", True),
    build_mnist_network("relu", "glorot_uniform", False),
    build_mnist_network("relu", "glorot_uniform", True)
]

for network in networks:
    network.fit(mnist_x_train, mnist_y_train, epochs=10, batch_size=32, verbose=0)
    test_loss, test_acc = network.evaluate(mnist_x_test, mnist_y_test, verbose=0)
    print(test_acc)
```

Output:

0.9451000094413757
0.9549999833106995
0.9435999989509583
0.9496999979019165
0.9746999740600586
0.9814000129699707
0.9746999740600586
0.9840999841690063

Hyperparameter Tuning

- First an observation about validation sets:
 - When using neural networks, we typically have a large dataset.
 - Hence, we use holdout to split the dataset into train, validation and test sets.
 - If you have a smaller dataset, where you can afford to split off a test set, but you cannot afford to split off a validation set, then you might want to use k-fold cross-validation, as we did in our scikit-learn examples.
 - Keras does not have any in-built cross-validation functions. You'd have to write your own.

Hyperparameter Tuning

- Second, an observation about hyperparameter tuning:
 - Hyperparameter tuning involves training lots of different models with different values for the hyperparameters, and comparing their performance on the validation data.
 - This is often a problem with neural networks: training can be slow (due to large datasets and models that have many parameters) and so training lots of different models for hyperparameter tuning is very time-consuming.
 - People often don't bother! They just pick hyperparameter values out of their heads, or accept the Keras defaults.
 - If you can afford to be more systematic, there is a separate library (which you would need to install) called KerasTuner. It automates hyperparameter tuning, similar to what we saw in scikit-learn.

Next lecture

Overfitting in Neural Network

20th October 2023

IT496: Introduction to Data Mining



Lecture 29-30

Overfitting in Neural Networks

(Slides are created from the lecture notes of Dr. Derek Bridge, UCC, Ireland)

Arpit Rana
26th - 27th October 2023

Problems and Solutions in Training Deep Neural Network

- ***Vanishing Gradient Problem***

- Better Initialization
- Non-saturating Activation Functions
- Batch Normalization



- ***Need enough (labelled) training data***

- Reusing pretrained layers (transfer learning)
- Unsupervised pre-training
- Pre-training on an auxiliary task

- ***Training may be extremely slow***

- Using faster optimizers
- Learning rate scheduling

- ***Overfitting***

- Reducing the network size
- Weight/Max-norm regularization
- Dropout
- Early stopping



We will not cover these topics in detail.

Overfitting

DNNs typically have tens of thousands of parameters, sometimes even millions.

- This increases their (model) capacity: ability to fit a huge variety of complex datasets.
- This also makes the network prone to overfitting the training set.

Overfitting

Reminder. If your model overfits, your main options are:

- gather more training examples;
- remove noise in the training examples;
- change model: move to a less complex model;
- simplify by reducing the number of features;
- stick with your existing model but add constraints (if you can) to reduce its complexity.

Overfitting in Neural Networks

We will look at the following ways:

- reducing the network's size – an example of moving to a less complex model;
- weight regularization – an example of adding constraints to reduce complexity;
- dropout – an example of adding constraints to reduce complexity;
- max-norm regularization – again an example of adding constraints to reduce complexity; and
- early stopping – a somewhat different way of avoiding overfitting.

A Network that Overfits a Little

```
# A network that overfits (a little!)

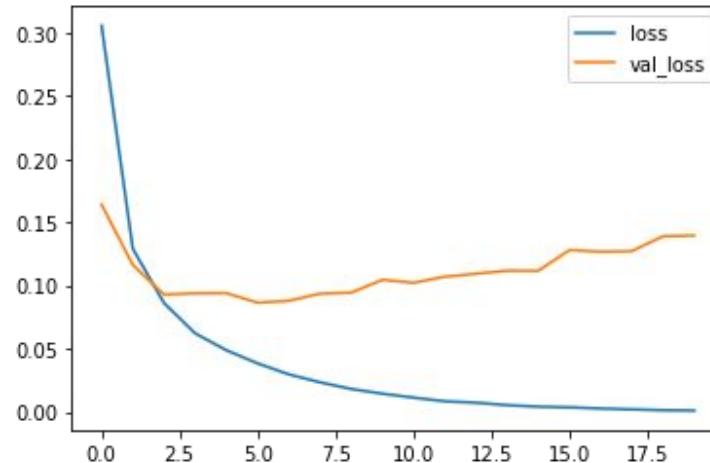
inputs = Input(shape=(28 * 28,))
x = Rescaling(scale=1./255)(inputs)
x = Dense(1024, activation="relu")(x)
x = Dense(1024, activation="relu")(x)
outputs = Dense(10, activation="softmax")(x)
overfitting_model = Model(inputs, outputs)
overfitting_model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="sparse_categorical_crossentropy")

history = overfitting_model.fit(mnist_x_train, mnist_y_train, epochs=20, batch_size=32,
                                 verbose=0, validation_split=0.20)
history.history["loss"][-1], history.history["val_loss"][-1]
```

(0.0010047738905996084, 0.13949735462665558)

A Network that Overfits a Little

```
pd.DataFrame(history.history).plot()
```



Reducing Network Size

- We can make the model (neural network) less complex by reducing the number of parameters.
- Obviously enough, this is achieved by:
 - reducing the number of hidden layers, and/or
 - reducing the number of neurons within the hidden layers.

Reducing Network Size

```
# Smaller network

inputs = Input(shape=(28 * 28,))
x = Rescaling(scale=1./255)(inputs)
x = Dense(256, activation="relu")(x)
outputs = Dense(10, activation="softmax")(x)
smaller_model = Model(inputs, outputs)
smaller_model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="sparse_categorical_crossentropy")

history = smaller_model.fit(mnist_x_train, mnist_y_train, epochs=20, batch_size=32, verbose=0, validation_split=0.2)
history.history["loss"][-1], history.history["val_loss"][-1]
```

(0.06080113351345062, 0.09995315223932266)

Weight Regularization

- For linear regression, we used regularization to ensure that the coefficients β took only small values by penalizing large values in the loss function.
 - Ridge: we penalized by the l_2 -norm (the sum of their squares).
 - Lasso: we penalized by the l_1 -norm (the sum of their absolute values).
 - Elastic Net: we penalized by the mix of both of the above.
 - A hyperparameter λ , called the 'regularization parameter' controlled the balance between fitting the data versus shrinking the parameters.

Weight Regularization

- Weight Regularization in neural networks is the same idea, but applied to the weights in the layers (not their biases) of a network.

```
layer = keras.layers.Dense(100, activation="elu",
                          kernel_initializer="he_normal",
                          kernel_regularizer=keras.regularizers.l2(0.01))
```

- The `l2()` function returns a regularizer that will be called to compute the regularization loss, at each step during training. This regularization loss is then added to the final loss.
- You can use `keras.regularizers.l1()` for l_1 regularization and `keras.regularizers.l1_l2()` for both l_1 and l_2 regularization.

Weight Regularization

- In case of applying the same activation function, initialization strategy, and the same regularizer to all layers, you can use Python's `functools.partial()` function.
- It lets you create a thin wrapper for any callable, with some default argument values.

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

Weight Regularization

```
# Regularized network

inputs = Input(shape=(28 * 28,))
x = Rescaling(scale=1./255)(inputs)
x = Dense(1024, activation="relu", kernel_regularizer=l2(0.0001))(x)
x = Dense(1024, activation="relu", kernel_regularizer=l2(0.0001))(x)
outputs = Dense(10, activation="softmax", kernel_regularizer=l2(0.0001))(x)
regularized_model = Model(inputs, outputs)
regularized_model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="sparse_categorical_crossentropy")

history = regularized_model.fit(mnist_x_train, mnist_y_train, epochs=20, batch_size=32, verbose=0, validation_split=0.2)
history.history["loss"][-1], history.history["val_loss"][-1]
```

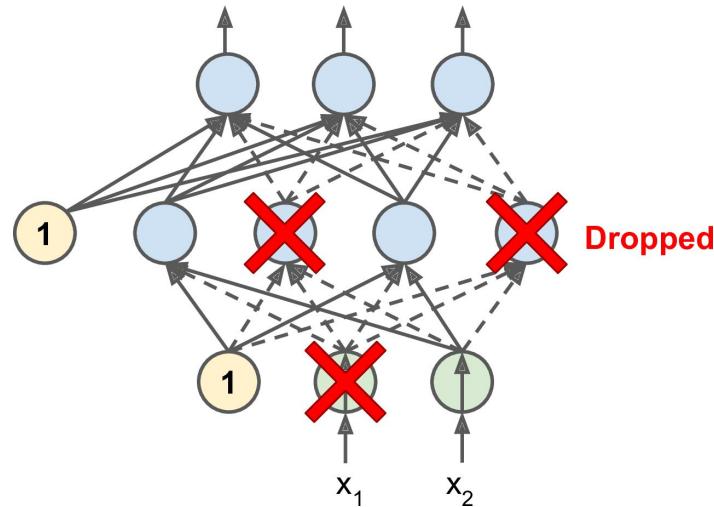
(0.056435953825712204, 0.11559036374092102)

- Weight regularization can work well when the network is small but has little effect on larger networks.
- For larger networks, a better option can be **dropout**.

Dropout

Imagine we have a layer that uses dropout with dropout rate p , e.g., $p = 0.5$

Then, in a given step of the backprop algorithm, each neuron in the layer (any layer except output layer) has probability p of being ignored – treated as if it were not there.



One Way of Doing Dropout

- Training. For any given mini-batch: // dropout rate is p .
 - In the forward propagation,
 - decide which neurons will be dropped (chosen with probability p);
 - set the activations of the dropped neurons to zero;
 - multiply the activations of the kept neurons by $1/(1 - p)$.
 - In the backpropagation, ignore the dropped out neurons.
- Note that different neurons will get dropped for each mini-batch.
- Testing. The Dropout layer does nothing at all, it just passes the inputs to the next layer.

One Way of Doing Dropout

- But why did we multiply activations by $1/(1 - p)$?
 - In testing, for $p = 0.5$ a neuron in the next layer will receive input from on average twice as many neurons as it did in training.
 - The multiplication by $1/(1 - p)$ compensates for this. This is called *keep probability*.

Why Does Dropout Reduce Overfitting?

- Consider a company whose employees were told to toss a coin every morning to decide whether to go to work or not.
 - The organization wants its employees must become more like generalists, less like specialists.
 - The organization would need to become more resilient. It could not rely on any one employee to perform critical tasks: the expertise would need to be spread across many employees.

Similarly, in dropout layers, neurons learn more robust features.

Why Does Dropout Reduce Overfitting?

- Another way to think about it.
 - Since a neuron can be present or absent, it's like training on a different neural network at each step.
 - There is a total of 2^N possible networks (where N is the total number of droppable neurons).
 - The final result is a bit like an ensemble of these many different virtual neural networks.

However, it typically increases the number of epochs needed for convergence (roughly double when $p = 0.5$).

Dropout in Keras

```
# Network with dropout

inputs = Input(shape=(28 * 28,))
x = Rescaling(scale=1./255)(inputs)
x = Dense(1024, activation="relu")(x)
x = Dropout(0.5)(x)
x = Dense(1024, activation="relu")(x)
x = Dropout(0.5)(x)
outputs = Dense(10, activation="softmax")(x)
model_with_dropout = Model(inputs, outputs)
model_with_dropout.compile(optimizer=RMSprop(learning_rate=0.0001), loss="sparse_categorical_crossentropy")

history = model_with_dropout.fit(mnist_x_train, mnist_y_train, epochs=20, batch_size=32, verbose=0, validation_split=0.2)
history.history["loss"][-1], history.history["val_loss"][-1]
```

(0.05647280439734459, 0.11740138381719589)

Monte Carlo (MC) Dropout

Yarin Gal and Zoubin Ghahramani (2016) introduced a powerful technique called MC Dropout:

- MC Dropout is the *use of dropout at inference time (forward passes only)* in order to add stochasticity to a network that can be used to generate a cohort of predictors/predictions to perform statistical analysis.
 - This can boost the performance of any trained dropout model, without having to retrain it or even modify it at all!
 - It also provides a much better measure of the model's uncertainty, and
 - It is also amazingly simple to implement.

Monte Carlo (MC) Dropout

Look at the following code that fully implements the MC *dropout*:

```
y_probas = np.stack([model(X_test_scaled, training=True)
                      for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

- The `model(X)` is similar to `model.predict(X)` except it returns a tensor rather than a Numpy array and it supports the `training` argument.
- So, we make 100 predictions over the test set and we stack them. As the dropout is on (`training=true`), all predictions will be different.
- The shape of `y_probas` will be [100, 10000, 10]. We average over the first dimension (`axis=0`), and get `y_proba`, an array of shape [10000, 10].

Monte Carlo (MC) Dropout

- Let's look at the model's prediction for the first instance in the test set, with dropout off:

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.99]],  
      dtype=float32)
```

- Compare this with the predictions made when dropout is activated:

```
>>> np.round(y_probas[:, :1], 2)
array([[[0. , 0. , 0. , 0. , 0. , 0.14, 0. , 0.17, 0. , 0.68]],  
      [[0. , 0. , 0. , 0. , 0. , 0.16, 0. , 0.2 , 0. , 0.64]],  
      [[0. , 0. , 0. , 0. , 0. , 0.02, 0. , 0.01, 0. , 0.97]],  
      [...]]  
  
>>> np.round(y_proba[:1], 2)
array([[0. , 0. , 0. , 0. , 0. , 0.22, 0. , 0.16, 0. , 0.62]],  
      dtype=float32)
```

- Looking at the standard deviation of the probability estimates.

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:1], 2)
array([[0. , 0. , 0. , 0. , 0. , 0.28, 0. , 0.21, 0.02, 0.32]],  
      dtype=float32)
```

Monte Carlo (MC) Dropout

If our model contains special layers during training (e.g., BatchNormalization), we should not force training mode as earlier.

- Instead, we create the subclass of the Dropout layer and override its `call()` method to force its `training` argument to True.

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

- We then use our new MCDropout layer in place of Dropout when defining our Sequential model.

Note: The number of Monte Carlo samples that we use (e.g., 100 in our previous example) is a hyperparameter: higher value gives more accurate predictions, however, increases the inference time.

Max-Norm Regularization

- For each neuron, it constrains the weights w of the incoming connections such that $\|w\|_2 \leq r$, where r is the max-norm hyperparameter and $\|\cdot\|_2$ is the l_2 norm.
- It does not add a regularization loss term to the overall loss function.
 - Instead, it is typically implemented by computing $\|w\|_2$ after each training step and rescaling w if needed,

$$w = \frac{wr}{\|w\|_2}$$

- Reducing r increases the amount of regularization and helps reduce overfitting. It can also help alleviate the unstable gradients (in absence of Batch Normalization).

Max-Norm Regularization

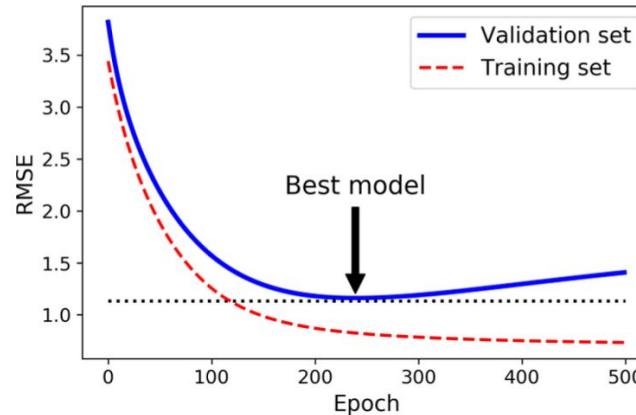
- In Keras, set every hidden layer's `kernel_constraint` argument to a `max_norm()` constraint, with the appropriate max value,

```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
                   kernel_constraint=keras.constraints.max_norm(1.))
```

- After each training step, the model's `fit()` method will call the object returned by `max_norm()`, passing it the layer's weights and getting rescaled weights in return, which then replace the layer's weights.
- We can also constrain the bias terms by setting the `bias_constraint` argument.

Early Stopping

- We know that a sign of overfitting is that validation error stops getting lower and starts getting larger.
- We can exploit this during Gradient Descent as another way of avoiding overfitting, known as early stopping:
 - During Gradient Descent, monitor validation error (or loss).
 - Interrupt training when the validation error has stopped improving for a certain number of epochs.



Early Stopping

In Keras, the `fit()` method accepts a `callbacks` argument that lets us specify a list of objects that Keras will call -

- at the start and end of training,
- at the start and end of each epoch, and even
- at the start and end of processing each batch.

Early Stopping in Keras

- In Keras, this is done using the EarlyStopping callback.
- The patience argument allows you to specify how many epochs must pass with no improvement relative to the current best.
- restore_best_weights=True restores the weights and biases from when validation error was at its lowest.

```
history = overfitting_model.fit(mnist_x_train, mnist_y_train, epochs=20, batch_size=32,
                                 verbose=0, validation_split=0.2,
                                 callbacks=[EarlyStopping(monitor="val_loss", patience=2, restore_best_weights=True)])
```

```
[(l, v) for l, v in zip(history.history["loss"], history.history["val_loss"])]
```

```
[(0.0012100160820409656, 0.1484818160533905),
 (0.0009289713925682008, 0.1520228236913681),
 (0.00046632185694761574, 0.15337863564491272)]
```

Early Stopping in Keras

- An advantage of early stopping is that we can be less concerned about choosing the number of epochs: just use something very large.
- But, now we have the problem of deciding on the patience. If runtime is your problem, then you can choose a low value. Otherwise, you choose a low value for 'easier' problems!

Conclusions

- Overfitting is a major problem but has many solutions.
- There are lots of solutions in addition to the ones above:
 - Remember Batch Normalization has a regularizing effect.
 - There are other techniques that we won't cover (e.g. Gradient clipping).
 - There are the things we've mentioned in an earlier lecture, especially getting more data!

Next lecture

Convolutional Neural Networks (CNNs)

IT496: Introduction to Data Mining



Lecture 31-32

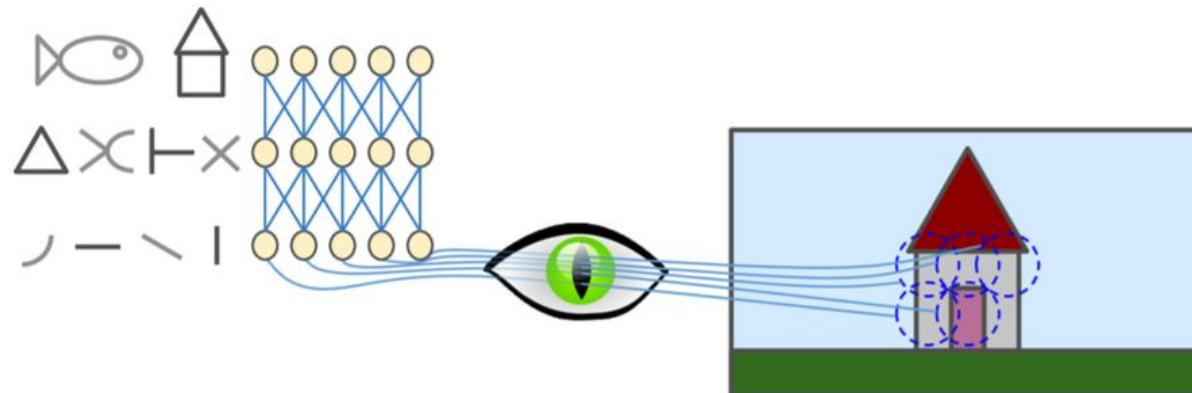
Convolutional Neural Networks

(Slides are created from the lecture notes of Dr. Derek Bridge, UCC, Ireland)

Arpit Rana
2nd / 3rd November 2023

Primate Vision

- David H. Hubel and Torsten Wiesel performed a series of experiments on cats in 1958 and 1959 and a few years later on monkeys.
- They showed that in the primate vision system, there seems to be a hierarchy of neurons within the visual cortex:

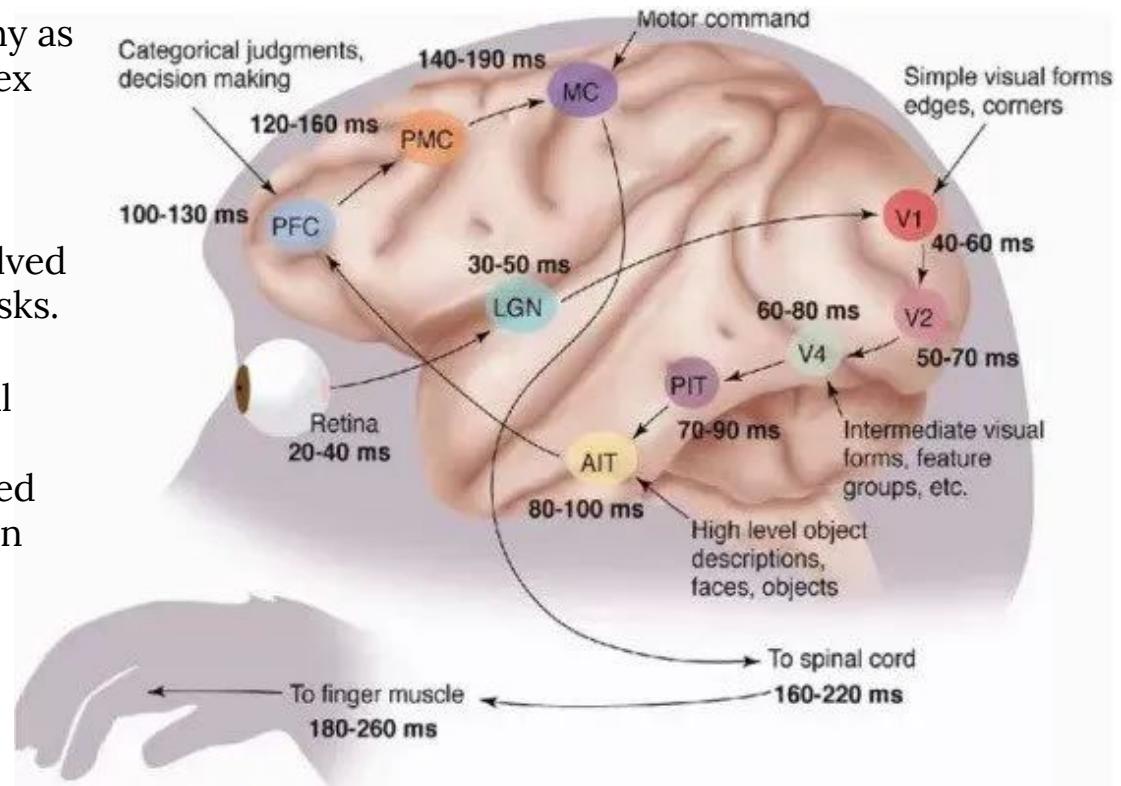


Primate Vision

- In the lowest layers,
 - neurons have small local *receptive fields*, i.e. they respond to stimuli in a limited region of the visual field; and
 - they respond to, e.g., spots of light.
- In higher layers,
 - they combine the outputs of neurons in the lower layers;
 - they have larger receptive fields; and
 - they respond to, e.g., lines at particular orientations (two neurons may have the same receptive field but react to different line orientations).
- In the highest layers,
 - they respond to ever more complex combinations, such as shapes and objects.

Visual Cortex in Human Brain

- There are perhaps as many as 8 layers in the visual cortex alone.
- The image shows the feedforward circuits involved in rapid categorization tasks.
- Numbers for each cortical stage corresponds to the shortest latencies observed and the more typical mean latencies.



Convolutional Neural Networks (CNNs)

- Yann LeCun et al. (1998) introduced the famous LeNet-5 architecture, widely used to recognize handwritten check numbers.
- It introduces two new building blocks: *convolutional layers* and *pooling layers*.

Motivation to Use CNNs

Why not simply use a regular deep neural network with fully connected layers for image recognition tasks?

- It breaks down for larger images because of the huge number of parameters it requires.
- For example,
 - A 100 x 100 image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just the first layer.
- CNNs solve this problem using *partially connected layers* and *weight sharing*.

Motivation to Use CNNs

Why not simply use a regular deep neural network with fully connected layers for image recognition tasks?

- It breaks down even for the small shifts in the original image..
- CNNs learn features that are *translation invariant*:
 - a feature map in a convolutional layer will recognize that feature anywhere in the image: bottom-left, top-right, ...

0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0



0	0	0	1	1	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	1
0	0	0	1	1	0

Shifted to the
right 1 pixel.

Motivation to Use CNNs

Why not simply use a regular deep neural network with fully connected layers for image recognition tasks?

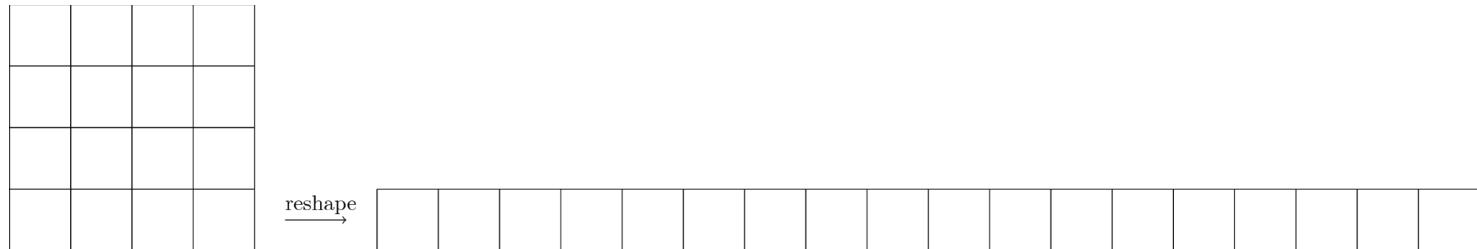
- It does not take advantage of spatial correlation in the original image..
- CNNs learn *spatial hierarchies* of features:
 - they take advantage of correlations that we observe in complex images.

Images are Rank-3 Tensors

Grayscale images:

- A grayscale image has a certain height h and width w . Therefore, it makes sense to represent them as rank 2 tensors (matrices) of integers in $[0, 255]$.
- So far, however, we have reshaped them into rank 1 tensors (vectors):

```
mnist_x_train = mnist_x_train.reshape((60000, 28 * 28))
```

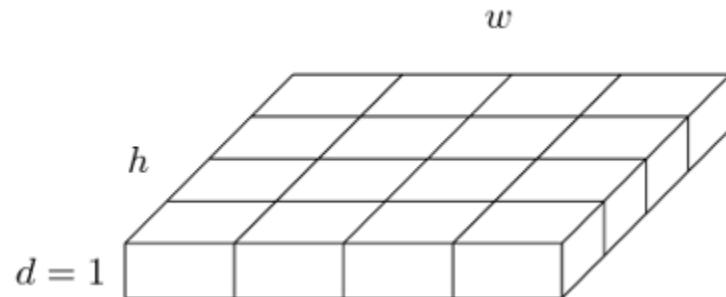


What is the disadvantage of this: what information gets destroyed?

Images are Rank-3 Tensors

- Henceforth, we will not flatten them in this way.
- In fact, for consistency with colour images, we will treat grayscale images as rank 3 tensors of shape:

```
mnist_x_train = mnist_x_train.reshape((60000, 28, 28, 1))
```

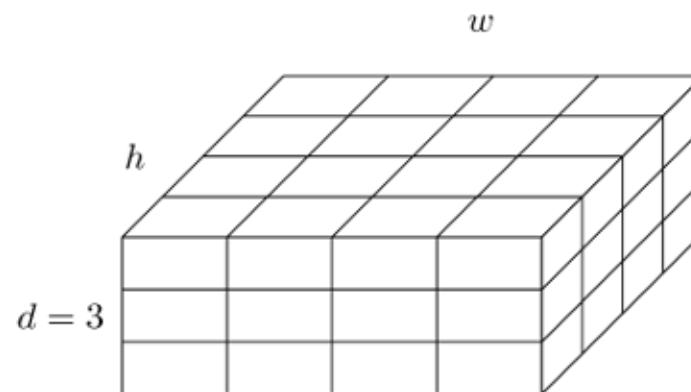


- Unlike the flattened representation, this shape makes it easier to match neurons with their corresponding inputs.

Images are Rank-3 Tensors

Colour images:

- These will be rank 3 tensors: height h , width w , and channels (or depth) d .
- $d = 3$. why?



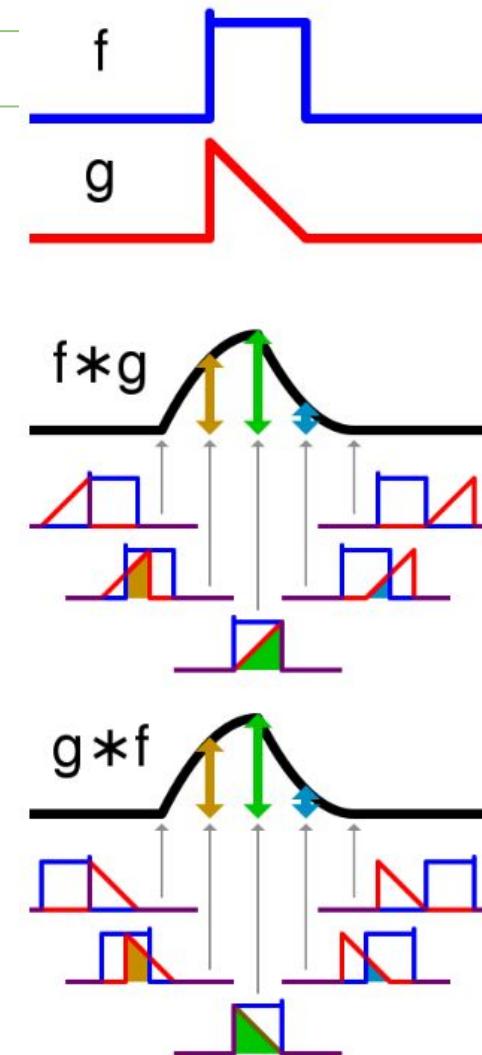
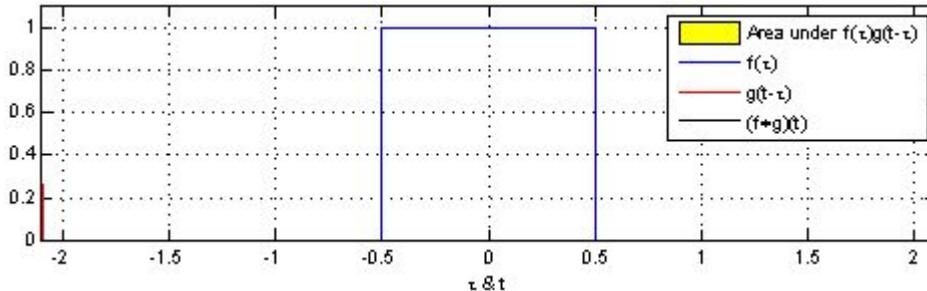
- Datasets of images:
 - Datasets of images (or mini-batches) will be rank 4 tensors: (m, h, w, d) .
- Why will datasets of videos be rank 5 tensors?

Convolution: A Mathematical Operation

A convolution is a mathematical operation that slides one function over another and measures the integral of their pointwise multiplication.

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t - \tau) d\tau.$$

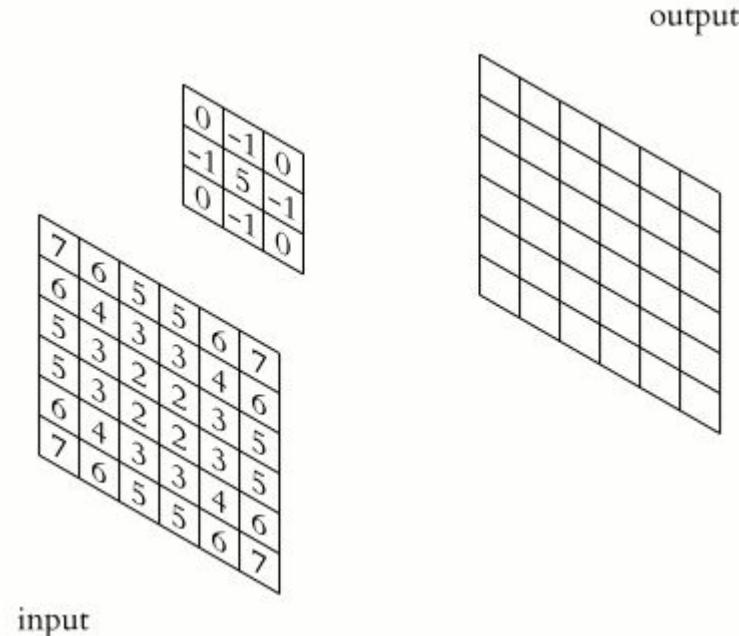
At each t , the convolution of f and g can be described as the area under the function $f(\tau)$ weighted by the function $g(-\tau)$ shifted by the amount t .



Convolution: A Mathematical Operation

For complex-valued functions f, g defined on the set \mathbb{Z} of integers, the discrete convolution of f and g is given by

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m],$$



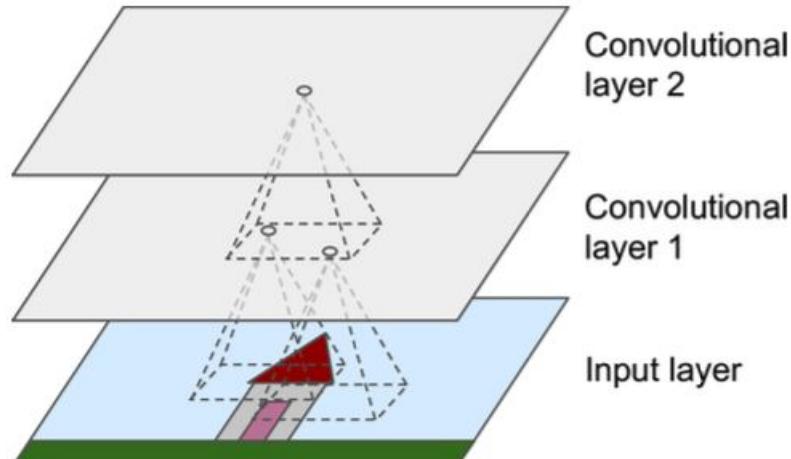
Discrete Finite 2D Convolution

Convolution Layer

- Consider a neural network whose inputs are images (each is a rank 3 tensor).
- A 2D convolutional layer is a rank 3 tensor of neurons, whose shape is (h, w, d) :
 - where d , the depth, is the number of *feature maps*
- For simplicity to begin with, let's assume $d=1$.

Convolution Layer

- Connections:
 - In the case of a dense layer, we saw that every neuron in a layer has connections from every neuron in the preceding layer.
 - But in the case of a convolutional layer, every neuron in a layer has connections from only a small rectangular window of neurons in the preceding layer, typically 3×3 or 5×5 or 7×7 .

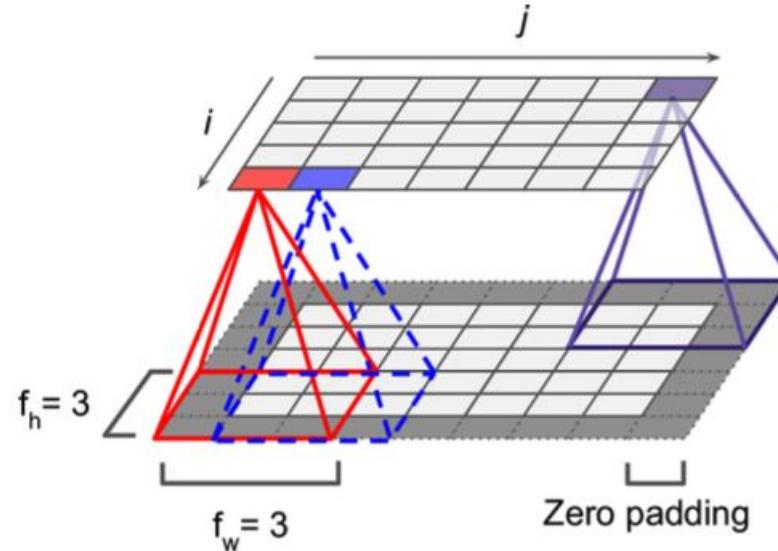


Connections between Convolution Layers

- Suppose the shape of the preceding layer is $(28, 28, 1)$ and the windows (receptive field) in the convolutional layer are 3×3 .
- This gives a convolutional layer whose height is 26 and whose width is 26. Why?
- A neuron located in row i , column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field.

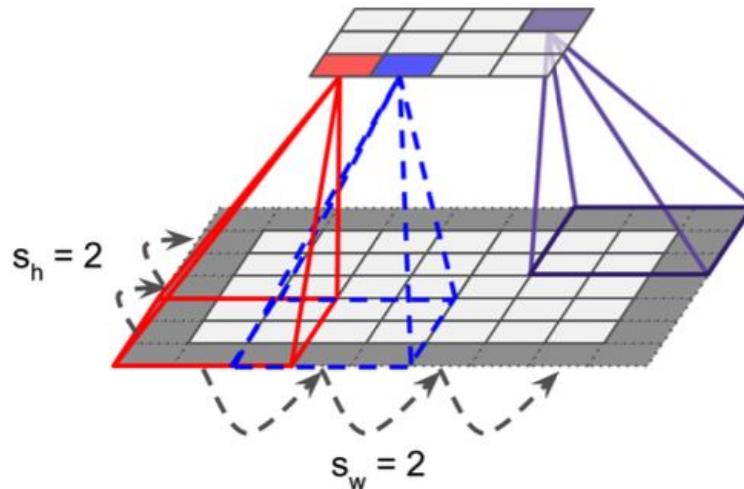
Connections between Convolution Layers

- In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs. This is called *zero padding*.



Reducing dimensionality using a Stride

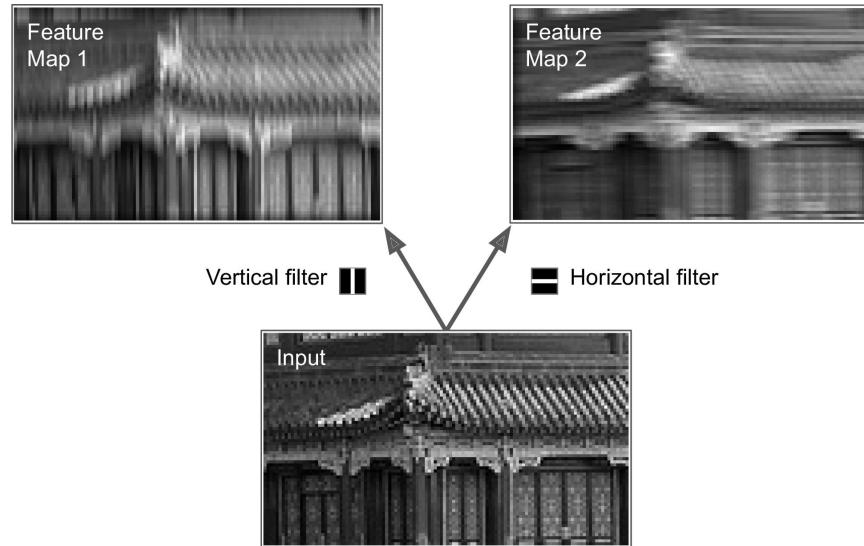
- It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields. The shift from one receptive field to the next is called the *stride*.



- A neuron located in row i , column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h - 1$, columns $j \times s_w$ to $j \times s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

Filters

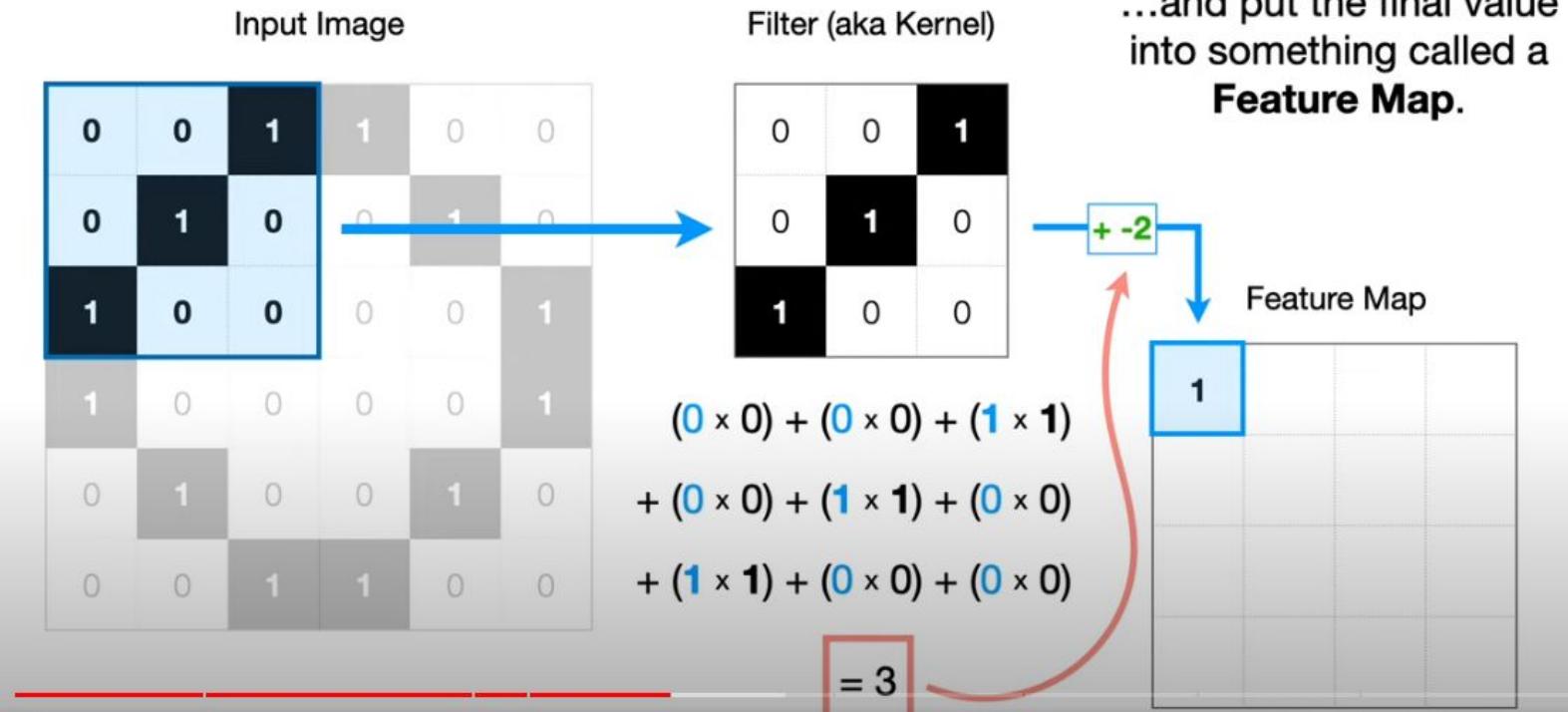
- A neuron's weights can be represented as a small image of the size of the receptive field. They are called *filters* or *convolution kernels*.
 - After initializing (through `kernel_initializer`), during training, the convolutional layer will *automatically* learn the most useful filters for its task.



Filters

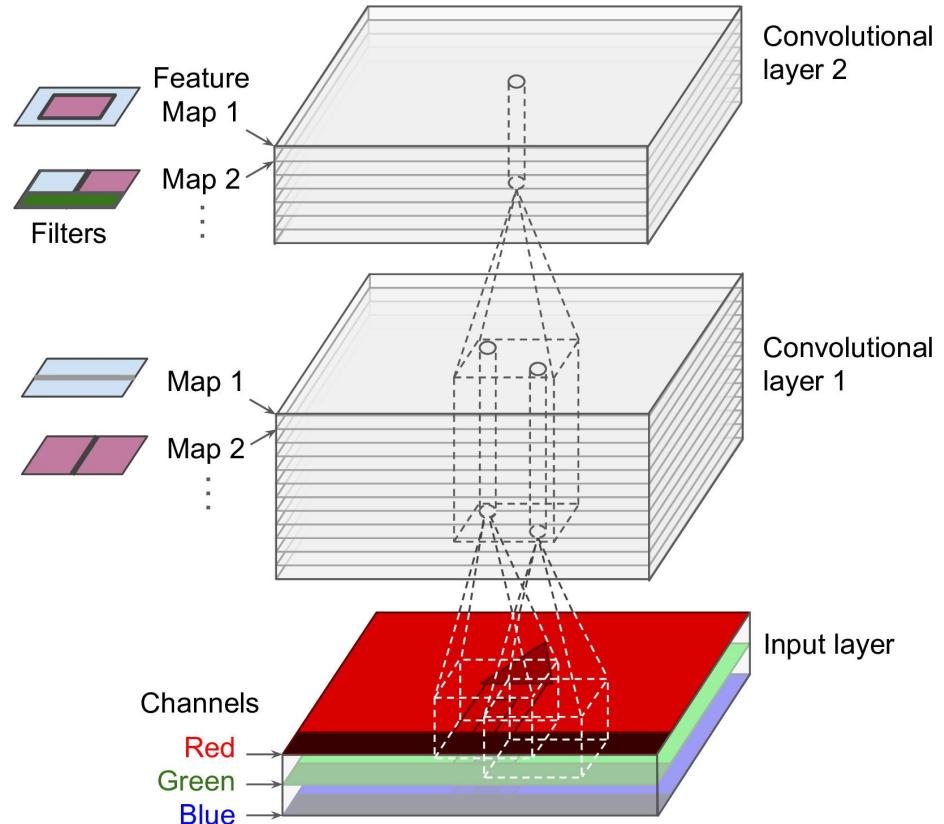
- A layer full of neurons using the same filter outputs a *feature map*, i.e., within one feature map, all neurons share the same weights and bias term!
- The idea of a feature map is that it will learn a specific aspect (feature) of its input:
 - e.g. the presence of a vertical line;
 - e.g.. the presence of a pair of eyes.

CNNs: Input Image to Feature Map



Stacking Multiple Feature Maps

- Now consider the case where $d > 1$: the convolutional layer comprises a stack of d feature maps.
- A neuron in a feature map in a convolutional layer is connected to a window of neurons in each of the feature maps of the previous layer
 - in the case of the first layer, in each of the channels of the input.
- This means that a feature map in one layer combines several feature maps (or channels) of the previous layer (the *spatial hierarchy*, mentioned earlier).



Stacking Multiple Feature Maps

- A neuron located in row i , column j of the feature map k in a given convolutional layer l is connected to the outputs of the neurons in the previous layer $l - 1$,
 - located in rows $i \times s_h$ to $i \times s_h + f_h - 1$ and
 - columns $j \times s_w$ to $j \times s_w + f_w - 1$, across all feature maps (in layer $l - 1$).
- Note that all neurons located in the same row i and column j but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

Stacking Multiple Feature Maps

- Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with } \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$ is the output of the neuron located in row i , column j in feature map k of the convolutional layer (layer l).
- As explained earlier, s_h and s_w are the vertical and horizontal strides, f_h and f_w are the height and width of the receptive field, and $f_{n'}$ is the number of feature maps in the previous layer (layer $l - 1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l - 1$, row i' , column j' , feature map k' (or channel k' if the previous layer is the input layer).
- b_k is the bias term for feature map k (in layer l). You can think of it as a knob that tweaks the overall brightness of the feature map k .
- $w_{u,v,k',k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u , column v (relative to the neuron's receptive field), and feature map k' .

Convolution Layer in Keras

- The following code creates a Conv2D layer in keras with
 - 32 filters (i.e., 32 feature maps),
 - each 3×3 ,
 - using a stride of 1 (both horizontally and vertically),
 - SAME padding (another padding type is VALID), and
 - applying the ReLU activation function to its outputs.

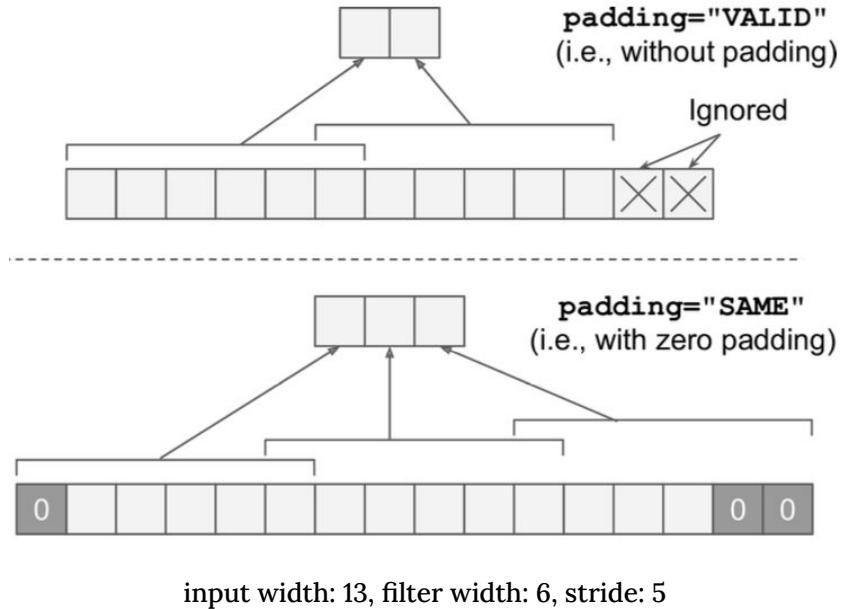
Convolution Layer in Keras

- **kernel_size** can be an integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window.
 - A single integer means the same value for all spatial dimensions.
- **strides** is equal to 1, however, it could also be a 1D array with 4 elements
 - batch stride (to skip some instances)
 - *vertical stride (s_h)*
 - *horizontal stride (s_w)*
 - channel stride (to skip some of the previous layer feature maps or channels)
- **activation** specifies the activation function to use. If we don't specify anything, no activation is applied on the feature maps.

Convolution Layer in Keras

padding may be of two types:

- If set to "**valid**", the convolutional layer does not use zero padding, and may ignore some rows and columns at the bottom and right of the input image, depending on the stride.
- If set to "**same**", the convolutional layer uses zero padding if necessary. In this case, the number of output neurons is equal to the number of input neurons divided by the stride, rounded up.
- When `padding="same"` and `strides=1`, the output has the same size as the input.



Pooling Layers

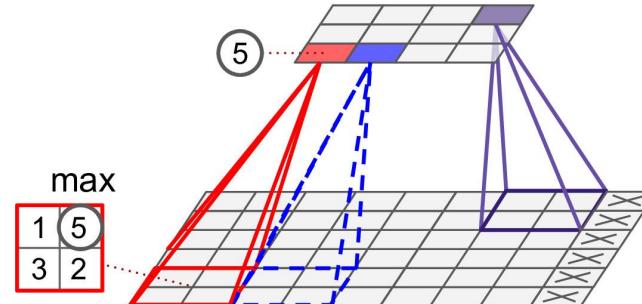
- The goal is to have a layer that shrinks the number of neurons in higher layers:
 - to reduce the amount of computation;
 - to reduce memory usage;
 - to reduce the number of parameters to be learned, thus reducing the risk of overfitting; and
 - to create a hierarchy in which higher convolutional layers contain information about the totality of the original input image.

Pooling Layers

- Again, it works on rectangular windows: neurons in the pooling layer are connected to windows of neurons in the previous layer
 - typically 2×2 ;
 - typically *adjacent* rather than overlapping.
- For example,
 - If the previous layer has height h and width w , and the pooling layer uses adjacent 2×2 pooling windows, then the pooling layer will have height $h/2$ and width $w/2$.
 - A pooling layer typically works on every input channel independently, so the output depth is the same as the input depth.

Types of Pooling Layers

- Pooling layers have no weights: nothing to learn.
- In a *max pooling* layer,
 - a neuron in the pooling layer receives the outputs of the neurons in the window in the previous layer and outputs only the largest of them.



Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

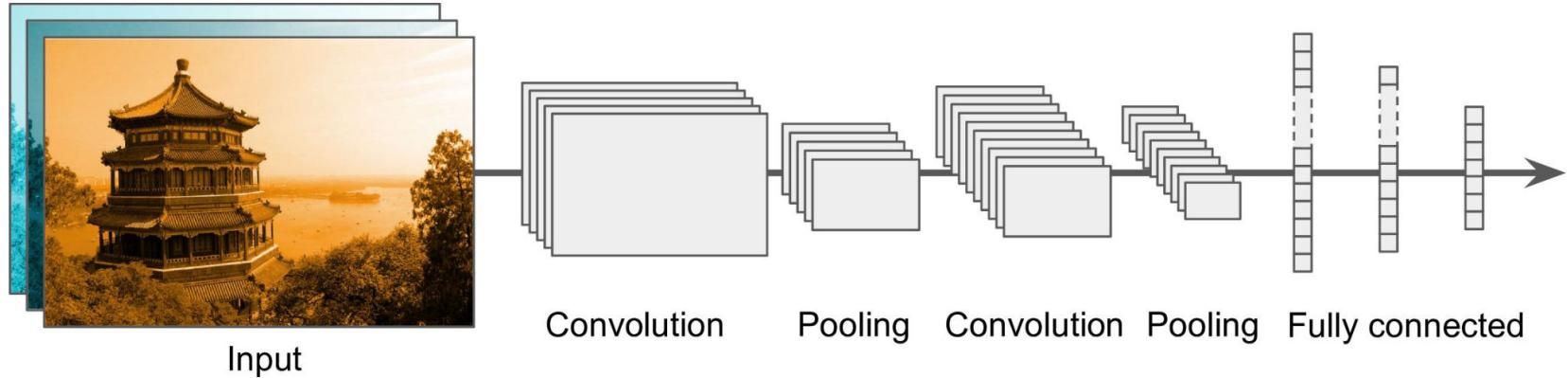
Types of Pooling Layers

- The following code creates a max pooling layer.

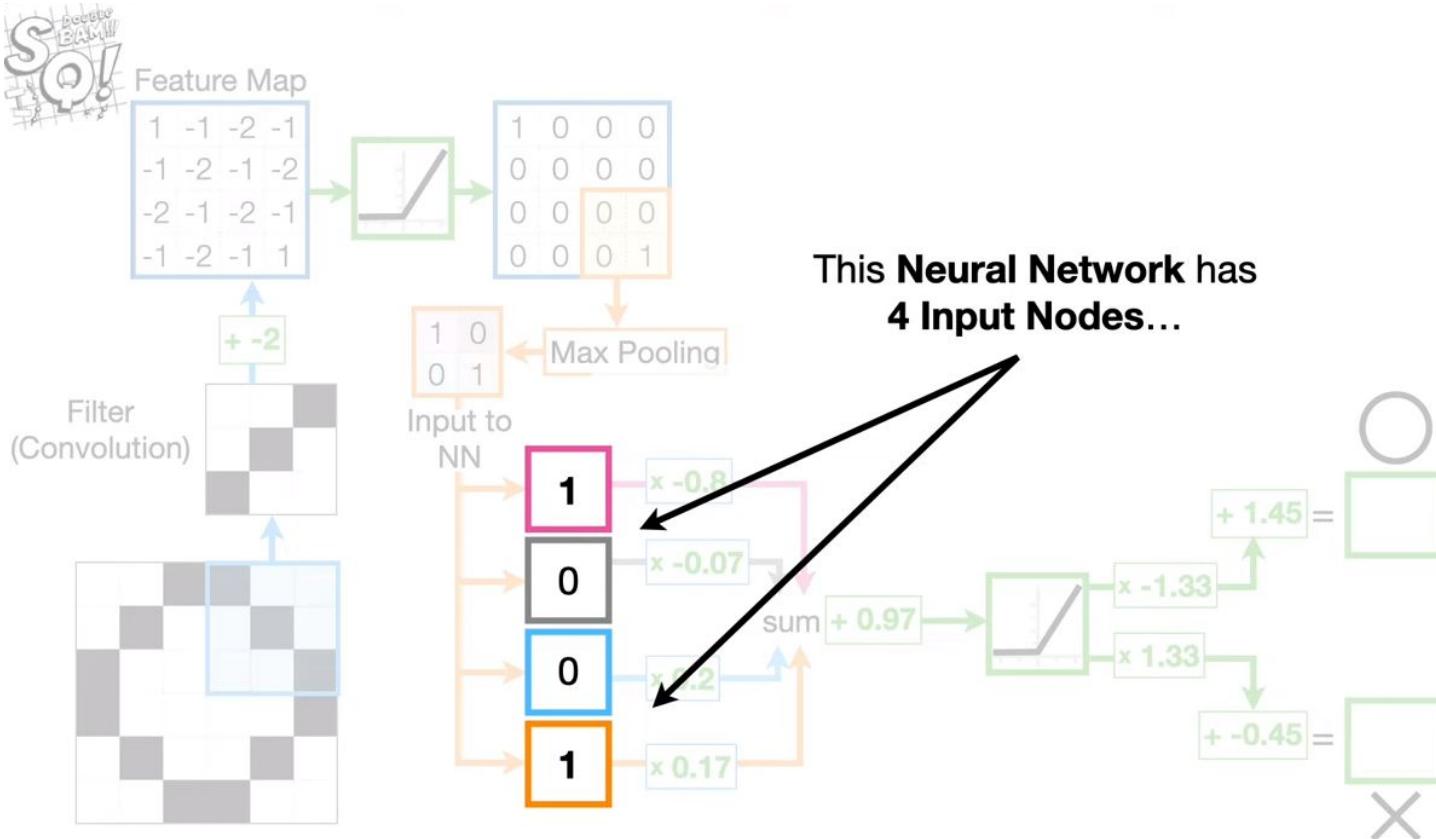
```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

- To create an average pooling layer, just use `AvgPool2D` instead of `MaxPool2D`.
- `AvgPool2D` works exactly like a max pooling layer, except it computes the *mean* rather than the *max*.

A Typical CNN Architecture



Typical Working of CNNs



Implementation in Keras

- Here is how we can implement a simple CNN to tackle the fashion MNIST dataset

```
from functools import partial

DefaultConv2D = partial(keras.layers.Conv2D, kernel_size=3, activation='relu',
                       padding="SAME")

convnet = keras.models.Sequential([
    DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(pool_size=2),
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    keras.layers.MaxPooling2D(pool_size=2),
    DefaultConv2D(filters=256),
    DefaultConv2D(filters=256),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Flatten(),
    keras.layers.Dense(units=128, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units=64, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units=10, activation='softmax')
])
```

```
convnet.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_5 (Conv2D)	(None, 28, 28, 64)	3200
max_pooling2d_3 (MaxPooling2D)	(None, 14, 14, 64)	0
conv2d_6 (Conv2D)	(None, 14, 14, 128)	73856
conv2d_7 (Conv2D)	(None, 14, 14, 128)	147584
max_pooling2d_4 (MaxPooling2D)	(None, 7, 7, 128)	0
conv2d_8 (Conv2D)	(None, 7, 7, 256)	295168
conv2d_9 (Conv2D)	(None, 7, 7, 256)	590080
max_pooling2d_5 (MaxPooling2D)	(None, 3, 3, 256)	0
flatten_2 (Flatten)	(None, 2304)	0
dense_6 (Dense)	(None, 128)	295040
dropout_2 (Dropout)	(None, 128)	0
dense_7 (Dense)	(None, 64)	8256
dropout_3 (Dropout)	(None, 64)	0
dense_8 (Dense)	(None, 10)	650
<hr/>		
Total params: 1413834 (5.39 MB)		
Trainable params: 1413834 (5.39 MB)		
Non-trainable params: 0 (0.00 Byte)		

Implementation in Keras

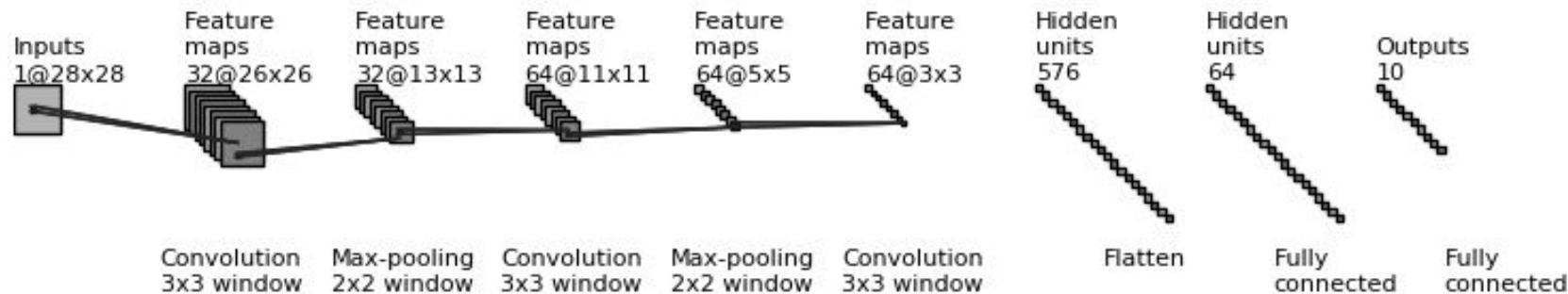
- Here is how we can implement a simple CNN to tackle the fashion MNIST dataset

```
convnet.fit(X_train_full, Y_train_full, epochs=20, batch_size=32,
            verbose=0, validation_split=0.2,
            callbacks=[keras.callbacks.EarlyStopping(monitor="val_loss", patience=2, restore_best_weights=True)])  
  
test_loss, test_acc = convnet.evaluate(X_test, Y_test)  
test_acc
```

- This CNN reaches over 92% accuracy on the test set. Not the best, however, much better than the dense networks.

Check Your Understanding

- Do you understand the numbers in the code?
- Do you understand the numbers in the output of `convnet.summary()` ?
- Do you understand the diagram below?



Final Remarks on Convolution Layer

- Note how convolutional layers are computationally efficient:
 - They have fewer parameters than dense layers (although, care here, because each one is involved in a more multiplications).
 - They can be easily parallelised.
- This is one reason for their popularity.

Next lecture

Training CNNs

7th November 2023

IT496: Introduction to Data Mining



Lecture 33-34

Training Convolutional Neural Networks

(Slides are created from the lecture notes of Dr. Derek Bridge, UCC, Ireland)

Arpit Rana
9th / 10th November 2023

Convolution Layers: Hyperparameters

- Convolution layers have quite a few hyperparameters:
 - number of filters,
 - their height and width,
 - the strides,
 - padding type, and
 - the activation function
- Using cross-validation to find the right hyperparameter values is very time consuming.
- There are common CNN architectures: LeNet5, AlexNet, GoogLeNet, VGGNet, ResNet, Xception, SENet, etc.; these can give some idea of which hyperparameter values work best in practice.
- We will NOT discuss all of them in this course.

Convolution Layers: Memory Requirements

- Convolution layers require a huge amount of RAM
 - during training, the backward pass of backprop requires all the intermediate values computed during the forward pass.
- For example,

```
200      5  
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,  
                           padding="SAME", activation="relu")
```

Input (RGB) image dimensions: 150 x 100 x 3

Number of parameters: $(5 \times 5 \times 3 + 1) \times 200 = 15,200$

Total float multiplications: $200 \times 150 \times 100 \times 5 \times 5 \times 3$

Total Memory Required: $200 \times 150 \times 100 \times 32 = 96 \text{ M bits} \approx 12 \text{ MB}$

(just for an instance and for just one layer)

Convolution Layers: Memory Requirements

- During Training:

Everything computed during the forward pass needs to be preserved for the backward pass, so the total amount of RAM required by all layers is the minimum requirement.

- What about during Inference (when making a prediction for a new instance)?

The RAM occupied by one layer can be released as soon as the next layer has been computed, so we only need as much RAM as required by two consecutive layers.

Convolution Layers: Memory Requirements

- If training crashes because of an *out-of-memory error*, we have the following choices:
 - reducing the mini-batch size.
 - reducing dimensionality using a stride,
 - removing a few layers,
 - using 16-bit floats instead of 32-bit floats,
 - distribute the CNN across multiple devices.

Convolution Layers: Overfitting

Mainly two regularization techniques are used:

- Adding dropout layer
- Data Augmentation
 - Artificially increasing the size of the training set by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.
 - Simply adding white noise will not help; the modifications should be learnable.
 - However, this is not as good as additional real examples: these synthesized examples are correlated with each other and the originals from which they were generated.

Convolution Layers: Overfitting

- Data Augmentation
 - This forces the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures.

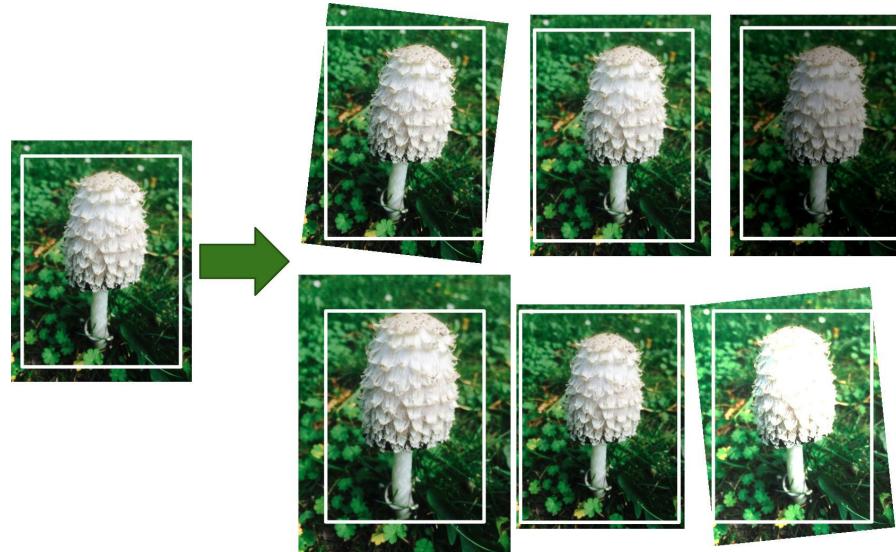


Image Recognition: Cats and Dogs

- There is a dataset, supplied by Microsoft researchers, for a Kaggle competition: [Dogs vs. Cats | Kaggle](#)
 - 12,500 medium-resolution JPEGs depicting cats and 12,500 depicting dogs.
- In the following implementation, we use a subset of the full dataset:
 - training set: 1000 cats and 1000 dogs;
 - validation set: 500 cats and 500 dogs;
 - test set: 500 cats and 500 dogs.

Image Recognition: Cats and Dogs - Data Preprocessing

- Keras gives us an extremely useful function: `image_dataset_from_directory` ([see the documentation](#)).
 - This function decodes images from one format into a grid with a certain number of channels. For example, if the raw images are JPEGs, it will decompress them.
 - Mostly, its default values for the arguments are what we want. Often we only need to think about the `directory` and the `label_mode`.

```
train_dataset = image_dataset_from_directory(directory=train_dir, label_mode="binary", image_size=(224, 224))
val_dataset = image_dataset_from_directory(directory=val_dir, label_mode="binary", image_size=(224, 224))
test_dataset = image_dataset_from_directory(directory=test_dir, label_mode="binary", image_size=(224, 224))
```

```
Found 2000 files belonging to 2 classes.
Found 1000 files belonging to 2 classes.
Found 1000 files belonging to 2 classes.
```

Image Recognition: Cats and Dogs - Creating the ConvNet

- Since the images are bigger than the Fashion MNIST ones, we use a network with more layers

```
inputs = Input(shape=(224, 224, 3))
x = Rescaling(scale=1./255)(inputs)
x = Conv2D(filters=128, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=128, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=64, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=32, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Flatten()(x)
x = Dense(512, activation="relu")(x)
outputs = Dense(1, activation="sigmoid")(x)
convnet = Model(inputs, outputs)
convnet.compile(optimizer=RMSprop(learning_rate=0.0001), loss="binary_crossentropy", metrics=["accuracy"])
```

Image Recognition: Cats and Dogs - Creating the ConvNet

```
: convnet.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	[(None, 224, 224, 3)]	0
rescaling (Rescaling)	(None, 224, 224, 3)	0
conv2d (Conv2D)	(None, 222, 222, 128)	3584
max_pooling2d (MaxPooling2D)	(None, 111, 111, 128)	0
conv2d_1 (Conv2D)	(None, 109, 109, 128)	147584
max_pooling2d_1 (MaxPooling 2D)	(None, 54, 54, 128)	0
conv2d_2 (Conv2D)	(None, 52, 52, 64)	73792
max_pooling2d_2 (MaxPooling 2D)	(None, 26, 26, 64)	0
conv2d_3 (Conv2D)	(None, 24, 24, 32)	18464
max_pooling2d_3 (MaxPooling 2D)	(None, 12, 12, 32)	0
flatten (Flatten)	(None, 4608)	0
dense (Dense)	(None, 512)	2359808
dense_1 (Dense)	(None, 1)	513
<hr/>		
Total params: 2,603,745		
Trainable params: 2,603,745		
Non-trainable params: 0		

Image Recognition: Cats and Dogs - Training and Testing

- We will use more epochs than before, but still with early stopping.

```
convnet_history = convnet.fit(train_dataset, epochs=30,  
                             validation_data=val_dataset,  
                             callbacks=[EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True)],  
                             verbose=0)
```

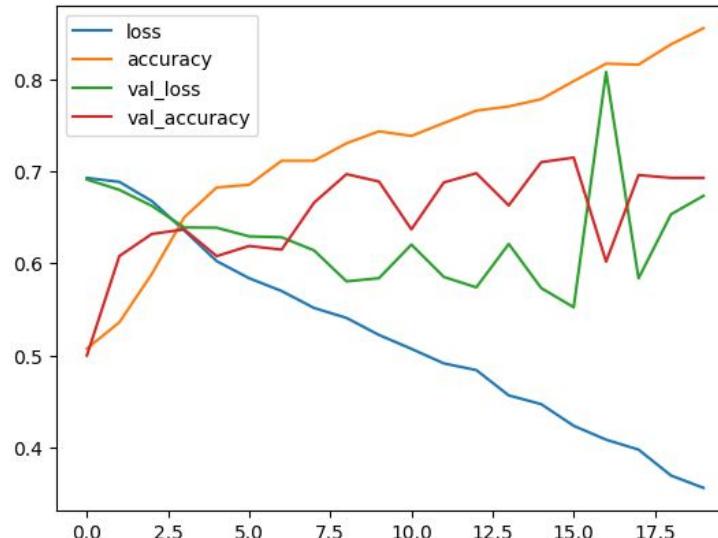


Image Recognition: Cats and Dogs - Data Augmentation

- In Keras, we add various layers to perform data augmentation.
- These layers will only augment the training data, not the validation or test data.
- We can *flip*, *rotate*, *zoom*, and *shift* the images.

```
augmentation_layers = Sequential([
    Input(shape=(224, 224, 3)),
    RandomFlip(mode="horizontal"),
    RandomRotation(factor=0.1),
    RandomZoom(height_factor=(-0.2, 0.2)),
    RandomTranslation(height_factor=0.2, width_factor=0.2)
])
```

Image Recognition: Cats and Dogs - Data Augmentation

- So now we add those layers to our network.

```
inputs = Input(shape=(224, 224, 3))
x = RandomFlip(mode="horizontal")(inputs)
x = RandomRotation(factor=0.1)(x)
x = RandomZoom(height_factor=(-0.2, 0.2))(x)
x = RandomTranslation(height_factor=0.2, width_factor=0.2)(x)
x = Rescaling(scale=1./255)(x)
x = Conv2D(filters=128, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=128, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=64, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=32, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Flatten()(x)
x = Dropout(0.5)(x)
x = Dense(512, activation="relu")(x)
outputs = Dense(1, activation="sigmoid")(x)
augmented_model = Model(inputs, outputs)
augmented_model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="binary_crossentropy", metrics=["accuracy"])
```

Image Recognition: Cats and Dogs - Training and Testing

- We will use more epochs than before, but still with early stopping.

```
augmented_model_history = augmented_model.fit(train_dataset, epochs=60,
                                              validation_data=val_dataset,
                                              callbacks=[EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True)],
                                              verbose=0)
```

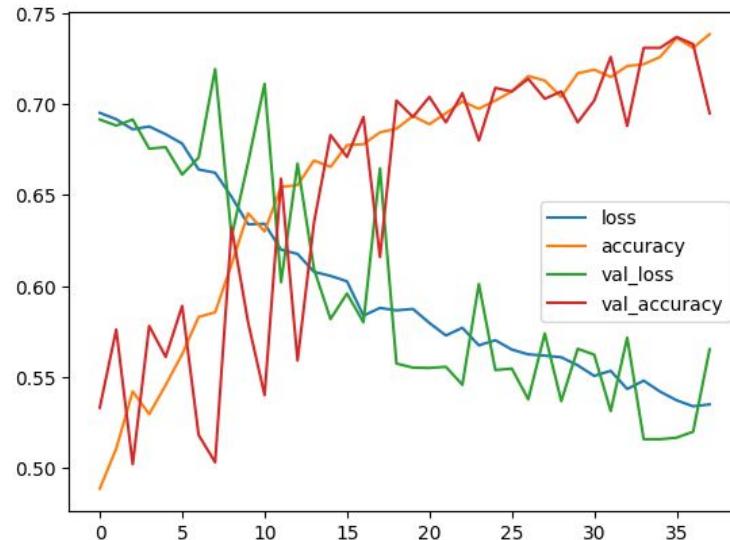


Image Recognition: Cats and Dogs - Training and Testing

- See the prediction on the following four images (`demo_dataset`)



```
augmented_model.predict(demo_dataset) # Class 0 is cat and class 1 is dog (alphabetic)
```

```
1/1 [=====] - 1s 600ms/step
array([[0.11105712],
       [0.9930074 ],
       [0.6058035 ],
       [0.65952003]], dtype=float32)
```

Pretrained Convolutional Neural Network

- A pretrained network is a saved network that was trained, usually on a large dataset.
- These are increasingly being made available for image classification, speech recognition and other tasks.
- Consider the ImageNet dataset (<http://www.image-net.org/>):
 - 1.4 million images, each manually labeled with one class per image;
 - thousands of classes, mostly animals and everyday objects;
 - annual competitions (ImageNet Large Scale Visual Recognition Challenges, ILSVRC), now hosted on Kaggle.

Pretrained Convolutional Neural Network

- We will see how to use a pretrained network: ResNet50

```
resnet50 = ResNet50(weights="imagenet", include_top=True, input_shape=(224, 224, 3))
```

- `include_top`: whether to include the fully-connected layer at the top of the network.

```
predictions = resnet50.predict(demo_dataset)
decode_predictions(predictions, top=3)
```

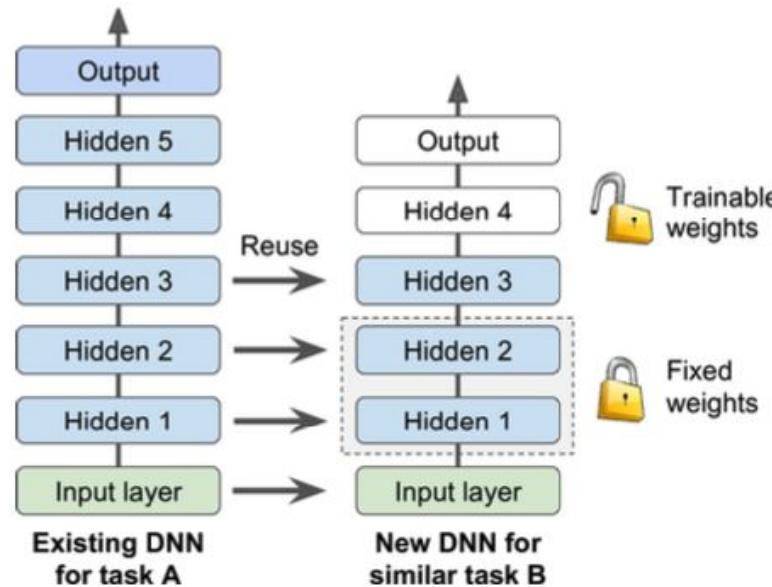
```
[[('n02123159', 'tiger_cat', 0.38074818),
 ('n02123045', 'tabby', 0.3423185),
 ('n02124075', 'Egyptian_cat', 0.12753917)],
 [(['n02105412', 'kelpie', 0.41296554),
  ('n02099712', 'Labrador_retriever', 0.19615647),
  ('n02099849', 'Chesapeake_Bay_retriever', 0.10481451)],
 [(['n02412080', 'ram', 0.8103329),
  ('n02444819', 'otter', 0.040895212),
  ('n02396427', 'wild_boar', 0.025922885)],
 [(['n04398044', 'teapot', 0.99999344),
  ('n03063689', 'coffeepot', 6.545879e-06),
  ('n04560804', 'water_jug', 1.7925926e-08)]]
```

Transfer Learning

- Transfer learning:
 - taking a model that was learned when solving one problem and re-using it for solving a different but related problem.
- Advantages:
 - it speeds-up training for the new problem;
 - it means that less training data may be needed for the new problem.

Transfer Learning

- Deep neural networks are more amenable to transfer learning than many other machine learning techniques:
 - take a pre-trained network;
 - re-use its lower layers, even *frozening* their weights.



Transfer Learning

- For example:
 - We have several networks that are pre-trained on ImageNet, including ResNet50:
 - trained to classify images into 1000 classes (various animals, vehicles, etc.).
 - We can re-use the lower layers in a new network that is trained to classify images of just cats and dogs, or different types of vehicles, or for face recognition, or perhaps even facial expression recognition.
- Of course, this will only work well if the original and new tasks share similar low-level features.
 - The more similar the new problem is to the original problem, the more layers we may want to re-use.

Re-using the Convolutional Base of a Pretrained ConvNet

- Convolutional Neural Networks typically comprise two parts:
 - the *convolutional base*: the convolutional and pooling layers;
 - the densely-connected top layers for, e.g. classification.
- We want to reuse the convolutional base:
 - the features learned by these layers are likely to be more generic;
 - the features learned by the top layers will be more specific to the original task.

Image Recognition: Cats and Dogs - Using Transfer Learning

- We *freeze* the weights in the layers of the convolutional base. If we did not, then the features that ResNet50 learned previously would be lost.

```
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions

resnet50_base = ResNet50(weights="imagenet", include_top=False, input_shape=(224, 224, 3))

resnet50_base.trainable = False
```

Image Recognition: Cats and Dogs - Using Transfer Learning

We will re-use the convolutional base of the ResNet50 model within a new network for classifying cats and dogs.

- We create our new network but we add the pre-trained convolutional base of the ResNet50 model, just as we would add a layer.
- We can precede it by a function to *preprocess* the data into a form that ResNet50 expects. This even allows us to exclude the Rescaling layer.
- We could even add the augmentation layers if we wanted.

```
inputs = Input(shape=(224, 224, 3))
x = preprocess_input(inputs)
x = resnet50_base(x)
x = Flatten()(x)
outputs = Dense(1, activation="sigmoid")(x)
transfer_model = Model(inputs=inputs, outputs=outputs)
```

Image Recognition: Cats and Dogs - Using Transfer Learning

```
: transfer_model.summary()
```

Model: "model_2"

Layer (type)	Output Shape	Param #
<hr/>		
input_6 (InputLayer)	[(None, 224, 224, 3)]	0
tf.__operators__.getitem (SlicingOpLambda)	(None, 224, 224, 3)	0
tf.nn.bias_add (TFOpLambda)	(None, 224, 224, 3)	0
resnet50 (Functional)	(None, 7, 7, 2048)	23587712
flatten_2 (Flatten)	(None, 100352)	0
dense_4 (Dense)	(None, 1)	100353
<hr/>		
Total params: 23,688,065		
Trainable params: 100,353		
Non-trainable params: 23,587,712		

Image Recognition: Cats and Dogs - Using Transfer Learning

```
transfer_model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="binary_crossentropy", metrics=["accuracy"])
```

```
transfer_model_history = transfer_model.fit(train_dataset, epochs=30,
                                             validation_data=val_dataset,
                                             callbacks=[EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True)],
                                             verbose=0)
```

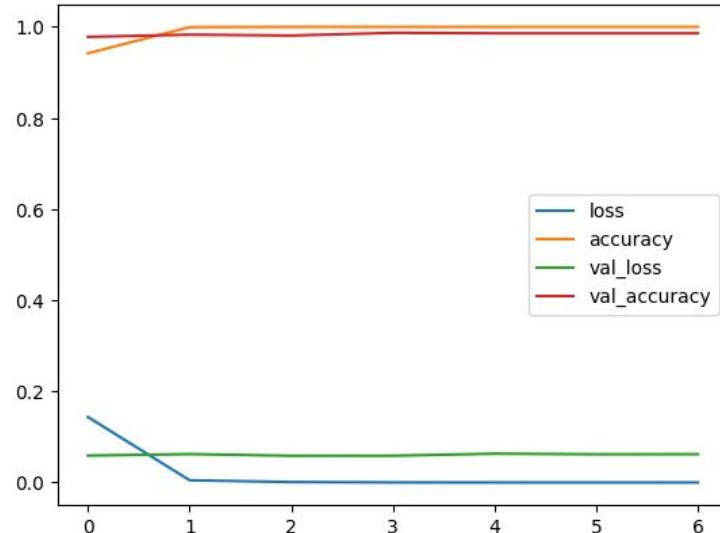


Image Recognition: Cats and Dogs - Using Transfer Learning

Now that our new top layers are well-trained, we can *unfreeze* all layers in the base (or just the top ones in the base) and continue training.

- Simplest is to unfreeze all of them, like this: `# resnet50_base.trainable = True`
- But, for reasons we will not unfreeze BatchNormalization layers:

```
for layer in resnet50_base.layers:  
    if isinstance(layer, BatchNormalization):  
        layer.trainable = False  
    else:  
        layer.trainable = True
```

Image Recognition: Cats and Dogs - Using Transfer Learning

- In Keras, re-compilation is needed at this point. (We re-compile, but we do not build a new model as the whole idea is to continue to tune the one we have already built.)

```
transfer_model.compile(optimizer=RMSprop(learning_rate=0.001), loss="binary_crossentropy", metrics=["accuracy"])
```

```
transfer_model_history = transfer_model.fit(train_dataset, epochs=30,
                                             validation_data=val_dataset,
                                             callbacks=[EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True)],
                                             verbose=0)
```

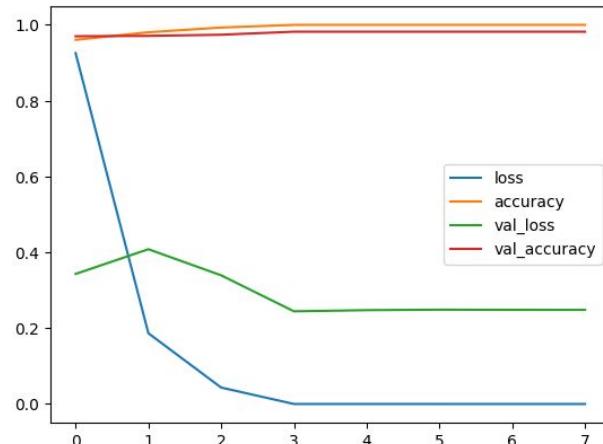


Image Recognition: Cats and Dogs - Using Transfer Learning

- Now, we test and compare all the version of our convnets on the test set.

```
# The original convnet
test_loss, test_acc = convnet.evaluate(test_dataset)
test_acc
```

```
32/32 [=====] - 138s 3s/step - loss: 0.5667 - accuracy: 0.7280
0.7279999852180481
```

```
# The convnet with dropout, trained using data augmentation
test_loss, test_acc = augmented_model.evaluate(test_dataset)
test_acc
```

```
32/32 [=====] - 3s 70ms/step - loss: 0.5555 - accuracy: 0.7150
0.7149999737739563
```

```
# The convnet that was trained and tuned using transfer learning
test_loss, test_acc = transfer_model.evaluate(test_dataset)
test_acc
```

```
32/32 [=====] - 4s 96ms/step - loss: 0.2624 - accuracy: 0.9790
0.9789999723434448
```

Concluding Remarks on CNNs

- We saw that transfer learning helps us in cases where we have more limited data.
- Consider tasks that involve audio, text and time series data. For these tasks, Recurrent Neural Networks would be the obvious choice (we will see in IT492: Recommendation Systems).
 - But, in some cases, 1D (one-dimensional) convolutional networks can be used successfully instead.
 - Although, to be fair, these days Transformers (also will be covered in IT492) are displacing Recurrent Neural Networks and 1D convnets in many cases.
- Consider tasks that involve graphs (nodes and edges). We can process these using Graph Neural Networks. There are graph convolutions that can be used within these neural networks.

Next lecture

Clustering

21st November 2023

IT496: Introduction to Data Mining



Lecture 35-36

Clustering Analysis

Arpit Rana
21st / 23rd November 2023

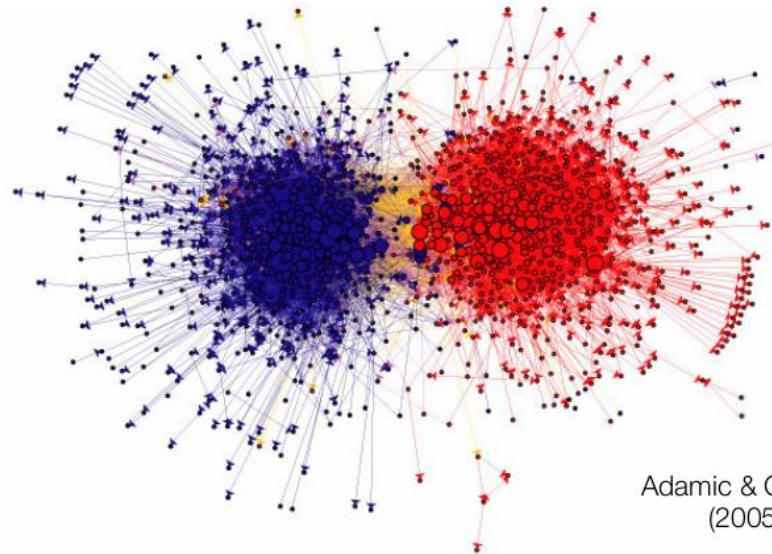
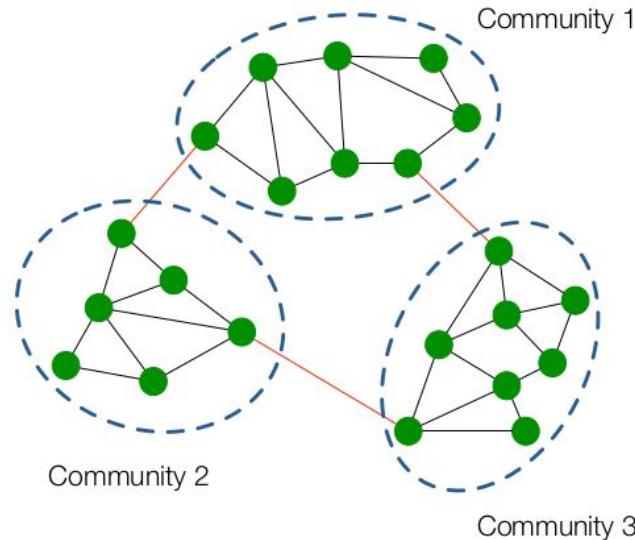
Applications of Clustering

Market segmentation: Unsupervised task that attempts to automatically grouping customers into separate clusters, so that customers in the same cluster have similar needs and respond similarly to a marketing action.



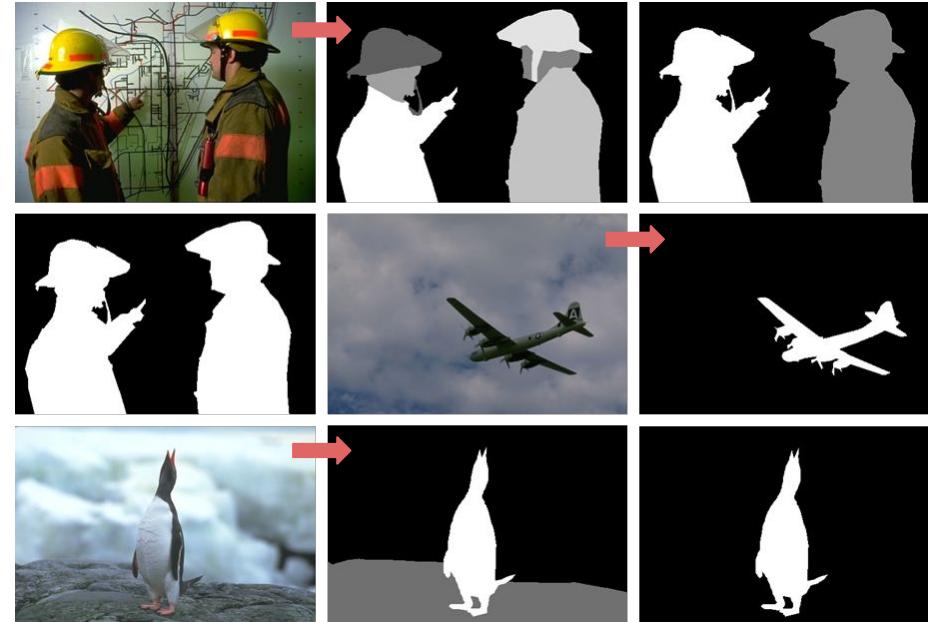
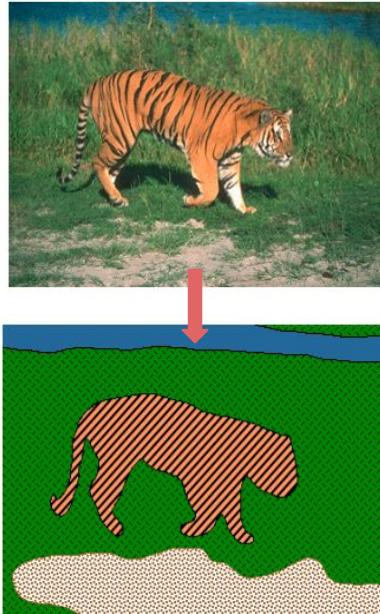
Applications of Clustering

Community Detection: Given a social network, apply clustering to identify communities of users who are well-connected to one another, and who are separated from other communities.



Applications of Clustering

Image segmentation: Unsupervised task in computer vision that attempts to automatically split an image into regions with similar *colour* or *texture*, or *both*. Aim is to partition the image into its constituent “objects”.



Applications of Clustering

Document Clustering: Automatically group related documents together based on similar content (e.g. related articles on Google News).

The screenshot shows the Google News interface. At the top, there's a search bar and a navigation bar with 'Ireland edition' and 'Modern' options. The main area is divided into sections: 'Top Stories' on the left and 'World' on the right.

Top Stories:

- World
- Ottawa
- Oscar Pistorius
- Jean-Claude Juncker
- Kenny G
- Mexico
- Syria
- Iran
- European Union
- Students
- Bessbrook
- Ireland
- Business
- Technology
- Entertainment
- Sports
- Science
- Health
- More Top Stories

World Section:

Gunman who opened fire in Canada's parliament is 'son of country's immigration ...

Irish Independent - 4 minutes ago

Michael Zehaf-Bibeau, the slain 32-year-old suspected killer of a Canadian Forces soldier near Parliament Hill, was a petty criminal - a man who had had a religious awakening in recent years and seemed to have become mentally unstable, it is reported by ...

Ottawa shooting: Suspected gunman Michael Zehaf-Bibeau BBC News

Terrorism rocks Ottawa Toronto Star

See realtime coverage

From Canada: Matt Gurney: Ottawa just had its trial by fire. Thank God, it passed National Post

In-depth: Ottawa Reeling as Soldier Dies After Attack at Parliament Bloomberg

Live Updating: Ottawa shooting: Live updates as Canadian PM condemns 'brutal' attack that left ... Mirror.co.uk

Wikipedia: 2014 shootings at Parliament Hill, Ottawa

Related:

- Ottawa »
- Parliament of Canada »
- Canada »

US-led airstrikes in Syria killed over 500, say activists

The Hindu - 16 minutes ago

U.S.-led coalition airstrikes on Syria have killed more than 500 people, mainly Islamic militants, since they began last month, activists said on Thursday, as warplanes targeted an oil field in the eastern Deir el-Zour Province near Iraq.

Alleged: Mexican mayor 'masterminded' disappearance of 43 students

Washington Post - 45 minutes ago

The last time anyone saw the 43 college students abducted in southwestern Mexico last month, they were being crammed into patrol cars in Iguala, a town some 125 miles from Mexico City.

China shares fall to one-month low on liquidity concerns, Hong Kong edges lower

Economic Times - 4 hours ago

SHANGHAI: China shares fell to one-month lows by midday on Thursday, dampened by concerns over liquidity amid a rush of initial public offerings as well as profit-taking pressure.

Applications of Clustering

Topic modeling: Unsupervised task of discovering the underlying thematic structure in a text corpus - i.e. the key “topics” in the data.



Clustering

Grouping examples in the absence of any external information is called *Clustering*.

- No labelled training examples to learn from.
- Generally we will not know in advance how many clusters are present in the data.

Clustering

Clusters are inferred from the data such that -

- Examples within a cluster should be similar.
- Examples from different clusters should be dissimilar.

Secondary goals in clustering

- Avoid very small and very large clusters
- Define clusters that are easy to explain to the user
- Many others . . .

Clustering

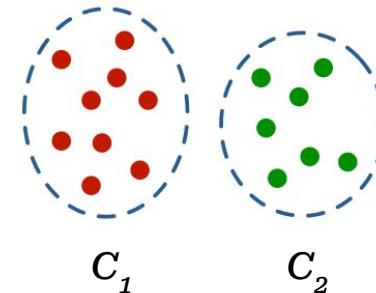
Clusters are inferred from the data without human input.

- However, there are many ways of influencing the outcome of clustering:
 - number of clusters,
 - similarity measure,
 - representation of examples (e.g., documents),
 - ...

Clustering: Types

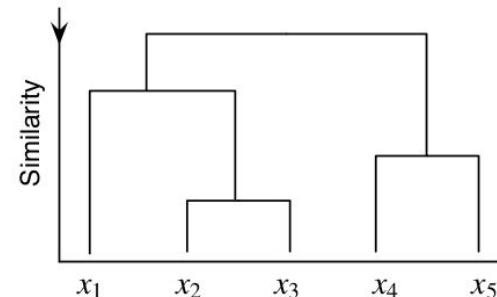
Flat algorithms (Partitioning)

- Usually start with a random (partial) partitioning of examples into groups
- Refine iteratively
- Main algorithm: *K-means*



Hierarchical algorithms

- Create a hierarchy
- Bottom-up, *agglomerative*
- Top-down, *divisive*



Clustering: Types

Hard clustering

- Each example is in exactly one cluster.
 - More common and easier to do

Soft clustering

- An example can be in more than one cluster.
 - Makes more sense for browsable hierarchies
 - You may want to put sneakers in two clusters:
 - Sports apparel
 - Shoes
- You can only do that with a soft clustering approach.

We will do *flat, hard clustering* only in this course.

Clustering

Flat algorithms compute a partition of N examples into a set of K clusters.

- **Given:** a set of examples and the number K
- **Find:** a partition into K clusters that optimizes the chosen partitioning criterion
- **Global optimization:** exhaustively enumerate partitions, pick optimal one
 - Not tractable
- **Effective heuristic method:** K -means algorithm

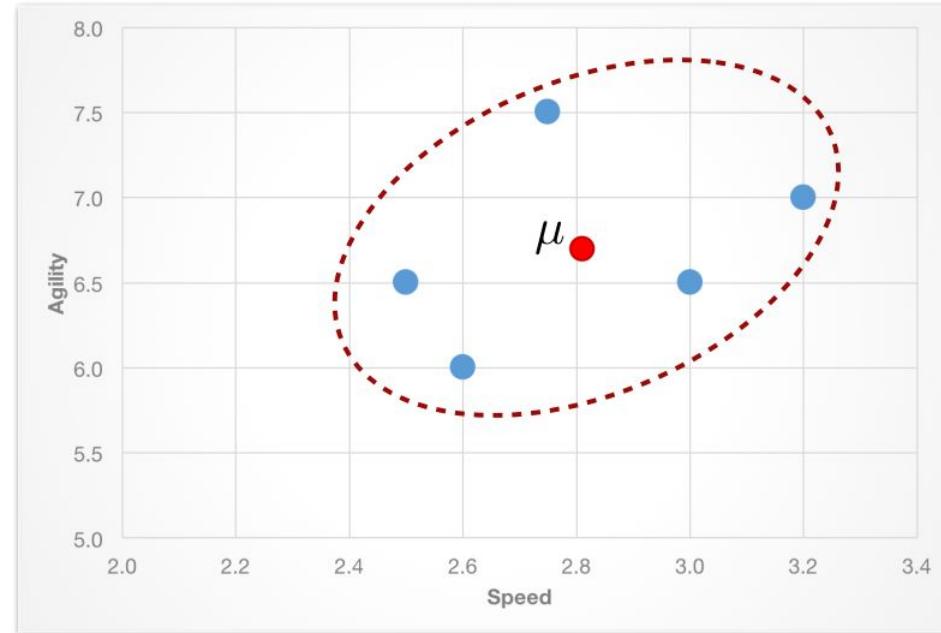
Clustering: What is Centroid?

Centroid: The mean vector of all items assigned to a given cluster (i.e. the mean of their feature vectors).

Athlete	Speed	Agility
1	2.6	6.0
2	3.0	6.5
3	2.5	6.5
4	3.2	7.0
5	2.8	7.5
Centroid	2.82	6.7

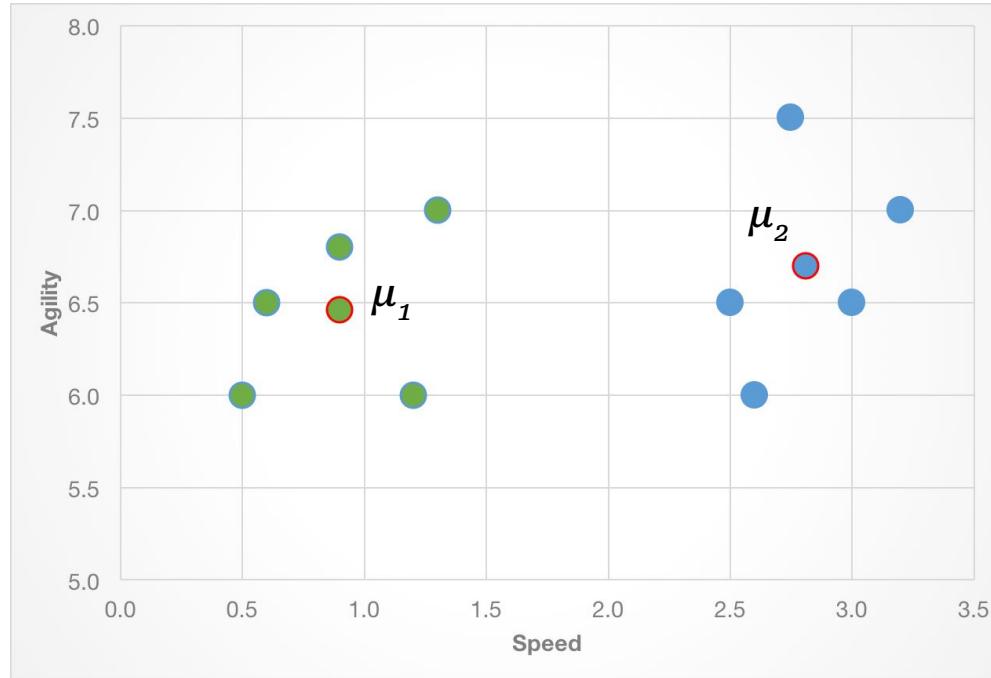
$$(2.6 + 3.0 + 2.5 + 3.2 + 2.8)/5 \\ = 2.82$$

$$(6.0 + 6.5 + 6.5 + 7.0 + 7.5)/5 \\ = 6.7$$



Clustering: Cluster Assignment based on Centroid

Each of the k clusters in a clustering can be represented by its own centroid μ_i . Example of two clusters, with centroids shown:



Clustering: Assignment based Clustering

Given a set X of data points, we want a set C of k centers $\{\mu_1, \mu_2, \dots, \mu_k\}$ which minimises some cost function -

$$\text{minimize } \sum_{x \in X} d(x, C)^2$$

Here, $d(x, C) = \min_{\mu_i \in C} d(x, \mu_i)$

Often $d()$ is the Euclidean function

$$d(x, \mu) = \sqrt{\sum_{j=1}^m (x_j - \mu_j)^2}$$

sum of squared difference over all m feature values

Clustering: Assignment based Clustering

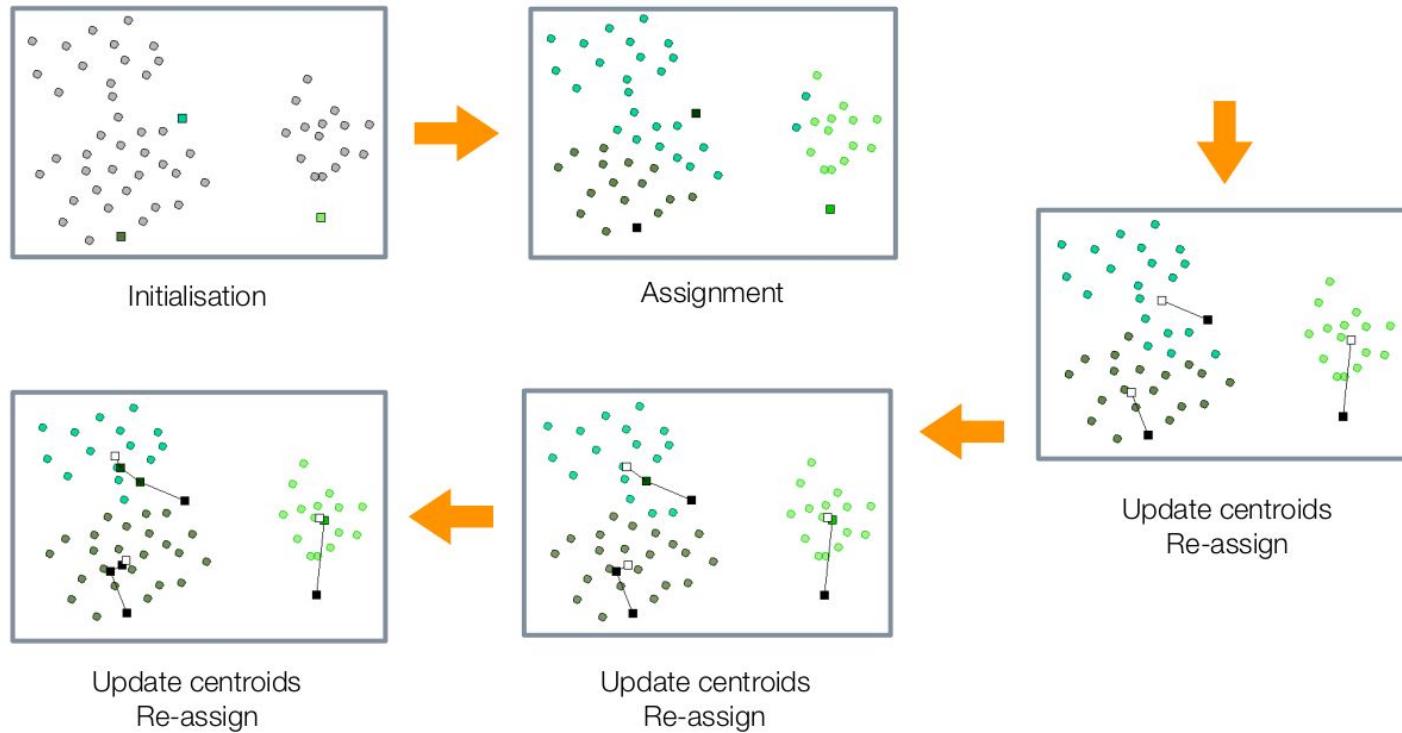
$$\text{minimize} \sum_{x \in X} d(x, C)^2 \quad \text{with} \quad d(x, C) = \min_{\mu_i \in C} d(x, \mu_i) \quad \text{and} \quad d(x, \mu) = \sqrt{\sum_{j=1}^m (x_j - \mu_j)^2}$$

- Minimising the k -Means objective is *NP-hard*
- Finding the optimal solution requires considering all K^n possible ways of assigning n data points to K clusters.
 - Each data point can be assigned to any one of the k clusters, leading to an exponential number of possible combinations.

Clustering: Assignment based Clustering

- *Lloyd's algorithm* is often used as a heuristic to minimise it
 - Reduce Sum of Squared Error (SSE) via a two step iterative process:
 - *Reassign* items to their nearest cluster centroid
 - *Update* the centroids based on the new assignments
 - Repeatedly apply these two steps until the algorithm converges to a final result

Clustering: Lloyd's Algorithm



Clustering: Lloyd's Algorithm for K-means

Inputs

- Data: Set of unlabelled items
- k : User-specified target number of clusters
- Maximum number of iterations to run

Algorithm Steps

- **Initialisation:** Select k initial cluster centroids (e.g. at random)
- **Assignment step:** Assign every item to its nearest cluster centroid (e.g. using Euclidean distance).
- **Update step:** Recompute the centroids of the clusters based on the new cluster assignments, where a centroid is the mean point of its cluster.
- Go back to Step 2, until when no reassessments occur (or until a maximum number of iterations is reached).

Clustering: Lloyd's Algorithm for K-means

Lloyd's algorithm converges as the cost decreases monotonically.

- It may not converge in polynomial time -- there are examples where the algorithm takes exponentially many steps.
- The algorithm works well in practice.
- We often stop after a pre-defined number of iterations.
- No guarantee on the cost of the solution.

Clustering: Lloyd's Algorithm for K-means

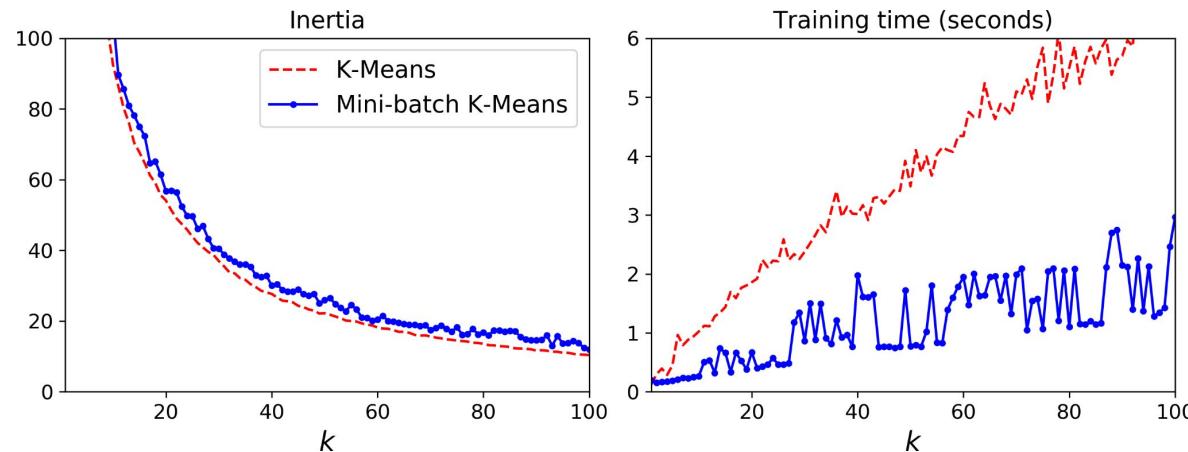
Complexity Analysis

- Computing a distance of two vectors is $O(M)$.
- Assignment step: $O(KNM)$ (we need to compute KN example-centroid distances)
- Update step: $O(NM)$ (we need to add each of the example's $< M$ values to one of the centroids)
- Assume number of iterations bounded by I
 - Overall complexity: $O(IKNM)$ – linear in all important dimensions
- However: This is not a real worst-case analysis.

Mini-Batch K-means

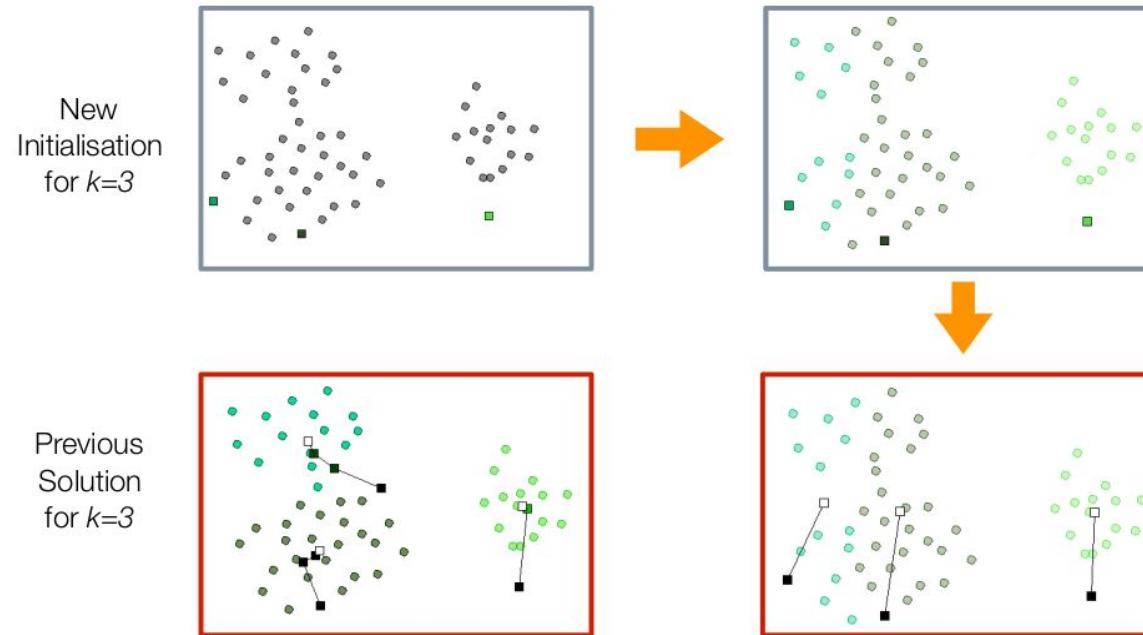
Instead of using the full dataset at each iteration,

- the algorithm is capable of using mini-batches, moving the centroids just slightly at each iteration.
- This speeds up the algorithm typically by a factor of 3 or 4 and makes it possible to cluster huge datasets that do not fit in memory.



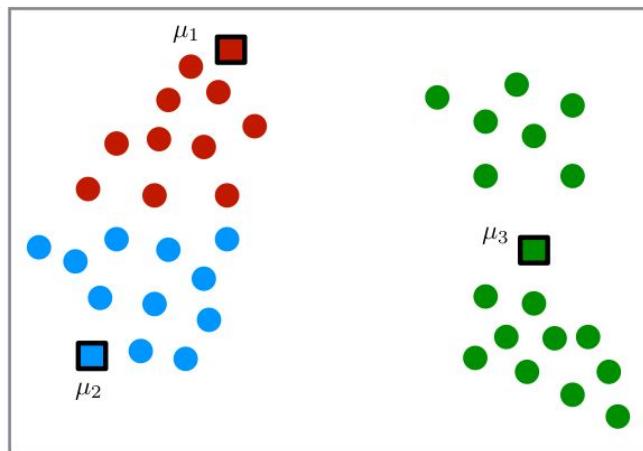
Lloyd's Algorithm: Cluster Initialization

Results produced by Lloyd's algorithm are often highly dependent on the initial solution. Different starting positions can lead to different local minima - i.e. different clusterings of the same data.

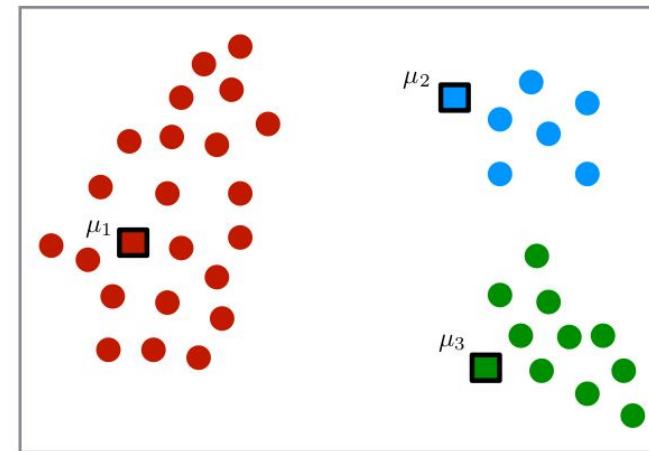


Lloyd's Algorithm: Cluster Initialization

A poor choice of initial centroids will often lead to a poor clustering that is not useful. A better initialisation will lead to different clusters.



Initialisation 1



Initialisation 2

Common strategy: Run the algorithm multiple times, select the solution(s) that scores well according to some validation measure.

K-Means++ Cluster Initialiser

k-Means++ Initialiser:

- Start with $C = \emptyset$
- Pick $x \in X$ uniformly at random and add it to C
- Repeat $k - 1$ times:
 - Pick an $x \in X$ with probability proportional to $d(x, C)^2$
 - Add x to C

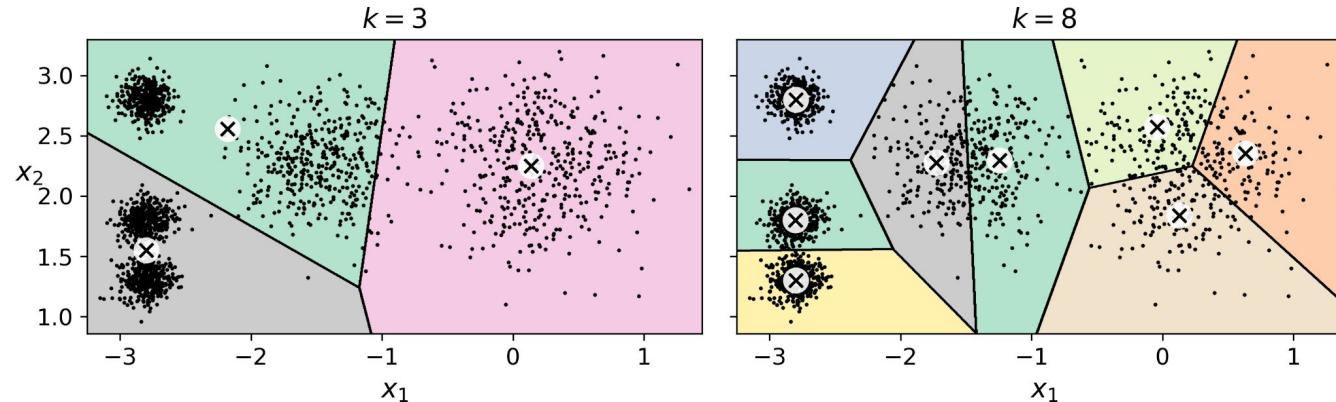
tends to select centroids that are distant from one another

Guarantee: Let C be the solution returned by k -Means++ and let C^* be the optimal solution. Then,
 $E[Cost(C)] \leq O(\log k) \cdot Cost(C^*)$

K-Means Clustering: How Many Clusters?

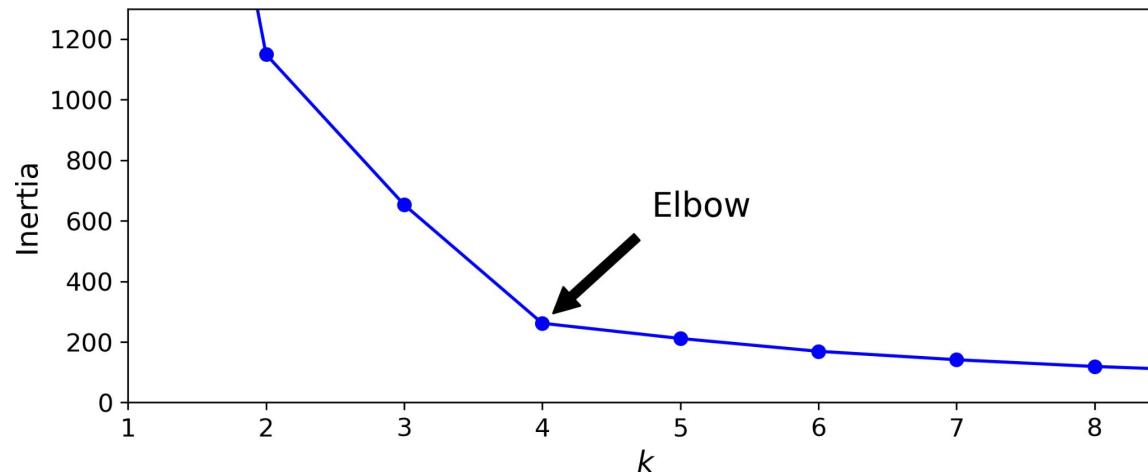
Key input parameter k - how many clusters?

- k too low → “smearing” of clusters that should not be merged.
- k too high → “over-clustering” of the data into many small, similar Clusters.



K-Means Clustering: How Many Clusters?

- The *inertia* is not a good performance metric when trying to choose k since it keeps getting lower as we increase k .
- The inertia drops very quickly as we increase k up to 4, but then it decreases much more slowly as we keep increasing k .



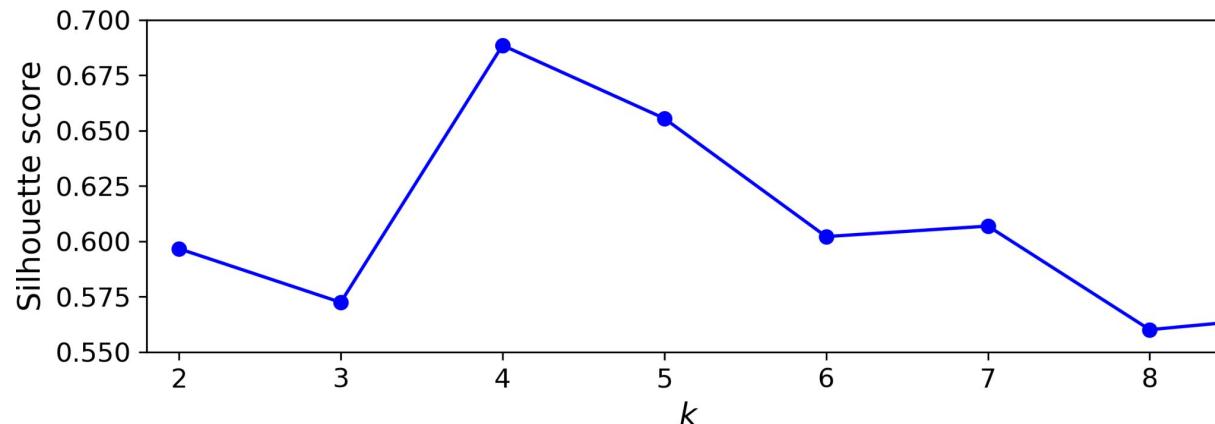
K-Means Clustering: How Many Clusters?

More precise way to use *silhouette score*, which is the mean *silhouette coefficient* over all the instances.

- An instance's *silhouette coefficient* is equal to $(b-a) / \max(a, b)$ where
 - a is the mean distance to the other instances in the same cluster (it is the mean intra-cluster distance), and
 - b is the mean nearest-cluster distance, that is the mean distance to the instances of the next closest cluster (defined as the one that minimizes b , excluding the instance's own cluster).

K-Means Clustering: How Many Clusters?

- The *silhouette coefficient* can vary between -1 and +1:
 - a coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters,
 - a coefficient close to 0 means that it is close to a cluster boundary, and
 - a coefficient close to -1 means that the instance may have been assigned to the wrong cluster.



K-Means Clustering

Advantages

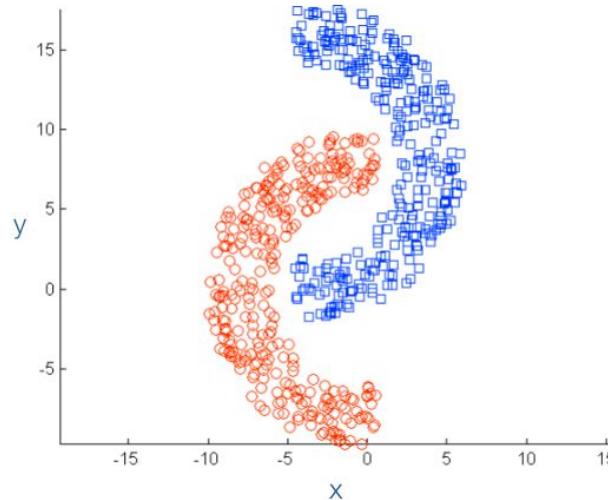
- Fast (for small dataset), easy to implement.
- “Good enough” in a wide variety of tasks and domains.

Disadvantages

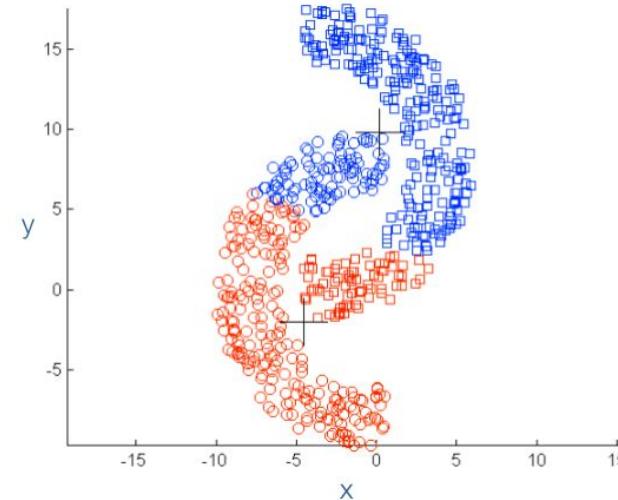
- Must pre-specify number of clusters k .
- Lloyd's algorithm is highly sensitive to choice of initial clusters.
- Assumes that each cluster is spherical in shape and data examples are largely concentrated near its centroid.
- Traditional objective can give undue influence to outliers.
- Iterative process can lead to empty clusters, particularly for higher values of k .

K-Means Clustering: Limitations

Example: *k*-Means assumes that clusters are spherical in shape and data examples are largely concentrated near its centroid.



Original “correct” groups in
the data



Clusters identified by
k-means for $k=2$

End of the Course
