

Course Logistics and Introduction

CS771: Introduction to Machine Learning
Piyush Rai

Course Logistics

- Course Name: Introduction to Machine Learning – CS771
 - An introductory course – supposed to be your first intro to the subject
- Usually 3 lectures every week in form of videos (hosted on mooKIT)
 - Think can these as Mon/Wed/Fri lectures in the usual classroom setting
 - mooKIT URL: <https://hello.iitk.ac.in/cs771a/> (CC id and password to be used for login)
- An additional discussion session every Monday, 6pm-7pm (via YouTube Live)
- All material will be posted on the mooKIT page for the course
- Q/A and announcements on Piazza. Please sign up



Course Team



Soumya Banerjee
soumyab@cse.iitk.ac.in



Shivam Bansal
sbansal@cse.iitk.ac.in



Dhanajit Brahma
dhanajit@cse.iitk.ac.in



Amit Chandak
amitch@cse.iitk.ac.in



Neeraj Matiyali
neermat@cse.iitk.ac.in



Pratik Mazumder
pratikm@cse.iitk.ac.in



Course Team



Avik Pal
avikpal@cse.iitk.ac.in



Niravkumar Panchal
nirav@cse.iitk.ac.in



Hemant Sadana
soumyab@cse.iitk.ac.in



Rahul Sharma
rsharma@cse.iitk.ac.in



Piyush Rai
piyush@cse.iitk.ac.in



Workload and Grading Policy

- 4 homework assignments (theory + programming) worth 50%
 - Theory part: Derivations/analysis
 - Programming part: Implement/use ML algos, analysis of results. Must be done in Python (learn if not already familiar)
 - Must be typeset in LaTeX (learn if not already familiar)
 - To be submitted via **Gradescope** (login details will be provided)
- Quizzes and exams (mid-sem and end-sem) worth 50%
 - Will be held online – details later
 - Exact break-up of individual components will be announced in a few days

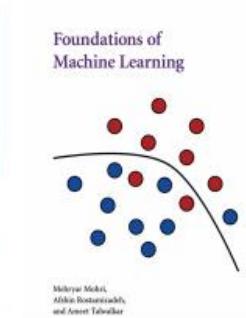
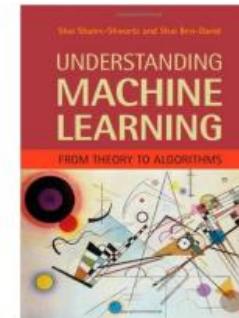
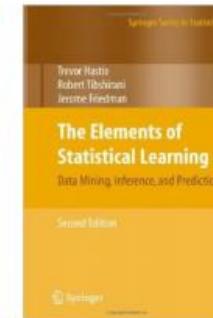
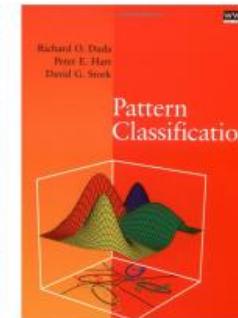
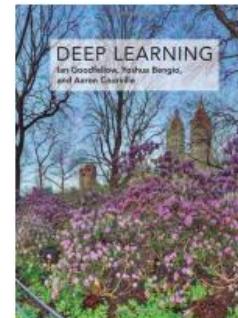
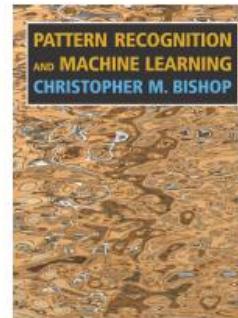
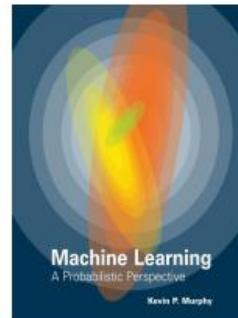
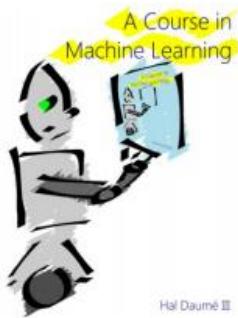
Python: <https://www.geeksforgeeks.org/python-programming-language/>

LaTeX: www.sharelatex.com/blog/latex-guides/beginners-tutorial.html
www.overleaf.com/learn/latex/Tutorials



Textbook and References

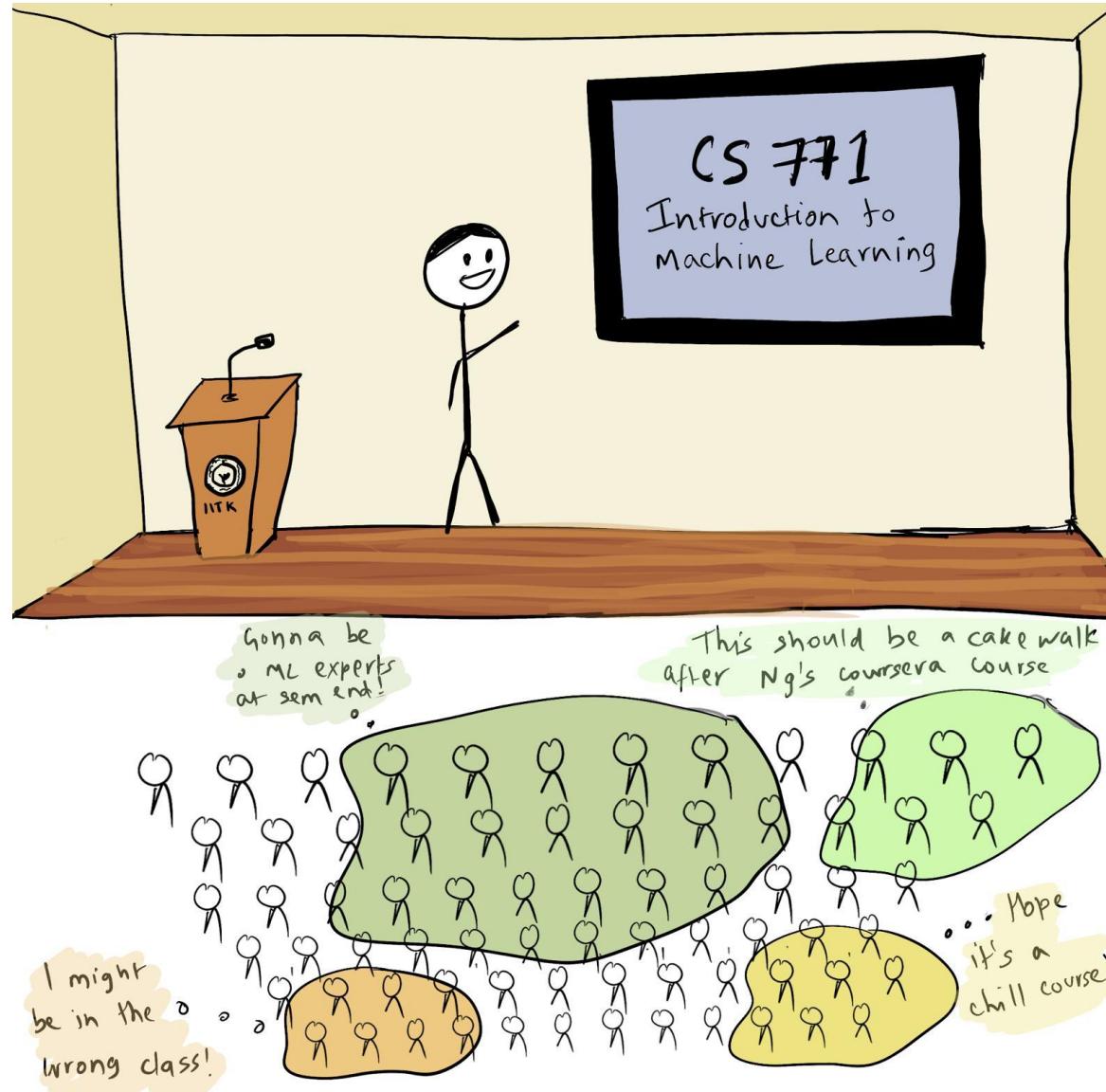
- Many excellent texts but none “required”. Some include:



- Different books might vary in terms of
 - Set of topics covered
 - Flavor (e.g., classical statistics, deep learning, probabilistic/Bayesian, theory)
 - Terminology and notation (beware of this especially)
- We will provide you the reading material from the relevant sources



Course Goals



Course Real Goals..

- Introduction to the foundations of machine learning models and algos
- Focus on developing the ability to
 - Understand the underlying principles behind ML models and algos
 - Understand how to implement and evaluate them
 - Understand/develop intuition on choosing the right ML model/algo for your problem
- (Hopefully) inspire you to work on and learn more about ML
- Not an introduction to popular software frameworks and libraries, such as scikit-learn, PyTorch, Tensorflow, etc
 - Can explore once you have some understanding of various ML techniques

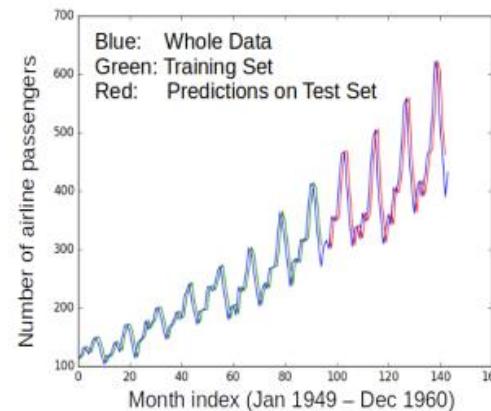


Introduction to Machine Learning



Machine Learning (ML)

- Designing algorithms that ingest data and learn a model of the data
- The learned model can be used to
 - Detect patterns/structures/themes/trends etc. in the data
 - Make predictions about future data and make decisions



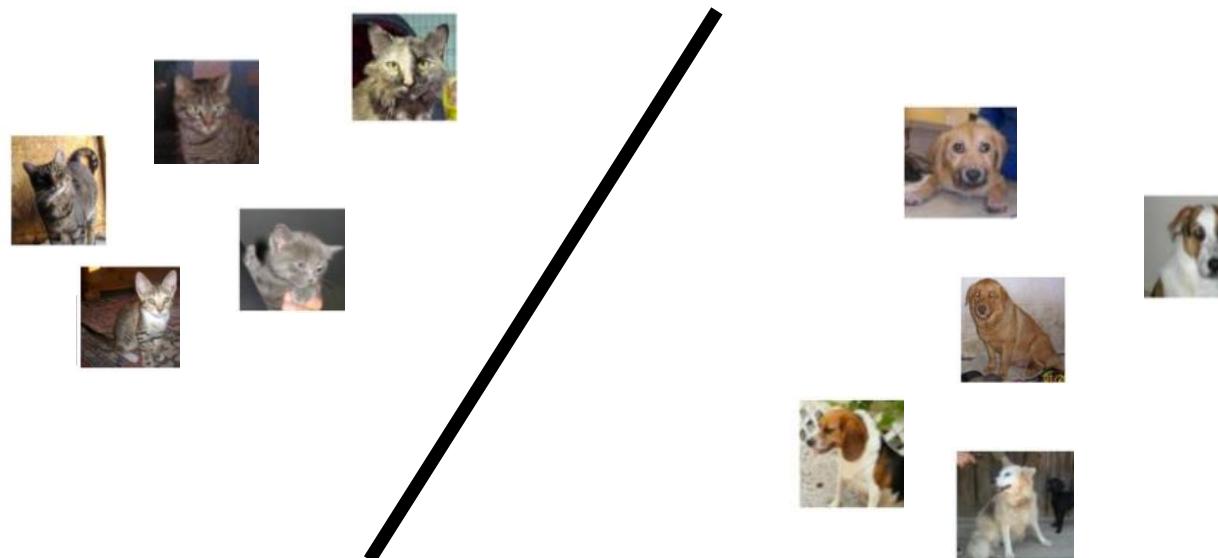
- Modern ML algorithms are heavily “data-driven”
 - No need to pre-define and hard-code all the rules (infeasible/impossible anyway)
 - The rules are not “static”; can adapt as the ML algo ingests more and more data



ML: From What It Does to How It Does It?

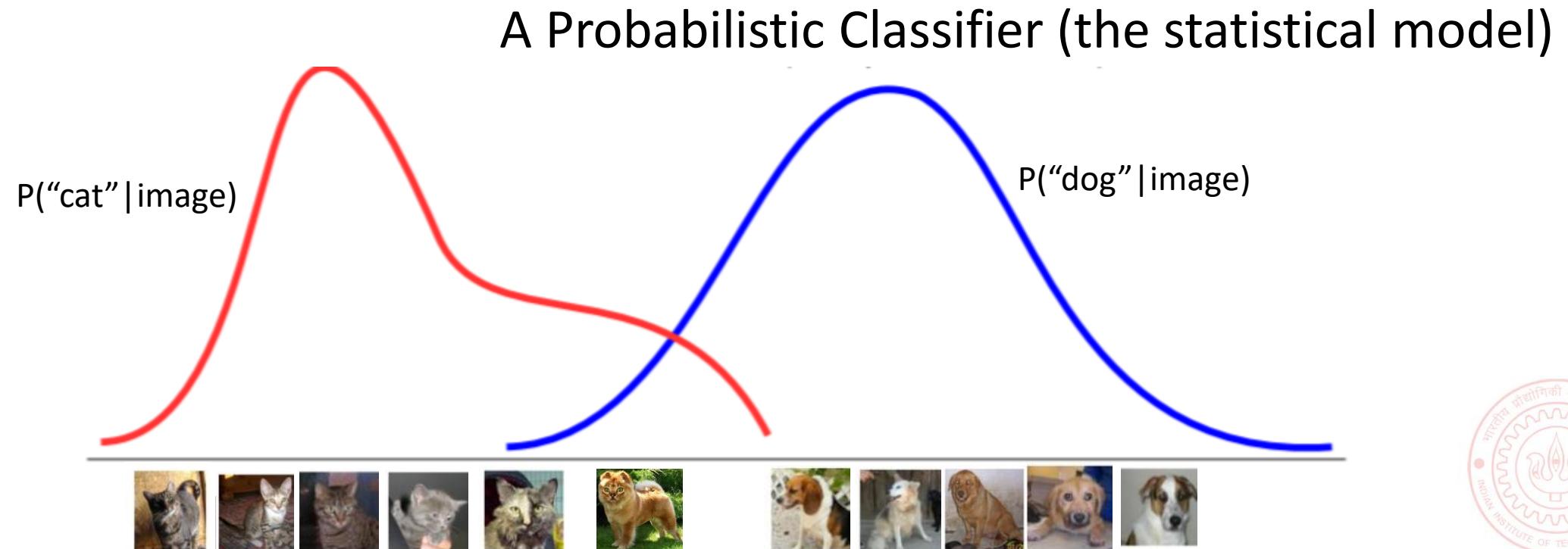
- ML enables intelligent systems to be **data-driven** rather than **rule-driven**
- How: By supplying **training data** and building **statistical models** of data
- Pictorial illustration of an ML model for binary classification:

A Linear Classifier (the statistical model)



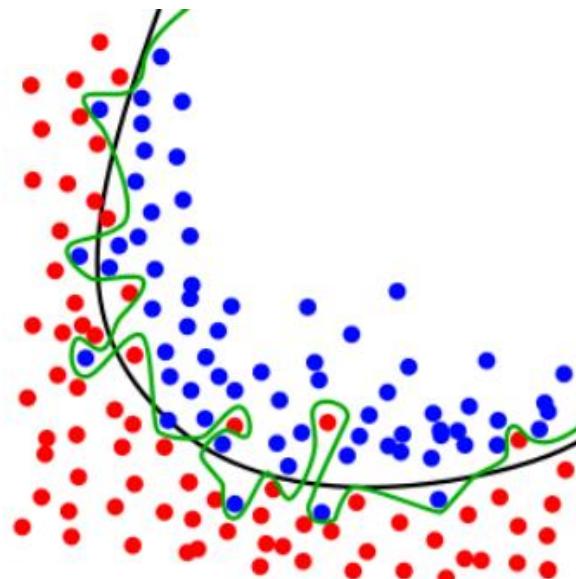
ML: From What It Does to How It Does It?

- ML enables intelligent systems to be **data-driven** rather than **rule-driven**
- How: By supplying **training data** and building **statistical models** of data
- Pictorial illustration of an ML model for binary classification:



Overfitting = Bad ML

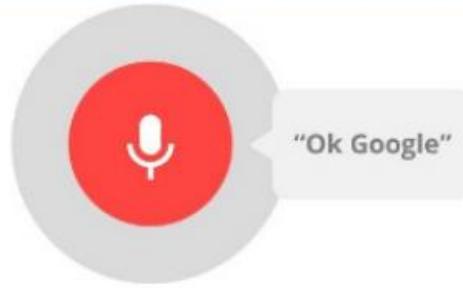
- Doing perfectly on training data is not good enough



- A good ML model must generalize well on unseen (test data)
- Simpler models should be preferred over more complex ones!



ML Applications Abound..



Key Enablers for Modern ML

- Availability of large amounts of data to train ML models

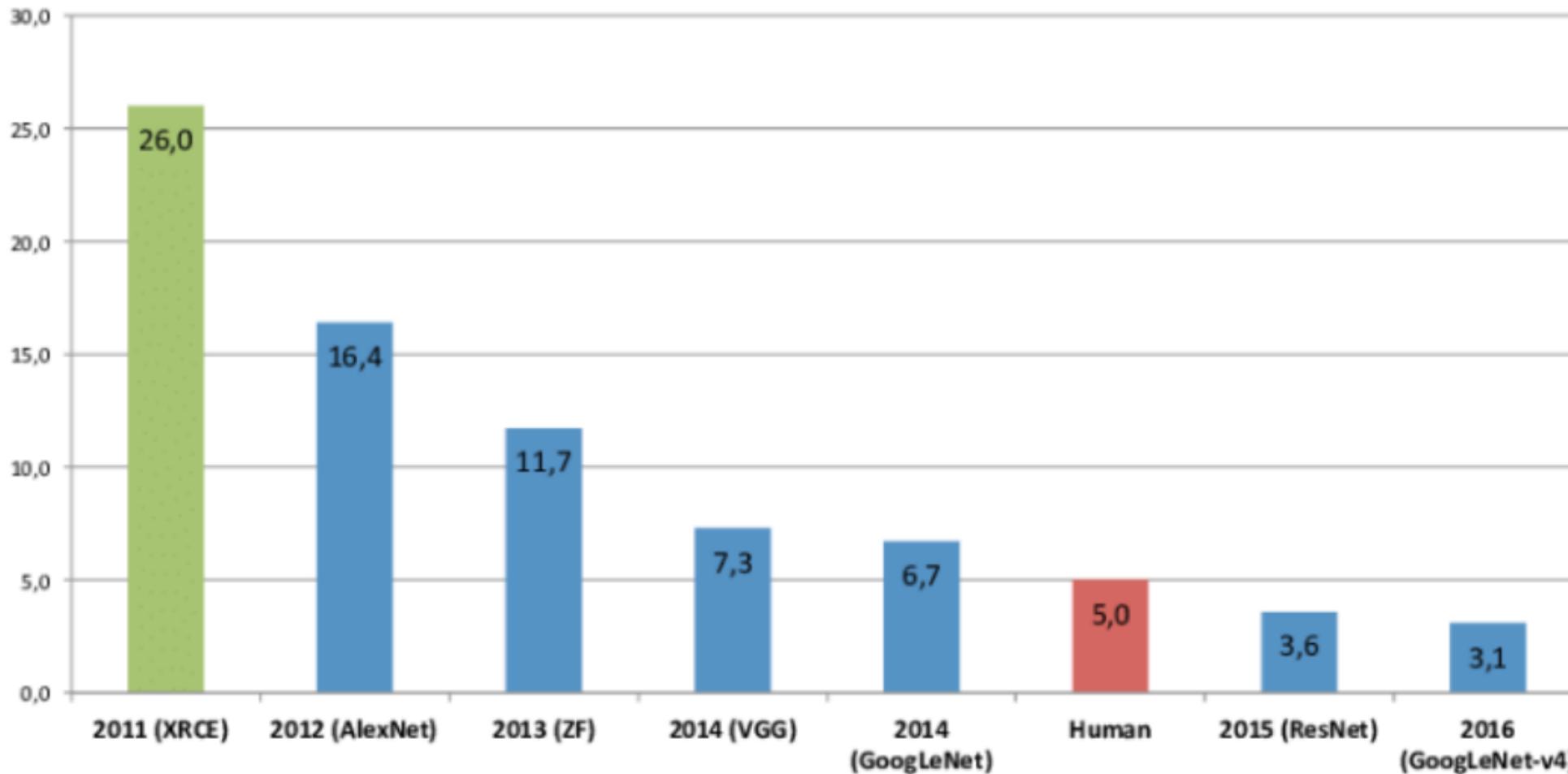


- Increased computing power (e.g., GPUs)



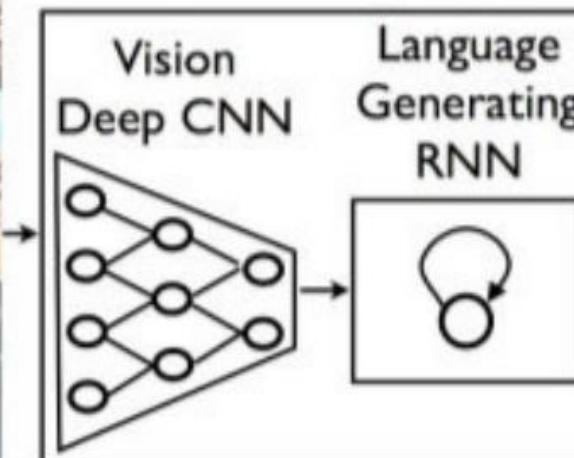
ML: Some Success Stories

ML algorithms can learn to recognize images better than humans!



ML: Some Success Stories

ML algorithms can learn to generate captions for images



**A group of people
shopping at an
outdoor market.**

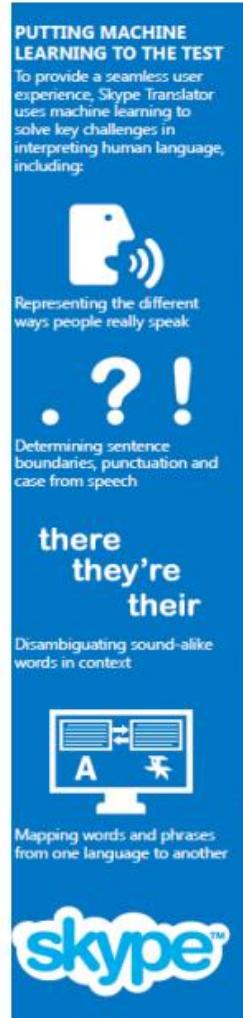
**There are many
vegetables at the
fruit stand.**

<http://arxiv.org/abs/1411.4555> “Show and Tell: A Neural Image Caption Generator”

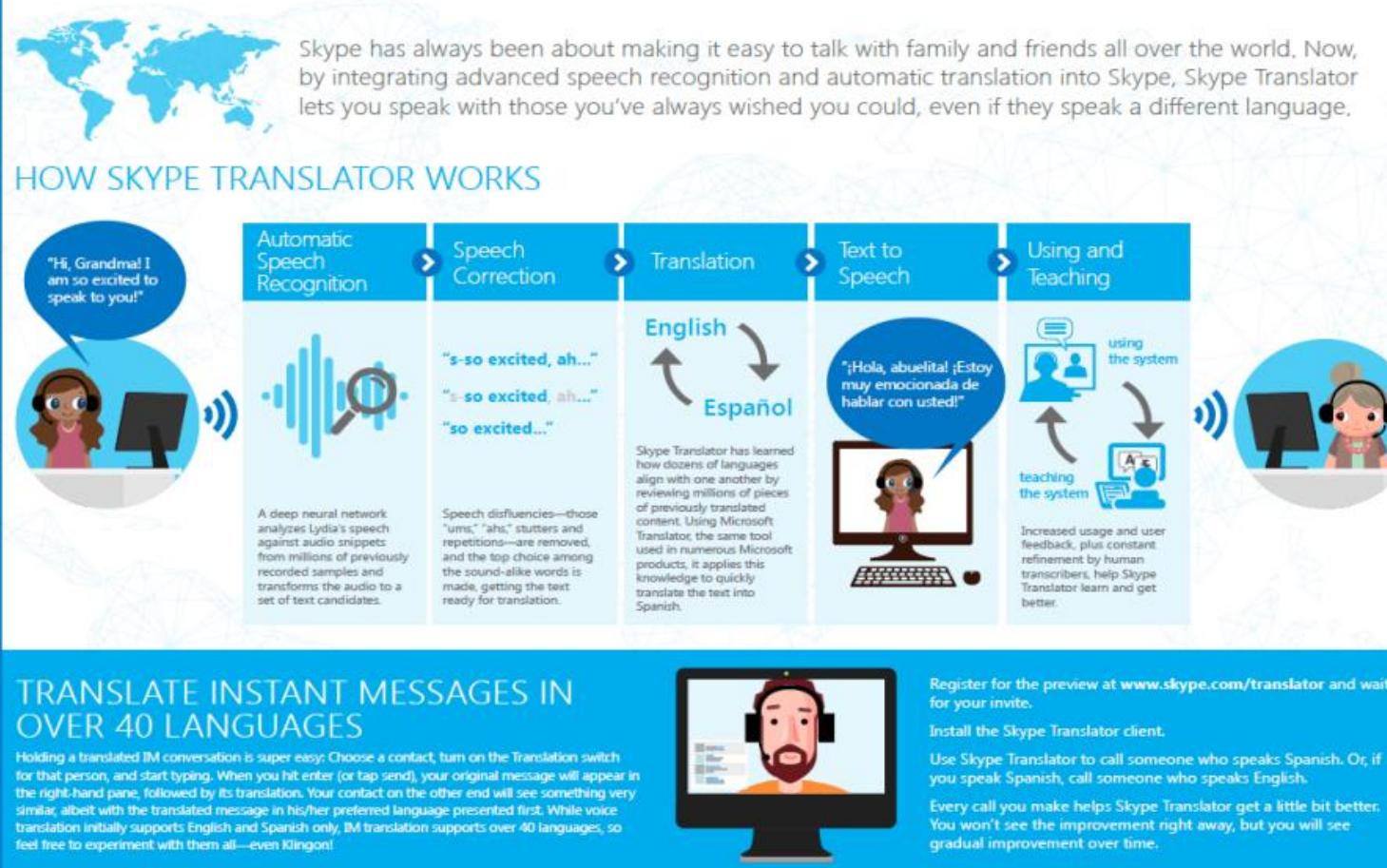


ML: Some Success Stories

ML algorithms can learn to translate speech in **real time**



NOW YOU'RE SPEAKING MY LANGUAGE (LITERALLY)



ML: Some Success Stories

■ Automatic Program Correction

```

1 #include<stdio.h>
2 int main(){
3     int a;
4     scanf("%d", a);
5     printf("ans=%d",
6            a+10);
7     return 0;
8 }
```

```

1 #include<stdio.h>
2 int main(){
3     int a;
4     scanf("%d", &a );
5     printf("ans=%d",
6            a+10);
7     return 0;
8 }
```

Figure 1: Left: erroneous program, Right: fix by TRACER. The compiler message read: *Line-4, Column-9: warning: format '%d' expects argument of type 'int *', but argument 2 has type 'int'*.

```

1 #include<stdio.h>
2 int main(){
3     int x,x1,d;
4     // ...
5     d=(x-x1)(x-x1);
6     return d;
7 }
```

```

1 #include<stdio.h>
2 int main(){
3     int x,x1,d;
4     // ...
5     d=(x-x1)*(x-x1);
6     return d;
7 }
```

Figure 2: Left: erroneous program, Right: fix by TRACER. The compiler message read: *Line-5, Column-11: error: called object type 'int' is not a function or function pointer*.



ML: Some Success Stories

- ML based colorimetry for water quality assessment
- Take uncontaminated water sample
- Spike it with known concentration of various compounds (e.g., lead, iron, fluoride, etc)
- Dip a test strip (one square to measure each compound) in the contaminated water for some time.
- Take a picture of the strip using a phone camera to capture how the colors have changed
- Train an ML model to predict concentration levels of various compounds based on color levels in the images



Good ML Systems Should be Fair and Unbiased

- Good ML should not just be about getting high accuracies
- Should also ensure that the ML models are fair and unbiased



An image captioning system should not always assume a specific gender in examples like the above



Don't want a self-driving car that is more likely to hit black people than white people



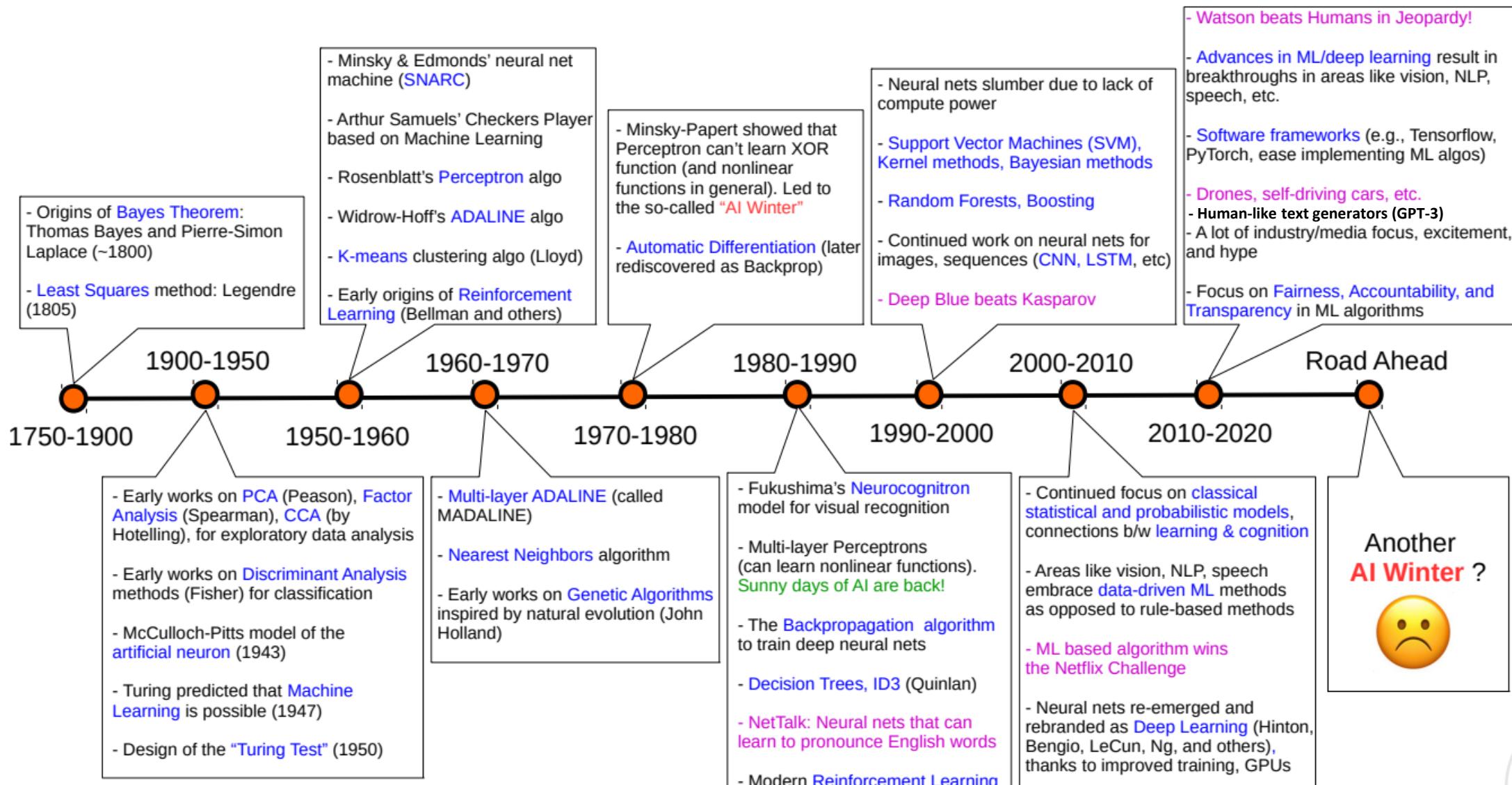
Criminals?

Not Criminals?

Don't want a predictive policing system that predicts criminality using facial features

- A lot of recent focus on Fairness and Transparency of ML systems

Looking Back Before We Start: History of ML



Next Class

- Various Flavors of ML problems
- Data and features
- Basic mathematical operations on data and features



Warming-up to Machine Learning, Data and Features

CS771: Introduction to Machine Learning
Piyush Rai

Plan for today

- Types of ML problems
- Typical workflow of ML problems
- Various perspectives of ML problems
- Data and Features
- Some basic operations of data and features



Keep in mind: ML is like an exam

- It's the performance on the D-day which matters
- In an exam, our success is measured based on how well we did on the questions in the test (not on the questions we practiced on)
- Likewise, in ML, success of the learned model is measured based on how well it predicts/fits the future **test data** (not the training data)

In Machine Learning, **generalization** performance
on the test data matters

Plus, of course,
issues such as
fairness



A Loose Taxonomy of ML

4



Learning using **labeled** data

Some examples of supervised learning problems

- Classification
- Regression
- Ranking

Supervised Learning

Learning using **unlabeled** data

“Labeled” means, during training, for each input, the corresponding output is available (i.e., the machine learner is explicitly told that a cat image is of a cat)

Unsupervised Learning

Some examples of unsupervised learning problems

- Clustering
- Dimensionality Reduction
- Unsupervised Probability Density Estimation

Machine Learning

RL doesn't use “labeled” or “unlabeled” data in the traditional sense! In RL, an agent learns via its interactions with an environment

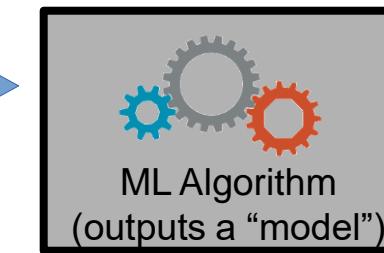
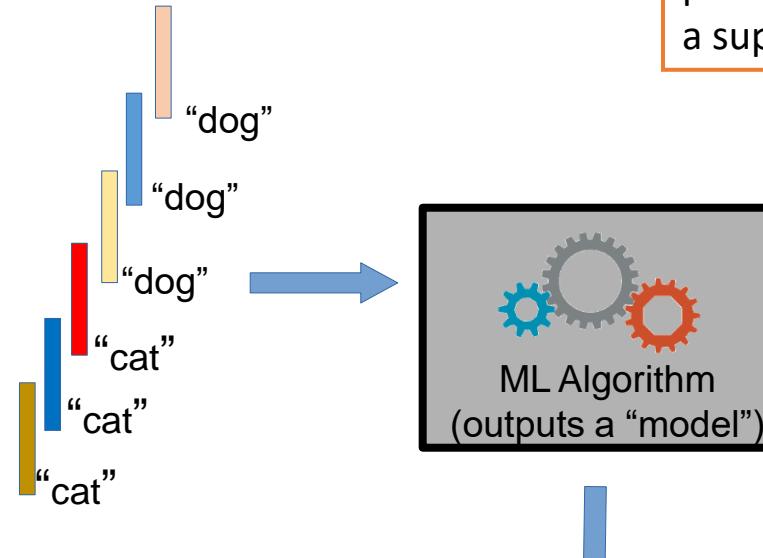
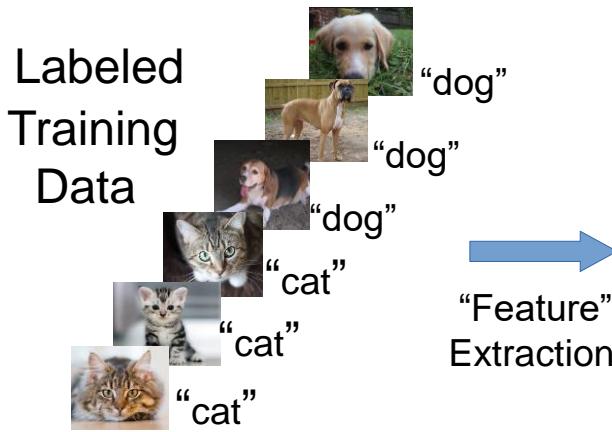
Reinforcement Learning

Many other specialized flavors of ML also exist, some of which include

- Semi-supervised Learning
- Active Learning
- Transfer Learning
- Multitask Learning
- Imitation Learning (somewhat related to RL)
- Zero-Shot Learning
- Few-Shot Learning
- Continual learning

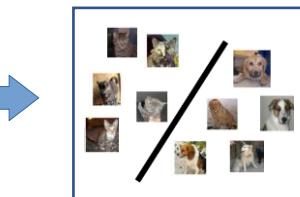
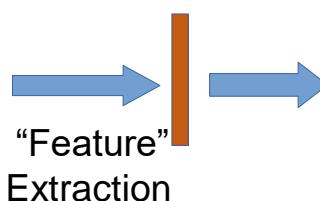
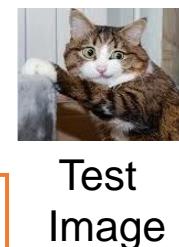


A Typical Supervised Learning Workflow



Feature extraction converts raw inputs to a **numeric representation** that the ML algo can understand and work with. More on feature extraction later.

Indeed. **Deep Learning** algos do precisely that! (**feature + model learning**). More on Deep Learning later.



Note: This example is for the problem of **binary classification**, a supervised learning problem



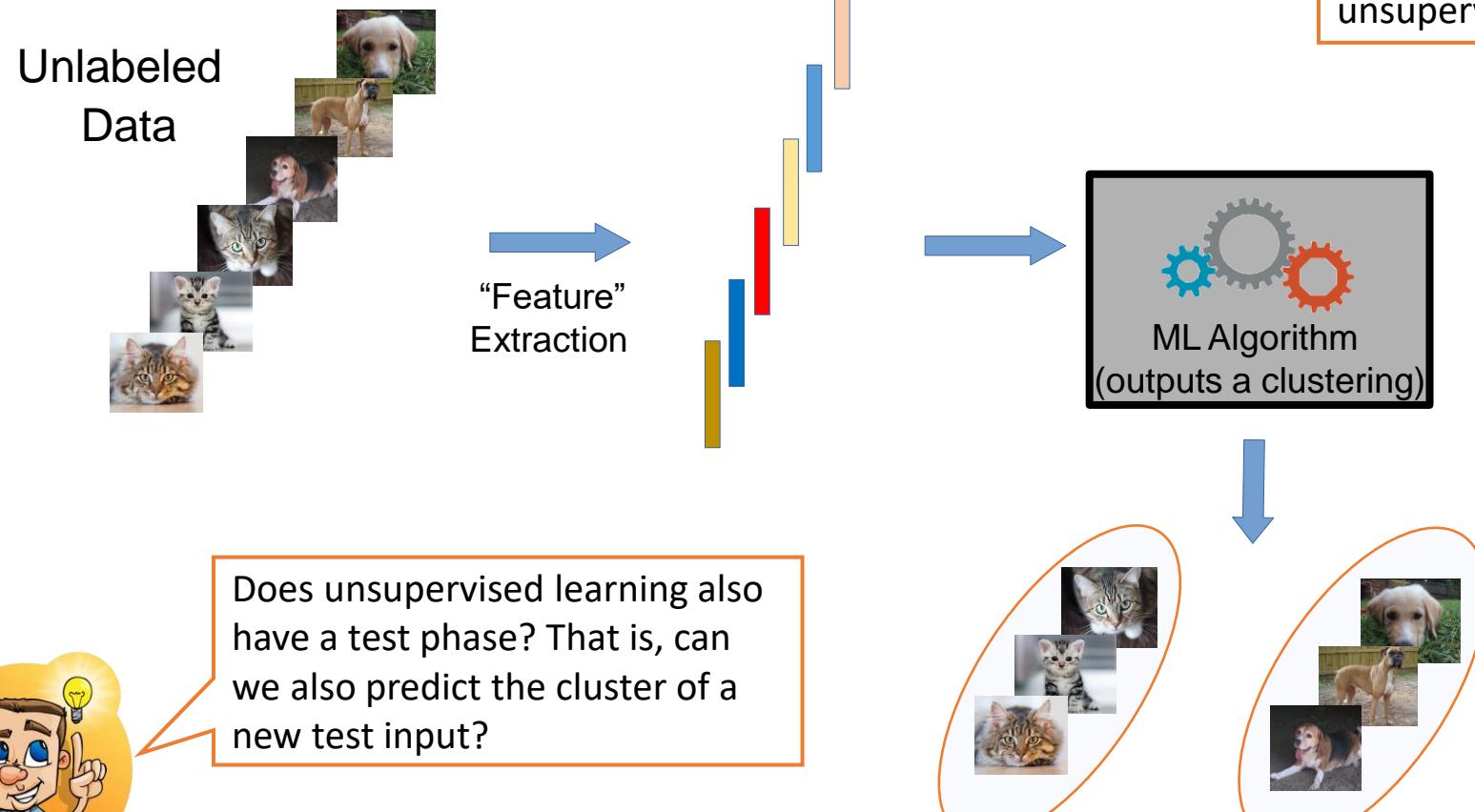
Is feature extraction done “manually” as a pre-processing step before the ML algo starts working? Can’t we “automate” this part? Can’t we “learn” good features directly from raw inputs?



Predicted Label
(cat/dog)



A Typical Unsupervised Learning Workflow



Does unsupervised learning also have a test phase? That is, can we also predict the cluster of a new test input?

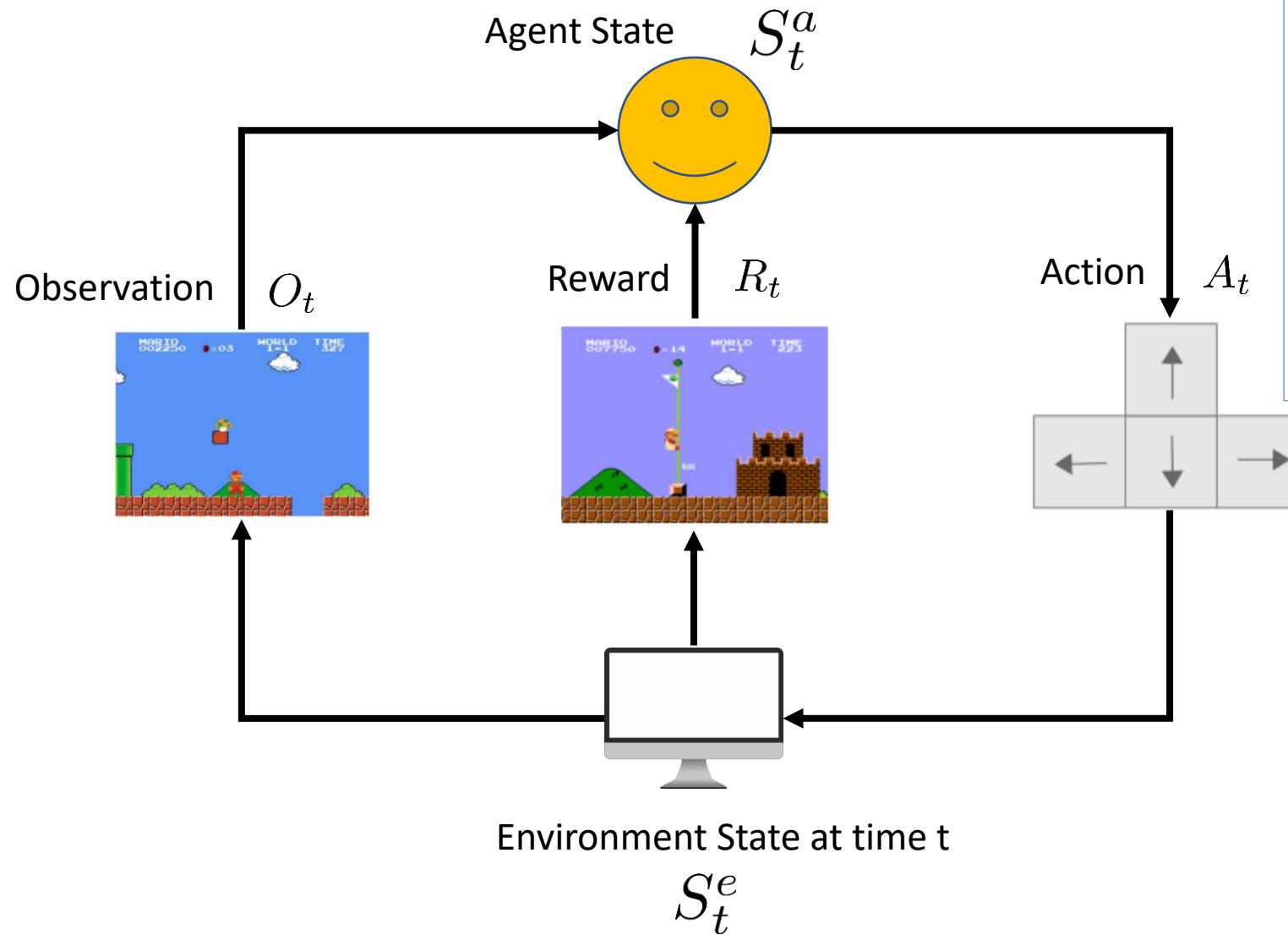
Note: This example is for the problem of **data clustering**, an unsupervised learning problem



Yes. In this example, given a new “test” cat/dog image, we can assign it to the cluster with closer centroid



A Typical Reinforcement Learning Workflow



Wish to teach an agent optimal policy for some task

Agent does the following repeatedly

- Senses/observes the environment
- Takes an action based on its current policy
- Receives a reward for that action
- Updates its policy

Agent's goal is to maximize its overall reward

There IS supervision, not explicit
(as in Supervised Learning) but
rather implicit (feedback based)



ML: Some Perspectives

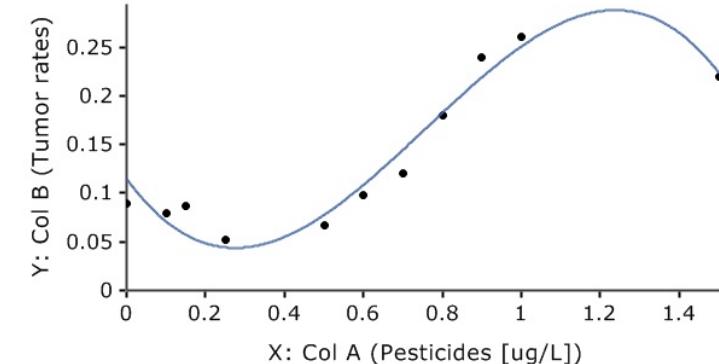
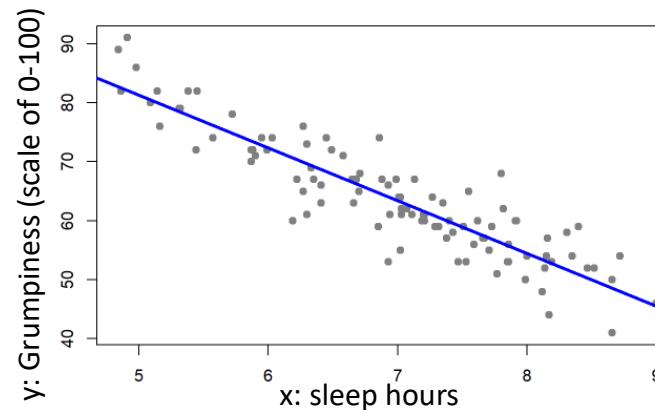


Geometric Perspective

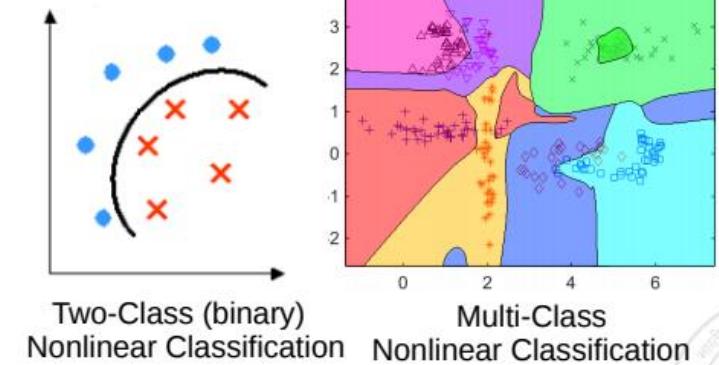
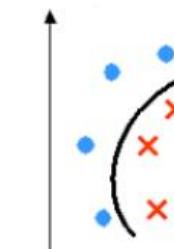
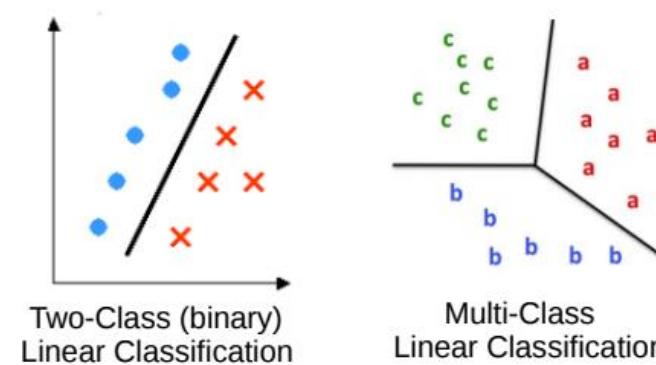
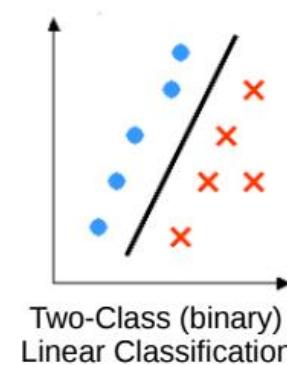
Recall that feature extraction converts inputs into a **numeric representation**

- Basic fact: Inputs in ML problems can often be represented as **points or vectors** in some vector space
- Doing ML on such data can thus be seen from a geometric view

Regression: A supervised learning problem. Goal is to model the relationship between input (x) and real-valued output (y). This is akin to a **line or curve fitting** problem

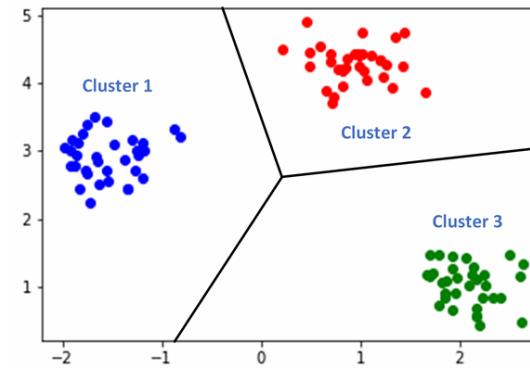


Classification: A supervised learning problem. Goal is to learn a to predict which of the two or more classes an input belongs to. Akin to learning **linear/nonlinear separator** for the inputs



Geometric Perspective

Clustering: An unsupervised learning problem. Goal is to group inputs in a few clusters **based on their similarities with each other**



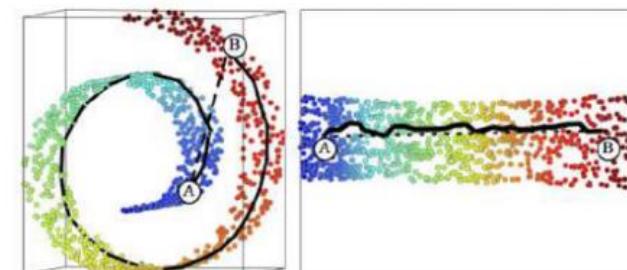
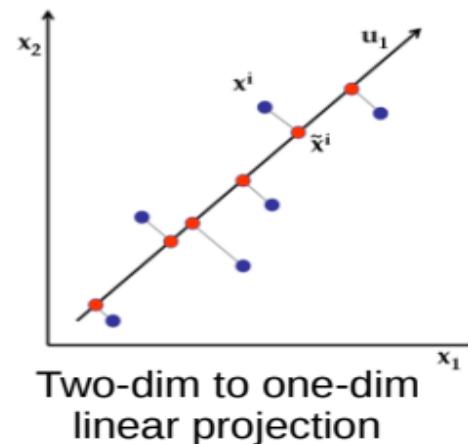
Clustering looks like classification to me. Is there any difference?



Yes. In clustering, we don't know the labels. Goal is to separate them without any labeled "supervision"



Dimensionality Reduction: An unsupervised learning problem. Goal is to **compress the size** of each input without losing much information present in the data

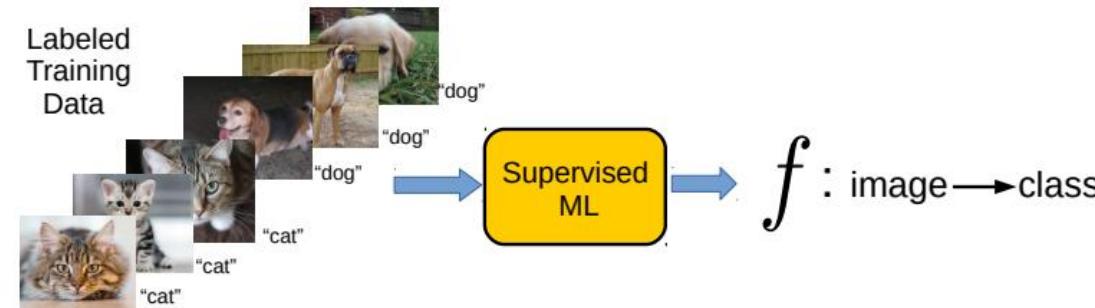


Three-dim to two-dim nonlinear projection (a.k.a. manifold learning)



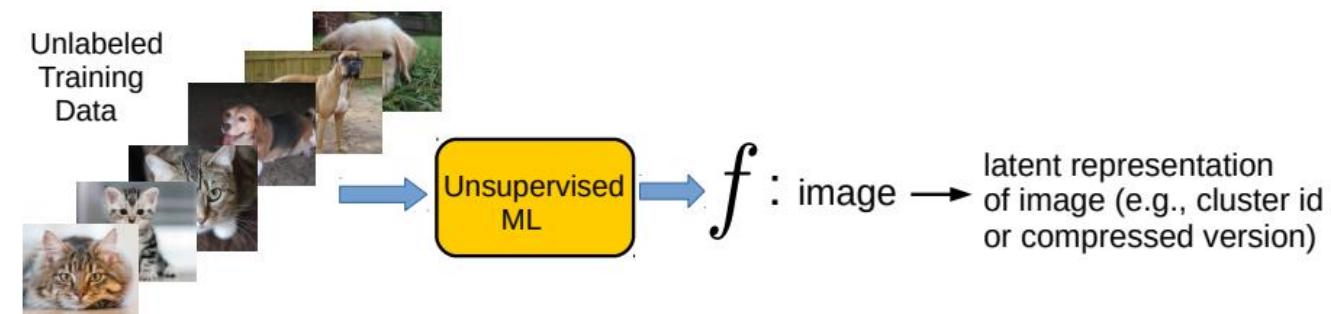
Perspective as function approximation

- Supervised Learning (“predict output given input”) can be usually thought of as learning a **function f** that maps each input to the corresponding output



- Unsupervised Learning (“model/compress inputs”) can also be usually thought of as learning a **function f** that maps each input to a compact representation

Harder since we don't know the labels in this case

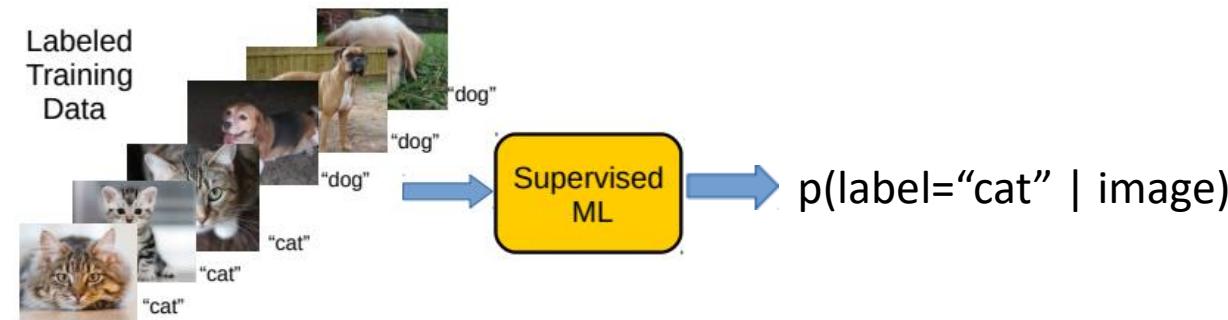


- Reinforcement Learning can also be seen as doing function approximation



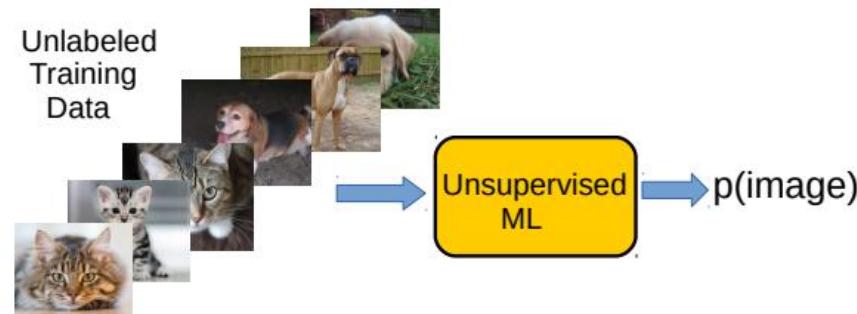
Perspective as probability estimation

- Supervised Learning (“predict output given input”) can be thought of as estimating the **conditional probability** of each possible output given an input



- Unsupervised Learning (“model/compress inputs”) can be thought of as estimating the **probability density** of the inputs

Harder since we don't know the labels in this case



Don't worry if this doesn't make much sense as of now 😊 But the basic idea is to learn the underlying data distribution using the unlabeled inputs; many ways to do this as we will see later



- Reinforcement Learning can also be seen as estimating probability densities

Data and Features



Data and Features

Features represent semantics of the inputs. Being able to extract good features is key to the success of ML algos



- ML algos require a numeric **feature representation** of the inputs
- Features can be obtained using one of the two approaches
 - Approach 1: Extracting/constructing features manually from raw inputs
 - Approach 2: Learning the features from raw inputs
- Approach 1 is what we will focus on primarily for now
- Approach 2 is what is followed in **Deep Learning** algorithms (will see later)
- Approach 1 is not as powerful as Approach 2 but still used widely



Example: Feature Extraction for Text Data

- Consider some text data consisting of the following sentences:

- John likes to watch movies
- Mary likes movies too
- John also likes football

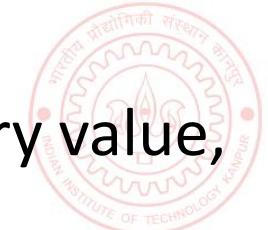
BoW is just one of the many ways of doing feature extraction for text data. Not the most optimal one, and has various flaws (can you think of some?), but often works reasonably well



- Want to construct a **feature representation** for these sentences
- Here is a “**bag-of-words**” (BoW) feature representation of these sentences

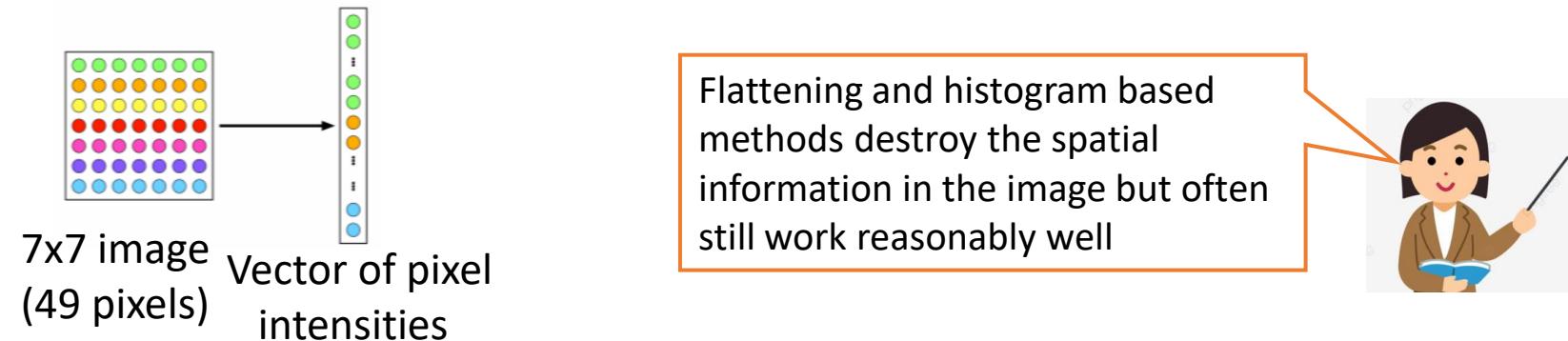
$$\begin{array}{l}
 \text{Sentence 1} \quad \left(\begin{matrix} \text{John} & \text{likes} & \text{to} & \text{watch} & \text{movies} & \text{Mary} & \text{too} & \text{also} & \text{football} \end{matrix} \right) \\
 \text{Sentence 2} \quad \left(\begin{matrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{matrix} \right) \\
 \text{Sentence 3} \quad \left(\begin{matrix} 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{matrix} \right) \\
 \text{Sentence 4} \quad \left(\begin{matrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{matrix} \right)
 \end{array}$$

- Each sentence is now represented as a **binary vector** (each feature is a binary value, denoting presence or absence of a word). BoW is also called “**unigram**” rep.

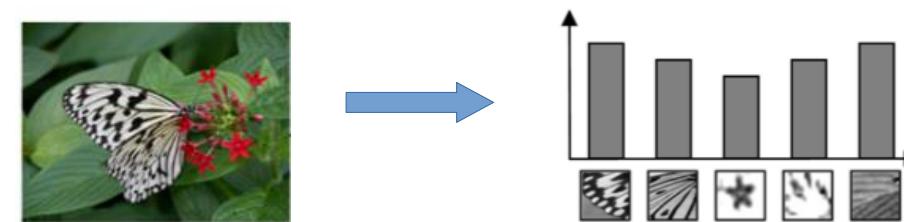


Example: Feature Extraction for Image Data

- A very simple feature extraction approach for image data is **flattening**



- Histogram** of visual patterns is another popular feature extr. method for images



- Many other manual feature extraction techniques developed in computer vision and image processing communities (SIFT, HoG, and others)



Feature Selection

- Not all the extracted features may be relevant for learning the model (some may even confuse the learner)
- Feature selection** (a step after feature extraction) can be used to identify the features that matter, and discard the others, for more effective learning

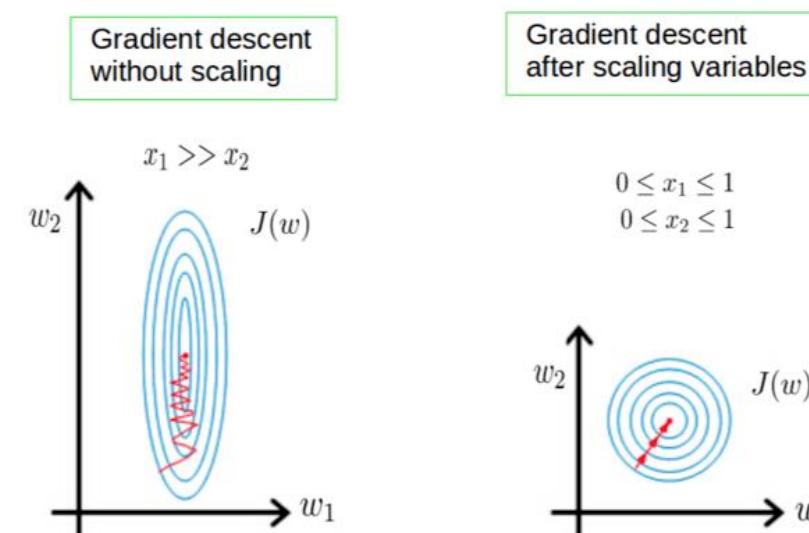
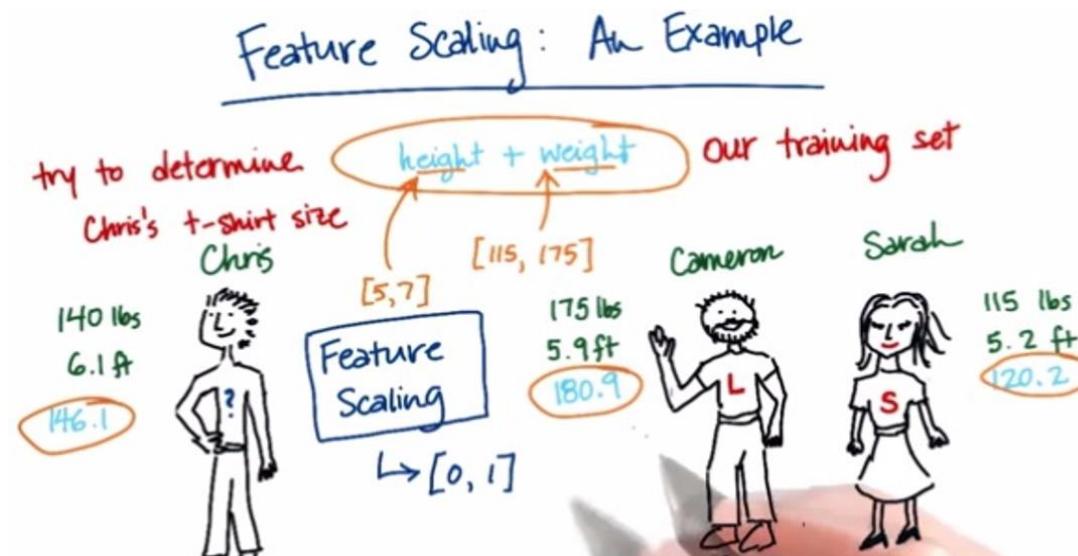


- Many techniques exist – some based on intuition, some based on algorithmic principles (will visit feature selection later)
- More common in supervised learning but can also be done for unsup. learning



Some More Postprocessing: Feature Scaling

- Even after feature selection, the features may not be on the same scale
- This can be problematic when comparing two inputs – features that have larger scales may dominate the result of such comparisons
- Therefore helpful to standardize the features (e.g., by bringing all of them on the same scale such as between 0 to 1)



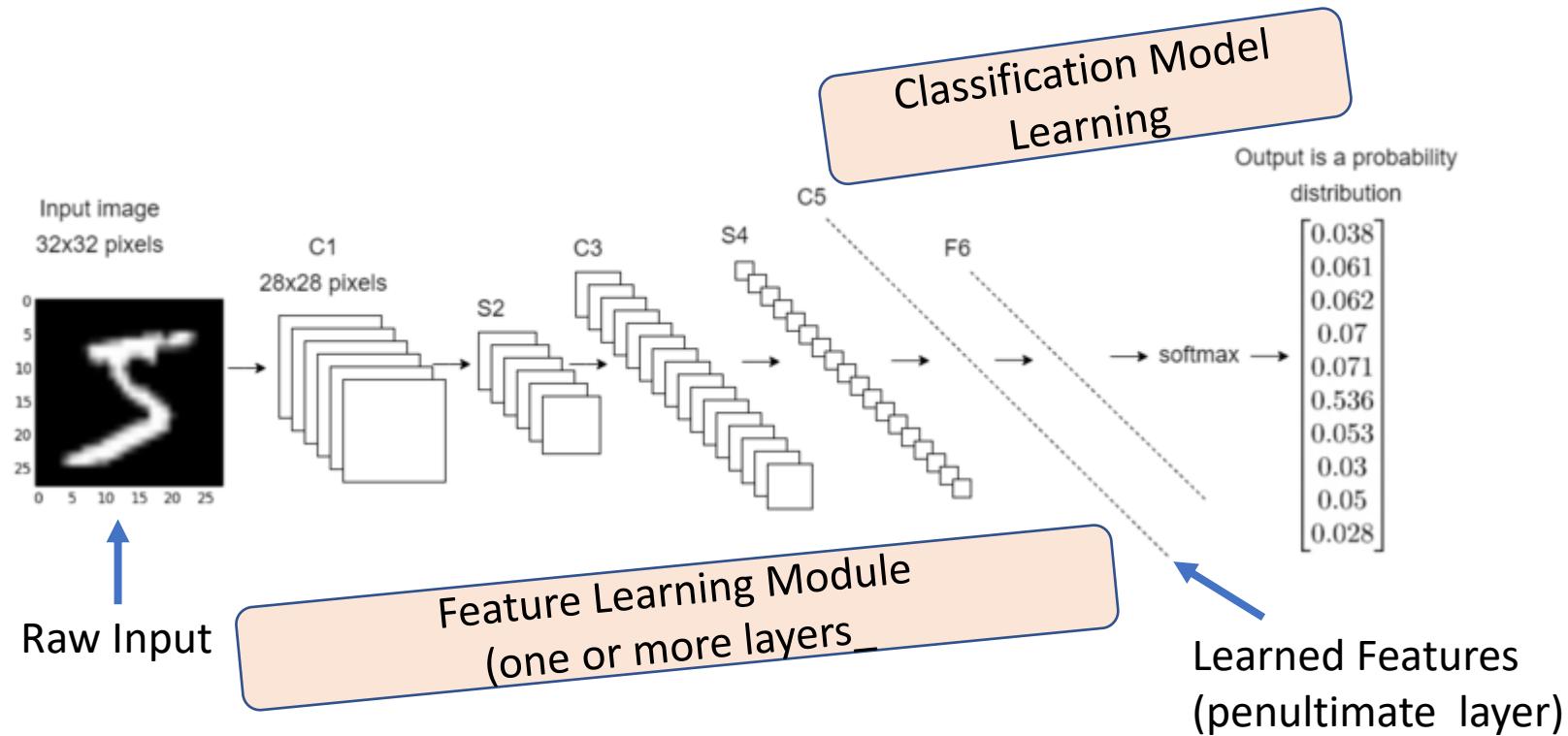
- Also helpful for stabilizing the optimization techniques used in ML algos



Deep Learning: An End-to-End Approach to ML

Deep Learning = ML with **automated feature learning** from the raw inputs

Feature extraction part is automated via the feature learning module



Some Notation/Nomenclature/Convention

- Sup. learning requires training data as N input-output pairs $\{(\mathbf{x}_n, y_n)\}_{n=1}^N$
- Unsupervised learning requires training data as N inputs $\{\mathbf{x}_n\}_{n=1}^N$
- Each input \mathbf{x}_n is (usually) a vector containing the values of the **features** or **attributes** or **covariates** that encode properties of the it represents, e.g.,
 - For a 7×7 image: \mathbf{x}_n can be a 49×1 vector of pixel intensities
- (In sup. Learning) Each y_n is the **output** or **response** or **label** associated with input \mathbf{x}_n (and its value is known for the training inputs)
- Output can be a scalar, a vector of numbers, or even an structured object (more on this later)

RL and other flavors
of ML problems also
use similar notation



Size or length of the input \mathbf{x}_n is
commonly known as **data/input
dimensionality** or **feature dimensionality**



Types of Features and Types of Outputs

- Features as well as outputs can be real-valued, binary, categorical, ordinal, etc.
- **Real-valued:** Pixel intensity, house area, house price, rainfall amount, temperature, etc
- **Binary:** Male/female, adult/non-adult, or any yes/no or present/absent type value
- **Categorical/Discrete:** Zipcode, blood-group, or any “one from a finite many choices” value
- **Ordinal:** Grade (A/B/C etc.) in a course, or any other type where relative values matter
- Often, the features can be of mixed types (some real, some categorical, some ordinal, etc.)



Some Basic Operations of Inputs

- Assume each input feature vector $\mathbf{x}_n \in R^D$ to of size D

- Given N inputs $\{\mathbf{x}_n\}_{n=1}^N$, their average or mean can be computed as

$$\boldsymbol{\mu} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n$$

What does such a
“mean” represent?



If inputs are all cat images,
mean vector would represents
what an “average” cat looks like



- Can compute the Euclidean distance between any pair of inputs \mathbf{x}_n and \mathbf{x}_m

$$d(\mathbf{x}_n, \mathbf{x}_m) = \|\mathbf{x}_n - \mathbf{x}_m\| = \sqrt{(\mathbf{x}_n - \mathbf{x}_m)^\top (\mathbf{x}_n - \mathbf{x}_m)} = \sqrt{\sum_{d=1}^D (x_{nd} - x_{md})^2}$$

- .. or Euclidean distance between an input \mathbf{x}_n and the mean $\boldsymbol{\mu}$ of all inputs

- .. and various other operations that we will look at later..



Next Class

- Introduction to Supervised Learning
- A simple Supervised Learning algorithm based on computing distances



Getting Started with Supervised Learning, Learning by Computing Distances (1)

CS771: Introduction to Machine Learning
Piyush Rai

Supervised Learning

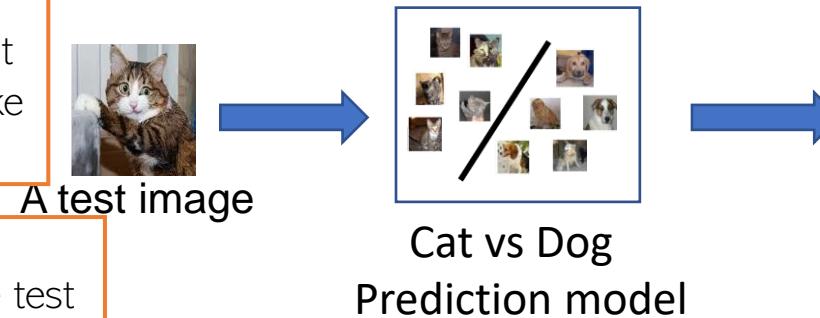


Important: In ML (not just sup. learning but also unsup. and RL), training and test datasets should be “similar” (we don’t like “out-of-syllabus” questions in exams ☺)



In the above example, it means that we can't have test data with BnW images or sketches of cats and dogs

More formally, the train and test data distributions should be the same



Predicted Label
(cat/dog)

Does it mean ML is useless if this assumption is violated?



Of course not. ☺ Many ML techniques exist to handle such situations (a bit advanced but will touch upon those later)

Will give you Just the names for now – **domain adaptation, covariate shift, transfer learning, etc**



Some Types of Supervised Learning Problems

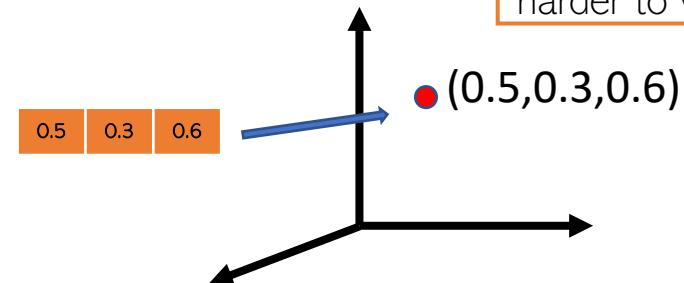
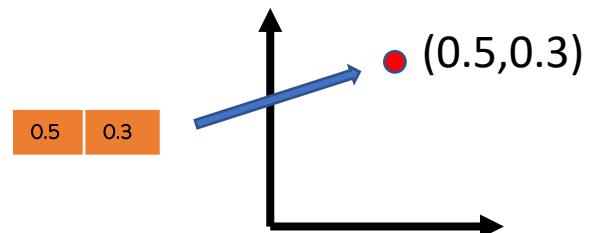
- Consider building an ML module for an e-mail client
- Some tasks that we may want this module to perform
 - Predicting whether an email of spam or normal: [Binary Classification](#)
 - Predicting which of the many folders the email should be sent to: [Multi-class Classification](#)
 - Predicting all the relevant tags for an email: [Tagging](#) or [Multi-label Classification](#)
 - Predicting what's the spam-score of an email: [Regression](#)
 - Predicting which email(s) should be shown at the top: [Ranking](#)
 - Predicting which emails are work/study-related emails: [One-class Classification](#)
- These predictive modeling tasks can be formulated as supervised learning problems
- Today: A very simple supervised learning model for binary/multi-class classification
 - This model doesn't require any fancy maths – just computing means and distances



Some Notation and Conventions

- In ML, inputs are usually represented by vectors
- A vector consists of an array of scalar values
- Geometrically, a vector is just a point in a vector space, e.g.,
 - A length 2 vector is a point in 2-dim vector space
 - A length 3 vector is a point in 3-dim vector space

0.5	0.3	0.6	0.1	0.2	0.5	0.9	0.2	0.1	0.5
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----



- Unless specified otherwise
 - Small letters in bold font will denote vectors, e.g., **x**, **a**, **b** etc.
 - Small letters in normal font to denote scalars, e.g. *x*, *a*, *b*, etc
 - Capital letters in bold font will denote matrices (2-dim arrays), e.g., **X**, **A**, **B**, etc



Some Notation and Conventions

- A single vector will be assumed to be of the form $\mathbf{x} = [x_1, x_2, \dots, x_D]$
- Unless specified otherwise, vectors will be assumed to be column vectors
 - So we will assume $\mathbf{x} = [x_1, x_2, \dots, x_D]$ to be a column vector of size $D \times 1$
 - Assuming each element to be real-valued scalar, $\mathbf{x} \in \mathbb{R}^{D \times 1}$ or $\mathbf{x} \in \mathbb{R}^D$ (\mathbb{R} : space of reals)
- If $\mathbf{x} = [x_1, x_2, \dots, x_D]$ is a feature vector representing, say an image, then
 - D denotes the dimensionality of this feature vector (number of features)
 - x_i (a scalar) denotes the value of i^{th} feature in the image
- For denoting multiple vectors, we will use a subscript with each vector, e.g.,
 - N images denoted by N feature vectors $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$, or compactly as $\{\mathbf{x}_n\}_{n=1}^N$
 - The vector \mathbf{x}_n denotes the n^{th} image
 - x_{ni} (a scalar) denotes the i^{th} feature ($i = 1, 2, \dots, D$) of the n^{th} image



Some Basic Operations on Vectors

- Addition/subtraction of two vectors gives another vector of the same size
- The mean μ (average or centroid) of N vectors $\{\mathbf{x}_n\}_{n=1}^N$

$$\mu = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \quad (\text{of the same size as each } \mathbf{x}_n)$$

- The inner/dot product of two vectors $\mathbf{a} \in \mathbb{R}^D$ and $\mathbf{b} \in \mathbb{R}^D$

Assuming both \mathbf{a} and \mathbf{b}
have unit Euclidean norm

$$\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^\top \mathbf{b} = \sum_{i=1}^D a_i b_i \quad (\text{a real-valued number denoting how "similar" } \mathbf{a} \text{ and } \mathbf{b} \text{ are})$$

- For a vector $\mathbf{a} \in \mathbb{R}^D$, its Euclidean norm is defined via its inner product with itself

$$\|\mathbf{a}\|_2 = \sqrt{\mathbf{a}^\top \mathbf{a}} = \sqrt{\sum_{i=1}^d a_i^2}$$

- Also the Euclidean distance of \mathbf{a} from origin
- Note: Euclidean norm is also called L2 norm



Computing Distances

- Euclidean (L2 norm) distance between two vectors $\mathbf{a} \in \mathbb{R}^D$ and $\mathbf{b} \in \mathbb{R}^D$

$$d_2(\mathbf{a}, \mathbf{b}) = \|\mathbf{a} - \mathbf{b}\|_2 = \sqrt{\sum_{i=1}^D (a_i - b_i)^2} = \sqrt{(\mathbf{a} - \mathbf{b})^\top (\mathbf{a} - \mathbf{b})} = \sqrt{\mathbf{a}^\top \mathbf{a} + \mathbf{b}^\top \mathbf{b} - 2\mathbf{a}^\top \mathbf{b}}$$

Sqrt of Inner product of the difference vector!
Another expression in terms of inner products of individual vectors

- Weighted Euclidean distance between two vectors $\mathbf{a} \in \mathbb{R}^D$ and $\mathbf{b} \in \mathbb{R}^D$



Useful tip: Can achieve the effect of feature scaling (recall last lecture) by using weighted Euclidean distances!

$$d_w(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^D w_i (a_i - b_i)^2} = \sqrt{(\mathbf{a} - \mathbf{b})^\top \mathbf{W} (\mathbf{a} - \mathbf{b})}$$

\mathbf{W} is a $D \times D$ diagonal matrix with weights w_i on its diagonals. Weights may be known or even learned from data (in ML problems)

Note: If \mathbf{W} is a $D \times D$ symmetric matrix then it is called the **Mahalanobis distance** (more on this later)

- Absolute (L1 norm) distance between two vectors $\mathbf{a} \in \mathbb{R}^D$ and $\mathbf{b} \in \mathbb{R}^D$



L1 norm distance is also known as the **Manhattan distance** or **Taxicab norm** (it's a very natural notion of distance between two points in some vector space)

Yes. Another, although less commonly used, distance is the L-infinity distance (equals to max of abs-value of element-wise difference between two vectors)

$$d_1(\mathbf{a}, \mathbf{b}) = \|\mathbf{a} - \mathbf{b}\|_1 = \sum_{i=1}^D |a_i - b_i|$$

Apart from L2 and L1, there other ways of defining distances?



Our First Supervised Learner



Prelude: A Very Primitive Classifier

- Consider a binary classification problem – cat vs dog

- Assume training data with just 2 images – one



and one



- Given a new test image (cat/dog), how do we predict its label?

- A simple idea: Predict using its distance from each of the 2 training images

$$d(\boxed{\text{Test image}}, \boxed{\text{cat}}) < d(\boxed{\text{Test image}}, \boxed{\text{dog}}) ? \text{ Predict cat } \underline{\text{else}} \text{ dog}$$



Wait. Is it ML? Seems to be like just a simple “rule”. Where is the “learning” part in this?

Some possibilities: Use a feature learning/selection algorithm to extract features, and use a Mahalanobis distance where you learn the W matrix (instead of using a predefined W), using “distance metric learning” techniques

The idea also applies to multi-class classification: Use one image per class, and predict label based on the distances of the test image from all such images



Excellent question! Glad you asked! Even this simple model can be learned. For example, for the feature extraction/selection part and/or for the distance computation part



Improving Our Primitive Classifier

- Just one input per class may not sufficiently capture variations in a class
- A natural improvement can be by using more inputs per class

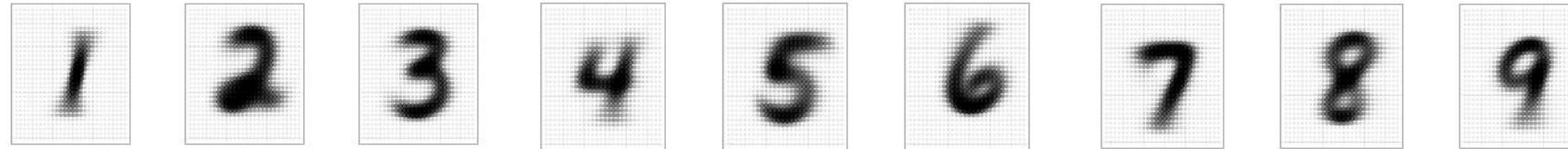


- We will consider two approaches to do this
 - Learning with Prototypes (LwP)
 - Nearest Neighbors (NN – not “neural networks”, at least not for now ☺)
- Both LwP and NN will use multiple inputs per class but in different ways



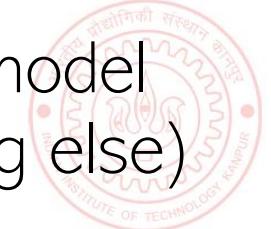
Learning with Prototypes (LwP)

- Basic idea: Represent each class by a “prototype” vector
- Class Prototype: The “mean” or “average” of inputs from that class



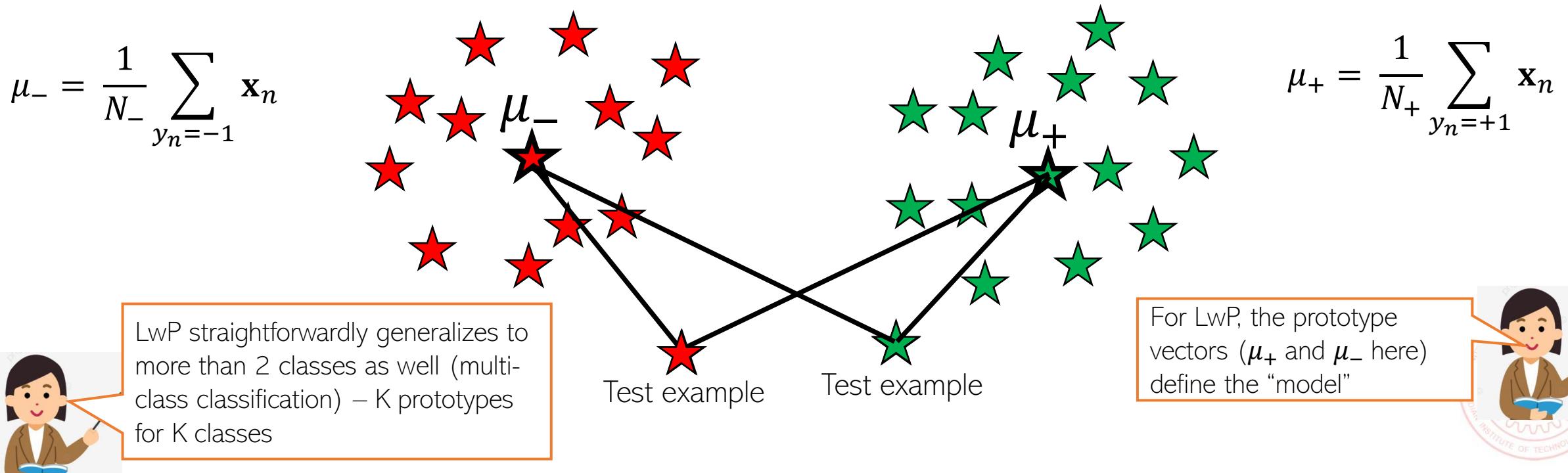
Averages (prototypes) of each of the handwritten digits 1-9

- Predict label of each test input based on its distances from the class prototypes
 - Predicted label will be the class that is the closest to the test input
- How we compute distances can have an effect on the accuracy of this model (may need to try Euclidean, weight Euclidean, Mahalanobis, or something else)



Learning with Prototypes (LwP): An Illustration

- Suppose the task is binary classification (two classes assumed pos and neg)
- Training data: N labelled examples $\{(\mathbf{x}_n, y_n)\}_{n=1}^N$, $\mathbf{x}_n \in \mathbb{R}^D$, $y_n \in \{-1, +1\}$
 - Assume N_+ example from positive class, N_- examples from negative class
 - Assume green is positive and red is negative

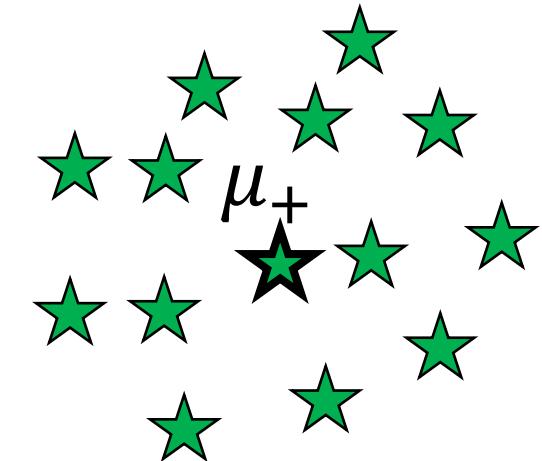
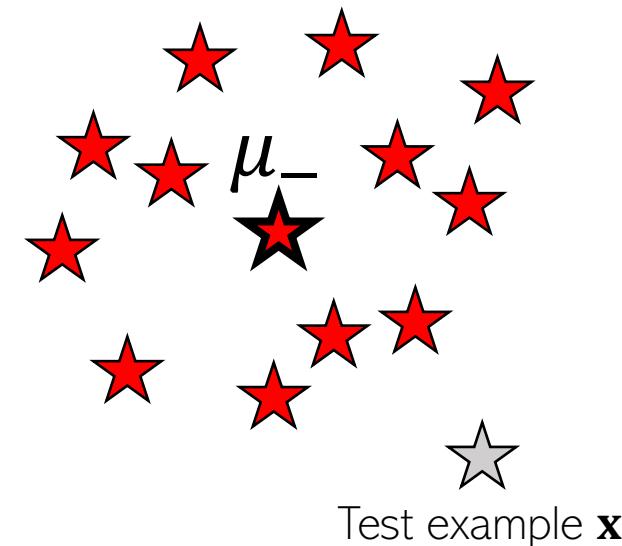


LwP: The Prediction Rule, Mathematically

- What does the prediction rule for LwP look like mathematically?
- Assume we are using Euclidean distances here

$$\|\boldsymbol{\mu}_- - \mathbf{x}\|^2 = \|\boldsymbol{\mu}_-\|^2 + \|\mathbf{x}\|^2 - 2\langle \boldsymbol{\mu}_-, \mathbf{x} \rangle$$

$$\|\boldsymbol{\mu}_+ - \mathbf{x}\|^2 = \|\boldsymbol{\mu}_+\|^2 + \|\mathbf{x}\|^2 - 2\langle \boldsymbol{\mu}_+, \mathbf{x} \rangle$$



Prediction Rule: Predict label as +1 if $f(\mathbf{x}) = \|\boldsymbol{\mu}_- - \mathbf{x}\|^2 - \|\boldsymbol{\mu}_+ - \mathbf{x}\|^2 > 0$ otherwise -1



LwP: The Prediction Rule, Mathematically

- Let's expand the prediction rule expression a bit more

$$\begin{aligned}
 f(\mathbf{x}) &= \|\boldsymbol{\mu}_- - \mathbf{x}\|^2 - \|\boldsymbol{\mu}_+ - \mathbf{x}\|^2 \\
 &= \|\boldsymbol{\mu}_-\|^2 + \|\mathbf{x}\|^2 - 2\langle \boldsymbol{\mu}_-, \mathbf{x} \rangle - \|\boldsymbol{\mu}_+\|^2 - \|\mathbf{x}\|^2 + 2\langle \boldsymbol{\mu}_+, \mathbf{x} \rangle \\
 &= 2\langle \boldsymbol{\mu}_+ - \boldsymbol{\mu}_-, \mathbf{x} \rangle + \|\boldsymbol{\mu}_-\|^2 - \|\boldsymbol{\mu}_+\|^2 \\
 &= \langle \mathbf{w}, \mathbf{x} \rangle + b
 \end{aligned}$$

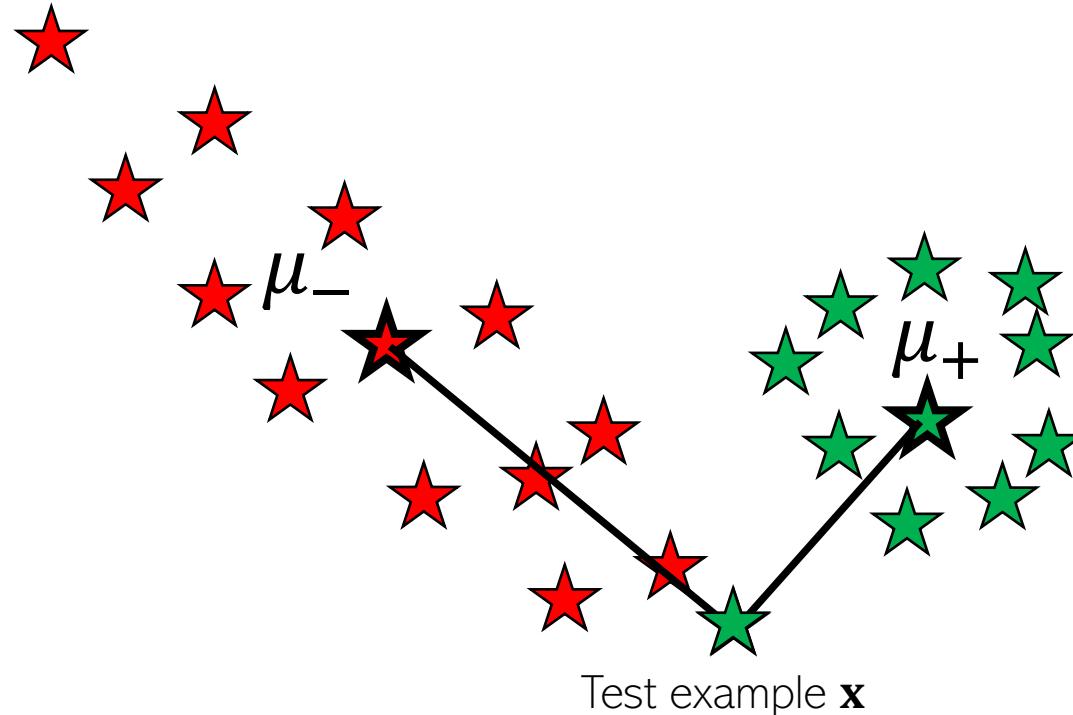
- Thus LwP with Euclidean distance is equivalent to a linear model with
 - Weight vector $\mathbf{w} = 2(\boldsymbol{\mu}_+ - \boldsymbol{\mu}_-)$
 - Bias term $b = \|\boldsymbol{\mu}_-\|^2 - \|\boldsymbol{\mu}_+\|^2$
- Prediction rule therefore is: Predict +1 if $\langle \mathbf{w}, \mathbf{x} \rangle + b > 0$, else predict -1

Will look at linear models more formally and in more detail later



LwP: Some Failure Cases

- Here is a case where LwP with Euclidean distance may not work well



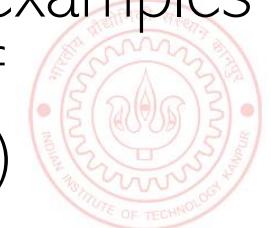
Can use feature scaling or use Mahalanobis distance to handle such cases (will discuss this in the next lecture)



- In general, if classes are not equisized and spherical, LwP with Euclidean distance will usually not work well (but improvements possible; will discuss later)

LwP: Some Key Aspects

- Very simple, interpretable, and lightweight model
 - Just requires computing and storing the class prototype vectors
- Works with any number of classes (thus for multi-class classification as well)
- Can be generalized in various ways to improve it further, e.g.,
 - Modeling each class by a [probability distribution](#) rather than just a prototype vector
 - Using distances other than the standard Euclidean distance (e.g., Mahalanobis)
- With a learned distance function, can work very well even with very few examples from each class (used in some “few-shot learning” models nowadays – if interested, please refer to “Prototypical Networks for Few-shot Learning”)



Next Lecture

- Fixing LwP
- Nearest Neighbors



Learning by Computing Distances (2): Wrapping-up LwP, Nearest Neighbors

CS771: Introduction to Machine Learning

Piyush Rai

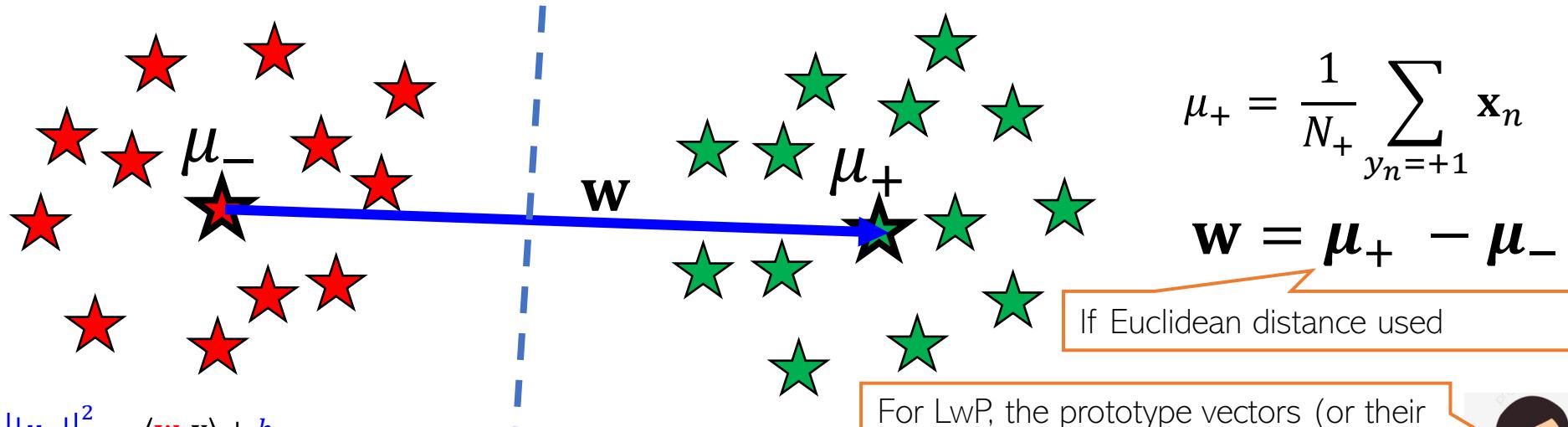
Learning with Prototypes (LwP)

$$\mu_- = \frac{1}{N_-} \sum_{y_n=-1} \mathbf{x}_n$$

Prediction rule for LwP
(for binary classification
with Euclidean distance)

$$f(\mathbf{x}) = 2\langle \mu_+ - \mu_-, \mathbf{x} \rangle + \|\mu_-\|^2 - \|\mu_+\|^2 = \langle \mathbf{w}, \mathbf{x} \rangle + b$$

$f(\mathbf{x}) > 0$ then predict +1 otherwise -1



For LwP, the prototype vectors (or their difference) define the “model”. μ_+ and μ_- (or just \mathbf{w} in the Euclidean distance case) are the **model parameters**.



Exercise: Show that for the bin. classfn case

$$f(\mathbf{x}) = \sum_{n=1}^N \alpha_n \langle \mathbf{x}_n, \mathbf{x} \rangle + b$$

Note: Even though $f(\mathbf{x})$ can be expressed in this form, if $N > D$, this may be more expensive to compute ($O(N)$ time) as compared to $\langle \mathbf{w}, \mathbf{x} \rangle + b$ ($O(D)$ time).

So the “score” of a test point \mathbf{x} is a weighted sum of its similarities with each of the N training inputs. Many supervised learning models have $f(\mathbf{x})$ in this form as we will see later

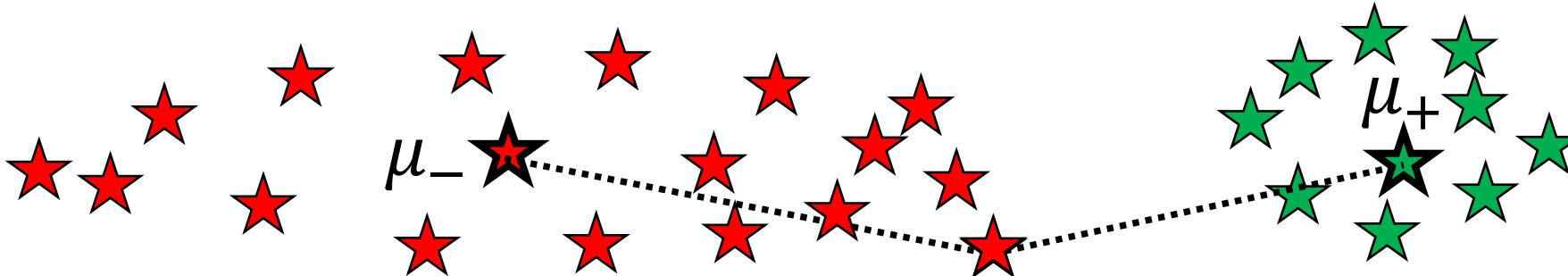
Can throw away training data after computing the prototypes and just need to keep the model parameters for the test time in such “parametric” models

However the form $f(\mathbf{x}) = \sum_{n=1}^N \alpha_n \langle \mathbf{x}_n, \mathbf{x} \rangle + b$ is still very useful as we will see later when we discuss **kernel methods**



Improving LwP when classes are complex-shaped

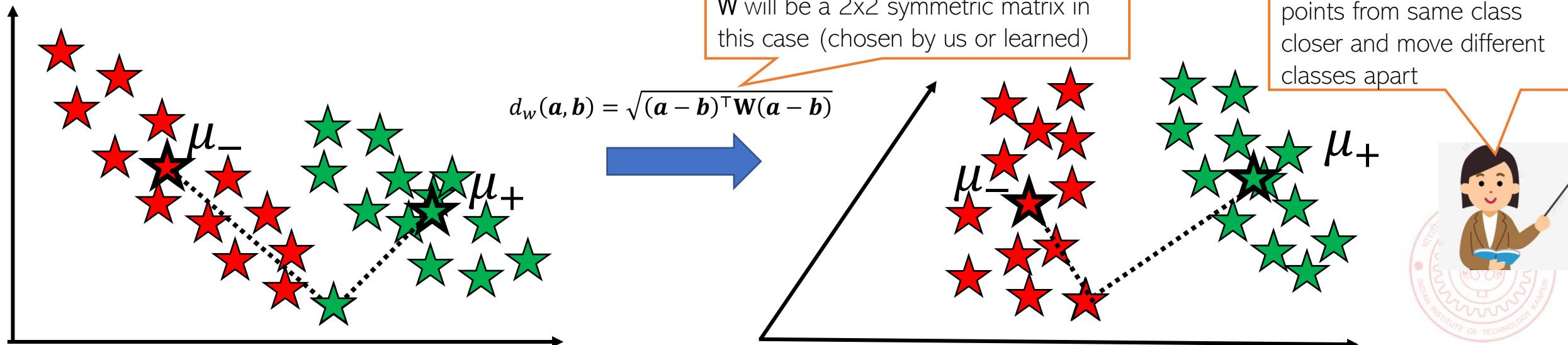
- Using weighted Euclidean or Mahalanobis distance can sometimes help



$$d_w(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^D w_i (a_i - b_i)^2}$$

Use a smaller w_i for the horizontal axis feature in this example

- Note: Mahalanobis distance also has the effect of rotating the axes which helps



Improving LwP when classes are complex-shaped

- Even with weighted Euclidean or Mahalanobis dist, LwP still a linear classifier ☹
- **Exercise:** Prove the above fact. You may use the following hint
 - Mahalanobis dist can be written as $d_w(\mathbf{a}, \mathbf{b}) = \sqrt{(\mathbf{a} - \mathbf{b})^\top \mathbf{W}(\mathbf{a} - \mathbf{b})}$
 - \mathbf{W} is a symmetric matrix and thus can be written as $\mathbf{A}\mathbf{A}^\top$ for any matrix \mathbf{A}
 - Showing for Mahalanobis is enough. Weighted Euclidean is a special case with diag \mathbf{W}
- LwP can be extended to learn nonlinear decision boundaries if we use nonlinear distances/similarities (more on this when we talk about kernels)



Note: Modeling each class by not just a mean by a probability distribution can also help in learning nonlinear decision boundaries. More on this when we discuss probabilistic models for classification



LwP as a subroutine in other ML models

- For data-clustering (unsupervised learning), K -means clustering is a popular algo



- K -means also computes means/centres/prototypes of groups of unlabeled points
- Harder than LwP since labels are unknown. But we can do the following
 - Guess the label of each point, compute means using guess labels
 - Refine labels using these means (assign each point to the current closest mean)
 - Repeat until means don't change anymore
- Many other models also use LwP as a subroutine

Will see K-means in detail later



Supervised Learning using Nearest Neighbors



Nearest Neighbors

- Another supervised learning technique based on computing distances
- Very simple idea. Simply do the following at test time
 - Compute distance of of the test point from all the training points
 - Sort the distances to find the “nearest” input(s) in training data
 - Predict the label using **majority** or **avg** label of these inputs
- Can use Euclidean or other dist (e.g., Mahalanobis). Choice imp just like LwP
- Unlike LwP which does prototype based comparison, nearest neighbors method looks at the labels of individual training inputs to make prediction
- Applicable to both classifn as well as regression (LwP only works for classifn)

Wait. Did you say distance from ALL the training points? That's gonna be sooooo expensive! 😞

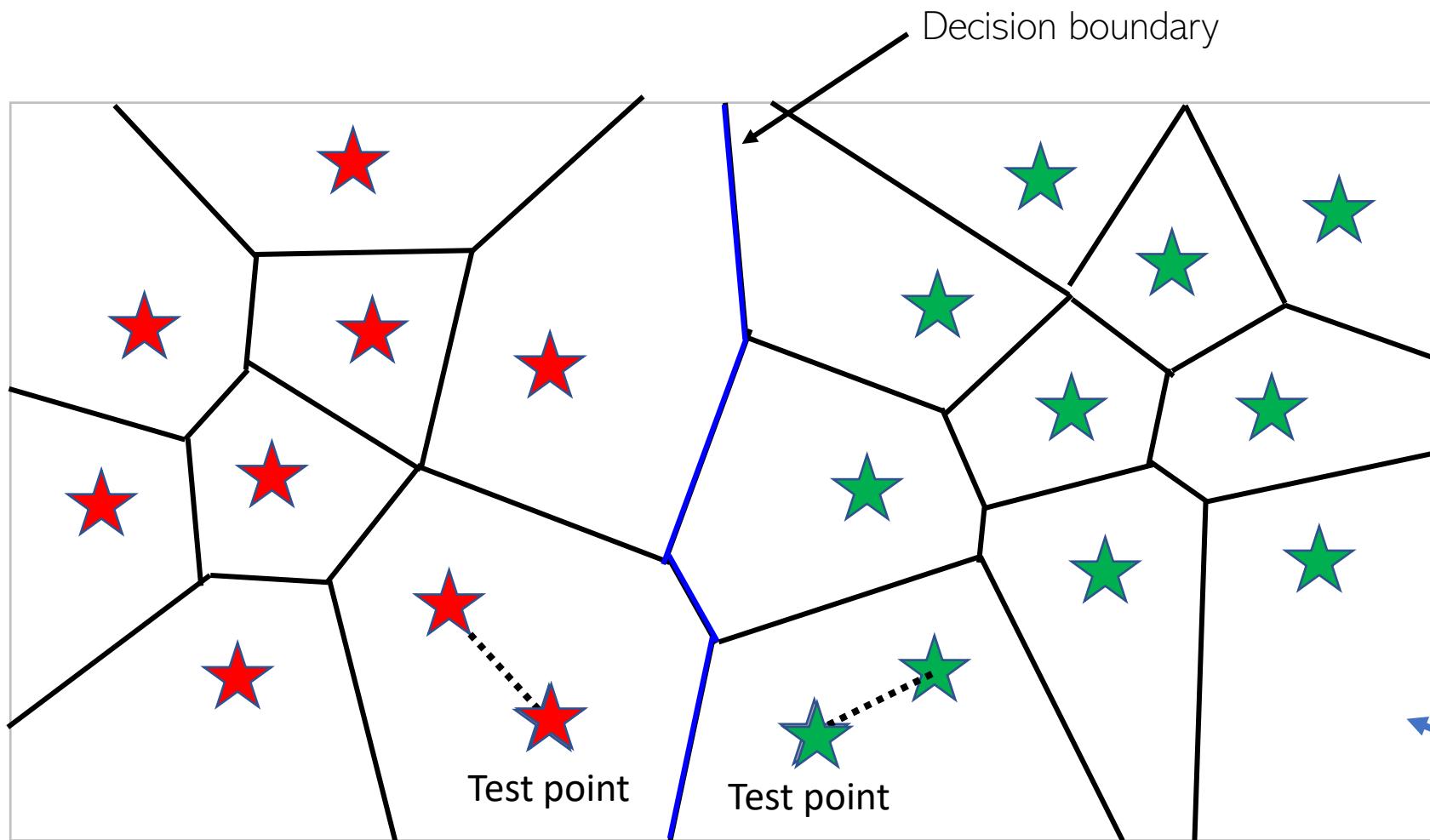
Yes, but let's not worry about that at the moment. There are ways to speed up this step



Nearest Neighbors for Classification



Nearest Neighbor (or “One” Nearest Neighbor)



Interesting. Even with Euclidean distances, it can learn nonlinear decision boundaries?



Indeed. And that's possible since it is a "local" method (looks at a local neighborhood of the test point to make prediction)

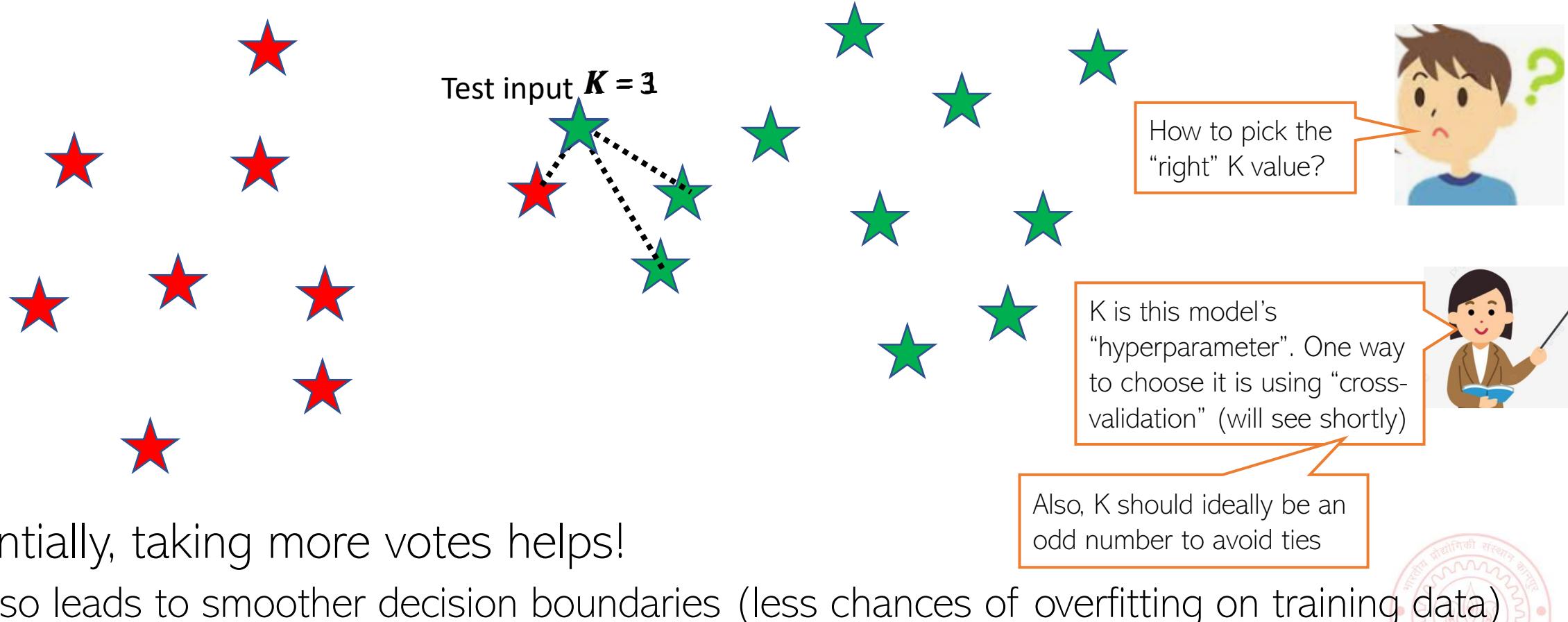


Nearest neighbour approach induces a **Voronoi tessellation/partition** of the input space (all test points falling in a cell will get the label of the training input in that cell)



K Nearest Neighbors (KNN)

- In many cases, it helps to look at not one but $K > 1$ nearest neighbors

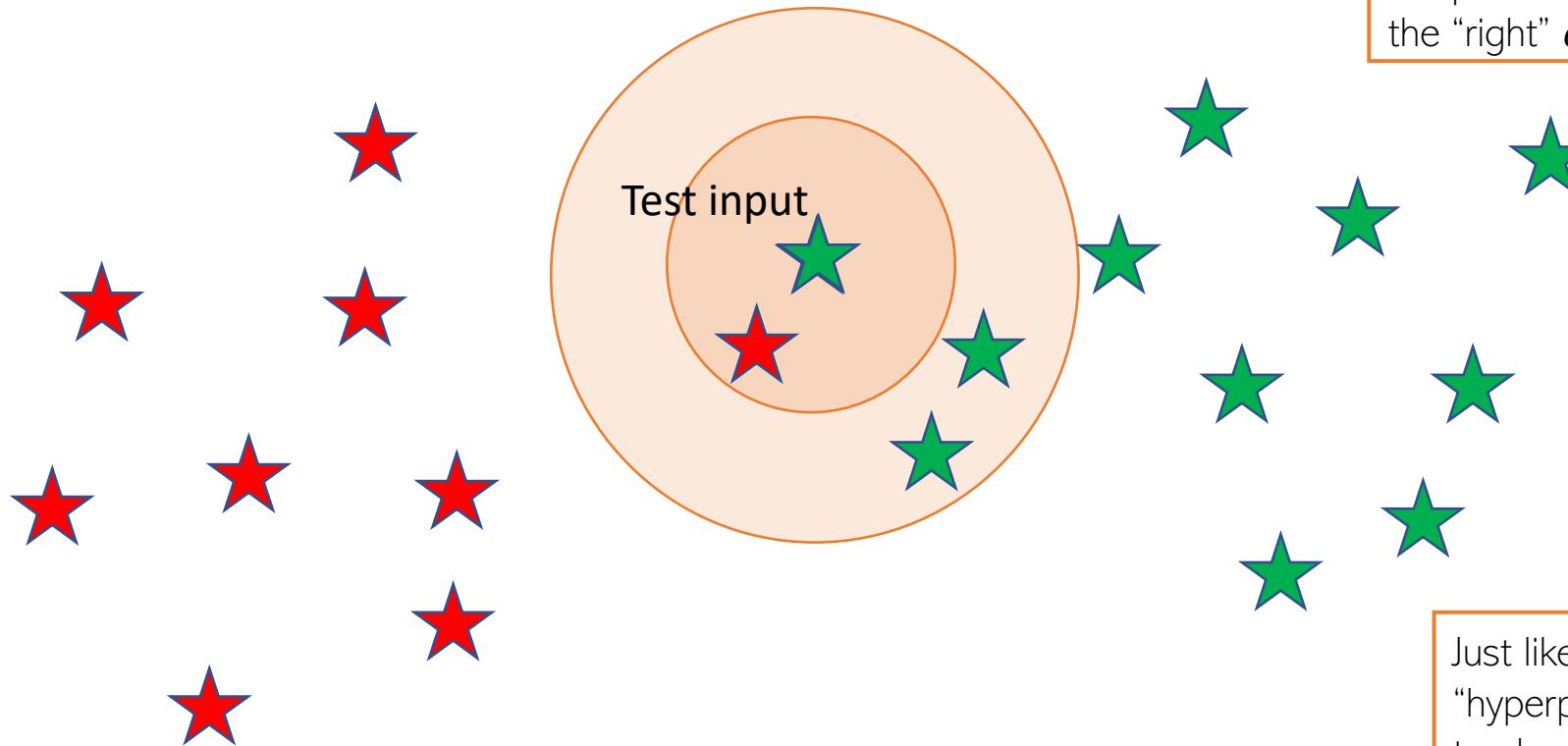


- Essentially, taking more votes helps!

- Also leads to smoother decision boundaries (less chances of overfitting on training data)

ϵ -Ball Nearest Neighbors (ϵ -NN)

- Rather than looking at a fixed number K of neighbors, can look inside a ball of a given radius ϵ , around the test input

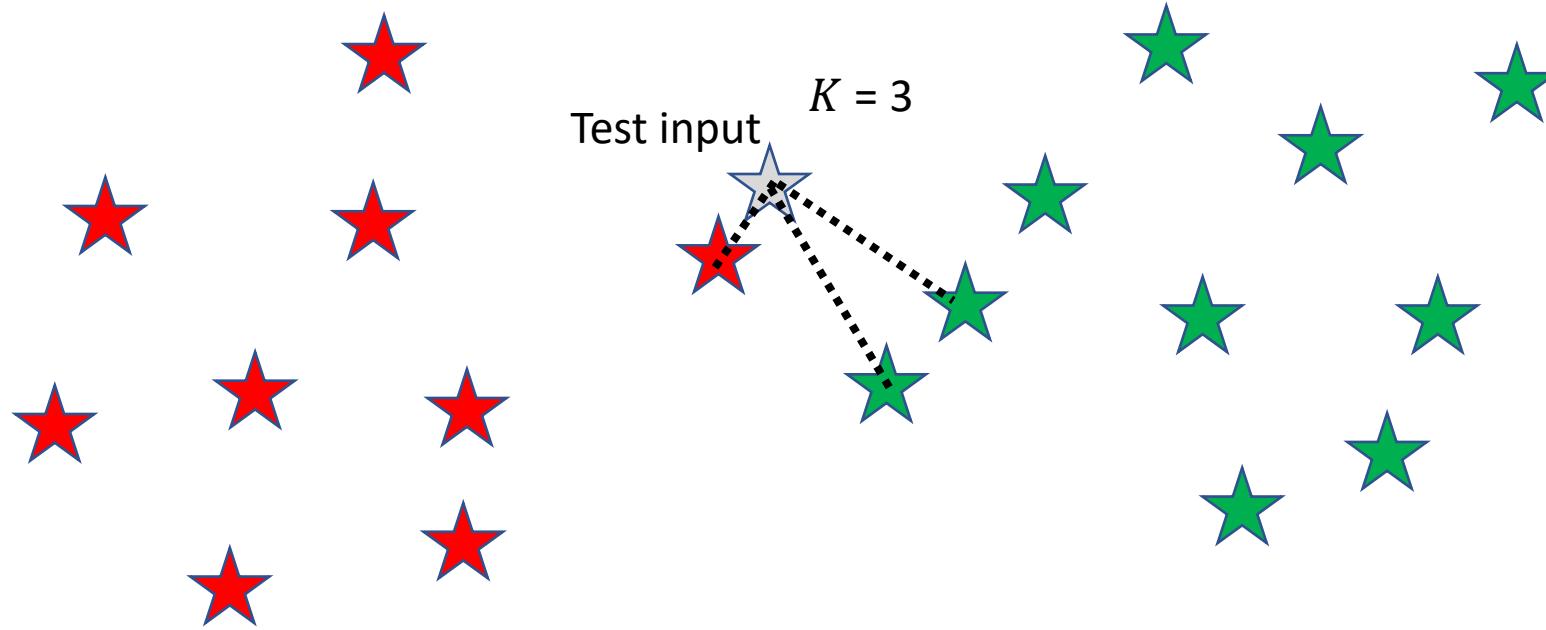


Just like K , ϵ is also a “hyperparameter”. One way to choose it is using “cross-validation” (will see shortly)



Distance-weighted KNN and ϵ -NN

- The standard KNN and ϵ -NN treat all nearest neighbors equally (all vote equally)



- An improvement: When voting, give more importance to closer training inputs

Unweighted KNN prediction:

$$\frac{1}{3} \text{ red star} + \frac{1}{3} \text{ green star} + \frac{1}{3} \text{ green star} = \text{green star}$$

Weighted KNN prediction:

$$\frac{3}{5} \text{ red star} + \frac{1}{5} \text{ green star} + \frac{1}{5} \text{ green star} = \text{red star}$$

In weighted approach, a single red training input is being given 3 times more importance than the other two green inputs since it is sort of "three times" closer to the test input than the other two green inputs

ϵ -NN can also be made weighted likewise



KNN/ ϵ -NN for Other Supervised Learning Problems¹³

- Can apply KNN/ ϵ -NN for other supervised learning problems as well, such as
 - Multi-class classification
 - Regression
 - Tagging/multi-label learning
- We can also try the weighted versions for such problems, just like we did in the case of binary classification
- For multi-class, simply used the same majority rule like in binary classfn case
 - Just a simple difference that now we have more than 2 classes
- For regression, simply compute the average of the outputs of nearest neighbors
- For multi-label learning, each output is a binary vector (presence/absence of tag)
 - Just compute the average of the binary vectors
 - Result won't be a binary vector but we can report the best tags based on magnitudes

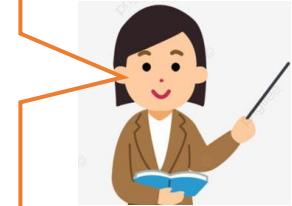


KNN Prediction Rule: The Mathematical Form

- Let's denote the set of K nearest neighbors of an input \mathbf{x} by $N_K(\mathbf{x})$
- The unweighted KNN prediction \mathbf{y} for a test input \mathbf{x} can be written as

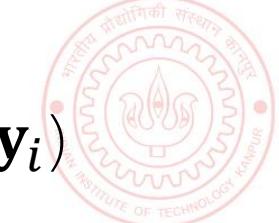
$$\mathbf{y} = \frac{1}{K} \sum_{i \in N_K(\mathbf{x})} \mathbf{y}_i$$

Assuming discrete labels with 5 possible values, the one-hot representation will be a all zeros vector of size 5, except a single 1 denoting the value of the discrete label, e.g., if label = 3 then one-hot vector = [0,0,**1**,0,0]



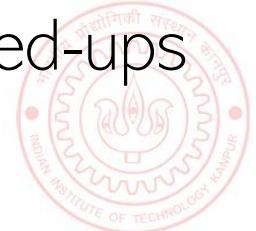
- This form makes direct sense of regression and for cases where each output is a vector (e.g., **multi-class classification** where each output is a discrete value which can be represented as a **one-hot vector**, or **tagging/multi-label classification** where each output is a **binary vector**)

- For binary classification, assuming labels as +1/-1, we predict $\text{sign}(\frac{1}{K} \sum_{i \in N_K(\mathbf{x})} \mathbf{y}_i)$



Nearest Neighbors: Some Comments

- An old, classic but still very widely used algorithm
 - Can sometimes give deep neural networks a run for their money ☺
- Can work very well in practice with the right distance function
- Comes with very nice theoretical guarantees
- Also called a memory-based or **instance-based** or **non-parametric** method
 - No “model” is learned here (unlike LwP). Prediction step uses all the training data
- Requires lots of storage (need to keep all the training data at test time)
- Prediction step can be slow at test time
 - For each test point, need to compute its distance from all the training points
- Clever data-structures or data-summarization techniques can provide speed-ups



Next Lecture

- Hyperparameter/model selection via cross-validation
- Learning with Decision Trees



Learning using Decision Trees

CS771: Introduction to Machine Learning

Piyush Rai

Plan for today

- Wrap-up the discussion of nearest neighbors
 - How to speed-up nearest neighbors at test time?
- Model/hyperparameter Selection
- Learning with [Decision Trees](#)

Remember: There is no “training” stage in the standard nearest neighbors (unless we are learning the feature of distance function from the training data)



The training data itself is the model that we need to keep at test time. Recall that NN is a memory-based or nonparametric approach



Speeding-up Nearest Neighbors

- Can use techniques to reduce the training set size
 - Several data summarization techniques exist that discard redundant training inputs
 - Now we will require fewer number of distance computations for each test input
- Can use techniques to reduce the data dimensionality (no. of features)
 - Won't reduce no. of distance computations but each distance computation will be faster
- Compressing each input into a small binary vector (a type of dim-red)
 - Distance/similarity computation between bin. vecs is very fast (can even be done in H/W)
- Various other techniques as well, e.g.,
 - Locality Sensitive Hashing (group training inputs into buckets)
 - Clever data structures (e.g., k-D trees) to organize training inputs
 - Use a divide-and-conquer type approach to narrow down the search region

We will look at Decision Trees which is also like a divide-and-conquer approach



Hyperparameter Selection

- Every ML model has some hyperparameters that need to be tuned, e.g.,
 - K in KNN or ϵ in ϵ -NN
 - Choice of distance to use in LwP or nearest neighbors
- Would like to choose h.p. values that would give best performance on test data

Oops, sorry!
What to do
then?



Is validation set a good proxy to test set?

Okay. So I can try multiple hyperparam values and choose the one that gives the best accuracy on the **test data**. Simple, isn't it?

Beware. You are committing a crime. Never Ever touch your test data while building the model



Use **cross-validation** - use a part of your training data (we will call it “validation/held-out set”) as test data. That's not a crime. ☺

Usually yes since training set and test sets are assumed to be similar (plus, you are careful in choosing your validation set)

• Every ML model has some hyperparameters that need to be tuned, e.g.
• K in KNN or ϵ in ϵ NN
• Choice of distance to use in LwP or nearest neighbors

• Would like to choose h.p. values that would give best performance on test data



Cross-Validation

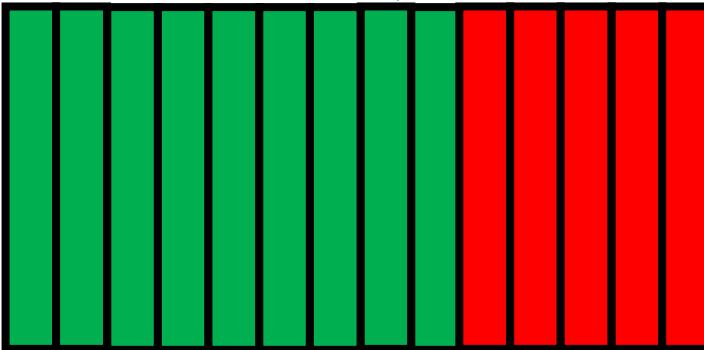
Training Set (assuming bin. class. problem)



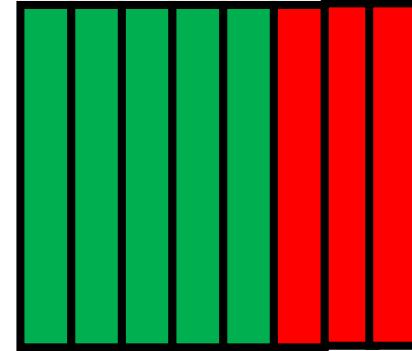
Actual Training Set

Randomly Split

Validation Set



Test Set



No peeking while building the model



What if the random split is unlucky (i.e., validation data is not like test data)?



Note: Not just h.p. selection; we can also use CV to pick the best ML model from a set of different ML models (e.g., say we have to pick between two models we may have trained - LwP and nearest neighbors. Can use CV to choose the better one.)



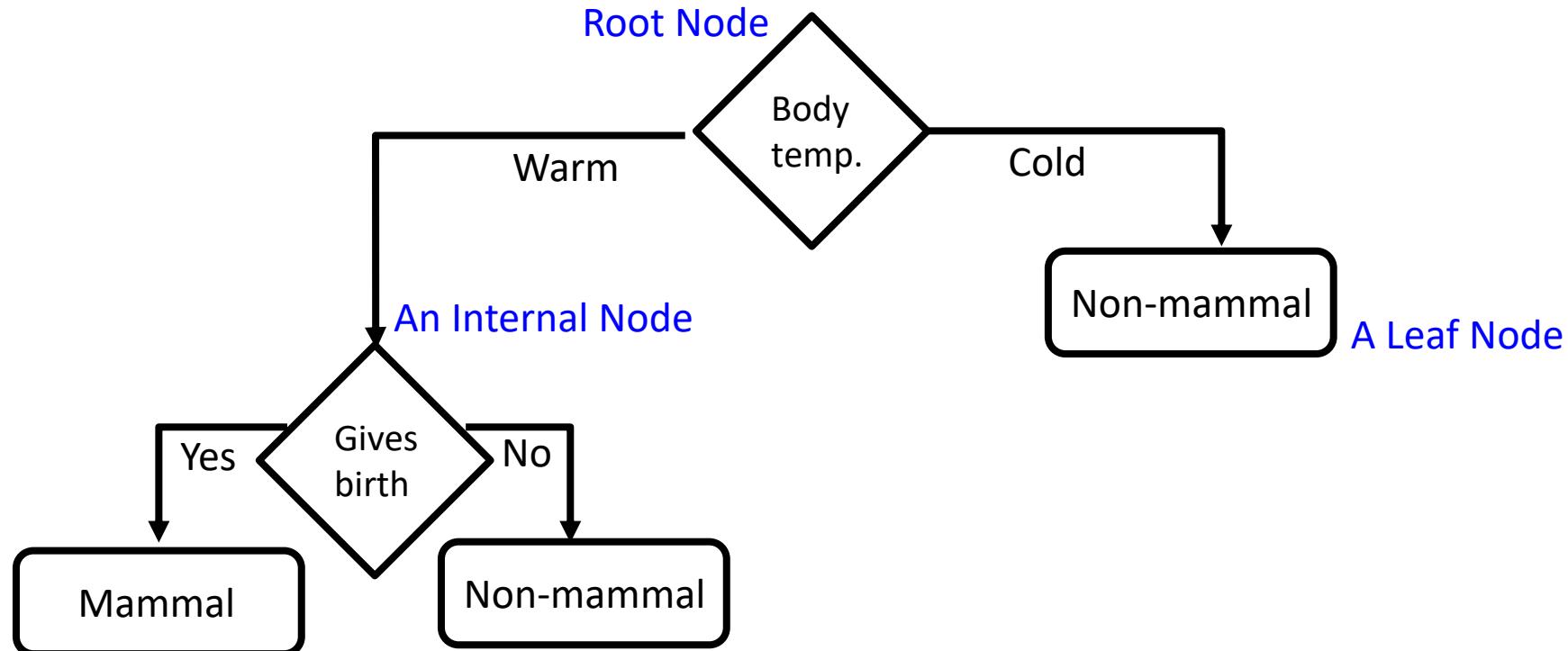
Randomly split the original training data into actual training set and validation set. Using the actual training set, train several times, each time using a different value of the hyperparam. Pick the hyperparam value that gives best accuracy on the validation set



If you fear an unlucky split, try multiple splits. Pick the hyperparam value that gives the **best average CV accuracy across all such splits**. If you are using N splits, this is called N-fold cross validation

Decision Trees

- A Decision Tree (DT) defines a hierarchy of rules to make a prediction

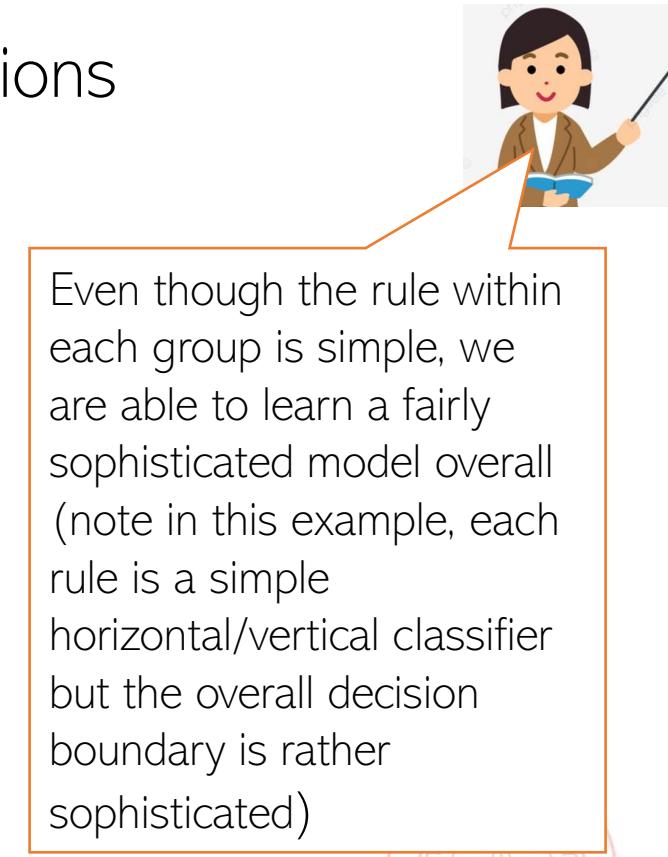
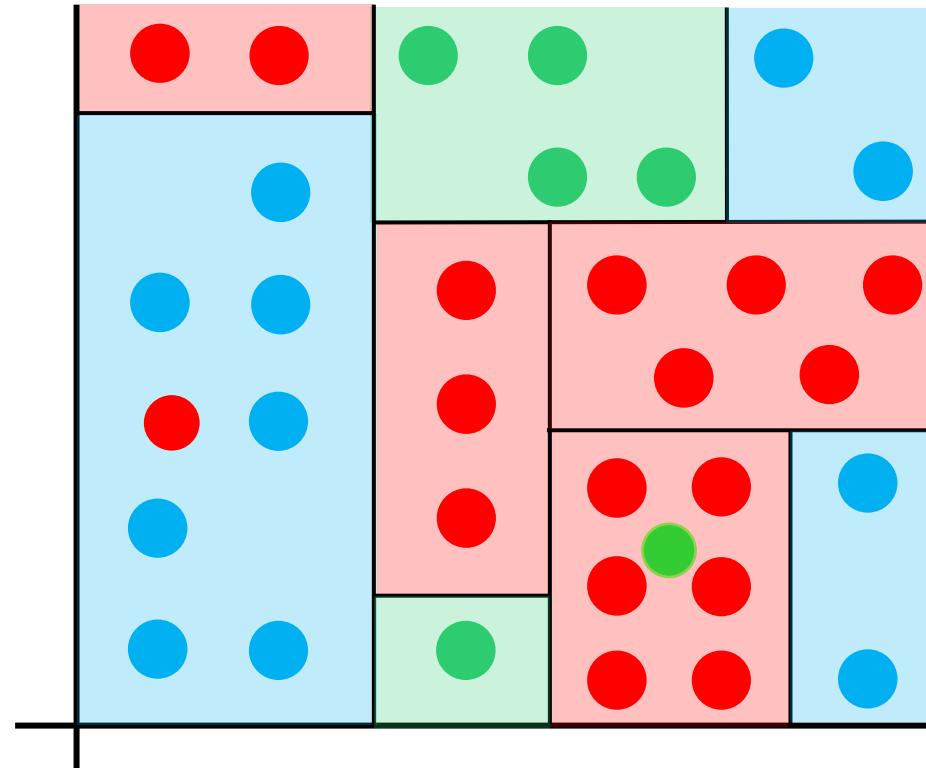
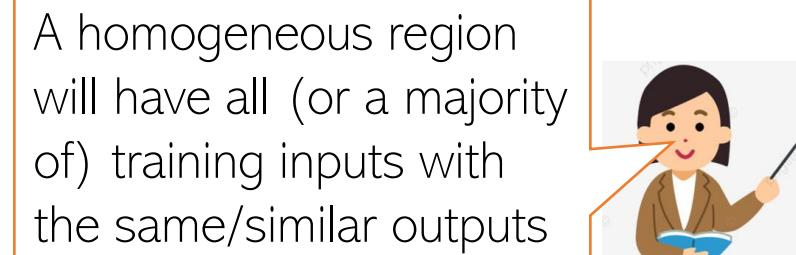
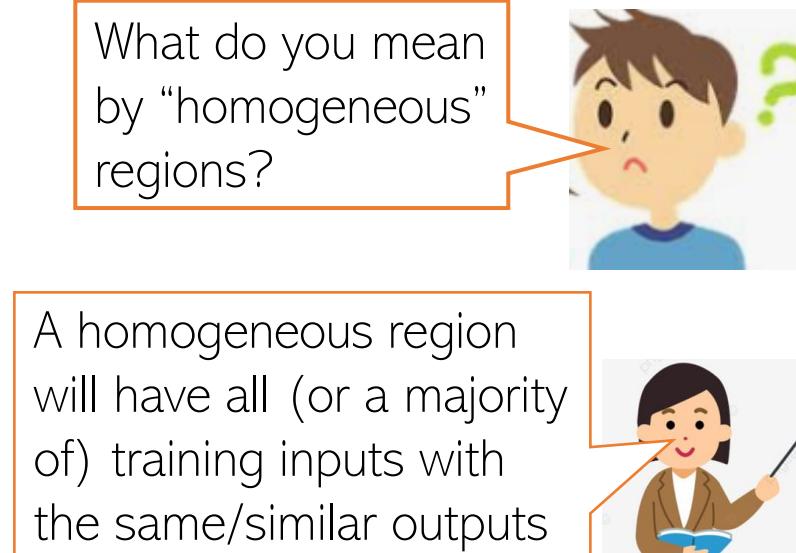


- Root and internal nodes test rules. Leaf nodes make predictions
- Decision Tree (DT) learning is about learning such a tree from labeled data



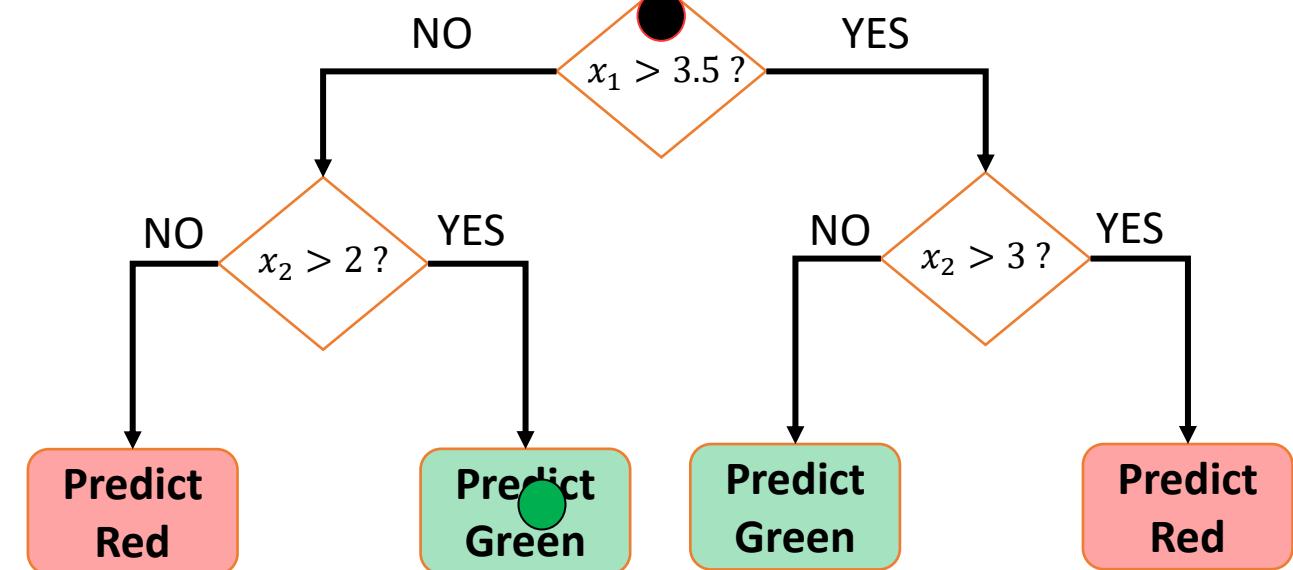
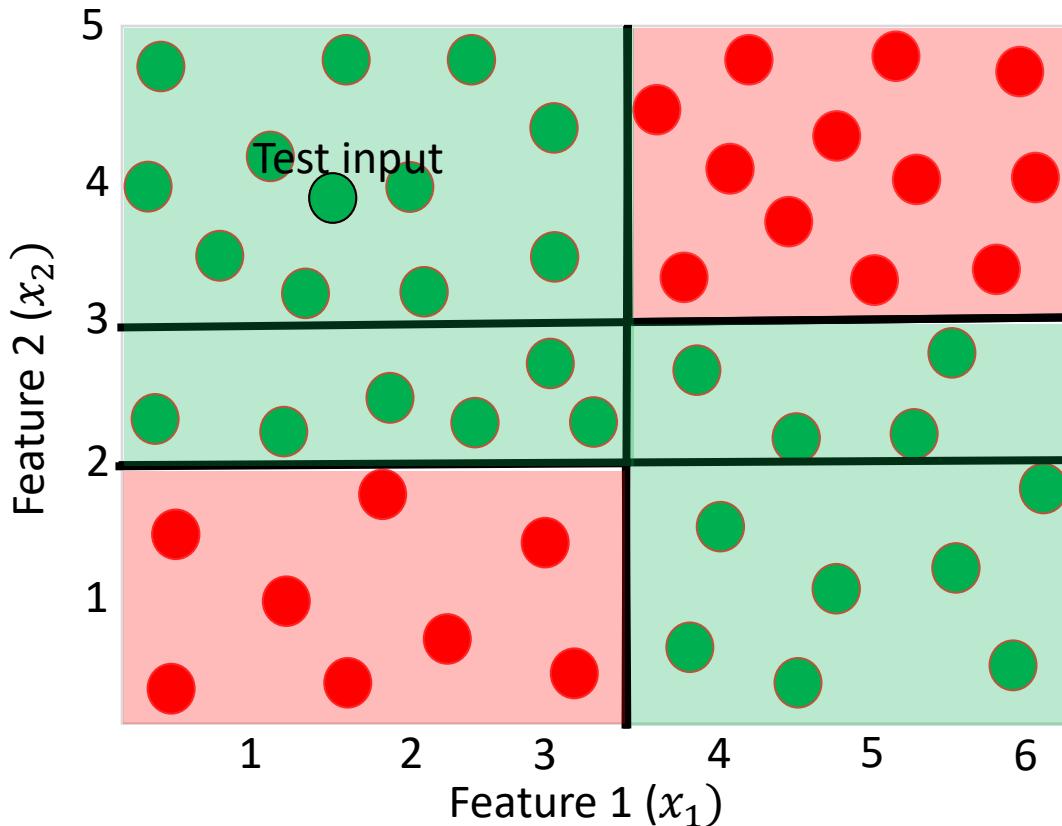
Decision Trees for Supervised Learning

- The basic idea is very simple
- Recursively partition the training data into homogeneous regions



- Within each group, fit a simple supervised learner (e.g., predict the majority label)

Decision Trees for Classification



DT is very efficient at test time: To predict the label of a test point, nearest neighbors will require computing distances from 48 training inputs. DT predicts the label by doing just 2 feature-value comparisons! Way more fast!!!

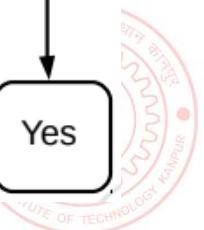
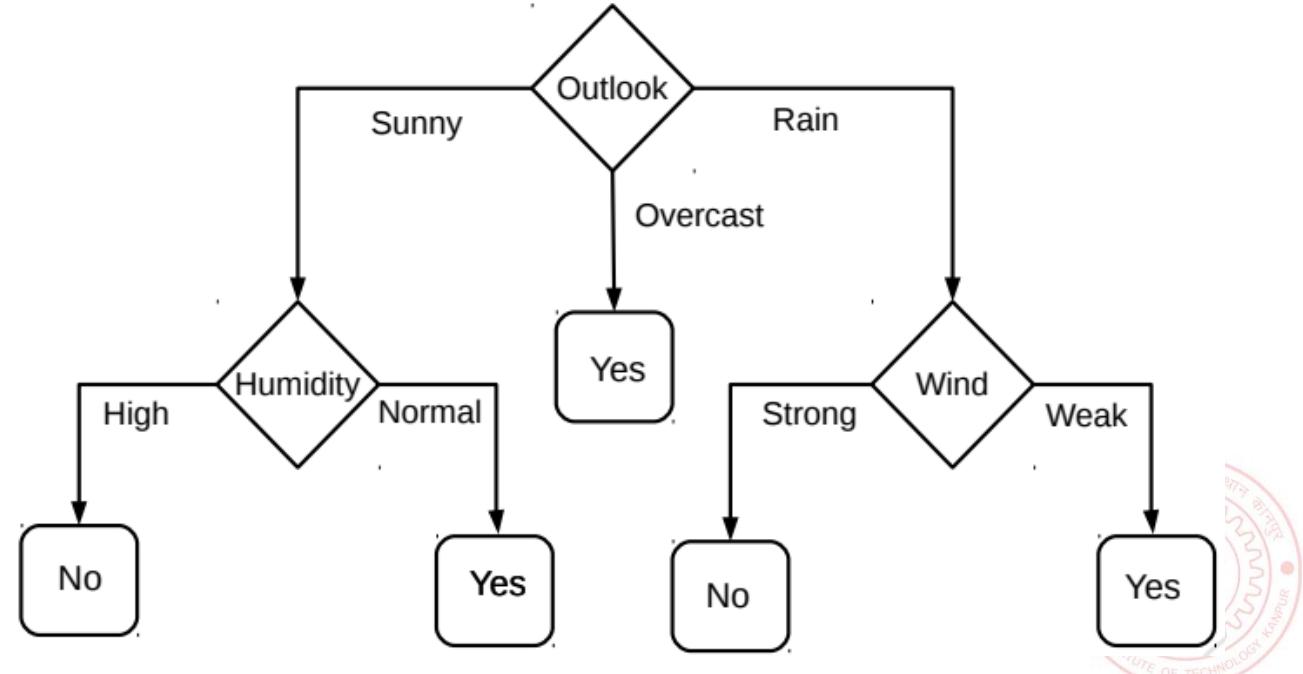
Remember: Root node contains all training inputs
Each leaf node receives a subset of training inputs



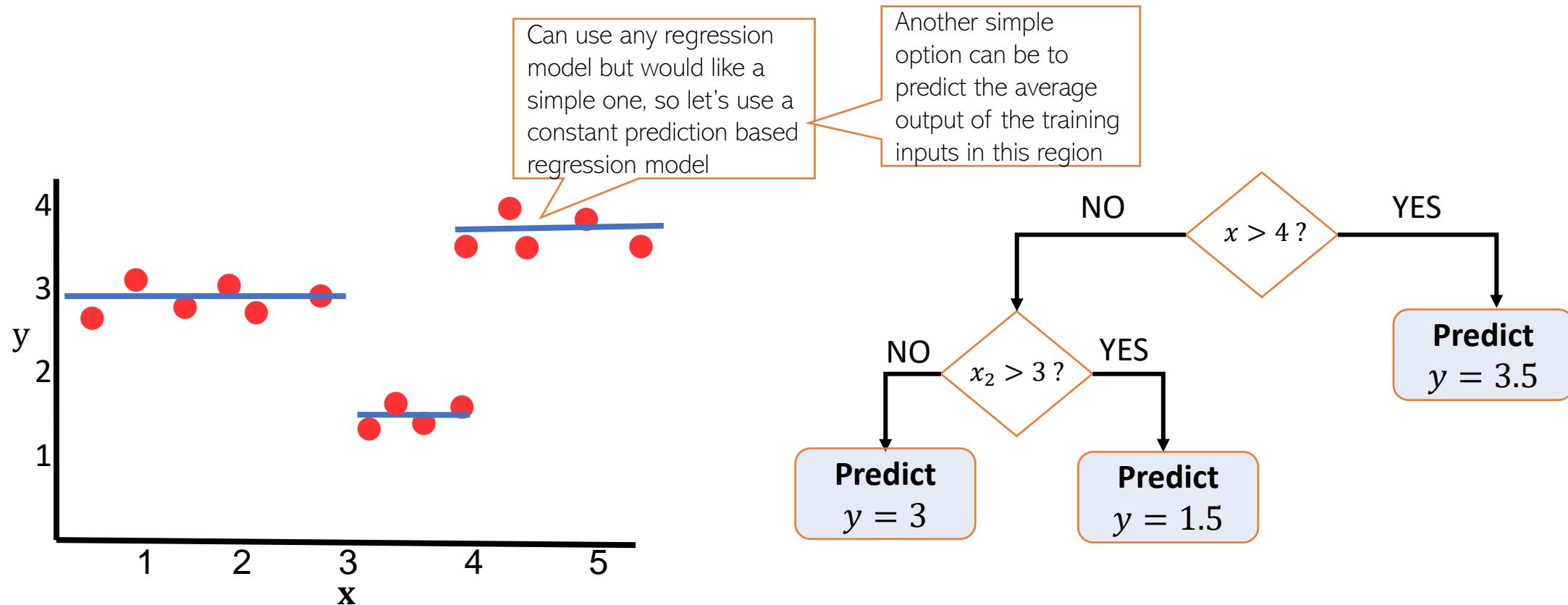
Decision Trees for Classification: Another Example

- Deciding whether to play or not to play Tennis on a Saturday
- Each input (Saturday) has 4 categorical features: Outlook, Temp., Humidity, Wind
- A binary classification problem (play vs no-play)
- Below Left: Training data, Below Right: A decision tree constructed using this data

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



Decision Trees for Regression



To predict the output for a test point, nearest neighbors will require computing distances from 15 training inputs. DT predicts the label by doing just at most feature-value comparisons! Way more fast!!!



Decision Trees: Some Considerations

- What should be the **size/shape** of the DT?

- Number of internal and leaf nodes
- Branching factor of internal nodes
- Depth of the tree

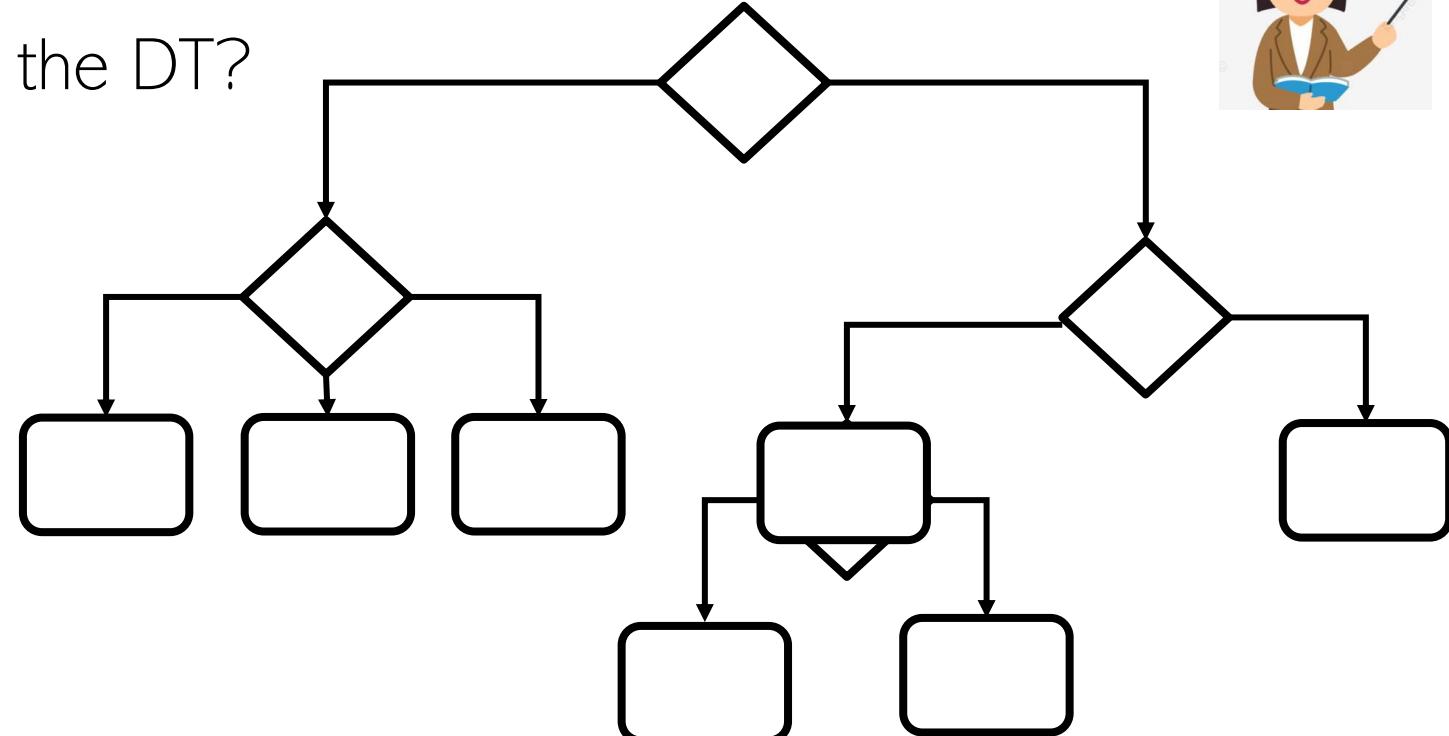
- Split criterion at internal nodes

- Use another classifier?
- Or maybe by doing a simpler test?

- What to do at the leaf node? Some options:

- Make a constant prediction for each test input reaching there
- Use a nearest neighbor based prediction using training inputs at that leaf node
- Train and predict using some other sophisticated supervised learner on that node

Usually, cross-validation can be used to decide size/shape

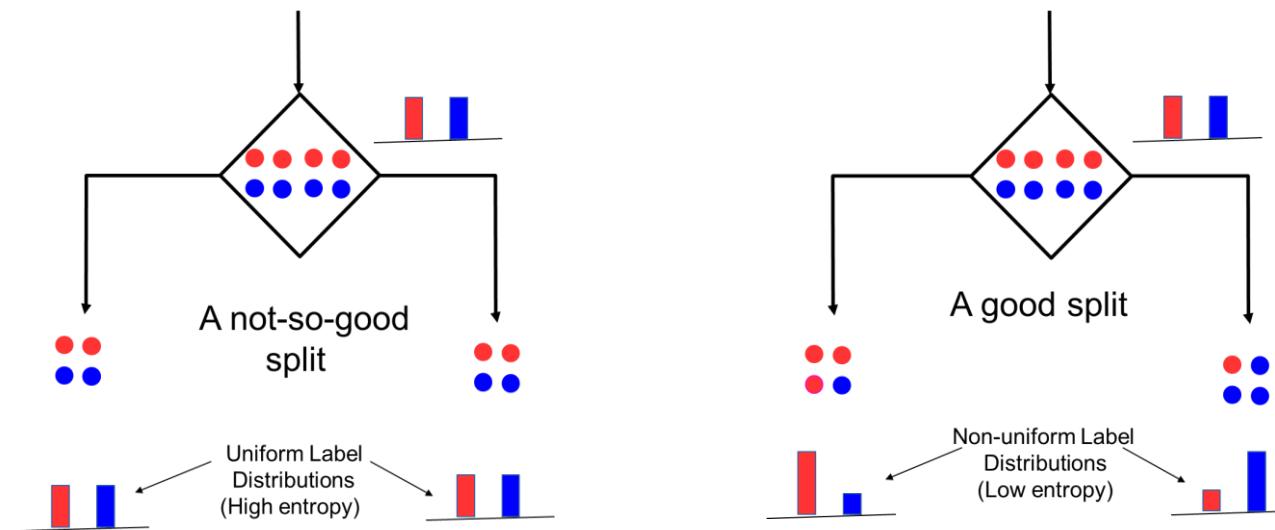


Usually, constant prediction at leaf nodes used since it will be very fast



How to Split at Internal Nodes?

- Recall that each internal node receives a subset of all the training inputs
- Regardless of the criterion, the split should result in as “pure” groups as possible
 - A pure group means that the majority of the inputs have the same label/output



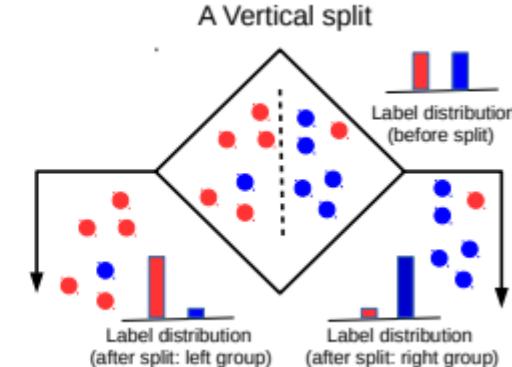
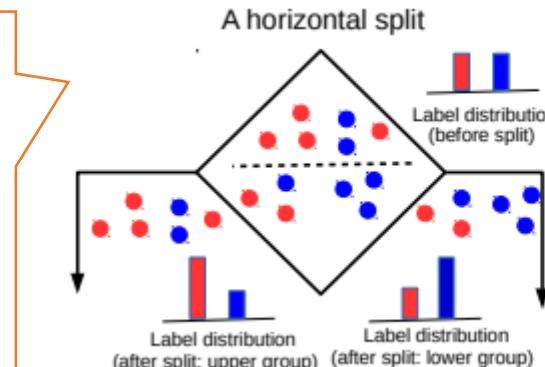
- For classification problems (discrete outputs), entropy is a measure of purity
 - Low entropy \Rightarrow high purity (less uniform label distribution)
 - Splits that give the largest reduction (before split vs after split) in entropy are preferred (this reduction is also known as “information gain”)



Techniques to Split at Internal Nodes?

- Each internal node decides which outgoing branch an input should be sent to
- This decision/split can be done using various ways, e.g.,
 - Testing the value of a single feature at a time (such internal node called “Decision Stump”)

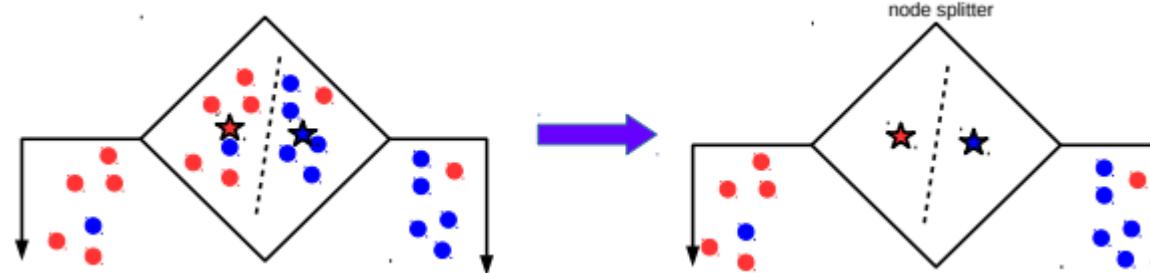
With this approach, all features and all possible values of each feature need to be evaluated in selecting the feature to be tested at each internal node (can be slow but can be made faster using some tricks)



DT methods based on testing a single feature at each internal node are faster and more popular (e.g., ID3, C4.5 algos)



- Learning a classifier (e.g., LwP or some more sophisticated classifier)



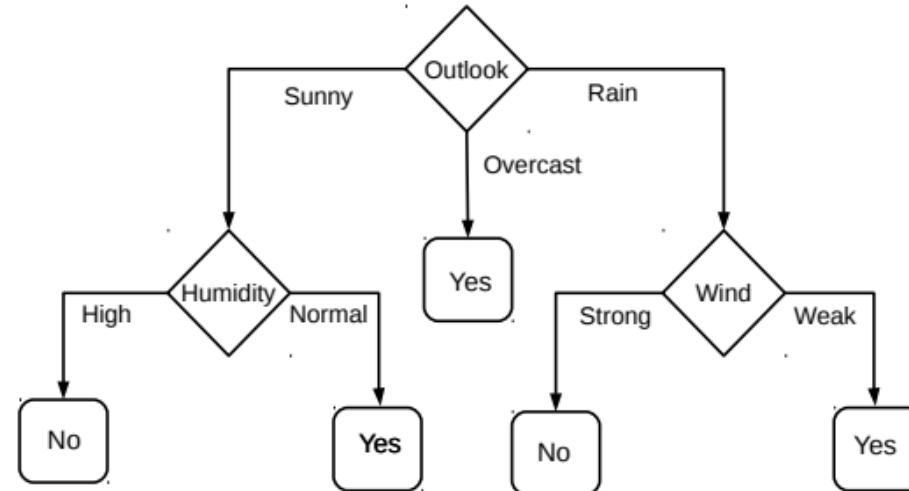
DT methods based on learning and using a separate classifier at each internal node are less common. But this approach can be very powerful and are sometimes used in some advanced DT methods



Decision Tree Construction: An Example

- Let's consider the playing Tennis example
- Assume each internal node will test the value of one of the features

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



- Question: Why does it make more sense to test the feature "outlook" first?
- Answer: Of all the 4 features, it's the most informative
 - It has the highest **information gain** as the root node

Next Class

- Wrap-up Decision Trees
- Linear Models



Decision Trees (Wrap-up) and Linear Models

CS771: Introduction to Machine Learning

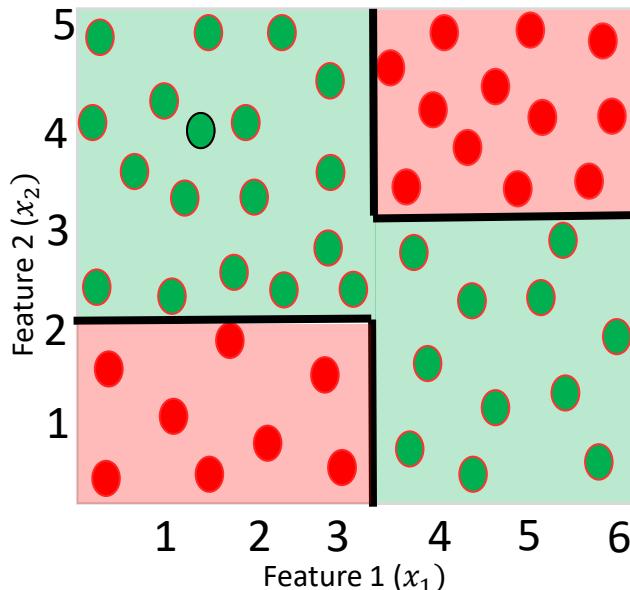
Piyush Rai

Plan for today

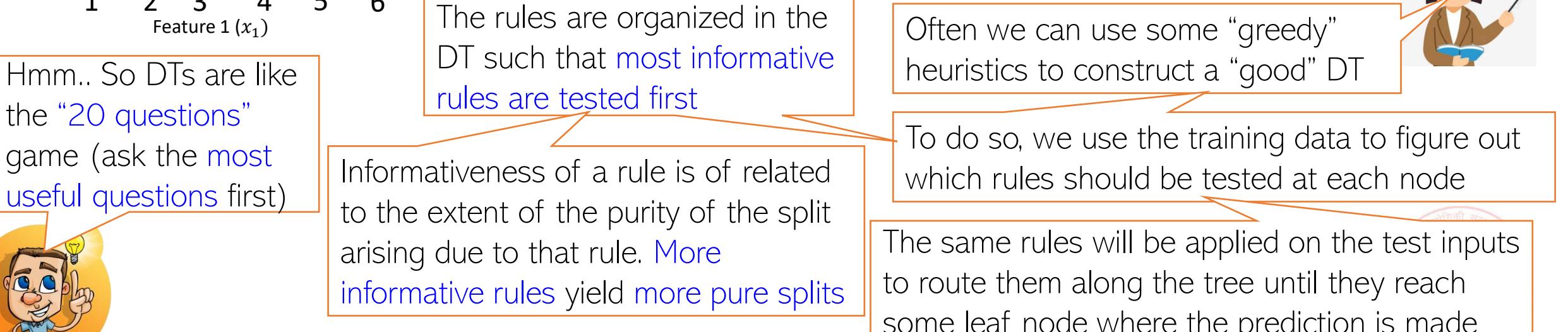
- Wrap-up the discussion of decision trees
 - How to learn decision trees from training data
- Introduction to Linear Models



Constructing Decision Trees



Hmm.. So DTs are like the “20 questions” game (ask the **most useful questions** first)



Given some training data, what's the “optimal” DT?



3

How to decide which rules to test for and in what order?

How to assess informativeness of a rule?

In general, constructing DT is an intractable problem (NP-hard)



Often we can use some “greedy” heuristics to construct a “good” DT

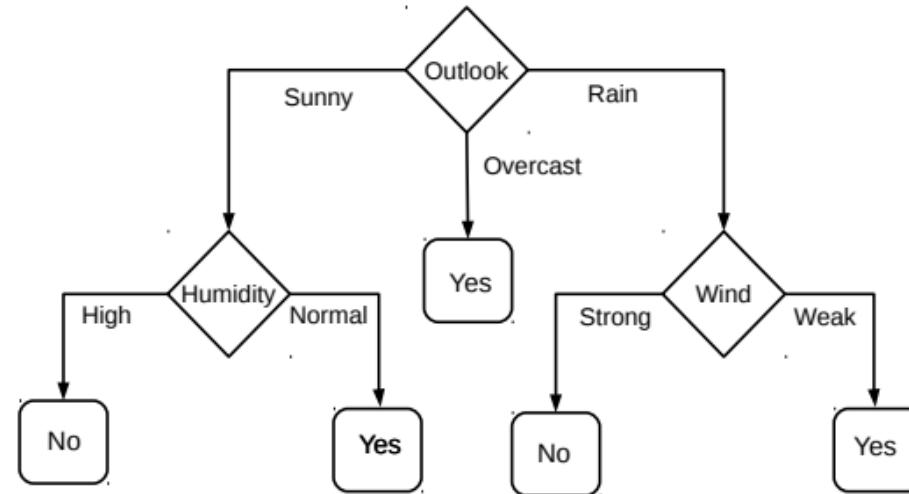
To do so, we use the training data to figure out which rules should be tested at each node

The same rules will be applied on the test inputs to route them along the tree until they reach some leaf node where the prediction is made

Decision Tree Construction: An Example

- Let's consider the playing Tennis example
- Assume each internal node will test the value of one of the features

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



- Question: Why does it make more sense to test the feature "outlook" first?
- Answer: Of all the 4 features, it's the most informative
 - It has the highest **information gain** as the root node

Entropy and Information Gain

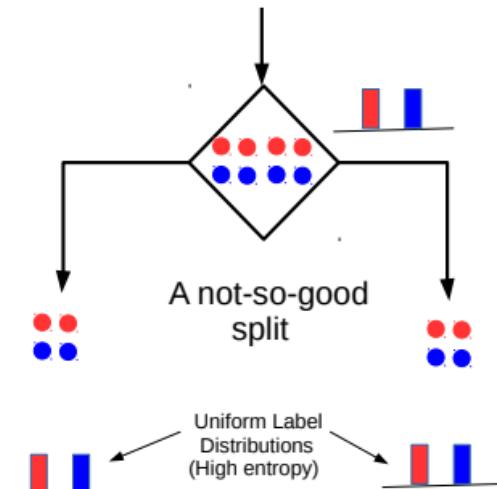
- Assume a set of labelled inputs \mathbf{S} from C classes, p_c as fraction of class c inputs
- Entropy of the set \mathbf{S} is defined as $H(\mathbf{S}) = - \sum_{c \in C} p_c \log p_c$
- Suppose a rule splits \mathbf{S} into two smaller disjoint sets \mathbf{S}_1 and \mathbf{S}_2
- Reduction in entropy after the split is called information gain

Uniform sets (all classes roughly equally present) have high entropy; skewed sets low

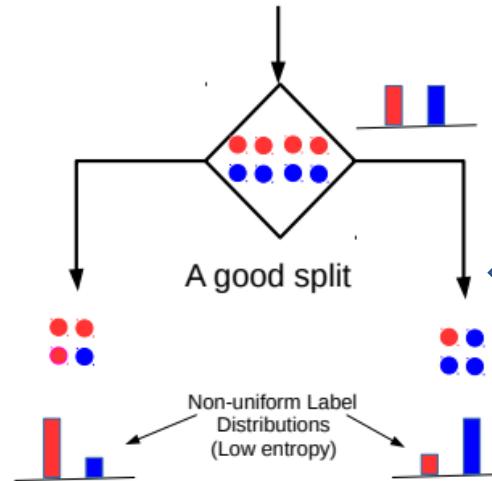


$$IG = H(S) - \frac{|S_1|}{|S|} H(S_1) - \frac{|S_2|}{|S|} H(S_2)$$

This split has a low IG
(in fact zero IG)



vs



This split has higher IG



Entropy and Information Gain

- Let's use IG based criterion to construct a DT for the Tennis example
- At root node, let's compute IG of each of the 4 features
- Consider feature "wind". Root contains all examples $S = [9+, 5-]$

$$H(S) = -(9/14) \log_2(9/14) - (5/14) \log_2(5/14) = 0.94$$

$$S_{\text{weak}} = [6+, 2-] \Rightarrow H(S_{\text{weak}}) = 0.811$$

$$S_{\text{strong}} = [3+, 3-] \Rightarrow H(S_{\text{strong}}) = 1$$

$$IG(S, \text{wind}) = H(S) - \frac{|S_{\text{weak}}|}{|S|} H(S_{\text{weak}}) - \frac{|S_{\text{strong}}|}{|S|} H(S_{\text{strong}}) = 0.94 - \frac{8}{14} * 0.811 - \frac{6}{14} * 1 = 0.048$$

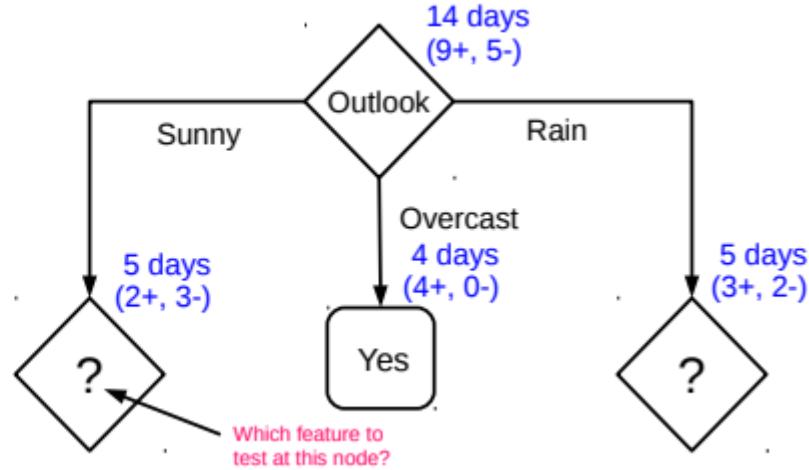
- Likewise, at root: $IG(S, \text{outlook}) = 0.246$, $IG(S, \text{humidity}) = 0.151$, $IG(S, \text{temp}) = 0.029$
- Thus we choose "outlook" feature to be tested at the root node
- Now how to grow the DT, i.e., what to do at the next level? Which feature to test next?
- Rule: Iterate - for each child node, select the feature with the highest IG

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



Growing the tree

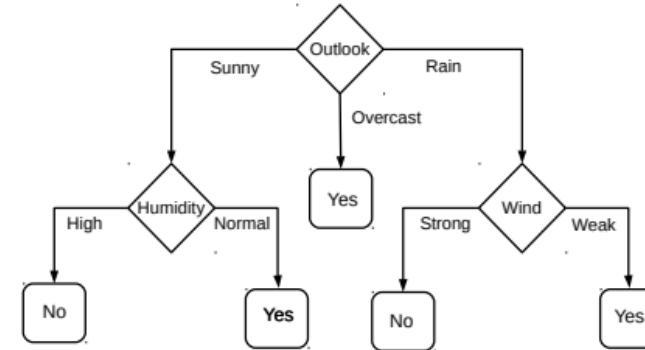
day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



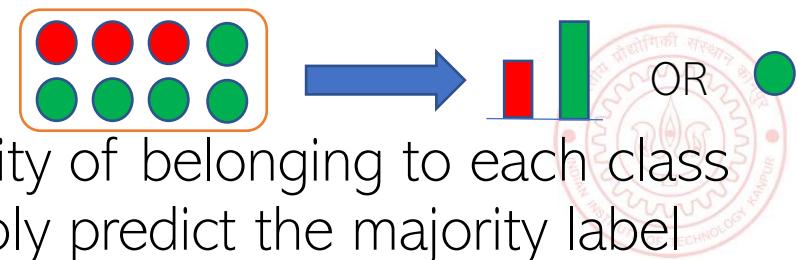
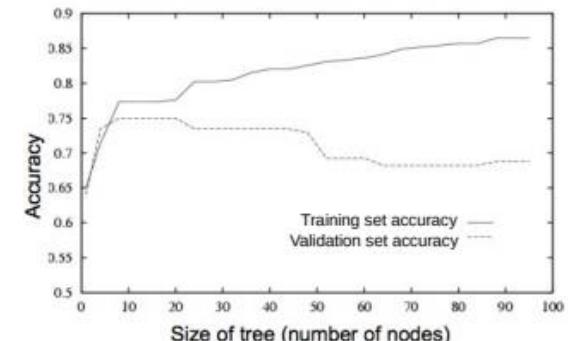
- Proceeding as before, for level 2, left node, we can verify that
 - $IG(S, \text{temp}) = 0.570$, $IG(S, \text{humidity}) = 0.970$, $IG(S, \text{wind}) = 0.019$
- Thus humidity chosen as the feature to be tested at level 2, left node
- No need to expand the middle node (already “pure” - all “yes” training examples ☺)
- Can also verify that wind has the largest IG for the right node
- Note: If a feature has already been tested along a path earlier, we don’t consider it again

When to stop growing the tree?

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



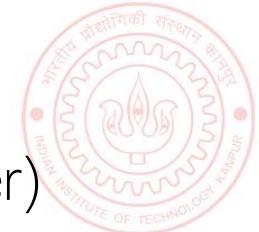
- Stop expanding a node further (i.e., make it a leaf node) when
 - It consist of all training examples having the same label (the node becomes “pure”)
 - We run out of features to test along the path to that node
 - The DT starts to overfit (can be checked by monitoring the validation set accuracy)
- To help prevent the tree from growing too much!
- Important:** No need to obsess too much for purity
 - It is okay to have a leaf node that is not fully pure, e.g., this
 - At test inputs that reach an impure leaf, can predict probability of belonging to each class (in above example, $p(\text{red}) = 3/8$, $p(\text{green}) = 5/8$), or simply predict the majority label



Avoiding Overfitting in DTs

- Desired: a DT that is not too big in size, yet fits the training data reasonably
- Note: An example of a very simple DT is “[decision-stump](#)”
 - A decision-stump only tests the value of a single feature (or a simple rule)
 - Not very powerful in itself but often used in large ensembles of decision stumps
- Mainly two approaches to prune a complex DT
 - Prune while building the tree (stopping early)
 - Prune after building the tree (post-pruning)
- Criteria for judging which nodes could potentially be pruned
 - Use a validation set (separate from the training set)
 - Prune each possible node that doesn't hurt the accuracy on the validation set
 - Greedily remove the node that improves the validation accuracy the most
 - Stop when the validation set accuracy starts worsening
 - Use model complexity control, such as Minimum Description Length (will see later)

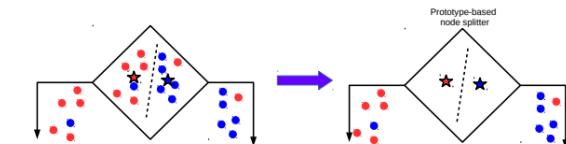
Either can be done
using a validation set



Decision Trees: Some Comments

- Gini-index defined as $\sum_{c=1}^C p_c(1 - p_c)$ can be an alternative to IG
- For DT regression¹, variance in the outputs can be used to assess purity
 - An illustration on the next slide
- When features are real-valued (no finite possible values to try), things are a bit more tricky
 - Can use tests based on thresholding feature values (recall our synthetic data examples)
 - Need to be careful w.r.t. number of threshold points, how fine each range is, etc.
- More sophisticated decision rules at the internal nodes can also be used
 - Basically, need some rule that splits inputs at an internal node into homogeneous groups
 - The rule can even be a machine learning classification algo (e.g., LwP or a deep learner)
 - However, in DTs, we want the tests to be fast so single feature based rules are preferred
- Need to take care handling training or test inputs that have some features missing

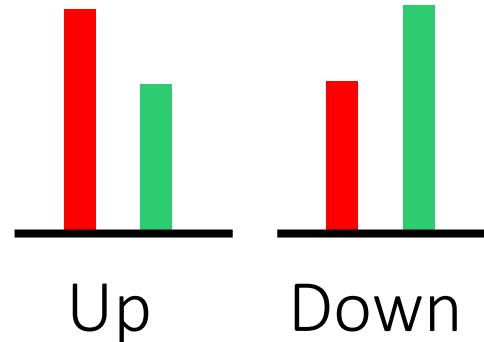
For regression, outputs are real-valued and we don't have a "set" of classes, so quantities like entropy/IG/gini etc. are undefined



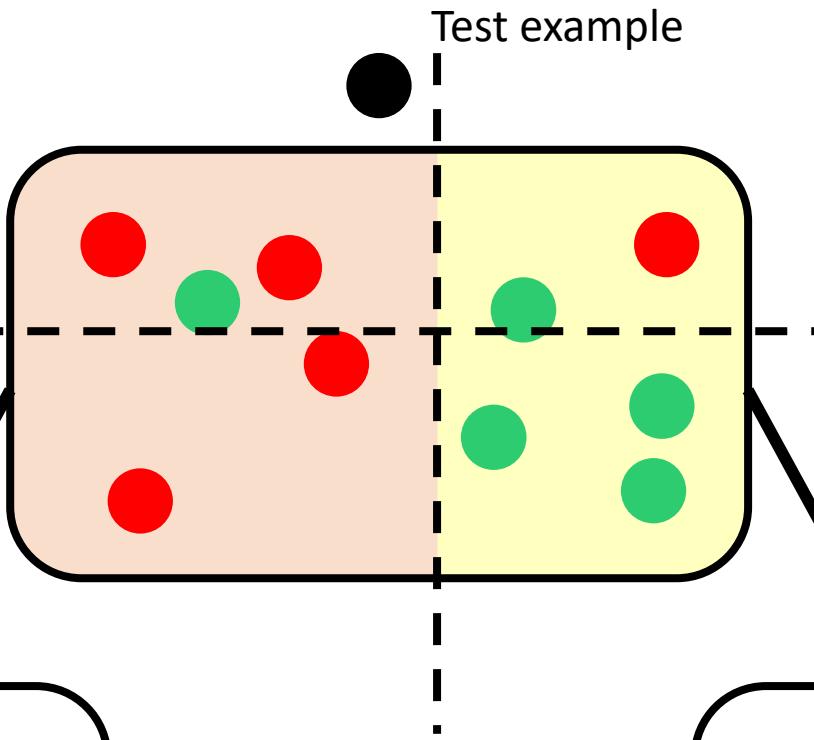
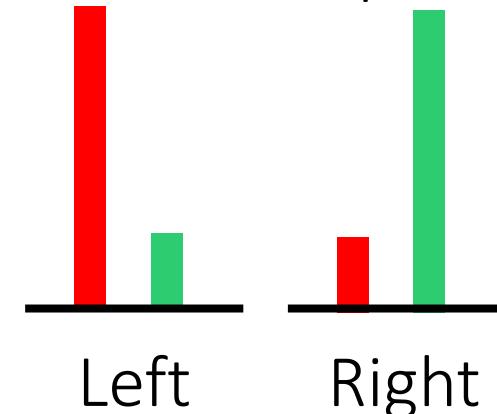
¹Breiman, Leo; Friedman, J. H.; Olshen, R. A.; Stone, C. J. (1984). Classification and regression trees

An Illustration: DT with Real-Valued Features

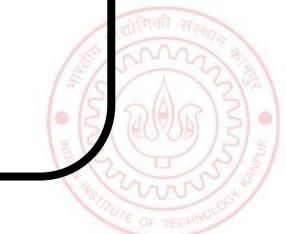
“Best” (purest possible)
Horizontal Split



“Best”(purest possible)
Vertical Split



Between the best horizontal vs best vertical split, the vertical split is better (more pure), hence we use this rule for the internal node

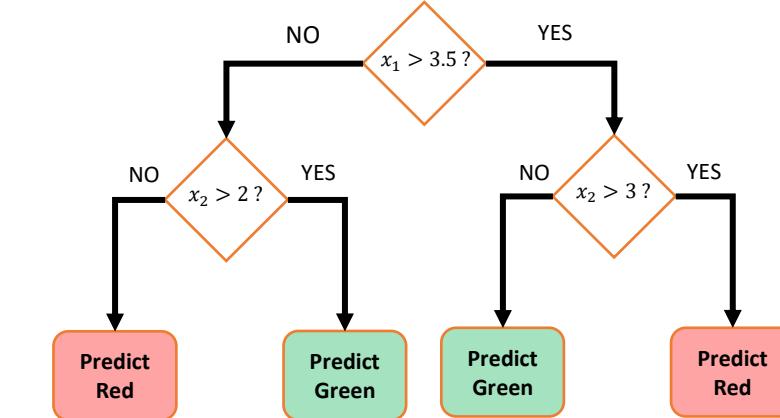
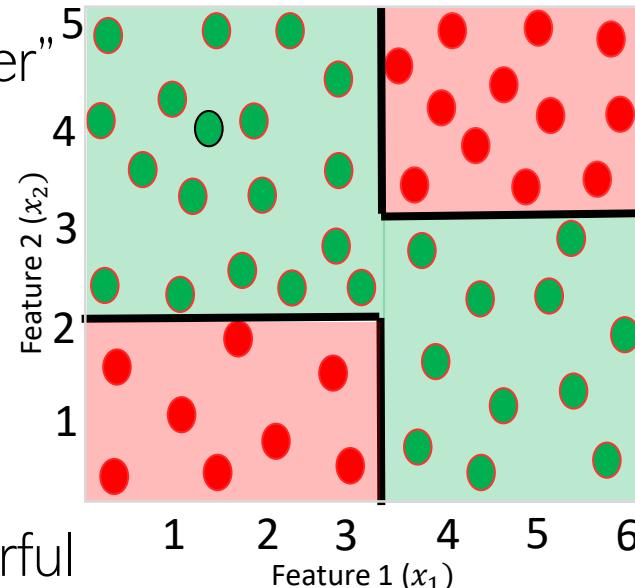


Decision Trees: A Summary

Some key strengths:

- Simple and easy to interpret
- Nice example of “divide and conquer” paradigm in machine learning
- Easily handle different types of features (real, categorical, etc.)
- Very fast at test time
- Multiple DTs can be combined via **ensemble methods**: more powerful (e.g., Decision Forests; will see later)
- Used in several real-world ML applications, e.g., recommender systems, gaming (Kinect)

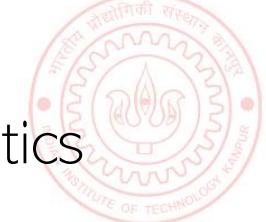
.. thus helping us learn complex rule as a combination of several simpler rules



Human-body pose estimation

Some key weaknesses:

- Learning optimal DT is (NP-hard) intractable. Existing algos mostly greedy heuristics
- Can sometimes become very complex unless some pruning is applied



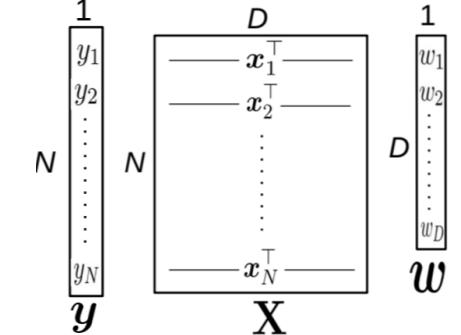
Linear Regression

CS771: Introduction to Machine Learning
Piyush Rai

Linear Regression

- Given: Training data with N input-output pairs $\{(\mathbf{x}_n, y_n)\}_{n=1}^N$, $\mathbf{x}_n \in \mathbb{R}^D$, $y_n \in \mathbb{R}$

- Goal: Learn a model to predict the output for new test inputs



- Assume the function that approximates the I/O relationship to be a linear model

$$y_n \approx f(\mathbf{x}_n) = \mathbf{w}^\top \mathbf{x}_n \quad (n = 1, 2, \dots, N)$$

Can also write all of them compactly using matrix-vector notation as $\mathbf{y} \approx \mathbf{X}\mathbf{w}$

- Let's write the total error or "loss" of this model over the training data as

Goal of learning is to find the \mathbf{w} that minimizes this loss + does well on test data

$$L(\mathbf{w}) = \sum_{n=1}^N \ell(y_n, \mathbf{w}^\top \mathbf{x}_n)$$

Unlike models like KNN and DT, here we have an explicit problem-specific objective (loss function) that we wish to optimize for

$\ell(y_n, \mathbf{w}^\top \mathbf{x}_n)$ measures the prediction error or "loss" or "deviation" of the model on a single training input (\mathbf{x}_n, y_n)



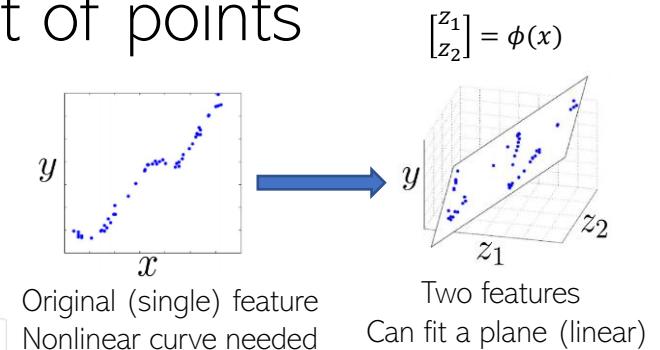
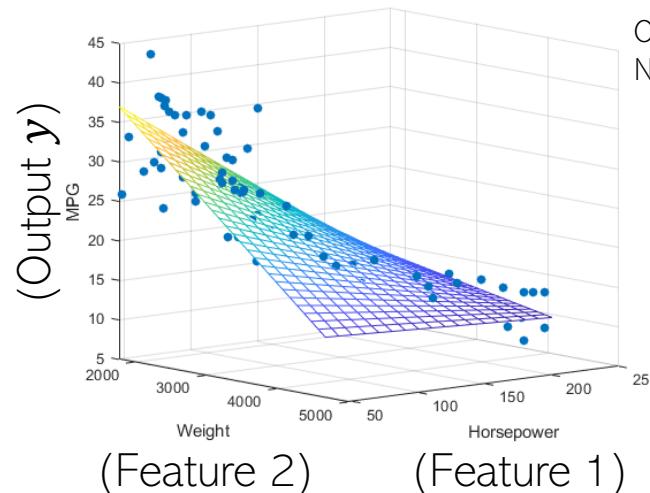
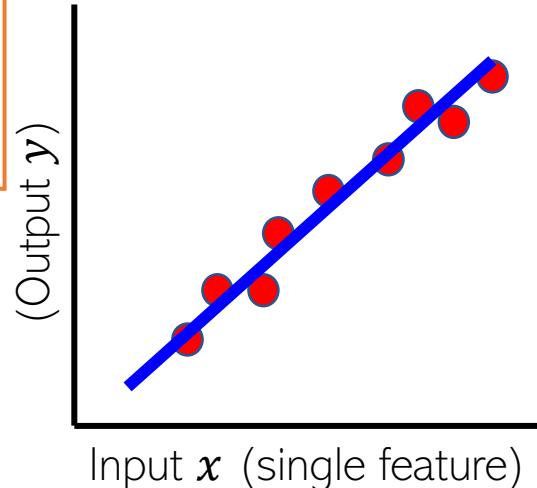
Linear Regression: Pictorially

- Linear regression is like fitting a line or (hyper)plane to a set of points

What if a line/plane doesn't model the input-output relationship very well, e.g., if their relationship is better modeled by a nonlinear curve or curved surface?



Do linear models become useless in such cases?



No. We can even fit a curve using a linear model after suitably transforming the inputs
 $y \approx \mathbf{w}^T \phi(\mathbf{x})$

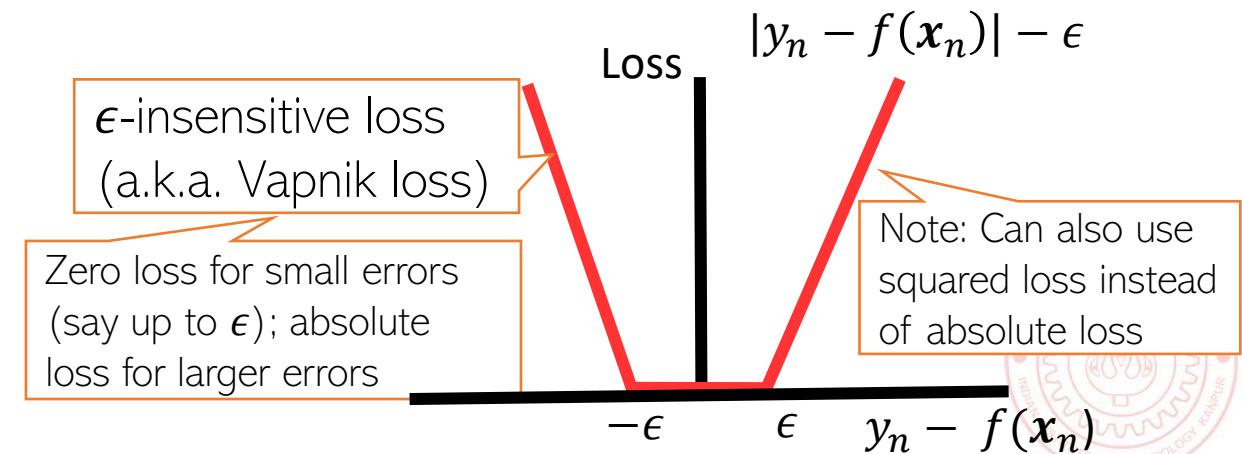
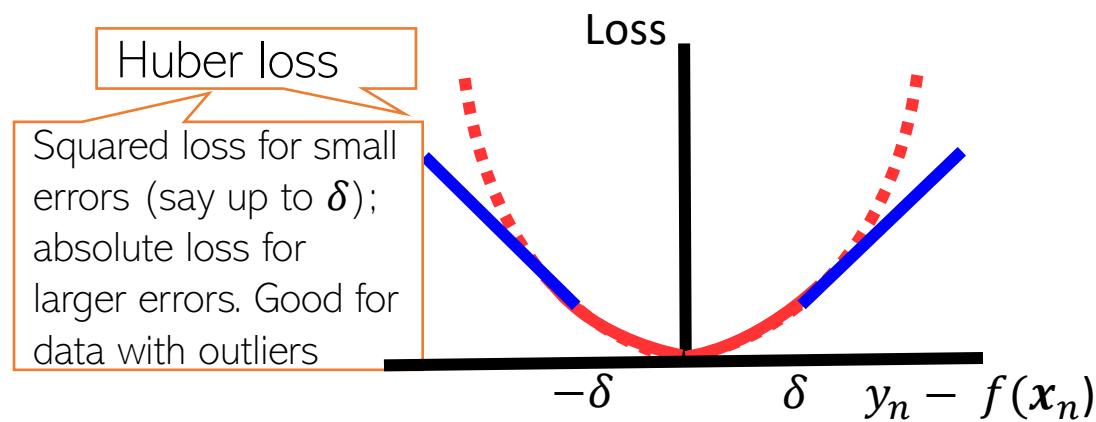
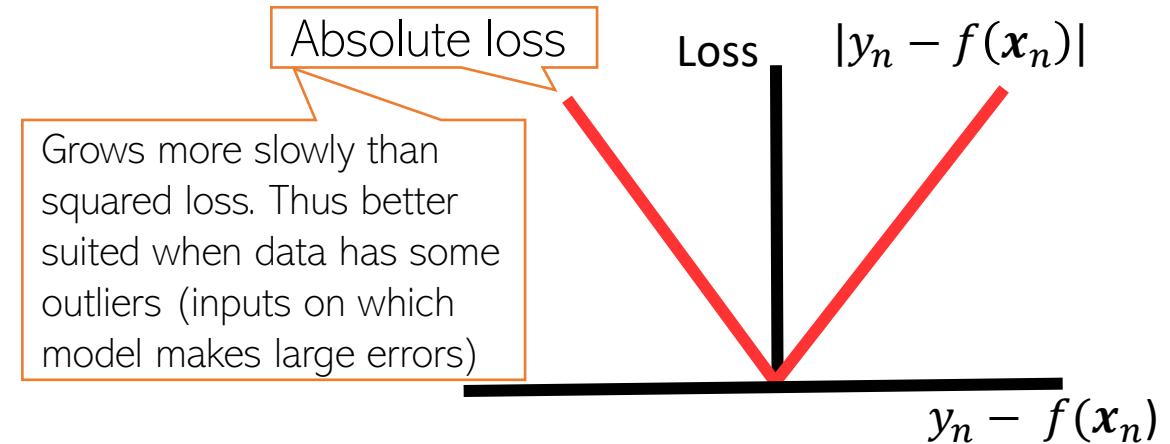
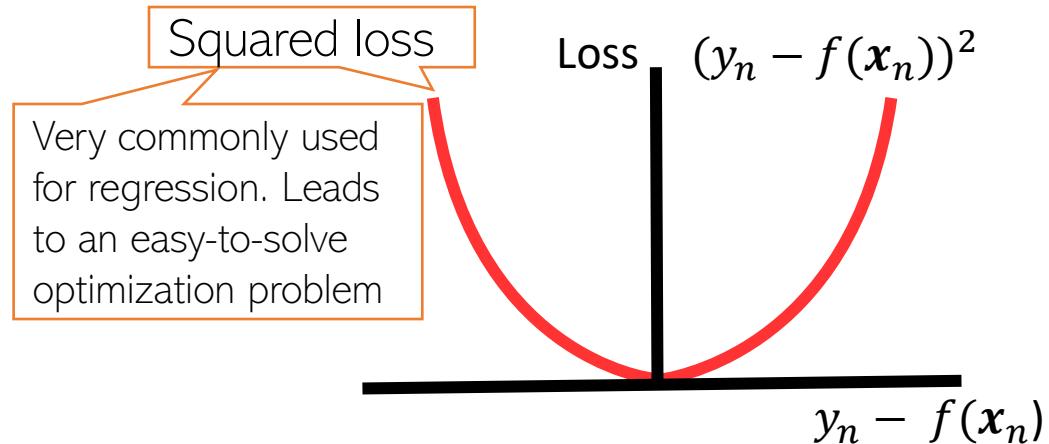
The transformation $\phi(\cdot)$ can be predefined or learned (e.g., using [kernel methods](#) or a deep neural network based feature extractor). More on this later

- The line/plane must also predict outputs the unseen (test) inputs well

Loss Functions for Regression

- Many possible loss functions for regression problems

Choice of loss function usually depends on the nature of the data. Also, some loss functions result in easier optimization problem than others



Linear Regression with Squared Loss

- In this case, the loss func will be

In matrix-vector notation, can write it compactly
as $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 = (\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w})$

$$L(\mathbf{w}) = \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2$$

- Let us find the \mathbf{w} that optimizes (minimizes) the above squared loss

- We need calculus and optimization to do this!

The “least squares” (LS) problem
Gauss-Legendre, 18th century)

- The LS problem can be solved easily and has a **closed form** solution

$$\mathbf{w}_{LS} = \arg \min_{\mathbf{w}} L(\mathbf{w}) = \arg \min_{\mathbf{w}} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2$$

Closed form
solutions to ML
problems are rare.

$$\mathbf{w}_{LS} = \left(\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top \right)^{-1} \left(\sum_{n=1}^N y_n \mathbf{x}_n \right) = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$



$D \times D$ matrix inversion – can be expensive.
Ways to handle this. Will see later

Proof: A bit of calculus/optim. (more on this later)

- We wanted to find the minima of $L(\mathbf{w}) = \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2$
- Let us apply basic rule of calculus: Take first derivative of $L(\mathbf{w})$ and set to zero

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 = \sum_{n=1}^N 2(y_n - \mathbf{w}^\top \mathbf{x}_n) \frac{\partial}{\partial \mathbf{w}} (y_n - \mathbf{w}^\top \mathbf{x}_n) = 0$$

Partial derivative of dot product w.r.t each element of \mathbf{w}

Chain rule of calculus

Result of this derivative is \mathbf{x}_n - same size as \mathbf{w}

- Using the fact $\frac{\partial}{\partial \mathbf{w}} \mathbf{w}^\top \mathbf{x}_n = \mathbf{x}_n$, we get $\sum_{n=1}^N 2(y_n - \mathbf{w}^\top \mathbf{x}_n)\mathbf{x}_n = 0$
- To separate \mathbf{w} to get a solution, we write the above as

$$\sum_{n=1}^N 2\mathbf{x}_n(y_n - \mathbf{x}_n^\top \mathbf{w}) = 0 \quad \rightarrow \quad \sum_{n=1}^N y_n \mathbf{x}_n - \mathbf{x}_n \mathbf{x}_n^\top \mathbf{w} = 0$$

$$\mathbf{w}_{LS} = (\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top)^{-1} (\sum_{n=1}^N y_n \mathbf{x}_n) = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$



Problem(s) with the Solution!

- We minimized the objective $L(\mathbf{w}) = \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2$ w.r.t. \mathbf{w} and got

$$\mathbf{w}_{LS} = (\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top)^{-1} (\sum_{n=1}^N y_n \mathbf{x}_n) = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

- Problem: The matrix $\mathbf{X}^\top \mathbf{X}$ may not be invertible
 - This may lead to non-unique solutions for \mathbf{w}_{opt}
 - Problem: Overfitting since we only minimized loss defined on training data
 - Weights $\mathbf{w} = [w_1, w_2, \dots, w_D]$ may become arbitrarily large to fit training data perfectly
 - Such weights may perform poorly on the test data however
 - One Solution: Minimize a **regularized objective** $L(\mathbf{w}) + \lambda R(\mathbf{w})$
 - The reg. will prevent the elements of \mathbf{w} from becoming too large
 - Reason: Now we are minimizing **training error** + **magnitude of vector \mathbf{w}**
- $R(\mathbf{w})$ is called the **Regularizer** and measures the “magnitude” of \mathbf{w}
- $\lambda \geq 0$ is the reg. hyperparam. Controls how much we wish to regularize (needs to be tuned via cross-validation)

Regularized Least Squares (a.k.a. Ridge Regression)⁸

- Recall that the regularized objective is of the form $L_{reg}(\mathbf{w}) = L(\mathbf{w}) + \lambda R(\mathbf{w})$
- One possible/popular regularizer: the squared Euclidean (ℓ_2 squared) norm of \mathbf{w}

$$R(\mathbf{w}) = \|\mathbf{w}\|_2^2 = \mathbf{w}^\top \mathbf{w}$$

- With this regularizer, we have the regularized least squares problem as



Why is the method
called "ridge" regression

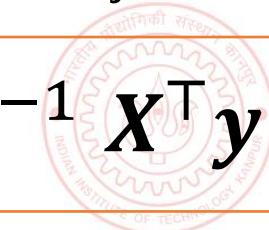
$$\begin{aligned}\mathbf{w}_{ridge} &= \arg \min_{\mathbf{w}} L(\mathbf{w}) + \lambda R(\mathbf{w}) \\ &= \arg \min_{\mathbf{w}} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \lambda \mathbf{w}^\top \mathbf{w}\end{aligned}$$



Look at the form of the solution. We are adding a small value λ to the diagonals of the DxD matrix $\mathbf{X}^\top \mathbf{X}$ (like adding a ridge/mountain to some land)

- Proceeding just like the LS case, we can find the optimal \mathbf{w} which is given by

$$\mathbf{w}_{ridge} = (\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top + \lambda I_D)^{-1} (\sum_{n=1}^N y_n \mathbf{x}_n) = (\mathbf{X}^\top \mathbf{X} + \lambda I_D)^{-1} \mathbf{X}^\top \mathbf{y}$$



A closer look at ℓ_2 regularization

- The regularized objective we minimized is

$$L_{reg}(\mathbf{w}) = \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \lambda \mathbf{w}^\top \mathbf{w}$$

- Minimizing $L_{reg}(\mathbf{w})$ w.r.t. \mathbf{w} gives a solution for \mathbf{w} that

- Keeps the training error small
- Has a small ℓ_2 squared norm $\mathbf{w}^\top \mathbf{w} = \sum_{d=1}^D w_d^2$

Good because, consequently, the individual entries of the weight vector \mathbf{w} are also prevented from becoming too large

- Small entries in \mathbf{w} are good since they lead to “smooth” models

$\mathbf{x}_n =$	1.2	0.5	2.4	0.3	0.8	0.1	0.9	2.1
$\mathbf{x}_m =$	1.2	0.5	2.4	0.3	$0.8 + \epsilon$	0.1	0.9	2.1

Exact same feature vectors only differing in just one feature by a small amount

$$\begin{aligned} y_n &= 0.8 \\ y_m &= 100 \end{aligned}$$

Very different outputs though (maybe one of these two training ex. is an outlier)

A typical \mathbf{w} learned without ℓ_2 reg.	3.2	1.8	1.3	2.1	10000	2.5	3.1	0.1
--	-----	-----	-----	-----	-------	-----	-----	-----

Just to fit the training data where one of the inputs was possibly an outlier, this weight became too big. Such a weight vector will possibly do poorly on normal test inputs



Remember – in general, weights with large magnitude are bad since they can cause overfitting on training data and may not work well on test data

Not a “smooth” model since its test data predictions may change drastically even with small changes in some feature’s value

Other Ways to Control Overfitting

- Use a regularizer $R(\mathbf{w})$ defined by other norms, e.g.,



ℓ_1 norm regularizer

$$\|\mathbf{w}\|_1 = \sum_{d=1}^D |w_d|$$

When should I used these regularizers instead of the ℓ_2 regularizer?

Automatic feature selection? Wow, cool!!! But how exactly?

$$\|\mathbf{w}\|_0 = \#\text{nnz}(\mathbf{w})$$

ℓ_0 norm regularizer (counts number of nonzeros in \mathbf{w})

Use them if you have a very large number of features but many irrelevant features. These regularizers can help in **automatic feature selection**



Using such regularizers gives a **sparse** weight vector \mathbf{w} as solution

sparse means many entries in \mathbf{w} will be zero or near zero. Thus those features will be considered irrelevant by the model and will not influence prediction

- Use non-regularization based approaches

- Early-stopping (stopping training just when we have a decent val. set accuracy)
- Dropout (in each iteration, don't update some of the weights)
- Injecting noise in the inputs

All of these are very popular ways to control overfitting in deep learning models. More on these later when we talk about deep learning

Linear Regression as Solving System of Linear Eqs

- The form of the lin. reg. model $\mathbf{y} \approx \mathbf{X}\mathbf{w}$ is akin to a system of linear equation
- Assuming N training examples with D features each, we have

First training example: $y_1 = x_{11}w_1 + x_{12}w_2 + \dots + x_{1D}w_D$

Second training example: $y_2 = x_{21}w_1 + x_{22}w_2 + \dots + x_{2D}w_D$

⋮

N -th training example: $y_N = x_{N1}w_1 + x_{N2}w_2 + \dots + x_{ND}w_D$

Note: Here x_{nd} denotes the d^{th} feature of the n^{th} training example

N equations and D unknowns here (w_1, w_2, \dots, w_D)

- However, in regression, we rarely have $N = D$ but rather $N > D$ or $N < D$
 - Thus we have an **underdetermined** ($N < D$) or **overdetermined** ($N > D$) system
 - Methods to solve over/underdetermined systems can be used for lin-reg as well
 - Many of these methods don't require expensive matrix inversion

Solving lin-reg
as system of lin eq.

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

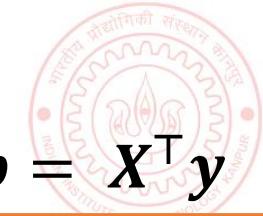


$$\boxed{\mathbf{A}\mathbf{w} = \mathbf{b}}$$

Now solve this!

where $\mathbf{A} = \mathbf{X}^\top \mathbf{X}$, and $\mathbf{b} = \mathbf{X}^\top \mathbf{y}$

System of lin. Eqns with D equations and D unknowns



Next Lecture

- Solving linear regression using iterative optimization methods
 - Faster and don't require matrix inversion
- Brief intro to optimization techniques



Essential Calculus and Optimization for ML (1)

CS771: Introduction to Machine Learning

Piyush Rai

Calculus and Optimization for ML

- Regularized Linear Regression (a.k.a. Ridge Regression)

$$\mathbf{w}_{ridge} = \arg \min_{\mathbf{w}} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \lambda \mathbf{w}^\top \mathbf{w} = (\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top + \lambda I_D)^{-1} (\sum_{n=1}^N y_n \mathbf{x}_n)$$

Problem more compactly written as $\arg \min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_2^2$

Solution more compactly as $(\mathbf{X}^\top \mathbf{X} + \lambda I_D)^{-1} \mathbf{X}^\top \mathbf{y}$

- Getting closed-form soln required simple calculus, but is expensive to compute
 - Especially when D is very large (since we need to invert a $D \times D$ matrix)
- How to solve this and other (possibly more difficult) optimization problems arising in ML efficiently?
- What's the basic calculus and optimization knowledge we need for ML?



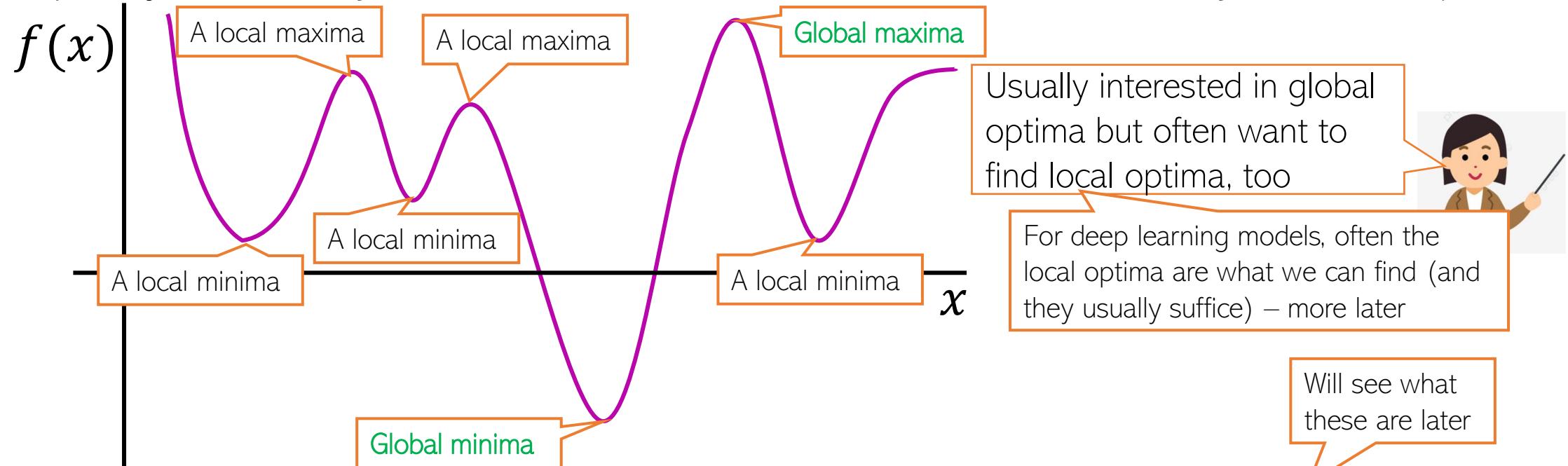
Functions and their optima

3

The objective function of the ML problem we are solving (e.g., squared loss for regression)

Assume unconstrained for now, i.e., just a real-valued number/vector

- Many ML problems require us to optimize a function f of some variable(s) x
- For simplicity, assume f is a scalar-valued function of a scalar x ($f: \mathbb{R} \rightarrow \mathbb{R}$)



- Any function has one/more optima (maxima, minima), and maybe saddle points
- Finding the optima or saddles requires derivatives/gradients of the function

Derivatives

Will sometimes use $f'(x)$ to denote the derivative

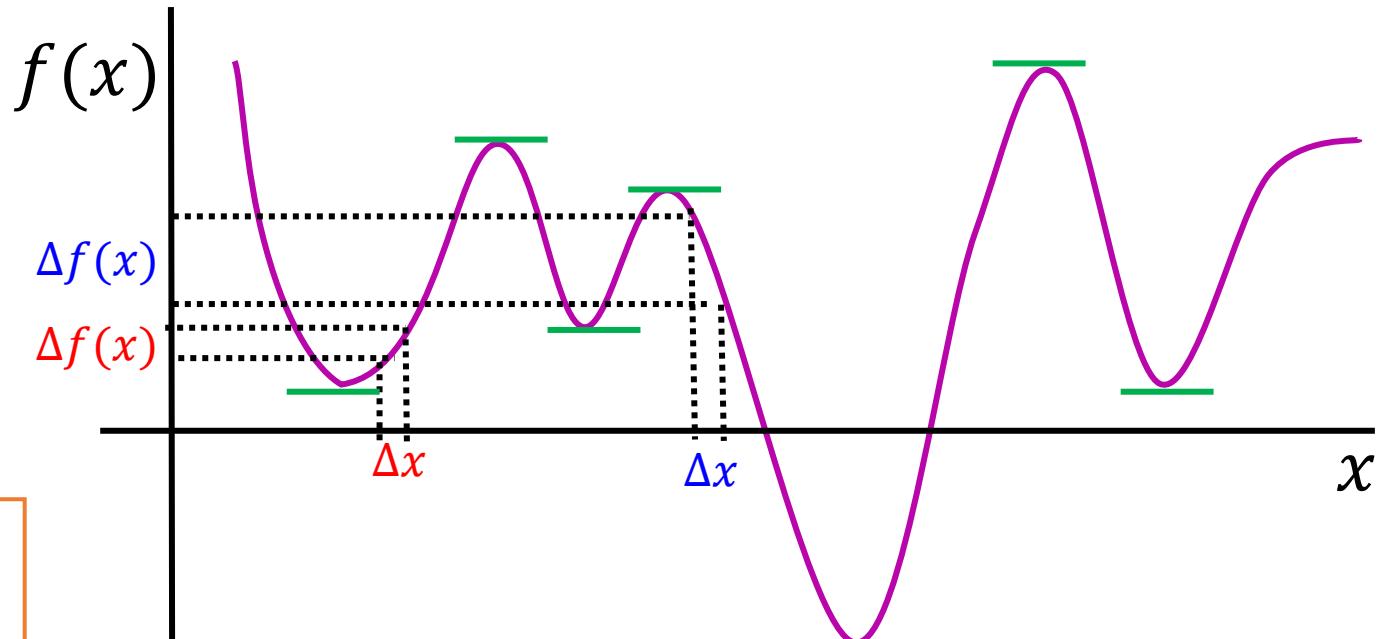


- Magnitude of derivative at a point is the rate of change of the func at that point

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{\Delta f(x)}{\Delta x}$$

Sign is also important: Positive derivative means f is **increasing** at x if we increase the value of x by a very small amount; negative derivative means it is **decreasing**

Understanding how f changes its value as we change x is helpful to understand optimization (minimization/maximization) algorithms



- Derivative becomes zero at stationary points (optima or saddle points)
 - The function becomes “flat” ($\Delta f(x) = 0$ if we change x by a very little at such points)
 - These are the points where the function has its maxima/minima (unless they are saddles)



Rules of Derivatives

Some basic rules of taking derivatives

- Sum Rule: $(f(x) + g(x))' = f'(x) + g'(x)$
- Scaling Rule: $(a \cdot f(x))' = a \cdot f'(x)$ if a is not a function of x
- Product Rule: $(f(x) \cdot g(x))' = f'(x) \cdot g(x) + g'(x) \cdot f(x)$
- Quotient Rule: $(f(x)/g(x))' = (f'(x) \cdot g(x) - g'(x)f(x))/(g(x))^2$
- Chain Rule: $(f(g(x)))' \stackrel{\text{def}}{=} (f \circ g)'(x) = f'(g(x)) \cdot g'(x)$

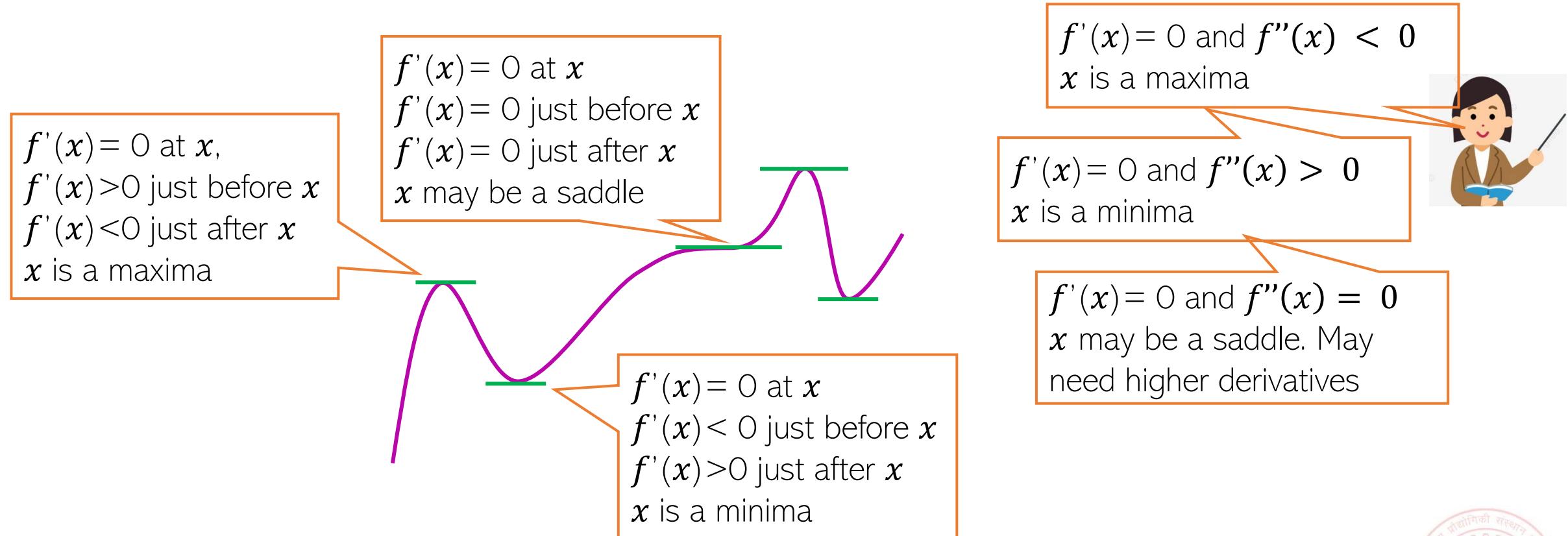


We already used some of these (sum, scaling and chain) when calculating the derivative for the linear regression model



Derivatives

- How the derivative itself changes tells us about the function's optima

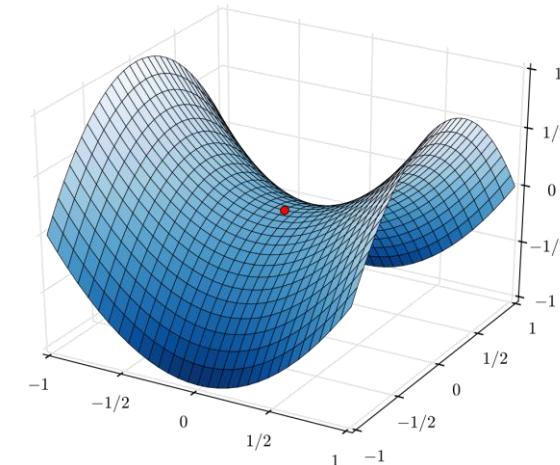
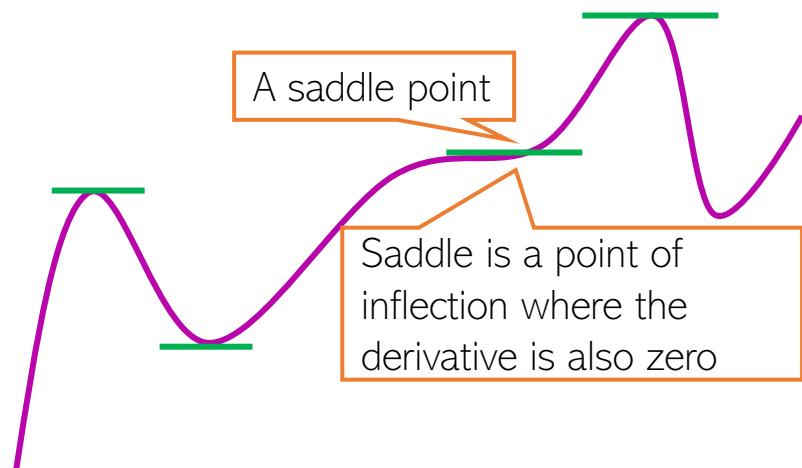


- The second derivative $f''(x)$ can provide this information

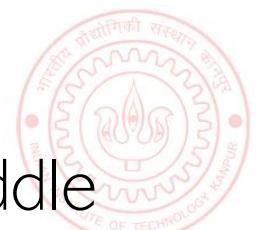


Saddle Points

- Points where derivative is zero but are neither minima nor maxima



- Saddle points are very common for loss functions of deep learning models
 - Need to be handled carefully during optimization
- Second or higher derivative may help identify if a stationary point is a saddle

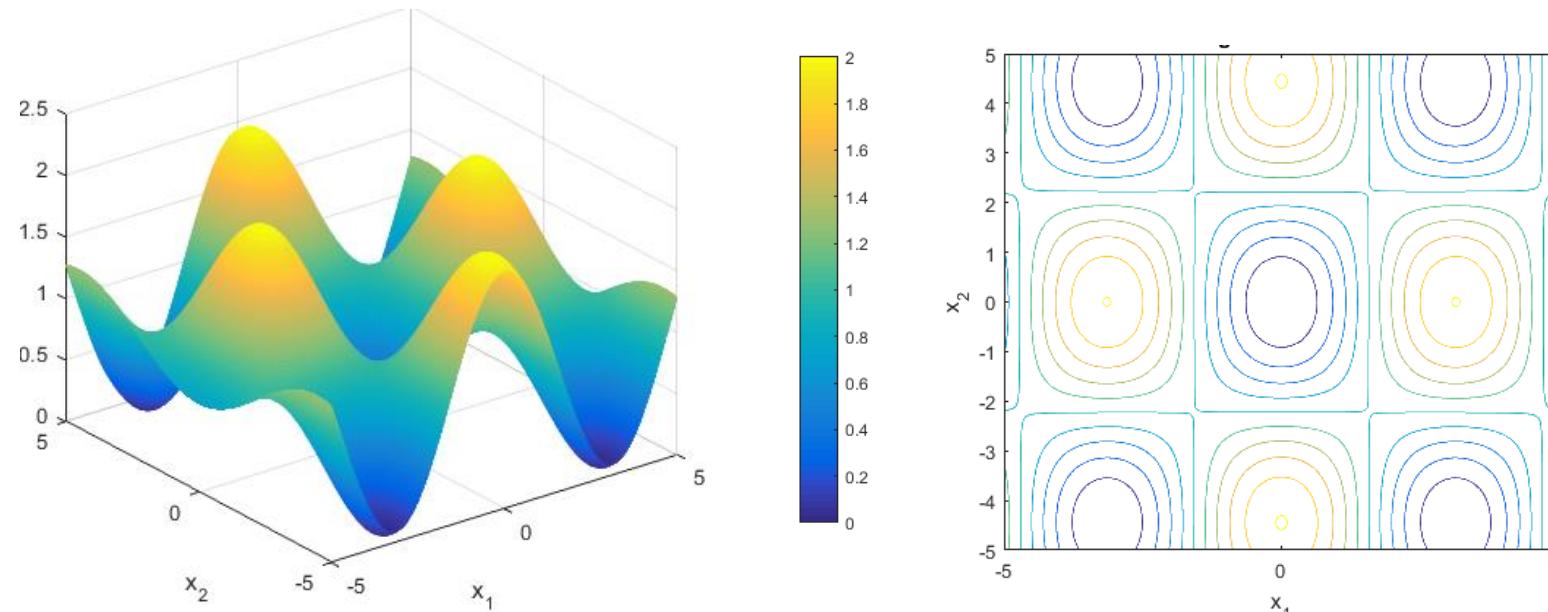


Multivariate Functions

- Most functions that we see in ML are multivariate function
- Example: Loss fn $L(\mathbf{w})$ in lin-reg was a multivar function of D -dim vector \mathbf{w}

$$L(\mathbf{w}): \mathbb{R}^D \rightarrow \mathbb{R}$$

- Here is an illustration of a function of 2 variables (4 maxima and 5 minima)



Derivatives of Multivariate Functions

- Can define derivative for a multivariate functions as well via the gradient
 - Gradient of a function $f(\mathbf{x}) : \mathbb{R}^D \rightarrow \mathbb{R}$ is a $D \times 1$ vector of partial derivatives
- $$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_D} \right)$$

Each element in this gradient vector tells us how much f will change if we move a little along the corresponding (akin to one-dim case)
- Optima and saddle points defined similar to one-dim case
 - Required properties that we saw for one-dim case must be satisfied along all the directions
 - The second derivative in this case is known as the Hessian



The Hessian

- For a multivar scalar valued function $f(\mathbf{x}): \mathbb{R}^D \rightarrow \mathbb{R}$, Hessian is a $D \times D$ matrix

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_D} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \dots & \frac{\partial^2 f}{\partial x_2 \partial x_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_D \partial x_1} & \frac{\partial^2 f}{\partial x_D \partial x_2} & \dots & \frac{\partial^2 f}{\partial x_D^2} \end{bmatrix}$$

Gives information about the **curvature** of the function at point \mathbf{x}

Note: If the function itself is vector valued, e.g., $f(\mathbf{x}): \mathbb{R}^D \rightarrow \mathbb{R}^K$ then we will have K such $D \times D$ Hessian matrices, one for each output dimension of f

A square, symmetric $D \times D$ matrix \mathbf{M} is PSD if $\mathbf{x}^\top \mathbf{M} \mathbf{x} \geq \mathbf{0} \quad \forall \mathbf{x} \in \mathbb{R}^D$
Will be NSD if $\mathbf{x}^\top \mathbf{M} \mathbf{x} \leq \mathbf{0} \quad \forall \mathbf{x} \in \mathbb{R}^D$



PSD if all eigenvalues are non-negative

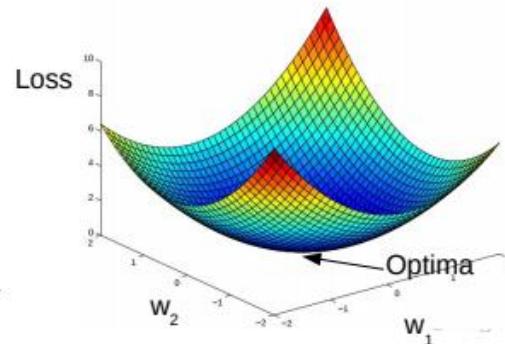
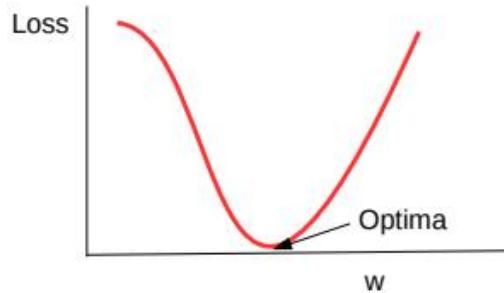
- The Hessian matrix can be used to assess the optima/saddle points

- $\nabla f(\mathbf{x}) = \mathbf{0}$ and $\nabla^2 f(\mathbf{x})$ is a positive semi-definite (PSD) matrix then \mathbf{x} is a minima
- $\nabla f(\mathbf{x}) = \mathbf{0}$, and $\nabla^2 f(\mathbf{x})$ is a negative semi-definite (NSD) matrix then \mathbf{x} is a maxima



Convex and Non-Convex Functions

- A function being optimized can be either **convex** or **non-convex**
- Here are a couple of examples of convex functions

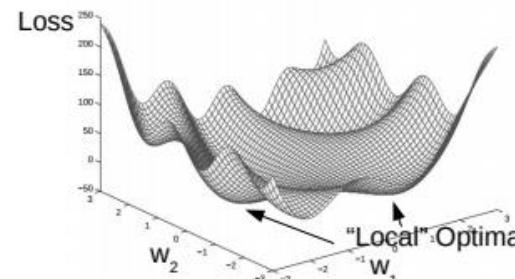
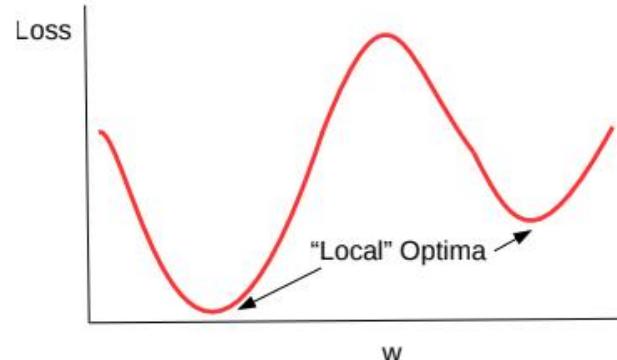


Convex functions are bowl-shaped.
They have a unique optima (minima)



Negative of a convex function is called
a **concave** function, which also has a
unique optima (maxima)

- Here are a couple of examples of non-convex functions



Non-convex functions have
multiple minima. Usually harder
to optimize as compared to
convex functions



Loss functions of most
deep learning models are
non-convex

Convex Sets

- A set S of points is a convex set, if for any two points $x, y \in S$, and $0 \leq \alpha \leq 1$

z is also called a “convex combination” of two points

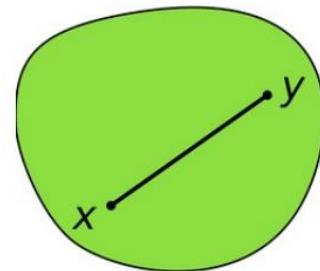
$$z = \alpha x + (1 - \alpha)y \in S$$

Can also define convex combination of N points x_1, x_2, \dots, x_N as $z = \sum_{i=1}^N \alpha_i x_i$

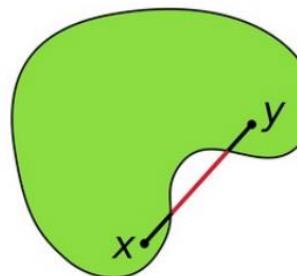


- Above means that all points on the line-segment between x and y lie within S

A Convex Set



A Non-convex Set

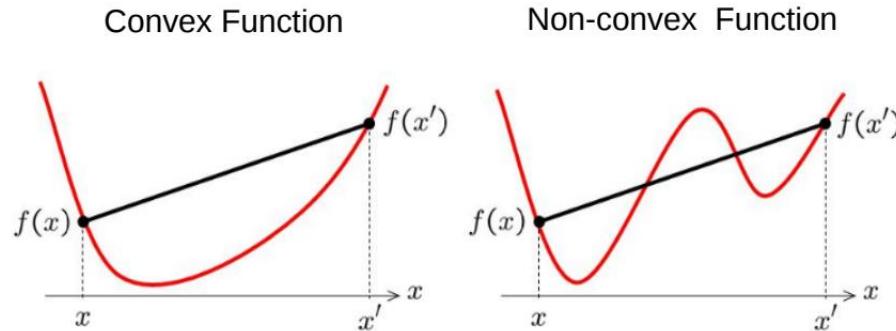


- The domain of a convex function needs to be a convex set



Convex Functions

- Informally, $f(x)$ is convex if all of its chords lie above the function everywhere



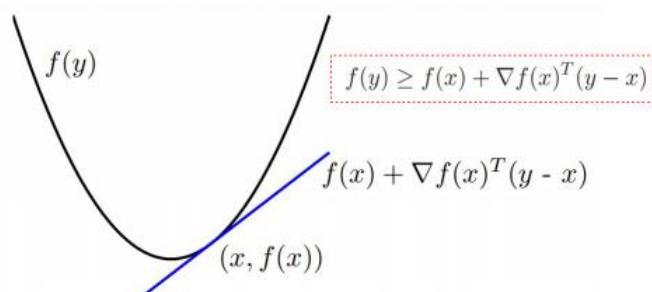
Note: "Chord lies above function" more formally means

If f is convex then given
 $\alpha_1, \dots, \alpha_n$ s.t $\sum_{i=1}^n \alpha_i = 1$

$$f\left(\sum_{i=1}^n \alpha_i x_i\right) \leq \sum_{i=1}^n \alpha_i f(x_i)$$

Jensen's Inequality

- Formally, (assuming differentiable function), some tests for convexity:
 - First-order convexity (graph of f must be above all the tangents)



- Second derivative a.k.a. Hessian (if exists) must be positive semi-definite

Some Basic Rules for Convex Functions

- Some basic rules to check if $f(x)$ is convex or not

- All linear and affine functions (e.g., $ax + b$) are convex
- $\exp(ax)$ is convex for $x \in \mathbb{R}$, for any $a \in \mathbb{R}$
- $\log(x)$ is concave (not convex) for $x > 0$
- x^a is convex for $x > 0$, for any $a \geq 1$ and $a < 0$, concave for $0 \leq a \leq 1$
- $|x|^a$ is convex for $x \in \mathbb{R}$, for any $a \geq 1$
- All norms in \mathbb{R}^D are convex
- Non-negative weighted sum of convex functions is also a convex function
- Affine transformation preserves convexity: if $f(x)$ is convex then $f(x) = f(ax + b)$ is also convex
- Some rules to check whether composition $f(x) = h(g(x))$ of two functions h and g is convex

f is convex if h is convex and nondecreasing, and g is convex,

f is convex if h is convex and nonincreasing, and g is concave,

f is concave if h is concave and nondecreasing, and g is concave,

f is concave if h is concave and nonincreasing, and g is convex.



Coming up next

- Gradients when the function is non-differentiable
- Solving optimization problems
- Iterative optimization algorithms, such as gradient descent and its variants



Optimization for ML (2)

CS771: Introduction to Machine Learning

Piyush Rai

The Plan

- Some basic techniques for solving optimization problems
 - First-order optimality
 - Gradient descent
- Dealing with non-differentiable functions
 - Sub-gradients and sub-differential



Optimization Problems in ML

- The general form of an optimization problem in ML will usually be

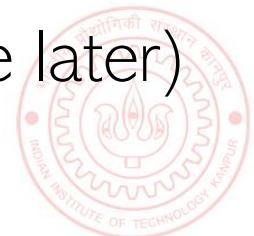
Usually a sum of the
training error + regularizer

$$\mathbf{w}_{opt} = \arg \min_{\mathbf{w} \in \mathcal{C}} L(\mathbf{w})$$

- Here $L(\mathbf{w})$ denotes the loss function to be optimized
- \mathcal{C} is the constraint set that the solution must belong to, e.g.,
 - Non-negativity constraint: All entries in \mathbf{w}_{opt} must be non-negative
 - Sparsity constraint: \mathbf{w}_{opt} is a sparse vector with atmost K non-zeros
- If no \mathcal{C} is specified, it is an unconstrained optimization problem
- Constrained opt. probs can be converted into unconstrained opt. (will see later)
- For now, assume we have an unconstrained optimization problem

However, possible to have linear/ridge regression where solution has some constraints (e.g., non-neg, sparsity, or even both)

Linear and ridge regression that we saw were unconstrained (\mathbf{w}_{opt} was a real-valued vector)

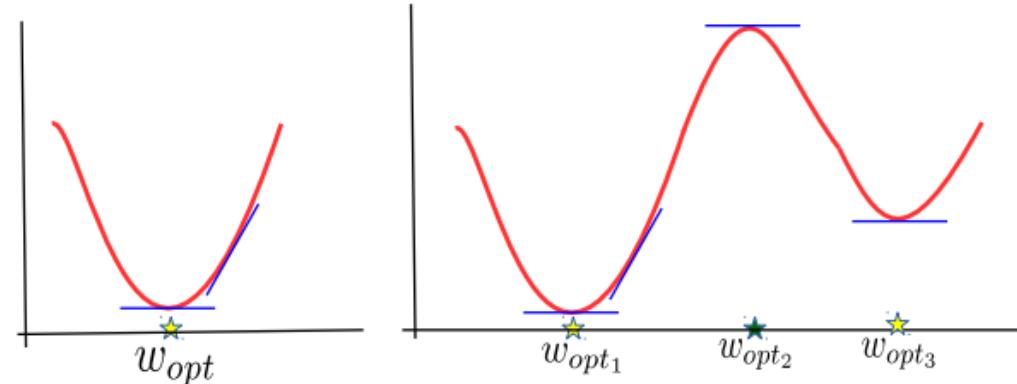


Methods for Solving Optimization Problems



Method 1: Using First-Order Optimality

- Very simple. Already used this approach for linear and ridge regression



Called “first order” since only gradient is used and gradient provides the first order info about the function being optimized



The approach works only for very simple problems where the objective is convex and there are no constraints on the values \mathbf{w} can take

- First order optimality: The gradient \mathbf{g} must be equal to zero at the optima

$$\mathbf{g} = \nabla_{\mathbf{w}}[L(\mathbf{w})] = \mathbf{0}$$

- Sometimes, setting $\mathbf{g} = \mathbf{0}$ and solving for \mathbf{w} gives a closed form solution
- If closed form solution is not available, the gradient vector \mathbf{g} can still be used in iterative optimization algos, like [gradient descent](#)



Method 2: Iterative Optimiz. via Gradient Descent



Can I used this approach to solve **maximization** problems?

Fact: Gradient gives the direction of **steepest change** in function's value

For max. problems we can use gradient **ascent**

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \eta_t \mathbf{g}^{(t)}$$

Will move in the direction of the gradient

Iterative since it requires several steps/iterations to find the optimal solution



For convex functions, GD will converge to the global minima

Good initialization needed for non-convex functions

Gradient Descent

- Initialize \mathbf{w} as $\mathbf{w}^{(0)}$
- For iteration $t = 0, 1, 2, \dots$ (or until convergence)
 - Calculate the gradient $\mathbf{g}^{(t)}$ using the current iterates $\mathbf{w}^{(t)}$
 - Set the learning rate η_t
 - Move in the opposite direction of gradient

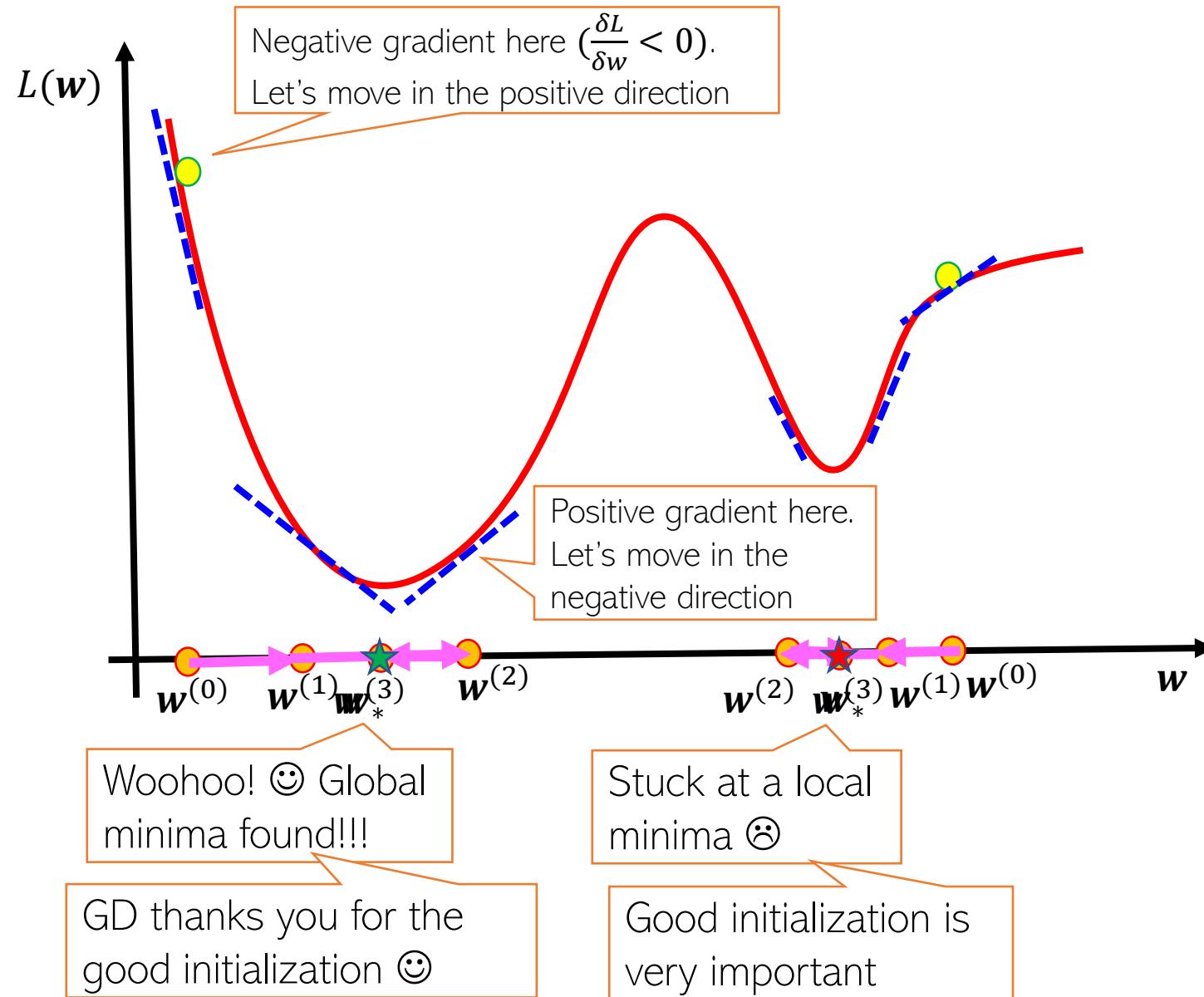
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \mathbf{g}^{(t)}$$

Will see the justification shortly

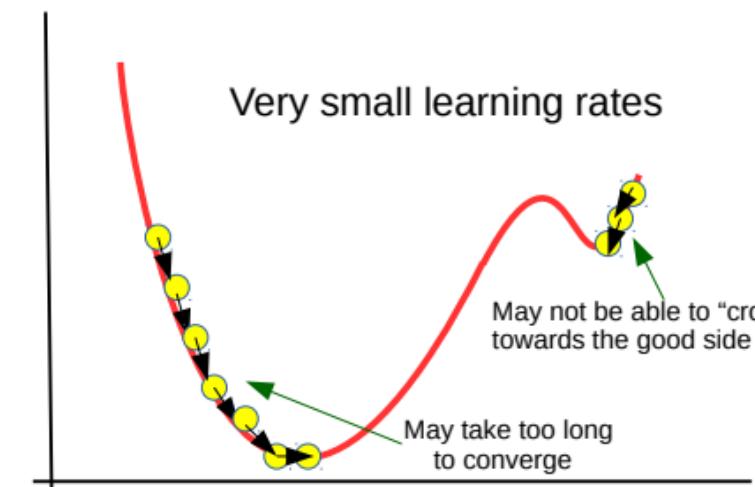
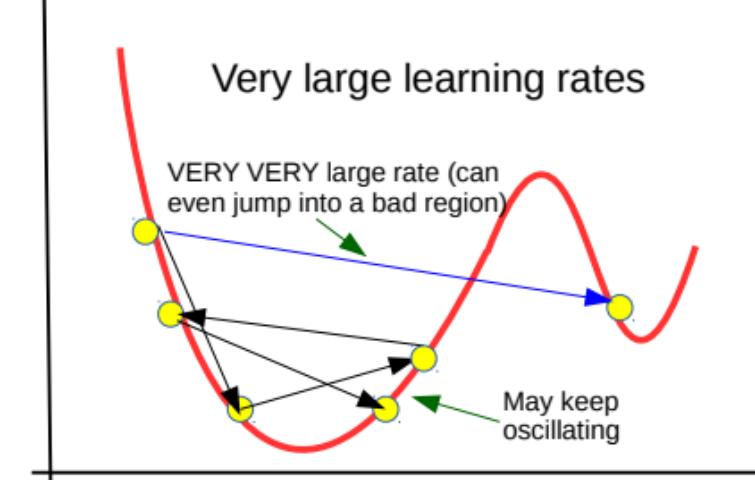
The learning rate very imp. Should be set carefully (fixed or chosen adaptively). Will discuss some strategies later

Sometimes may be tricky to assess convergence? Will see some methods later

Gradient Descent: An Illustration



Learning rate is very important



GD: An Example

- Let's apply GD for least squares linear regression

$$\mathbf{w}_{ridge} = \arg \min_{\mathbf{w}} L_{reg}(\mathbf{w}) = \arg \min_{\mathbf{w}} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2$$

- The gradient: $\mathbf{g} = - \sum_{n=1}^N 2(y_n - \mathbf{w}^\top \mathbf{x}_n) \mathbf{x}_n$

- Each GD update will be of the form

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \eta_t \sum_{n=1}^N 2 \left(y_n - \mathbf{w}^{(t)}^\top \mathbf{x}_n \right) \mathbf{x}_n$$

- Exercise: Assume $N = 1$, and show that GD update improves prediction on the training input (\mathbf{x}_n, y_n) , i.e, y_n is closer to $\mathbf{w}^{(t+1)^\top} \mathbf{x}_n$ than to $\mathbf{w}^{(t)^\top} \mathbf{x}_n$

- This is sort of a proof that GD updates are “corrective” in nature (and it actually is true not just for linear regression but can also be shown for various other ML models)

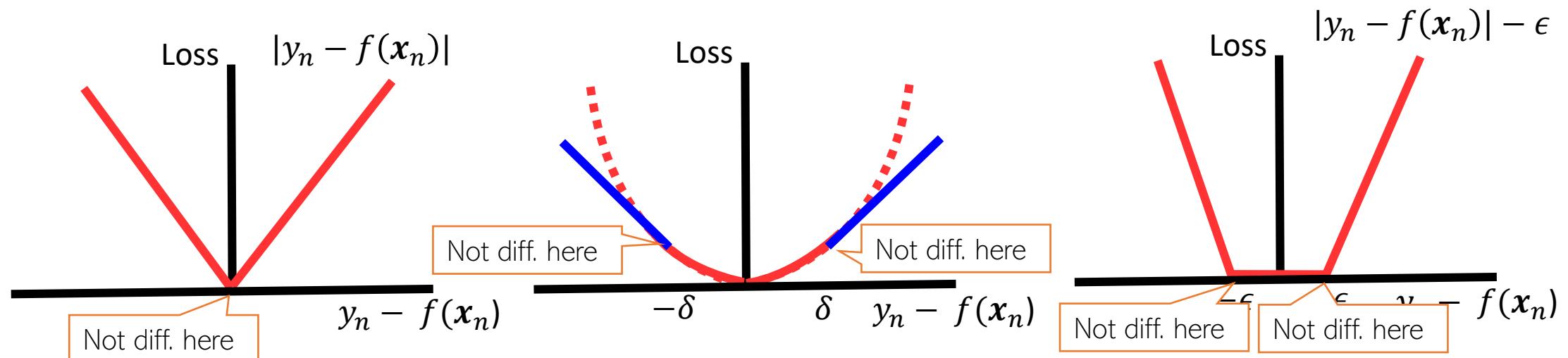
Prediction error of current model
 $\mathbf{w}^{(t)}$ on the n^{th} training example

Training examples
on which the
current model's
error is large
contribute more to
the update



Dealing with Non-differentiable Functions

- In many ML problems, the objective function will be non-differentiable
- Some examples that we have already seen: Linear regression with absolute loss, or Huber loss, or ϵ -insensitive loss; even ℓ_1 norm regularizer is non-diff

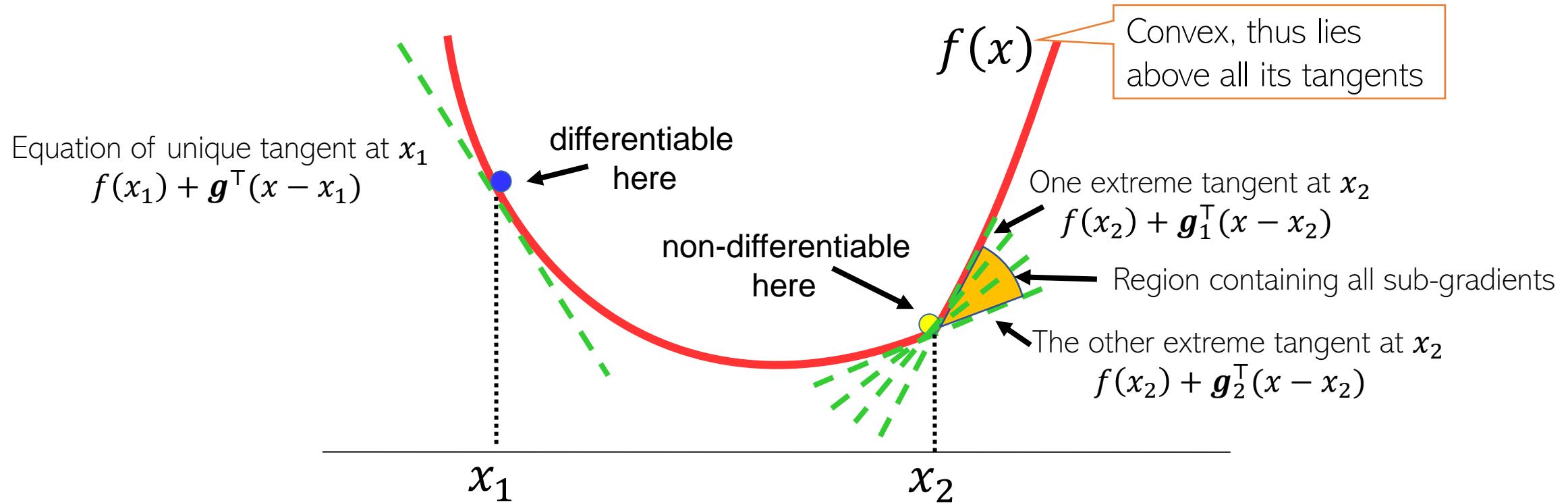


- Basically, any function in which there are points with kink is non-diff
 - At such points, the function is non-differentiable and thus **gradients not defined**
 - Reason: Can't define a unique tangent at such points



Sub-gradients

- For convex non-diff fn, can define sub-gradients at point(s) of non-differentiability



- For a convex, non-diff function $f(\mathbf{x})$, sub-gradient at \mathbf{x}_* is any vector \mathbf{g} s.t. $\forall \mathbf{x}$

$$f(\mathbf{x}) \geq f(\mathbf{x}_*) + \mathbf{g}^\top(\mathbf{x} - \mathbf{x}_*)$$

Sub-gradients, Sub-differential, and Some Rules

- Set of all sub-gradient at a non-diff point \mathbf{x}_* is called the **sub-differential**

$$\partial f(\mathbf{x}_*) \triangleq \{\mathbf{g} : f(\mathbf{x}) \geq f(\mathbf{x}_*) + \mathbf{g}^\top (\mathbf{x} - \mathbf{x}_*) \quad \forall \mathbf{x}\}$$

- Some basic rules of sub-diff calculus to keep in mind

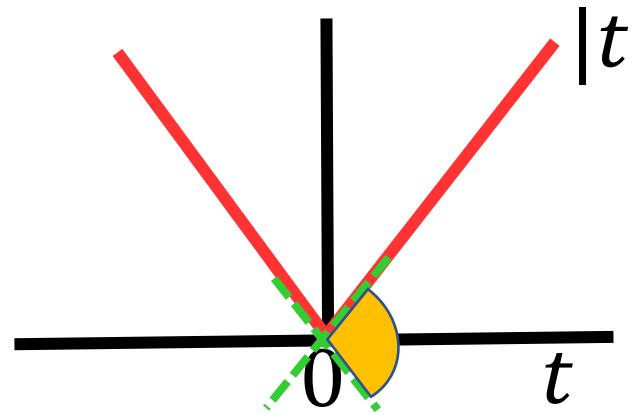
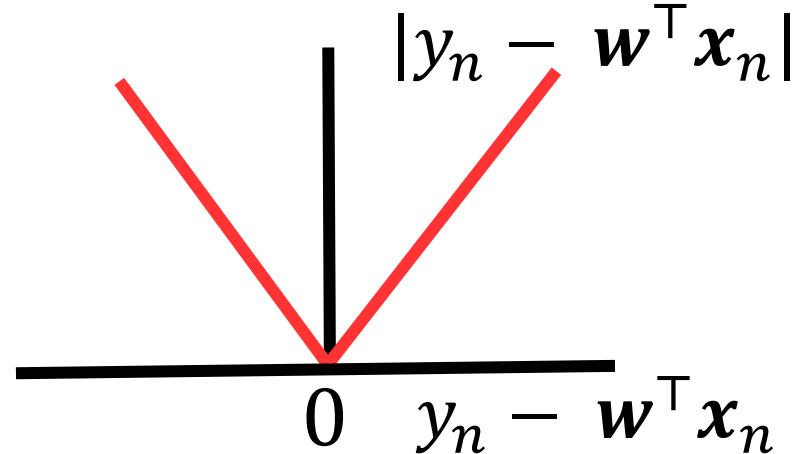
- Scaling rule: $\partial(c \cdot f(\mathbf{x})) = c \cdot \partial f(\mathbf{x}) = \{c \cdot \mathbf{v} : \mathbf{v} \in \partial f(\mathbf{x})\}$
- Sum rule: $\partial(f(\mathbf{x}) + g(\mathbf{x})) = \partial f(\mathbf{x}) + \partial g(\mathbf{x}) = \{\mathbf{u} + \mathbf{v} : \mathbf{u} \in \partial f(\mathbf{x}), \mathbf{v} \in \partial g(\mathbf{x})\}$
- Affine trans: $\partial f(\mathbf{a}^\top \mathbf{x} + b) = \partial f(t) \cdot \mathbf{a} = \{c \cdot \mathbf{a} : c \in \partial f(t)\}$, where $t = \mathbf{a}^\top \mathbf{x} + b$
- Max rule: If $h(\mathbf{x}) = \max\{f(\mathbf{x}), g(\mathbf{x})\}$ then we calculate $\partial h(\mathbf{x})$ at \mathbf{x}_* as
 - If $f(\mathbf{x}_*) > g(\mathbf{x}_*)$, $\partial h(\mathbf{x}_*) = \partial f(\mathbf{x}_*)$, If $g(\mathbf{x}_*) > f(\mathbf{x}_*)$, $\partial h(\mathbf{x}_*) = \partial g(\mathbf{x}_*)$
 - If $f(\mathbf{x}_*) = g(\mathbf{x}_*)$, $\partial h(\mathbf{x}_*) = \{\alpha \mathbf{a} + (1 - \alpha) \mathbf{b} : \mathbf{a} \in \partial f(\mathbf{x}_*), \mathbf{b} \in \partial g(\mathbf{x}_*), \alpha \in [0,1]\}$

The affine transform rule is a special case of the more general [chain rule](#)

- \mathbf{x}_* is a stationary point for a non-diff function $f(\mathbf{x})$ if the zero vector belongs to the sub-differential at \mathbf{x}_* , i.e., $\mathbf{0} \in \partial f(\mathbf{x}_*)$



Sub-Gradient For Absolute Loss Regression



$$\partial|t| = \begin{cases} 1 & \text{if } t > 0 \\ -1 & \text{if } t < 0 \\ [-1, +1] & \text{if } t = 0 \end{cases}$$

- The loss function for linear reg. with absolute loss: $L(\mathbf{w}) = |y_n - \mathbf{w}^\top \mathbf{x}_n|$
- Non-differentiable at $y_n - \mathbf{w}^\top \mathbf{x}_n = 0$
- Can use the affine transform rule of sub-diff calculus
- Assume $t = y_n - \mathbf{w}^\top \mathbf{x}_n$. Then $\partial L(\mathbf{w}) = -\mathbf{x}_n \partial|t|$
 - $\partial L(\mathbf{w}) = -\mathbf{x}_n \times 1 = -\mathbf{x}_n$ if $t > 0$
 - $\partial L(\mathbf{w}) = -\mathbf{x}_n \times -1 = \mathbf{x}_n$ if $t < 0$
 - $\partial L(\mathbf{w}) = -\mathbf{x}_n \times c = -c\mathbf{x}_n$ where $c \in [-1, +1]$ if $t = 0$



Sub-Gradient Descent

- Suppose we have a non-differentiable function $L(\mathbf{w})$
- Sub-gradient descent is almost identical to GD except we use subgradients

Sub-Gradient Descent

- Initialize \mathbf{w} as $\mathbf{w}^{(0)}$
- For iteration $t = 0, 1, 2, \dots$ (or until convergence)
 - Calculate the sub-gradient $\mathbf{g}^{(t)} \in \partial L(\mathbf{w}^{(t)})$
 - Set the learning rate η_t
 - Move in the opposite direction of subgradient

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \mathbf{g}^{(t)}$$



Coming up next

- Making GD faster: Stochastic gradient descent
- Constrained optimization
- Co-ordinate descent
- Alternating optimization
- Practical issue in optimization for ML



Optimization for ML (3)

CS771: Introduction to Machine Learning

Piyush Rai

Stochastic Gradient Descent (SGD)

Writing as an average instead of sum.
Won't affect minimization of $L(\mathbf{w})$

- Consider a loss function of the form $L(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N \ell_n(\mathbf{w})$

- The (sub)gradient in this case can be written as

$$\mathbf{g} = \nabla_{\mathbf{w}} L(\mathbf{w}) = \nabla_{\mathbf{w}} \left[\frac{1}{N} \sum_{n=1}^N \ell_n(\mathbf{w}) \right] = \frac{1}{N} \sum_{n=1}^N \mathbf{g}_n$$

(Sub)gradient of the loss on n^{th} training example

- Stochastic Gradient Descent (SGD) approximates \mathbf{g} using a single training example
- At iter. t , pick an index $i \in \{1, 2, \dots, N\}$ uniformly randomly and approximate \mathbf{g} as

$$\mathbf{g} \approx \mathbf{g}_i = \nabla_{\mathbf{w}} \ell_i(\mathbf{w})$$

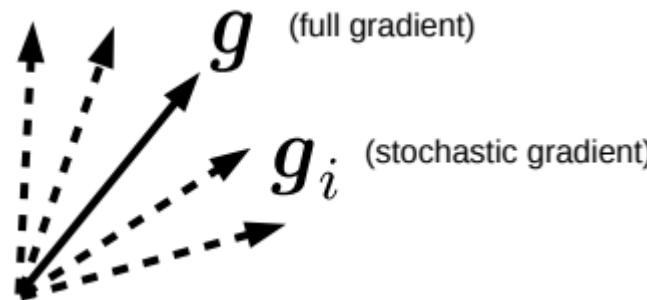
Can show that \mathbf{g}_i is an unbiased estimate of \mathbf{g} , i.e., $\mathbb{E}[\mathbf{g}_i] = \mathbf{g}$

- May take more iterations than GD to converge but each iteration is much faster ☺
 - SGD per iter cost is $O(D)$ whereas GD per iter cost is $O(ND)$



Minibatch SGD

- Gradient approximation using a single training example may be noisy



The approximation may have a **high variance** – may slow down convergence, updates may be unstable, and may even give sub-optimal solutions (e.g., local minima where GD might have given global minima)

- We can use $B > 1$ unif. rand. chosen train. ex. with indices $\{i_1, i_2, \dots, i_B\} \in \{1, 2, \dots, N\}$
- Using this “minibatch” of examples, we can compute a minibatch gradient

$$\mathbf{g} \approx \frac{1}{B} \sum_{b=1}^B \mathbf{g}_{i_b}$$

- Averaging helps in reducing the variance in the stochastic gradient
- Time complexity is $O(BD)$ per iteration in this case



Constrained Optimization



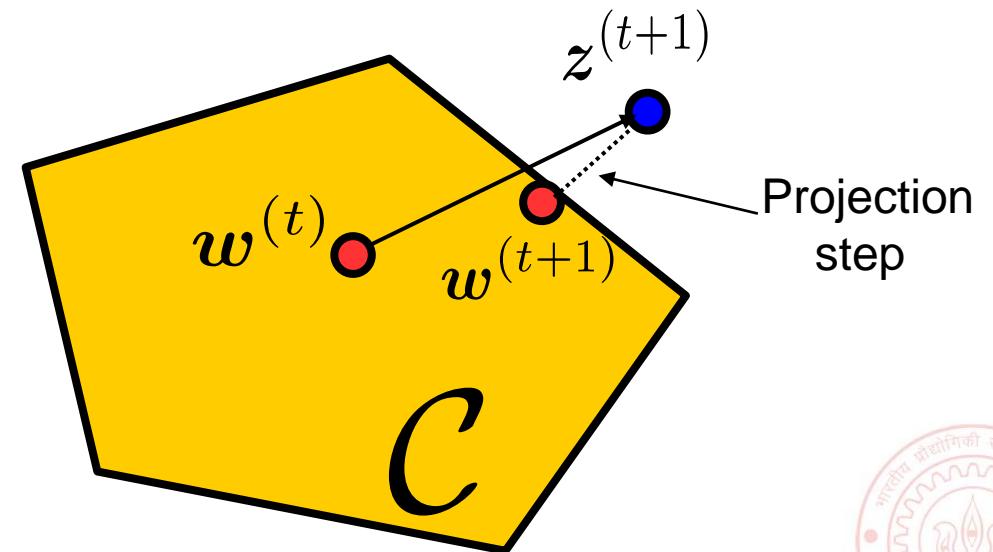
Projected Gradient Descent

- Consider an optimization problem of the form

$$\mathbf{w}_{opt} = \arg \min_{\mathbf{w} \in \mathcal{C}} L(\mathbf{w})$$

- Projected GD is very similar to GD with an extra **projection step**
- Each iteration t will be of the form

- Perform update: $\mathbf{z}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \mathbf{g}^{(t)}$
- Check if $\mathbf{z}^{(t+1)}$ satisfies constraints
 - If $\mathbf{z}^{(t+1)} \in \mathcal{C}$, set $\mathbf{w}^{(t+1)} = \mathbf{z}^{(t+1)}$
 - If $\mathbf{z}^{(t+1)} \notin \mathcal{C}$, project as $\mathbf{w}^{(t+1)} = \Pi_{\mathcal{C}}[\mathbf{z}^{(t+1)}]$



Projected GD: How to Project?

- Here projecting a point means finding the “closest” point from the constraint set

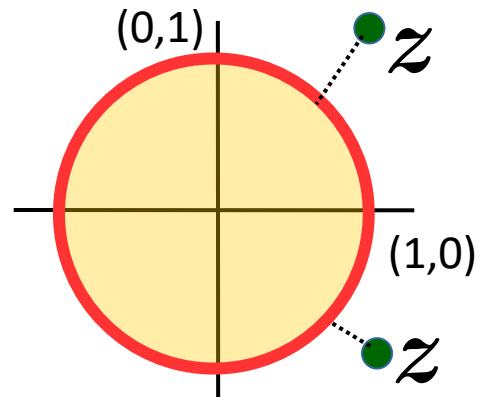
$$\Pi_{\mathcal{C}}[\mathbf{z}] = \arg \min_{\mathbf{w} \in \mathcal{C}} \|\mathbf{z} - \mathbf{w}\|^2$$

- For some sets \mathcal{C} , the projection step is easy

Projected GD commonly used only when the projection step is simple and efficient to compute



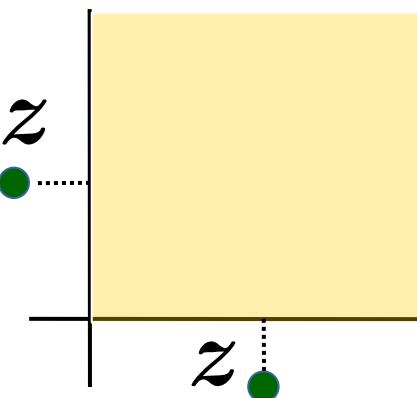
\mathcal{C} : Unit radius ℓ_2 ball



Projection = Normalize to unit Euclidean length vector

$$\hat{\mathbf{x}} = \begin{cases} \mathbf{x} & \text{if } \|\mathbf{x}\|_2 \leq 1 \\ \frac{\mathbf{x}}{\|\mathbf{x}\|_2} & \text{if } \|\mathbf{x}\|_2 > 1 \end{cases}$$

\mathcal{C} : Set of non-negative reals



Projection = Set each negative entry in \mathbf{z} to be zero

$$\hat{\mathbf{x}}_i = \begin{cases} \mathbf{x}_i & \text{if } \mathbf{x}_i \geq 0 \\ 0 & \text{if } \mathbf{x}_i < 0 \end{cases}$$



Proximal Gradient Descent

- Consider minimizing a regularized loss function of the form

$$\arg \min_w L(\mathbf{w}) + R(\mathbf{w})$$

Note: The reg. hyperparam. λ
assumed part of $R(\mathbf{w})$ itself

- Proximal GD popular when regularizer $R(\mathbf{w})$ is non-differentiable
- Basic idea: Do GD on $L(\mathbf{w})$ and use a prox. operator to regularize via $R(\mathbf{w})$
- For a func. R , its prox. operator is $\text{prox}_R(\mathbf{z}) = \arg \min_w \left[R(\mathbf{w}) + \frac{1}{2} \|\mathbf{z} - \mathbf{w}\|_2^2 \right]$

Proximal GD

- Assume reg. loss function of the form $L(\mathbf{w}) + R(\mathbf{w})$
- Initialize \mathbf{w} as $\mathbf{w}^{(0)}$
- For iteration $t = 0, 1, 2, \dots$ (or until convergence)
 - Calculate the (sub)gradient of train. Loss (w/o reg.)
 $\mathbf{g}^{(t)} \in \partial L(\mathbf{w}^{(t)})$
 - Set learning rate η_t
 - Step 1: $\mathbf{z}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \mathbf{g}^{(t)}$
 - Step 2: $\mathbf{w}^{(t+1)} = \text{prox}_R(\mathbf{z}^{(t+1)})$

Special Cases

For $R(\mathbf{w}) = 0.5 \times \|\mathbf{w}\|_2^2$
 $\text{prox}_R(\mathbf{z}) = \mathbf{z}/2$ i.e. scaling

That is, regularize
by reducing the
value of each
component of the
vector \mathbf{z} by half

If $R(\mathbf{w})$ defines a set based constraint

$$R(\mathbf{w}) := \mathbf{w} \in \mathcal{C}$$

$$\text{prox}_R(\mathbf{z}) = \arg \min_{\mathbf{w} \in \mathcal{C}} \|\mathbf{z} - \mathbf{w}\|^2$$

Prox. GD becomes equivalent
to projected GD



Constrained Opt. via Lagrangian

- Consider the following constrained minimization problem (using f instead of L)

$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w}} f(\mathbf{w}), \quad \text{s.t.} \quad g(\mathbf{w}) \leq 0$$

- Note: If constraints of the form $g(\mathbf{w}) \geq 0$, use $-g(\mathbf{w}) \leq 0$
- Can handle multiple inequality and equality constraints too (will see later)
- Can transform the above into the following equivalent unconstrained problem

$$\hat{\mathbf{w}} = \arg \min \mathbf{f}(\mathbf{w}) + c(\mathbf{w})$$

$$c(\mathbf{w}) = \max_{\alpha \geq 0} \alpha g(\mathbf{w}) = \begin{cases} \infty, & \text{if } g(\mathbf{w}) > 0 \quad (\text{constraint violated}) \\ 0 & \text{if } g(\mathbf{w}) \leq 0 \quad (\text{constraint satisfied}) \end{cases}$$

- Our problem can now be written as

$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w}} \left\{ f(\mathbf{w}) + \arg \max_{\alpha \geq 0} \alpha g(\mathbf{w}) \right\}$$



Constrained Opt. via Lagrangian

- Therefore, we can write our original problem as

$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w}} \left\{ f(\mathbf{w}) + \arg \max_{\alpha \geq 0} \alpha g(\mathbf{w}) \right\} = \arg \min_{\mathbf{w}} \left\{ \arg \max_{\alpha \geq 0} \{ f(\mathbf{w}) + \alpha g(\mathbf{w}) \} \right\}$$

The Lagrangian: $\mathcal{L}(\mathbf{w}, \boldsymbol{\alpha})$

- The Lagrangian is now optimized w.r.t. \mathbf{w} and $\boldsymbol{\alpha}$ (Lagrange multiplier)
- We can define Primal and Dual problem as

$$\hat{\mathbf{w}}_P = \arg \min_{\mathbf{w}} \left\{ \arg \max_{\alpha \geq 0} \{ f(\mathbf{w}) + \alpha g(\mathbf{w}) \} \right\} \quad (\text{primal problem})$$

$$\hat{\mathbf{w}}_D = \arg \max_{\alpha \geq 0} \left\{ \arg \min_{\mathbf{w}} \{ f(\mathbf{w}) + \alpha g(\mathbf{w}) \} \right\} \quad (\text{dual problem})$$

Both equal if $f(\mathbf{w})$ and the set $g(\mathbf{w}) \leq 0$ are convex

$$\alpha_D g(\hat{\mathbf{w}}_D) = 0$$

complimentary slackness/Karush-Kuhn-Tucker (KKT) condition



Constrained Opt. with Multiple Constraints

- We can also have multiple inequality and equality constraints

$$\begin{aligned}\hat{\mathbf{w}} &= \arg \min_{\mathbf{w}} f(\mathbf{w}) \\ \text{s.t.} \quad g_i(\mathbf{w}) &\leq 0, \quad i = 1, \dots, K \\ h_j(\mathbf{w}) &= 0, \quad j = 1, \dots, L\end{aligned}$$

- Introduce Lagrange multipliers $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_K]$ and $\boldsymbol{\beta} = [\beta_1, \beta_2, \dots, \beta_L]$
- The Lagrangian based primal and dual problems will be

$$\hat{\mathbf{w}}_P = \arg \min_{\mathbf{w}} \left\{ \arg \max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta}} \left\{ f(\mathbf{w}) + \sum_{i=1}^K \alpha_i g_i(\mathbf{w}) + \sum_{j=1}^L \beta_j h_j(\mathbf{w}) \right\} \right\}$$

$$\hat{\mathbf{w}}_D = \arg \max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta}} \left\{ \arg \min_{\mathbf{w}} \left\{ f(\mathbf{w}) + \sum_{i=1}^K \alpha_i g_i(\mathbf{w}) + \sum_{j=1}^L \beta_j h_j(\mathbf{w}) \right\} \right\}$$



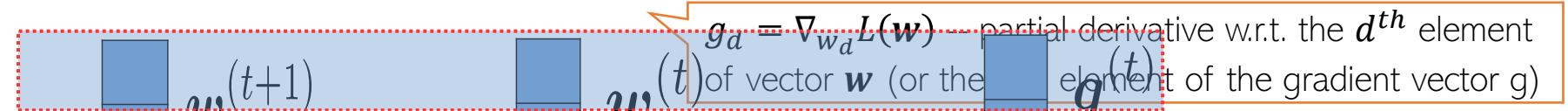
Some other useful optimization methods



Co-ordinate Descent (CD)

- Standard gradient descent update for : $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \mathbf{g}^{(t)}$
- CD: In each iter, update only one entry (co-ordinate) of \mathbf{w} . Keep all others fixed

$$w_d^{(t+1)} = w_d^{(t)} - \eta_t g_d^{(t)} \quad d \in \{1, 2, \dots, D\}$$



$w_d^{(t+1)} = w_d^{(t)} - \eta_t g_d^{(t)}$

$g_d = \nabla_{w_d} L(\mathbf{w})$ partial derivative w.r.t. the d^{th} element
of vector \mathbf{w} (or the d^{th} element of the gradient vector \mathbf{g})

- Cost of each update is now independent of D
- In each iter, can choose co-ordinate to update unif. randomly or in cyclic order
- Instead of updating a single co-ord, can also update "blocks" of co-ordinates
 - Called Block co-ordinate descent (BCD)
- To avoid $O(D)$ cost of gradient computation, can cache previous computations
 - Recall that grad. computations may have terms like $\mathbf{w}^T \mathbf{x}$ – if just one co-ordinate of \mathbf{w} changes, we should avoid computing the new $\mathbf{w}^T \mathbf{x}$ ($= \sum_d w_d x_d$) from scratch

Alternating Optimization (ALT-OPT)

- Consider opt. problems with several variables, say two variables \mathbf{w}_1 and \mathbf{w}_2

$$\{\hat{\mathbf{w}}_1, \hat{\mathbf{w}}_2\} = \arg \min_{\mathbf{w}_1, \mathbf{w}_2} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2)$$

- Often, this “joint” optimization is hard/impossible to solve
- We can take an alternating optimization approach to solve such problems

ALT-OPT

- ① Initialize one of the variables, e.g., $\mathbf{w}_2 = \mathbf{w}_2^{(0)}$, $t = 0$
- ② Solve $\mathbf{w}_1^{(t+1)} = \arg \min_{\mathbf{w}_1} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2^{(t)})$ // \mathbf{w}_2 “fixed” at its most recent value $\mathbf{w}_2^{(t)}$
- ③ Solve $\mathbf{w}_2^{(t+1)} = \arg \min_{\mathbf{w}_2} \mathcal{L}(\mathbf{w}_1^{(t+1)}, \mathbf{w}_2)$ // \mathbf{w}_1 “fixed” at its most recent value $\mathbf{w}_1^{(t+1)}$
- ④ $t = t + 1$. Go to step 2 if not converged yet.

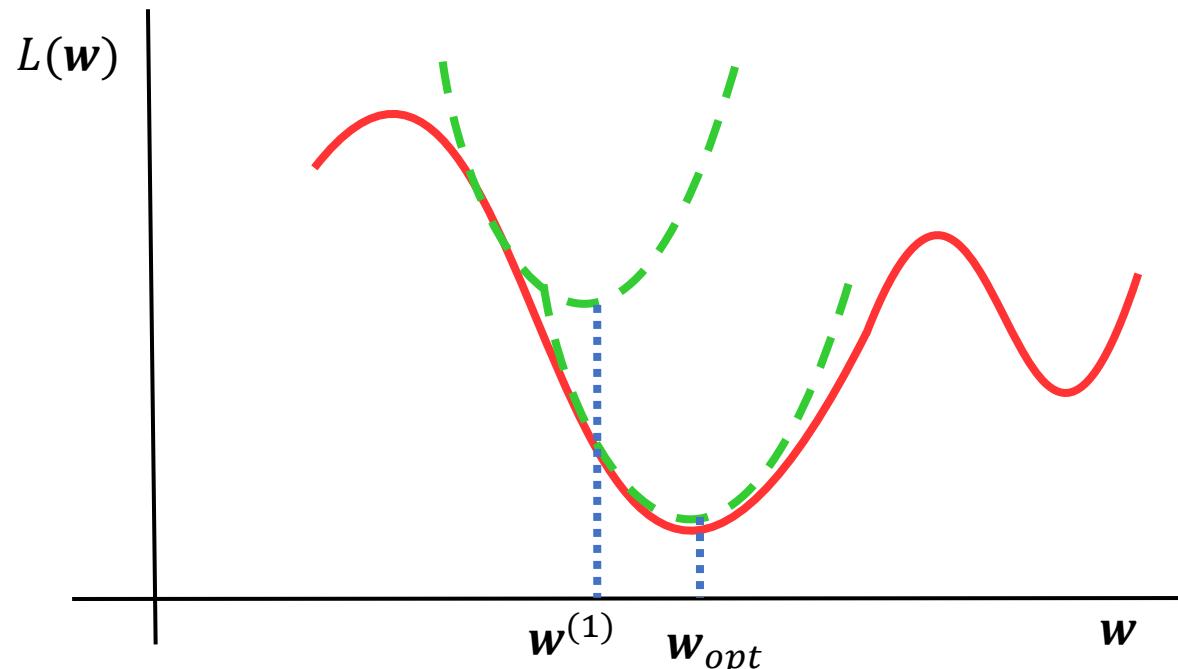
- Usually converges to a local optima. But very very useful. Will see examples later
 - Also related to the Expectation-Maximization (EM) algorithm which we will see later



Newton's Method

- Unlike GD and its variants, Newton's method uses **second-order** information (second derivative, a.k.a. the Hessian)
- At each point $\mathbf{w}^{(t)}$, minimize the quadratic (second-order) approx. of $L(\mathbf{w})$

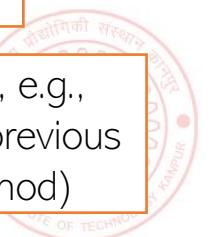
$$\mathbf{w}^{(t+1)} = \arg \min_{\mathbf{w}} [L(\mathbf{w}^{(t)}) + \nabla L(\mathbf{w}^{(t)})^\top (\mathbf{w} - \mathbf{w}^{(t)}) + \frac{1}{2} (\mathbf{w} - \mathbf{w}^{(t)})^\top \nabla^2 L(\mathbf{w}^{(t)}) (\mathbf{w} - \mathbf{w}^{(t)})]$$



Show that $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - (\nabla^2 L(\mathbf{w}^{(t)}))^{-1} \nabla L(\mathbf{w}^{(t)})$
 $= \mathbf{w}^{(t)} - (\mathbf{H}^{(t)})^{-1} \mathbf{g}^{(t)}$

Converges much faster than GD (very fast for convex functions). Also no “learning rate”. But per iteration cost is slower due to Hessian computation and inversion

Faster versions of Newton's method also exist, e.g., those based on approximating Hessian using previous gradients (see L-BFGS which is a popular method)



Coming up next

- Some practical issue in optimization for ML
- Wrapping up the discussion of optimization techniques
- Probabilistic models for ML



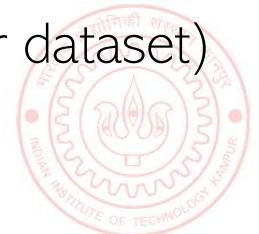
Optimization for ML (4)

CS771: Introduction to Machine Learning

Piyush Rai

Some Practical Aspects: Initialization

- Iterative opt. algos like GD, SGD, etc need to be initialized to “good” values
 - Bad initialization can result on bad local optima
- Mainly a concern for non-convex loss functions, not so much for convex loss functions
 - If the goal is to learn the same model but for a different training set
- Transfer Learning: Initialize using params of a model trained on a related dataset
- Initialize using solution of a simpler but related model
 - E.g., for multitask regression (say T coupled regression problems), initialize using the solutions of the T independently trained regression problems
- For deep learning models, initialization is very important
 - Transfer learning approach is often used (initialize using “pre-trained” model from another dataset)
 - Bad initialization can make the model be stuck at saddle points. Need more care.
 - Random restarts: Running with several random initializations can often help



Some Practical Aspects: Assessing Convergence

- Various ways to assess convergence, e.g. consider converged if
 - The objective's value (on train set) ceases to change much across iterations

$$L(\mathbf{w}^{(t+1)}) - L(\mathbf{w}^{(t)}) < \epsilon \quad (\text{for some small pre-defined } \epsilon)$$

- The parameter values cease to change much across iterations

$$\|\mathbf{w}^{(t+1)} - \mathbf{w}^{(t)}\| < \tau \quad (\text{for some small pre-defined } \tau)$$

- Above condition is also equivalent to saying that the gradients are close to zero

$$\|\mathbf{g}^{(t)}\| \rightarrow 0$$

Caution: May not yet be at the optima. Use at your own risk!

- The objective's value has become small enough that we are happy with 😊
- Use a validation set to assess if the model's performance is acceptable ([early stopping](#))



Some Practical Aspects: Learning Rate (Step Size)

- Some guidelines to select good learning rate (a.k.a. step size) η_t
 - C is a hyperparameter
- For convex functions, setting η_t something like C/t or C/\sqrt{t} often works well
 - These step-sizes are actually theoretically optimal in some settings
 - In general, we want the learning rates to satisfy the following conditions
 - $\eta_t \rightarrow 0$ as t becomes very very large
 - $\sum \eta_t = \infty$ (needed to ensure that we can potentially reach anywhere in the parameter space)
 - Sometimes carefully chosen constant learning rates (usually small, or initially large and later small) also work well in practice
- Can also search for the “best” step-size by solving an opt. problem in each step

$$\eta_t = \arg \min_{\eta \geq 0} f(\mathbf{w}^{(t)} - \eta \cdot \mathbf{g}^{(t)})$$
 - Also called “line search”
 - A one-dim optimization problem (note that $\mathbf{w}^{(t)}$ and $\mathbf{g}^{(t)}$ are fixed)
- An faster alternative to line search is the [Armijo-Goldstein rule](#)
 - Starting with current (or some large) learning rate (from prev. iter), and try a few values in decreasing order until the objective’s value has a sufficient reduction



Some Practical Aspects: Adaptive Gradient Methods

- Can also use different learning rate in different dimensions

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \mathbf{e}^{(t)} \odot \mathbf{g}^{(t)}$$

Vector of learning rates
along each dimension

Element-wise product of
two vectors

$$\mathbf{e}_d^{(t)} = \frac{1}{\sqrt{\epsilon + \sum_{\tau=1}^t (\mathbf{g}_d^{(\tau)})^2}}$$

If some dimension had big updates recently (marked by large gradient values), slow down along those directions by using smaller learning rates - AdaGrad (Duchi et al, 2011)

- Can use a momentum term to stabilize gradients by reusing info from past grads

- Move faster along directions that were previously good
- Slow down along directions where gradient has changed abruptly

β usually set
as 0.9

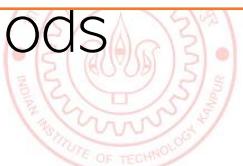
The “momentum” term.
Set to 0 at initialization

$$\begin{aligned} \mathbf{m}^{(t)} &= \beta \mathbf{m}^{(t-1)} + \eta_t \mathbf{g}^{(t)} \\ \mathbf{w}^{(t+1)} &\leftarrow \mathbf{w}^{(t)} - \mathbf{m}^{(t)} \end{aligned}$$

In an even faster version of this, $\mathbf{g}^{(t)}$ is replaced by the gradient computed at the next step if previous direction were used, i.e., $\nabla L(\mathbf{w}^{(t)} - \beta \mathbf{m}^{(t-1)})$. Called Nesterov's Accelerated Gradient (NAG) method

- Also exists several more advanced methods that combine the above methods

- RMS-Prop: AdaGrad + Momentum, Adam: NAG + RMS-Prop
- These methods are part of packages such as PyTorch, Tensorflow, etc



Optimization for ML: Some Final Comments

- Gradient methods are simple to understand and implement
- More sophisticated optimization methods also often use gradient methods
- Backpropagation algo used in deep neural nets is GD + chain rule of differentiation
- Use subgradient methods if function not differentiable
- Constrained optimization can use Lagrangian or projected/proximal GD
- Second order methods such as Newton's method faster but computationally expensive
- But computing all this gradient related stuff by hand looks scary to me. Any help?
 - Don't worry. Automatic Differentiation (AD) methods available now (will see them later)
 - AD only requires specifying the loss function (especially useful for deep neural nets)
 - Many packages such as Tensorflow, PyTorch, etc. provide AD support
 - But having a good understanding of optimization is still helpful



Probabilistic Machine Learning (1): Some Basics of Probability

CS771: Introduction to Machine Learning

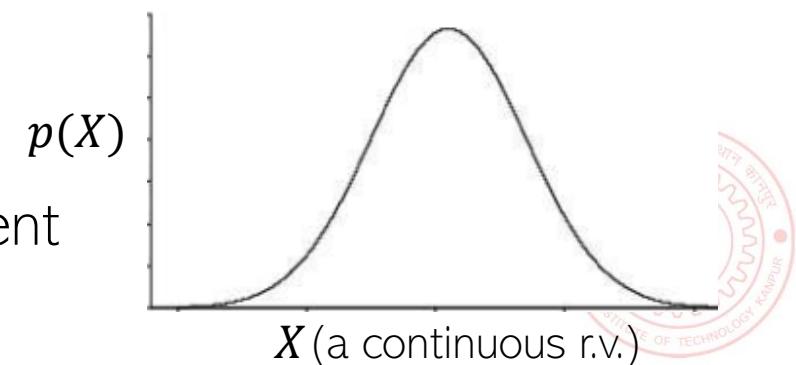
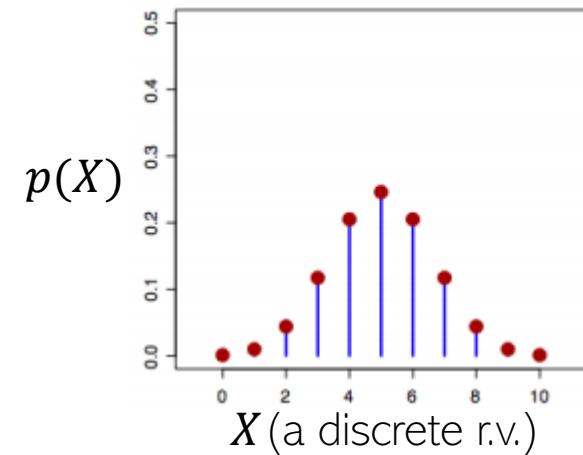
Piyush Rai

Some Probability Basics



Random Variables

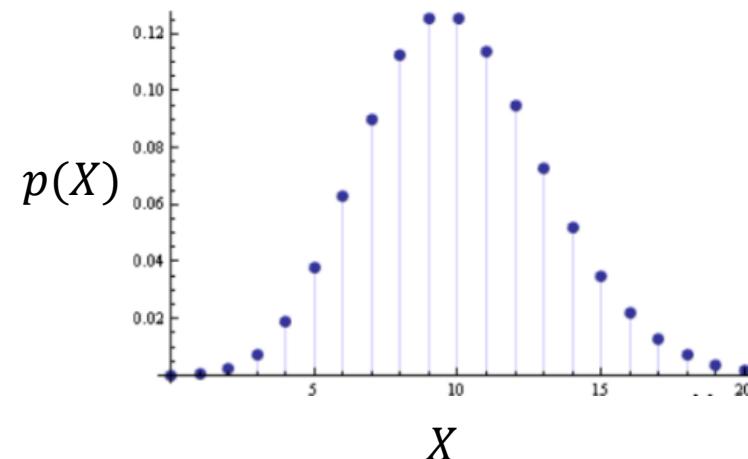
- Informally, a random variable (r.v.) X denotes possible outcomes of an event
- Can be discrete (i.e., finite many possible outcomes) or continuous
- Some examples of discrete r.v.
 - $X \in \{0, 1\}$ denoting outcomes of a coin-toss
 - $X \in \{1, 2, \dots, 6\}$ denoting outcome of a dice roll
- Some examples of continuous r.v.
 - $X \in (0, 1)$ denoting the bias of a coin
 - $X \in \mathbb{R}$ denoting heights of students in CS771
 - $X \in \mathbb{R}$ denoting time to get to your hall from the department



Discrete Random Variables

- For a discrete r.v. X , $p(x)$ denotes $p(X = x)$ - probability that $X = x$
- $p(X)$ is called the **probability mass function** (PMF) of r.v. X
 - $p(x)$ or $p(X = x)$ is the value of the PMF at x

$$\begin{aligned} p(x) &\geq 0 \\ p(x) &\leq 1 \\ \sum_x p(x) &= 1 \end{aligned}$$



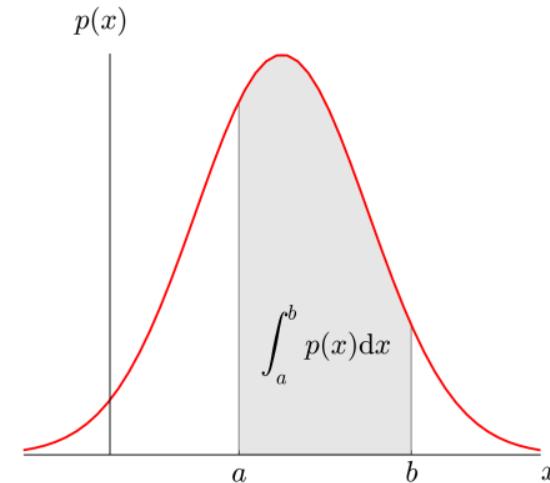
Continuous Random Variables

- For a continuous r.v. X , a *probability* $p(X = x)$ or $p(x)$ is meaningless
- For cont. r.v., we talk in terms of prob. within an interval $X \in (x, x + \delta x)$
 - $p(x)\delta x$ is the prob. that $X \in (x, x + \delta x)$ as $\delta x \rightarrow 0$
 - $p(x)$ is the probability density at $X = x$



Yes, probability density at a point x can very well be larger than 1. The integral however must be equal to 1

$$\begin{aligned} p(x) &\geq 0 \\ \cancel{p(x)} &\leq 1 \\ \int p(x)dx &= 1 \end{aligned}$$



A word about notation

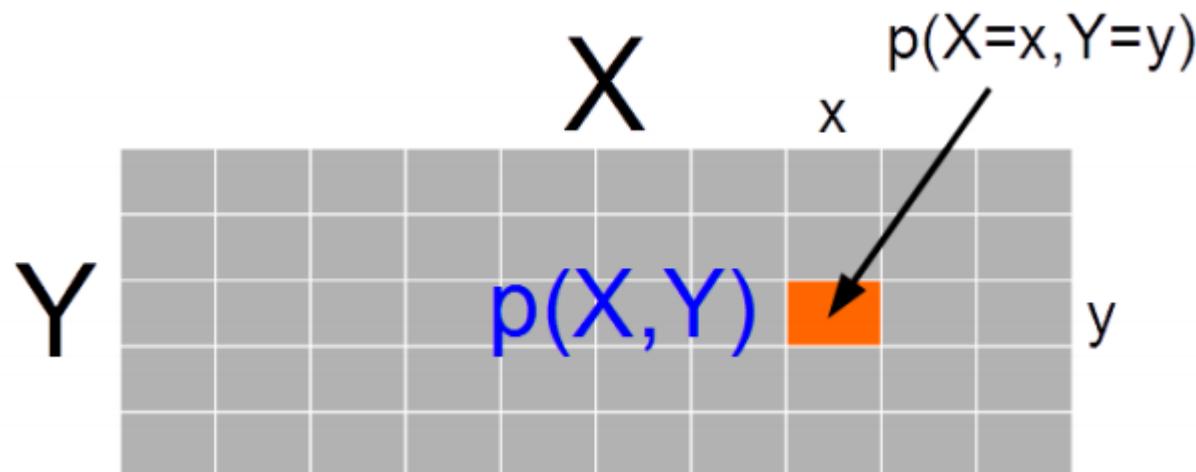
- $p(\cdot)$ can mean different things depending on the context
- $p(X)$ denotes the distribution (PMF/PDF) of an r.v. X
- $p(X = x)$ or $p_X(x)$ or simply $p(x)$ denotes the prob. or prob. density at value x
 - Actual meaning should be clear from the context (but be careful)
- Exercise same care when $p(\cdot)$ is a specific distribution (Bernoulli, Gaussian, etc.)
- The following means generating a random sample from the distribution $p(X)$

$$x \sim p(X)$$

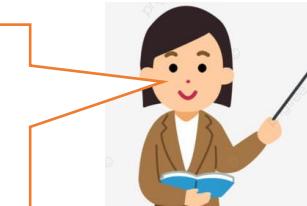


Joint Probability Distribution

- Joint prob. dist. $p(X, Y)$ models probability of co-occurrence of two r.v. X, Y
- For discrete r.v., the joint PMF $p(X, Y)$ is like a table (that sums to 1)



For 3 r.v.'s, we will likewise have a “cube” for the PMF. For more than 3 r.v.'s too, similar analogy holds



$$\sum_x \sum_y p(X = x, Y = y) = 1$$

- For two continuous r.v.'s X and Y , we have joint PDF $p(X, Y)$

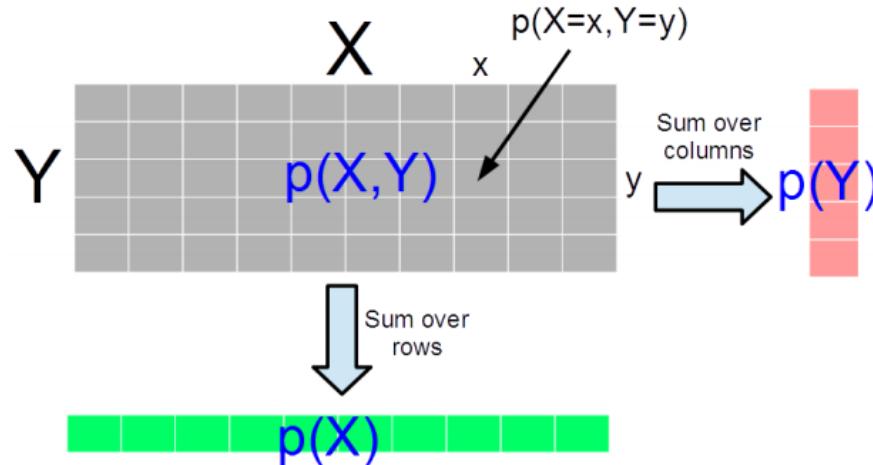
$$\int_x \int_y p(X = x, Y = y) dx dy = 1$$

For more than two r.v.'s, we will likewise have a multi-dim integral for this property



Marginal Probability Distribution

- Consider two r.v.'s X and Y (discrete/continuous – both need not of same type)
- Marg. Prob. is PMF/PDF of one r.v. accounting for all possibilities of the other r.v.
- For discrete r.v.'s, $p(X) = \sum_y p(X, Y = y)$ and $p(Y) = \sum_x p(X = x, Y)$
- For discrete r.v. it is the sum of the PMF table along the rows/columns



The definition also applied for two sets of r.v.'s and marginal of one set of r.v.'s is obtained by summing over all possibilities of the second set of r.v.'s



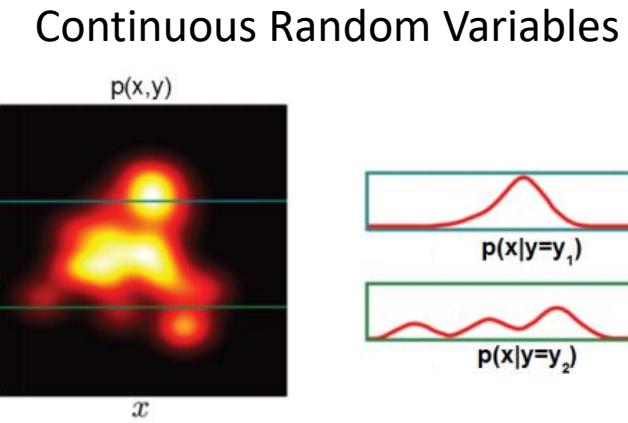
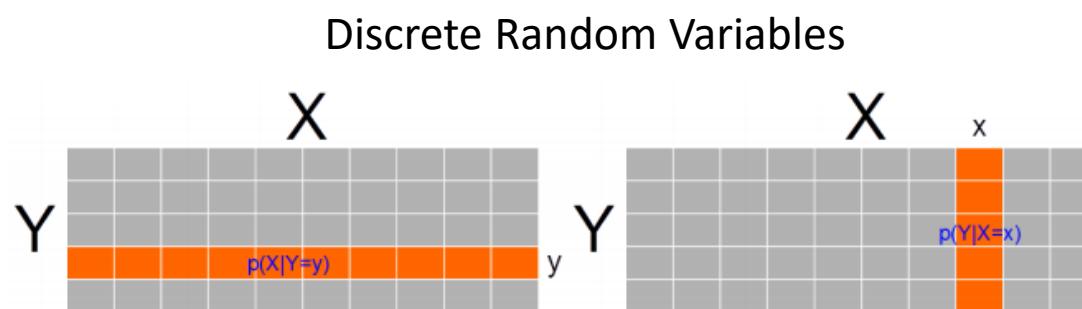
For discrete r.v.'s, marginalization is called summing over, for continuous r.v.'s, it is called "integrating out"

- For continuous r.v.'s, $p(X) = \int_y p(X, Y = y) dy$, $p(Y) = \int_x p(X = x, Y) dx$



Conditional Probability Distribution

- Consider two r.v.'s X and Y (discrete/continuous – both need not of same type)
- Conditional PMF/PDF $p(X|Y)$ is the prob. dist. of one r.v. X , fixing other r.v. Y
- $p(X|Y = y)$ or $p(Y | X = x)$ like taking a slice of the joint dist. $p(X, Y)$



- Note: A conditional PMF/PDF may also be conditioned on something that is not the value of an r.v. but some fixed quantity in general

We will see cond. dist. of output y given weights w (r.v.) and features \mathbf{X} written as $p(y|w, \mathbf{X})$

Some Basic Rules

- **Sum Rule:** Gives the marginal probability distribution from joint probability distribution

$$\text{For discrete r.v.: } p(X) = \sum_Y p(X, Y)$$

$$\text{For continuous r.v.: } p(X) = \int_Y p(X, Y) dY$$

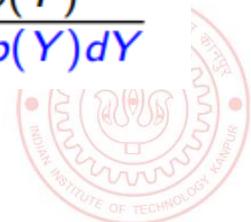
- **Product Rule:** $p(X, Y) = p(Y|X)p(X) = p(X|Y)p(Y)$
- **Bayes' rule:** Gives conditional probability distribution (can derive it from product rule)

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}$$

$$\text{For discrete r.v.: } p(Y|X) = \frac{p(X|Y)p(Y)}{\sum_Y p(X|Y)p(Y)}$$

$$\text{For continuous r.v.: } p(Y|X) = \frac{p(X|Y)p(Y)}{\int_Y p(X|Y)p(Y) dY}$$

- **Chain Rule:** $p(X_1, X_2, \dots, X_N) = p(X_1)p(X_2|X_1)\dots p(X_N|X_1, \dots, X_{N-1})$



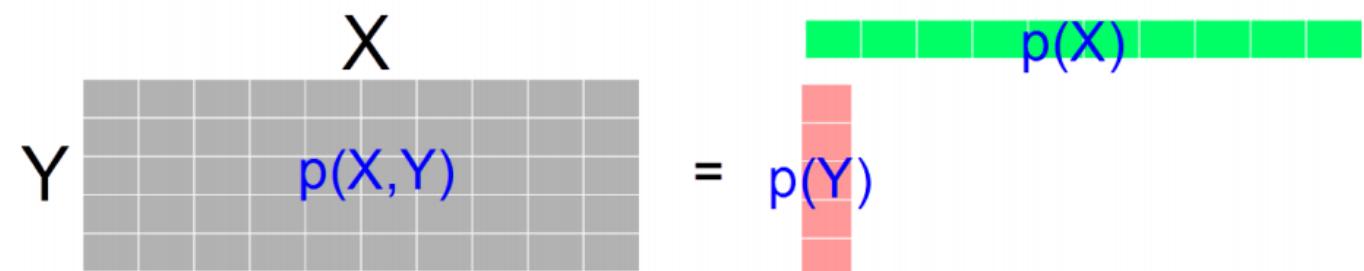
Independence

- X and Y are independent when knowing one tells nothing about the other

$$p(X|Y = y) = p(X)$$

$$p(Y|X = x) = p(Y)$$

$$p(X, Y) = p(X)p(Y)$$



- The above is the marginal independence ($X \perp\!\!\!\perp Y$)
- Two r.v.'s X and Y may not be marginally indep but may be given the value of another r.v. Z

$$p(X, Y|Z = z) = p(X|Z = z)p(Y|Z = z)$$

$$X \perp\!\!\!\perp Y|Z$$



Coming up next

- Some other basic concepts from probability and statistics
- Probabilistic models and parameter estimation in probabilistic models
 - MLE, MAP, Bayesian approaches



Probabilistic Machine Learning (2): Probability Basics (Contd)

CS771: Introduction to Machine Learning
Piyush Rai

Expectation

- Expectation of a random variable tells the expected or average value it takes
- Expectation of a discrete random variable $X \in S_X$ having PMF $p(X)$

$$\mathbb{E}[X] = \sum_{x \in S_X} xp(x)$$

Probability that $X = x$

- Expectation of a continuous random variable $X \in S_X$ having PDF $p(X)$

$$\mathbb{E}[X] = \int_{x \in S_X} xp(x)dx$$

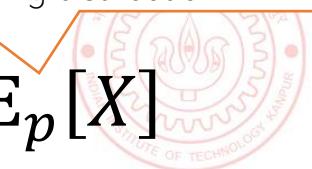
Probability density at $X = x$

Note that this exp. is w.r.t. the distribution $p(f(X))$ of the r.v. $f(X)$

- The definition applies to functions of r.v. too (e.g., $\mathbb{E}[f(X)]$)

Often the subscript is omitted but do keep in mind the underlying distribution

- Exp. is always w.r.t. the prob. dist. $p(X)$ of the r.v. and often written as $\mathbb{E}_p[X]$



Expectation: A Few Rules

X and Y need not be even independent. Can be discrete or continuous

- Expectation of sum of two r.v.'s: $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$

- Proof is as follows

- Define $Z = X + Y$

$$\begin{aligned}
 \mathbb{E}[Z] &= \sum_{z \in S_Z} z \cdot p(Z = z) \quad \text{s.t. } z = x + y \text{ where } x \in S_X \text{ and } y \in S_Y \\
 &= \sum_{x \in S_X} \sum_{y \in S_Y} (x + y) \cdot p(X = x, Y = y) \\
 &= \sum_x \sum_y x \cdot p(X = x, Y = y) + \sum_x \sum_y y \cdot p(X = x, Y = y) \\
 &= \sum_x x \sum_y p(X = x, Y = y) + \sum_y y \sum_x p(X = x, Y = y) \\
 &= \sum_x x \cdot p(X = x) + \sum_y y \cdot p(Y = y) \quad \text{Used the rule of marginalization of joint dist. of two r.v.'s} \\
 &= \mathbb{E}[X] + \mathbb{E}[Y]
 \end{aligned}$$



Expectation: A Few Rules (Contd)

- Expectation of a scaled r.v.: $\mathbb{E}[\alpha X] = \alpha \mathbb{E}[X]$
- Linearity of expectation: $\mathbb{E}[\alpha X + \beta Y] = \alpha \mathbb{E}[X] + \beta \mathbb{E}[Y]$
- (More General) Lin. of exp.: $\mathbb{E}[\alpha f(X) + \beta g(Y)] = \alpha \mathbb{E}[f(X)] + \beta \mathbb{E}[g(Y)]$
- Exp. of product of two **independent** r.v.'s: $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$
- Law of the Unconscious Statistician (LOTUS): Given an r.v. X with a known prob. dist. $p(X)$ and another random variable $Y = g(X)$ for some function g

$$\mathbb{E}[Y] = \mathbb{E}[g(X)] = \sum_{y \in S_Y} y p(y) = \sum_{x \in S_X} g(x) p(x)$$

- Rule of iterated expectation: $\mathbb{E}_{p(X)}[X] = \mathbb{E}_{p(Y)}[\mathbb{E}_{p(X|Y)}[X|Y]]$

α is a real-valued scalar

α and β are real-valued scalars

f and g are arbitrary functions.

Requires finding $p(Y)$

Requires only $p(X)$ which we already have

LOTUS also applicable
for continuous r.v.'s



Variance and Covariance

- Variance of a scalar r.v. tells us about its spread around its mean value $\mathbb{E}[X] = \mu$

$$\text{var}[X] = \mathbb{E}[(X - \mu)^2] = \mathbb{E}[X^2] - \mu^2$$

- Standard deviation is simply the square root of variance
- For two scalar r.v.'s X and Y , the covariance is defined by

$$\text{cov}[X, Y] = \mathbb{E}[\{X - \mathbb{E}[X]\}\{Y - \mathbb{E}[Y]\}] = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]$$

- For two vector r.v.'s X and Y (assume column vec), the covariance matrix is defined by

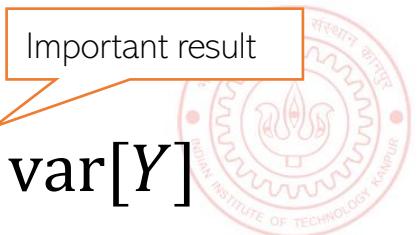
$$\text{cov}[X, Y] = \mathbb{E}[\{X - \mathbb{E}[X]\}\{Y^\top - \mathbb{E}[Y^\top]\}] = \mathbb{E}[XY^\top] - \mathbb{E}[X]\mathbb{E}[Y^\top]$$

- Cov. of components of a vector r.v. X : $\text{cov}[X] = \text{cov}[X, X]$

- Note: The definitions apply to functions of r.v. too (e.g., $\text{var}[f(X)]$)

- Note: Variance of sum of independent r.v.'s: $\text{var}[X + Y] = \text{var}[X] + \text{var}[Y]$

Important result



Transformation of Random Variables

- Suppose $Y = f(X) = AX + b$ be a linear function of a vector-valued r.v. X (A is a matrix and b is a vector, both constants)
- Suppose $\mathbb{E}[X] = \mu$ and $\text{cov}[X] = \Sigma$, then for the vector-valued r.v. Y

$$\mathbb{E}[Y] = \mathbb{E}[AX + b] = A\mu + b$$

$$\text{cov}[Y] = \text{cov}[AX + b] = A\Sigma A^\top$$

- Likewise, if $Y = f(X) = a^\top X + b$ be a linear function of a vector-valued r.v. X (a is a vector and b is a scalar, both constants)
- Suppose $\mathbb{E}[X] = \mu$ and $\text{cov}[X] = \Sigma$, then for the scalar-valued r.v. Y

$$\mathbb{E}[Y] = \mathbb{E}[a^\top X + b] = a^\top \mu + b$$

$$\text{var}[Y] = \text{var}[a^\top X + b] = a^\top \Sigma a$$



Common Probability Distributions

Important: We will use these extensively to model data as well as parameters of models

- Some common discrete distributions and what they can model
 - **Bernoulli**: Binary numbers, e.g., outcome (head/tail, 0/1) of a coin toss
 - **Binomial**: Bounded non-negative integers, e.g., # of heads in n coin tosses
 - **Multinomial/multinoulli**: One of K (>2) possibilities, e.g., outcome of a dice roll
 - **Poisson**: Non-negative integers, e.g., # of words in a document
- Some common continuous distributions and what they can model
 - **Uniform**: numbers defined over a fixed range
 - **Beta**: numbers between 0 and 1, e.g., probability of head for a biased coin
 - **Gamma**: Positive unbounded real numbers
 - **Dirichlet**: vectors that sum of 1 (fraction of data points in different clusters)
 - **Gaussian**: real-valued numbers or real-valued vectors



Coming up next

- Probabilistic Modeling
- Basics of parameter estimation for probabilistic models



Probabilistic Machine Learning (3): Parameter Estimation via Maximum Likelihood

CS771: Introduction to Machine Learning

Piyush Rai

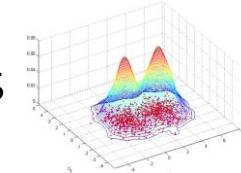
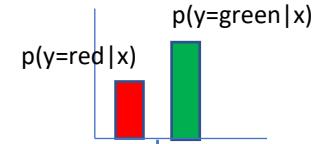
Probabilistic ML: Some Motivation

- In many ML problems, we want to model and reason about data probabilistically
- At a high-level, this is the density estimation view of ML, e.g.,

- Given input-output pairs $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)\}$ estimate the conditional $p(y|\mathbf{x})$
- Given inputs $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$, estimate the distribution $p(\mathbf{x})$ of the inputs
- Note 1: These dist. will depend on some parameters θ (to be estimated), and written as

$$p(y|\mathbf{x}, \theta) \quad \text{or} \quad p(\mathbf{x}|\theta)$$

- Note 2: These dist. sometimes assumed to have a specific form, but sometimes not
- Assuming the form of the distribution to be known, the goal in estimation is to use the observed data to estimate the parameters of these distributions



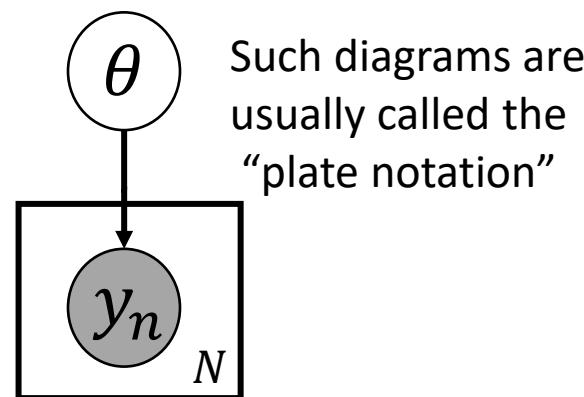
Probabilistic Modeling: The Basic Idea

- Assume N observations $\mathbf{y} = \{y_1, y_2, \dots, y_N\}$, generated from a presumed prob. model

$$y_n \sim p(y|\theta) \quad \forall n \quad (\text{assumed independently \& identically distributed (i.i.d.)})$$

- Here $p(y|\theta)$ is a conditional distribution, conditioned on params θ (to be learned)
 - Note: θ may be fixed unknown or an unknown random variable (we will study both cases)

The parameters θ may themselves depend on other unknown/known parameters (called hyperparameters), which may depend on other unknowns, and so on. ☺ This is essentially "hierarchical" modeling (will see various examples later)



Such diagrams are usually called the "plate notation"

The Predictive dist. tells us how likely each possible value of a new observation y_* is. Example: if y_* denotes the outcome of a coin toss, then what is $p(y_* = \text{"head"} | \mathbf{y})$, given N previous coin tosses $\mathbf{y} = \{y_1, y_2, \dots, y_N\}$



- Some of the tasks that we may be interested in

- Parameter estimation:** Estimating the unknown parameters θ (and other unknowns θ depends on)
- Prediction:** Estimating the **predictive distribution** of new data, i.e., $p(y_* | \mathbf{y})$ - this is also a conditional distribution (conditioned on past data $\mathbf{y} = \{y_1, y_2, \dots, y_N\}$, as well as θ and other things)



Parameter Estimation in Probabilistic Models

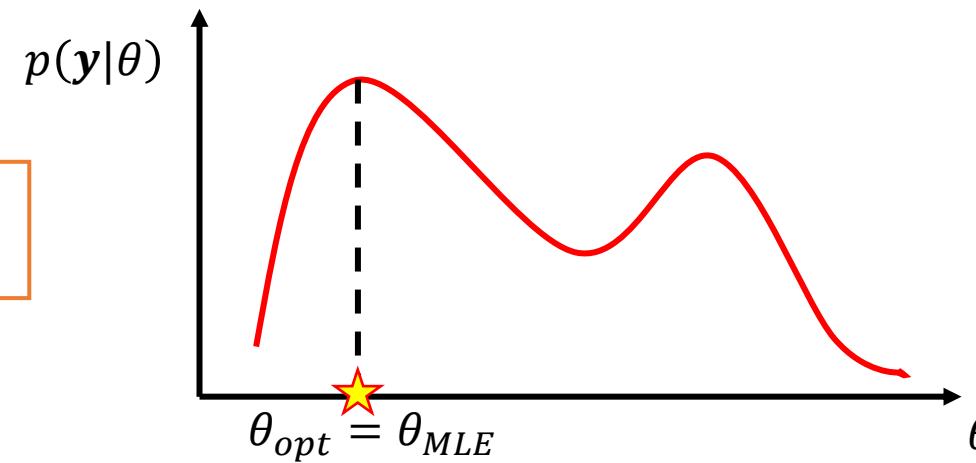
- Since data is assumed to be i.i.d., we can write down its total probability as

$$p(\mathbf{y}|\theta) = p(y_1, y_2, \dots, y_N | \theta) = \prod_{n=1}^N p(y_n | \theta)$$

- $p(\mathbf{y}|\theta)$ called “likelihood” - probability of observed data as a function of params θ



How do I find the best θ ?



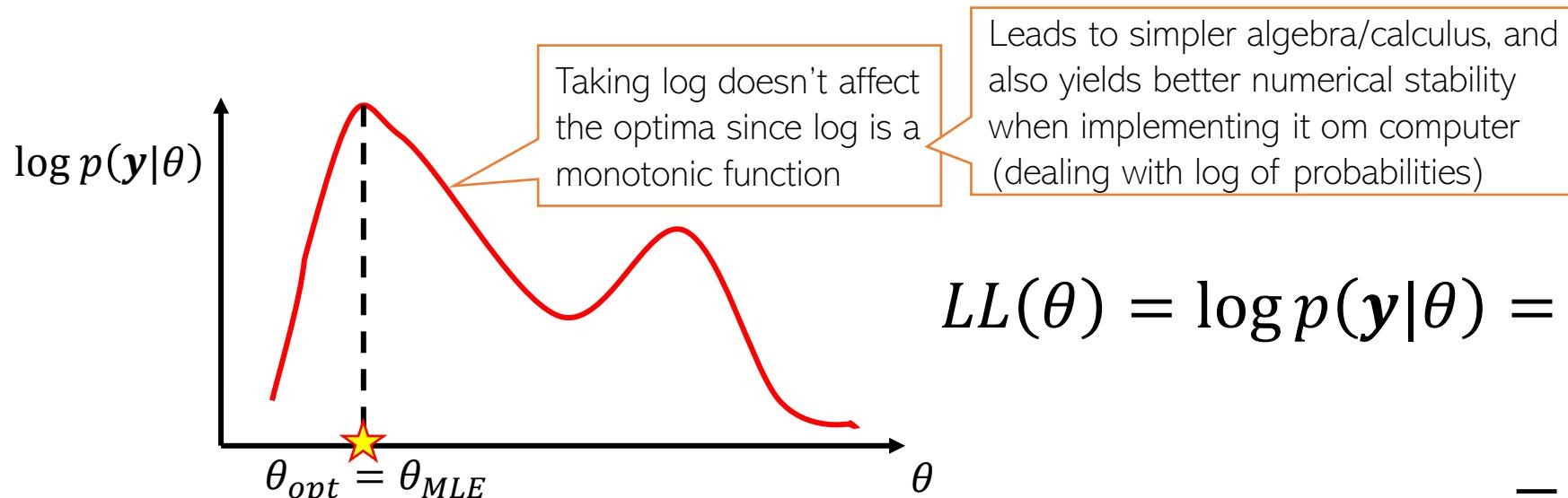
Well, one option is to find the θ that maximizes the likelihood (probability of the observed data) – basically, which value of θ makes the observed data most likely to have come from the assumed distribution $p(\mathbf{y}|\theta)$ --- Maximum Likelihood Estimation (MLE)



- In parameter estimation, the goal is to find the “best” θ , given observed data \mathbf{y}
- Note: Instead of finding single best, sometimes may be more informative to learn a distribution for θ (can tell us about uncertainty in our estimate of θ – more later)

Maximum Likelihood Estimation (MLE)

- The goal in MLE is to find the optimal θ by maximizing the likelihood
- In practice, we maximize the log of the likelihood (**log-likelihood** in short)



$$\begin{aligned} LL(\theta) &= \log p(\mathbf{y}|\theta) = \log \prod_{n=1}^N p(y_n|\theta) \\ &= \sum_{n=1}^N \log p(y_n|\theta) \end{aligned}$$

- Thus the MLE problem is

$$\theta_{MLE} = \operatorname{argmax}_{\theta} LL(\theta) = \operatorname{argmax}_{\theta} \sum_{n=1}^N \log p(y_n|\theta)$$

- This is now an optimization (maximization problem). Note: θ may have constraints

Maximum Likelihood Estimation (MLE)

- The MLE problem can also be easily written as a minimization problem

$$\theta_{MLE} = \operatorname{argmax}_{\theta} \sum_{n=1}^N \log p(y_n | \theta) = \operatorname{argmin}_{\theta} \sum_{n=1}^N -\log p(y_n | \theta)$$

- Thus MLE can also be seen as minimizing the negative log-likelihood (NLL)

$$\theta_{MLE} = \operatorname{argmin}_{\theta} NLL(\theta)$$

- NLL is analogous to a loss function

- The negative log-lik ($-\log p(y_n | \theta)$) is akin to the loss on each data point

- Thus doing MLE is akin to minimizing training loss

Negative Log-Likelihood (NLL)



Indeed. It may overfit. Several ways to prevent it:
Use regularizer or other strategies to prevent overfitting. Alternatives, use “prior” distributions on the parameters θ that we are trying to estimate (which will kind of act as a regularizer as we will see shortly)



Such priors have various other benefits as we will see later



Does it mean MLE could overfit? If so, how to prevent this?



MLE: An Example

- Consider a sequence of N coin toss outcomes (observations)
- Each observation y_n is a binary **random variable**. Head: $y_n = 1$, Tail: $y_n = 0$
- Each y_n is assumed generated by a **Bernoulli distribution** with param $\theta \in (0,1)$

$$p(y_n|\theta) = \text{Bernoulli}(y_n|\theta) = \theta^{y_n} (1 - \theta)^{1-y_n}$$

- Here θ the unknown param (probability of head). Want to estimate it using MLE
- Log-likelihood:** $\sum_{n=1}^N \log p(y_n|\theta) = \sum_{n=1}^N [y_n \log \theta + (1 - y_n) \log (1 - \theta)]$
- Maximizing log-lik (or minimizing NLL) w.r.t. θ will give a closed form expression



I tossed a coin 5 times – gave 1 head and 4 tails. Does it mean $\theta = 0.2??$ The MLE approach says so. What if I see 0 head and 5 tails. Does it mean $\theta = 0?$

$$\theta_{MLE} = \frac{\sum_{n=1}^N y_n}{N}$$

Thus MLE solution is simply the fraction of heads! ☺ Makes intuitive sense!

Indeed – if you want to trust MLE solution. But with small number of training observations, MLE may overfit and may not be reliable. We will soon see better alternatives that use **prior distributions**!



Probability
of a head

Take deriv. set it
to zero and solve.
Easy optimization

Coming up next

- Prior distributions and their role in parameter estimation
 - Maximum-a-Posteriori (MAP) Estimation
 - Fully Bayesian inference
- Probabilistic modeling for regression and classification problems



Probabilistic Machine Learning (4): Parameter Estimation: MAP and Bayesian Inference

CS771: Introduction to Machine Learning

Piyush Rai

MLE and Its Shortcomings..

- MLE finds parameters that make the observed data most probable

$$\theta_{MLE} = \operatorname{argmax}_{\theta} \sum_{n=1}^N \log p(y_n | \theta) = \operatorname{argmin}_{\theta} \sum_{n=1}^N -\log p(y_n | \theta)$$

Log-likelihood
Neg. log-likelihood (NLL)

- No provision to control overfitting (MLE is just like minimizing training loss)
- How do we regularize probabilistic models in a principled way?
- Also, MLE gives only a single “best” answer (“**point estimate**”)
 - .. and it may not be very reliable, especially when we have very little data
 - Desirable: Report a probability distribution over the learned params instead of point est
- Prior distributions provide a nice way to accomplish such things!

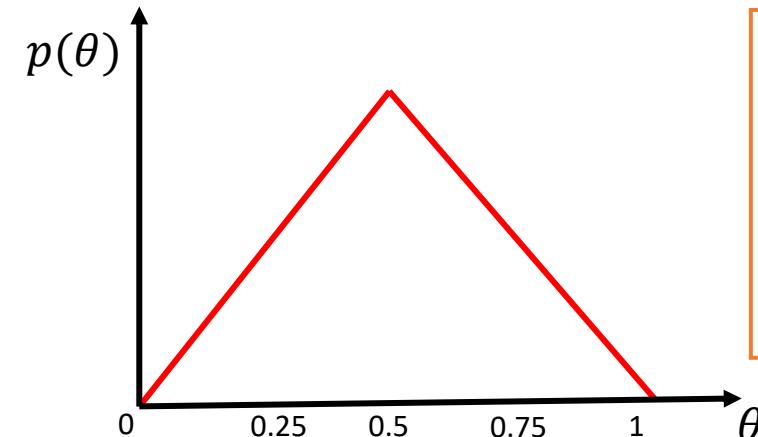
This distribution can give us a sense about the uncertainty in the parameter estimate



Priors

- Can specify our prior belief about likely param values via a prob. dist., e.g.,

This is a rather simplistic/contrived prior. ☺
 Just to illustrate the basic idea. We will see more concrete examples of priors shortly.
 Also, the prior usually depends (assumed conditioned on) on some fixed/learnable hyperparameters (say some α and β , and written as $p(\theta|\alpha, \beta)$)



A possible prior for the coin bias estimation problem. The unknown θ is being treated as a **random variable**, not simply a fixed unknown as we treated it as in MLE



- Once we observe the data \mathbf{y} , apply Bayes rule to update prior into posterior

$$\text{Posterior} \xrightarrow{\quad\quad\quad} p(\theta|\mathbf{y}) = \frac{\text{Prior} \cdot \text{Likelihood}}{\text{Marginal likelihood}} = \frac{p(\theta)p(\mathbf{y}|\theta)}{p(\mathbf{y})}$$

Note: Marginal lik. is hard to compute in general as it requires a **summation or integral** which may not be easy (will briefly look at this in CS771, although will stay away going too deep in this course – CS775 does that in more detail)

- Two ways now to report the answer now:

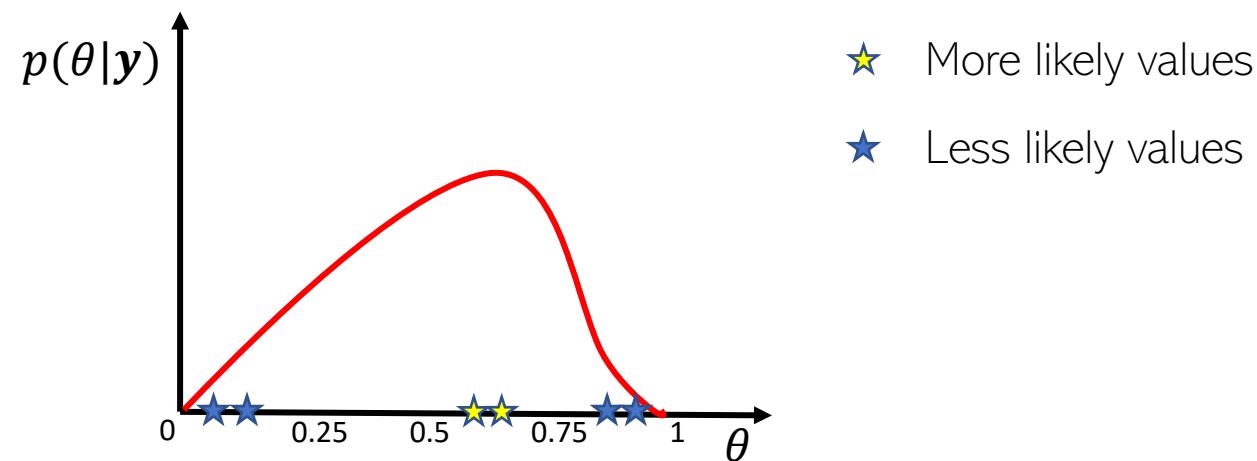
- Report the **maxima** (mode) of the posterior: $\arg \max_{\theta} p(\theta|\mathbf{y})$
- Report the **full posterior** (and its properties, e.g., mean, mode, variance, quantiles, etc.)

Maximum-a-posteriori (MAP) estimation

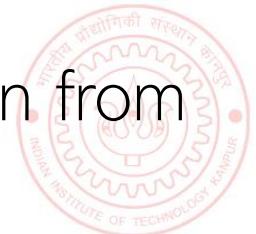
Fully Bayesian inference

Posterior

- Posterior distribution tells us how probable different parameter values are after we have observed some data
- Height of posterior at each value gives the posterior probability of that value



- Can think of the posterior as a “hybrid” obtained by combining information from the likelihood and the prior



Maximum-a-Posteriori (MAP) Estimation

- The MAP estimation approach reports the maxima/mode of the posterior

$$\theta_{MAP} = \arg \max_{\theta} p(\theta|y) = \arg \max_{\theta} \log p(\theta|y) = \arg \max_{\theta} \log \frac{p(\theta)p(y|\theta)}{p(y)}$$

- Since $p(y)$ is constant w.r.t. θ , the above simplifies to

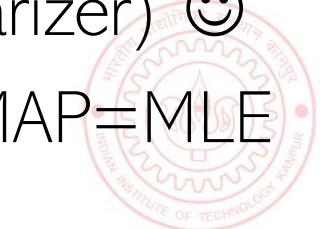
$$\begin{aligned}\theta_{MAP} &= \arg \max_{\theta} [\log p(y|\theta) + \log p(\theta)] \\ &= \arg \min_{\theta} [-\log p(y|\theta) - \log p(\theta)]\end{aligned}$$

$$\boxed{\theta_{MAP} = \arg \min_{\theta} [NLL(\theta) - \log p(\theta)]}$$

The NLL term acts like the training loss and the (negative) log-prior acts as regularizer. Keep in mind this analogy. ☺



- Same as MLE with an extra log-prior-distribution term (acts as a regularizer) ☺
- If the prior is absent or uniform (all values equally likely a prior) then MAP=MLE

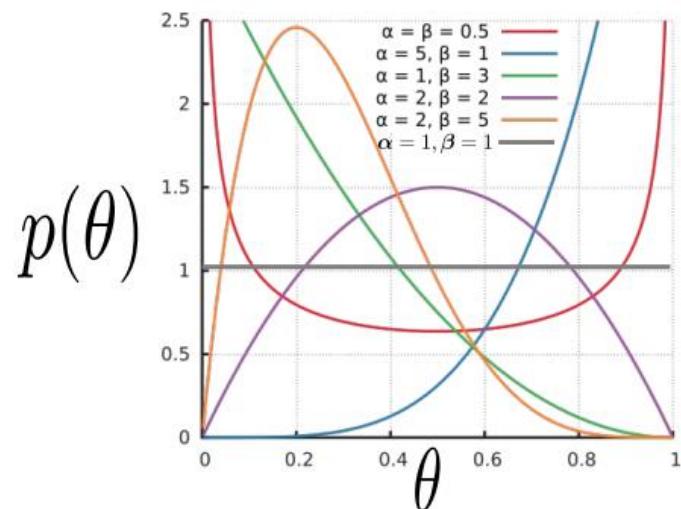


MAP Estimation: An Example

- Let's again consider the coin-toss problem (estimating the bias of the coin)
- Each likelihood term is Bernoulli

$$p(y_n|\theta) = \text{Bernoulli}(y_n|\theta) = \theta^{y_n} (1 - \theta)^{1-y_n}$$

- Also need a prior since we want to do MAP estimation
- Since $\theta \in (0,1)$, a reasonable choice of prior for θ would be Beta distribution



$$p(\theta|\alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1 - \theta)^{\beta-1}$$

The gamma function

Using $\alpha = 1$ and $\beta = 1$ will make the Beta prior a uniform prior

α and β (both non-negative reals) are the two hyperparameters of this Beta prior

Can set these based on intuition, cross-validation, or even learn them

MAP Estimation: An Example (Contd)

- The log posterior for the coin-toss model is log-lik + log-prior

$$LP(\theta) = \sum_{n=1}^N \log p(y_n|\theta) + \log p(\theta|\alpha, \beta)$$

- Plugging in the expressions for Bernoulli and Beta and ignoring any terms that don't depend on θ , the log posterior simplifies to

$$LP(\theta) = \sum_{n=1}^N [y_n \log \theta + (1 - y_n) \log(1 - \theta)] + (\alpha - 1) \log \theta + (\beta - 1) \log(1 - \theta)$$

- Maximizing the above log post. (or min. of its negative) w.r.t. θ gives

Using $\alpha = 1$ and $\beta = 1$ gives us the same solution as MLE

Recall that $\alpha = 1$ and $\beta = 1$ for Beta distribution is in fact equivalent to a uniform prior (hence making MAP equivalent to MLE)

$$\theta_{MAP} = \frac{\sum_{n=1}^N y_n + \alpha - 1}{N + \alpha + \beta - 2}$$

Such interpretations of prior's hyperparameters as being "pseudo-observations" exist for various other prior distributions as well (in particular, distributions belonging to "exponential family" of distributions)

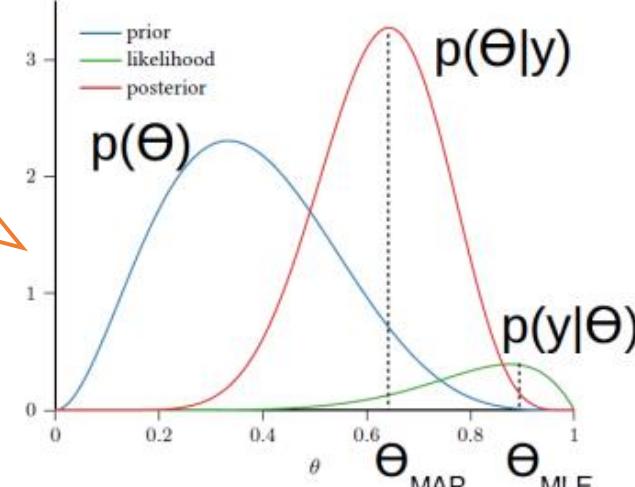
Prior's hyperparameters have an interesting interpretation. Can think of $\alpha - 1$ and $\beta - 1$ as the number of heads and tails, respectively, before starting the coin-toss experiment (akin to "pseudo-observations")



Fully Bayesian Inference

- MLE/MAP only give us a point estimate of θ

MAP estimate is more robust than MLE (due to the regularization effect) but the estimate of uncertainty is missing in both approaches – both just return a single “optimal” solution by solving an optimization problem



Interesting fact to keep in mind: Note that the use of the prior is making the MLE solution move towards the prior (MAP solution is kind of a “compromise between MLE solution of the mode of the prior) 😊



Fully Bayesian inference

- If we want more than just a point estimate, we can compute the full posterior

Computable analytically only when the prior likelihood are “friends” with each other (i.e., they form a **conjugate pair** of distributions (distributions from **exponential family** have conjugate priors)

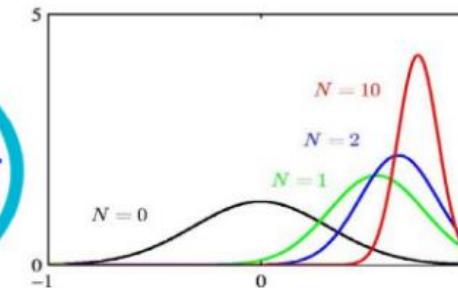
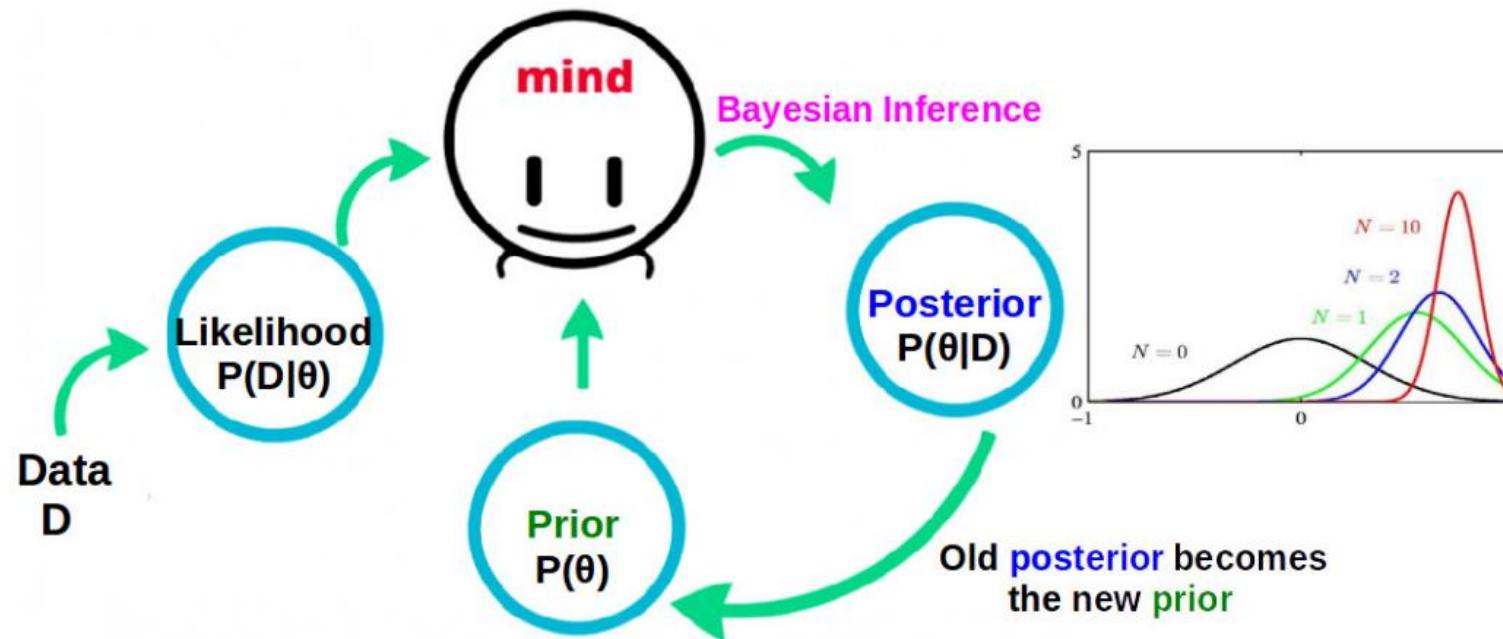
$$p(\theta|y) = \frac{p(\theta)p(y|\theta)}{p(y)}$$

An example: Bernoulli and Beta are conjugate. Will see some more such pairs

In other cases, the posterior needs to be approximated (will see 1-2 such cases in this course; more detailed treatment in the advanced course on probabilistic modeling and inference)

“Online” Nature of Bayesian Inference

- Fully Bayesian inference fits naturally into an “online” learning setting



Also, the posterior becomes more and more “concentrated” as the number of observations increases. For very large N , you may expect it to be peak around the MLE solution



- Our belief about θ keeps getting updated as we see more and more data



Fully Bayesian Inference: An Example

- Let's again consider the coin-toss problem

Bernoulli likelihood: $p(y_n|\theta) = \text{Bernoulli}(y_n|\theta) = \theta^{y_n} (1 - \theta)^{1-y_n}$

Beta prior: $p(\theta) = \text{Beta}(\theta|\alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1 - \theta)^{\beta-1}$

- The posterior can be computed as

$$p(\theta|y) = \frac{p(\theta)p(y|\theta)}{p(y)} = \frac{p(\theta) \prod_{n=1}^N p(y_n|\theta)}{p(y)}$$

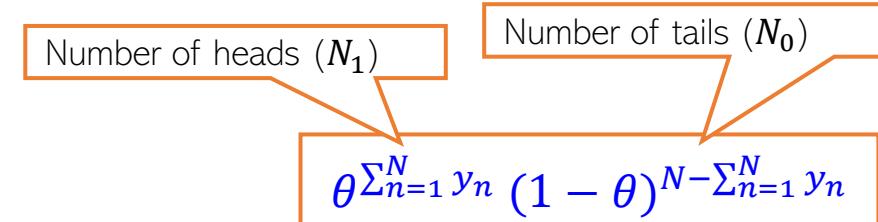
This is the numerator integrated/marginalized over θ : $p(y) = \int p(\theta, y)d\theta = \int p(\theta)p(y|\theta)d\theta$

In general, hard but with conjugate pairs of prior and likelihood, we don't need to compute this, as we will see in this example ☺

This, of course, is not always possible but only in simple cases like this

Also, if you get more observations, you can treat the current posterior as the new prior and obtain a new posterior using these extra observations

Posterior is the same distribution as the prior (both Beta), just with updated hyperparameters (property when likelihood and prior are conjugate to each other)



$$= \frac{\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1-\theta)^{\beta-1} \prod_{n=1}^N \theta^{y_n} (1-\theta)^{1-y_n}}{\int \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1-\theta)^{\beta-1} \prod_{n=1}^N \theta^{y_n} (1-\theta)^{1-y_n} d\theta}$$

$$\propto \theta^{\alpha+N_1-1} (1-\theta)^{\beta+N_0-1}$$

Parts coming from the numerator, which consist of θ terms. We have ignored other constants in the numerator, and the whole denominator which is also constant w.r.t. θ

Found the posterior just by simple inspection without having to calculate the constant of proportionality ☺

Aha! This is nothing but $\text{Beta}(\theta|\alpha + N_1, \beta + N_0)$

Conjugacy

- Many pairs of distributions are conjugate to each other

- Bernoulli (likelihood) + Beta (prior) \Rightarrow Beta posterior
- Binomial (likelihood) + Beta (prior) \Rightarrow Beta posterior
- Multinomial (likelihood) + Dirichlet (prior) \Rightarrow Dirichlet posterior
- Poisson (likelihood) + Gamma (prior) \Rightarrow Gamma posterior
- Gaussian (likelihood) + Gaussian (prior) \Rightarrow Gaussian posterior
- and many other such pairs ..

Not true in general, but in some cases (e.g., when mean of the Gaussian prior is fixed)

- Tip: If two distr are conjugate to each other, their functional forms are similar
- Example: Bernoulli and Beta have the forms

$$\text{Bernoulli}(y|\theta) = \theta^y (1-\theta)^{1-y}$$

$$\text{Beta}(\theta|\alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1-\theta)^{\beta-1}$$

This is why, when we multiply them while computing the posterior, the exponents get added and we get the same form for the posterior as the prior but with just updated hyperparameter. Also, we can identify the posterior and its hyperparameters simply by inspection



Probabilistic Models: Making Predictions

- Having estimated θ , we can now use it to make predictions
- Prediction entails computing the **predictive distribution** of a new observation, say y_*

$$p(y_*|\mathbf{y}) = \int p(y_*, \theta|\mathbf{y})d\theta$$

Marginalizing over the unknown θ

Conditional distribution of the new observation, given past observations

$$= \int p(y_*|\theta, \mathbf{y})p(\theta|\mathbf{y})d\theta$$

Decomposing the joint using chain rule

$$= \int p(y_*|\theta)p(\theta|\mathbf{y})d\theta$$

Assuming i.i.d. data, given θ , y_* does not depend on \mathbf{y}

- When doing MLE/MAP, we approximate the posterior $p(\theta|\mathbf{y})$ by a single point θ_{opt}

$$p(y_*|\mathbf{y}) = \int p(y_*|\theta)p(\theta|\mathbf{y})d\theta \approx p(y_*|\theta_{opt})$$

A "plug-in prediction" (simply plugged in the single estimate we had)

- When doing fully Bayesian est, getting the predictive dist. Will require computing

$$p(y_*|\mathbf{y}) = \int p(y_*|\theta)p(\theta|\mathbf{y})d\theta$$

$$\mathbb{E}_{p(\theta|\mathbf{y})}[p(y_*|\theta)]$$

This computes the predictive distribution by averaging over the **full posterior** – basically calculate $p(y_*|\theta)$ for each possible θ , weighs it by how likely this θ is under the posterior $p(\theta|\mathbf{y})$, and sum all such posterior weighted predictions. Note that not each value of theta is given equal importance here in the averaging



Probabilistic Models: Making Predictions (Example)¹³

- For coin-toss example, let's compute probability of the $(N + 1)^{th}$ toss showing head
- This can be done using the MLE/MAP estimate, or using the full posterior

$$\theta_{MLE} = \frac{N_1}{N} \quad \theta_{MAP} = \frac{N_1 + \alpha - 1}{N + \alpha + \beta - 2} \quad p(\theta|\mathbf{y}) = \text{Beta}(\theta|\alpha + N_1, \beta + N_0)$$

- Thus for this example (where observations are assumed to come from a Bernoulli)

$$\text{MLE prediction: } p(y_{N+1} = 1|\mathbf{y}) = \int p(y_{N+1} = 1|\theta)p(\theta|\mathbf{y})d\theta \approx p(y_{N+1} = 1|\theta_{MLE}) = \theta_{MLE} = \frac{N_1}{N}$$

$$\text{MAP prediction: } p(y_{N+1} = 1|\mathbf{y}) = \int p(y_{N+1} = 1|\theta)p(\theta|\mathbf{y})d\theta \approx p(y_{N+1} = 1|\theta_{MAP}) = \theta_{MAP} = \frac{N_1 + \alpha - 1}{N + \alpha + \beta - 2}$$

$$\text{Fully Bayesian: } p(y_{N+1} = 1|\mathbf{y}) = \int p(y_{N+1} = 1|\theta)p(\theta|\mathbf{y})d\theta = \int \theta p(\theta|\mathbf{y})d\theta = \int \theta \text{Beta}(\theta|\alpha + N_1, \beta + N_0)d\theta = \frac{N_1 + \alpha}{N + \alpha + \beta}$$



Again, keep in mind that the posterior weighted averaged prediction used in the fully Bayesian case would usually not be as simple to compute as it was in this case. We will look at some hard cases later

Expectation of θ under the Beta posterior that we computed using fully Bayesian inference



Probabilistic Modeling: A Summary

- Likelihood corresponds to a loss function; prior corresponds to a regularizer
- Can choose likelihoods and priors based on the nature/property of data/parameters
- MLE estimation = unregularized loss function minimization
- MAP estimation = regularized loss function minimization
- Allows us to do fully Bayesian learning (learning the full distribution of the parameters)
- Makes robust predictions by posterior averaging (rather than using point estimate)
- Many other benefits, such as
 - Estimate of confidence in the model's prediction (useful for doing [Active Learning](#))
 - Can do automatic model selection, hyperparameter estimation, handle missing data, etc.
 - Formulate latent variable models
 - .. and many other benefits (a proper treatment deserves a separate course, but we will see some of these in this course, too)



Coming up next

- Probabilistic modeling for regression and classification problems



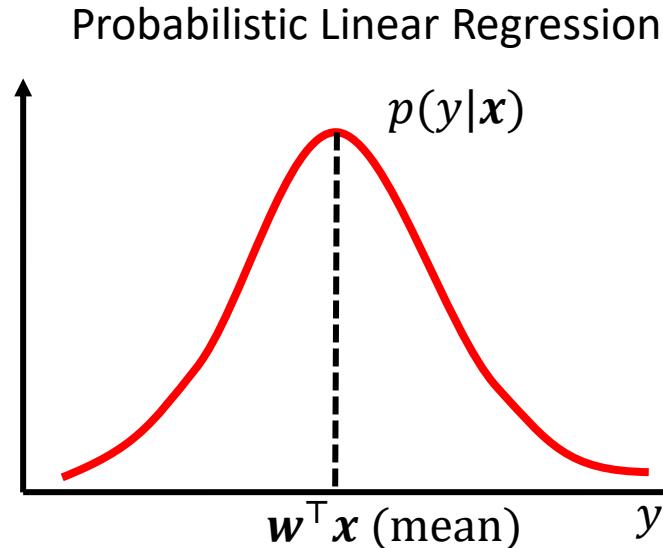
Probabilistic Models for Supervised Learning(1): Probabilistic Linear Regression

CS771: Introduction to Machine Learning

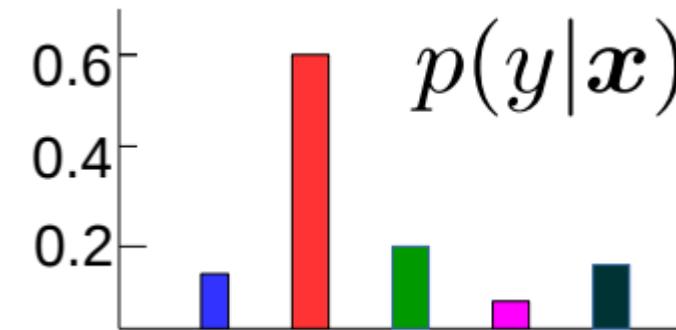
Piyush Rai

Probabilistic Models for Supervised Learning

- Goal: Learn the conditional distribution of output given input, i.e., $p(y|x)$



Probabilistic Classification

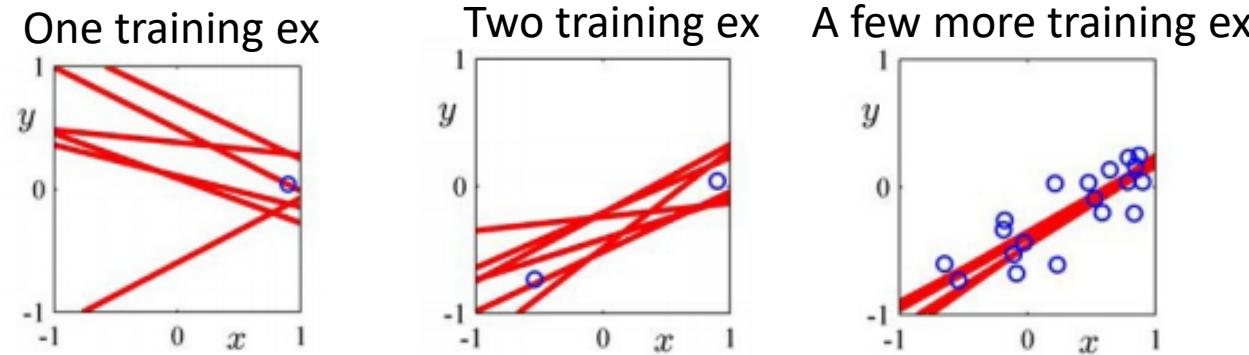


- $p(y|x)$ is more informative than a single prediction y
 - From $p(y|x)$, can get “expected” or “most likely” output y
 - For classifn, “soft” predictions (e.g., rather than yes/no, prob. of “yes”)
 - “Uncertainty” in the predicted output y (e.g., by looking at the variance of $p(y|x)$)
- Can also learn a distribution over the model params using **fully Bayesian inference**

Such uncertainty also helps in “active learning” where we wish to identify “difficult” (and hence more useful) training examples

Distribution over model parameters??

- Recall that linear/ridge regression gave a single “optimal” weight vector
- With a probabilistic model for linear regression, we have two options
 - Use MLE/MAP to get a single “optimal” weight vector
 - Use fully Bayesian inference to learn a distribution over weight vectors (figure below)



$$p(y_*|X, y) = \int p(y_*|\mathbf{w}, \mathbf{x}) p(\mathbf{w}|X, y) d\mathbf{w}$$

Posterior predictive distribution by doing posterior weighted averaging over all possible \mathbf{w} , not just the most likely one. Thus more robust predictions especially if we are uncertain about the best solution.

Predictive distribution using a single \mathbf{w} (plug-in predictive distribution)

How important/like this \mathbf{w} is under the posterior distribution (its posterior probability)

Rather than returning just a single “best” solution (a line in this example), the fully Bayesian approach would give us several “probable” lines (consistent with training data) by learning the full posterior distribution over the model parameters (each of which corresponds to a line)



In this course, we will mostly focus on probabilistic ML when using MLE/MAP and predictive distributions computed using a single best estimate (MLE/MAP). We will only briefly look some simple examples with fully Bayesian approach (CS772/775 covers this approach in greater depth)

Probabilistic Models for Supervised Learning

- Usually two ways to model the conditional distribution $p(y|\mathbf{x})$
- Approach 1: Don't model \mathbf{x} , and model $p(y|\mathbf{x})$ directly using a prob. distribution

"discriminative" sup learning

Gaussian distribution

$$p(y|\mathbf{x}, \mathbf{w}) = \mathcal{N}(y|\mathbf{w}^\top \mathbf{x}, \beta^{-1})$$

Probabilistic linear regression

$$p(y|\mathbf{x}, \mathbf{w}) = \text{Bernoulli}(y|\sigma(\mathbf{w}^\top \mathbf{x}))$$

The "sigmoid" function

Probabilistic linear
binary classification

We assume the conditional distribution to be some appropriate distribution and treat the weights \mathbf{w} as learnable parameters of the model (using MLE/MAP/fully Bayesian inference). Need not be a linear model – can replace $\mathbf{w}^\top \mathbf{x}$ by a nonlinear function $f(\mathbf{x})$



- Approach 2: Model both \mathbf{x} and y via their joint distr. and get the conditional as

"generative" sup learning

Called "generative" because we are learning the generative distributions for output as well as inputs

$$p(y|\mathbf{x}, \theta) = \frac{p(\mathbf{x}, y|\theta)}{p(\mathbf{x}|\theta)}$$

Here θ denotes all the model parameters that we need to model the joint distribution of \mathbf{x} and y (will see examples later)

$$p(y=k|\mathbf{x}, \theta) = \frac{p(\mathbf{x}, y=k|\theta)}{p(\mathbf{x}|\theta)} = \frac{p(\mathbf{x}|y=k, \theta)p(y=k|\theta)}{\sum_{\ell=1}^K p(\mathbf{x}|y=\ell, \theta)p(y=\ell|\theta)}$$

Prob. distribution of inputs from class k

For a multi-class classification model with K classes



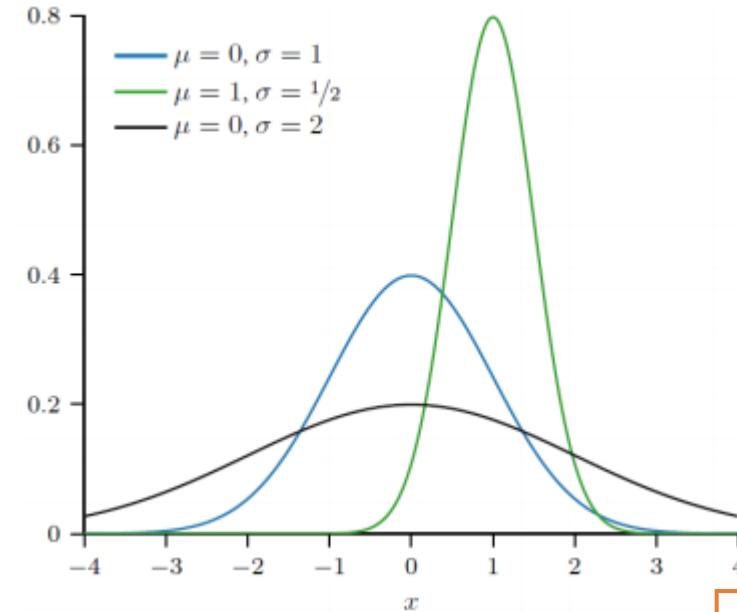
Brief Detour (Gaussian Distribution)



Gaussian Distribution (Univariate)

- Distribution over real-valued scalar random variables $x \in \mathbb{R}$
- Defined by a scalar mean μ and a scalar variance σ^2

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$



- Mean: $\mathbb{E}[x] = \mu$
- Variance: $\text{var}[x] = \sigma^2$
- Inverse of variance is called precision: $\beta = \frac{1}{\sigma^2}$.

$$\mathcal{N}(x|\mu, \beta) = \sqrt{\frac{\beta}{2\pi}} \exp\left[-\frac{\beta}{2}(x-\mu)^2\right]$$

Gaussian PDF in terms of precision



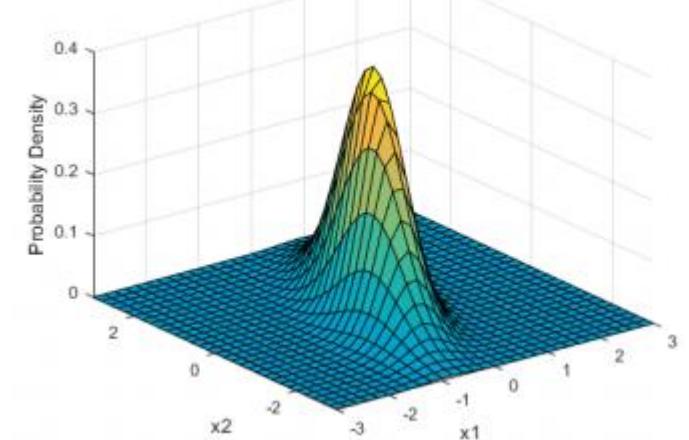
Gaussian Distribution (Multivariate)

- Distribution over real-valued vector random variables $\mathbf{x} \in \mathbb{R}^D$
- Defined by a mean vector $\boldsymbol{\mu} \in \mathbb{R}^D$ and a covariance matrix $\boldsymbol{\Sigma}$

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^D |\boldsymbol{\Sigma}|}} \exp[-(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})]$$

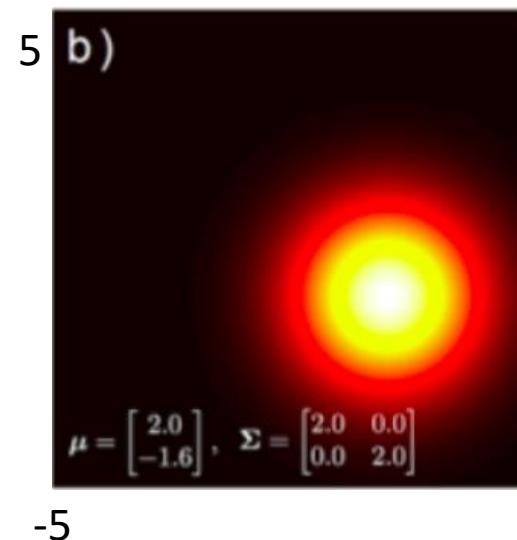
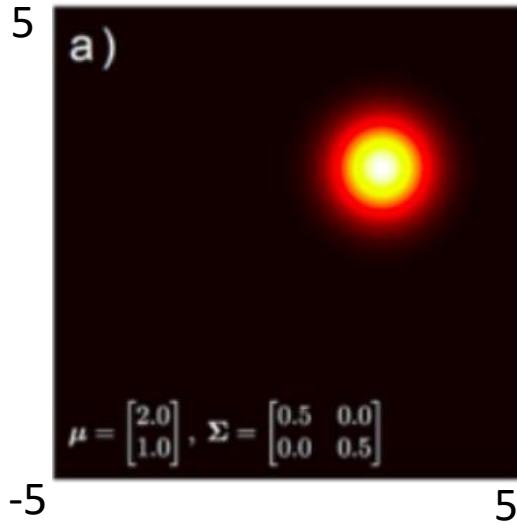
- Note: The cov. matrix $\boldsymbol{\Sigma}$ must be symmetric and PSD
 - All eigenvalues are positive
 - $\mathbf{z}^\top \boldsymbol{\Sigma} \mathbf{z} \geq 0$ for any real vector \mathbf{z}
- The covariance matrix also controls the shape of the Gaussian

A two-dimensional Gaussian

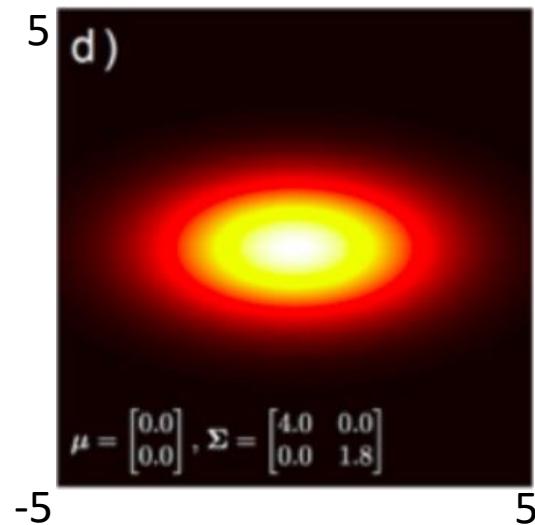
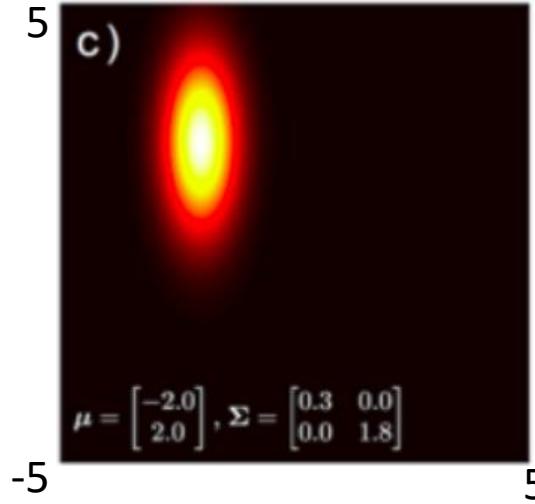


Covariance Matrix for Multivariate Gaussian

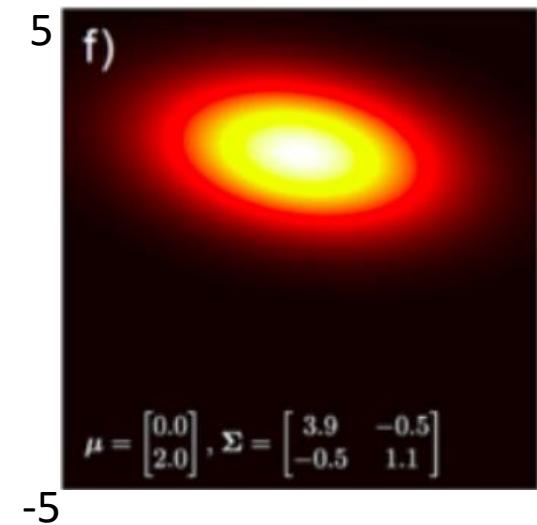
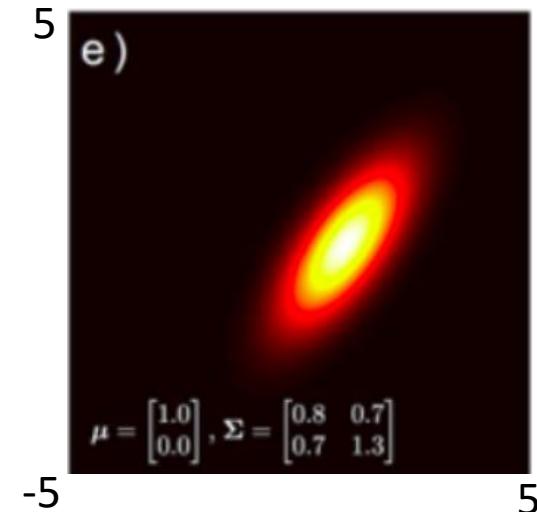
Spherical Covariance



Diagonal Covariance



Full Covariance



Spherical: Equal spreads (variances) along all dimensions



Diagonal: Unequal spreads (variances) along all directions but still axis-parallel

Full: Unequal spreads (variances) along all directions and also spreads along oblique directions



Probabilistic Linear Regression

$$p(y|x, w) = \mathcal{N}(y|w^T x, \beta^{-1})$$

Gaussian distribution

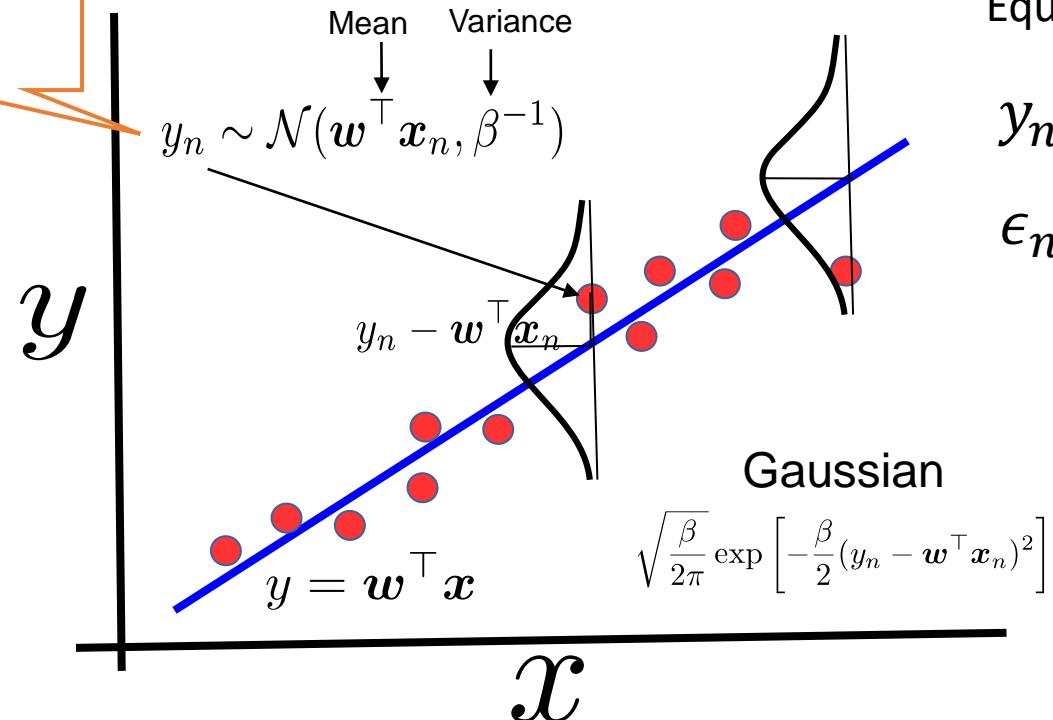
Other distributions can also be used for probabilistic linear regression (e.g., Laplace) as we will see later



Linear Regression: A Probabilistic View

Defines our likelihood model:
 $p(y_n | \mathbf{w}, \mathbf{x}_n)$ - Gaussian

Output y_n assumed generated from a Gaussian with mean $\mathbf{w}^\top \mathbf{x}_n$



Equivalently:

$$y_n = \mathbf{w}^\top \mathbf{x}_n + \epsilon_n$$

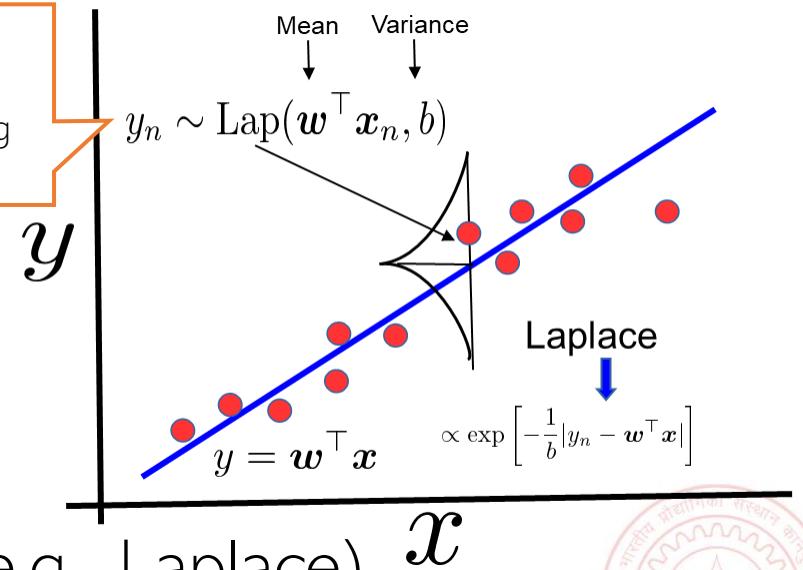
$$\epsilon_n \sim \mathcal{N}(0, \beta^{-1})$$

Output y_n generated from a linear model and then zero mean Gaussian noise added



Note the term in the Gaussian's exponent – just like a squared error we saw for least squares regression ☺

Using a Laplace distribution would correspond to using an absolute loss



- Several variants of this basic model are possible
 - Other distributions to model the additive noise (e.g., Laplace)
 - Different noise variance/precision for each output: $y_n \sim \mathcal{N}(\mathbf{w}^\top \mathbf{x}_n, \beta_n^{-1})$

Heteroskedastic noise

MLE for Probabilistic Linear Regression

- Since each likelihood term is a Gaussian, we have

Also note that \mathbf{x}_n is fixed here but the likelihood depend on it, so it is being conditioned on

Omitting β from the conditioning side for brevity

$$p(y_n | \mathbf{w}, \mathbf{x}_n) = \mathcal{N}(y_n | \mathbf{w}^\top \mathbf{x}_n, \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left[-\frac{\beta}{2}(y_n - \mathbf{w}^\top \mathbf{x}_n)^2\right]$$

Exercise: Verify that you can also write the overall likelihood as a single N dimensional Gaussian with mean $\mathbf{X}\mathbf{w}$ and cov. matrix $\beta^{-1}\mathbf{I}_N$

- Thus the overall likelihood (assuming i.i.d. responses) will be

$$p(\mathbf{y} | \mathbf{X}, \mathbf{w}) = \prod_{n=1}^N p(y_n | \mathbf{x}_n, \mathbf{w}) = \left(\frac{\beta}{2\pi}\right)^{N/2} \exp\left[-\frac{\beta}{2} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2\right]$$

- Log-likelihood (ignoring constants w.r.t. \mathbf{w})

$$\log p(\mathbf{y} | \mathbf{X}, \mathbf{w}) \propto -\frac{\beta}{2} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2$$

MLE for probabilistic linear regression with Gaussian noise is equivalent to least squares regression without any regularization (with solution $\hat{\mathbf{w}}_{MLE} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$)



- Negative log likelihood (NLL) in this case is similar to squared loss function



MAP Estimation for Prob. Lin. Reg.: The Prior

- For MAP estimation, we need a prior distribution over the parameters $\mathbf{w} \in \mathbb{R}^D$
- A reasonable prior for real-valued vectors can be a multivariate Gaussian

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \mathbf{w}_0, \Sigma)$$

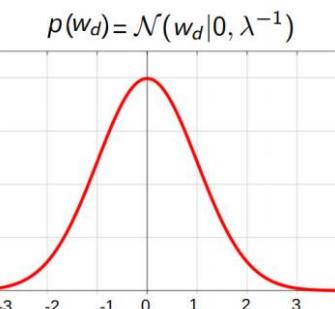
Equivalent to saying that *a priori* we expect the solution to be close to some vector \mathbf{w}_0
(subject to Σ being such that the variances is not too large)

- A specific example of a multivariate Gaussian prior in this problem

Omitting
 λ for brevity

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \mathbf{0}, \lambda^{-1} \mathbf{I}_D) = \prod_{d=1}^D \mathcal{N}(w_d | 0, \lambda^{-1}) = \prod_{d=1}^D p(w_d)$$

The precision λ of the Gaussian prior controls how aggressively the prior pushes the elements towards mean (0)



$$\mathcal{N}(w_d | 0, \lambda^{-1}) = \sqrt{\frac{\lambda}{2\pi}} \exp\left[-\frac{\lambda}{2} w_d^2\right]$$

$$\mathcal{N}(\mathbf{w} | \mathbf{0}, \lambda^{-1} \mathbf{I}_D) = \left(\frac{\lambda}{2\pi}\right)^{D/2} \exp\left[-\frac{\lambda}{2} \sum_{d=1}^D w_d^2\right] = \left(\frac{\lambda}{2\pi}\right)^{D/2} \exp\left[-\frac{\lambda}{2} \mathbf{w}^\top \mathbf{w}\right]$$

This is essentially like a regularizer that pushes elements of \mathbf{w} to be small (we will see shortly)

Aha! This $\mathbf{w}^\top \mathbf{w}$ term reminds me of the ℓ_2 regularizer ☺



That's indeed the case ☺



MAP Estimation for Probabilistic Linear Regression

- The MAP objective (log-posterior) will be the log-likelihood + $\log p(\mathbf{w})$

$$-\frac{\beta}{2} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 - \frac{\lambda}{2} \mathbf{w}^\top \mathbf{w}$$

In the likelihood and prior, ignored terms that don't depend on \mathbf{w}

- Maximizing this is equivalent to minimizing the following w.r.t. \mathbf{w}

$$\hat{\mathbf{w}}_{MAP} = \arg \min_{\mathbf{w}} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \frac{\lambda}{\beta} \mathbf{w}^\top \mathbf{w}$$

Not surprising since MAP estimation indeed optimizes a regularized loss function! 😊



- This is equivalent to ridge regression with regularization hyperparameter $\frac{\lambda}{\beta}$
- The solution will be $\hat{\mathbf{w}}_{MAP} = (\mathbf{X}^\top \mathbf{X} + \frac{\lambda}{\beta} \mathbf{I}_D)^{-1} \mathbf{X}^\top \mathbf{y}$



Fully Bayesian Inference for Prob. Linear Regression

- Can also compute the full posterior distribution over \mathbf{w}

$$p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = \frac{p(\mathbf{w})p(\mathbf{y}|\mathbf{X}, \mathbf{w})}{p(\mathbf{y}|\mathbf{X})}$$

For brevity, we have not shown the dependence of the various distributions here on the hyperparameters λ and β

- Likelihood and prior are conjugate (both Gaussians) - posterior will be Gaussian



Deriving this result requires a bit of algebra (not too hard though).

We already know that the result will be Gaussian (due to conjugacy) – just need to multiply and rearrange terms to bring the result into a Gaussian form and identify the mean and covariance of that Gaussian – can be done using the “completing the squares” trick. Don’t even need to worry about calculating the marginal. Will provide a note

$$p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = \mathcal{N}(\boldsymbol{\mu}_N, \boldsymbol{\Sigma}_N)$$

$$\boldsymbol{\mu}_N = (\mathbf{X}^\top \mathbf{X} + \frac{\lambda}{\beta} \mathbf{I}_D)^{-1} \mathbf{X}^\top \mathbf{y}$$

$$\boldsymbol{\Sigma}_N = (\beta \mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1}$$

Posterior's mean is the same as the MAP solution since the mean and mode of a Gaussian are the same!

Note: λ and β are assumed to be fixed; otherwise, the problem is a bit harder (beyond the scope of CS771)

Alternatively, just think of the posterior $p(\mathbf{w}|\mathbf{X}, \mathbf{y})$ as a reverse conditional of the likelihood $p(\mathbf{y}|\mathbf{X}, \mathbf{w})$ and apply standard results of Gaussians distributions (see maths refresher slides from Week 0)

We now have a distribution over the possible solutions – it has a mean but we can generate other plausible solutions by sampling from this posterior. Each sample will give a weight vector



Prob. Linear Regression: The Predictive Distribution

- Want the predictive distribution $p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y})$ of the output y_* for a new input \mathbf{x}_* .
- With MLE/MAP estimate of \mathbf{w} , we will use the plug-in predictive

$$p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}) \approx p(y_* | \mathbf{x}_*, \mathbf{w}_{MLE}) = \mathcal{N}(\mathbf{w}_{MLE}^\top \mathbf{x}_*, \beta^{-1}) \quad - \text{MLE prediction}$$

$$p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}) \approx p(y_* | \mathbf{x}_*, \mathbf{w}_{MAP}) = \mathcal{N}(\mathbf{w}_{MAP}^\top \mathbf{x}_*, \beta^{-1}) \quad - \text{MAP prediction}$$

- When doing fully Bayesian inference, can compute the posterior predictive dist.

$$p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}) = \int p(y_* | \mathbf{x}_*, \mathbf{w}) p(\mathbf{w} | \mathbf{X}, \mathbf{y}) d\mathbf{w}$$

- Requires an integral but has a closed form

$$p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}) = \mathcal{N}(\mu_N^\top \mathbf{x}_*, \beta^{-1} + \mathbf{x}_*^\top \Sigma_N \mathbf{x}_*)$$

Mean prediction

Not true in general for Prob.
Lin. Reg. but because the
hyperparameters λ and β are
treated as fixed

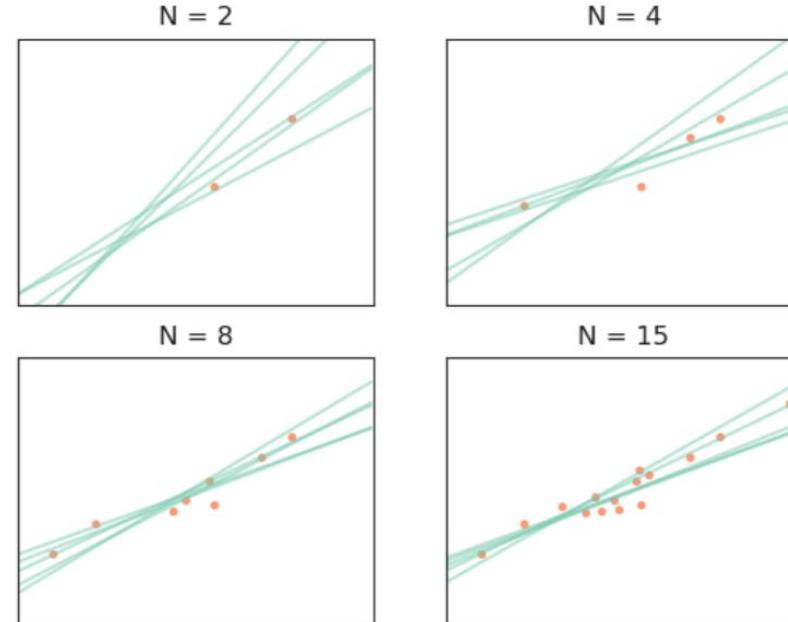
Input-specific predictive variance
unlike the MLE/MAP based
predictive where it was β^{-1} (and
was same for all test inputs)

- Input-specific predictive uncertainty useful in problems where we want confidence estimates of the predictions made by the model (e.g., Active Learning)



Fully Bayesian Linear Regression – Pictorially

- Each sample from posterior $p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = \mathcal{N}(\boldsymbol{\mu}_N, \boldsymbol{\Sigma}_N)$ will give a weight vector \mathbf{w}
 - In case of lin. reg., each weight vector corresponds to a regression line



The posterior sort of represents an ensemble of solutions (not all are equally good but we can use all of them in an “importance-weighted” fashion to make the prediction using the posterior predictive distribution)



Importance of each solution in this ensemble is its posterior probability $p(\mathbf{w}|\mathbf{X}, \mathbf{y})$

- Each weight vector will give a different set of predictions on test data
 - These different predictions will give us a variance (uncertainty) estimate in model’s prediction
 - The uncertainty decreases as N increases (we become more sure when we see more training data)



MLE, MAP/Fully Bayesian Lin. Reg: Summary

- MLE/MAP give point estimate of \mathbf{w}
 - MLE/MAP based prediction uses that single point estimate of \mathbf{w}
- Fully Bayesian approach gives the full posterior of \mathbf{w}
 - Fully Bayesian prediction does posterior averaging (computes posterior predictive distribution)
- Some things to keep in mind:
 - MLE estimation of a parameter leads to unregularized solutions
 - MAP estimation of a parameter leads to regularized solutions
 - A Gaussian likelihood model corresponds to using squared loss
 - A Gaussian prior on parameters acts as an ℓ_2 regularizer
 - Other likelihoods/priors can be chosen (result in other loss functions and regularizers)
- Can extend Bayesian linear regression to handle nonlinear regression
 - Using kernel based feature mapping $\phi(x)$: Gaussian Process regression

E.g., using Laplace distribution for likelihood is equivalent to absolute loss, using it as a prior is equivalent to ℓ_1 regularization



Evaluation Measures for Regression Models

- Plotting the prediction \hat{y}_n vs truth y_n for the validation/test set
- Residual Sum of Squares (RSS) on the validation/test set

$$RSS(\mathbf{w}) = \sum_{n=1}^N (y_n - \hat{y}_n)^2$$

- RMSE (Root Mean Squared Error) $\triangleq \sqrt{\frac{1}{N} RSS(\mathbf{w})}$
- Coefficient of determination or R^2

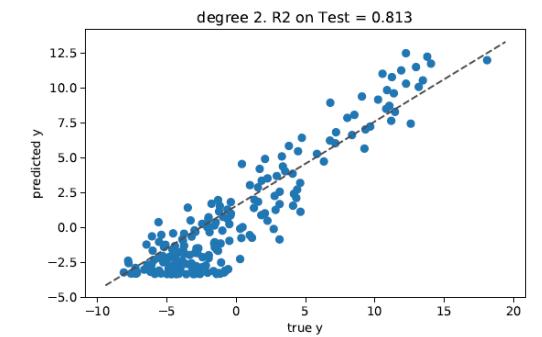
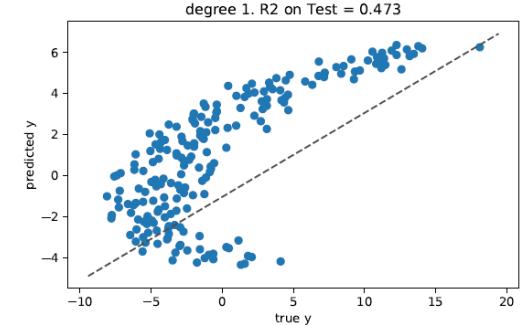
$$R^2 = 1 - \frac{\sum_{n=1}^N (y_n - \hat{y}_n)^2}{\sum_{n=1}^N (y_n - \bar{y})^2}$$

Unlike RSS and RMSE, it is always between 0 and 1 and hence interpretable

\bar{y} is empirical mean of true responses, i.e., $\frac{1}{N} \sum_{n=1}^N y_n$

“relative” error w.r.t. a model that makes a constant prediction \bar{y} for all inputs

Plots of true vs predicted outputs and R^2 for two regression models



Coming up next

- Probabilistic modeling classification problems
 - Logistic regression and softmax regression



Probabilistic Models for Supervised Learning(2): Logistic and Softmax Regression

CS771: Introduction to Machine Learning

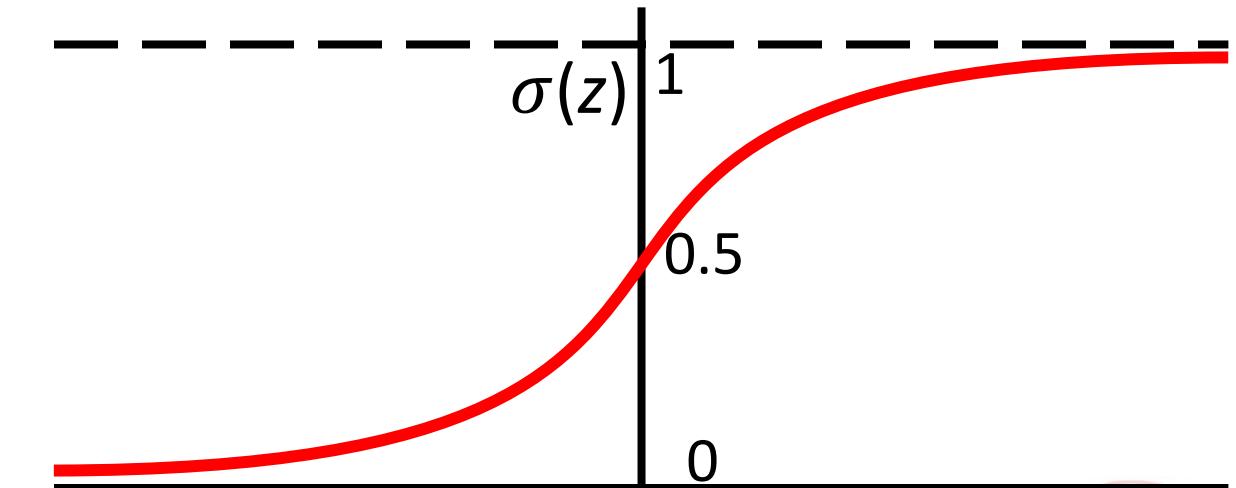
Piyush Rai

Logistic Regression (LR)

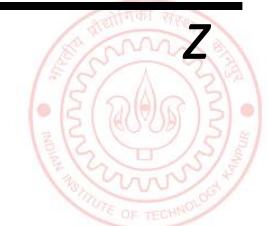
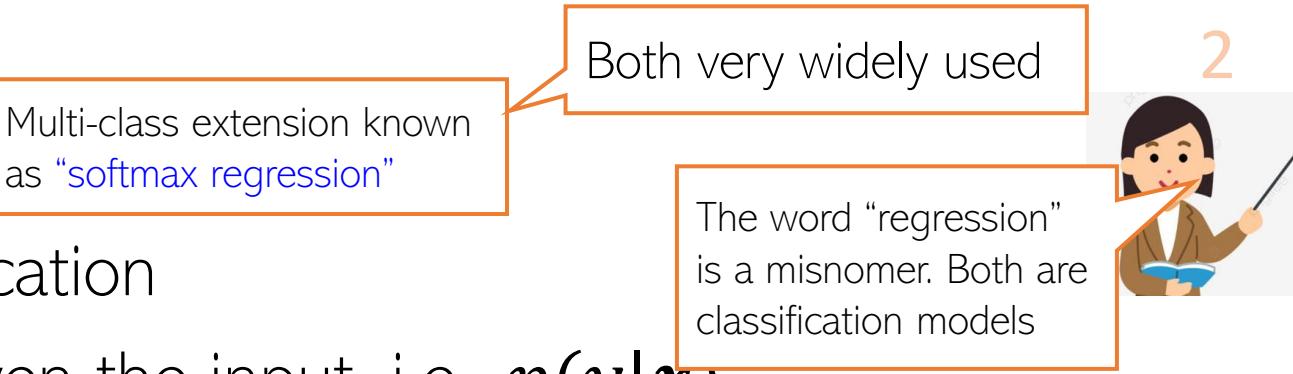
- A probabilistic model for binary classification
- Learns the PMF of the output label given the input, i.e., $p(y|\mathbf{x})$
- A **discriminative model**: Does not model inputs \mathbf{x} (only relationship b/w \mathbf{x} and y)
- Uses the **sigmoid function** to define the conditional probability of y being 1

$$\begin{aligned}\mu_x = p(y = 1 | \mathbf{w}, \mathbf{x}) &= \sigma(\mathbf{w}^\top \mathbf{x}) \\ &= \frac{1}{1 + \exp(-\mathbf{w}^\top \mathbf{x})} \\ &= \frac{\exp(\mathbf{w}^\top \mathbf{x})}{1 + \exp(\mathbf{w}^\top \mathbf{x})}\end{aligned}$$

A linear model



- Here $\mathbf{w}^\top \mathbf{x}$ is the score for input \mathbf{x} . The sigmoid turns it into a probability

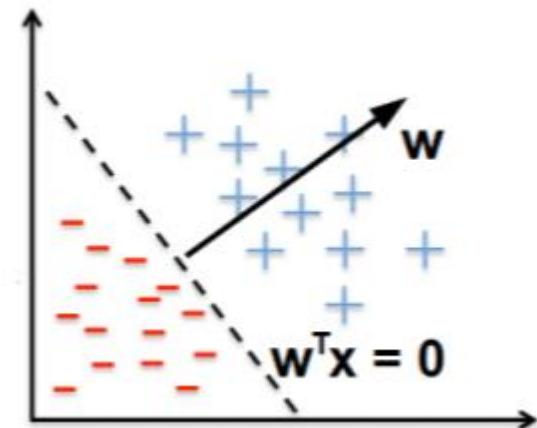


LR: Decision Boundary

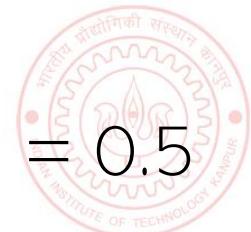
- At the decision boundary where both classes are equiprobable

$$\begin{aligned} p(y = 1 | \mathbf{x}, \mathbf{w}) &= p(y = 0 | \mathbf{x}, \mathbf{w}) \\ \frac{\exp(\mathbf{w}^\top \mathbf{x})}{1 + \exp(\mathbf{w}^\top \mathbf{x})} &= \frac{1}{1 + \exp(\mathbf{w}^\top \mathbf{x})} \\ \exp(\mathbf{w}^\top \mathbf{x}) &= 1 \\ \mathbf{w}^\top \mathbf{x} &= 0 \end{aligned}$$

A linear hyperplane



- Very large positive $\mathbf{w}^\top \mathbf{x}$ means $p(y = 1 | \mathbf{w}, \mathbf{x})$ close to 1
- Very large negative $\mathbf{w}^\top \mathbf{x}$ means $p(y = 0 | \mathbf{w}, \mathbf{x})$ close to 1
- At decision boundary, $\mathbf{w}^\top \mathbf{x} = 0$ implies $p(y = 1 | \mathbf{w}, \mathbf{x}) = p(y = 0 | \mathbf{w}, \mathbf{x}) = 0.5$



MLE for Logistic Regression

Assumed 0/1, not -1/+1

- Likelihood (PMF of each input's label) is Bernoulli with prob $\mu_n = \frac{\exp(\mathbf{w}^\top \mathbf{x}_n)}{1+\exp(\mathbf{w}^\top \mathbf{x}_n)}$
 $p(y_n|\mathbf{w}, \mathbf{x}_n) = \text{Bernoulli}(\mu_n) = \mu_n^{y_n}(1 - \mu_n)^{1-y_n}$
- Overall likelihood, assuming i.i.d. observations

$$p(\mathbf{y}|\mathbf{w}, \mathbf{X}) = \prod_{n=1}^N p(y_n|\mathbf{w}, \mathbf{x}_n) = \prod_{n=1}^N \mu_n^{y_n}(1 - \mu_n)^{1-y_n}$$

- The negative log-likelihood $NLL(\mathbf{w}) = -\log p(\mathbf{y}|\mathbf{w}, \mathbf{X})$ simplifies to

Loss function

$$NLL(\mathbf{w}) = \sum_{n=1}^N -[y_n \log \mu_n + (1 - y_n) \log (1 - \mu_n)]$$

"cross-entropy" loss (a popular loss function for classification)

Very large loss if y_n close to 1 and μ_n close to 0, or vice-versa

- Plugging in $\mu_n = \frac{\exp(\mathbf{w}^\top \mathbf{x}_n)}{1+\exp(\mathbf{w}^\top \mathbf{x}_n)}$ and simplifying

Good news: For LR, NLL is convex

No closed-form expression for
 $\hat{\mathbf{w}}_{MLE} = \arg \min_{\mathbf{w}} NLL(\mathbf{w})$

Iterative opt needed (gradient or Hessian based). Exercise: Try working out the gradient of NLL and notice the expression's form

$$NLL(\mathbf{w}) = - \sum_{n=1}^N [y_n \mathbf{w}^\top \mathbf{x}_n - \log (1 + \exp(\mathbf{w}^\top \mathbf{x}_n))]$$



An Alternate Notation

- If we assume the label y_n as -1/+1 (not 0/1), the likelihood can be written as

$$p(y_n|\mathbf{w}, \mathbf{x}_n) = \frac{1}{1 + \exp(-y_n \mathbf{w}^\top \mathbf{x}_n)} = \sigma(y_n \mathbf{w}^\top \mathbf{x}_n)$$

- Slightly more convenient notation: A single expression gives the probabilities of both possible label values
- In this case, the total negative log-likelihood will be

$$NLL(\mathbf{w}) = \sum_{n=1}^N -\log p(y_n|\mathbf{w}, \mathbf{x}_n) = \sum_{n=1}^N \log (1 + \exp(-y_n \mathbf{w}^\top \mathbf{x}_n))$$



MAP Estimation for Logistic Regression

- Need a prior on the weight vector $\mathbf{w} \in \mathbb{R}^D$
- Just like probabilistic linear regression, can use a zero-mean Gaussian prior

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \mathbf{0}, \lambda^{-1} \mathbf{I}_D) \propto \exp\left(-\frac{\lambda}{2} \mathbf{w}^\top \mathbf{w}\right)$$

Or NLL – log of prior

- The MAP objective (log of posterior) will be log-likelihood + log of prior
- Therefore the MAP solution (ignoring terms that don't depend on \mathbf{w}) will be

$$\hat{\mathbf{w}}_{MAP} = \arg \min_{\mathbf{w}} NLL(\mathbf{w}) + \frac{\lambda}{2} \mathbf{w}^\top \mathbf{w}$$

Good news:
convex objective

- Just like MLE case, no closed form solution. Iterative opt methods needed
 - Highly efficient solvers (both first and second order) exist for MLE/MAP estimation for LR



Fully Bayesian Inference for Logistic Regression

- Doing fully Bayesian inference would require computing the posterior

$$p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = \frac{p(\mathbf{w})p(\mathbf{y}|\mathbf{X}, \mathbf{w})}{p(\mathbf{y}|\mathbf{X})} = \frac{p(\mathbf{w}) \prod_{n=1}^N p(y_n|\mathbf{w}, \mathbf{x}_n)}{\int p(\mathbf{w}) \prod_{n=1}^N p(y_n|\mathbf{w}, \mathbf{x}_n) d\mathbf{w}}$$

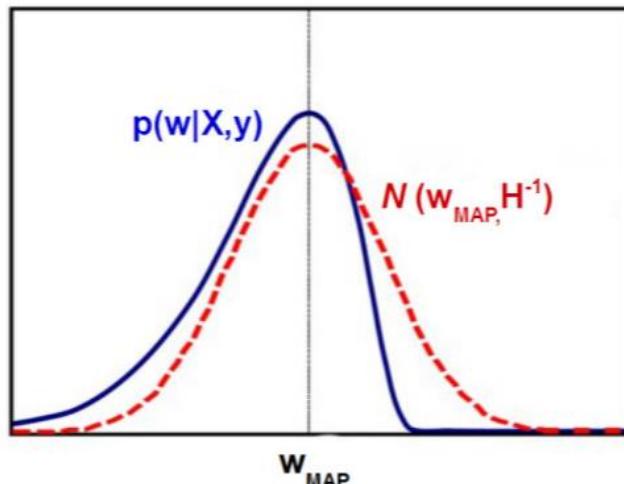
Gaussian

Bernoulli

Unfortunately, Gaussian and Bernoulli are not conjugate with each other, so analytic expression for the posterior can't be obtained unlike prob. linear regression



- Need to approximate the posterior in this case
- We will use a simple approximation called Laplace approximation



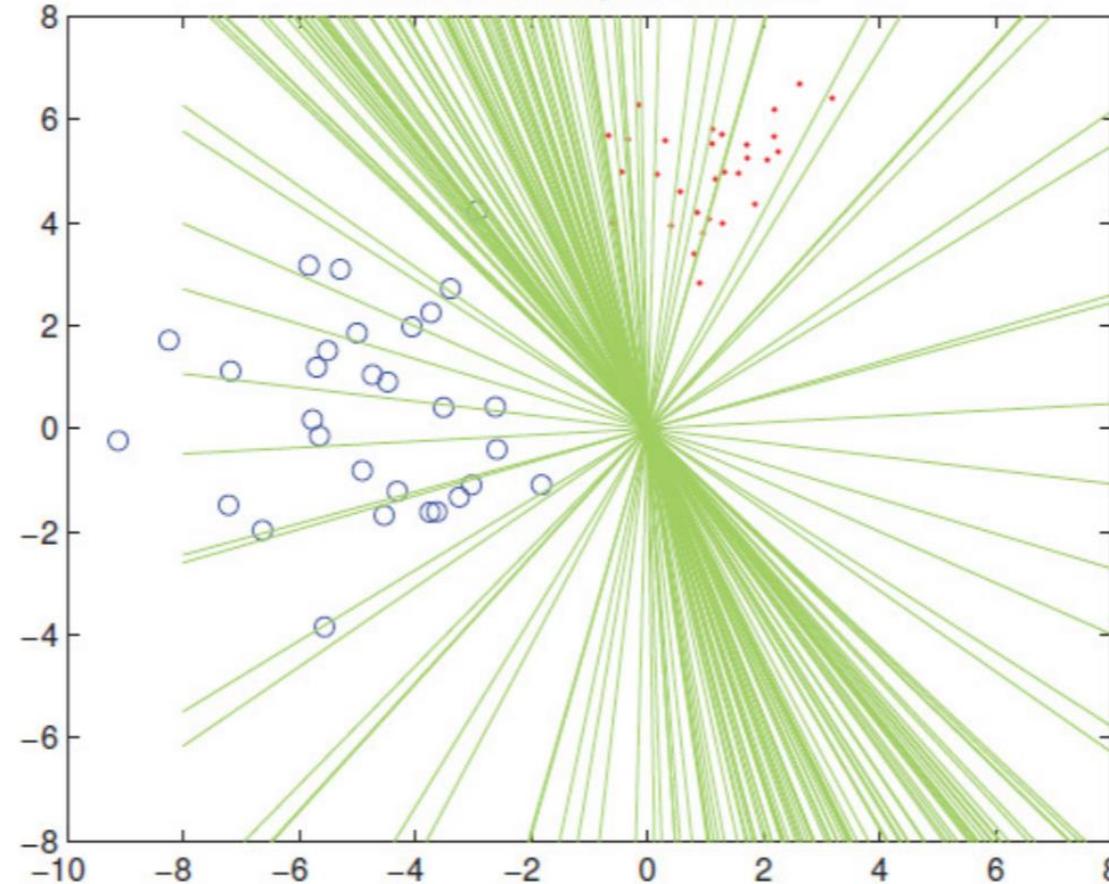
Approximates the posterior of \mathbf{w} by a Gaussian whose mean is the MAP solution $\hat{\mathbf{w}}_{MAP}$ and covariance matrix is the inverse of the Hessian (Hessian: second derivative of the negative log-posterior of the LR model)

Can also employ more advanced posterior approximation methods, like MCMC and variational inference (beyond the scope of CS771)



Posterior for LR: An Illustration

- Can sample from the posterior of the LR model
- Each sample will give a weight vec defining a hyperplane separator



Not all separators are equally good; their goodness depends on their posterior probabilities



When making predictions, we can still use all of them but weighted by their importance based on their posterior probabilities

That's exactly what we do when computing the predictive distribution



Logistic Regression: Predictive Distribution

- When using MLE/MAP solution $\hat{\mathbf{w}}_{opt}$, can use the plug-in predictive distribution

$$\begin{aligned} p(y_* = 1 | \mathbf{x}_*, \mathbf{X}, \mathbf{y}) &= \int p(y_* = 1 | \mathbf{w}, \mathbf{x}_*) p(\mathbf{w} | \mathbf{X}, \mathbf{y}) d\mathbf{w} \\ &\approx p(y_* = 1 | \hat{\mathbf{w}}_{opt}, \mathbf{x}_*) = \sigma(\hat{\mathbf{w}}_{opt}^\top \mathbf{x}_n) \end{aligned}$$

$$p(y_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}) = \text{Bernoulli}[\sigma(\hat{\mathbf{w}}_{opt}^\top \mathbf{x}_n)]$$

- When using fully Bayesian inference, we must compute the posterior predictive

$$p(y_* = 1 | \mathbf{x}_*, \mathbf{X}, \mathbf{y}) = \int p(y_* = 1 | \mathbf{w}, \mathbf{x}_*) p(\mathbf{w} | \mathbf{X}, \mathbf{y}) d\mathbf{w}$$

Integral not tractable and must be approximated

sigmoid

Gaussian (if using Laplace approx.)

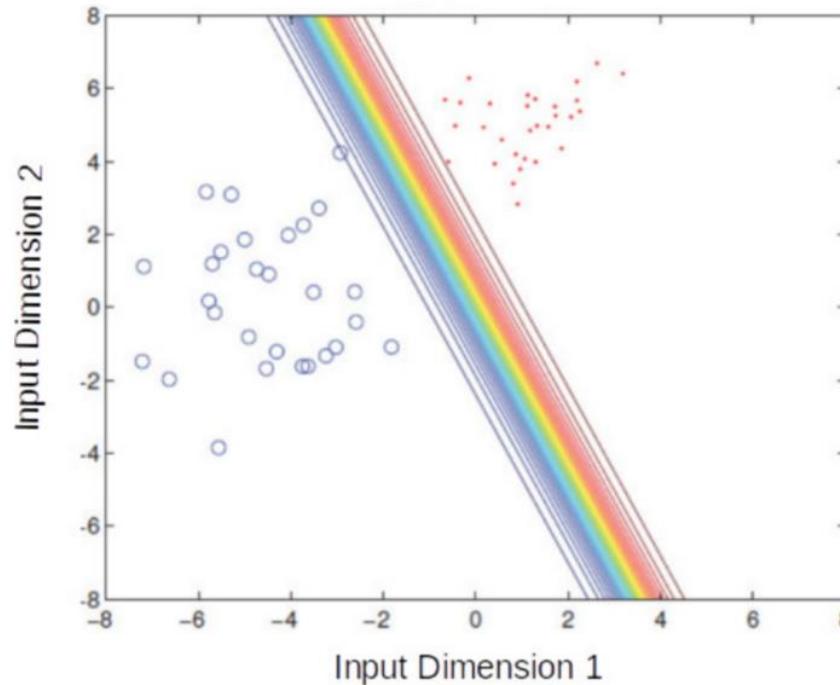
Monte-Carlo approximation of this integral is one possible way

Generate M samples $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_M$, from the Gaussian approx. of posterior and use $p(y_* = 1 | \mathbf{x}_*, \mathbf{X}, \mathbf{y}) \approx \frac{1}{M} \sum_{m=1}^M p(y_* = 1 | \mathbf{w}_m, \mathbf{x}_*) = \frac{1}{M} \sum_{m=1}^M \sigma(\mathbf{w}_m^\top \mathbf{x}_n)$

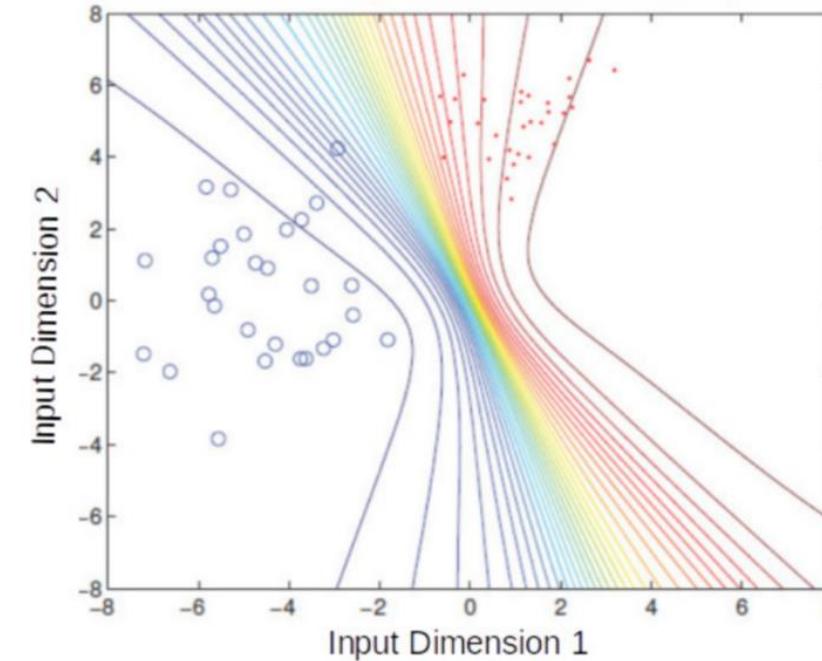


LR: Plug-in Prediction vs Posterior Averaging

Logistic Regression decision boundary
when using a point estimate of w



Logistic Regression decision boundary
when using posterior averaging



Posterior averaging is like using an ensemble of models. In this example, each model is a linear classifier but the ensemble-like effect resulted in nonlinear boundaries



Multiclass Logistic (a.k.a. Softmax) Regression

- Also called multinoulli/multinomial regression: Basically, LR for $K > 2$ classes
- In this case, $y_n \in \{1, 2, \dots, K\}$ and label probabilities are defined as

$$p(y_n = k | \mathbf{x}_n, \mathbf{W}) = \frac{\exp(\mathbf{w}_k^\top \mathbf{x}_n)}{\sum_{\ell=1}^K \exp(\mathbf{w}_\ell^\top \mathbf{x}_n)} = \mu_{nk}$$

Softmax function

Also note that $\sum_{\ell=1}^K \mu_{n\ell} = 1$ for any input \mathbf{x}_n



- K weight vecs $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K$ (one per class), each D -dim, and $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K]$
- Each likelihood $p(y_n | \mathbf{x}_n, \mathbf{W})$ is a multinoulli distribution. Therefore total likelihood

$$p(\mathbf{y} | \mathbf{X}, \mathbf{W}) = \prod_{n=1}^N \prod_{\ell=1}^K \mu_{n\ell}^{y_{n\ell}}$$

Notation: $y_{n\ell} = 1$ if true class of \mathbf{x}_n is ℓ and $y_{n\ell'} = 0 \forall \ell' \neq \ell$

- Can do MLE/MAP/fully Bayesian estimation for \mathbf{W} similar to LR model



Coming up next

- Generative models for supervised learning



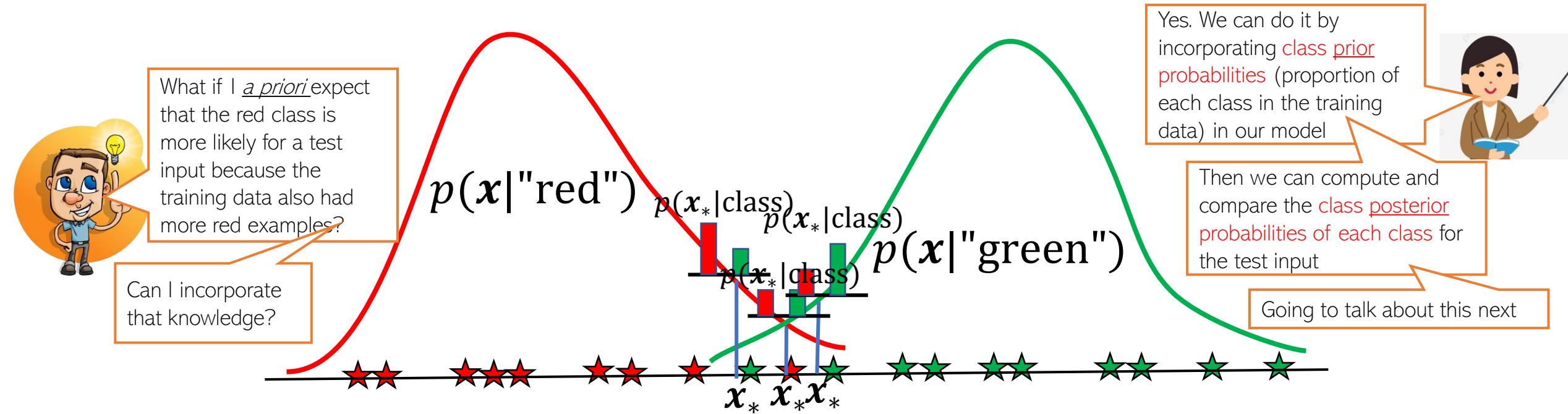
Probabilistic Models for Supervised Learning (3): Generative Classification and Regression

CS771: Introduction to Machine Learning

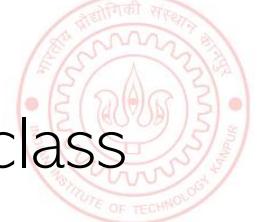
Piyush Rai

Generative Classification: A Basic Idea

- Learn the probability distribution of inputs from each class (“class-conditional”)



- Usually assume some form (e.g., Gaussian) and estimate the parameters of that distribution (using MLE/MAP/fully Bayesian approach)
- Predict label of a test input \mathbf{x}_* by comparing its probabilities under each class
 - Or can report the probability of belonging to each class (soft prediction)



Generative Classification: More Generally..

- Consider a classification problem with $K \geq 2$ classes
- The **class prior probability** of each class $k \in \{1, 2, \dots, K\}$ is $p(y = k)$
- Can use Bayes rule to compute **class posterior probability** for a test input \mathbf{x}_*

$$p(y_* = k | \mathbf{x}_*, \theta) = \frac{p(\mathbf{x}_*, y_* = k | \theta)}{p(\mathbf{x}_* | \theta)} = \frac{p(y_* = k | \theta)p(\mathbf{x}_* | y_* = k, \theta)}{p(\mathbf{x}_* | \theta)}$$

Class prior distribution for class k Class-conditional distribution of inputs from class k

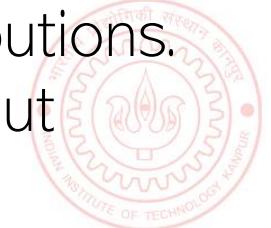
This is just the marginal distribution of the joint distribution in the numerator (summed over all K values of y_*)

θ collectively denotes the parameters the joint distribution of inputs and labels depends on

Setting $p(y_* = k | \theta) = 1/K$ will give us the approach that predicts by comparing the probabilities $p(\mathbf{x}_* | y_* = k, \theta)$ of \mathbf{x}_* under each of the classes



- We will first **estimate the parameters** of class prior and class-conditional distributions.
- Once estimated, we can use the above rule to predict the label for any test input
 - Can use MLE/MAP/fully Bayesian approach. We will only consider MLE/MAP here



Estimating Class Priors

Note: Can also do MAP estimation using a [Dirichlet prior](#) on π (this is akin to using Beta prior for doing MAP estimation for the bias of a coin). May try this as an exercise



- Estimating class priors $p(y = k)$ is usually straightforward in gen. classification
- Roughly speaking, it is the proportion of training examples from each class
 - Note: The above is true only when doing MLE (as we will see shortly)
 - If estimating class priors using MAP/fully Bayesian, they will be a “smooth version” of the proportions (because of the effect of regularization)
- The class prior distribution is assumed to be a [discrete distribution \(multinoulli\)](#)

$$p(y|\boldsymbol{\pi}) = \text{multinoulli}(y|\pi_1, \pi_2, \dots, \pi_K) = \prod_{k=1}^K \pi_k^{\mathbb{I}[y=k]}$$

$\pi_k = p(y = k)$

These probabilities sum to 1: $\sum_{k=1}^K \pi_k = 1$

Generalization of Bernoulli

- Given N i.i.d. labelled examples $\{(x_n, y_n)\}_{n=1}^N$, $y_n \in \{1, 2, \dots, K\}$ the MLE soln

$$\boldsymbol{\pi}_{MLE} = \operatorname{argmax}_{\boldsymbol{\pi}} \sum_{n=1}^N \log p(y_n|\boldsymbol{\pi})$$

Subject to constraint $\sum_{k=1}^K \pi_k = 1$

Can use [Lagrange based opt.](#) (note that we have an equality constraint)



Exercise: Verify that the MLE solution will be $p(y = k) = \pi_k = N_k/N$ where $N_k = \sum_{n=1}^N \mathbb{I}[y = k]$ (the frac. of class k examples)

Estimating Class-Conditionals

To be estimated using inputs from class k

- Can assume an appropriate distribution $p(\mathbf{x}|y = k, \theta)$ for inputs of each class
- If \mathbf{x} is D -dim, it will be a D -dim. distribution. Choice depends on various factors
 - Nature of input features, e.g.,
 - If $\mathbf{x} \in \mathbb{R}^D$, can use a D -dim Gaussian $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$
 - If $\mathbf{x} \in \{0,1\}^D$, can use D Bernoullis (one for each feature)
 - Can also choose more flexible/complex distributions if possible to estimate
 - Amount of training data available
 - With little data from a class, difficult to estimate the params of its class-cond. distribution
- Once decided the form of class-cond, estimate θ via MLE/MAP/Bayesian infer.
 - This essentially is a **density estimation** problem for the class-cond.
 - In principle, can use any density estimation method

Some workarounds: Use strong **regularization**, or a **simple form** of the class-conditional (e.g., use a spherical/diagonal rather than a full covariance if the class-cond is Gaussian), or assume features are independent given class ("naïve Bayes" assumption)

A big issue especially if the number of features (D) is very large



Gen. Classifn. using Gaussian Class-conditionals

- The generative classification model $p(y = k|x) = \frac{p(y=k)p(x|y=k)}{p(x|\theta)}$
- Assume each class-conditional $p(x|y = k)$ to be a Gaussian

$$\mathcal{N}(x|\mu_k, \Sigma_k) = \frac{1}{\sqrt{(2\pi)^D |\Sigma_k|}} \exp[-(x - \mu_k)^\top \Sigma_k^{-1} (x - \mu_k)]$$

A benefit of modeling each class by a distribution (recall that LwP had issues)



Since the Gaussian's covariance models its shape, we can learn the shape of each class ☺

- Class prior is multinoulli (we already saw): $p(y = k) = \pi_k, \pi_k \in (0,1), \sum_{k=1}^K \pi_k = 1$
- Let's denote the parameters of the model collectively by $\theta = \{\pi_k, \mu_k, \Sigma_k\}_{k=1}^K$
 - Can estimate these using MLE/MAP/Bayesian inference
 - Already saw the MLE solution for π : $\pi_k = N_k/N$ (can also do MAP)
 - MLE solution for $\mu_k = \frac{1}{N_k} \sum_{y_n=k} \mathbf{x}_n$, $\Sigma_k = \frac{1}{N_k} \sum_{y_n=k} (\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^\top$
- If using point est (MLE/MAP) for θ , predictive distribution will be

Can predict the most likely class for the test input \mathbf{x}_* by comparing these probabilities for all values of k

$$p(y_* = k|x_*, \theta) = \frac{\pi_k |\Sigma_k|^{-1/2} \exp\left[-\frac{1}{2}(\mathbf{x}_* - \mu_k)^\top \Sigma_k^{-1} (\mathbf{x}_* - \mu_k)\right]}{\sum_{k=1}^K \pi_k |\Sigma_k|^{-1/2} \exp\left[-\frac{1}{2}(\mathbf{x}_* - \mu_k)^\top \Sigma_k^{-1} (\mathbf{x}_* - \mu_k)\right]}$$

Can also do MAP estimation for μ_k, Σ_k using a Gaussian prior on μ_k and inverse Wishart prior on Σ_k

Exercise: Try to derive this. I will provide a separate note containing the derivation

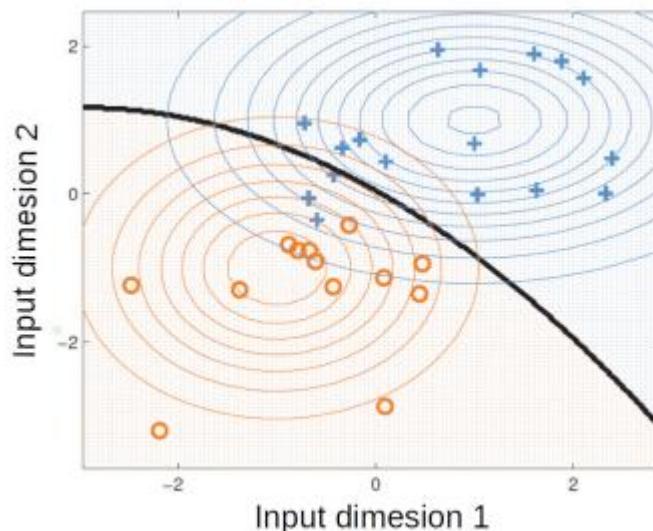
Note that the exponent has a Mahalanobis distance like term. Also, accounts for the fraction of training examples in class k

Decision Boundary with Gaussian Class-Conditional

- As we saw, the prediction rule when using Gaussian class-conditional

$$p(y = k|x, \theta) = \frac{\pi_k |\Sigma_k|^{-1/2} \exp\left[-\frac{1}{2}(x - \mu_k)^\top \Sigma_k^{-1}(x - \mu_k)\right]}{\sum_{k=1}^K \pi_k |\Sigma_k|^{-1/2} \exp\left[-\frac{1}{2}(x - \mu_k)^\top \Sigma_k^{-1}(x - \mu_k)\right]}$$

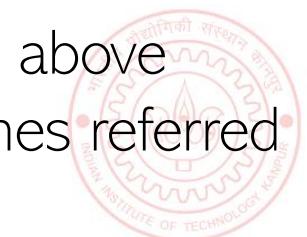
- The decision boundary between any pair of classes will be a **quadratic curve**



Reason: For any two classes k and k' at the decision boundary, we will have $p(y = k|x, \theta) = p(y = k'|x, \theta)$. Comparing **their logs** and ignoring terms that don't contain x , can easily see that

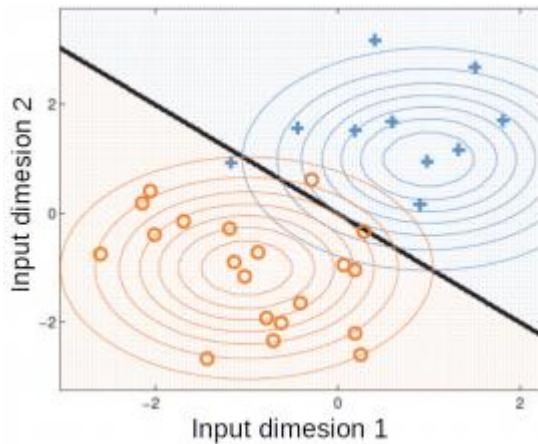
$$(x - \mu_k)^\top \Sigma_k^{-1}(x - \mu_k) - (x - \mu_{k'})^\top \Sigma_{k'}^{-1}(x - \mu_{k'}) = 0$$

Decision boundary contains all inputs x that satisfy the above. This is a **quadratic function** of x (this model is sometimes referred to **Quadratic Discriminant Analysis**)



Decision Boundary with Gaussian Class-Conditional

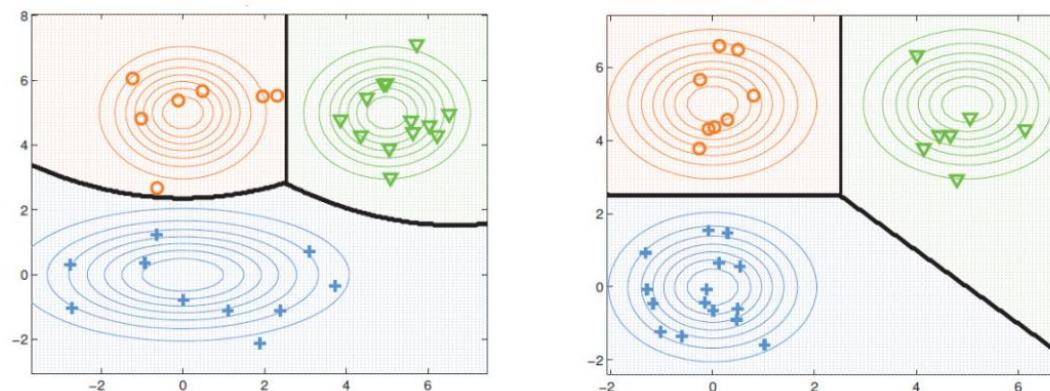
- Assume all classes are modeled using the same covariance matrix $\Sigma_k = \Sigma, \forall k$
- In this case, the decision boundary b/w any pair of classes will be **linear**



Reason: Again using $p(y = k|x, \theta) = p(y = k'|x, \theta)$, comparing their logs and ignoring terms that don't contain \mathbf{x} , we have

$$(\mathbf{x} - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) - (\mathbf{x} - \boldsymbol{\mu}_{k'})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}_{k'}) = 0$$

Quadratic terms of \mathbf{x} will cancel out; only linear terms will remain; hence decision boundary will be a linear function of \mathbf{x} (**Exercise:** Verify that we can indeed write the decision boundary between this pair of classes as $\mathbf{w}^T \mathbf{x} + b = 0$ where \mathbf{w} and b depend on $\boldsymbol{\mu}_k, \boldsymbol{\mu}_{k'}$ and $\boldsymbol{\Sigma}$)



If we assume the covariance matrices of the assumed Gaussian class-conditionals for any pair of classes to be equal, then the learned separation boundary b/w this pair of classes will be linear; otherwise, quadratic as shown in the figure on left



A Closer Look at the Linear Case

- For the linear case (when $\Sigma_k = \Sigma, \forall k$), the class posterior probability

$$p(y = k|x, \theta) \propto \pi_k \exp \left[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) \right]$$

- Expanding further, we can write the above as

$$p(y = k|x, \theta) \propto \exp \left[\boldsymbol{\mu}_k^\top \boldsymbol{\Sigma}^{-1} \mathbf{x} - \frac{1}{2} \boldsymbol{\mu}_k^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_k + \log \pi_k \right] \exp \left[\mathbf{x}^\top \boldsymbol{\Sigma}^{-1} \mathbf{x} \right]$$

- Therefore, the above class posterior probability can be written as

$$p(y = k|x, \theta) = \frac{\exp [\mathbf{w}_k^\top \mathbf{x} + b_k]}{\sum_{k=1}^K \exp [\mathbf{w}_k^\top \mathbf{x} + b_k]} \quad \mathbf{w}_k = \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_k \quad b_k = -\frac{1}{2} \boldsymbol{\mu}_k^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_k + \log \pi_k$$

If all Gaussians class-cond have the same covariance matrix (basically, of all classes are assumed to have the same shape)

- The above has *exactly* the same form as **softmax classification** (thus softmax is a special case of a generative classification model with Gaussian class-conditionals)



A Very Special Case: LwP Revisited

- Note the prediction rule when $\Sigma_k = \Sigma, \forall k$

$$\begin{aligned}\hat{y} = \arg \max_k p(y = k | \mathbf{x}) &= \arg \max_k \pi_k \exp \left[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) \right] \\ &= \arg \max_k \log \pi_k - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)\end{aligned}$$

- Also assume all classes to have equal no. of training examples, i.e., $\pi_k = 1/K$. Then

$$\hat{y} = \arg \min_k (\mathbf{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)$$

The Mahalanobis distance matrix = $\boldsymbol{\Sigma}^{-1}$

- Equivalent to assigning \mathbf{x} to the “closest” class in terms of a **Mahalanobis distance**
- If we further assume $\boldsymbol{\Sigma} = \mathbf{I}_D$ then the above is exactly the LwP rule



Generative Classification: Some Comments

- A simple but powerful approach to probabilistic classification
- Especially easy to learn if class-conditionals are simple
 - E.g., Gaussian with diagonal covariances \Rightarrow Gaussian naïve Bayes
 - Another popular model is multinomial naïve Bayes (widely used for document classification)
 - The naïve Bayes assumption: features are conditional independent given class label

$$p(\mathbf{x}|y = k) = \prod_{d=1}^D p(x_d|y = k)$$

Benefit: Instead of estimating a D -dim distribution which may be hard (if we don't have enough data), we will estimate D one-dim distributions (much simpler task)

- Can choose the form of class-conditionals $p(\mathbf{x}|y = k)$ based on the type of inputs \mathbf{x}
 - Will see such methods later
- Can handle missing data (e.g., if some part of the input \mathbf{x} is missing) or missing labels
 - Will see such methods later
- Generative models are also useful for unsup. and semi-sup. learning



Generative Models for Regression

- Yes, we can even model regression problems using a generative approach
- Note that the output y is not longer discrete (so no notion of a class-conditional)
- However, the basic rule of recovering a **conditional** from **joint** would still apply

$$p(y|\mathbf{x}, \theta) = \frac{p(\mathbf{x}, y|\theta)}{p(\mathbf{x}|\theta)}$$

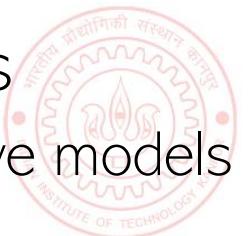
- Thus we can model the joint distribution $p(\mathbf{x}, y|\theta)$ of features \mathbf{x} and outputs $y \in \mathbb{R}$
 - If features are real-valued the we can model $p(\mathbf{x}, y|\theta)$ using a $(D + 1)$ -dim Gaussian
 - From this $(D + 1)$ -dim Gaussian, we can get $p(y|\mathbf{x}, \theta)$ using Gaussian conditioning formula
 - If joint is Gaussian, any subset of variables (y here), given the rest (\mathbf{x} here) is also a Gaussian!
 - Refer to the Gaussian results from maths refresher slides for the result





Discriminative vs Generative

- Recall that discriminative approaches model $p(y|x)$ directly
- Generative approaches model $p(y|x)$ via $p(x,y)$
- **Number of parameters:** Discriminative models have fewer parameters to be learned
 - Just the weight vector/matrix w/W in case of logistic/softmax classification
- **Ease of parameter estimation:** Debatable as to which one is easier
 - For “simple” class-conditionals, easier for gen. classifn model (often closed-form solution)
 - Parameter estimation for discriminative models (logistic/softmax) usually requires iterative methods (although objective functions usually have global optima)
- **Dealing with missing features:** Generative models can handle this easily
 - E.g., by integrating out the missing features while estimating the parameters)
- **Inputs with features having mixed types:** Generative model can handle this
 - Appropriate $p(x_d|y)$ for each type of feature in the input. Difficult for discriminative models



Discriminative vs Generative (Contd)

- **Leveraging unlabeled data:** Generative models can handle this easily by treating the missing labels as **latent variables** and are ideal for **Semi-supervised Learning**. Discriminative models can't do it easily
- **Adding data from new classes:** Discriminative model will need to be re-trained on all classes all over again. Generative model will just require estimating the class-cond of newly added classes
- **Have lots of labeled training data?** Discriminative models usually work very well
- **Final Verdict?** Despite generative classification having some clear advantages, both methods can be quite powerful (the actual choice may be dictated by the problem)
 - Important to be aware of their strengths/weaknesses, and also the connections between these
- **Possibility of a Hybrid Design?** Yes, Generative and Disc. models can be combined, e.g.,
 - “Principled Hybrids of Generative and Discriminative Models” (Lassere et al, 2006)
 - “Deep Hybrid Models: Bridging Discriminative & Generative Approaches” (Kuleshov & Ermon, 2017)



Coming up next

- Large-margin hyperplane based classifiers (support vector machines)
- Kernel methods for learning nonlinear models



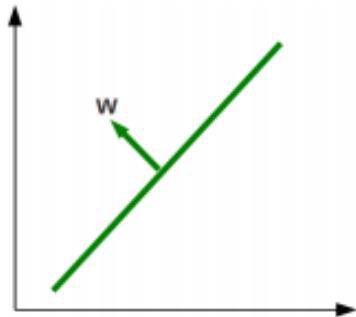
Hyperplane based Classifiers (1): The Perceptron Algorithm

CS771: Introduction to Machine Learning

Piyush Rai

Hyperplane

- Separates a D -dimensional space into two **half-spaces** (positive and negative)
- Defined by a normal vector $\mathbf{w} \in \mathbb{R}^D$ (pointing towards positive half-space)



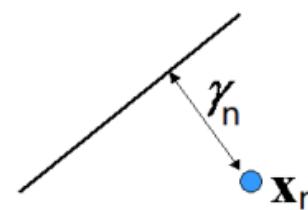
$b > 0$ means moving $\mathbf{w}^\top \mathbf{x} = 0$ along the direction of \mathbf{w} ; $b < 0$ means in opp. dir.

$$\mathbf{w}^\top \mathbf{x} + b = 0$$

- Equation of the hyperplane: $\mathbf{w}^\top \mathbf{x} = 0$
- Assumption: The hyperplane passes through origin. If not, add a bias term b
- Distance of a point \mathbf{x}_n from a hyperplane $\mathbf{w}^\top \mathbf{x} + b = 0$

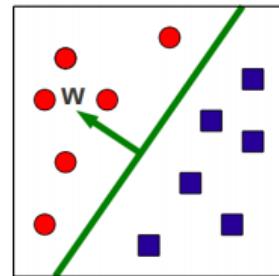
Can be positive or negative

$$\gamma_n = \frac{\mathbf{w}^\top \mathbf{x}_n + b}{\|\mathbf{w}\|}$$



Hyperplane based (binary) classification

- Basic idea: Learn to separate two classes by a hyperplane $\mathbf{w}^T \mathbf{x} + b = 0$



Prediction Rule

$$y_* = \text{sign}(\mathbf{w}^T \mathbf{x}_* + b)$$

For multi-class classification with hyperplanes, there will be multiple hyperplanes (e.g., one for each pair of classes); more on this later



- The hyperplane may be “implied” by the model, or learned directly
 - Implied: Prototype-based classification, nearest neighbors, generative classification, etc
 - Directly learned: Logistic regression, Perceptron, Support Vector Machine (SVM), etc
- The “direct” approach defines a model with params \mathbf{w} (and optionally a bias param b)
 - The parameters are learned by optimizing a classification loss function (will soon see examples)
 - These are also discriminative approaches – \mathbf{x} is not modeled but treated as fixed (given)
- The hyperplane need not be linear (e.g., can be made nonlinear using kernels; later)



Interlude: Loss Functions for Classification

- In regression (assuming linear model $\hat{y} = \mathbf{w}^\top \mathbf{x}$), some common loss fn

$$\ell(y, \hat{y}) = (y - \hat{y})^2$$

$$\ell(y, \hat{y}) = |y - \hat{y}|$$

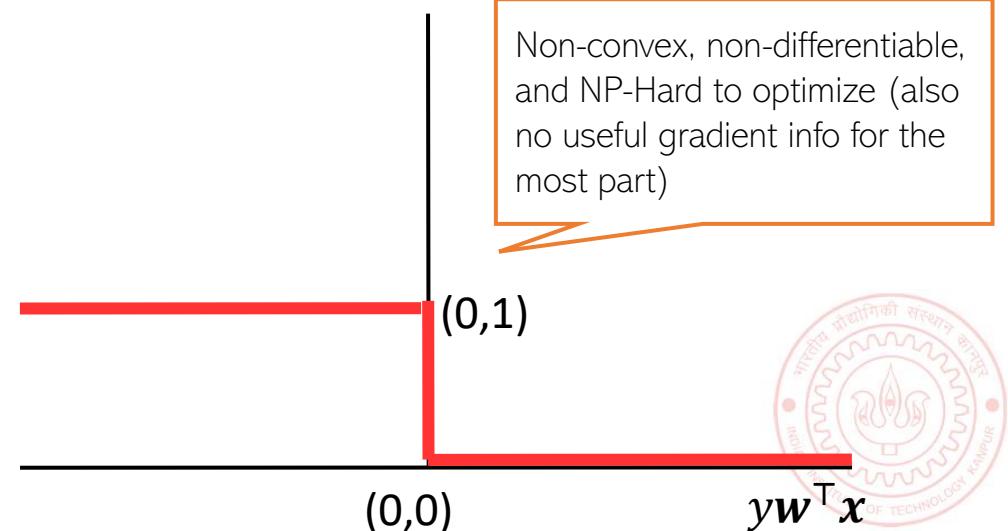
- These measure the difference between the true output and model's prediction
- What about loss functions for classification where $\hat{y} = \text{sign}(\mathbf{w}^\top \mathbf{x})$?
- Perhaps the most natural classification loss function would be a “0-1 Loss”

- Loss = 1 if $\hat{y} \neq y$ and Loss = 0 if $\hat{y} = y$.
- Assuming labels as +1/-1, it means

$$\ell(y, \hat{y}) = \begin{cases} 1 & \text{if } y\mathbf{w}^\top \mathbf{x} < 0 \\ 0 & \text{if } y\mathbf{w}^\top \mathbf{x} \geq 0 \end{cases}$$

Same as $\mathbb{I}[y\mathbf{w}^\top \mathbf{x} < 0]$ or $\mathbb{I}[\text{sign}(\mathbf{w}^\top \mathbf{x}) \neq y]$

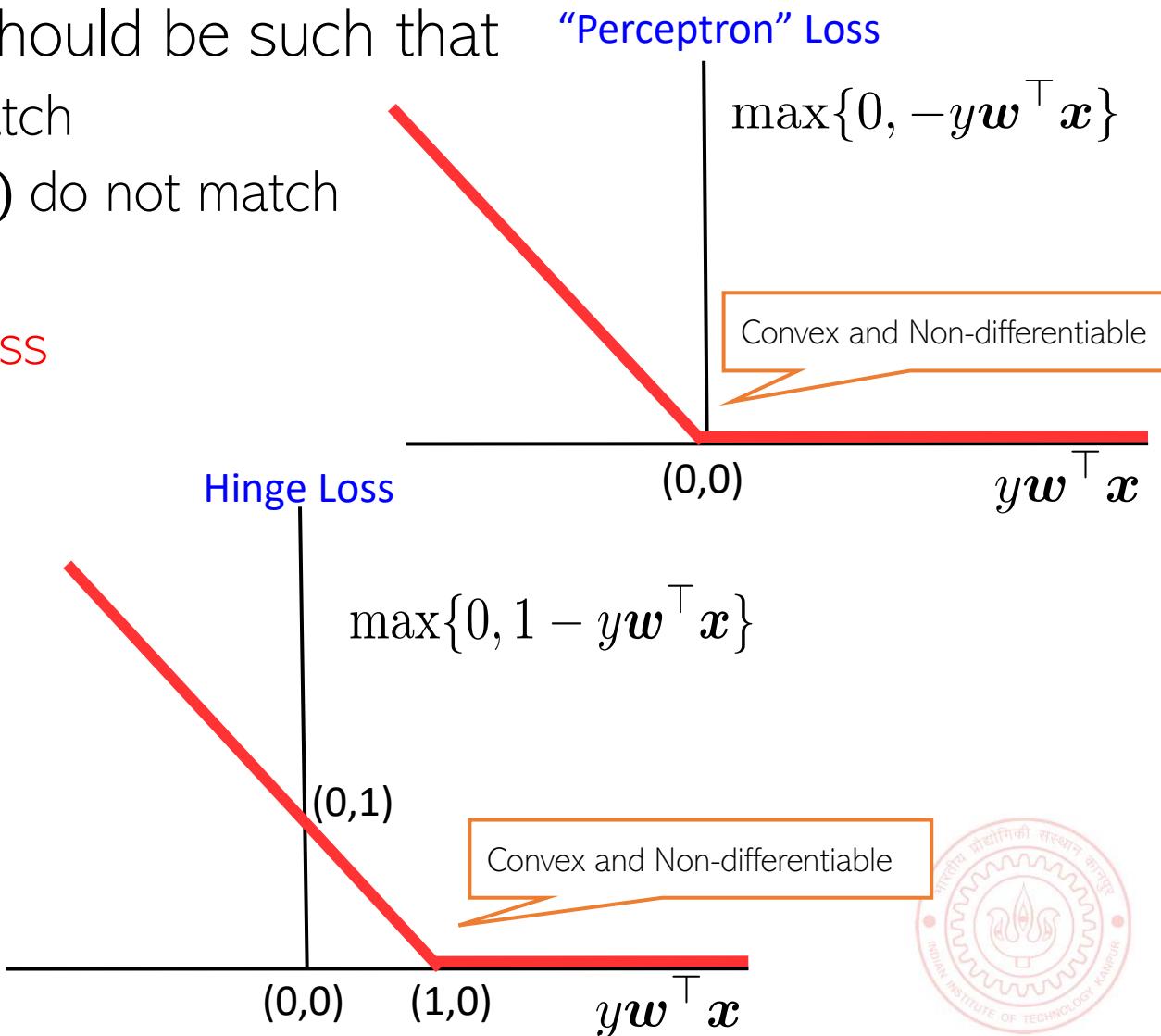
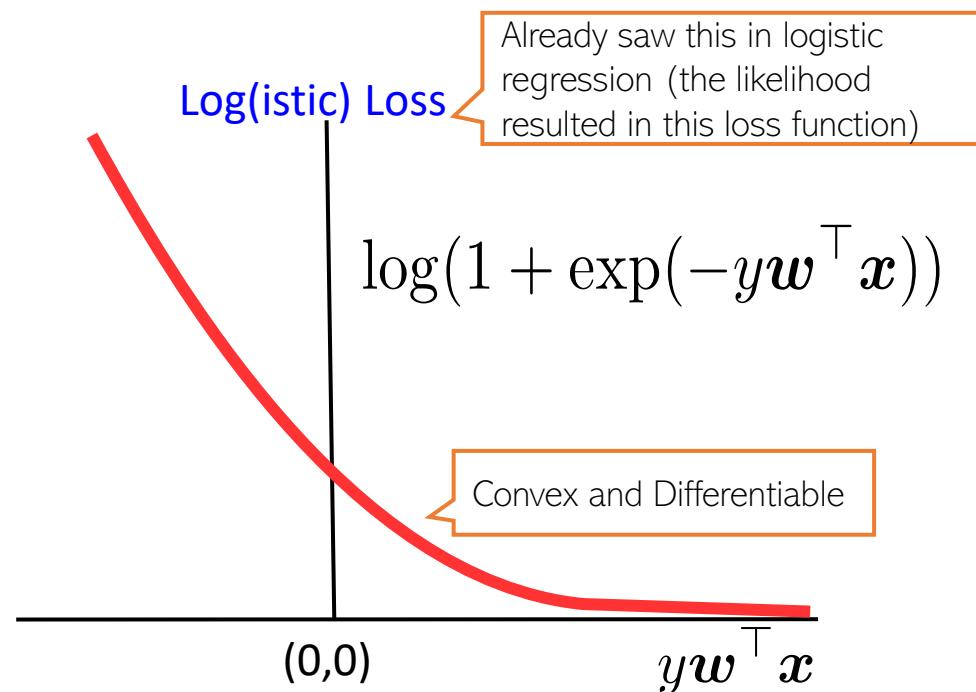
0-1 Loss



Interlude: Loss Functions for Classification

- An ideal loss function for classification should be such that

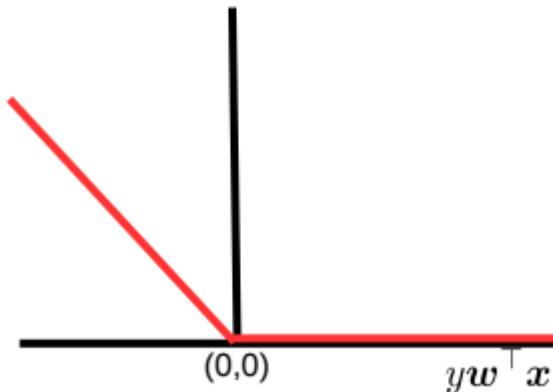
- Loss is small/zero if y and $\text{sign}(\mathbf{w}^\top \mathbf{x})$ match
- Loss is large/non-zero if y and $\text{sign}(\mathbf{w}^\top \mathbf{x})$ do not match
- Large positive $y\mathbf{w}^\top \mathbf{x} \Rightarrow$ small/zero loss
- Large negative $y\mathbf{w}^\top \mathbf{x} \Rightarrow$ large/non-zero loss



Learning by Optimizing Perceptron Loss

- Let's ignore the bias term \mathbf{b} for now. So the hyperplane is simply $\mathbf{w}^\top \mathbf{x} = 0$
- The Perceptron loss function: $L(\mathbf{w}) = \sum_{n=1}^N \max\{0, -y_n \mathbf{w}^\top \mathbf{x}_n\}$. Let's do SGD

"Perceptron" Loss: $\max\{0, -y\mathbf{w}^\top \mathbf{x}\}$

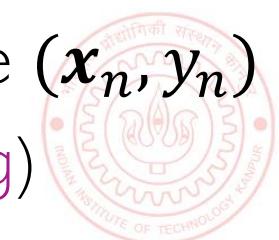


Subgradients w.r.t. \mathbf{w}

$$\mathbf{g}_n = \begin{cases} 0, & \text{for } y_n \mathbf{w}^\top \mathbf{x}_n > 0 \\ -y_n \mathbf{x}_n, & \text{for } y_n \mathbf{w}^\top \mathbf{x}_n < 0 \\ k y_n \mathbf{x}_n, & \text{for } y_n \mathbf{w}^\top \mathbf{x}_n = 0 \quad (\text{where } k \in [-1, 0]) \end{cases}$$

One randomly chosen example in each iteration

- If we use $k = 0$ then $\mathbf{g}_n = 0$ for $y_n \mathbf{w}^\top \mathbf{x}_n \geq 0$, and $\mathbf{g}_n = -y_n \mathbf{x}_n$ for $y_n \mathbf{w}^\top \mathbf{x}_n < 0$
- Non-zero gradients only when the model makes a mistake on current example (\mathbf{x}_n, y_n)
- Thus SGD will update \mathbf{w} only when there is a mistake (mistake-driven learning)



The Perceptron Algorithm

- Stochastic Sub-grad desc on Perceptron loss is also known as the Perceptron algorithm

Stochastic SubGD

Note: An example may get chosen several times during the entire run

① Initialize $\mathbf{w} = \mathbf{w}^{(0)}$, $t = 0$, set $\eta_t = 1, \forall t$

② Pick some (\mathbf{x}_n, y_n) randomly.

③ If current \mathbf{w} makes a **mistake** on (\mathbf{x}_n, y_n) , i.e., $y_n \mathbf{w}^{(t)^\top} \mathbf{x}_n < 0$

$$\begin{aligned}\mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + y_n \mathbf{x}_n \\ t &= t + 1\end{aligned}$$

④ If not converged, go to step 2.

Mistake condition

Updates are “corrective”: If $y_n = +1$ and $\mathbf{w}^\top \mathbf{x}_n < 0$, after the update $\mathbf{w}^\top \mathbf{x}_n$ will be less negative. Likewise, if $y_n = -1$ and $\mathbf{w}^\top \mathbf{x}_n > 0$, after the update $\mathbf{w}^\top \mathbf{x}_n$ will be less positive



If training data is linearly separable, the Perceptron algo will converge in a finite number of iterations (Block & Novikoff theorem)

- An example of an **online learning** algorithm (processes one training ex. at a time)

- Assuming $\mathbf{w}^{(0)} = 0$, easy to see that the final \mathbf{w} has the form $\mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n$

■ α_n is total number of mistakes made by the algorithm on example (\mathbf{x}_n, y_n)

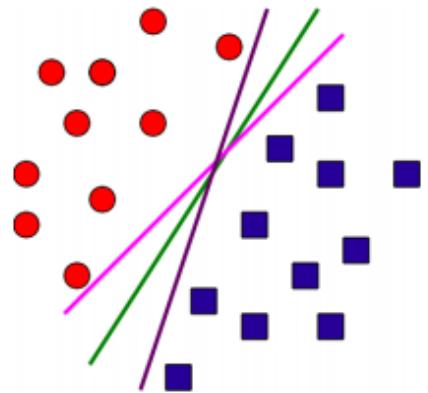
■ As we'll see, many other models also have weights \mathbf{w} in the form $\mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n$

Meaning of α_n may be different



Perceptron and (lack of) Margins

- Perceptron would learn a hyperplane (of many possible) that separates the classes



Basically, it will learn the hyperplane which corresponds to the \mathbf{w} that minimizes the Perceptron loss

- Doesn't guarantee any "margin" around the hyperplane

- The hyperplane can get arbitrarily close to some training example(s) on either side
- This may not be good for generalization performance

Kind of an "unsafe" situation to have
– ideally would like it to be reasonably away from closest training examples from either class

$\gamma > 0$ is some pre-specified margin

- Can artificially introduce margin by changing the mistake condition to $y_n \mathbf{w}^T \mathbf{x}_n < \gamma$
- Support Vector Machine (SVM) does it directly by learning the max. margin hyperplane



Coming up next

- Support Vector Machines



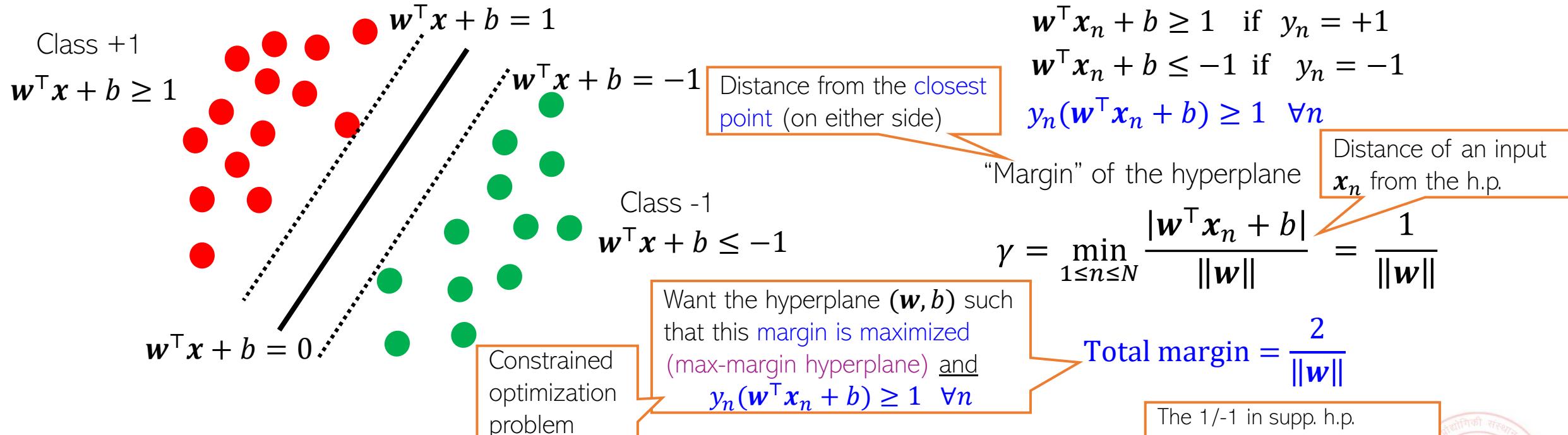
Hyperplane based Classifiers (2): Large-Margin Classification - SVM

CS771: Introduction to Machine Learning

Piyush Rai

Support Vector Machine (SVM)

- Hyperplane based classifier. Ensures a large margin around the hyperplane
- Will assume a linear hyperplane to be of the form $\mathbf{w}^T \mathbf{x} + b = 0$ (nonlinear ext. later)



- Two other “supporting” hyperplanes defining a “no man’s land”
 - Ensure that zero training examples fall in this region (will relax later)
 - The SVM idea: Position the hyperplane s.t. this region is as “wide” as possible

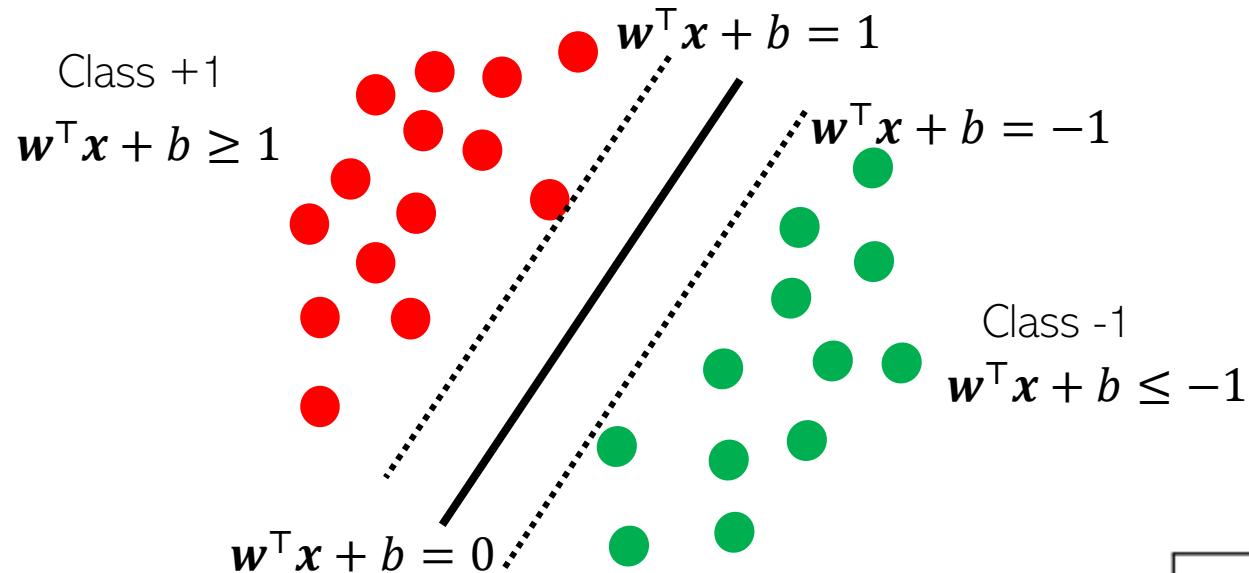
SVM originally proposed by Vapnik and colleagues in early 90s

The 1/-1 in supp. h.p. equations is arbitrary; can replace by any scalar m/-m and solution won’t change, except a simple scaling of \mathbf{w}



Hard-Margin SVM

- Hard-Margin: Every training example must fulfil margin condition $y_n(\mathbf{w}^\top \mathbf{x}_n + b) \geq 1$
- Meaning: Must not have any example in the no-man's land

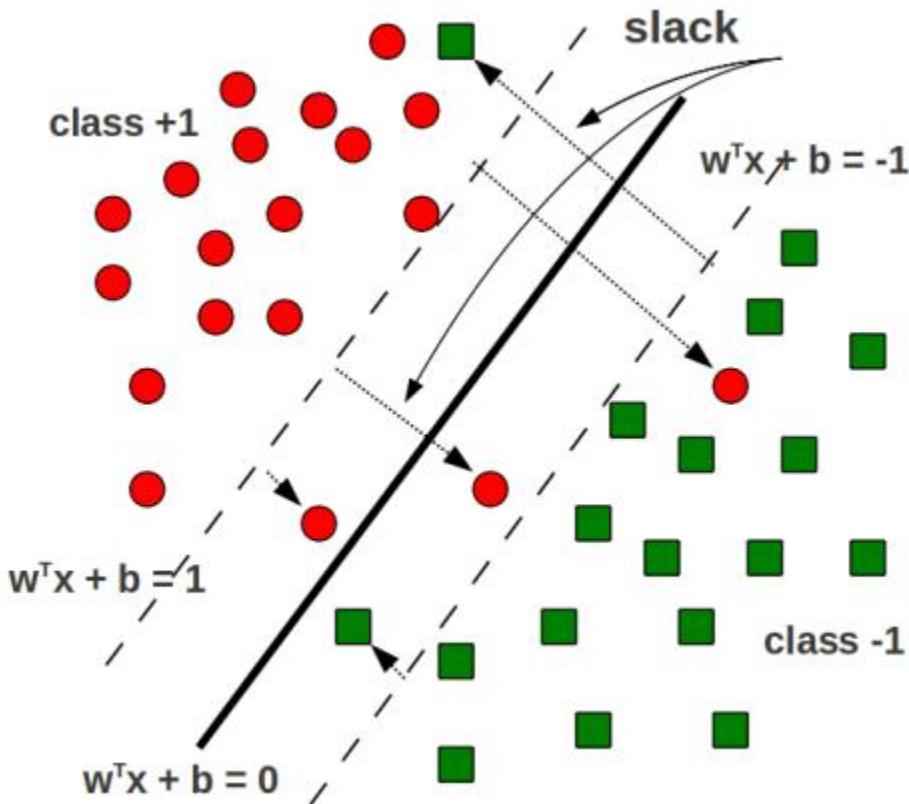


Constrained optimization problem with N inequality constraints. Objective and constraints both are convex

- Also want to maximize margin $2\gamma = \frac{2}{\|\mathbf{w}\|}$
- Equivalent to minimizing $\|\mathbf{w}\|^2$ or $\frac{\|\mathbf{w}\|^2}{2}$
- The objective func. for hard-margin SVM

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & f(\mathbf{w}, b) = \frac{\|\mathbf{w}\|^2}{2} \\ \text{subject to} \quad & y_n(\mathbf{w}^\top \mathbf{x}_n + b) \geq 1, \quad n = 1, \dots, N \end{aligned}$$

Soft-Margin SVM (More Commonly Used)



Soft-margin constraint:

- Allow some training examples to fall within the no-man's land (margin region)
- Even okay for some training examples to fall totally on the wrong side of h.p.
- Extent of “violation” by a training input (\mathbf{x}_n, y_n) is known as **slack** $\xi_n \geq 0$
- $\xi_n > 1$ means totally on the wrong side

$$\mathbf{w}^\top \mathbf{x}_n + b \geq 1 - \xi_n \quad \text{if } y_n = +1$$

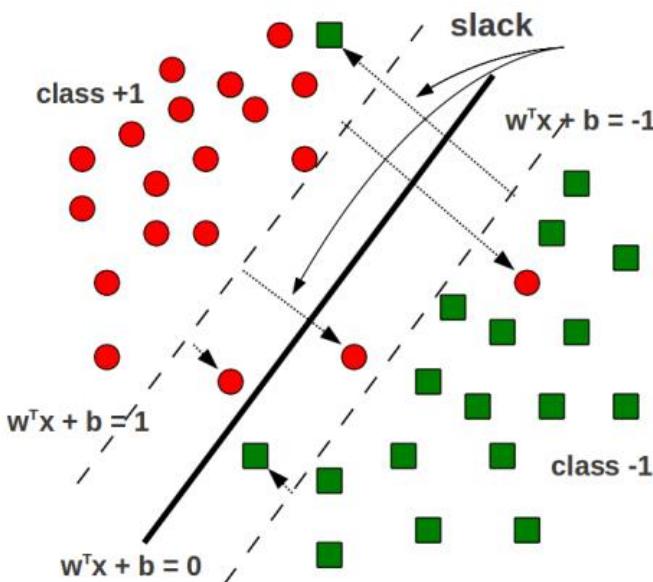
$$\mathbf{w}^\top \mathbf{x}_n + b \leq -1 + \xi_n \quad \text{if } y_n = -1$$

$$y_n(\mathbf{w}^\top \mathbf{x}_n + b) \geq 1 - \xi_n \quad \forall n$$



Soft-Margin SVM (Contd)

- Goal: Still want to maximize the margin such that
 - Soft-margin constraints $y_n(\mathbf{w}^\top \mathbf{x}_n + b) \geq 1 - \xi_n$ are satisfied for all training ex.
 - Do not have too many margin violations (sum of slacks $\sum_{n=1}^N \xi_n$ should be small)



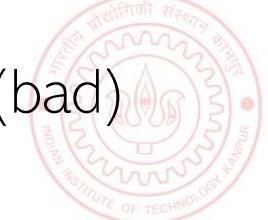
- The objective func. for soft-margin SVM

$$\begin{aligned} & \min_{\mathbf{w}, b, \xi} f(\mathbf{w}, b, \xi) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n \\ & \text{subject to } y_n(\mathbf{w}^\top \mathbf{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0 \end{aligned}$$

Inversely prop.
to margin Trade-off hyperparam training
error Constrained optimization
problem with $2N$ inequality
constraints. Objective and
constraints both are convex

$n = 1, \dots, N$

- Hyperparameter C controls the trade off between large margin and small training error (need to tune)
 - Large C : small training error but also small margin (bad)
 - Small C : large margin but large training error (bad)



Solving the SVM Problem



Solving Hard-Margin SVM

- The hard-margin SVM optimization problem is

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & f(\mathbf{w}, b) = \frac{\|\mathbf{w}\|^2}{2} \\ \text{subject to} \quad & 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \quad n = 1, \dots, N \end{aligned}$$

- A constrained optimization problem. One option is to solve using Lagrange's method
- Introduce Lagrange multipliers α_n ($n = 1, \dots, N$), one for each constraint, and solve

$$\min_{\mathbf{w}, b} \max_{\alpha \geq 0} \mathcal{L}(\mathbf{w}, b, \alpha) = \frac{\|\mathbf{w}\|^2}{2} + \sum_{n=1}^N \alpha_n \{1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)\}$$

- $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_N]$ denotes the vector of Lagrange multipliers
- It is easier (and helpful; we will soon see why) to solve the dual: min and then max



Solving Hard-Margin SVM

- The dual problem (min then max) is

$$\max_{\alpha \geq 0} \min_{w, b} \mathcal{L}(w, b, \alpha) = \frac{w^T w}{2} + \sum_{n=1}^N \alpha_n \{1 - y_n(w^T x_n + b)\}$$

Note: if we ignore the bias term b then we don't need to handle the constraint $\sum_{n=1}^N \alpha_n y_n = 0$
(problem becomes a bit more easy to solve)



Otherwise, the α_n 's are coupled and some opt. techniques such as coordinate ascent can't easily be applied

- Take (partial) derivatives of \mathcal{L} w.r.t. w and b and setting them to zero gives (verify)

$$\frac{\partial \mathcal{L}}{\partial w} = 0 \Rightarrow w = \sum_{n=1}^N \alpha_n y_n x_n$$

$$\frac{\partial \mathcal{L}}{\partial b} = 0 \Rightarrow \sum_{n=1}^N \alpha_n y_n = 0$$

α_n tells us how important training example (x_n, y_n) is

- The solution w is simply a weighted sum of all the training inputs

- Substituting $w = \sum_{n=1}^N \alpha_n y_n x_n$ in the Lagrangian, we get the dual problem as (verify)

This is also a "quadratic program" (QP) – a quadratic function of the variables α

$$\max_{\alpha \geq 0} \mathcal{L}_D(\alpha) = \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{m,n=1}^N \alpha_m \alpha_n y_m y_n (x_m^T x_n)$$

Note that inputs appear only as pairwise dot products. This will be useful later on when we make SVM nonlinear using kernel methods

Maximizing a concave function
(or minimizing a convex function)
s.t. $\alpha \geq 0$ and $\sum_{n=1}^N \alpha_n y_n = 0$.
Many methods to solve it.

$$\max_{\alpha \geq 0} \mathcal{L}_D(\alpha) = \alpha^T \mathbf{1} - \frac{1}{2} \alpha^T G \alpha$$

G is an $N \times N$ p.s.d. matrix, also called the Gram Matrix, $G_{nm} = y_n y_m x_n^T x_m$, and $\mathbf{1}$ is a vector of all 1s



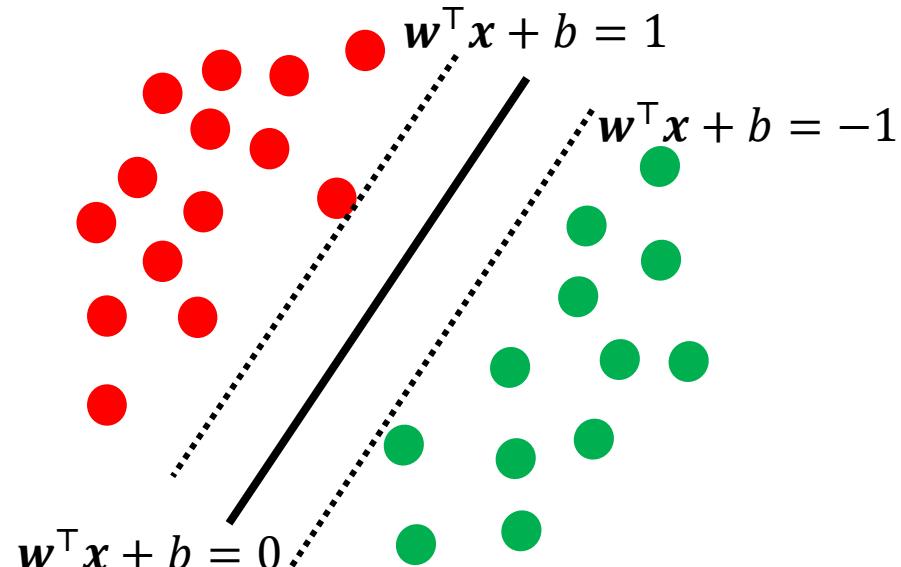
Solving Hard-Margin SVM

- Once we have the α_n 's by solving the dual, we can get \mathbf{w} and b as

$$\mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n \quad (\text{we already saw this})$$

$$b = -\frac{1}{2} (\min_{n: y_n=+1} \mathbf{w}^T \mathbf{x}_n + \max_{n: y_n=-1} \mathbf{w}^T \mathbf{x}_n) \quad (\text{exercise})$$

- A nice property: Most α_n 's in the solution will be zero (sparse solution)



- Reason: KKT conditions
- For the optimal α_n 's, we must have
- Thus α_n nonzero only if $y_n(\mathbf{w}^T \mathbf{x}_n + b) = 1$, i.e., the training example lies on the boundary

$$\alpha_n \{1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)\} = 0$$

- These examples are called support vectors



Solving Soft-Margin SVM

- Recall the soft-margin SVM optimization problem

$$\begin{aligned} \min_{\mathbf{w}, b, \xi} \quad & f(\mathbf{w}, b, \xi) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n \\ \text{subject to} \quad & 1 \leq y_n(\mathbf{w}^T \mathbf{x}_n + b) + \xi_n, \quad -\xi_n \leq 0 \quad n = 1, \dots, N \end{aligned}$$

- Here $\xi = [\xi_1, \xi_2, \dots, \xi_N]$ is the vector of **slack variables**
- Introduce Lagrange multipliers α_n, β_n for each constraint and solve Lagrangian

$$\min_{\mathbf{w}, b, \xi} \max_{\alpha \geq 0, \beta \geq 0} \mathcal{L}(\mathbf{w}, b, \xi, \alpha, \beta) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n + \sum_{n=1}^N \alpha_n \{1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) - \xi_n\} - \sum_{n=1}^N \beta_n \xi_n$$

- The terms in red color above were not present in the hard-margin SVM
- Two set of dual variables $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_N]$ and $\boldsymbol{\beta} = [\beta_1, \beta_2, \dots, \beta_N]$
- Will eliminate the primal var \mathbf{w}, b, ξ to get dual problem containing the dual variables





Solving Soft-Margin SVM

- The Lagrangian problem to solve

Note: if we ignore the bias term b then we don't need to handle the constraint $\sum_{n=1}^N \alpha_n y_n = 0$ (problem becomes a bit more easy to solve)

Otherwise, the α_n 's are coupled and some opt. techniques such as co-ordinate aspect can't easily applied

$$\min_{\mathbf{w}, b, \xi} \max_{\alpha \geq 0, \beta \geq 0} \mathcal{L}(\mathbf{w}, b, \xi, \alpha, \beta) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n + \sum_{n=1}^N \alpha_n \{1 - y_n(\mathbf{w}^\top \mathbf{x}_n + b) - \xi_n\} - \sum_{n=1}^N \beta_n \xi_n$$

- Take (partial) derivatives of \mathcal{L} w.r.t. \mathbf{w} , b , and ξ_n and setting to zero gives

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n, \quad \text{Weighted sum of training inputs}$$

$$\frac{\partial \mathcal{L}}{\partial b} = 0 \Rightarrow \sum_{n=1}^N \alpha_n y_n = 0, \quad \frac{\partial \mathcal{L}}{\partial \xi_n} = 0 \Rightarrow C - \alpha_n - \beta_n = 0$$

- Using $C - \alpha_n - \beta_n = 0$ and $\beta_n \geq 0$, we have $\alpha_n \leq C$ (for hard-margin, $\alpha_n \geq 0$)
- Substituting these in the Lagrangian \mathcal{L} gives the Dual problem

Given α , \mathbf{w} and b can be found just like the hard-margin SVM case

$$\max_{\alpha \leq C, \beta \geq 0} \mathcal{L}_D(\alpha, \beta) = \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{m,n=1}^N \alpha_m \alpha_n y_m y_n (\mathbf{x}_m^\top \mathbf{x}_n) \quad \text{s.t.} \quad \sum_{n=1}^N \alpha_n y_n = 0$$

The dual variables β don't appear in the dual problem!

Maximizing a concave function (or minimizing a convex function)
s.t. $\alpha \leq C$ and $\sum_{n=1}^N \alpha_n y_n = 0$.

Many methods to solve it.

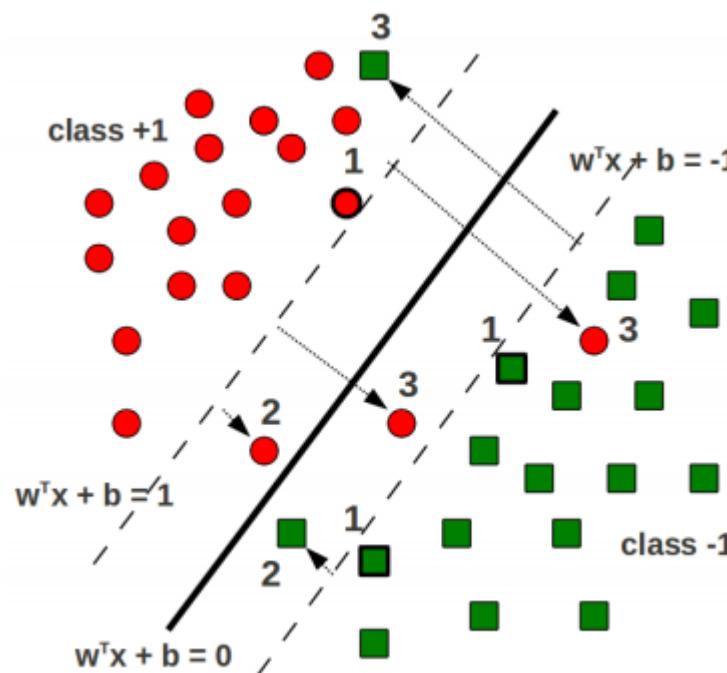
$$\max_{\alpha \leq C} \mathcal{L}_D(\alpha) = \alpha^\top \mathbf{1} - \frac{1}{2} \alpha^\top \mathbf{G} \alpha$$

In the solution, α will still be sparse just like the hard-margin SVM case. Nonzero α_n correspond to the support vectors



Support Vectors in Soft-Margin SVM

- The hard-margin SVM solution had only one type of support vectors
 - All lied on the supporting hyperplanes $\mathbf{w}^T \mathbf{x}_n + b = 1$ and $\mathbf{w}^T \mathbf{x}_n + b = -1$
- The soft-margin SVM solution has three types of support vectors (with nonzero α_n)



1. Lying on the supporting hyperplanes
2. Lying within the margin region but still on the correct side of the hyperplane
3. Lying on the wrong side of the hyperplane (misclassified training examples)



SVMs via Dual Formulation: Some Comments

- Recall the final dual objectives for hard-margin and soft-margin SVM

Hard-Margin SVM: $\max_{\alpha \geq 0} \mathcal{L}_D(\alpha) = \alpha^\top \mathbf{1} - \frac{1}{2} \alpha^\top \mathbf{G} \alpha$

Note: Both these ignore the bias term b otherwise will need another constraint $\sum_{n=1}^N \alpha_n y_n = 0$

Soft-Margin SVM: $\max_{\alpha \leq C} \mathcal{L}_D(\alpha) = \alpha^\top \mathbf{1} - \frac{1}{2} \alpha^\top \mathbf{G} \alpha$

- The dual formulation is nice due to two primary reasons
 - Allows conveniently handling the margin based constraint (via Lagrangians)
 - Allows learning nonlinear separators by replacing inner products in $G_{nm} = y_n y_m \mathbf{x}_n^\top \mathbf{x}_m$ by general kernel-based similarities (more on this when we talk about kernels)
- However, dual formulation can be expensive if N is large (esp. compared to D)
 - Need to solve for N variables $\alpha = [\alpha_1, \alpha_2, \dots, \alpha_N]$
 - Need to pre-compute and store $N \times N$ gram matrix \mathbf{G}
- Lot of work on speeding up SVM in these settings (e.g., can use co-ord. descent for α)



Solving for SVM in the Primal

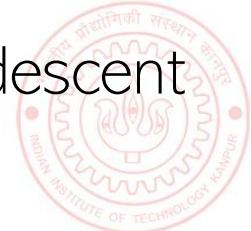
- Maximizing margin subject to constraints led to the soft-margin formulation of SVM

$$\begin{aligned} & \arg \min_{\mathbf{w}, b, \xi} \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n \\ \text{subject to } & y_n(\mathbf{w}^\top \mathbf{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0 \quad n = 1, \dots, N \end{aligned}$$

- Note that slack ξ_n is the same as $\max\{0, 1 - y_n(\mathbf{w}^\top \mathbf{x}_n + b)\}$, i.e., hinge loss for (\mathbf{x}_n, y_n)
- Thus the above is equivalent to minimizing the ℓ_2 regularized hinge loss

$$\mathcal{L}(\mathbf{w}, b) = \sum_{n=1}^N \max\{0, 1 - y_n(\mathbf{w}^\top \mathbf{x}_n + b)\} + \frac{\lambda}{2} \mathbf{w}^\top \mathbf{w}$$

- Sum of slacks is like sum of hinge losses, C and λ play similar roles
- Can learn (\mathbf{w}, b) directly by minimizing $\mathcal{L}(\mathbf{w}, b)$ using (stochastic) (sub)grad. descent
 - Hinge-loss version preferred for linear SVMs, or with other regularizers on \mathbf{w} (e.g., ℓ_1)



SVM: Summary

- A hugely (perhaps the most!) popular classification algorithm
- Reasonably mature, highly optimized SVM softwares freely available (perhaps the reason why it is more popular than various other competing algorithms)
- Some popular ones: libSVM, LIBLINEAR, sklearn also provides SVM
- Lots of work on scaling up SVMs* (both large N and large D)
- Extensions beyond binary classification (e.g., multiclass, structured outputs)
- Can even be used for regression problems (Support Vector Regression)
- Nonlinear extensions possible via kernels

* See: "Support Vector Machine Solvers" by Bottou and Lin



Coming up next

- A co-ordinate ascent algorithm for solving the SVM dual
- Multi-class SVM
- One-class SVM
- Kernel methods and nonlinear SVM via kernels



Hyperplane based Classifiers (3): SVM – Some Extensions

CS771: Introduction to Machine Learning

Piyush Rai

Plan

- A co-ordinate ascent based optimization algo for SVM
- Some extensions of binary SVM
 - Multi-class classification using SVM
 - One-class classification (a.k.a. novelty/outlier detection) SVM



A Co-ordinate Ascent Algorithm for SVM

- Recall the dual objective of soft-margin SVM (assuming no bias \mathbf{b})

$$\underset{\mathbf{0} \leq \boldsymbol{\alpha} \leq C}{\operatorname{argmax}} \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{m,n=1}^N \alpha_m \alpha_n y_m y_n \mathbf{x}_m^\top \mathbf{x}_n$$

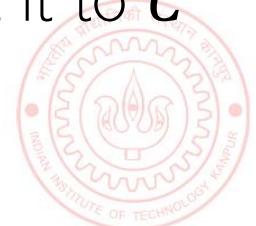
- Focusing on just one of the components of $\boldsymbol{\alpha}$ (say α_n), the objective becomes

$$\underset{0 \leq \alpha_n \leq C}{\operatorname{argmax}} \alpha_n - \frac{1}{2} \alpha_n^2 \|\mathbf{x}_n\|^2 - \frac{1}{2} \alpha_n y_n \sum_{m \neq n} \alpha_m y_m \mathbf{x}_m^\top \mathbf{x}_n$$

Can compute these in
the beginning itself

Can efficiently compute it if we also store \mathbf{w} .
It is equal to $\mathbf{w}^\top \mathbf{x}_n - \alpha_n y_n \|\mathbf{x}_n\|^2$

- The above is a simple quadratic maximization of a concave function: Global maxima
- If constraint violated, project α_n in $[0, C]$: If $\alpha_n < 0$, set it to 0, if $\alpha_n > C$, set it to C
- Can cycle through each coordinate α_n in a random or cyclic fashion



Multi-class SVM

- Multiclass SVMs (assuming $K > 2$ classes) use K wt vectors $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K]$

Prediction at test time: $\hat{y}_* = \operatorname{argmax}_{k \in \{1, 2, \dots, K\}} \mathbf{w}_k^\top \mathbf{x}_*$

- Like binary SVM, can formulate a maximum-margin problem (without or with slacks)

$$\hat{\mathbf{W}} = \arg \min_{\mathbf{w}} \sum_{k=1}^K \frac{\|\mathbf{w}_k\|^2}{2}$$

$$\text{s.t. } \mathbf{w}_{y_n}^\top \mathbf{x}_n \geq \mathbf{w}_k^\top \mathbf{x}_n + 1 \quad \forall k \neq y_n$$

Score on correct class

Score on an incorrect class $k \neq y_n$

$$\hat{\mathbf{W}} = \arg \min_{\mathbf{w}} \sum_{k=1}^K \frac{\|\mathbf{w}_k\|^2}{2} + C \sum_{n=1}^N \xi_n$$

$$\text{s.t. } \mathbf{w}_{y_n}^\top \mathbf{x}_n \geq \mathbf{w}_k^\top \mathbf{x}_n + 1 - \xi_n \quad \forall k \neq y_n$$

- The version with slack corresponds to minimizing a multi-class hinge loss

$$\mathcal{L}(\mathbf{W}) = \sum_{n=1}^N \max \left\{ 0, 1 + \max_{k \neq y_n} \mathbf{w}_k^\top \mathbf{x}_n - \mathbf{w}_{y_n}^\top \mathbf{x}_n \right\} + \frac{\lambda}{2} \sum_{k=1}^K \|\mathbf{w}_k\|^2$$

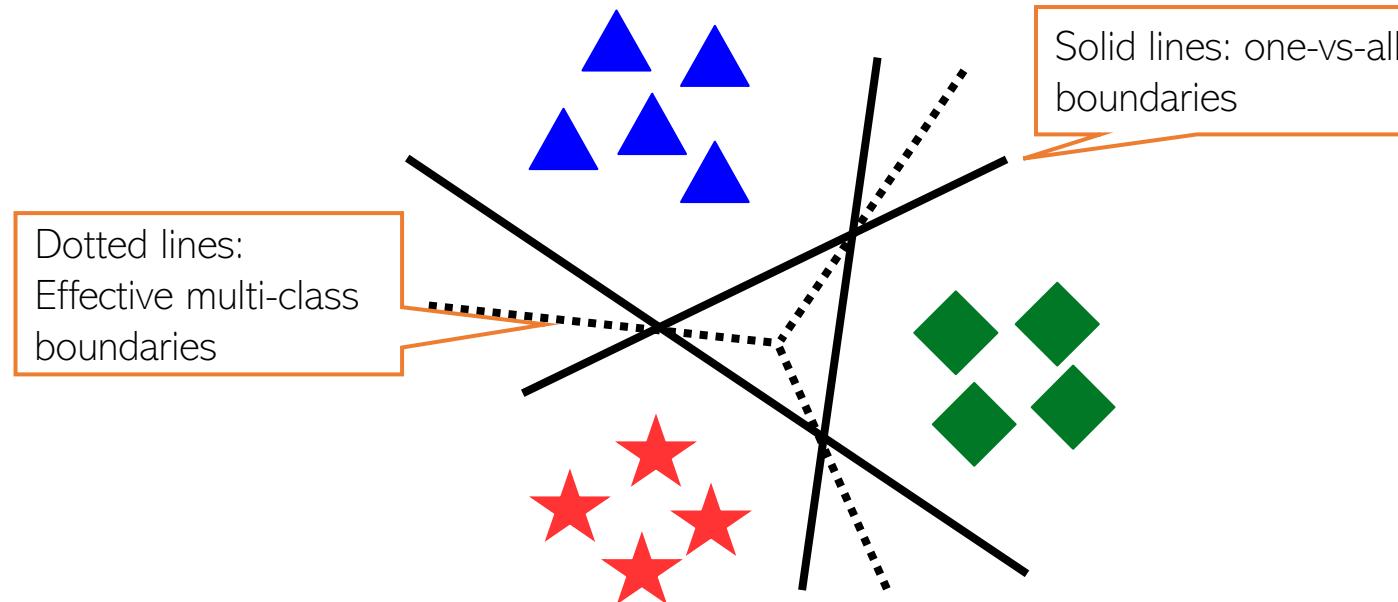
Loss=0 if score on correct class is at least 1 more than score on next best scoring class

Crammer-Singer
Multi-class SVM



Multi-class SVM using Binary SVM

- Can use binary classifiers to solve multiclass problems
- One-vs-All (also called One-vs-Rest): Construct K binary classification problems



- All-Pairs: Learn K -choose-2 binary classifiers, one for each pair of classes (j, k)

Whichever class k wins the most over other classes (or has the largest total scores against all other classes) is the prediction

$$y_* = \arg \max_k \sum_{j \neq k} \mathbf{w}_{j,k}^\top \mathbf{x}_*$$

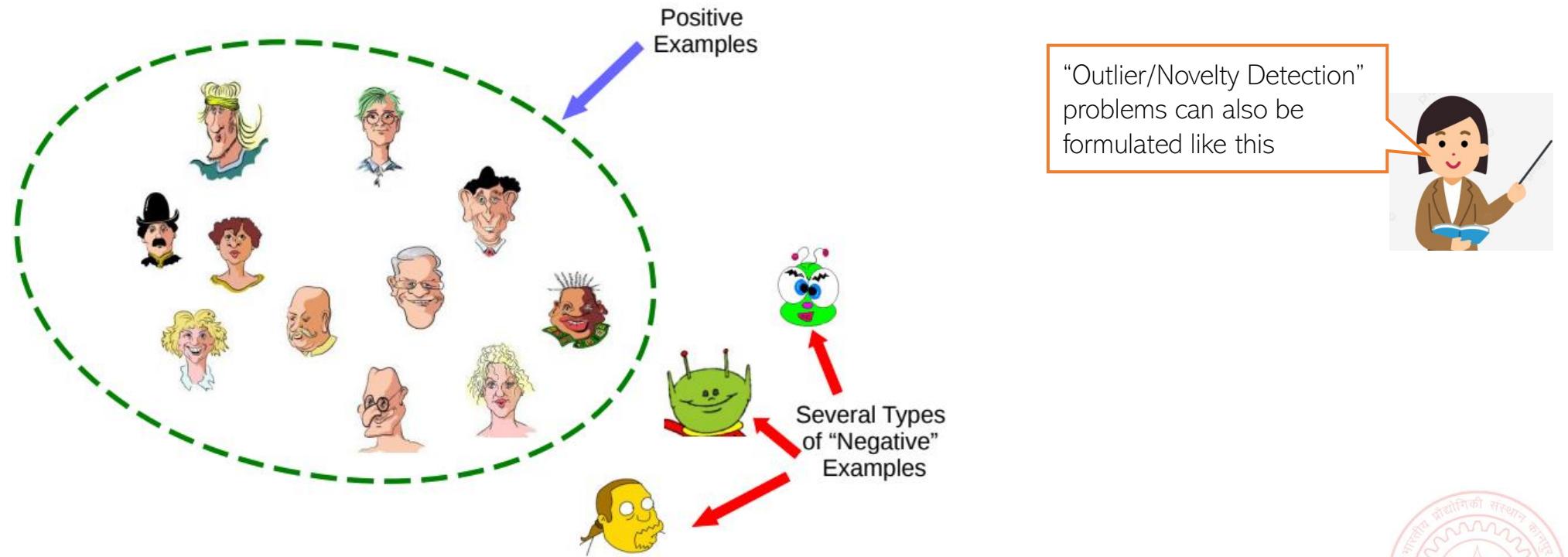
Weight vector of the pairwise classifier for class j and k

Positive score if class k wins over class j in pairwise comparison

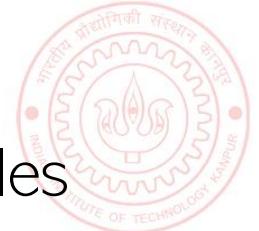


One-class Classification

- Can we learn from examples of just one class, say positive examples?
- May be desirable if there are many types of negative examples



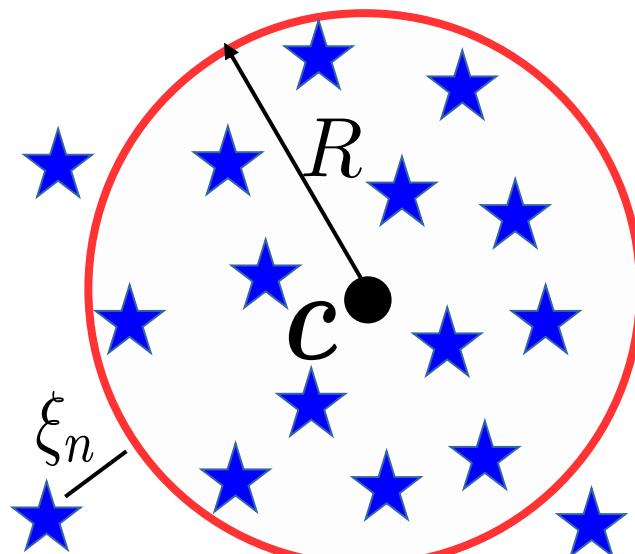
- One-class classification is an approach to learn using only one class of examples



One-class Classification via SVM-type Methods

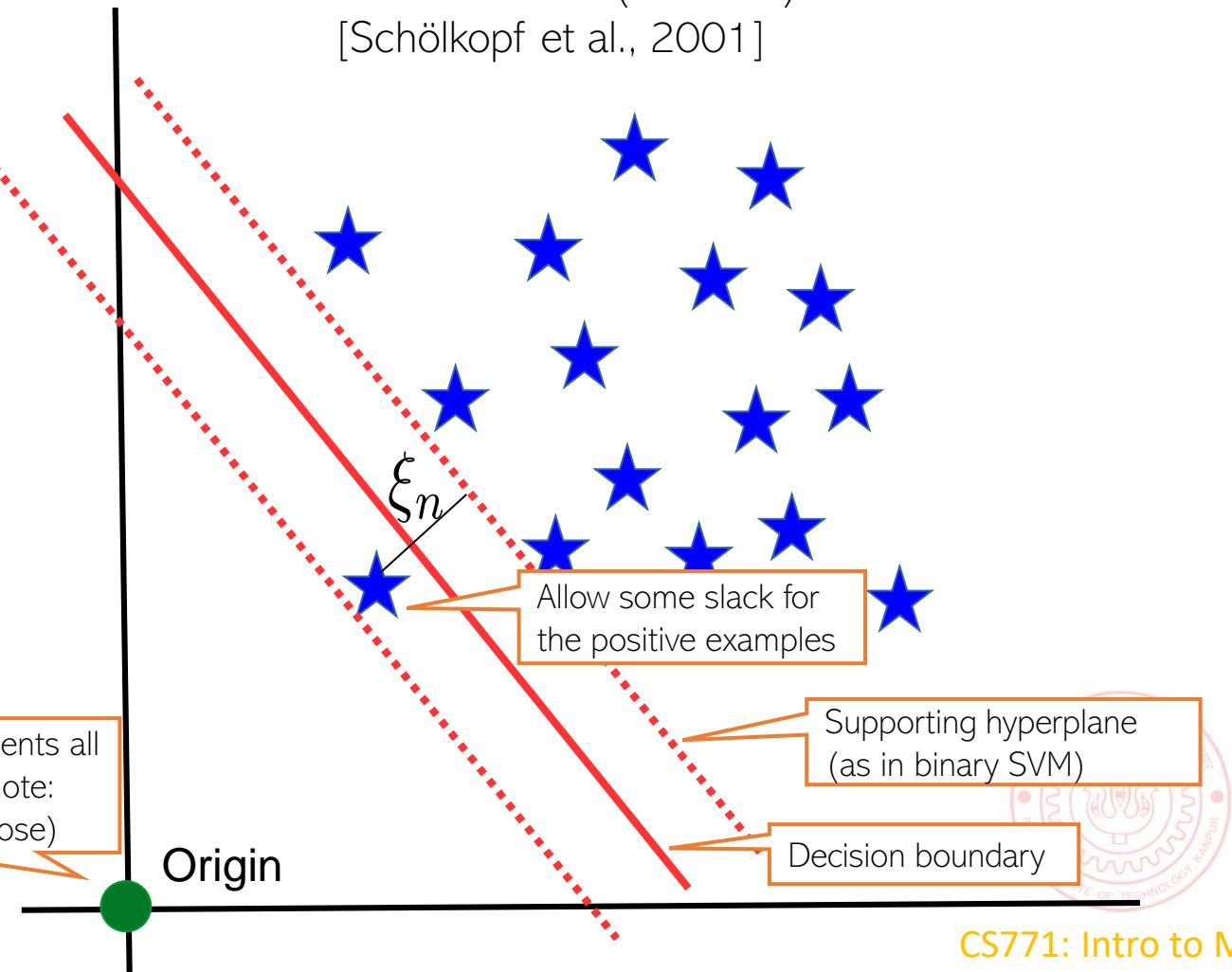
- There are two popular SVM-type approaches to solve one-class problems

"Support Vector Data Description" (SVDD)
[Tax and Duin, 2004]

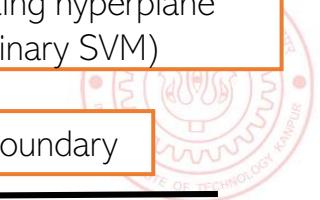


Learn a ball of smallest possible radius R centered at location c that enclosed all positive examples (all some positives to "slack off" and fall outside)

"One-Class SVM" (OC-SVM)
[Schölkopf et al., 2001]

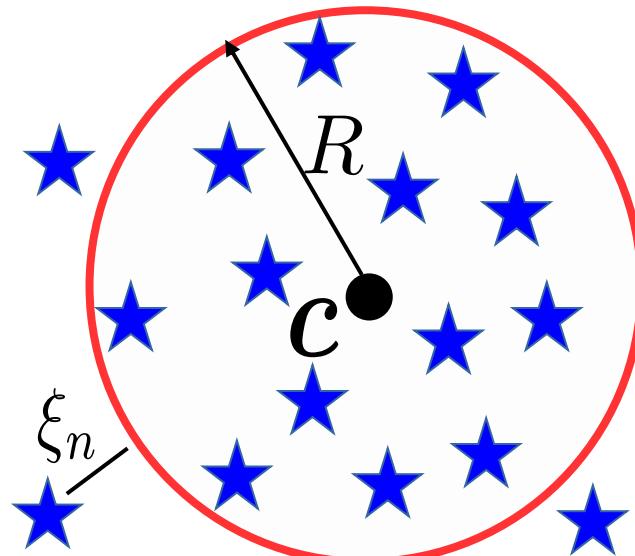


Pretend that origin represents all the negative examples (note: we aren't given any of those)



One-class Classification via SVM-type Methods

“Support Vector Data Description” (SVDD)
[Tax and Duin, 2004]



$$\begin{aligned} & \arg \min_{R, c, \xi} R^2 + \frac{1}{\nu N} \sum_{n=1}^N \xi_n \\ \text{s.t. } & \|x_n - c\|^2 \leq R^2 + \xi_n \quad \forall n \\ & \xi_n \geq 0 \end{aligned}$$

Want to keep the ball's radius as small as possible

Want all training examples to fall within the ball (up to some slack ξ_n)

Hyperparameter ν to trade-off b/w the two terms

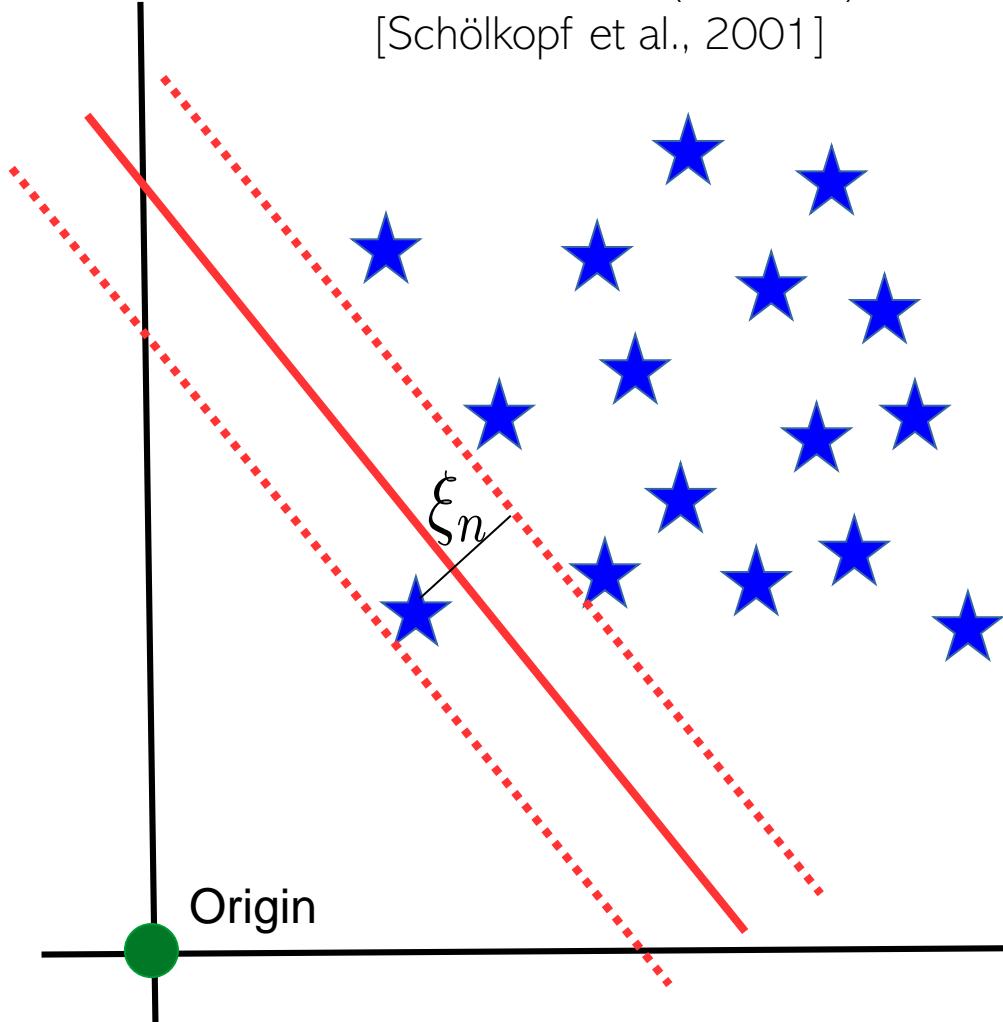
Want to keep training error (sum of slacks) to be small

Prediction Rule: $y_* = +1$ if $\|x_* - c\|^2 - R^2 < 0$



One-class Classification via SVM-type Methods

“One-Class SVM” (OC-SVM)
[Schölkopf et al., 2001]



Maximize the margin
(similar to binary SVM)

$$\arg \min_{\mathbf{w}, \rho, \xi} \|\mathbf{w}\|^2 + \frac{1}{\nu N} \sum_{n=1}^N \xi_n - \rho$$

s.t. $\mathbf{w}^\top \mathbf{x}_n \geq \rho - \xi_n \quad \forall n$

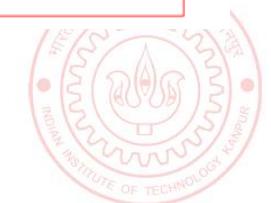
$$\xi_n \geq 0$$

Want to keep training error
(sum of slacks) to be small

An offset term
(want it large)

Want a sufficiently
large score (say ρ)

Prediction Rule: $y_* = +1 \quad \text{if } \mathbf{w}^\top \mathbf{x}_* > \rho$



Coming up next

- Kernel methods and nonlinear SVM via kernels



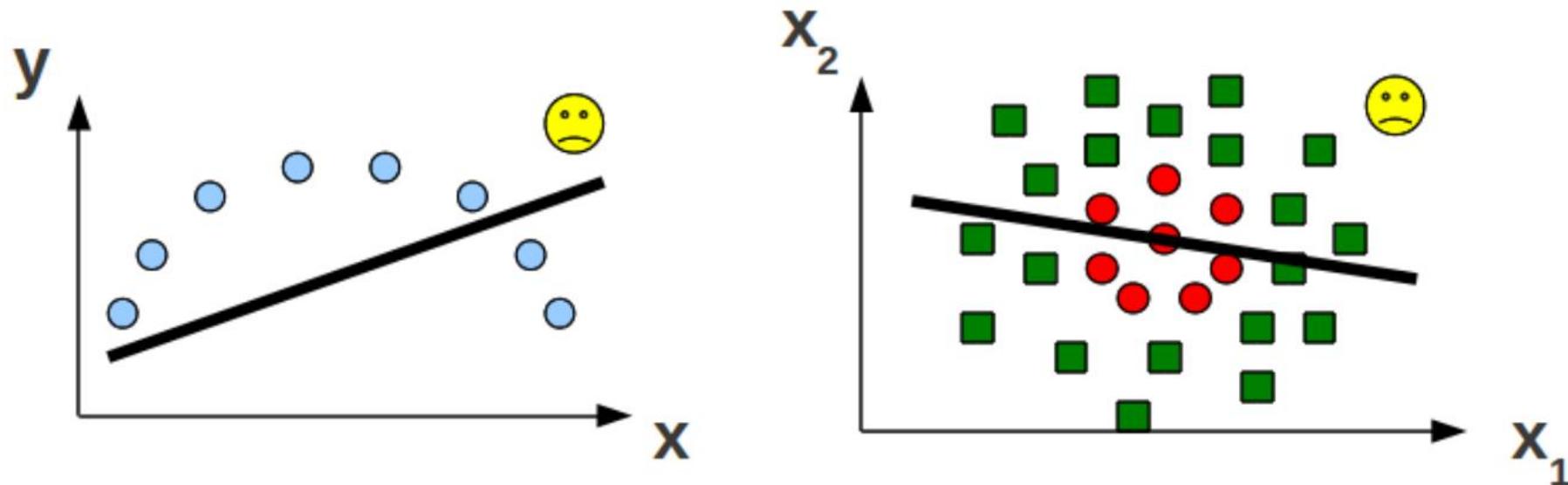
Turning Linear Models into Nonlinear Models using Kernel Methods

CS771: Introduction to Machine Learning

Piyush Rai

Linear Models

- Nice and interpretable but can't learn “difficult” nonlinear patterns



- So, are linear models useless for such problems?



Linear Models for Nonlinear Problems

- Consider the following one-dimensional inputs from two classes

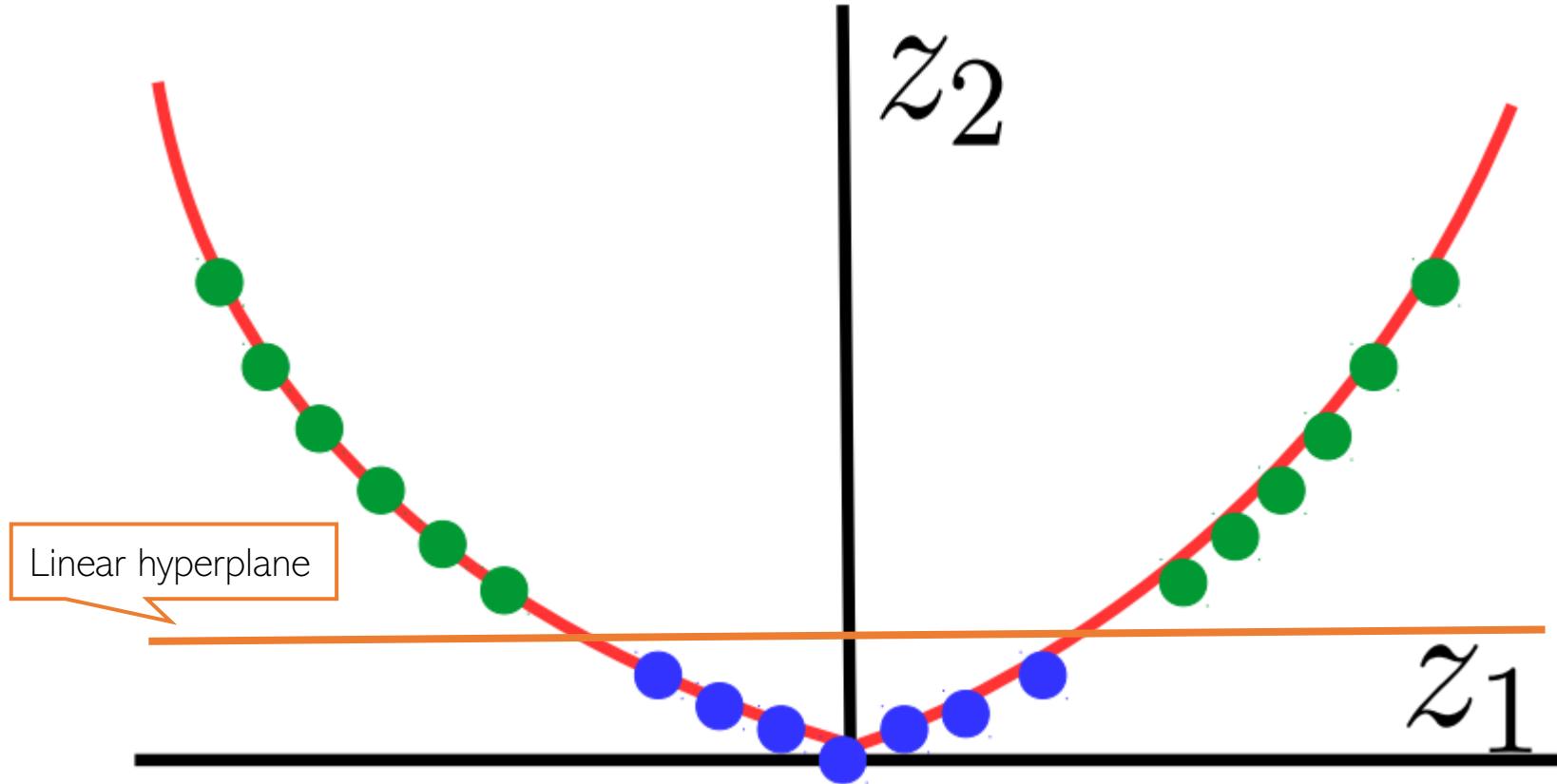


- Can't separate using a linear hyperplane



Linear Models for Nonlinear Problems

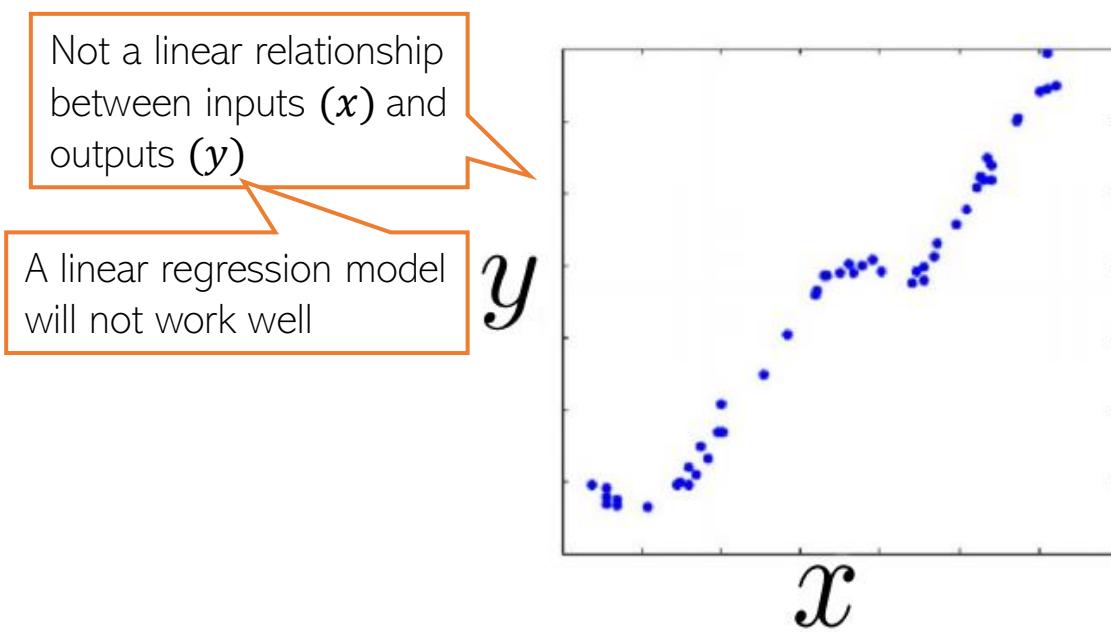
- Consider mapping each x to two-dimensions as $x \rightarrow z = [z_1, z_2] = [x, x^2]$



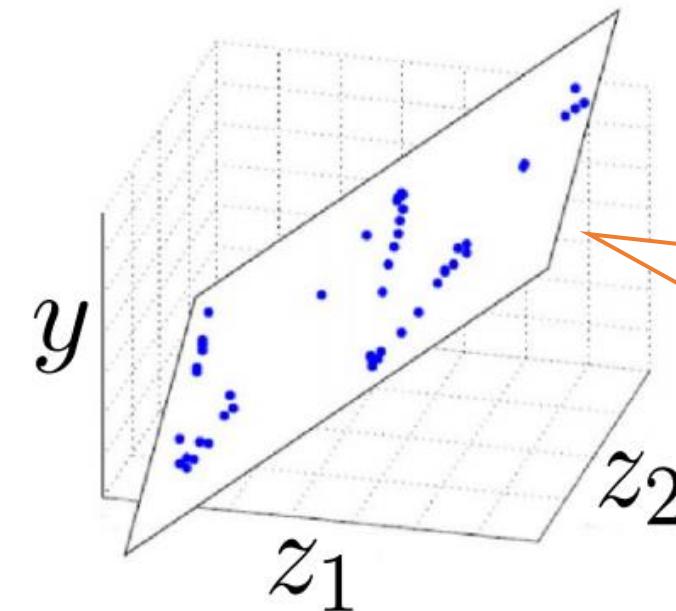
- Classes are now linearly separable in the two-dimensional space

Linear Models for Nonlinear Problems

- The same idea can be applied for nonlinear regression as well



$$x \rightarrow z = [z_1, z_2] = [x, \cos(x)]$$

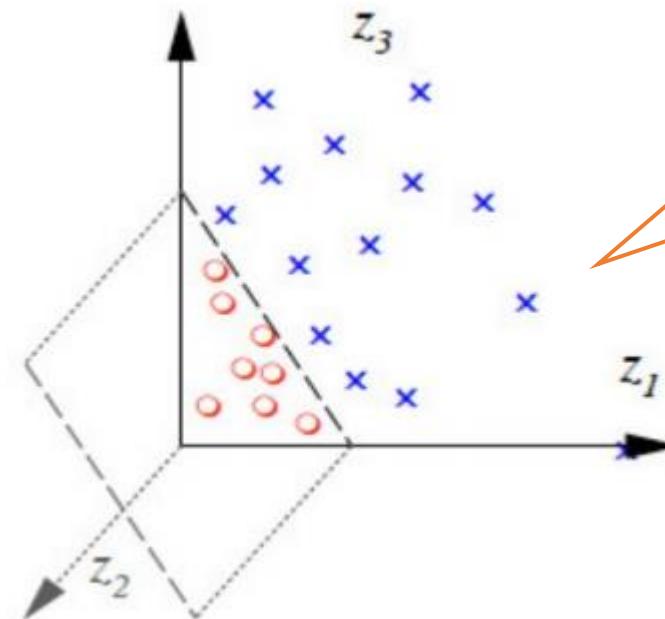
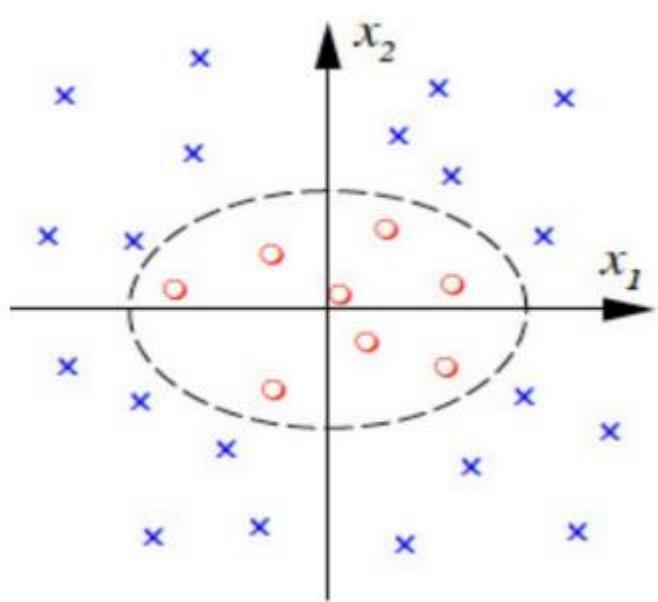


Linear Models for Nonlinear Problems

- Can assume a feature mapping ϕ that maps/transforms the inputs to a “nice” space

$$\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{2}x_1x_2, x_2^2)$$



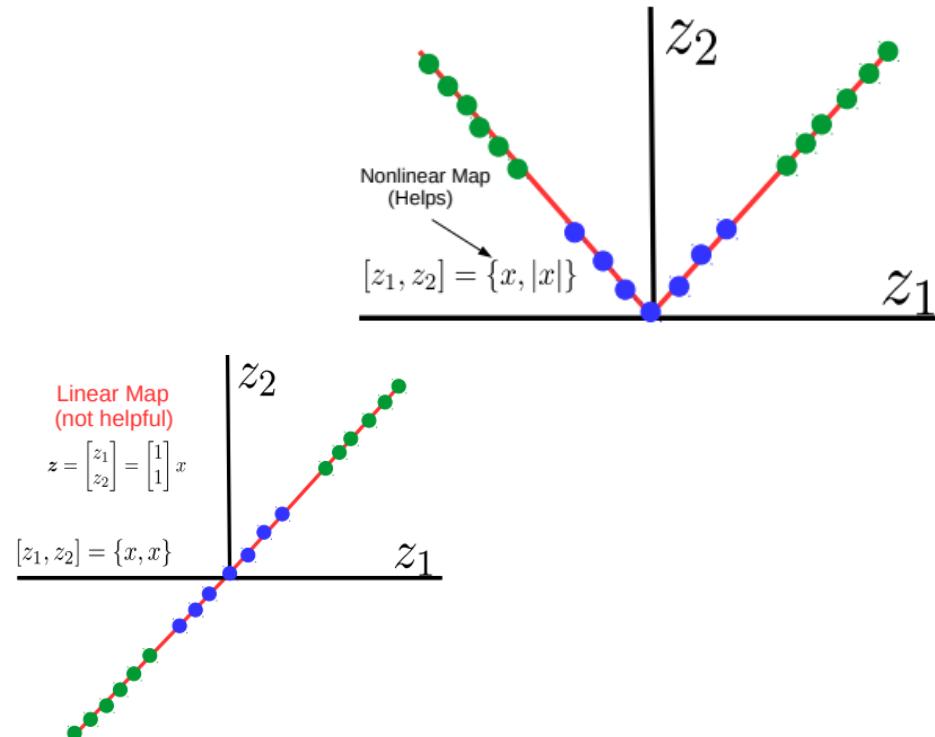
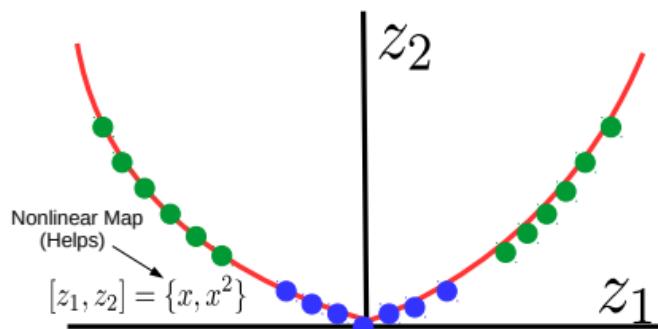
The linear model in the new feature space corresponds to a nonlinear model in the original feature space

- .. and then happily apply a linear model in the new space!



Not Every Mapping is Helpful

- Not every higher-dim mapping helps in learning nonlinear patterns
- Must be a nonlinear mapping
- For the nonlin classfn problem we saw earlier, consider some possible mappings



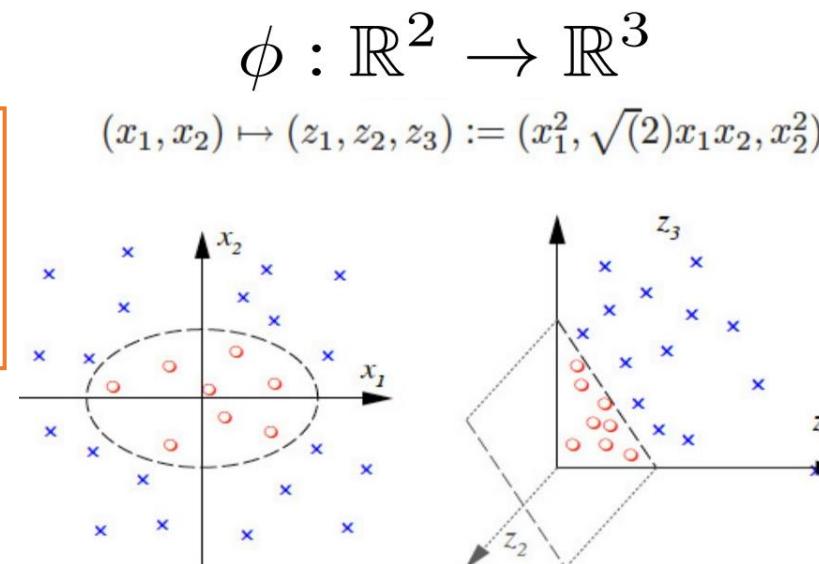
How to get these “good” (nonlinear) mappings?

- Can try to learn the mapping from the data itself (e.g., using **deep learning** - later)
- Can use pre-defined “good” mappings (e.g., defined by kernel functions - today’s topic)



Even if I knew a good mapping, it seems I need to apply it for every input. Won’t this be computationally expensive?

Also, the number of features will increase? Will it not slow down the learning algorithm?



Thankfully, using kernels, you don’t need to compute these mappings explicitly



The kernel will define an “implicit” feature mapping

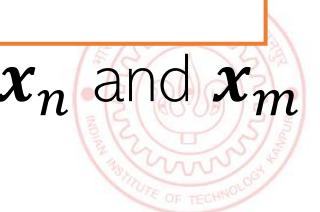
Important: The idea can be applied to any ML algo in which training and test stage only require computing pairwise similarities b/w inputs

In a high-dim space implicitly defined by an underlying mapping ϕ associated with this kernel function $k(\dots)$

- Kernel: A function $k(\dots)$ that gives dot product similarity b/w two inputs, say \mathbf{x}_n and \mathbf{x}_m

Important: As we will see, computing $k(\dots)$ does not require computing the mapping ϕ

$$k(\mathbf{x}_n, \mathbf{x}_m) = \phi(\mathbf{x}_n)^T \phi(\mathbf{x}_m)$$



Kernels as (Implicit) Feature Maps

- Consider two inputs (in the same two-dim feature space): $\mathbf{x} = [x_1, x_2]$, $\mathbf{z} = [z_1, z_2]$
- Suppose we have a function $k(\cdot, \cdot)$ which takes two inputs \mathbf{x} and \mathbf{z} and computes

Called the
“kernel function”

$$k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z})^2$$

Can think of this as a notion
of similarity b/w \mathbf{x} and \mathbf{z}

This is not a dot/inner product
similarity but similarity using a
more general function of \mathbf{x} and \mathbf{z}
(square of dot product)

Didn't need to compute $\phi(\mathbf{x})$
explicitly. Just using the definition
of the kernel $k(\mathbf{x}, \mathbf{z}) =$
 $(\mathbf{x}^\top \mathbf{z})^2$ implicitly gave us this
mapping for each input

$$= (x_1 z_1 + x_2 z_2)^2$$

$$= x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 x_2 z_1 z_2$$

$$= (x_1^2, \sqrt{2}x_1 x_2, x_2^2)^\top (z_1^2, \sqrt{2}z_1 z_2, z_2^2)$$

$$= \phi(\mathbf{x})^\top \phi(\mathbf{z})$$

Dot product similarity in
the new feature space
defined by the mapping ϕ

Remember that a kernel
does two things: Maps
the data implicitly into a
new feature space
(feature transformation)
and computes pairwise
similarity between any
two inputs under the new
feature representation



Thus kernel function $k(\mathbf{x}, \mathbf{z}) =$
 $(\mathbf{x}^\top \mathbf{z})^2$
implicitly defined a feature mapping
 ϕ such that for $\mathbf{x} = [x_1, x_2]$,
 $\phi(\mathbf{x}) = (x_1^2, \sqrt{2}x_1 x_2, x_2^2)$

- Also didn't have to compute $\phi(\mathbf{x})^\top \phi(\mathbf{z})$. Defn $k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z})^2$ gives that

Kernel Functions

As we saw, kernel function $k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z})^2$ implicitly defines a feature mapping ϕ such that for a two-dim $\mathbf{x} = [x_1, x_2]$, $\phi(\mathbf{x}) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$

- Every kernel function k implicitly defines a feature mapping ϕ
- ϕ takes input $\mathbf{x} \in \mathcal{X}$ (e.g., \mathbb{R}^D) and maps it to a new “feature space” \mathcal{F}
- The kernel function k can be seen as taking two points as inputs and computing their inner-product based similarity in the \mathcal{F} space

$$\phi : \mathcal{X} \rightarrow \mathcal{F}$$

$$k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}, \quad k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})$$

For some kernels, as we will see shortly, $\phi(\mathbf{x})$ (and thus the new feature space \mathcal{F}) can be very **high-dimensional** or even be **infinite dimensional** (but we don't need to compute it anyway, so it is not an issue)

- \mathcal{F} needs to be a vector space with a dot product defined on it (a.k.a. a **Hilbert space**)
- Is any function $k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})$ for some ϕ a kernel function?
 - No. The function k must satisfy **Mercer's Condition**



Kernel Functions

- For $k(\cdot, \cdot)$ to be a kernel function
 - k must define a dot product for some Hilbert Space
 - Above is true if k is **symmetric** and **positive semi-definite** (p.s.d.) function (though there are exceptions; there are also “indefinite” kernels)

For all “square integrable” functions f
 (such functions satisfy $\int f(x)^2 dx < \infty$)

$$k(\mathbf{x}, \mathbf{z}) = k(\mathbf{z}, \mathbf{x})$$

$$\iint f(\mathbf{x})k(\mathbf{x}, \mathbf{z})f(\mathbf{z})d\mathbf{x}d\mathbf{z} \geq 0$$

Loosely speaking a PSD function here means that if we evaluate this function for N inputs (N^2 pairs) then the $N \times N$ matrix will be PSD (also called a kernel matrix)

- The above condition is essentially known as Mercer’s Condition
- Let k_1, k_2 be two kernel functions then the following are as well
 - $k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z}) + k_2(\mathbf{x}, \mathbf{z})$: simple sum
 - $k(\mathbf{x}, \mathbf{z}) = \alpha k_1(\mathbf{x}, \mathbf{z})$: scalar product
 - $k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z})k_2(\mathbf{x}, \mathbf{z})$: direct product of two kernels

Can easily verify that the Mercer’s Condition holds

Can also combine these rules and the resulting function will also be a kernel function



Some Pre-defined Kernel Functions

- Linear kernel: $k(\mathbf{x}, \mathbf{z}) = \mathbf{x}^\top \mathbf{z}$
- Quadratic Kernel: $k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z})^2$ or $k(\mathbf{x}, \mathbf{z}) = (1 + \mathbf{x}^\top \mathbf{z})^2$
- Polynomial Kernel (of degree d): $k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z})^d$ or $k(\mathbf{x}, \mathbf{z}) = (1 + \mathbf{x}^\top \mathbf{z})^d$
- Radial Basis Function (RBF) or “Gaussian” Kernel: $k(\mathbf{x}, \mathbf{z}) = \exp[-\gamma \|\mathbf{x} - \mathbf{z}\|^2]$
 - Gaussian kernel gives a similarity score between 0 and 1
 - $\gamma > 0$ is a hyperparameter (called the kernel bandwidth parameter)
 - The RBF kernel corresponds to an infinite dim. feature space \mathcal{F} (i.e., you can't actually write down or store the map $\phi(\mathbf{x})$ explicitly – but we don't need to do that anyway ☺)
 - Also called “stationary kernel”: only depends on the distance between \mathbf{x} and \mathbf{z} (translating both by the same amount won't change the value of $k(\mathbf{x}, \mathbf{z})$)
 - Kernel hyperparameters (e.g., d, γ) can be set via cross-validation

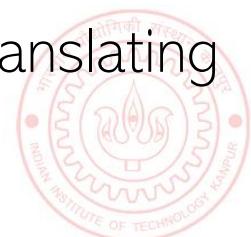
Several other kernels proposed for non-vector data, such as trees, strings, etc

Remember that kernels are a notion of similarity between pairs of inputs



Kernels can have a pre-defined form or can be learned from data (a bit advanced for this course)

Controls how the distance between two inputs should be converted into a similarity



RBF Kernel = Infinite Dimensional Mapping

- We saw that the RBF/Gaussian kernel is defined as $k(\mathbf{x}, \mathbf{z}) = \exp[-\gamma \|\mathbf{x} - \mathbf{z}\|^2]$
- Using this kernel corresponds to mapping data to infinite dimensional space

$$\begin{aligned}
 k(x, z) &= \exp[-(x - z)^2] \quad (\text{assuming } \gamma = 1 \text{ and } x \text{ and } z \text{ to be scalars}) \\
 &= \exp(-x^2) \exp(-z^2) \exp(2xz) \\
 &= \exp(-x^2) \exp(-z^2) \sum_{k=1}^{\infty} \frac{2^k x^k z^k}{k!} \\
 &= \phi(x)^T \phi(z)
 \end{aligned}$$

Thus an infinite-dim vector (ignoring the constants coming from the 2^k and $k!$ terms)

- Here $\phi(\mathbf{x}) = [\exp(-x^2)x^1, \exp(-x^2)x^2, \exp(-x^2)x^3, \dots, \exp(-x^2)x^\infty]$
- But again, note that we never need to compute $\phi(\mathbf{x})$ to compute $k(\mathbf{x}, \mathbf{z})$
 - $k(\mathbf{x}, \mathbf{z})$ is easily computable from its definition itself ($\exp[-(x - z)^2]$ in this case)



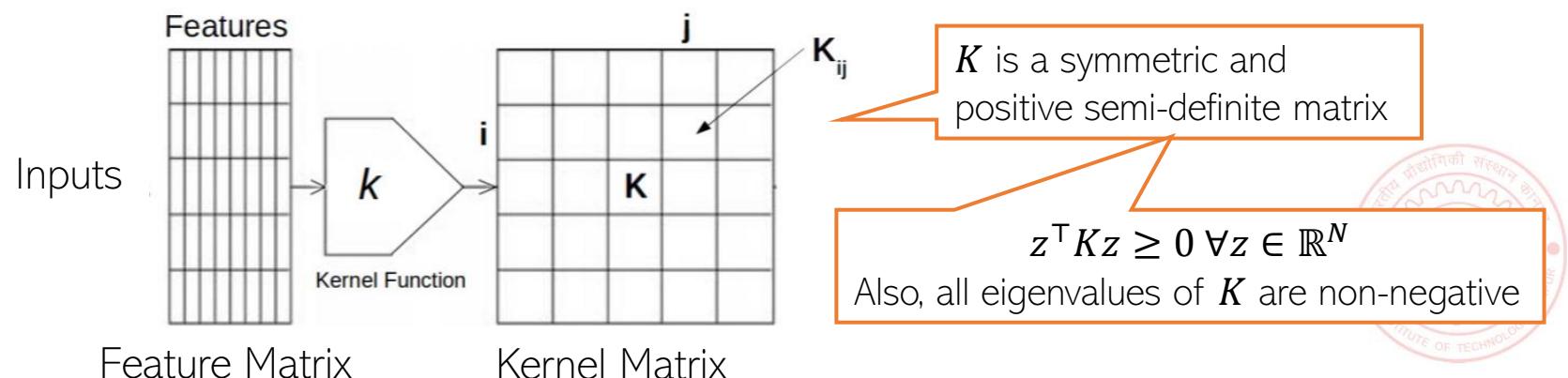
Kernel Matrix

- Kernel based ML algos work with **kernel matrices** rather than feature vectors
- Given N inputs, the kernel function k can be used to construct a Kernel Matrix \mathbf{K}
- The kernel matrix \mathbf{K} is of size $N \times N$ with each entry defined as

$$K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$$

Note again that we don't need to compute ϕ and this dot product explicitly

- K_{ij} : Similarity between the i^{th} and j^{th} inputs in the kernel induced feature space ϕ



Coming up next..

- Applying kernel methods for SVM and ridge regression



Turning Linear Models into Nonlinear Models using Kernel Methods (Contd)

CS771: Introduction to Machine Learning

Piyush Rai

Using Kernels

- Kernels can turn many linear models into nonlinear models
- Recall that $k(\mathbf{x}, \mathbf{z})$ represents a dot product in some high-dim feature space \mathcal{F}
- **Important:** Any ML model/algo in which, during training and test, inputs only appear as dot product can be “kernelized”
- Just replace each term of the form $\mathbf{x}_i^T \mathbf{x}_j$ by $\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j) = K_{ij}$
- Most ML models/algos can be easily kernelized, e.g.,
 - Distance based methods, Perceptron, SVM, linear regression, etc.
 - Many of the unsupervised learning algorithms too can be kernelized (e.g., K-means clustering, Principal Component Analysis, etc. - will see later)
 - Let's look at two examples: Kernelized SVM and Kernelized Ridge Regression



An Aside: Kernelizing a Euclidean Distance

- Many algorithms, e.g., LwP, KNN, etc. use Euclidean distances, e.g.,

$$d(a, b) = \|a - b\|^2 = \|a\|^2 + \|b\|^2 - 2a^\top b = a^\top a + b^\top b - 2a^\top b$$

- This can be kernelized as well by replacing the above norms and inner products by their kernelized versions, assuming a kernel k with feature map ϕ

$$\begin{aligned} d(\phi(a), \phi(b)) &= \|\phi(a) - \phi(b)\|^2 \\ &= \phi(a)^\top \phi(a) + \phi(b)^\top \phi(b) - 2\phi(a)^\top \phi(b) \\ &= k(a, a) + k(b, b) - 2k(a, b) \end{aligned}$$



Nonlinear SVM using Kernels



Kernelized SVM Training

- Recall the soft-margin linear SVM objective (with no bias term)

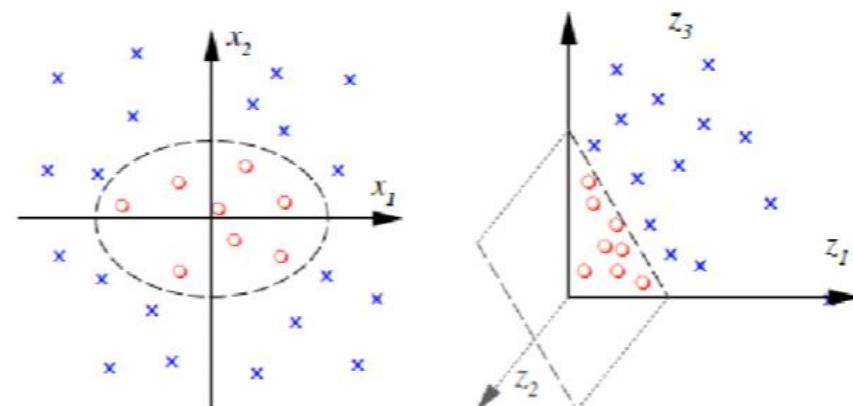
$$\underset{0 \leq \alpha \leq C}{\operatorname{argmax}} \quad \boldsymbol{\alpha}^T \mathbf{1} - \frac{1}{2} \boldsymbol{\alpha}^T \mathbf{G} \boldsymbol{\alpha}$$

$G_{ij} = y_i y_j \mathbf{x}_i^T \mathbf{x}_j$

Inputs only appear
as dot products ☺

- To kernelize, we can simply replace $G_{ij} = y_i y_j \mathbf{x}_i^T \mathbf{x}_j$ by $y_i y_j K_{ij}$
 - .. where $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ for a suitable kernel function k

- The problem can now be solved just like the linear SVM case
- The new SVM learns a linear separator in kernel-induced feature space \mathcal{F}
 - This corresponds to a **non-linear separator** in the original feature space \mathcal{X}



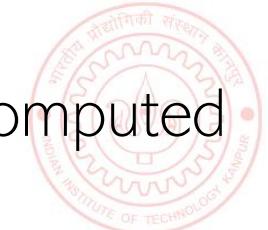
Kernelized SVM Prediction

- SVM weight vector for the kernelized case will be $\mathbf{w} = \sum_{n=1}^N \alpha_n y_n \phi(\mathbf{x}_n)$
- Note: We can't store \mathbf{w} unless the feature mapping $\phi(\mathbf{x}_n)$ is finite dimensional
 - In practice, we store the α_n 's and the training data for test time (just like KNN)
 - In fact, need to store only training examples for which α_n is nonzero (i.e., the support vectors)

- Prediction for a new test input \mathbf{x}_* (assuming hyperplane's bias $b = 0$) will be

$$y_* = \text{sign}(\mathbf{w}^\top \phi(\mathbf{x}_*)) = \text{sign}\left(\sum_{n=1}^N \alpha_n y_n \phi(\mathbf{x}_n)^\top \phi(\mathbf{x}_*)\right) = \text{sign}\left(\sum_{n=1}^N \alpha_n y_n k(\mathbf{x}_n, \mathbf{x}_*)\right)$$

- Note that the prediction cost also scales linearly with N (unlike a linear model where we only need to compute $\mathbf{w}^\top \mathbf{x}_*$, whose cost only depends on D , not N)
- Also note that, for unkernelized (i.e., linear) SVM, $\mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n$ can be computed and stored as a $D \times 1$ vector and we can compute $\mathbf{w}^\top \mathbf{x}_*$ in $O(D)$ time



Nonlinear Ridge Regression using Kernels



Kernelized Ridge Regression

- Recall the ridge regression problem: $\mathbf{w} = \arg \min_{\mathbf{w}} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \lambda \mathbf{w}^\top \mathbf{w}$
- The solution to this problem was



They do; with a bit of algebra ☺

$$\mathbf{w} = \left(\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top + \lambda \mathbf{I}_D \right) \left(\sum_{n=1}^N y_n \mathbf{x}_n \right) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \mathbf{y}$$

Inputs don't appear to be as inner product. No hope of kernelization?



- Can use matrix inversion lemma $(\mathbf{F}\mathbf{H}^{-1}\mathbf{G} - \mathbf{E})^{-1}\mathbf{F}\mathbf{H}^{-1} = \mathbf{E}^{-1}\mathbf{F}(\mathbf{G}\mathbf{E}^{-1}\mathbf{F} - \mathbf{H})^{-1}$
- Using the lemma, can rewrite \mathbf{w} as

$$\mathbf{w} = \mathbf{X}^\top (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \mathbf{y} = \mathbf{X}^\top \boldsymbol{\alpha} = \sum_{n=1}^N \alpha_n \mathbf{x}_n \quad \text{where } \boldsymbol{\alpha} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \mathbf{y} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \mathbf{y}$$

$N \times 1$ vector of dual variables

Note: Not sparse unlike SVM

- Kernelized weight vector will be $\mathbf{w} = \sum_{n=1}^N \alpha_n \phi(\mathbf{x}_n)$

Prediction cost is also linear in N (like KNN)

- Prediction for a test input \mathbf{x}_* will be $\mathbf{w}^\top \phi(\mathbf{x}_*) = \sum_{n=1}^N \alpha_n \phi(\mathbf{x}_n)^\top \phi(\mathbf{x}_*) = \sum_{n=1}^N \alpha_n k(\mathbf{x}_n, \mathbf{x}_*)$



Speeding-up Kernel Methods



Speeding-up Kernel Methods

- Kernel methods, unlike linear models are slow at training and test time
- Would be nice if we could easily compute mapping $\phi(\mathbf{x})$ associated with kernel k
- Then we could apply linear models directly on $\phi(\mathbf{x})$ without having to kernelize
- But this is in general not possible since $\phi(\mathbf{x})$ is very high/infinite dimensional
- An alternative: Get a good set of low-dim features $\psi(\mathbf{x}) \in \mathbb{R}^L$ using the kernel k
- If $\psi(\mathbf{x})$ is a good approximation to $\phi(\mathbf{x})$ then we can use $\psi(\mathbf{x})$ in a linear model

Goodness Criterion: $\psi(\mathbf{x}_i)^\top \psi(\mathbf{x}_j) \approx \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$

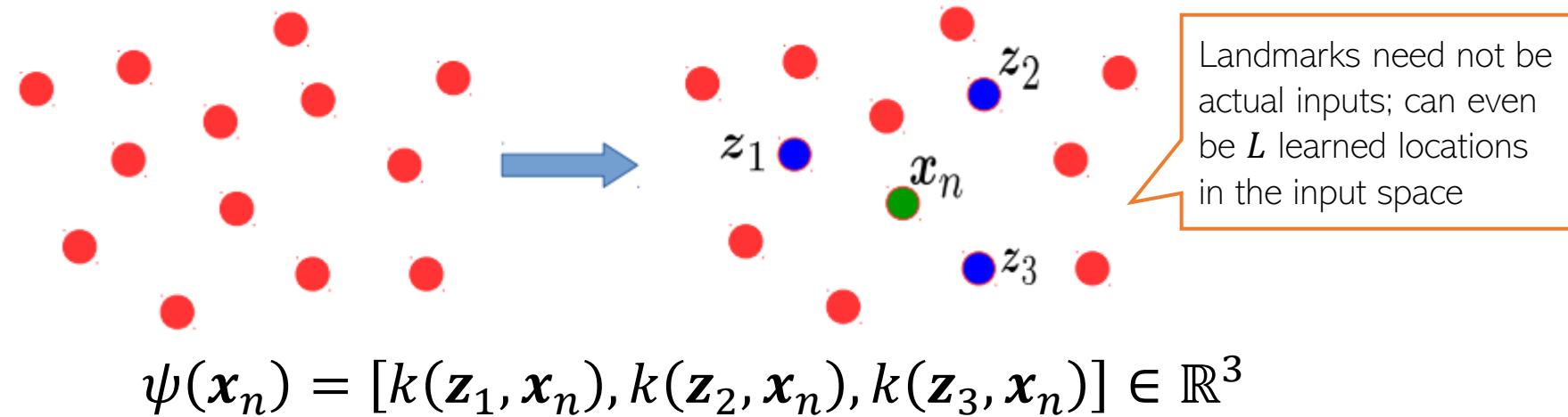
... which also means $\psi(\mathbf{x}_i)^\top \psi(\mathbf{x}_j) \approx k(\mathbf{x}_i, \mathbf{x}_j)$

- Will look at two popular approaches: **Landmarks** and **Random Features**



Extracting Features using Kernels: Landmarks

- Suppose we choose a small set of L “landmark” inputs $\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_L$ in the training data



- For each input \mathbf{x}_n , using a kernel k , define an L -dimensional feature vector as follows

$$\psi(\mathbf{x}_n) = [k(\mathbf{z}_1, \mathbf{x}_n), k(\mathbf{z}_2, \mathbf{x}_n), \dots, k(\mathbf{z}_L, \mathbf{x}_n)] \in \mathbb{R}^L$$

- Can now apply a linear model on ψ representation (L -dimensional now) of the inputs
- This will be fast both at training as well as test time if L is small
- No need to kernelize the linear model while still reaping the benefits of kernels ☺



Extracting Feat. using Kernels: Random Features

- Many kernel functions* can be written as

$$k(\mathbf{x}_n, \mathbf{x}_m) = \phi(\mathbf{x}_n)^\top \phi(\mathbf{x}_m) = \mathbb{E}_{\mathbf{w} \sim p(\mathbf{w})}[t_{\mathbf{w}}(\mathbf{x}_n)t_{\mathbf{w}}(\mathbf{x}_m)]$$

.. where $t_{\mathbf{w}}(\cdot)$ is a function with params $\mathbf{w} \in \mathbb{R}^L$ with \mathbf{w} drawn from some distr. $p(\mathbf{w})$

- Example: For the RBF kernel, $t_{\mathbf{w}}(\cdot)$ is cosine func. and $p(\mathbf{w})$ is zero mean Gaussian

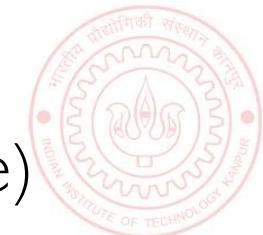
$$k(\mathbf{x}_n, \mathbf{x}_m) = \mathbb{E}_{\mathbf{w} \sim p(\mathbf{w})}[\cos(\mathbf{w}^\top \mathbf{x}_n) \cos(\mathbf{w}^\top \mathbf{x}_m)]$$

- Given $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_L$ from $p(\mathbf{w})$, using Monte-Carlo approx. of above expectation

$$k(\mathbf{x}_n, \mathbf{x}_m) \approx \frac{1}{L} \sum_{\ell=1}^L \cos(\mathbf{w}_\ell^\top \mathbf{x}_n) \cos(\mathbf{w}_\ell^\top \mathbf{x}_m) = \psi(\mathbf{x}_n)^\top \psi(\mathbf{x}_m)$$

.. where $\psi(\mathbf{x}_n) = \frac{1}{\sqrt{L}} [\cos(\mathbf{w}_1^\top \mathbf{x}_n), \dots, \cos(\mathbf{w}_L^\top \mathbf{x}_n)]$ is an L -dim vector

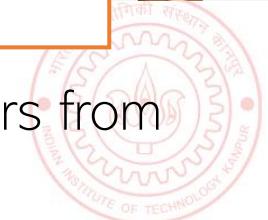
- Can apply a linear model on this L -dim rep. of the inputs (no need to kernelize)



Learning with Kernels: Some Aspects

- Storage/computational efficiency can be a bottleneck when using kernels
- During training, need to compute and store the $N \times N$ kernel matrix \mathbf{K} in memory
- Need to store training data (or at least support vectors in case of SVMs) at test time
- Test time can be slow: $O(N)$ cost to compute a quantity like $\sum_{n=1}^N \alpha_n k(\mathbf{x}_n, \mathbf{x}_*)$
- Approaches like landmark and random features can be used to speed up
- Choice of the right kernel is also very important
- Some kernels (e.g., RBF) work well for many problems but hyperparameters of the kernel function may need to be tuned via cross-validation
- Quite a bit of research on learning the right kernel from data
 - Learning a combination of multiple kernels ([Multiple Kernel Learning](#))
 - [Bayesian kernel methods](#) (e.g., [Gaussian Processes](#)) can learn the kernel hyperparameters from data (thus can be seen as learning the kernel)
 - Deep Learning can also be seen as learning the kernel from data (more on this later)

Also, a lot of recent work on connections between kernel methods and deep learning



Coming up next

- Unsupervised learning



Introduction to Unsupervised Learning, Data Clustering via K-means

CS771: Introduction to Machine Learning

Piyush Rai

Plan

- Introduction to unsupervised learning – clustering
- The K-means algorithm
- Some of the underlying maths behind K-means



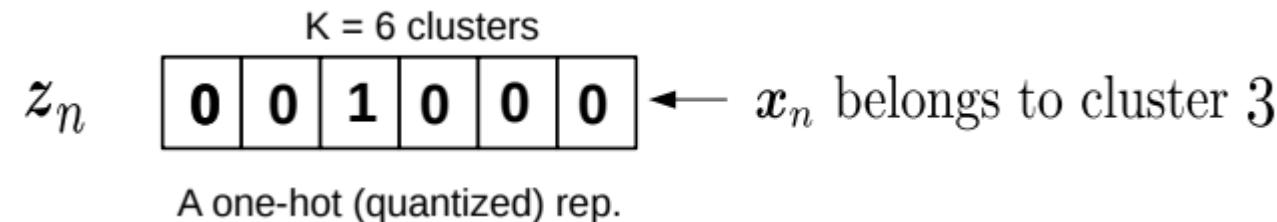
Unsupervised Learning

- It's about learning interesting/useful structures in the data (unsupervisedly!)
- There is no supervision (no labels/responses), only inputs $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$
- Some examples of unsupervised learning
 - Clustering: Grouping similar inputs together (and dissimilar ones far apart)
 - Dimensionality Reduction: Reducing the data dimensionality
 - Estimating the probability density of data (which distribution “generated” the data)
- Most unsup. learning algos can also be seen as learning a new representation of data
 - For example, hard clustering can be used to get a one-hot representation

Already looked at some basic approaches (e.g., estimating a Gaussian given observed data)

Each point belongs deterministically to a single cluster

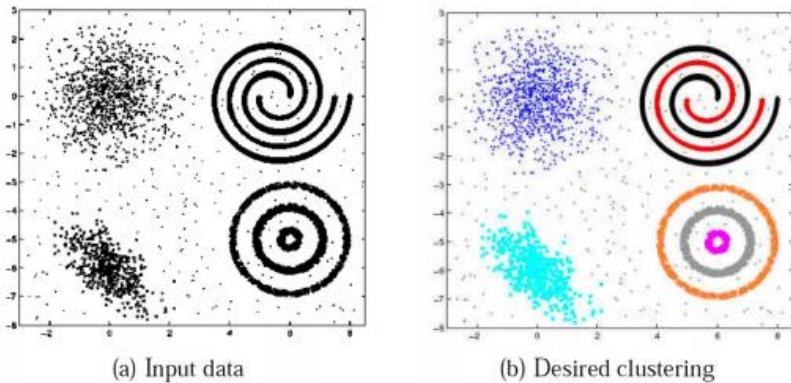
In contrast, there is also soft/probabilistic clustering in which \mathbf{z}_n will be a probability vector that sums to 1 (will see later)



Clustering

In some cases, we may not know the right number of clusters in the data and may want to learn that (technique exists for doing this but beyond the scope)

- Given: N unlabeled examples $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$; desired no. of partitions K
- Goal: Group the examples into K “homogeneous” partitions



Picture courtesy: “Data Clustering: 50 Years Beyond K-Means”, A.K. Jain (2008)

- Loosely speaking, it is classification without ground truth labels
- A good clustering is one that achieves
 - High within-cluster similarity
 - Low inter-cluster similarity



Similarity can be Subjective

- Clustering only looks at similarities b/w inputs, since no labels are given
- Without labels, similarity can be hard to define



- Thus using the right distance/similarity is very important in clustering
- In some sense, related to asking: “Clustering based on what”?



Clustering: Some Examples

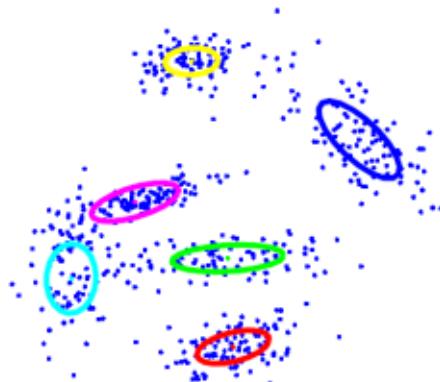
- Document/Image/Webpage Clustering
- Image Segmentation (clustering pixels)



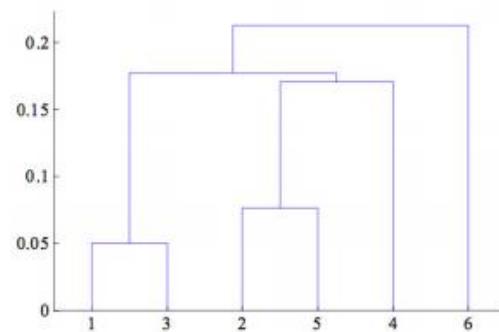
- Clustering web-search results
- Clustering (people) nodes in (social) networks/graphs
- .. and many more..

Types of Clustering

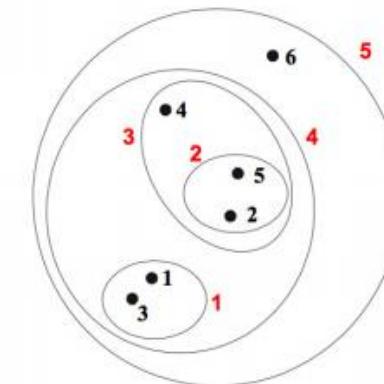
- Flat or Partitional clustering
 - Partitions are independent of each other



- Hierarchical clustering
 - Partitions can be visualized using a tree structure (a dendrogram)



In hierarchical clustering, we can look at a clustering for any given number of cluster by “cutting the dendrogram at an appropriate level (so K does not have to be specified)



Hierarchical clustering gives us a clustering at multiple levels of granularity



Flat Clustering: K -means algorithm (Lloyd, 1957)

- **Input:** N unlabeled examples $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$; $\mathbf{x}_n \in \mathbb{R}^D$; desired no. of partitions K
- **Desired Output:** Cluster assignments of these N examples and K cluster means
- **Initialize:** The K cluster means denoted by $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \dots, \boldsymbol{\mu}_K$; with each $\boldsymbol{\mu}_k \in \mathbb{R}^D$
 - Usually initialized randomly, but good initialization is crucial; many smarter initialization heuristics exist (e.g., K -means++, Arthur & Vassilvitskii, 2007)
- **Iterate:**
 - (Re)-Assign each input \mathbf{x}_n to its closest cluster center (based on the smallest Eucl. distance)

\mathcal{C}_k : Set of examples assigned to cluster k with center $\boldsymbol{\mu}_k$

$$\mathcal{C}_k = \{n : k = \arg \min_k \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2\}$$

- Update the cluster means

$$\boldsymbol{\mu}_k = \text{mean}(\mathcal{C}_k) = \frac{1}{|\mathcal{C}_k|} \sum_{n \in \mathcal{C}_k} \mathbf{x}_n$$

- Repeat until not converged

Some ways to declare convergence if between two successive iterations:

- Cluster means don't change
- Cluster assignments don't change
- Clustering "loss" doesn't change by much



K-means algorithm: Summarized Another Way

- Notation: $z_n \in \{1, 2, \dots, K\}$ or \mathbf{z}_n is a K -dim one-hot vector
 - $(z_{nk} = 1 \text{ and } z_n = k \text{ mean the same})$

K -means algo can also be seen as doing a compression by “quantization”: Representing each of the N inputs by one of the $K < N$ means

K-means Algorithm

- 1 Initialize K cluster means μ_1, \dots, μ_K
- 2 For $n = 1, \dots, N$, assign each point x_n to the closest cluster

$$z_n = \arg \min_{k \in \{1, \dots, K\}} \|x_n - \mu_k\|^2$$

- 3 Suppose $\mathcal{C}_k = \{x_n : z_n = k\}$. Re-compute the means

$$\mu_k = \text{mean}(\mathcal{C}_k), \quad k = 1, \dots, K$$

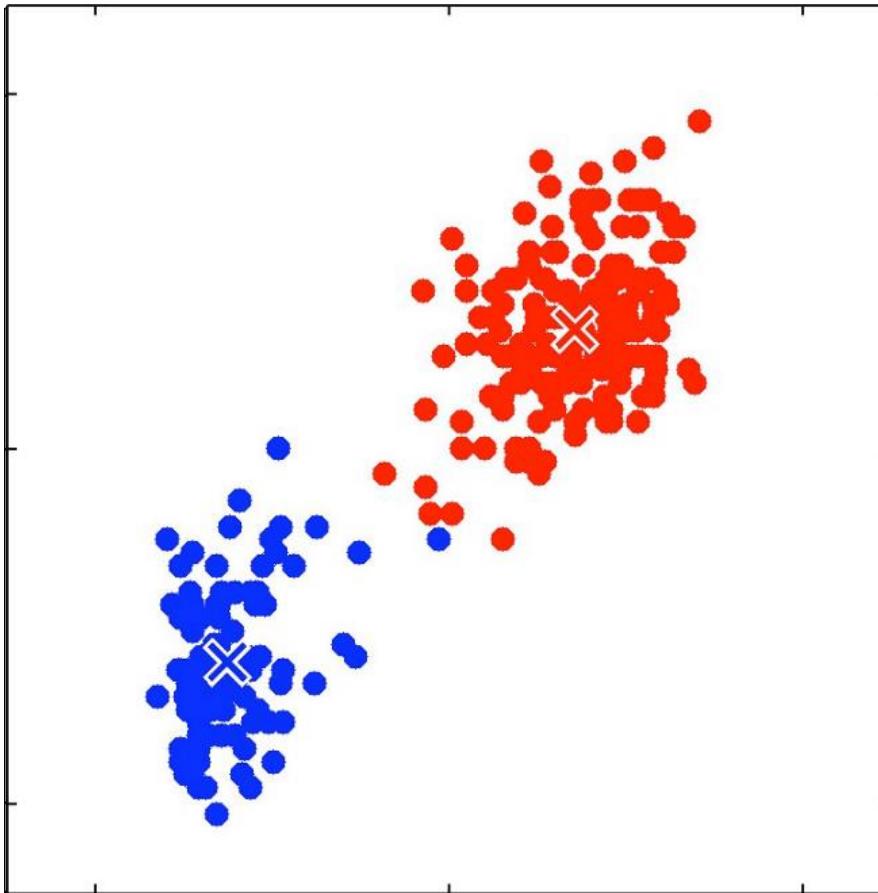
- 4 Go to step 2 if not yet converged

Can be fixed by modeling each cluster by a probability distribution, such as Gaussian (e.g., Gaussian Mixture Model; will see later)

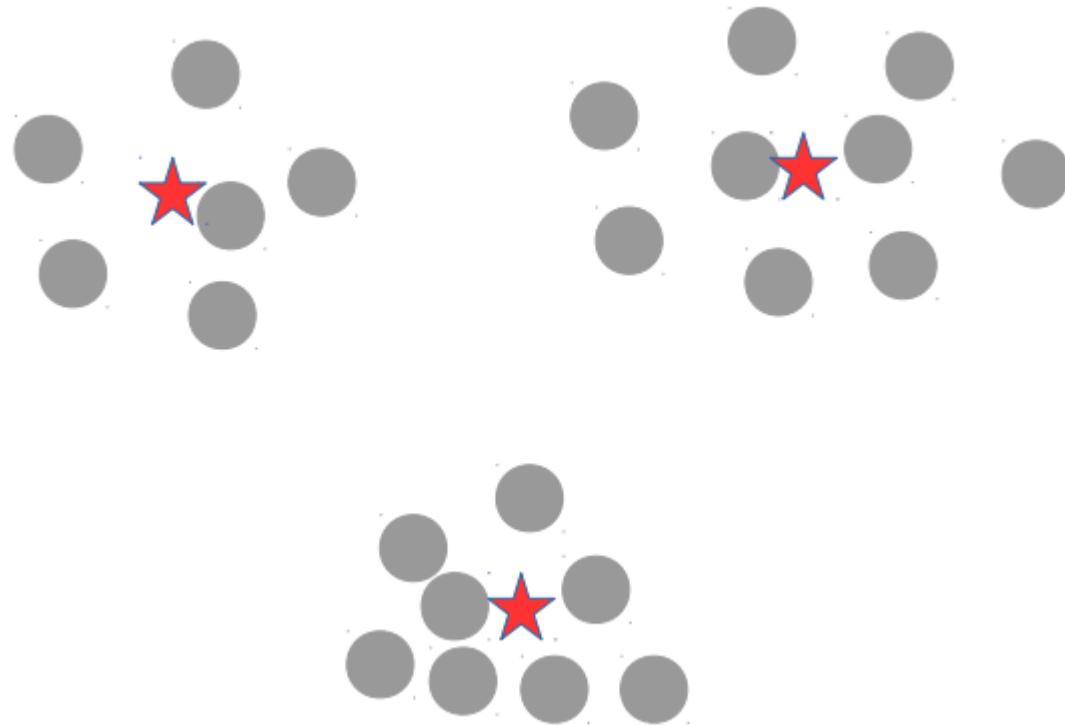
This basic K -means models each cluster by a single mean μ_k . Ignores size/shape of clusters



K-means: An Illustration



K-means = LwP with Unlabeled Data

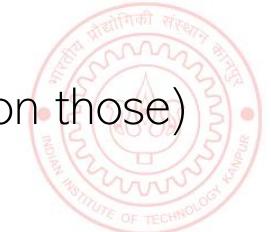


- Guess the means
- Predict the labels (cluster ids)
- Recompute the means using these predicted labels
- Repeat until not converged



The K -means Algorithm: Some Comments

- One of the most popular clustering algorithms
- Very widely used, guaranteed to converge (to a local minima; will see a proof)
- Can also be used as a sub-routine in graph clustering (in the Spectral Clustering algorithm): Inputs are given as an $N \times N$ adjacency matrix \mathbf{A} ($A_{nm} = 0/1$)
 - Perform a spectral decomposition of the graph Laplacian of \mathbf{A} to get \mathbf{F} (an $N \times K$ matrix)
 - Run K -means using rows of the \mathbf{F} matrix as the inputs
- Has some shortcomings but can be improved upon, e.g.,
 - Can be kernelized (using kernels or using kernel-based landmarks/random features)
 - More flexible cluster sizes/shapes via probabilistic models (e.g., every cluster is a Gaussian)
 - Soft-clustering (fractional/probabilistic memberships): \mathbf{z}_n is not one-hot but a probability vector
 - Overlapping clustering – an input can belong to multiple clusters: \mathbf{z}_n is a binary vector
 - .. even deep learning based K-means (use a deep network to extract features and run K -means on those)
- Let's look at K-means in some more detail now..



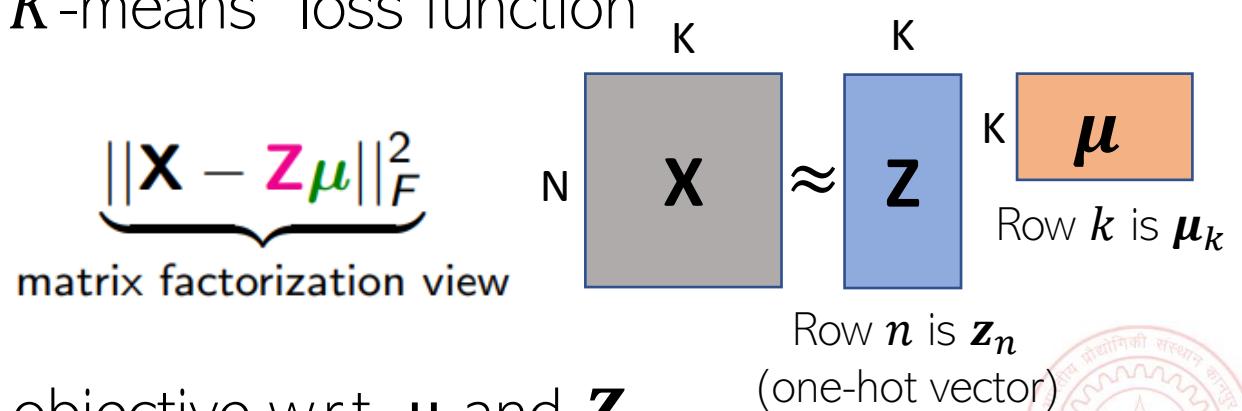
What Loss Function is K -means Optimizing?

- Let $\mu_1, \mu_2, \dots, \mu_K$ be the K cluster centroids/means
- Let $z_{nk} \in \{0, 1\}$ be s.t. $z_{nk} = 1$ if x_n belongs to cluster k , and 0 otherwise
- Define the distortion or “loss” for the cluster assignment of x_n

$$\ell(\mu, x_n, z_n) = \sum_{k=1}^K z_{nk} \|x_n - \mu_k\|^2$$

- Total distortion over all points defines the K -means “loss function”

$$L(\mu, X, Z) = \sum_{n=1}^N \sum_{k=1}^K z_{nk} \|x_n - \mu_k\|^2 = \underbrace{\|X - Z\mu\|_F^2}_{\text{matrix factorization view}}$$



- The K -means problem is to minimize this objective w.r.t. μ and Z
 - Alternating optimization on this loss would give the K -means (Lloyd’s) algorithm we saw earlier!



K-means Loss: Several Forms, Same Meaning!

- Notation: \mathbf{X} is $N \times D$
- \mathbf{Z} is $N \times K$ (each row is a one-hot z_n or equivalently $z_n \in \{1, 2, \dots, K\}$)
- $\boldsymbol{\mu}$ is $K \times D$ (each row is a $\boldsymbol{\mu}_k$)

Replacing the ℓ_2 (Euclidean) squared by ℓ_1 distances gives the K-medoids algorithm (more robust to outliers)



$$\mathcal{L}(\mathbf{X}, \mathbf{Z}, \boldsymbol{\mu}) = \sum_{n=1}^N \|\mathbf{x}_n - \boldsymbol{\mu}_{z_n}\|^2$$

Distortion on assigning
 \mathbf{x}_n to cluster z_n

$$\mathcal{L}(\mathbf{X}, \mathbf{Z}, \boldsymbol{\mu}) = \sum_{k=1}^K \underbrace{\sum_{n: z_n=k} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2}_{\text{within cluster variance}}$$

$$\mathcal{L}(\mathbf{X}, \mathbf{Z}, \boldsymbol{\mu}) = \sum_{n=1}^N \sum_{k=1}^K z_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2$$

$$\mathcal{L}(\mathbf{X}, \mathbf{Z}, \boldsymbol{\mu}) = \underbrace{\|\mathbf{X} - \mathbf{Z}\boldsymbol{\mu}\|_F^2}$$

as matrix factorization

$$\{\hat{\mathbf{Z}}, \hat{\boldsymbol{\mu}}\} = \arg \min_{\mathbf{Z}, \boldsymbol{\mu}} \mathcal{L}(\mathbf{X}, \mathbf{Z}, \boldsymbol{\mu})$$

Total “distortion” or
reconstruction error

- Note: Most unsup. learning algos try to minimize a distortion or recon error



Optimizing the K -means Loss Function

- So the K -means problem is

$$\{\hat{\mathbf{Z}}, \hat{\boldsymbol{\mu}}\} = \arg \min_{\mathbf{Z}, \boldsymbol{\mu}} \mathcal{L}(\mathbf{X}, \mathbf{Z}, \boldsymbol{\mu}) = \arg \min_{\mathbf{Z}, \boldsymbol{\mu}} \sum_{n=1}^N \sum_{k=1}^K z_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2$$

- Can't optimize it jointly for \mathbf{Z} and $\boldsymbol{\mu}$. Let's try alternating optimization for \mathbf{Z} and $\boldsymbol{\mu}$

Alternating Optimization for K -means Problem

- Fix $\boldsymbol{\mu}$ as $\hat{\boldsymbol{\mu}}$ and find the optimal \mathbf{Z} as

$$\hat{\mathbf{Z}} = \arg \min_{\mathbf{Z}} \mathcal{L}(\mathbf{X}, \mathbf{Z}, \hat{\boldsymbol{\mu}}) \quad (\text{still not easy - next slide})$$

- Fix \mathbf{Z} as $\hat{\mathbf{Z}}$ and find the optimal $\boldsymbol{\mu}$ as

$$\hat{\boldsymbol{\mu}} = \arg \min_{\boldsymbol{\mu}} \mathcal{L}(\mathbf{X}, \hat{\mathbf{Z}}, \boldsymbol{\mu})$$

- Go to step 1 if not yet converged



Solving for Z

- Solving for \mathbf{Z} with $\boldsymbol{\mu}$ fixed at $\hat{\boldsymbol{\mu}}$

$$\hat{\mathbf{Z}} = \arg \min_{\mathbf{Z}} \mathcal{L}(\mathbf{X}, \mathbf{Z}, \hat{\boldsymbol{\mu}}) = \arg \min_{\mathbf{Z}} \sum_{n=1}^N \sum_{k=1}^K z_{nk} \|\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k\|^2$$

- Still not easy. \mathbf{Z} is discrete and above is an NP-hard problem
 - Combinatorial optimization: K^N possibilities for \mathbf{Z} ($N \times K$ matrix with one-hot rows)
- Greedy approach: Optimize \mathbf{Z} one row (\mathbf{z}_n) at a time keeping all others \mathbf{z}_n 's (and the cluster means $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \dots, \boldsymbol{\mu}_K$) fixed

$$\hat{\mathbf{z}}_n = \arg \min_{\mathbf{z}_n} \sum_{k=1}^K z_{nk} \|\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k\|^2 = \arg \min_{\mathbf{z}_n} \|\mathbf{x}_n - \hat{\boldsymbol{\mu}}_{z_n}\|^2$$

- Easy to see that this is minimized by assigning \mathbf{x}_n to the closest mean
 - This is exactly what the K -means (Lloyd's) algo does!



Solving for μ

- Solving for μ with \mathbf{Z} fixed at $\hat{\mathbf{Z}}$

$$\hat{\mu} = \arg \min_{\mu} \mathcal{L}(\mathbf{X}, \hat{\mathbf{Z}}, \mu) = \arg \min_{\mu} \sum_{k=1}^K \sum_{n: \hat{z}_n=k} \|\mathbf{x}_n - \mu_k\|^2$$

- Not difficult to solve (each μ_k is a real-valued vector, can optimize easily)

$$\hat{\mu}_k = \arg \min_{\mu_k} \sum_{n: \hat{z}_n=k} \|\mathbf{x}_n - \mu_k\|^2$$

- Note that each μ_k can be optimized for independently
- (Verify) This is minimized by setting $\widehat{\mu_k}$ to be mean of points currently in cluster k
 - This is exactly what the K -means (Lloyd's) algo does!



Convergence of K -means algorithm

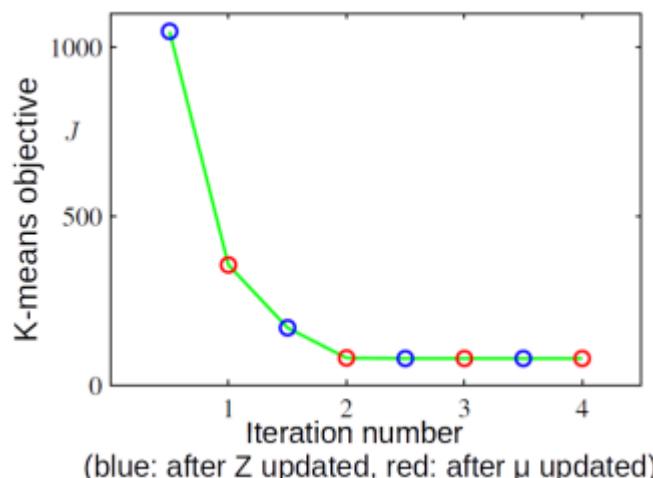
- Each step (updating \mathbf{Z} or $\boldsymbol{\mu}$) can never increase the K -means loss
- When we update \mathbf{Z} from $\mathbf{Z}^{(t-1)}$ to $\mathbf{Z}^{(t)}$

$$\mathcal{L}(\mathbf{X}, \mathbf{Z}^{(t)}, \boldsymbol{\mu}^{(t-1)}) \leq \mathcal{L}(\mathbf{X}, \mathbf{Z}^{(t-1)}, \boldsymbol{\mu}^{(t-1)}) \quad \text{because} \quad \mathbf{Z}^{(t)} = \arg \min_{\mathbf{Z}} \mathcal{L}(\mathbf{X}, \mathbf{Z}, \boldsymbol{\mu}^{(t-1)})$$

- When we update $\boldsymbol{\mu}$ from $\boldsymbol{\mu}^{(t-1)}$ to $\boldsymbol{\mu}^{(t)}$

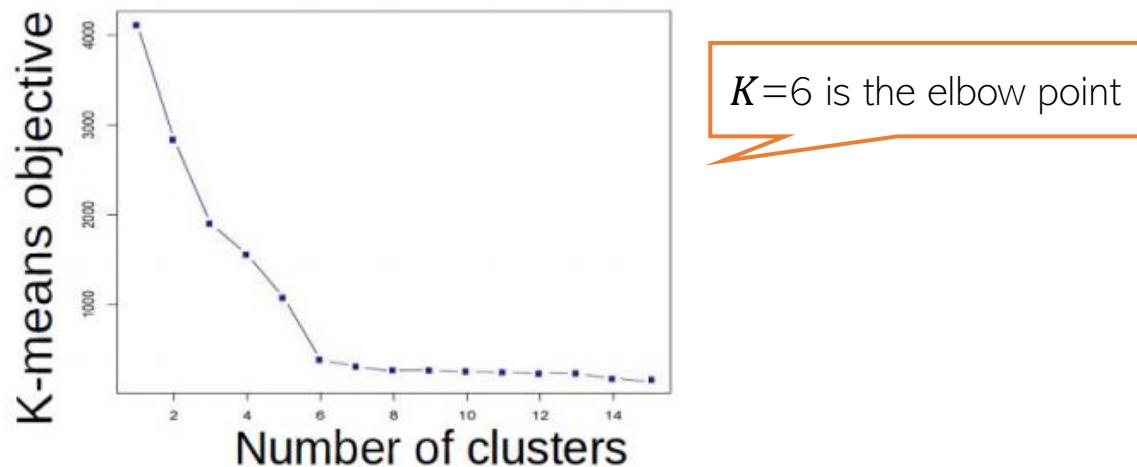
$$\mathcal{L}(\mathbf{X}, \mathbf{Z}^{(t)}, \boldsymbol{\mu}^{(t)}) \leq \mathcal{L}(\mathbf{X}, \mathbf{Z}^{(t)}, \boldsymbol{\mu}^{(t-1)}) \quad \text{because} \quad \boldsymbol{\mu}^{(t)} = \arg \min_{\boldsymbol{\mu}} \mathcal{L}(\mathbf{X}, \mathbf{Z}^{(t)}, \boldsymbol{\mu})$$

- Thus the K -means algorithm monotonically decreases the objective



K-means: Choosing K

- One way to select K for the K -means algorithm is to try different values of K , plot the K -means objective versus K , and look at the “elbow-point”



- Can also use information criterion such as AIC (Akaike Information Criterion)

$$AIC = 2\mathcal{L}(\hat{\mu}, \mathbf{X}, \hat{\mathbf{Z}}) + KD$$

and choose K which gives the **smallest AIC** (small loss + large K values penalized)

- More advanced approaches, such as nonparametric Bayesian methods (Dirichlet Process mixture models also used, not within K-means but with other clustering algos)



Coming up next

- Improvements to K-means
 - Soft-assignments
 - Handling complex cluster shapes (basic K-means assumes spherical clusters)
- Evaluating clustering algorithms (how to evaluate without true labels)
- Probabilistic approaches to clustering



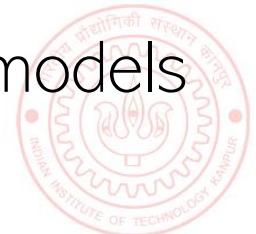
Data Clustering (Contd)

CS771: Introduction to Machine Learning

Piyush Rai

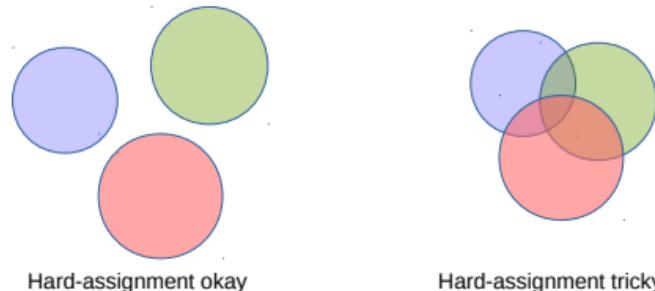
Plan

- K-means extensions
 - Soft clustering
 - Kernel K-means
- A few other popular clustering algorithms
 - Hierarchical Clustering
 - Agglomerative Clustering
 - Divisive Clustering
 - Graph Clustering
 - Spectral Clustering
 - Density-based clustering
 - DBSCAN
- Basic idea of probabilistic clustering methods, such as Gaussian mixture models
(details when we talk about latent variable models)



K-means: Hard vs Soft Clustering

- K-means makes hard assignments of points to clusters
 - Hard assignment: A point either completely belongs to a cluster or doesn't belong at all



A more principled extension of K-means for doing soft-clustering is via probabilistic mixture models such as the [Gaussian Mixture Model](#)

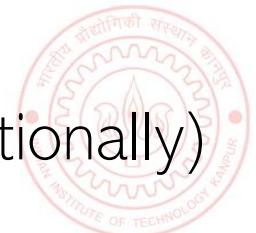


- When clusters overlap, soft assignment is preferable (i.e., probability of being assigned to each cluster: say $K = 3$ and for some point \mathbf{x}_n , $p_1 = 0.7, p_2 = 0.2, p_3 = 0.1$)
- A heuristic to get **soft assignments**: Transform distances from clusters into prob.

$$\sum_{k=1}^K \gamma_{nk} = 1$$

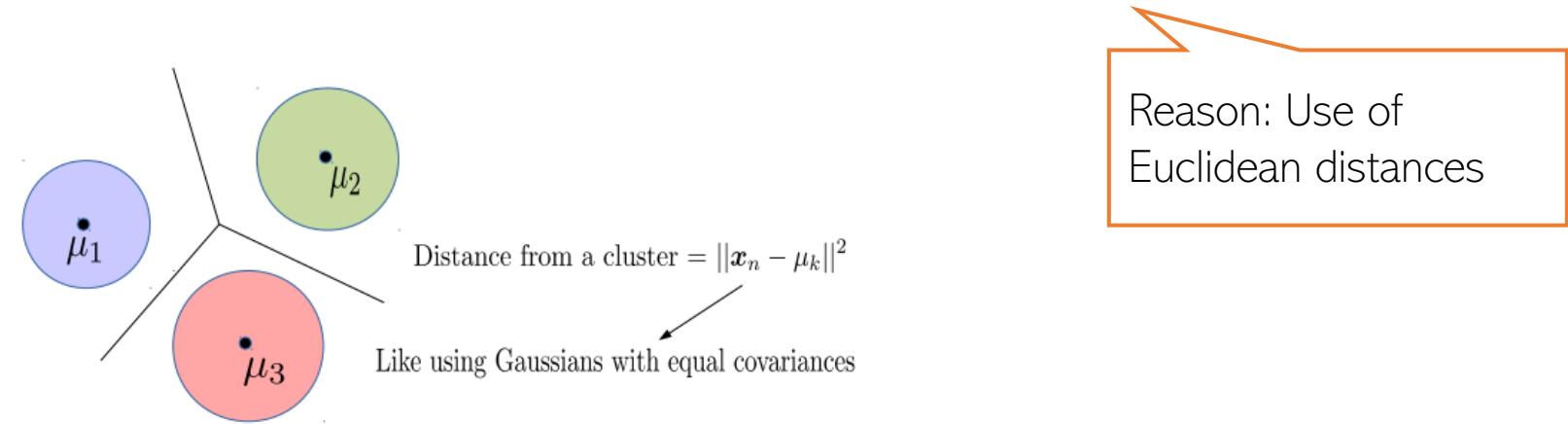
$$\gamma_{nk} = \frac{\exp(-\|\mathbf{x}_n - \mu_k\|^2)}{\sum_{\ell=1}^K \exp(-\|\mathbf{x}_n - \mu_\ell\|^2)} \quad (\text{prob. that } \mathbf{x}_n \text{ belongs to cluster } k)$$

- Cluster mean updates also change: $\mu_k = \frac{\sum_{n=1}^N \gamma_{nk} \mathbf{x}_n}{\sum_{n=1}^N \gamma_{nk}}$ (all points contribute, fractionally)

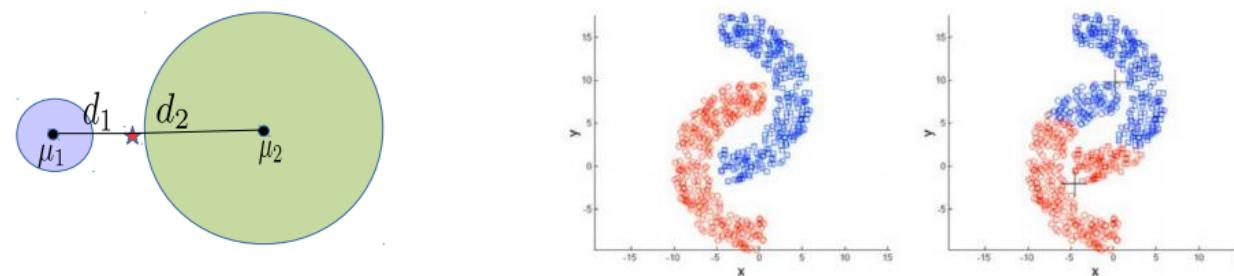


K-means: Decision Boundaries and Cluster Sizes/Shapes⁴

- K-mean assumes that the decision boundary between any two clusters is linear
- Reason: The K-means loss function implies assumes equal-sized, spherical clusters



- May do badly if clusters are not roughly equi-sized and convex-shaped



Kernel K-means

- Basic idea: Replace the Eucl. distances in K-means by the kernelized versions

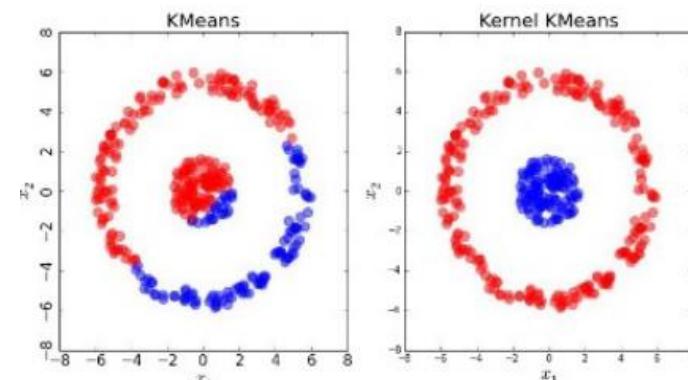
$$\begin{aligned} \|\phi(\mathbf{x}_n) - \phi(\boldsymbol{\mu}_k)\|^2 &= \|\phi(\mathbf{x}_n)\|^2 + \|\phi(\boldsymbol{\mu}_k)\|^2 - 2\phi(\mathbf{x}_n)^T \phi(\boldsymbol{\mu}_k) \\ &= k(\mathbf{x}_n, \mathbf{x}_n) + k(\boldsymbol{\mu}_k, \boldsymbol{\mu}_k) - 2k(\mathbf{x}_n, \boldsymbol{\mu}_k) \end{aligned}$$

Kernelized distance between input \mathbf{x}_n and mean of cluster k

- Here $k(\cdot, \cdot)$ denotes the kernel function and ϕ is its (implicit) feature map
- Note: $\phi(\boldsymbol{\mu}_k)$ is the mean of ϕ mappings of the data points assigned to cluster k

Not the same as the ϕ mapping of the mean of the data points assigned to cluster k

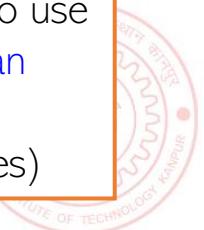
$$\phi(\boldsymbol{\mu}_k) = \frac{1}{|\mathcal{C}_k|} \sum_{n:z_n=k} \phi(\mathbf{x}_n)$$



Can also used landmarks or kernel random features idea to get new features and run standard k-means on those



Note: Apart from kernels, it is also possible to use other distance functions in K-means. [Bregman Divergence*](#) is such a family of distances (Euclidean and Mahalanobis are special cases)



Hierarchical Clustering

- Can be done in two ways: Agglomerative or Divisive

Similarity between two clusters (or two set of points) is needed in HC algos (e.g., this can be average pairwise similarity between the inputs in the two clusters)

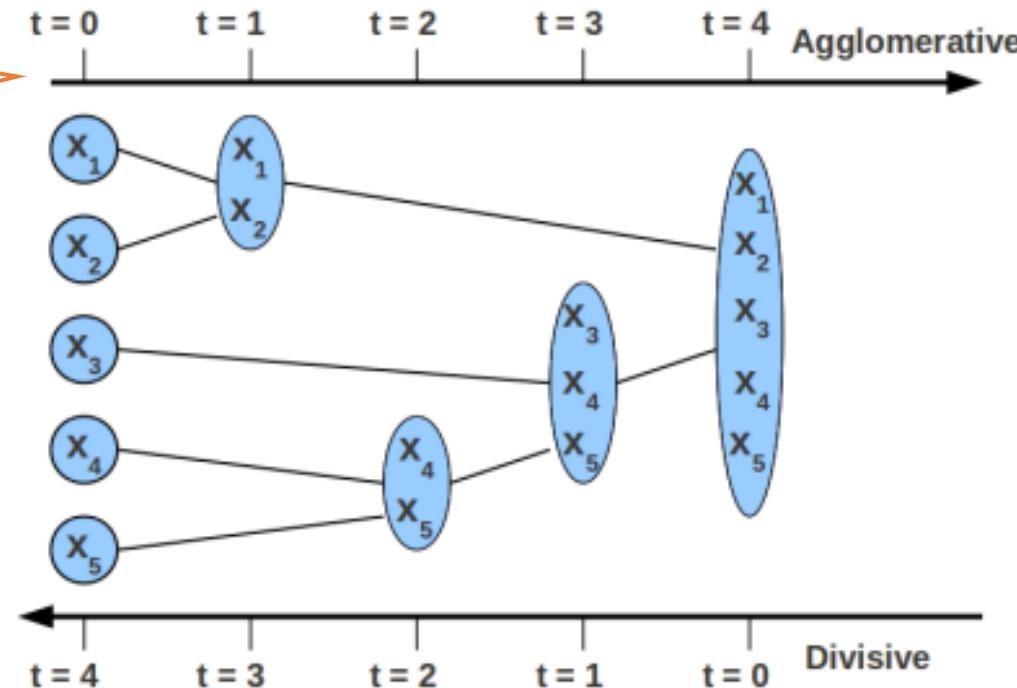


Agglomerative: Start with each point being in a singleton cluster

At each step, greedily merge two most "similar" sub-clusters

Stop when there is a single cluster containing all the points

Learns a dendrogram-like structure with inputs at the leaf nodes. **Can then choose how many clusters we want**



Keep recursing until the desired number of clusters found

At each step, break a cluster into (at least) two smaller homogeneous sub-clusters

Divisive: Start with all points being in a single cluster

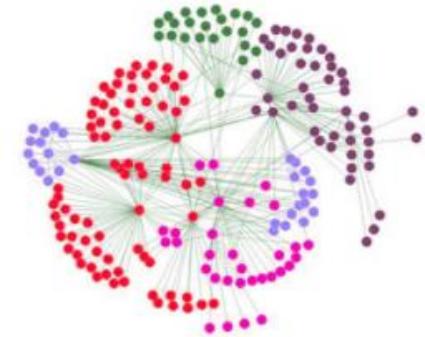
Tricky because no labels (unlike Decision Trees)

- Agglomerative is more popular and simpler than divisive (the latter usually needs complicated heuristics to decide cluster splitting).
- Neither uses any loss function



Graph Clustering

- Often the data is given in form of a graph, not feat. vec.
 - Usually in form of a pairwise similarity matrix \mathbf{A} of size $N \times N$
 - $A_{nm} \geq 0$ is assumed to be the similarity between two nodes/inputs with indices n and m
- Examples: Social networks and various interaction networks
- Goal is to cluster the nodes/inputs into K clusters (flat partitioning)
- One scheme is to somehow get an embedding of the graph nodes to get feature vector for each node and run K -means or kernel K -means or any other clustering algo
- Another way is to perform direct graph clustering
- **Spectral clustering** is such a popular graph clustering algorithm



Various graph embedding algorithms exist (e.g., node2vec)



Spectral Clustering

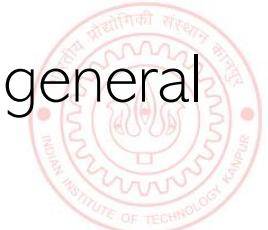
Spectral clustering has a beautiful theory behind it (won't get into it in this course; may refer to a very nice tutorial article listed below, if interested)



- We are given the node-node similarity matrix \mathbf{A} of size $N \times N$
- Compute the graph Laplacian $\mathcal{L} = \mathbf{D} - \mathbf{A}$
 - \mathbf{D} is a diagonal matrix s.t. $D_{nn} = \sum_{m=1}^N A_{nm}$ (sum of similarities of node n with all other nodes)
- Note: Often, we work with a normalized graph Laplacian $\mathcal{L}^{norm} = \mathbf{I} - \mathbf{D}^{-1/2} \mathbf{A} \mathbf{D}^{-1/2}$
- Given the graph Laplacian, solve this spectral decomposition problem

$$\mathbf{U} = \underset{\mathbf{U} \in \mathbb{R}^{N \times K}}{\operatorname{argmax}} \operatorname{trace}(\mathbf{U}^\top \mathcal{L} \mathbf{U}) \quad \text{s. t. } \mathbf{U}^\top \mathbf{U} = \mathbf{I}$$

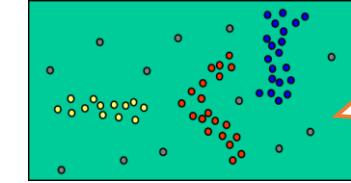
Meaning \mathbf{U} has orthonormal columns
- Now run K -means on the \mathbf{U} matrix as the feature matrix of the N nodes
- Note: Spectral clustering* is also closely related to kernel K -means (but more general since \mathbf{A} can represent any graph) and “normalized cuts” for graphs



Density based Clustering - DBSCAN

- DBSCAN: Density Based Spatial Clustering of Applications with Noise
 - Uses notion of density of points (not in the sense of probability density) around a point

DBSCAN treats densely connected points as a cluster, regardless of the shape of the cluster
- Has some very nice properties
 - Does not require specifying the number of clusters
 - Can learn arbitrary shaped clusters (since it only considers of density of points)
 - Robust against **outliers** (leaves them unclustered!), unlike other clust. algos like *K*-means

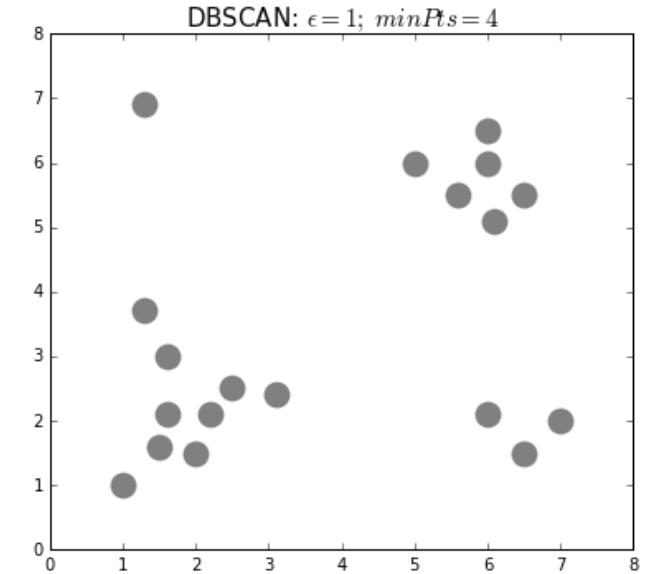
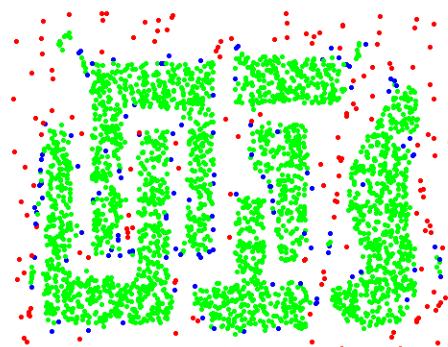
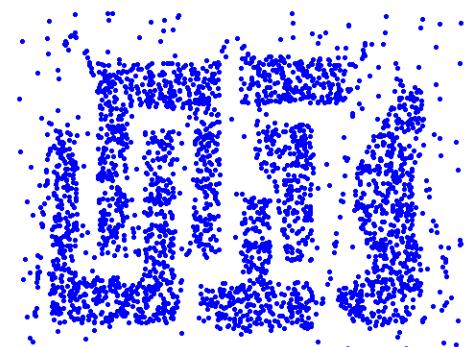

 Grey points left unclustered since they are most likely outliers
- Basic idea in DBSCAN is as follows
 - Want all points within a cluster to be at most **ϵ distance** apart from each other

Accuracy of DBSCAN depends crucially on ϵ and minPoint hyperparams
 - Want at least **minPoints** points within ϵ distance of a point (such a point is called “**core**” point)
 - Points that don’t have minPoints within ϵ distance are called “**border**” points
 - Points that are neither core nor border point are **outliers**



DBSCAN (Contd)

- The animation on the right shows DBSCAN in action
- The basic algorithm is as follows
 - A point is chosen at random
 - If more than minPoint neighbors $\leq \epsilon$ distance, then call it core point
 - Check if more points fall within ϵ distance of core/its neighbors
 - If yes, include them too in the same cluster
 - Once done with this cluster, pick another point randomly and repeat
- An example of clustering obtained by DBSCAN



Green points are core points,
blue points are border points, red
points are outliers

DBSCAN is mostly a heuristic
based algorithm. No loss
function unlike K-means



Going the Probabilistic Way..

- Assume a generative model for inputs and Θ denotes all the unknown params
- Clustering then boils down to computing posterior cluster probability $p(\mathbf{z}_n | \mathbf{x}_n, \Theta)$ where $\mathbf{z}_n \in \{1, 2, \dots, K\}$ denote the cluster assignment of \mathbf{x}_n

$$p(\mathbf{z}_n = k | \mathbf{x}_n, \Theta) = \frac{p(\mathbf{z}_n = k | \Theta) p(\mathbf{x}_n | \mathbf{z}_n = k, \Theta)}{p(\mathbf{x}_n | \Theta)} \quad (\text{from Bayes rule})$$

- Assuming prior $p(\mathbf{z}_n | \Theta)$ to be multinoulli with prob. vector $\pi = [\pi_1, \pi_2, \dots, \pi_K]$ and each of the class-conditional $p(\mathbf{x}_n | \mathbf{z}_n = k, \Theta)$ to be a Gaussian $\mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$

$$p(\mathbf{z}_n = k | \mathbf{x}_n, \Theta) \propto \pi_k \times \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad (\text{Here } \Theta = \{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^K)$$

Posterior prob. Of cluster assignment also depends on prior probability (fraction of points in that cluster if using MLE)

Different clusters can have different covariances (hence different shapes)

- We know how to estimate Θ if \mathbf{z}_n were known (recall generative classification)
- But since we don't know \mathbf{z}_n , need to estimate both (and ALT-OPT can be used)

Just like in K-means



Going the Probabilistic Way..

- At a high-level, a probabilistic clustering algorithm would look somewhat like this

Sketch of a Probabilistic Clustering Algorithm

- ① Initialize the model parameters Θ somehow Akin to initializing the cluster means in K-means
- ② Given the current Θ , estimate \mathbf{Z} (cluster assignments) in a soft/hard way Akin to computing cluster assignments in K-means

$$p(\mathbf{z}_n = k | \mathbf{x}_n, \Theta) = \gamma_{nk} = \frac{p(\mathbf{z}_n = k | \Theta)p(\mathbf{x}_n | \mathbf{z}_n = k, \Theta)}{p(\mathbf{x}_n | \Theta)}, \quad k = 1, \dots, K$$

OR

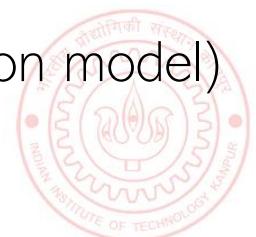
$$\hat{\mathbf{z}}_n = \arg \max_{k \in \{1, \dots, K\}} \gamma_{nk}$$
Akin to updating cluster means in K-means
- ③ Use $\{\hat{\mathbf{z}}_n\}_{n=1}^N$ (hard cluster labels) or $\{\gamma_{nk}\}_{n,k=1}^{N,K}$ (soft labels) to update Θ via MLE/MAP (similar to how we do for gen. classification where the labels are known)
- ④ Note: The soft-label based Θ updates slightly more involved (wait until we see EM)
- ⑤ Go to step 2 if not converged yet.

- The above algorithm is an instance of a more general Expectation Maximization (EM) algorithm for latent variable models (we will see this soon)



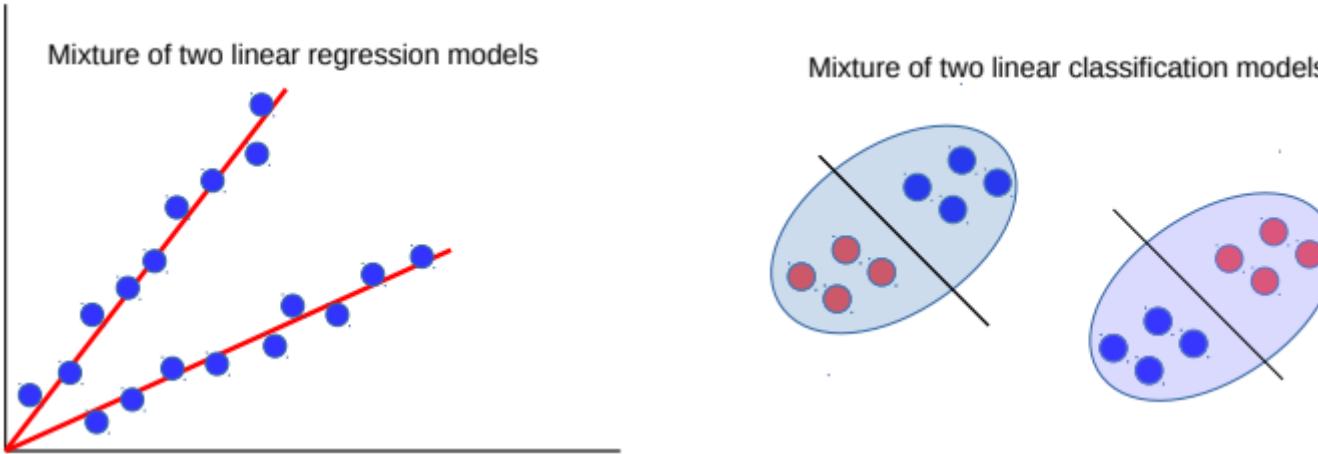
Clustering vs Classification

- Any clustering model (prob/non-prob) typically learns two type of quantities
 - Parameters Θ of the clustering model (e.g., cluster means in K-means)
 - Cluster assignments $\mathbf{Z} = \{\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_N\}$ for the points
- If cluster assignments \mathbf{Z} were known, learning the parameters Θ is just like learning the parameters of a classifn model (typically generative classification) using labeled data
- Thus helps to think of clustering as (generative) classification with unknown labels
- Therefore many clustering problems are typically solved in the following fashion
 1. Initialize Θ somehow
 2. Predict Z given current estimate of Θ
 3. Use the predicted Z to improve the estimate of Θ (like learning a generative classification model)
 4. Go to step 2 if not converged yet



Clustering can help supervised learning, too

- Often “difficult” sup. learning problems can be seen as mixture of simpler models
- Example: Nonlinear regression or nonlinear classification as mixture of linear models



- Don't know which point should be modeled by which linear model \Rightarrow Clustering
- Can therefore solve such problems as follows
 - Initialize each linear model somehow (maybe randomly)
 - Cluster the data by assigning each point to its “closest” linear model (one that gives lower error)
 - (Re-)Learn a linear model for each cluster’s data. Go to step 2 if not converged.

Such an approach is also an example of divide and conquer and is also known as “mixture of experts” (will see it more formally when we discuss latent variable models)



Coming up next

- Latent Variable Models
 - Mixture models using latent variables
 - Expectation Maximization algorithm



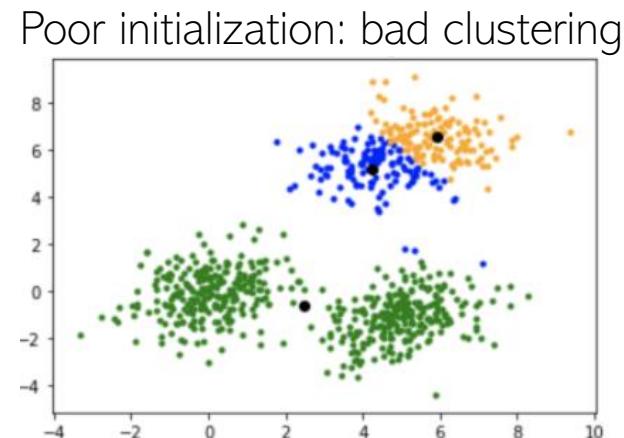
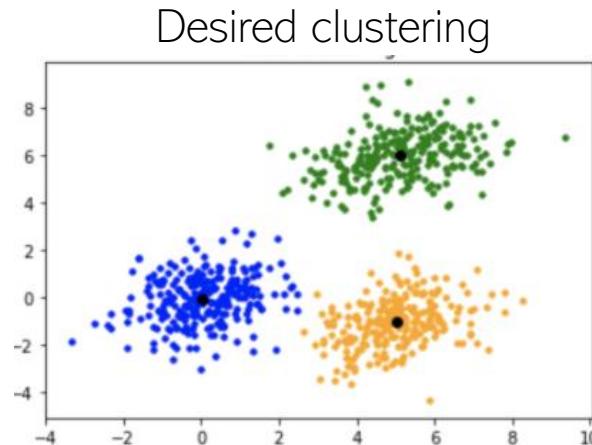
Data Clustering: Some Other Aspects (K-means++, Overlapping Clustering, Evaluation)

CS771: Introduction to Machine Learning

Piyush Rai

K-means++

- K -means results can be sensitive to initialization



- K -means++ (Arthur and Vassilvitskii, 2007) an improvement over K -means
 - Only difference is the way we initialize the cluster centers (rest of it is just K -means)
 - Basic idea: Initialize cluster centers such that they are reasonably far from each other
 - Note: In K -means++, the cluster centers are chosen to be K of the data points themselves



K-means++

- K-means++ works as follows
 - Choose the first cluster mean uniformly randomly to be one of the data points
 - The subsequent $K - 1$ cluster means are chosen as follows
 1. For each unselected point \mathbf{x} , compute its smallest distance $D(\mathbf{x})$ from already initialized means
 2. Select the next cluster mean unif. rand. to be one of the unselected points based on probability prop. to $D(\mathbf{x})^2$
 3. Repeat 1 and 2 until the $K - 1$ cluster means are initialized
 - Now run standard K-means with these initial cluster means
 - K-means++ initialization scheme sort of ensures that the initial cluster means are located in different clusters

Thus farthest points are most likely to be selected as cluster means



Overlapping Clustering

- Have seen hard clustering and soft clustering
- In hard clustering, z_n is a one-hot vector
- In soft clustering, z_n is a vector of probabilities
- Overlapping Clustering: A point can simultaneously belong to multiple clusters
 - This is different from soft-clustering
 - z_n would be a **binary vector**, rather than a one hot or probability vector, e.g.,

$$z_n = [1 \ 0 \ 0 \ 1 \ 0]$$

K=5 clusters with point x_n belonging (in whole, not in terms of probabilities) to clusters 1 and 4

- In general, more difficult than hard/soft clustering (for N data points and K clusters, the size of the space of possible solutions is not K^N but 2^{NK} - exp in both N and K)
- K-means has extensions* for doing overlapping clustering. There also exist latent variable models for doing overlapping clustering

*An extended version of the k-means method for overlapping clustering (Cleuziou, 2008); Non-exhaustive, Overlapping k-means (Whang et al, 2015)  E3771: Intro to ML

Evaluating Clustering Algorithms

- Clustering algos are in general harder to evaluate since we rarely know the ground truth clustering (since clustering is unsupervised)
- If ground truth labels not available, use output of clustering for some other task
 - For example, use cluster assignment z_n (hard or soft) as a new feature representation
 - Performance on some task using this new rep. is a measure of goodness of clustering
- If ground truth labels are available, can compare them with clustering based labels
 - Not straightforward to compute accuracy since the label identities may not be the same, e.g.,

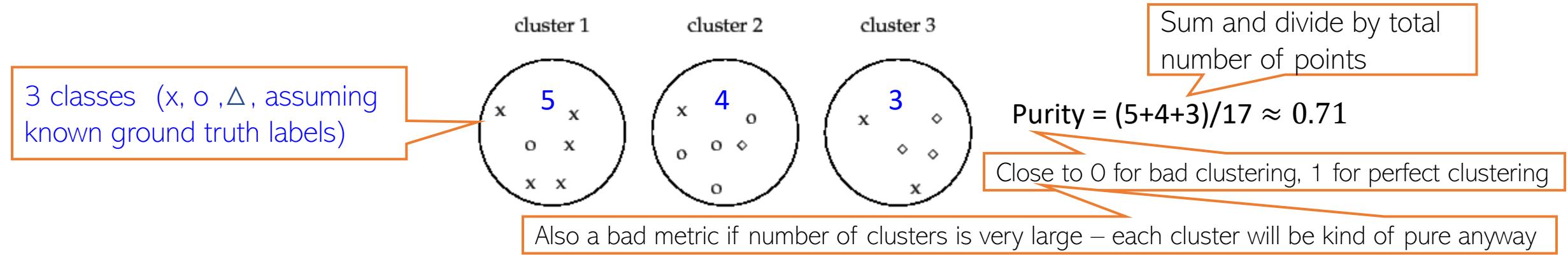
Ground truth = [1 1 1 0 0 0] Clustering = [0 0 0 1 1 1]

(Perfect clustering but zero “accuracy” if we just do a direct match)
 - There are various metrics that take into account the above fact
 - Purity, Rand Index, F-score, Normalized Mutual Information, etc



Evaluating Clustering Algorithms

- Purity: Looks at how many points in each cluster belong to the majority class in that cluster



- Rand Index (RI): Can also look at what fractions of pairs of points with same (resp. different) label are assigned to same (resp. different) cluster

F_β score is also popular

$$P = \frac{TP}{TP + FP} \quad R = \frac{TP}{TP + FN} \quad F_\beta = \frac{(\beta^2 + 1)PR}{\beta^2P + R}$$

Precision Recall

True Positive: No. of pairs with same true label and same cluster

True Negative: No. of pairs with diff true label and diff clusters

False Positive: No. of pairs with diff true label and same cluster

False Negative: No. of pairs with same true label and diff cluster

$$RI = \frac{TP + TN}{TP + FP + FN + TN}$$



Coming up next

- Latent variable models



Dimensionality Reduction: Principal Component Analysis and SVD

CS771: Introduction to Machine Learning

Piyush Rai

Dimensionality Reduction

- A broad class of techniques
 - Goal is to compress the original representation of the inputs
 - Example: Approximate each input $\mathbf{x}_n \in \mathbb{R}^D$, $n = 1, 2, \dots, N$ as a linear combination of $K < \min\{D, N\}$ “basis” vectors $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K$, each also $\in \mathbb{R}^D$
 - We have represented each $\mathbf{x}_n \in \mathbb{R}^D$ by a K -dim vector \mathbf{z}_n (a new feat. rep)
 - To store N such inputs $\{\mathbf{x}_n\}_{n=1}^N$, we need to keep \mathbf{W} and $\{\mathbf{z}_n\}_{n=1}^N$
 - Originally we required $N \times D$ storage, now $N \times K + D \times K = (N + D) \times K$ storage
 - If $K \ll \min\{D, N\}$, this yields substantial storage saving, hence good compression
- Note: These “basis” vectors need not necessarily be linearly independent. But for some dim. red. techniques, e.g., classic principal component analysis (PCA), they are
- Can think of \mathbf{W} as a linear mapping that transforms low-dim \mathbf{z}_n to high-dim \mathbf{x}_n
- Some dim-red techniques assume a nonlinear mapping function f such that $\mathbf{x}_n = f(\mathbf{z}_n)$
- For example, f can be modeled by a kernel or a deep neural net
- 

$$\mathbf{x}_n \approx \sum_{k=1}^K z_{nk} \mathbf{w}_k = \mathbf{W} \mathbf{z}_n$$

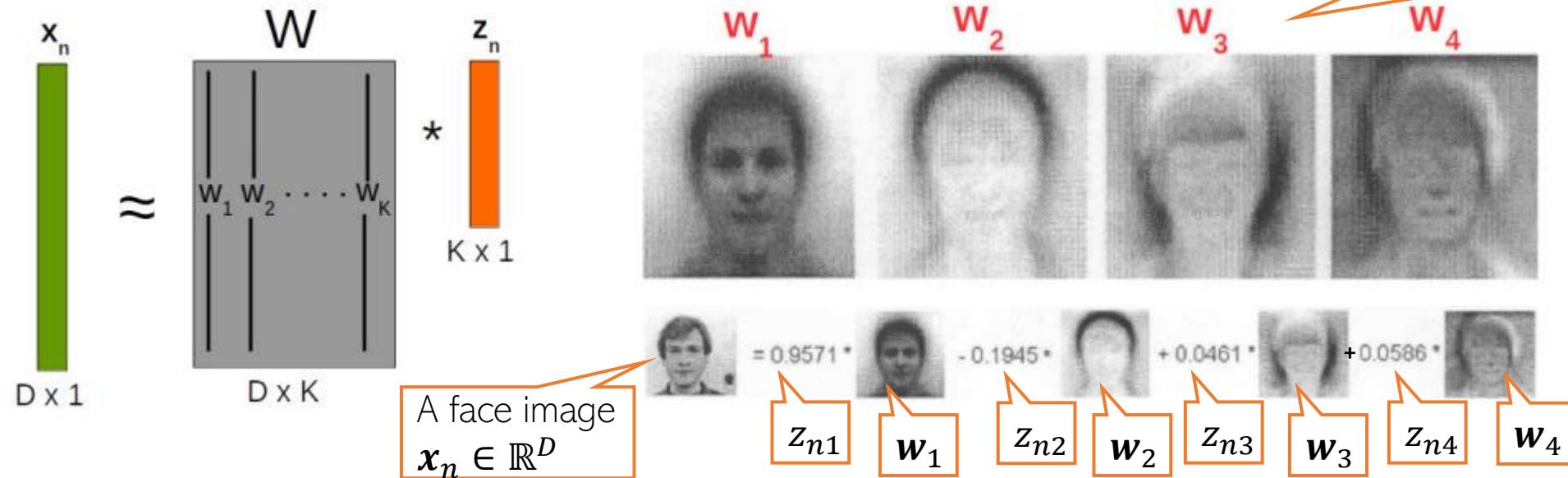
$\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K]$ is $D \times K$

$\mathbf{z}_n = [z_{n1}, z_{n2}, \dots, z_{nK}]$ is $K \times 1$

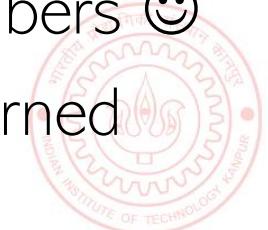


Dimensionality Reduction

- Dim-red for face images



- In this example, $\mathbf{z}_n \in \mathbb{R}^K$ ($K = 4$) is a low-dim feature rep. for $\mathbf{x}_n \in \mathbb{R}^D$ Like 4 new features
- Essentially, each face image in the dataset now represented by just 4 real numbers 😊
- Different dim-red algos differ in terms of how the basis vectors are defined/learned
 - .. And in general, how the function f in the mapping $\mathbf{x}_n = f(\mathbf{z}_n)$ is defined



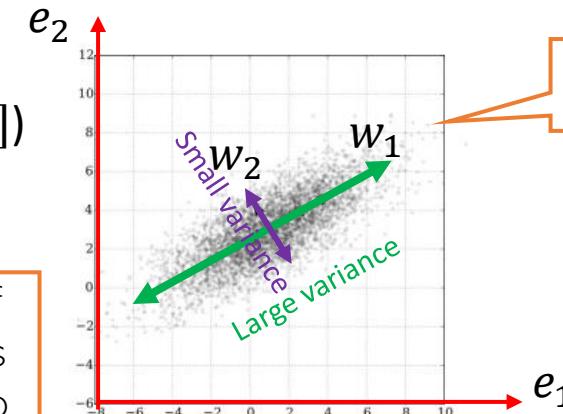
Principal Component Analysis (PCA)

- A classic linear dim. reduction method (Pearson, 1901; Hotelling, 1930)
- Can be seen as
 - Learning directions (co-ordinate axes) that capture maximum variance in data

e_1, e_2 : Standard co-ordinate axis ($x = [x_1, x_2]$)

w_1, w_2 : New co-ordinate axis ($z = [z_1, z_2]$)

To reduce dimension, can only keep the co-ordinates of those directions that have largest variances (e.g., in this example, if we want to reduce to one-dim, we can keep the co-ordinate z_1 of each point along w_1 and throw away z_2). We won't lose much information



PCA is essentially doing a change of axes in which we are representing the data

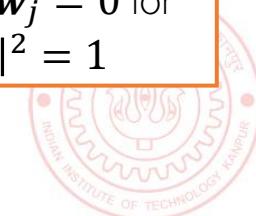
Each input will still have 2 co-ordinates, in the new co-ordinate system, equal to the distances measured from the new origin

- Learning projection directions that result in smallest reconstruction error

$$\operatorname{argmin}_{\mathbf{W}, \mathbf{Z}} \sum_{n=1}^N \|\mathbf{x}_n - \mathbf{W}\mathbf{z}_n\|^2 = \operatorname{argmin}_{\mathbf{W}, \mathbf{Z}} \|\mathbf{X} - \mathbf{Z}\mathbf{W}\|^2$$

Subject to orthonormality constraints: $\mathbf{w}_i^T \mathbf{w}_j = 0$ for $i \neq j$ and $\|\mathbf{w}_i\|^2 = 1$

- PCA also assumes that the projection directions are orthonormal

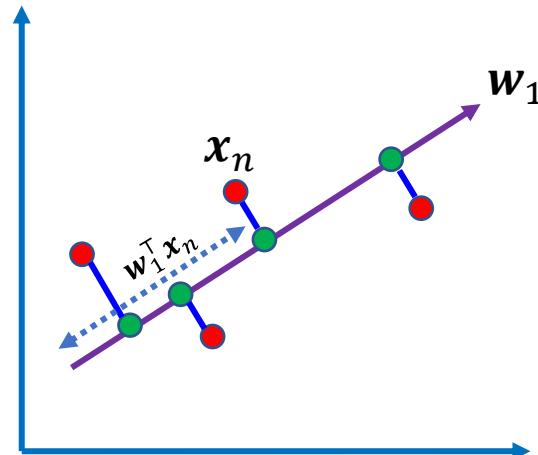


PCA: From the variance perspective



Solving PCA by Finding Max. Variance Directions

- Consider projecting an input $\mathbf{x}_n \in \mathbb{R}^D$ along a direction $\mathbf{w}_1 \in \mathbb{R}^D$
- Projection/embedding of \mathbf{x}_n (red points below) will be $\mathbf{w}_1^\top \mathbf{x}_n$ (green pts below)



Mean of projections of all inputs:

$$\frac{1}{N} \sum_{n=1}^N \mathbf{w}_1^\top \mathbf{x}_n = \mathbf{w}_1^\top \left(\frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \right) = \mathbf{w}_1^\top \boldsymbol{\mu}$$

Variance of the projections:

$$\frac{1}{N} \sum_{n=1}^N (\mathbf{w}_1^\top \mathbf{x}_n - \mathbf{w}_1^\top \boldsymbol{\mu})^2 = \frac{1}{N} \sum_{n=1}^N \{ \mathbf{w}_1^\top (\mathbf{x}_n - \boldsymbol{\mu}) \}^2 = \mathbf{w}_1^\top \mathbf{S} \mathbf{w}_1$$

- Want \mathbf{w}_1 such that variance $\mathbf{w}_1^\top \mathbf{S} \mathbf{w}_1$ is maximized

$$\operatorname{argmax}_{\mathbf{w}_1} \mathbf{w}_1^\top \mathbf{S} \mathbf{w}_1 \quad \text{s.t. } \mathbf{w}_1^\top \mathbf{w}_1 = 1$$

Need this constraint otherwise the objective's max will be infinity

\mathbf{S} is the $D \times D$ cov matrix of the data:

$$\mathbf{S} = \frac{1}{N} \sum_{n=1}^N (\mathbf{x}_n - \boldsymbol{\mu})(\mathbf{x}_n - \boldsymbol{\mu})^\top$$

For already centered data, $\boldsymbol{\mu} = \mathbf{0}$ and

$$\mathbf{S} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top = \frac{1}{N} \mathbf{X} \mathbf{X}^\top$$

Max. Variance Direction

- Our objective function was $\underset{\mathbf{w}_1}{\operatorname{argmax}} \mathbf{w}_1^\top \mathbf{S} \mathbf{w}_1$ s.t. $\mathbf{w}_1^\top \mathbf{w}_1 = 1$
- Can construct a Lagrangian for this problem

$$\underset{\mathbf{w}_1}{\operatorname{argmax}} \mathbf{w}_1^\top \mathbf{S} \mathbf{w}_1 + \lambda_1(1 - \mathbf{w}_1^\top \mathbf{w}_1)$$

- Taking derivative w.r.t. \mathbf{w}_1 and setting to zero gives $\mathbf{S} \mathbf{w}_1 = \lambda_1 \mathbf{w}_1$

Variance along the direction \mathbf{w}_1

Note: Total variance of the data is equal to the sum of eigenvalues of \mathbf{S} , i.e., $\sum_{d=1}^D \lambda_d$



PCA would keep the top $K < D$ such directions of largest variances

- Therefore \mathbf{w}_1 is an **eigenvector** of the cov matrix \mathbf{S} with eigenvalue λ_1
- Claim:** \mathbf{w}_1 is the eigenvector of \mathbf{S} with largest eigenvalue λ_1 . Note that

$$\mathbf{w}_1^\top \mathbf{S} \mathbf{w}_1 = \lambda_1 \mathbf{w}_1^\top \mathbf{w}_1 = \lambda_1$$

- Thus variance $\mathbf{w}_1^\top \mathbf{S} \mathbf{w}_1$ will be max. if λ_1 is the largest eigenvalue (and \mathbf{w}_1 is the corresponding top eigenvector; also known as the first **Principal Component**)
- Other large variance directions can also be found likewise (with each being orthogonal to all others) using the eigendecomposition of cov matrix \mathbf{S} (this is PCA)

Note: In general, \mathbf{S} will have D eigvecs



PCA: From the reconstruction perspective



Alternate Basis and Reconstruction

- Representing a data point $\mathbf{x}_n = [x_{n1}, x_{n2}, \dots, x_{nD}]^\top$ in the standard orthonormal basis $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_D\}$

$$\mathbf{x}_n = \sum_{d=1}^D z_{nd} \mathbf{e}_d$$

\mathbf{e}_d is a vector of all zeros except a single 1 at the d^{th} position. Also, $\mathbf{e}_d^\top \mathbf{e}_{d'} = 0$ for $d \neq d'$

- Let's represent the same data point in a new orthonormal basis $\{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_D\}$

z_{nd} is the projection of \mathbf{x}_n along the direction \mathbf{w}_d since $z_{nd} = \mathbf{w}_d^\top \mathbf{x}_n = \mathbf{x}_n^\top \mathbf{w}_d$ (verify)

$$\mathbf{x}_n = \sum_{d=1}^D z_{nd} \mathbf{w}_d$$

$\mathbf{z}_n = [z_{n1}, z_{n2}, \dots, z_{nD}]^\top$ denotes the co-ordinates of \mathbf{x}_n in the new basis

- Ignoring directions along which projection z_{nd} is small, we can approximate \mathbf{x}_n as

$$\mathbf{x}_n \approx \hat{\mathbf{x}}_n = \sum_{d=1}^K z_{nd} \mathbf{w}_d = \sum_{d=1}^K (\mathbf{x}_n^\top \mathbf{w}_d) \mathbf{w}_d = \sum_{d=1}^K (\mathbf{w}_d \mathbf{w}_d^\top) \mathbf{x}_n$$

Note that $\|\mathbf{x}_n - \sum_{d=1}^K (\mathbf{w}_d \mathbf{w}_d^\top) \mathbf{x}_n\|^2$ is the reconstruction error on \mathbf{x}_n . Would like it to minimize w.r.t. $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K$

- Now \mathbf{x}_n is represented by $K < D$ dim. rep. $\mathbf{z}_n = [z_{n1}, z_{n2}, \dots, z_{nK}]$ and (verify)

Also, $\mathbf{x}_n \approx \mathbf{W}_K \mathbf{z}_n$

$$\mathbf{z}_n \approx \mathbf{W}_K^\top \mathbf{x}_n$$

$\mathbf{W}_K = [\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K]$ is the "projection matrix" of size $D \times K$



Minimizing Reconstruction Error

- We plan to use only K directions $[\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K]$ so would like them to be such that the total reconstruction error is minimized

$$\mathcal{L}(\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K) = \sum_{n=1}^N \|\mathbf{x}_n - \hat{\mathbf{x}}_n\|^2 = \sum_{n=1}^N \left\| \mathbf{x}_n - \sum_{d=1}^K (\mathbf{w}_d \mathbf{w}_d^\top) \mathbf{x}_n \right\|^2 = C - \sum_{d=1}^K \mathbf{w}_d^\top \mathbf{S} \mathbf{w}_d \quad (\text{verify})$$

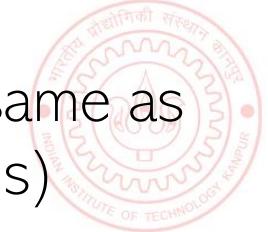
Constant; doesn't depend on the \mathbf{w}_d 's

Variance along \mathbf{w}_d

- Each optimal \mathbf{w}_d can be found by solving

$$\operatorname{argmin}_{\mathbf{w}_d} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K) = \operatorname{argmax}_{\mathbf{w}_d} \mathbf{w}_d^\top \mathbf{S} \mathbf{w}_d$$

- Thus minimizing the reconstruction error is equivalent to maximizing variance
- The K directions can be found by solving the eigendecomposition of \mathbf{S}
- Note: $\sum_{d=1}^K \mathbf{w}_d^\top \mathbf{S} \mathbf{w}_d = \operatorname{trace}(\mathbf{W}_K^\top \mathbf{S} \mathbf{W}_K)$
 - Thus $\operatorname{argmax}_{\mathbf{W}_K} \operatorname{trace}(\mathbf{W}_K^\top \mathbf{S} \mathbf{W}_K)$ s.t. orthonormality on columns of \mathbf{W}_k is the same as solving the eigendec. of \mathbf{S} (recall that Spectral Clustering also required solving this)



Principal Component Analysis

- Center the data (subtract the mean $\boldsymbol{\mu} = \frac{1}{N} \sum_{n=1}^N \boldsymbol{x}_n$ from each data point)
- Compute the $D \times D$ covariance matrix \mathbf{S} using the centered data matrix \mathbf{X} as

$$\mathbf{S} = \frac{1}{N} \mathbf{X}^\top \mathbf{X} \quad (\text{Assuming } \mathbf{X} \text{ is arranged as } N \times D)$$

- Do an eigendecomposition of the covariance matrix \mathbf{S} (many methods exist)
- Take top $K < D$ leading eigenvectors $\{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K\}$ with eigenvalues $\{\lambda_1, \lambda_2, \dots, \lambda_K\}$
- The K -dimensional projection/embedding of each input is

$$\mathbf{z}_n \approx \mathbf{W}_K^\top \mathbf{x}_n$$

$\mathbf{W}_K = [\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K]$ is the “projection matrix” of size $D \times K$

Note: Can decide how many eigvecs to use based on how much variance we want to capture (recall that each λ_k gives the variance in the k^{th} direction (and their sum is the total variance))



Singular Value Decomposition (SVD)

- Any matrix \mathbf{X} of size $N \times D$ can be represented as the following decomposition

$$\begin{matrix} & D \\ \begin{matrix} N & \end{matrix} & \mathbf{x} \end{matrix} = \begin{matrix} & N \\ N & \end{matrix} \mathbf{U} \begin{matrix} & D \\ N & \end{matrix} \begin{matrix} & D \\ & \Lambda \end{matrix} \begin{matrix} & D \\ D & \end{matrix} \mathbf{v}^T$$

$$\mathbf{X} = \mathbf{U} \Lambda \mathbf{V}^T = \sum_{k=1}^{\min\{N,D\}} \lambda_k \mathbf{u}_k \mathbf{v}_k^T$$

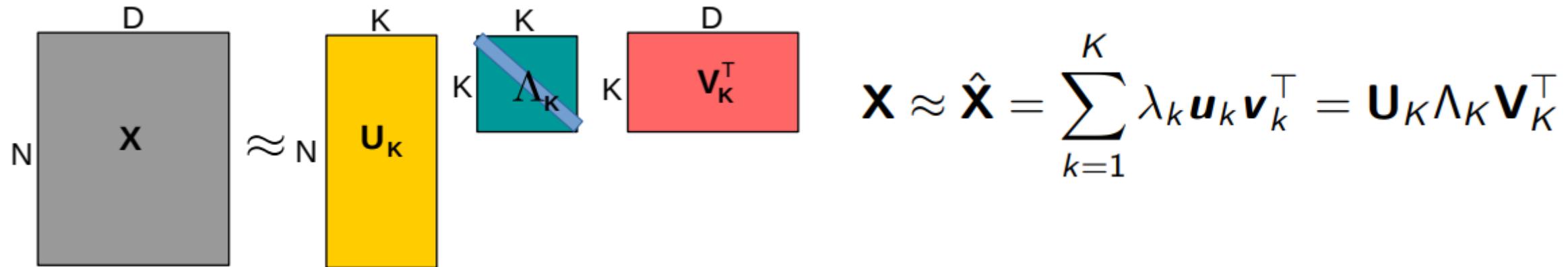
Diagonal matrix. If $N > D$, last $D - N$ rows are all zeros; if $D > N$, last $D - N$ columns are all zeros

- $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_N]$ is $N \times N$ matrix of **left singular vectors**, each $\mathbf{u}_n \in \mathbb{R}^N$
 - \mathbf{U} is also orthonormal
- $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_N]$ is $D \times D$ matrix of **right singular vectors**, each $\mathbf{v}_d \in \mathbb{R}^D$
 - \mathbf{V} is also orthonormal
- Λ is $N \times D$ with only $\min(N, D)$ diagonal entries - **singular values**
- Note: If \mathbf{X} is symmetric then it is known as eigenvalue decomposition ($\mathbf{U} = \mathbf{V}$)



Low-Rank Approximation via SVD

- If we just use the top $K < \min\{N, D\}$ singular values, we get a rank- K SVD

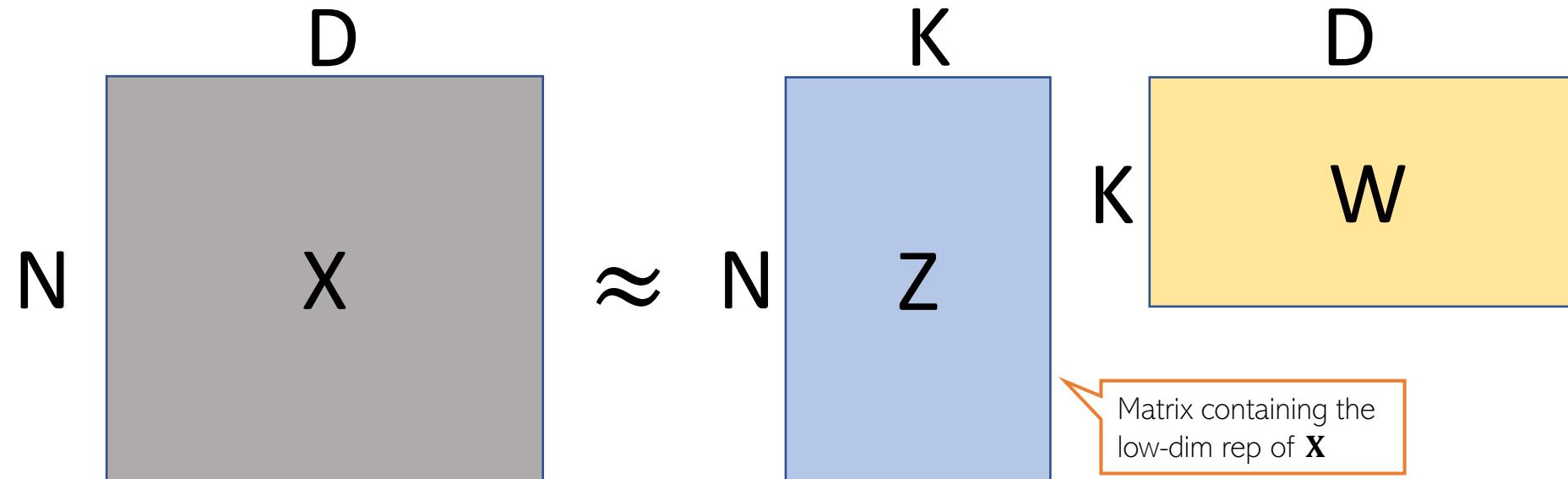


- Above SVD approx. can be shown to minimize the reconstruction error $\|\mathbf{X} - \hat{\mathbf{X}}\|$
 - Fact: SVD gives the best rank- K approximation of a matrix
- PCA is done by doing SVD on the covariance matrix \mathbf{S} (left and right singular vectors are the same and become eigenvectors, singular values become eigenvalues)



Dim-Red as Matrix Factorization

- If we don't care about the orthonormality constraints, then dim-red can also be achieved by solving a matrix factorization problem on the data matrix \mathbf{X}



$$\{\hat{\mathbf{Z}}, \hat{\mathbf{W}}\} = \operatorname{argmin}_{\mathbf{Z}, \mathbf{W}} \|\mathbf{X} - \mathbf{Z}\mathbf{W}\|^2$$

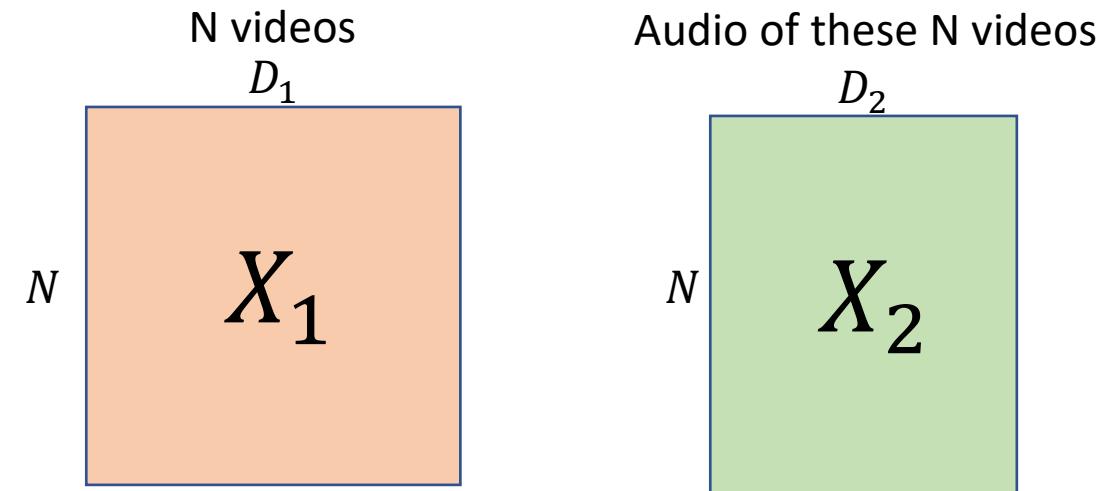
If $K < \min\{D, N\}$, such a factorization gives a low-rank approximation of the data matrix \mathbf{X}

- Can solve such problems using ALT-OPT
- Can impose various constraints on \mathbf{Z} and \mathbf{W} (e.g., sparsity, non-negativity, etc)



Joint Dim-Red

- Often we have two or more data sources with 1-1 correspondence between inputs



- Sometimes, we may want to perform a common dim-red for both sources to get a common feature rep which captures properties of both sources (or fused their info)
- This can be done by doing a joint dim-red of both sources. Many methods exists, e.g.,
 - Canonical Correlational Analysis (CCA): looks at cross-covar rather than variances
 - Joint Matrix Factorization

$$\operatorname{argmin}_{\mathbf{Z}, \mathbf{W}_1, \mathbf{W}_2} \|\mathbf{X}_1 - \mathbf{Z}\mathbf{W}_1\|^2 + \|\mathbf{X}_2 - \mathbf{Z}\mathbf{W}_2\|^2$$



Coming up next

- Some methods for computing eigenvectors
- Supervised dimensionality reduction
- Nonlinear dimensionality reduction
 - Kernel PCA
 - Manifold Learning



Dimensionality Reduction (Contd)

CS771: Introduction to Machine Learning

Piyush Rai

Plan

- A fast method for computing eigenvectors (power method)
- Supervised dimensionality reduction
- Nonlinear dimensionality reduction
 - Kernel PCA
 - Manifold Learning (LLE and SNE/tSNE)



Power Method for Computing Eigenvectors

- Eigen-decomposition is expensive in general – $O(D^3)$ for a $D \times D$ matrix
- For naïve methods, even to get one eigenvector, we need to perform full eigen-decom.
- If we want $K < D$ eigenvectors, there are some more efficient methods
- Power Method (a.k.a. Power Iteration) is one such iterative approach

$O(KD^2)$ cost to find
K top eigenvectors

- Sequentially finds the top K eigenvectors of a cov matrix $\mathbf{S} = \sum_{k=1}^D \lambda_k \mathbf{u}_k \mathbf{u}_k^\top$
- Based on the fact that any vector \mathbf{x} can be written as $\mathbf{x} = \sum_{k=1}^D z_k \mathbf{u}_k$, and thus

$$\mathbf{S}\mathbf{x} = \sum_{k=1}^D z_k \lambda_k \mathbf{u}_k \quad \text{and} \quad \underbrace{(\mathbf{S}\mathbf{S}^\top, \dots, \mathbf{S})}_{M \text{ times}} \mathbf{x} = \sum_{k=1}^D z_k \lambda_k^M \mathbf{u}_k$$

- Assuming $\lambda_1 > \lambda_2 \geq \lambda_3 \dots \geq \lambda_D$ then for large M

$$\underbrace{(\mathbf{S}\mathbf{S}^\top, \dots, \mathbf{S})}_{M \text{ times}} \mathbf{x} \approx z_1 \lambda_1^M \mathbf{u}_1$$



Power Method for Computing Eigenvectors

- So we had the following:

$$\underbrace{(\mathbf{S}\mathbf{S}^\top, \dots, \mathbf{S})}_{M \text{ times}} \mathbf{x} \approx z_1 \lambda_1^M \mathbf{u}_1$$

- This gives us a simple algorithm to get the **top eigenvector**

- Initialize $\mathbf{x}_0 \sim \mathcal{N}(0, \mathbf{I}_D)$
- For $m = 1, \dots, M$, compute \mathbf{x}_m as $\mathbf{x}_m = \mathbf{S}\mathbf{x}_{m-1}$ and normalize it as $\mathbf{x}_m = \mathbf{x}_m / \|\mathbf{x}_m\|_2$
- After convergence, \mathbf{x}_M is the largest eigenvector and $\|\mathbf{S}\mathbf{x}_M\|$ is the largest eigenvalue

Since eigenvectors should have unit norm

Using the fact $\mathbf{S}\mathbf{x} = \lambda\mathbf{x}$ and that \mathbf{x} has unit norm

- The main dominant cost is computing $\mathbf{S}\mathbf{x}_{m-1}$ whose cost is $\mathcal{O}(D^2)$
- Can use this technique to also obtain the remaining eigenvectors sequentially using a “peeling” technique



Power Method with Peeling Technique

- Can use Power Method with a “peeling” technique to get all the top K eigenvectors

- The basic procedure would be

- Initialize $\mathbf{S}^{(0)} = \mathbf{S}$ $\sum_{k=1}^D \lambda_k \mathbf{u}_k \mathbf{u}_k^\top$
- For $k = 1, \dots, K$

$$\{\mathbf{u}_k, \lambda_k\} = \text{POWER-METHOD}(\mathbf{S}^{(k-1)})$$

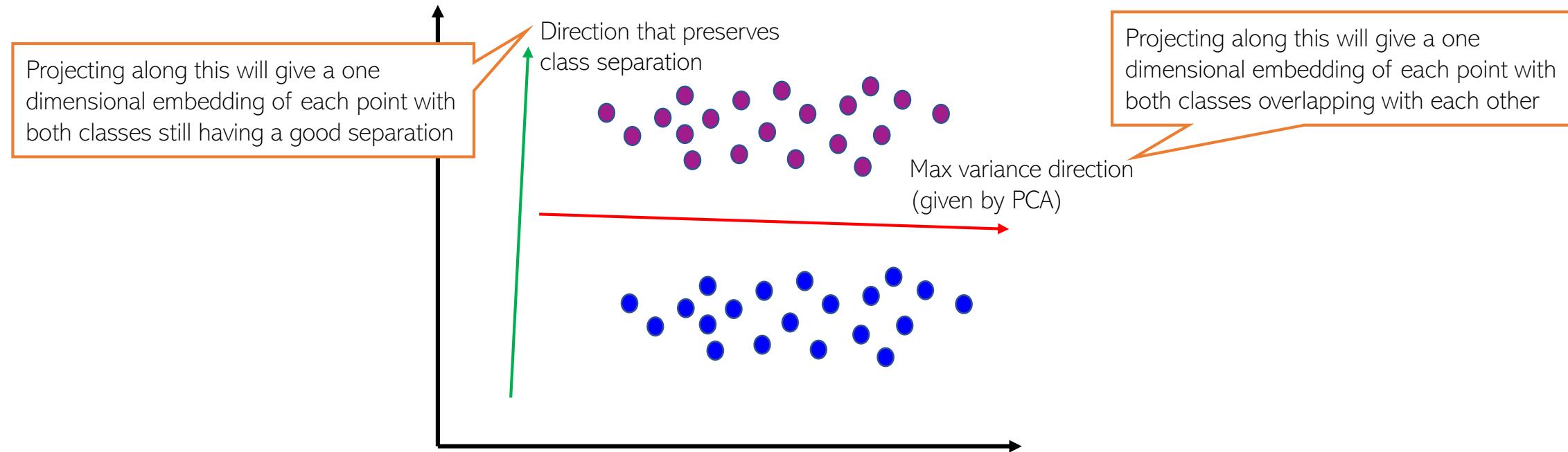
$$\mathbf{S}^{(k)} = \mathbf{S}^{(k-1)} - \lambda_k \mathbf{u}_k \mathbf{u}_k^\top \quad (\text{"Peeling" the covariance matrix})$$

- Each power iteration is $O(D^2)$, overall cost for getting K eigenvectors is $O(KD^2)$



Supervised Dimensionality Reduction

- Maximum variance directions may not be aligned with class separation directions



- Be careful when using PCA for supervised learning problems
- A better option would be to project such that
 - Points within the same class are close (low intra-class variance)
 - Points from different classes are well separated (the class means are far apart)



Supervised Dimensionality Reduction

- Many techniques. A simple yet popular one is Fisher Discriminant Analysis, also known as Linear Discriminant Analysis (FDA or LDA)

This LDA should not be confused with another very popular ML technique for finding topics in text data (Latent Dirichlet Allocation)

- For simplicity, assume two classes (can be generalized for more than 2 classes too)
- Suppose a projection direction \mathbf{u} . After projection the means of the two classes are

$$\mu_1 = \frac{1}{N_1} \sum_{n:y_n=1} \mathbf{u}^\top \mathbf{x}_n, \quad \mu_2 = \frac{1}{N_2} \sum_{n:y_n=2} \mathbf{u}^\top \mathbf{x}_n$$

- Total variance of the points after projection will be $s_1^2 + s_2^2$ where

$$s_1^2 = \frac{1}{N_1} \sum_{n:y_n=1} (\mathbf{u}^\top \mathbf{x}_n - \mu_1)^2, \quad s_2^2 = \frac{1}{N_2} \sum_{n:y_n=2} (\mathbf{u}^\top \mathbf{x}_n - \mu_2)^2$$

Here we considered projection to one dimension but can be generalized to projection to K dim



- Fisher discriminant analysis finds the optimal projection direction by solving

The solution to this problem involves solving an eigendecomposition problem that involves **within class** covariance matrices and **between class** covariance matrices

$$\arg \max_{\mathbf{u}} \frac{(\mu_1 - \mu_2)^2}{s_1^2 + s_2^2}$$

Push the means far apart

Make each class tightly packed after projection (small variance)



Dimensionality Reduction given Pairwise Distances between points



Dim. Reduction by Preserving Pairwise Distances

- PCA/SVD etc assume we are given points $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$ as vectors (e.g., in D dim)
- Often the data is given in form of **distances** d_{ij} between points ($i, j = 1, 2, \dots, N$)
- Would like to project data such that pairwise distances between points are preserved

$$\hat{\mathbf{Z}} = \arg \min_{\mathbf{Z}} \mathcal{L}(\mathbf{Z}) = \arg \min_{\mathbf{Z}} \sum_{i,j=1}^N (d_{ij} - \|\mathbf{z}_i - \mathbf{z}_j\|)^2$$

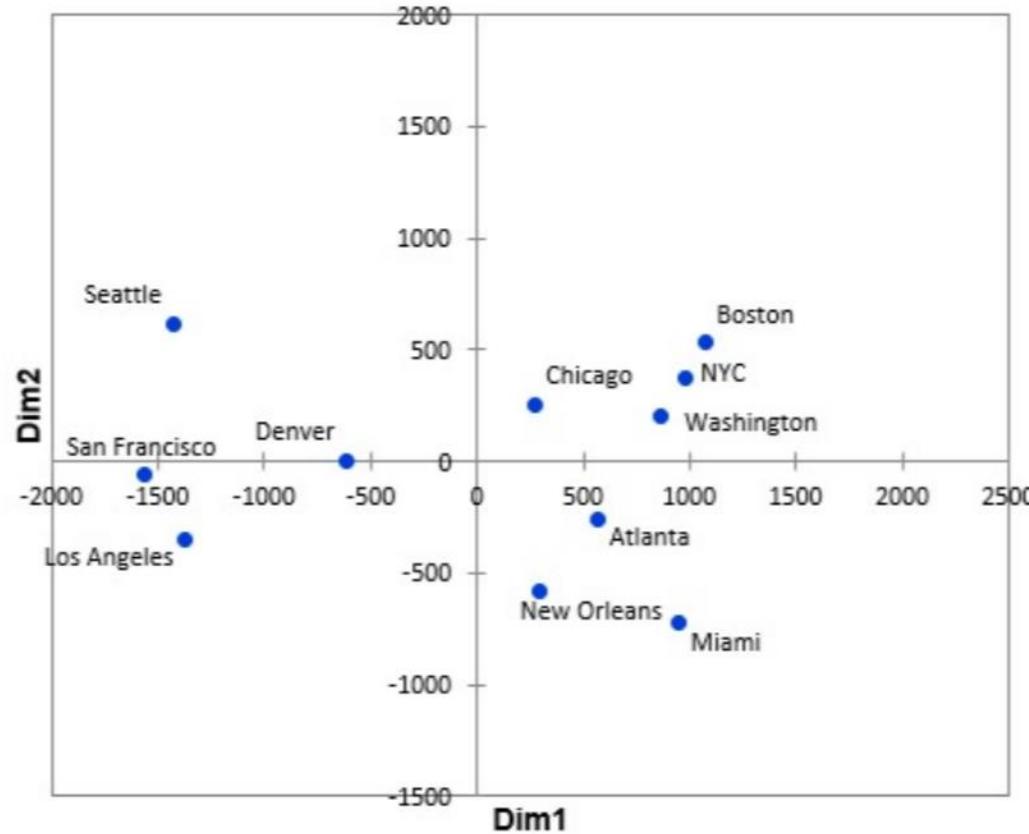
\mathbf{z}_i and \mathbf{z}_j denote low-dim embeddings/projections of points i and j

- Basically, if d_{ij} is large (resp. small), would like $\|\mathbf{z}_i - \mathbf{z}_j\|$ to be large (resp. small)
- Multi-dimensional Scaling (MDS) is one such algorithm
- Note: If d_{ij} is the Euclidean distance, MDS is equivalent to PCA
- The above approach tries to preserve all pairwise distances
 - Can try to preserve pairwise distances only between close-by points (i.e.. b/w nearest neighbors). It helps achieve non-linear dim red. Algos like Isomap and locally linear embedding (LLE) do this



MDS: An Example

- Result of applying MDS (with $K = 2$) on pairwise distances between some US cities



- MDS produces a 2D embedding such that geographically close cities are also close in embedding space

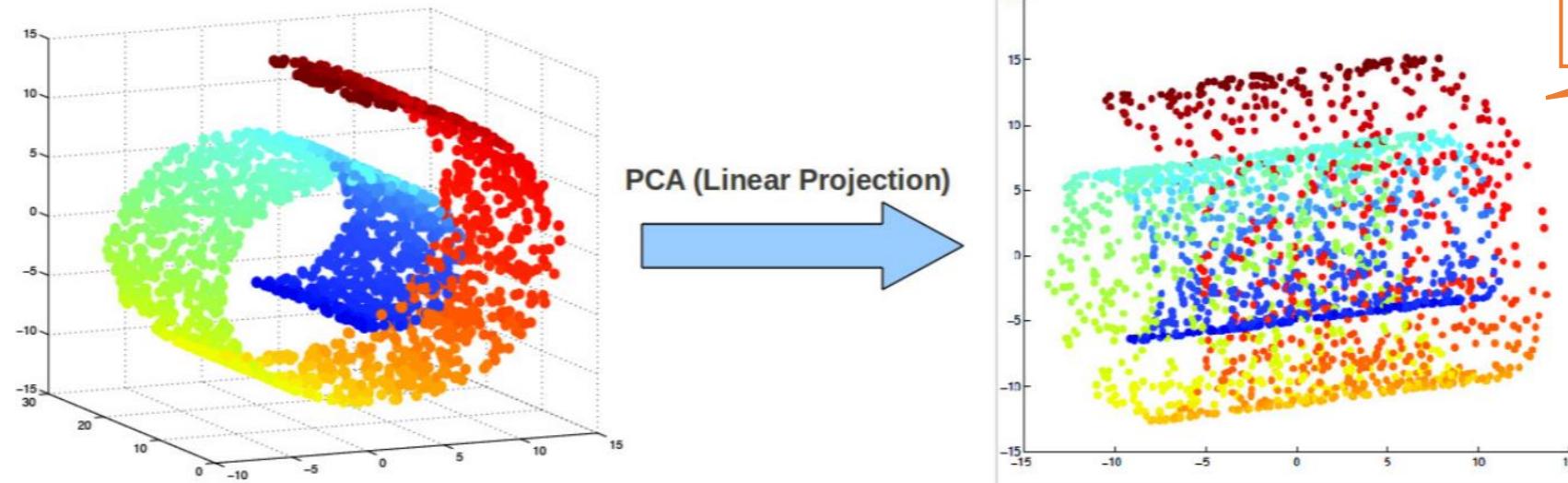


Nonlinear Dimensionality Reduction



Beyond Linear Projections

- Consider the swiss-roll dataset (points lying close to a manifold)

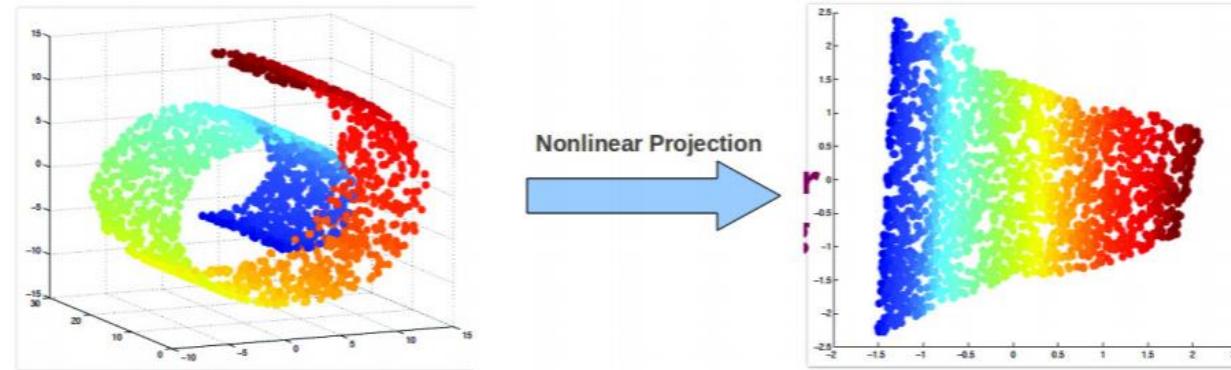


- Linear projection methods (e.g., PCA) can't capture intrinsic nonlinearities
 - Maximum variance directions may not be the most interesting ones



Nonlinear Dimensionality Reduction

- We want to learn a nonlinear low-dim projection



Relative positions of points preserved after the projection

- Some ways of doing this
 - Nonlinearize a linear dimensionality reduction method. E.g.:
 - Cluster data and apply linear PCA within each cluster ([mixture of PCA](#))
 - [Kernel PCA](#) (nonlinear PCA)
 - Using [manifold based methods](#) that intrinsically preserve nonlinear geometry, e.g.,
 - Locally Linear Embedding (LLE), Isomap
 - Maximum Variance Unfolding
 - Laplacian Eigenmap, and others such as SNE/tSNE, etc.
- .. or use unsupervised deep learning techniques (later)

Will look at KPCA,
LLE, SNE/tSNE



Kernel PCA

- Recall PCA: Given N observations $\mathbf{x}_n \in \mathbb{R}^D$, $n = 1, 2, \dots, N$,

$$\mathbf{S} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top$$

$D \times D$ cov matrix
assuming centered data

D eigenvectors of \mathbf{S}

$$\mathbf{S}\mathbf{u}_i = \lambda_i \mathbf{u}_i \quad \forall i = 1, \dots, D$$

- Assume a kernel k with associated M dimensional nonlinear map ϕ

$$\mathbf{C} = \frac{1}{N} \sum_{n=1}^N \phi(\mathbf{x}_n) \phi(\mathbf{x}_n)^\top$$

$M \times M$ cov matrix assuming
centered data in the kernel-
induced feature space

M eigenvectors of \mathbf{C}

$$\mathbf{C}\mathbf{v}_i = \lambda_i \mathbf{v}_i \quad \forall i = 1, \dots, M$$

- Would like to do it without computing \mathbf{C} and the mappings $\phi(\mathbf{x}_n)$'s since M can be very large (even infinite, e.g., when using an RBF kernel)
- Boils down to doing eigendecomposition of the $N \times N$ kernel matrix \mathbf{K} (PRML 12.3)

- Can verify that each \mathbf{v}_i above can be written as a lin-comb of the inputs: $\mathbf{v}_i = \sum_{n=1}^N \mathbf{a}_{in} \phi(\mathbf{x}_n)$
- Can show that finding $\mathbf{a}_i = [a_{i1}, a_{i2}, \dots, a_{iN}]$ reduces to solving an eigendecomposition of \mathbf{K}
- Note: Due to req. of centering, we work with a centered kernel matrix $\tilde{\mathbf{K}} = \mathbf{K} - \mathbf{1}_N \mathbf{K} - \mathbf{K} \mathbf{1}_N + \mathbf{1}_N \mathbf{K} \mathbf{1}_N$

$N \times N$ matrix of all 1s

Locally Linear Embedding

Several non-lin dim-red algos use this idea

Essentially, neighbourhood preservation, but only local

- Basic idea: If two points are **local neighbors** in the original space then they should be local neighbors in the projected space too
- Given N observations $\mathbf{x}_n \in \mathbb{R}^D$, $n = 1, 2, \dots, N$, LLE is formulated as

Solve this to learn weights W_{ij} such that each point \mathbf{x}_i can be written as a weighted combination of its local neighbors in the original feature space

$$\hat{\mathbf{W}} = \arg \min_{\mathbf{W}} \sum_{i=1}^N \left\| \mathbf{x}_i - \sum_{j \in \mathcal{N}(i)} W_{ij} \mathbf{x}_j \right\|^2$$

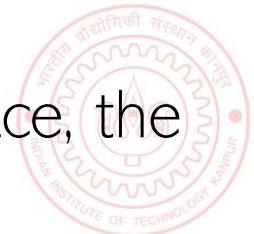
$\mathcal{N}(i)$ denotes the local neighbors (a predefined number, say K , of them) of point \mathbf{x}_i

- For each point $\mathbf{x}_n \in \mathbb{R}^D$, LLE learns $\mathbf{z}_n \in \mathbb{R}^K$, $n = 1, 2, \dots, N$ such that the same neighborhood structure exists in low-dim space too

Requires solving an eigenvalue problem

$$\hat{\mathbf{z}} = \arg \min_{\mathbf{z}} \sum_{i=1}^N \left\| \mathbf{z}_i - \sum_{j \in \mathcal{N}(i)} W_{ij} \mathbf{z}_j \right\|^2$$

- Basically, if point \mathbf{x}_i can be reconstructed from its neighbors in the original space, the same weights W_{ij} should be able to reconstruct \mathbf{z}_i in the new space too



SNE and t-SNE

Thus very useful if we want to visualize some high-dim data in two or three dims

- Also nonlin. dim-red methods, especially suited for projecting to 2D or 3D
- SNE stands for [Stochastic Neighbor Embedding](#) (Hinton and Roweis, 2002)
- Uses the idea of preserving [probabilistically defined neighborhoods](#)
- SNE, for each point \mathbf{x}_i , defines the probability of a point \mathbf{x}_j being its neighbor as

Neighbor probability
in the original space

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2/2\sigma^2)}{\sum_{k \neq i} \exp(-\|\mathbf{x}_i - \mathbf{x}_k\|^2/2\sigma^2)}$$

Neighbor probability in the
projected/embedding space

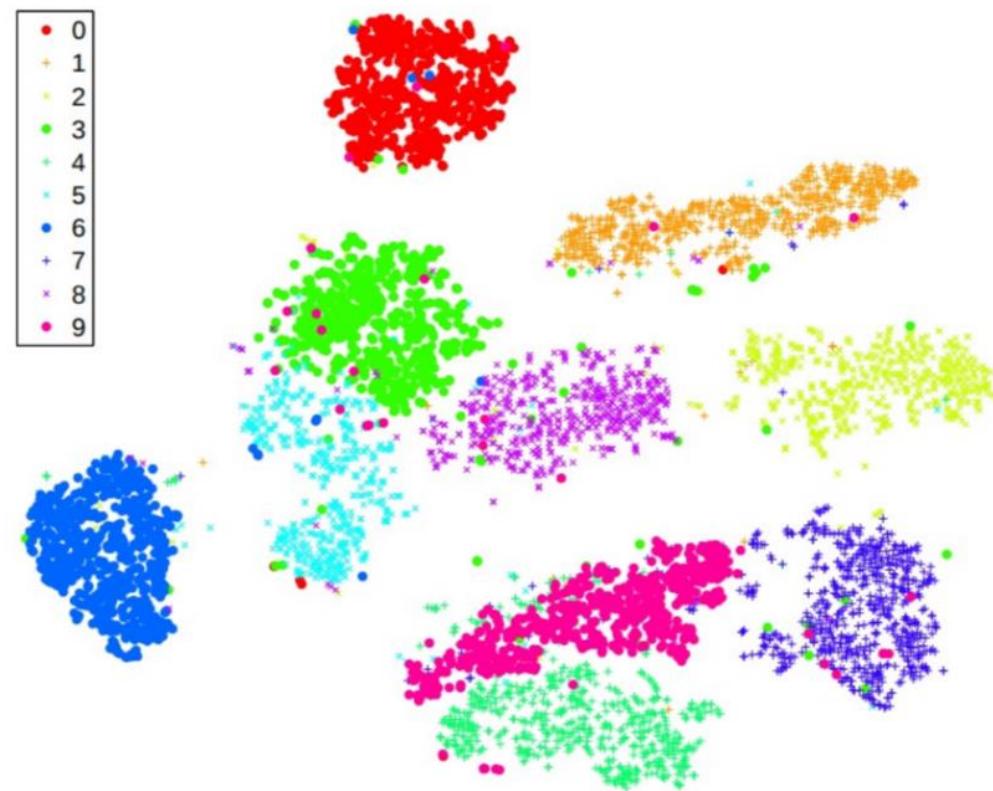
$$q_{j|i} = \frac{\exp(-\|\mathbf{z}_i - \mathbf{z}_j\|^2/2\sigma^2)}{\sum_{k \neq i} \exp(-\|\mathbf{z}_i - \mathbf{z}_k\|^2/2\sigma^2)}$$

- SNE ensures that neighbourhood distributions in both spaces are as close as possible
 - By minimizing their Kullback-Leibler divergence, summed over all points $\sum_{i=1}^N \sum_{j=1}^N KL(p_{j|i} || q_{j|i})$
- t-SNE (van der Maaten and Hinton, 2008) offers a couple of improvements to SNE
 - Learns \mathbf{z}_i 's by minimizing [symmetric KL divergence](#)
 - Uses [Student-t distribution](#) instead of Gaussian for defining $q_{j|i}$



SNE and t-SNE

- Especially useful for visualizing data by projecting into 2D or 3D



Result of visualizing MNIST digits data in 2D (Figure from van der Maaten and Hinton, 2008)



Latent Variable Models (LVMs), Parameter Estimation in LVM

CS771: Introduction to Machine Learning

Piyush Rai

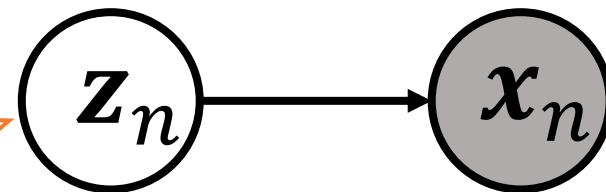
Generative Models with Latent Variables

- Have already looked at generative models for supervised learning
- Generative models are even more common/popular for unsupervised learning, e.g.,
 - Clustering
 - Dimensionality Reduction
 - Probability density estimation
- In such models, each data point is associated with a latent variable
 - Clustering: The cluster id \mathbf{z}_n (discrete, or a K-dim one-hot rep, or a vector of cluster membership probabilities)
 - Dimensionality reduction: The low-dim representation $\mathbf{z}_n \in \mathbb{R}^K$
- These latent variables will be treated as **random variables**, not just fixed unknowns
- Will therefore assume a suitable prior distribution on these and estimate their posterior
 - If we only need a point estimate (MLE/MAP) of these latent variables, that can be done too

Supervised/semi-supervised learning models can also have latent variables, depending on the problem formulation



Latent variable \mathbf{z}_n usually encodes some latent properties of the observation \mathbf{x}_n



As in unsup learning algos such as K -means, standard PCA, etc



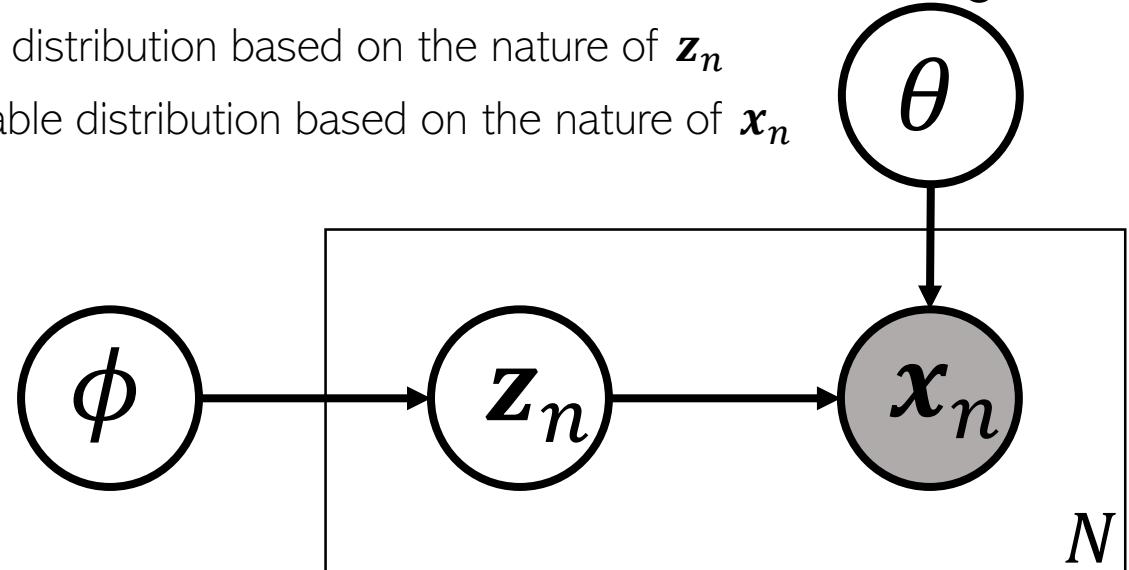
Generative Models with Latent Variables

3

- A typical generative model with latent variables might look like this

$p(\mathbf{z}_n|\phi)$: A suitable distribution based on the nature of \mathbf{z}_n

$p(\mathbf{x}_n|\mathbf{z}_n, \theta)$: A suitable distribution based on the nature of \mathbf{x}_n



- In this generative model, observations \mathbf{x}_n assumed generated via latent variables \mathbf{z}_n
- The unknowns in such latent var models (LVMs) are of two types
 - Global variables:** Shared by all data points (θ and ϕ in the above diagram)
 - Local variables:** Specific to each data point (\mathbf{z}_n 's in the above diagram)
- Note: Both global and local unknowns can be treated as r.v.'s

Such diagrams are called “plate notation”. The plate depicts replicas and the number (N here) denotes how many such instances



Grey nodes mean that they are observed; white nodes means that they are unknown. Some of the white nodes may be treated as latent variables, and some as fixed unknowns

Arrow directions denote dependence: In the generative model's description (likelihood and prior), the node with an incident arrow is dependent on the node where the arrow comes from

The distinction will go away we want to treat both sets of unknowns as random variables (discussion beyond the scope of this course but CS698X)

However, here we will only treat the local variables \mathbf{z}_n 's as random latent variable and regard θ and ϕ as other unknown “parameters” of the model

ML

An Example of a Generative LVM

If the \mathbf{z}_n were known, it just comes generative classification, for which we know how to estimate θ and ϕ , given training data

4



- Probabilistic Clustering can be formulated as a generative latent variable model
- Assume K probability distributions (e.g., Gaussians), one for each cluster

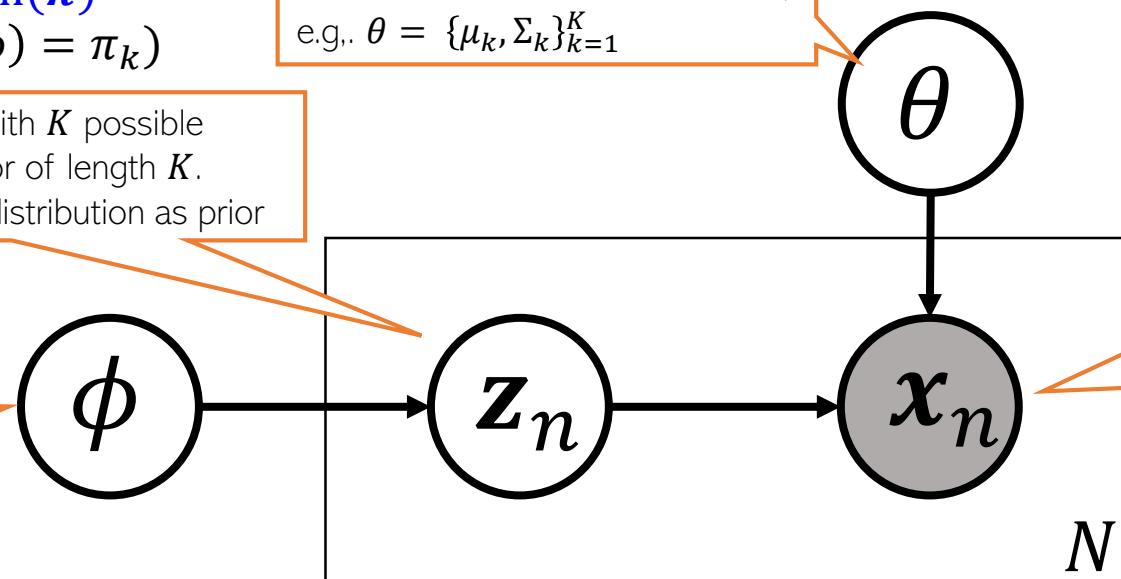
$$p(\mathbf{z}_n | \phi) = \text{multinoulli}(\boldsymbol{\pi})$$

(also means $p(\mathbf{z}_n = k | \phi) = \pi_k$)

Parameters of the K distributions,
e.g., $\theta = \{\mu_k, \Sigma_k\}_{k=1}^K$

Discrete latent variable (with K possible values) or a one-hot vector of length K . Modeled by a multinoulli distribution as prior

The parameter vector $\boldsymbol{\pi} = [\pi_1, \pi_2, \dots, \pi_K]$ of the multinoulli distribution



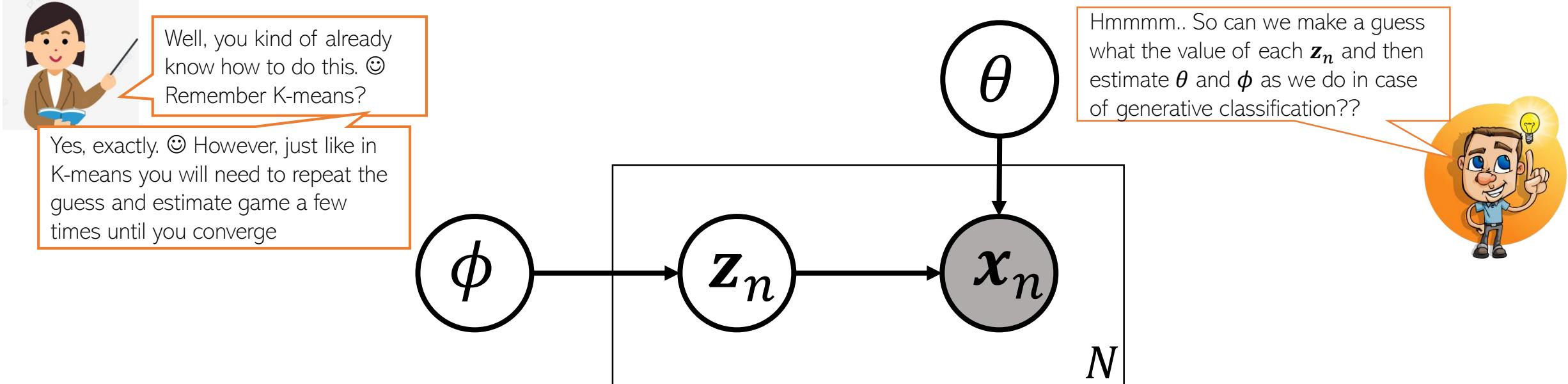
$$p(\mathbf{x}_n | \mathbf{z}_n = k, \theta) = \mathcal{N}(\mu_k, \Sigma_k)$$

- In any such LVM, ϕ denotes parameters of the prior distribution on \mathbf{z}_n
- .. and θ denotes parameters of the likelihood distribution on \mathbf{x}_n



Parameter Estimation for Generative LVM

- So how do we estimate the parameters of a generative LVM, say prob. clustering?



- The guess about \mathbf{z}_n can be in one of the two forms
 - A “hard” guess – a fixed value (some “optimal” value of the random variable \mathbf{z}_n)
 - The “expected” value $\mathbb{E}[\mathbf{z}_n]$ of the random variable \mathbf{z}_n
- Using the hard guess of \mathbf{z}_n will result in an ALT-OPT like algorithm
- Using the expected value of \mathbf{z}_n will give the so-called Expectation-Maximization (EM) algo

EM is pretty much like ALT-OPT but with soft/expected values of the latent variables

Parameter Estimation for Generative LVM

- Can we estimate parameters $(\theta, \phi) = \Theta$ (say) of an LVM **without estimating \mathbf{z}_n** ?
- In principle yes, but it is harder
- Given N observations $\mathbf{x}_n, n = 1, 2, \dots, N$, the MLE problem for Θ will be

$$\operatorname{argmax}_{\Theta} \sum_{n=1}^N \log p(\mathbf{x}_n | \Theta) = \operatorname{argmax}_{\Theta} \sum_{n=1}^N \log \sum_{\mathbf{z}_n} p(\mathbf{x}_n, \mathbf{z}_n | \Theta)$$

After the summation/integral on the RHS,
 $p(\mathbf{x}_n | \Theta)$ is no longer exp. family even if
 $p(\mathbf{z}_n | \phi)$ and $p(\mathbf{x}_n | \mathbf{z}_n, \phi)$ are in exp-fam ☺

The discussion here is also true
for MAP estimation of Θ

Also note that $p(\mathbf{x}_n, \mathbf{z}_n | \Theta) = p(\mathbf{z}_n | \phi)p(\mathbf{x}_n | \mathbf{z}_n, \theta)$

Summing over all possible values \mathbf{z}_n can take (would
be an integral instead of sum if \mathbf{z}_n is continuous)

- For the probabilistic clustering model (GMM) we saw, $p(\mathbf{x}_n | \Theta)$ will be

$$p(\mathbf{x}_n | \Theta) = \sum_{k=1}^K p(\mathbf{x}_n, \mathbf{z}_n = k | \Theta) = \sum_{k=1}^K p(\mathbf{z}_n = k | \phi)p(\mathbf{x}_n | \mathbf{z}_n = k, \theta) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}_n | \mu_k, \Sigma_k)$$

Convex combination (mixture)
of K Gaussians. No longer an
exp-family distribution

- MLE problem thus will be $\operatorname{argmax}_{\Theta} \sum_{n=1}^N \log \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}_n | \mu_k, \Sigma_k)$

The log of sum doesn't give us a simple
expression; MLE can still be done using
gradient based methods but update will
be complicated. ALT-OPT or EM make it
simpler by using guesses of \mathbf{z}_n 's

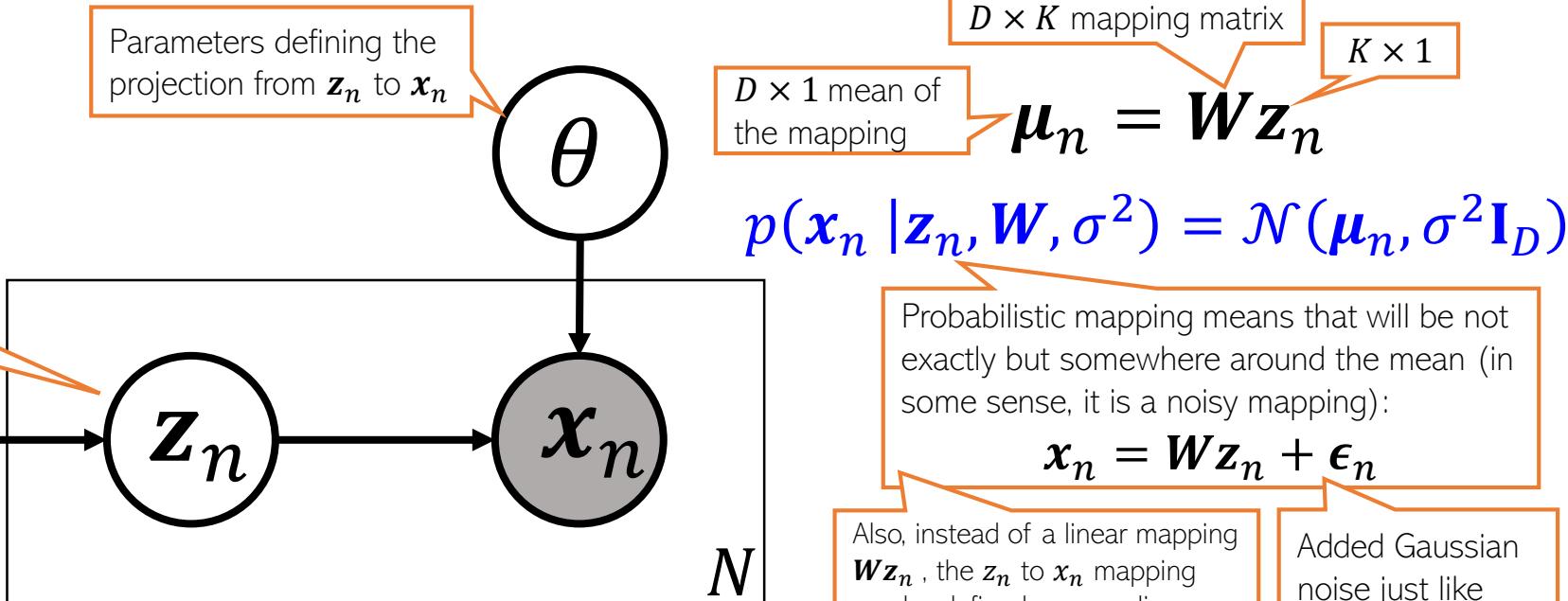
Another Example of a Gen. LVM

- Probabilistic PCA (PPCA) is another example of a generative latent var model
- Assume a K -dim latent var \mathbf{z}_n mapped to a D -dim observation \mathbf{x}_n via a prob. mapping

$$p(\mathbf{z}_n | \phi) = \mathcal{N}(0, \mathbf{I}_K)$$

Real-valued vector of length K . Modeled by a zero-mean K -dim Gaussian distribution as prior

The parameters of the Gaussian prior on \mathbf{z}_n . In this example, no such parameters are actually needed since mean is zero and cov matrix is identity, but can use nonzero mean and more general cov matrix for the Gaussian prior



- PPCA has several benefits over PCA, some of which include
 - Can use suitable distributions for \mathbf{x}_n to better capture properties of data
 - Parameter estimation can be done faster without eigen-decomposition (using ALT-OPT/EM algos)



Generative Models and Generative Stories

- Data generation for a generative model can be imagined via a generative story
- This story is just our hypothesis of how “nature” generated the data
- For the Gaussian mixture model (GMM), the (somewhat boring) story is as follows
 - For each data point \mathbf{x}_n with index $n = 1, 2, \dots, N$
 - Generate its cluster assignment by drawing from prior $p(z_n|\phi)$

$$\mathbf{z}_n \sim \text{multinoulli}(\boldsymbol{\pi})$$
 - Assuming $z_n = k$, generate the data point \mathbf{x}_n from $p(\mathbf{x}_n|\mathbf{z}_n, \theta)$

$$\mathbf{x}_n \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$
- Can imagine a similar story for PPCA with \mathbf{z}_n generated from $\mathcal{N}(0, \mathbf{I}_K)$ and then conditioned on \mathbf{z}_n , the observation \mathbf{x}_n generated from $p(\mathbf{x}_n | \mathbf{z}_n, \mathbf{W}, \sigma^2) = \mathcal{N}(\mathbf{W}\mathbf{z}_n, \sigma^2\mathbf{I}_D)$
- For GMM/PPCA, the story is rather simplistic but for more sophisticated models, gives an easy way to understand/explain the model, and data generation process



Coming up next

- ALT-OPT and EM algorithm for parameter estimation in LVMs
 - Will look at it through the example of a Gaussian Mixture Model (GMM)
 - Also, the PPCA model
 - Will also look at it for the general case as well



LVMs (Contd), Expectation Maximization (1)

CS771: Introduction to Machine Learning

Piyush Rai

Plan

- ALT-OPT and EM
 - Example: Gaussian Mixture Model for data clustering
- A deeper look at ALT-OPT and EM
- General recipe for doing ALT-OPT and EM for any LVM



Need for EM/ALT-OPT: Two Equivalent Perspectives³

1. Consider an LVM with **latent variables** and **parameters**. Trying to estimate parameters without also estimating the latent variables (by marginalizing them) is difficult

$$p(\mathbf{x}_n|\Theta) = \sum_{k=1}^K p(\mathbf{x}_n, \mathbf{z}_n = k|\Theta) = \sum_{k=1}^K p(\mathbf{z}_n = k|\phi)p(\mathbf{x}_n|\mathbf{z}_n = k, \Theta) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)$$

A Gaussian Mixture Model (GMM)

MLE for GMM with cluster ids
marginalized/summed/integrated out

$$\Theta_{MLE} = \underset{\Theta}{\operatorname{argmax}} \sum_{n=1}^N \log \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)$$

Can't get closed form expressions for
the π_k, μ_k, Σ_k due to "log of sum".
Have to use gradient based methods

This issue not
just for MLE for
GMM but MLE
for other LVMs
too

EM/ALT-OPT will help us "simulate" this condition
by making guesses about the values of \mathbf{z}_n 's

If we knew the \mathbf{z}_n 's, the problem will be much simpler; just like
MLE for generative classification with Gaussian class-conditional

Since no marginalization
of the \mathbf{z}_n 's required

2. Consider a complex prob. density (without any latent vars) for which MLE is hard

Directly defining a probability density as a mixture of Gaussians (\mathbf{x}_n is generated by the k^{th} Gaussian with probability π_k) without any reference to any latent variable whatsoever (we didn't define it as an LVM)

$$p(\mathbf{x}_n|\Theta) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)$$

MLE for the params Θ of this distribution will again be hard (as we already saw above). However, we can artificially introduce a latent variable \mathbf{z}_n with each data point \mathbf{x}_n , denoting which Gaussian generated \mathbf{x}_n

Can now apply ALT-OPT/EM to estimate parameters Θ + we get the latent variables \mathbf{z}_n as a "by-product" (though we may not be interested in learning \mathbf{z}_n 's if our goal is just density estimation, not clustering)

Now this prob. density estimation problem also becomes
Problem 1 above - a clustering problem with latent variables

Even though we didn't need the artificially introduced \mathbf{z}_n 's, their presence and doing ALT-OPT/EM made our job of estimating Θ easier!

Also in any LVM, given Θ ,
you can always estimate
 \mathbf{z}_n 's. Likewise, given \mathbf{z}_n , you
can always estimate Θ



Remember that GMM is just like
generative classification with
Gaussian class-conditionals and
training data labels unknown

ALT-OPT/EM for Gaussian Mixture Model



Detour: MLE for Generative Classification

- Assume a K class generative classification model with Gaussian class-conditionals
- Assume class $k = 1, 2, \dots, K$ is modeled by a Gaussian with mean μ_k and cov matrix Σ_k
- Can assume label y_n to be one-hot and then $y_{nk} = 1$ if $y_n = k$, and $y_{nk} = 0$, o/w
- Assuming class prior as $p(y_n = k) = \pi_k$, the model has params $\Theta = \{\pi_k, \mu_k, \Sigma_k\}_{k=1}^K$
- Given training data $\{\mathbf{x}_n, y_n\}_{n=1}^N$, the MLE solution will be

$$\hat{\pi}_k = \frac{1}{N} \sum_{n=1}^N y_{nk}$$

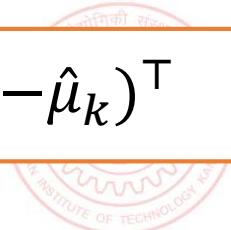
Same as $\frac{N_k}{N}$ where N_k is # of training ex. for which $y_n = k$

$$\hat{\mu}_k = \frac{1}{N_k} \sum_{n=1}^N y_{nk} \mathbf{x}_n$$

Same as $\frac{1}{N_k} \sum_{n:y_n=k}^N \mathbf{x}_n$

$$\hat{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^N y_{nk} (\mathbf{x}_n - \hat{\mu}_k)(\mathbf{x}_n - \hat{\mu}_k)^T$$

Same as $\frac{1}{N_k} \sum_{n:y_n=k}^N (\mathbf{x}_n - \hat{\mu}_k)(\mathbf{x}_n - \hat{\mu}_k)^T$



Detour: MLE for Generative Classification

- Here is a formal derivation of the MLE solution for $\Theta = \{\pi_k, \mu_k, \Sigma_k\}_{k=1}^K$

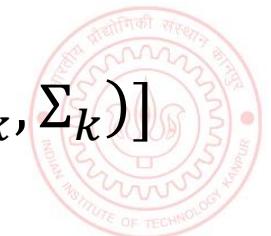
$$\begin{aligned}
 \widehat{\Theta} &= \operatorname{argmax}_{\Theta} p(X, y|\Theta) = \operatorname{argmax}_{\Theta} \prod_{n=1}^N p(x_n, y_n|\Theta) \text{ multinoulli} \\
 &= \operatorname{argmax}_{\Theta} \prod_{n=1}^N p(y_n|\Theta) p(x_n|y_n, \Theta) \text{ Gaussian} \\
 &= \operatorname{argmax}_{\Theta} \prod_{n=1}^N \prod_{k=1}^K \pi_k^{y_{nk}} \prod_{k=1}^K p(x_n|y_n=k, \Theta)^{y_{nk}} \\
 &= \operatorname{argmax}_{\Theta} \prod_{n=1}^N \prod_{k=1}^K [\pi_k p(x_n|y_n=k, \Theta)]^{y_{nk}} \\
 &= \operatorname{argmax}_{\Theta} \log \prod_{n=1}^N \prod_{k=1}^K [\pi_k p(x_n|y_n=k, \Theta)]^{y_{nk}} \\
 &= \operatorname{argmax}_{\Theta} \sum_{n=1}^N \sum_{k=1}^K y_{nk} [\log \pi_k + \log \mathcal{N}(x_n|\mu_k, \Sigma_k)]
 \end{aligned}$$

In general, in models with probability distributions from the **exponential family**, the MLE problem will usually have a simple analytic form

Also, due to the form of the likelihood (Gaussian) and prior (multinoulli), the MLE problem had a nice separable structure after taking the log

Can see that, when estimating the parameters of the k^{th} Gaussian (π_k, μ_k, Σ_k), we only will only need training examples from the k^{th} class, i.e., examples for which $y_{nk} = 1$

The form of this expression is important; will encounter this in GMM too



Detour: Exponential Family

Exp-fam dist also used for **Generalized Linear Models (GLM)** with $p(y|\mathbf{x}, \mathbf{w})$ modeled by an exp-fam distribution whose natural parameter is defined by $\mathbf{w}^\top \mathbf{x}$ (thus “linear”). Useful in problems where y is not real/categorical but a count, or positive real, etc



- Exponential Family is a family of prob. distributions that have the form

$$p(x|\theta) = h(x)\exp[\theta^\top T(x) - A(\theta)]$$

Lin reg, logistic reg, softmax reg
are also instances of GLMs

Even though their standard form may not look like this, they can be rewritten in this form after some algebra

- Many well-known distribution (Bernoulli, Binomial, multinoulli, Poisson, beta, gamma, Gaussian, etc.) are examples of exponential family distributions
- θ is called the **natural parameter** of the family
- $h(x)$, $T(x)$, and $A(\theta)$ are known functions (specific to the distribution)
- $T(x)$ is called the **sufficient statistics**: estimates of θ contain x in form of suff-stats
- Every exp. family distribution also has a conjugate distribution (often also in exp. family)
- Also, MLE/MAP is usually quite simple since $\log p(x|\theta)$ will have a simple expression
- Also useful in fully Bayesian inference since they have conjugate priors

Natural params are a function of the distribution parameters in the standard form



MLE for GMM

- Already saw that MLE is hard for GMM

$$\Theta_{MLE} = \operatorname{argmax}_{\Theta} \log p(\mathbf{X}|\Theta) = \operatorname{argmax}_{\Theta} \sum_{n=1}^N \log \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}_n | \mu_k, \Sigma_k)$$

- Two possible ways to solve this MLE problem

Will soon see how to
get these guesses

- If someone gave us optimal “point” guesses $\hat{\mathbf{z}}_n$ ’s of cluster ids \mathbf{z}_n ’s, we could do MLE for the parameters just like we did for generative classification with Gaussian class-conditionals

$$\Theta_{MLE} = \operatorname{argmax}_{\Theta} \log p(\mathbf{X}, \hat{\mathbf{Z}} | \Theta) = \operatorname{argmax}_{\Theta} \sum_{n=1}^N \sum_{k=1}^K \hat{z}_{nk} [\log \pi_k + \log \mathcal{N}(\mathbf{x}_n | \mu_k, \Sigma_k)]$$

In form of a probability distribution
instead of a single “optimal” guess

- Alternatively, if someone gave a “probabilistic” guess of \mathbf{z}_n ’s, we can do MLE for Θ as follows

$$\Theta_{MLE} = \operatorname{argmax}_{\Theta} \mathbb{E}[\log p(\mathbf{X}, \mathbf{Z} | \Theta)] = \operatorname{argmax}_{\Theta} \sum_{n=1}^N \sum_{k=1}^K \mathbb{E}[z_{nk}] [\log \pi_k + \log \mathcal{N}(\mathbf{x}_n | \mu_k, \Sigma_k)]$$

Similar to Approach 1 but
maximizes an expectation

The expectation is w.r.t a distribution of Z which we will see shortly

- Approach 1 is ALT-OPT and Approach 2 is Expectation Maximization (“soft” ALT-OPT). Both require alternating between estimating \mathbf{Z} and Θ until convergence



ALT-OPT for GMM



Keep in mind: In LVMs, assuming i.i.d. data, the quantity $\log p(\mathbf{X}|\Theta) = \sum_{n=1}^N \log p(\mathbf{x}_n|\Theta)$ is called **incomplete data log-likelihood (ILL)** whereas $\log p(\mathbf{X}, \mathbf{Z}|\Theta) = \sum_{n=1}^N \log p(\mathbf{x}_n, \mathbf{z}_n|\Theta)$ is called **complete data log-likelihood (CLL)**. Goal is to maximize ILL but ALT-OPT maximizes CLL (EM too will maximize the expectation of CLL). The latent vars \mathbf{z}_n 's "complete" the data \mathbf{x}_n ☺

- We will assume we have a “hard” (most probable) guess of \mathbf{z}_n , say $\hat{\mathbf{z}}_n$

- Then ALT-OPT would look like this

- Initialize $\Theta = \{\pi_k, \mu_k, \Sigma_k\}_{k=1}^K$ as $\widehat{\Theta}$
- Repeat the following until convergence
 - For each n , compute most probable value (our best guess) of \mathbf{z}_n as

$$\hat{\mathbf{z}}_n = \operatorname{argmax}_{k=1,2,\dots,K} p(z_n = k | \widehat{\Theta}, \mathbf{x}_n)$$

Proportional to prior prob times likelihood, i.e.,
 $p(z_n = k|\Theta) p(x_n|z_n = k, \Theta) = \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)$

Posterior probability of point \mathbf{x}_n belonging to cluster k

- Solve MLE problem for Θ using most probable \mathbf{z}_n 's

$$\widehat{\Theta} = \operatorname{argmax}_{\Theta} \sum_{n=1}^N \sum_{k=1}^K \hat{z}_{nk} [\log \pi_k + \log \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)]$$

Same objective function as generative K -class classification with Gaussian class-conditionals

Note: The objective function is $\sum_{n=1}^N \log p(\mathbf{x}_n, \hat{\mathbf{z}}_n|\Theta) = \sum_{n=1}^N \log p(\hat{\mathbf{z}}_n|\Theta) + \log p(\mathbf{x}_n|\hat{\mathbf{z}}_n, \Theta)$

$$\begin{aligned}\hat{\pi}_k &= \frac{1}{N} \sum_{n=1}^N \hat{z}_{nk} & N_k : \text{Effective number of points in cluster } k \\ \hat{\mu}_k &= \frac{1}{N_k} \sum_{n=1}^N \hat{z}_{nk} \mathbf{x}_n \\ \hat{\Sigma}_k &= \frac{1}{N_k} \sum_{n=1}^N \hat{z}_{nk} (\mathbf{x}_n - \hat{\mu}_k)(\mathbf{x}_n - \hat{\mu}_k)^T\end{aligned}$$

Does that matter? Should we worry that we aren't solving the actual problem anymore?

Not really; will see the justification soon ☺

But wait! This is not the same as $\sum_{n=1}^N \log p(\mathbf{x}_n|\Theta)$ which was the original MLE objective for this LVM ☹



Expectation-Maximization (EM) for GMM

- EM finds Θ_{MLE} by maximizing $\mathbb{E}[\log p(\mathbf{X}, \mathbf{Z}|\Theta)]$ rather than $\log p(\mathbf{X}, \widehat{\mathbf{Z}}|\Theta)$
- Note: Expectation will be w.r.t. the conditional posterior distribution of \mathbf{Z} , i.e., $p(\mathbf{Z}|\mathbf{X}, \Theta)$
- The EM algorithm for GMM operates as follows
 - Initialize $\Theta = \{\pi_k, \mu_k, \Sigma_k\}_{k=1}^K$ as $\widehat{\Theta}$
 - Repeat until convergence
 - Compute conditional posterior $p(\mathbf{Z}|\mathbf{X}, \widehat{\Theta})$. Since obs are i.i.d, compute separately for each n (and for $k = 1, 2, \dots, K$)
 - Update Θ by maximizing the expected complete data log-likelihood

Expectation of CLL

.. which we maximized
in ALT-OPT

It is "conditional" posterior
because it is also conditioned
on Θ , not just data X

Why w.r.t. this distribution?
Will see justification in a bit

Requires knowing Θ

Needed to get the expected CLL

Same as $p(z_{nk} = 1 | \mathbf{x}_n, \widehat{\Theta})$, just a
different notation

$$p(\mathbf{z}_n = k | \mathbf{x}_n, \widehat{\Theta}) \propto p(\mathbf{z}_n = k | \widehat{\Theta}) p(\mathbf{x}_n | \mathbf{z}_n = k, \widehat{\Theta}) = \hat{\pi}_k \mathcal{N}(\mathbf{x}_n | \hat{\mu}_k, \hat{\Sigma}_k)$$

- Update Θ by maximizing the expected complete data log-likelihood

$$\widehat{\Theta} = \operatorname{argmax}_{\Theta} \mathbb{E}_{p(\mathbf{Z}|\mathbf{X}, \widehat{\Theta})} [\log p(\mathbf{X}, \mathbf{Z}|\Theta)] = \sum_{n=1}^N \mathbb{E}_{p(\mathbf{z}_n|\mathbf{x}_n, \widehat{\Theta})} [\log p(\mathbf{x}_n, \mathbf{z}_n|\Theta)]$$

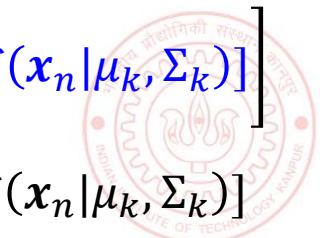
$$= \operatorname{argmax}_{\Theta} \mathbb{E} \left[\sum_{n=1}^N \sum_{k=1}^K z_{nk} [\log \pi_k + \log \mathcal{N}(\mathbf{x}_n | \mu_k, \Sigma_k)] \right]$$

$$= \operatorname{argmax}_{\Theta} \sum_{n=1}^N \sum_{k=1}^K \mathbb{E}[z_{nk}] [\log \pi_k + \log \mathcal{N}(\mathbf{x}_n | \mu_k, \Sigma_k)]$$

$$\hat{\pi}_k = \frac{1}{N} \sum_{n=1}^N \mathbb{E}[z_{nk}] \quad \hat{\mu}_k = \frac{1}{N_k} \sum_{n=1}^N \mathbb{E}[z_{nk}] \mathbf{x}_n$$

N_k : Effective number
of points in cluster k

$$\hat{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^N \mathbb{E}[z_{nk}] (\mathbf{x}_n - \hat{\mu}_k)(\mathbf{x}_n - \hat{\mu}_k)^T$$



EM for GMM (Contd)

- The EM algo for GMM required $\mathbb{E}[z_{nk}]$. Note $z_{nk} \in \{0,1\}$

$$\mathbb{E}[z_{nk}] = \gamma_{nk} = 0 \times p(z_{nk} = 0 | x_n, \hat{\Theta}) + 1 \times p(z_{nk} = 1 | x_n, \hat{\Theta}) = p(z_{nk} = 1 | x_n, \hat{\Theta}) \propto \hat{\pi}_k \mathcal{N}(x_n | \hat{\mu}_k, \hat{\Sigma}_k)$$

Reason: $\sum_{k=1}^K \gamma_{nk} = 1$

Need to normalize: $\mathbb{E}[z_{nk}] = \frac{\hat{\pi}_k \mathcal{N}(x_n | \hat{\mu}_k, \hat{\Sigma}_k)}{\sum_{\ell=1}^K \hat{\pi}_\ell \mathcal{N}(x_n | \hat{\mu}_\ell, \hat{\Sigma}_\ell)}$

EM for Gaussian Mixture Model

① Initialize $\Theta = \{\pi_k, \mu_k, \Sigma_k\}_{k=1}^K$ as $\Theta^{(0)}$, set $t = 1$

② E step: compute the expectation of each z_n (we need it in M step)

Soft K-means, which are more of a heuristic to get soft-clustering, also gave us probabilities but didn't account for cluster shapes or fraction of points in each cluster

Accounts for fraction of points in each cluster

$$\mathbb{E}[z_{nk}^{(t)}] = \gamma_{nk}^{(t)} = \frac{\pi_k^{(t-1)} \mathcal{N}(x_n | \mu_k^{(t-1)}, \Sigma_k^{(t-1)})}{\sum_{\ell=1}^K \pi_\ell^{(t-1)} \mathcal{N}(x_n | \mu_\ell^{(t-1)}, \Sigma_\ell^{(t-1)})} \quad \forall n, k$$

Accounts for cluster shapes (since each cluster is a Gaussian)

③ Given "responsibilities" $\gamma_{nk} = \mathbb{E}[z_{nk}]$, and $N_k = \sum_{n=1}^N \gamma_{nk}$, re-estimate Θ via MLE

$$\mu_k^{(t)} = \frac{1}{N_k} \sum_{n=1}^N \gamma_{nk}^{(t)} x_n$$

Effective number of points in the k^{th} cluster

M-step: $\Sigma_k^{(t)} = \frac{1}{N_k} \sum_{n=1}^N \gamma_{nk}^{(t)} (x_n - \mu_k^{(t)}) (x_n - \mu_k^{(t)})^\top$

$$\pi_k^{(t)} = \frac{N_k}{N}$$

④ Set $t = t + 1$ and go to step 2 if not yet converged



LVMs (Contd), Expectation Maximization (2)

CS771: Introduction to Machine Learning

Piyush Rai

What is EM Doing?

2

- The MLE problem was $\Theta_{MLE} = \operatorname{argmax}_{\Theta} \log p(\mathbf{X}|\Theta) = \operatorname{argmax}_{\Theta} \log \sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\Theta)$
 - Maximizing ILL
 - Assuming \mathbf{Z} to be discrete, else replace it by an integral
- What EM (and ALT-OPT in a crude way) did is max of CLL: $\Theta_{MLE} = \operatorname{argmax}_{\Theta} \mathbb{E}[\log p(\mathbf{X}, \mathbf{Z}|\Theta)]$
 - Easier than maximizing ILL
- But we did not solve the original problem. Is it okay?
- Assume $p_z = p(\mathbf{Z}|\mathbf{X}, \Theta)$ and $q(\mathbf{Z})$ to be some prob distribution over \mathbf{Z} , then

Function of a distribution q and parameter Θ

$$\log p(\mathbf{X}|\Theta) = \mathcal{L}(q, \Theta) + KL(q||p_z)$$

May verify this identity

- In the above $\mathcal{L}(q, \Theta) = \sum_{\mathbf{Z}} q(\mathbf{Z}) \log \left\{ \frac{p(\mathbf{X}, \mathbf{Z}|\Theta)}{q(\mathbf{Z})} \right\}$ and $KL(q||p_z) = -\sum_{\mathbf{Z}} q(\mathbf{Z}) \log \left\{ \frac{p(\mathbf{Z}|\mathbf{X}, \Theta)}{q(\mathbf{Z})} \right\}$
- Since KL is always non-negative $\log p(\mathbf{X}|\Theta) \geq \mathcal{L}(q, \Theta)$, so $\mathcal{L}(q, \Theta)$ is a lower-bound on ILL
- Thus if we maximize $\mathcal{L}(q, \Theta)$, it will also improve $\log p(\mathbf{X}|\Theta)$



What is EM Doing?

- As we saw, $\mathcal{L}(q, \Theta)$ depends on q and Θ
- Let's maximize $\mathcal{L}(q, \Theta)$ w.r.t. q with Θ fixed at Θ^{old}

The posterior distribution of Z given older parameters Θ^{old} (will need this posterior to get the expectation of CLL)

$$\hat{q} = \operatorname{argmax}_q \mathcal{L}(q, \Theta^{\text{old}}) = \operatorname{argmin}_q KL(q || p_z) = p_z = p(Z|X, \Theta^{\text{old}})$$

Since $\log p(X|\Theta) = \mathcal{L}(q, \Theta) + KL(q || p_z)$ is constant when Θ is held fixed at Θ^{old}

- Now let's maximize $\mathcal{L}(q, \Theta)$ w.r.t. Θ with q fixed at $\hat{q} = p_z = p(Z|X, \Theta^{\text{old}})$

$$\begin{aligned}\Theta^{\text{new}} &= \operatorname{argmax}_{\Theta} \mathcal{L}(\hat{q}, \Theta) = \operatorname{argmax}_{\Theta} \sum_Z p(Z|X, \Theta^{\text{old}}) \log \left\{ \frac{p(X, Z|\Theta)}{p(Z|X, \Theta^{\text{old}})} \right\} \\ &= \operatorname{argmax}_{\Theta} \sum_Z p(Z|X, \Theta^{\text{old}}) \log p(X, Z|\Theta) \\ &= \operatorname{argmax}_{\Theta} \mathbb{E}_{p(Z|X, \Theta^{\text{old}})} [\log p(X, Z|\Theta)] \\ &= \operatorname{argmax}_{\Theta} Q(\Theta, \Theta^{\text{old}})\end{aligned}$$

Maximization of expected CLL w.r.t. the posterior distribution of Z given older parameters Θ^{old}



The EM Algorithm in its general form..

- Maximization of $\mathcal{L}(q, \Theta)$ w.r.t. q and Θ gives the EM algorithm (Dempster, Laird, Rubin, 1977)

The EM Algorithm

- Initialize Θ as $\Theta^{(0)}$, set $t = 1$
- Step 1: Compute **posterior** of latent variables given current parameters $\Theta^{(t-1)}$

$$p(\mathbf{z}_n^{(t)} | \mathbf{x}_n, \Theta^{(t-1)}) = \frac{p(\mathbf{z}_n^{(t)} | \Theta^{(t-1)}) p(\mathbf{x}_n | \mathbf{z}_n^{(t)}, \Theta^{(t-1)})}{p(\mathbf{x}_n | \Theta^{(t-1)})} \propto \text{prior} \times \text{likelihood}$$

- Step 2: Now maximize the **expected complete data log-likelihood** w.r.t. Θ

$$\Theta^{(t)} = \arg \max_{\Theta} \mathcal{Q}(\Theta, \Theta^{(t-1)}) = \arg \max_{\Theta} \sum_{n=1}^N \mathbb{E}_{p(\mathbf{z}_n^{(t)} | \mathbf{x}_n, \Theta^{(t-1)})} [\log p(\mathbf{x}_n, \mathbf{z}_n^{(t)} | \Theta)]$$

- If not yet converged, set $t = t + 1$ and go to step 2.

- Note: If we can take the MAP estimate $\hat{\mathbf{z}}_n$ of \mathbf{z}_n (not full posterior) in Step 1 and maximize the CLL in Step 2 using that, i.e., do $\operatorname{argmax}_{\Theta} \sum_{n=1}^N [\log p(\mathbf{x}_n, \hat{\mathbf{z}}_n^{(t)} | \Theta)]$ this will be ALT-OPT



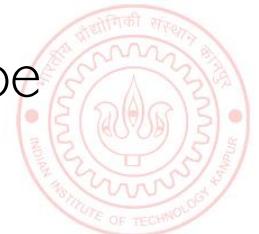
The Expected CLL

- Expected CLL in EM is given by (assume observations are i.i.d.)

$$\begin{aligned} \mathcal{Q}(\Theta, \Theta^{old}) &= \sum_{n=1}^N \mathbb{E}_{p(z_n|x_n, \Theta^{old})} [\log p(x_n, z_n | \Theta)] \\ &= \sum_{n=1}^N \mathbb{E}_{p(z_n|x_n, \Theta^{old})} [\log p(x_n|z_n, \Theta) + \log p(z_n | \Theta)] \end{aligned}$$

Was indeed the case of GMM: $p(z_n|\Theta)$ was multinoulli, $p(x_n|z_n, \Theta)$ was Gaussian

- If $p(z_n|\Theta)$ and $p(x_n|z_n, \Theta)$ are exp-family distributions, $\mathcal{Q}(\Theta, \Theta^{old})$ has a very simple form
- In resulting expressions, replace terms containing z_n 's by their respective expectations, e.g.,
 - z_n replaced by $\mathbb{E}_{p(z_n|x_n, \widehat{\Theta})}[z_n]$
 - $z_n z_n^\top$ replaced by $\mathbb{E}_{p(z_n|x_n, \widehat{\Theta})}[z_n z_n^\top]$
- However, in some LVMs, these expectations are intractable to compute and need to be approximated (beyond the scope of CS771)



EM: An Illustration

- As we saw, EM maximizes the lower bound $\mathcal{L}(q, \Theta)$ in two steps
- Step 1 finds the optimal q (call it \hat{q}) by setting it the posterior of Z given current Θ
- Step 2 maximizes $\mathcal{L}(\hat{q}, \Theta)$ w.r.t. Θ which gives a new Θ .

Alternating between them until convergence to some local optima

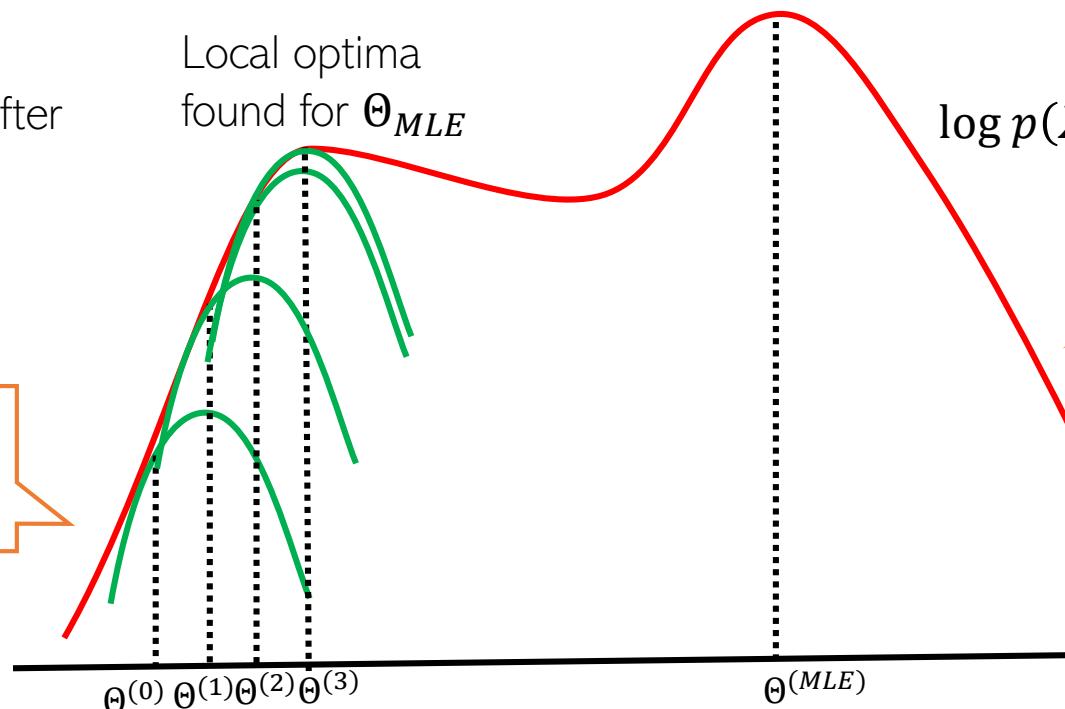
Makes $\mathcal{L}(q, \Theta)$ equal to $\log p(\mathbf{X}|\Theta)$; thus the curves touch at current Θ

Green curve: $\mathcal{L}(\hat{q}, \Theta)$ after setting q to \hat{q}

Local optima found for Θ_{MLE}

$\log p(\mathbf{X}|\Theta)$

Good initialization matters; otherwise would converge to a poor local optima



Note that Θ only changes in Step 2 so the objective $\log p(\mathbf{X}|\Theta)$ can only change in Step 2



Also kind of similar to Newton's method (and has second order like convergence behavior in some cases)

Unlike Newton's method, we don't construct and optimize a quadratic approximation, but a lower bound

Even though original MLE problem $\operatorname{argmax}_{\Theta} \log p(\mathbf{X}|\Theta)$ could be solved using gradient methods, EM often works faster and has cleaner updates

Recap: ALT-OPT vs EM

- ALT-OPT does the following

- ① Initialize $\Theta = \hat{\Theta}$
- ② Estimate \mathbf{Z} as $\hat{\mathbf{Z}} = \arg \max_{\mathbf{Z}} \log p(\mathbf{Z}|\mathbf{X}, \hat{\Theta})$
- ③ Estimate Θ as $\hat{\Theta} = \arg \max_{\Theta} \log p(\mathbf{X}, \hat{\mathbf{Z}}|\Theta)$
- ④ Go to step 2 if not converged

This step could potentially throw away a lot of information about the latent variable \mathbf{Z}

- EM addresses it using “soft” version of ALT-OPT

- ① Initialize $\Theta = \hat{\Theta}$
- ② Compute the posterior distribution of \mathbf{Z} , i.e., $p(\mathbf{Z}|\mathbf{X}, \hat{\Theta})$
- ③ Estimate Θ by maximizing the expected CLL $\hat{\Theta} = \mathbb{E}_{p(\mathbf{Z}|\mathbf{X}, \hat{\Theta})} [\log p(\mathbf{X}, \mathbf{Z}|\Theta)]$
- ④ Go to step 2 if not converged

ALT-OPT can be seen as an approximation of EM – the posterior $p(\mathbf{Z}|\mathbf{X}, \Theta)$ is replaced by a point mass at its mode



EM: Some Comments

- The E and M steps may not always be possible to perform exactly. Some reasons
 - Posterior of latent variables $p(\mathbf{Z}|\mathbf{X}, \Theta)$ may not be easy to find and may require approx.
 - Even if $p(\mathbf{Z}|\mathbf{X}, \Theta)$ is easy, expected CLL, i.e., $\mathbb{E}[\log p(\mathbf{X}, \mathbf{Z}|\Theta)]$ may still not be tractable

$$\mathbb{E}[\log p(\mathbf{X}, \mathbf{Z}|\Theta)] = \int \log p(\mathbf{X}, \mathbf{Z}|\Theta) p(\mathbf{Z}|\mathbf{X}, \Theta) d\mathbf{Z}$$

Monte-Carlo EM

..and may need to be approximated, e.g., using Monte-Carlo expectation

Gradient methods may still be needed for this step

- Maximization of the expected CLL may not be possible in closed form
- EM works even if the M step is only solved approximately ([Generalized EM](#))
- If M step has multiple parameters whose updates depend on each other, they are updated in an alternating fashion - called [Expectation Conditional Maximization \(ECM\)](#) algorithm
- Other advanced probabilistic inference algorithms are based on ideas similar to EM
 - E.g., Variational Bayesian inference a.k.a. [Variational Inference \(VI\)](#)
- EM is also related to non-convex optimization algorithms [Majorization-Maximization \(MM\)](#)



LVMs for Dimensionality Reduction

CS771: Introduction to Machine Learning

Piyush Rai

Plan

- A latent variable model for dimensionality reduction
 - Probabilistic PCA
- Expectation maximization (EM) algorithm for MLE for PPCA



Probabilistic PCA (PPCA)

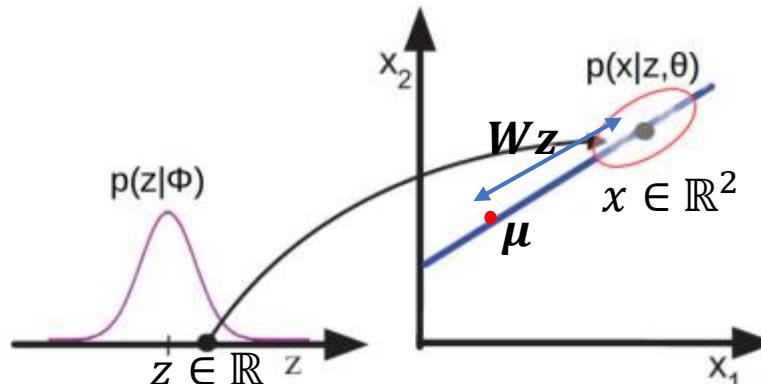
- Assume obs $\mathbf{x}_n \in \mathbb{R}^D$ as a linear mapping of a latent var $\mathbf{z}_n \in \mathbb{R}^K$ + Gaussian noise

$$\mathbf{x}_n = \boldsymbol{\mu} + \mathbf{W}\mathbf{z}_n + \boldsymbol{\epsilon}_n$$

D × 1 offset
D × K matrix
Drawn from a zero-mean D-dim Gaussian $\mathcal{N}(\mathbf{0}, \sigma^2 I_D)$

- Equivalent to saying $p(\mathbf{x}_n | \mathbf{z}_n, \boldsymbol{\mu}, \mathbf{W}, \sigma^2) = \mathcal{N}(\boldsymbol{\mu} + \mathbf{W}\mathbf{z}_n, \sigma^2 I_D)$
- Assume a zero-mean Gaussian prior on \mathbf{z}_n , so $p(\mathbf{z}_n) = \mathcal{N}(\mathbf{0}, I_K)$

A “reverse” (generative) way of thinking: first generate a low-dim latent variable \mathbf{z}_n and then map it to generate the high-dim observation \mathbf{x}_n



Need to estimate fewer parameters ($DK + D + 1$ as opposed to $O(D^2)$)

Thus PPCA does a low-rank approximation of the covariance matrix

- Joint distr. of \mathbf{x}_n and \mathbf{z}_n is Gaussian (since $p(\mathbf{x}_n | \mathbf{z}_n)$ and $p(\mathbf{z}_n)$ are individually Gaussian) and the marginal distribution of \mathbf{x}_n will be Gaussian

$$p(\mathbf{x}_n | \mathbf{W}, \sigma^2) = N(\mathbf{x}_n | \boldsymbol{\mu}, \mathbf{W}\mathbf{W}^\top + \sigma^2 I_D)$$

As $\sigma^2 \rightarrow 0$, the covariance become approx low-rank (rank K) and only $DK + D + 1$ params needed, as opposed to $O(D^2)$ for the full covariance

Benefits of Generative Models for Dim-Red

- One benefit: Once model parameters are learned, we can even generate new data, e.g.,
 - Generate a random \mathbf{z}_n from $\mathcal{N}(\mathbf{0}, I_K)$
 - Generate \mathbf{x}_n condition on \mathbf{z}_n from $\mathcal{N}(\boldsymbol{\mu} + \mathbf{W}\mathbf{z}_n, \sigma^2 I_D)$



(a) Training data



(b) Random samples

Generated using a more sophisticated generative model, not PPCA (but similar in formulation)

- Many other benefits. For example, can do dim-red, even if \mathbf{x}_n has part of it as missing.



Learning PPCA using EM

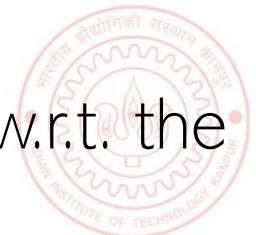
- The ILL is $p(\mathbf{x}_n | \boldsymbol{\mu}, \mathbf{W}, \sigma^2) = N(\mathbf{x}_n | \boldsymbol{\mu}, \mathbf{W}\mathbf{W}^\top + \sigma^2 I_D)$ with \mathbf{z}_n integrated out
- Ignoring $\boldsymbol{\mu}$ for notational simplicity, ILL is $p(\mathbf{x}_n | \mathbf{W}, \sigma^2) = N(\mathbf{x}_n | \mathbf{0}, \mathbf{W}\mathbf{W}^\top + \sigma^2 I_D)$
- Can maximize ILL but requires solving eigen-decomposition (PRML: 12.2.1)
- EM will instead maximize expected CLL, with CLL given by

$$\log p(\mathbf{X}, \mathbf{Z} | \mathbf{W}, \sigma^2) = \log \prod_{n=1}^N p(\mathbf{x}_n, \mathbf{z}_n | \mathbf{W}, \sigma^2) = \log \prod_{n=1}^N p(\mathbf{x}_n | \mathbf{z}_n, \mathbf{W}, \sigma^2) p(\mathbf{z}_n) = \sum_{n=1}^N \{\text{log } p(\mathbf{x}_n | \mathbf{z}_n, \mathbf{W}, \sigma^2) + \text{log } p(\mathbf{z}_n)\}$$

- Using $p(\mathbf{x}_n | \mathbf{z}_n, \mathbf{W}, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{D/2}} \exp \left[-\frac{(\mathbf{x}_n - \mathbf{W}\mathbf{z}_n)^\top (\mathbf{x}_n - \mathbf{W}\mathbf{z}_n)}{2\sigma^2} \right]$, $p(\mathbf{z}_n) \propto \exp \left[-\frac{\mathbf{z}_n^\top \mathbf{z}_n}{2} \right]$ and simplifying

$$\text{CLL} = - \sum_{n=1}^N \left\{ \frac{D}{2} \log \sigma^2 + \frac{1}{2\sigma^2} \|\mathbf{x}_n\|^2 - \frac{1}{\sigma^2} \mathbf{z}_n^\top \mathbf{W}^\top \mathbf{x}_n + \frac{1}{2\sigma^2} \text{tr}(\mathbf{z}_n \mathbf{z}_n^\top \mathbf{W}^\top \mathbf{W}) + \frac{1}{2} \text{tr}(\mathbf{z}_n \mathbf{z}_n^\top) \right\}$$

- Expected CLL will need $\mathbb{E}[\mathbf{z}_n]$ and $\mathbb{E}[\mathbf{z}_n \mathbf{z}_n^\top]$ where the expectations are w.r.t. the conditional posterior of \mathbf{z}_n



Learning PPCA using EM

- The EM algo for PPCA alternates between τ steps

- Compute conditional posterior of \mathbf{z}_n given parameters $\Theta = (\mathbf{W}, \sigma^2)$

$$p(\mathbf{z}_n | \mathbf{x}_n, \mathbf{W}, \sigma^2) = \mathcal{N}(\mathbf{M}^{-1} \mathbf{W}^\top \mathbf{x}_n, \sigma^2 \mathbf{M}^{-1}) \quad (\text{where } \mathbf{M} = \mathbf{W}^\top \mathbf{W} + \sigma^2 \mathbf{I}_K)$$

- Maximize the expected CLL $\mathbb{E}[\log p(\mathbf{X}, \mathbf{Z} | \mathbf{W}, \sigma^2)]$ w.r.t. Θ

$$-\sum_{n=1}^N \left\{ \frac{D}{2} \log \sigma^2 + \frac{1}{2\sigma^2} \|\mathbf{x}_n\|^2 - \frac{1}{\sigma^2} \mathbb{E}[\mathbf{z}_n]^\top \mathbf{W}^\top \mathbf{x}_n + \frac{1}{2\sigma^2} \text{tr}(\mathbb{E}[\mathbf{z}_n \mathbf{z}_n^\top] \mathbf{W}^\top \mathbf{W}) + \frac{1}{2} \text{tr}(\mathbb{E}[\mathbf{z}_n \mathbf{z}_n^\top]) \right\}$$

- Taking derivative of expected CLL w.r.t. \mathbf{W} and setting to zero gives

$$\mathbf{W} = \left[\sum_{n=1}^N \mathbf{x}_n \mathbb{E}[\mathbf{z}_n]^\top \right] \left[\sum_{n=1}^N \mathbb{E}[\mathbf{z}_n \mathbf{z}_n^\top] \right]^{-1}$$

Can likewise estimate σ^2 as well

- Required expectations can be found from the conditional posterior of \mathbf{z}_n

$$p(\mathbf{z}_n | \mathbf{x}_n, \mathbf{W}) = \mathcal{N}(\mathbf{M}^{-1} \mathbf{W}^\top \mathbf{x}_n, \sigma^2 \mathbf{M}^{-1}) \quad \text{where } \mathbf{M} = \mathbf{W}^\top \mathbf{W} + \sigma^2 \mathbf{I}_K$$

$$\mathbb{E}[\mathbf{z}_n] = \mathbf{M}^{-1} \mathbf{W}^\top \mathbf{x}_n$$

$$\mathbb{E}[\mathbf{z}_n \mathbf{z}_n^\top] = \mathbb{E}[\mathbf{z}_n] \mathbb{E}[\mathbf{z}_n]^\top + \text{cov}(\mathbf{z}_n) = \mathbb{E}[\mathbf{z}_n] \mathbb{E}[\mathbf{z}_n]^\top + \sigma^2 \mathbf{M}^{-1}$$



Full EM algo for PPCA

- Specify K , initialize \mathbf{W} and σ^2 randomly. Also center the data ($\mathbf{x}_n = \mathbf{x}_n - \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n$)
- E step:** For each n , compute $p(\mathbf{z}_n | \mathbf{x}_n)$ using current \mathbf{W} and σ^2 . Compute exp. for the M step

$$\begin{aligned} p(\mathbf{z}_n | \mathbf{x}_n, \mathbf{W}) &= \mathcal{N}(\mathbf{M}^{-1} \mathbf{W}^\top \mathbf{x}_n, \sigma^2 \mathbf{M}^{-1}) \quad \text{where } \mathbf{M} = \mathbf{W}^\top \mathbf{W} + \sigma^2 \mathbf{I}_K \\ \mathbb{E}[\mathbf{z}_n] &= \mathbf{M}^{-1} \mathbf{W}^\top \mathbf{x}_n \\ \mathbb{E}[\mathbf{z}_n \mathbf{z}_n^\top] &= \text{cov}(\mathbf{z}_n) + \mathbb{E}[\mathbf{z}_n] \mathbb{E}[\mathbf{z}_n]^\top = \mathbb{E}[\mathbf{z}_n] \mathbb{E}[\mathbf{z}_n]^\top + \sigma^2 \mathbf{M}^{-1} \end{aligned}$$

- M step:** Re-estimate \mathbf{W} and σ^2

$$\begin{aligned} \mathbf{W}_{new} &= \left[\sum_{n=1}^N \mathbf{x}_n \mathbb{E}[\mathbf{z}_n]^\top \right] \left[\sum_{n=1}^N \mathbb{E}[\mathbf{z}_n \mathbf{z}_n^\top] \right]^{-1} \\ \sigma_{new}^2 &= \frac{1}{ND} \sum_{n=1}^N \left\{ \|\mathbf{x}_n\|^2 - 2\mathbb{E}[\mathbf{z}_n]^\top \mathbf{W}_{new}^\top \mathbf{x}_n + \text{tr}(\mathbb{E}[\mathbf{z}_n \mathbf{z}_n^\top] \mathbf{W}_{new}^\top \mathbf{W}_{new}) \right\} \end{aligned}$$

- Set $\mathbf{W} = \mathbf{W}_{new}$ and $\sigma^2 = \sigma_{new}^2$. If not converged (monitor $p(\mathbf{X} | \Theta)$), go back to E step
- Note:** For $\sigma^2 = 0$, this EM algorithm can also be used to efficiently solve standard PCA (note that this EM algorithm doesn't require any eigen-decomposition)

Other Generative Models for Dim-Red

- Factor Analysis is similar to PPCA except that the noise covariance of a diagonal matrix instead of $\sigma^2 I$
- Can use a mixture of probabilistic PCA for nonlinear dimensionality reduction
 - Data assumed to come from a mixture of low-rank Gaussians
 - Each low-rank Gaussian is a PPCA model
 - Basically does clustering + dimensionality reduction in each cluster
- Variational auto-encoders (VAE): z_n to x_n mapping is defined by deep neural net
 - Will look at VAE and GAN briefly when talking about deep learning
- Generative adversarial networks (GAN) are models that can only generate
 - Some variants of GANs (e.g., bi-directional GAN) can also be used to learn z_n from x_n



Coming up next

- Deep neural networks



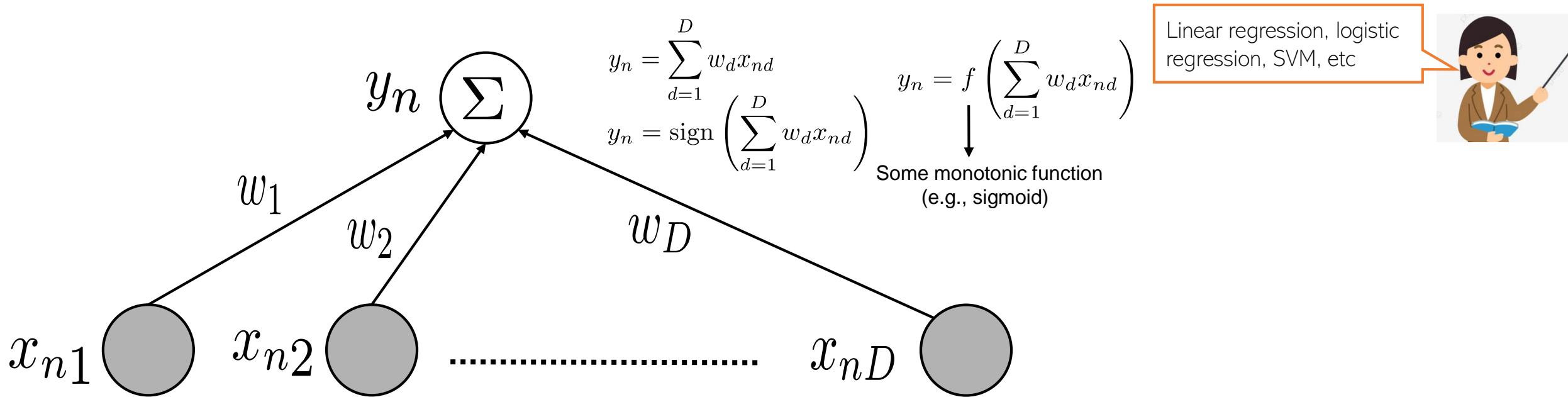
Introduction to Deep Learning (1)

CS771: Introduction to Machine Learning

Piyush Rai

Limitation of Linear Models

- Linear models: Output produced by taking a linear combination of input features



- This basic architecture is classically also known as the “Perceptron” (not to be confused with the Perceptron “algorithm”, which learns a linear classification model)
 - Although can kernelize to make them nonlinear
- This can't however learn nonlinear functions or nonlinear decision boundaries

Neural Networks: Multi-layer Perceptron (MLP)

- An MLP consists of an **input layer**, an **output layer**, and **one or more hidden layers**

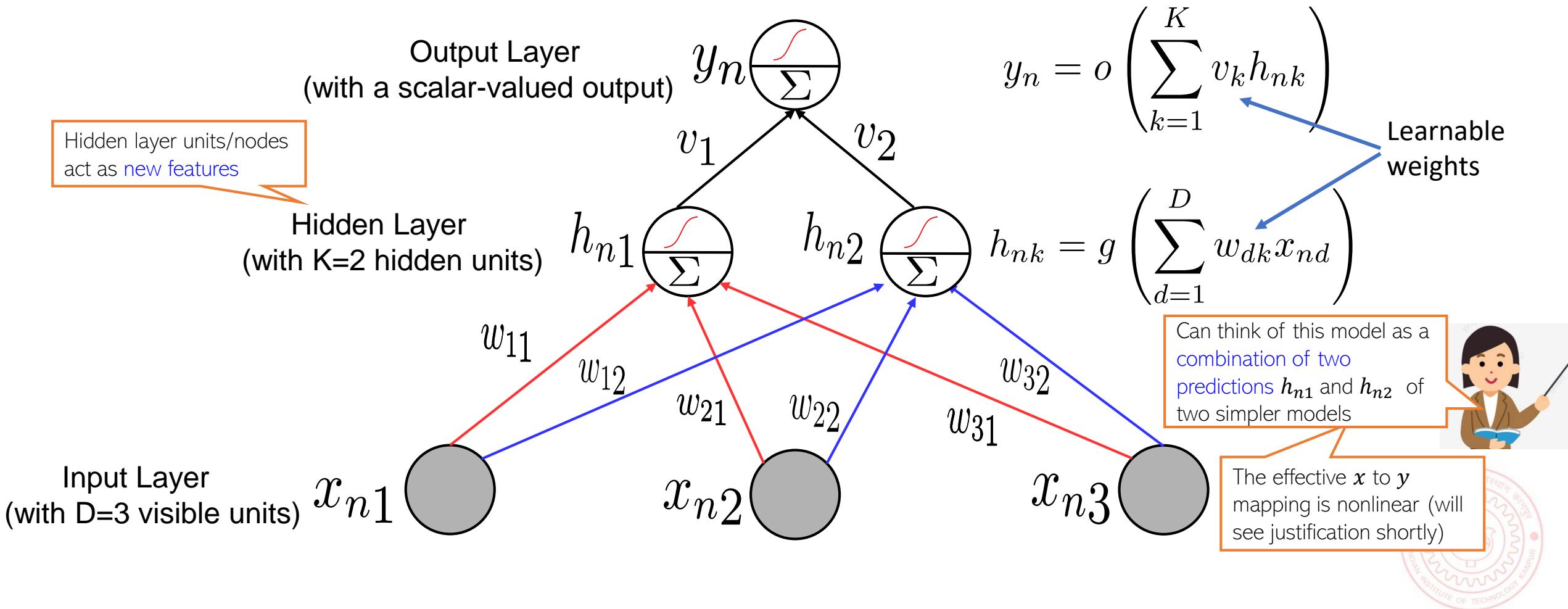


Illustration: Neural Net with One Hidden Layer

- Each input \mathbf{x}_n transformed into several pre-activations using linear models

$$a_{nk} = \mathbf{w}_k^\top \mathbf{x}_n = \sum_{d=1}^D w_{dk} x_{nd}$$

- Nonlinear activation applied on each pre-act.

$$h_{nk} = g(a_{nk})$$

- Linear model learned on the new “features” \mathbf{h}_n

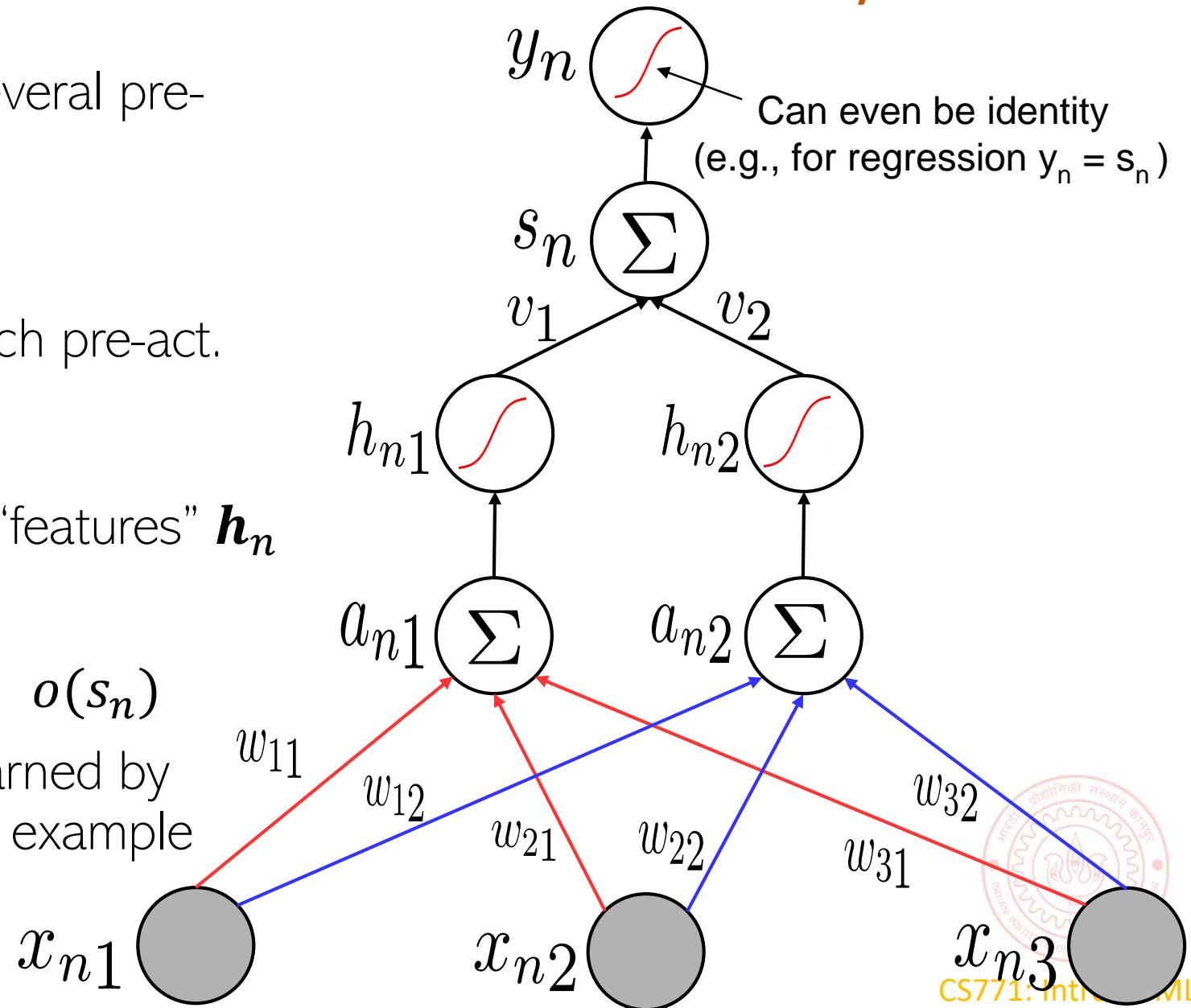
$$s_n = \mathbf{v}^\top \mathbf{h}_n = \sum_{k=1}^K v_k h_{nk}$$

- Finally, output is produced as $y = o(s_n)$

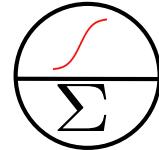
- Unknowns ($\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K, \mathbf{v}$) learned by minimizing some loss function, for example

$$\mathcal{L}(\mathbf{W}, \mathbf{v}) = \sum_{n=1}^N \ell(y_n, o(s_n))$$

(squared, logistic, softmax, etc)

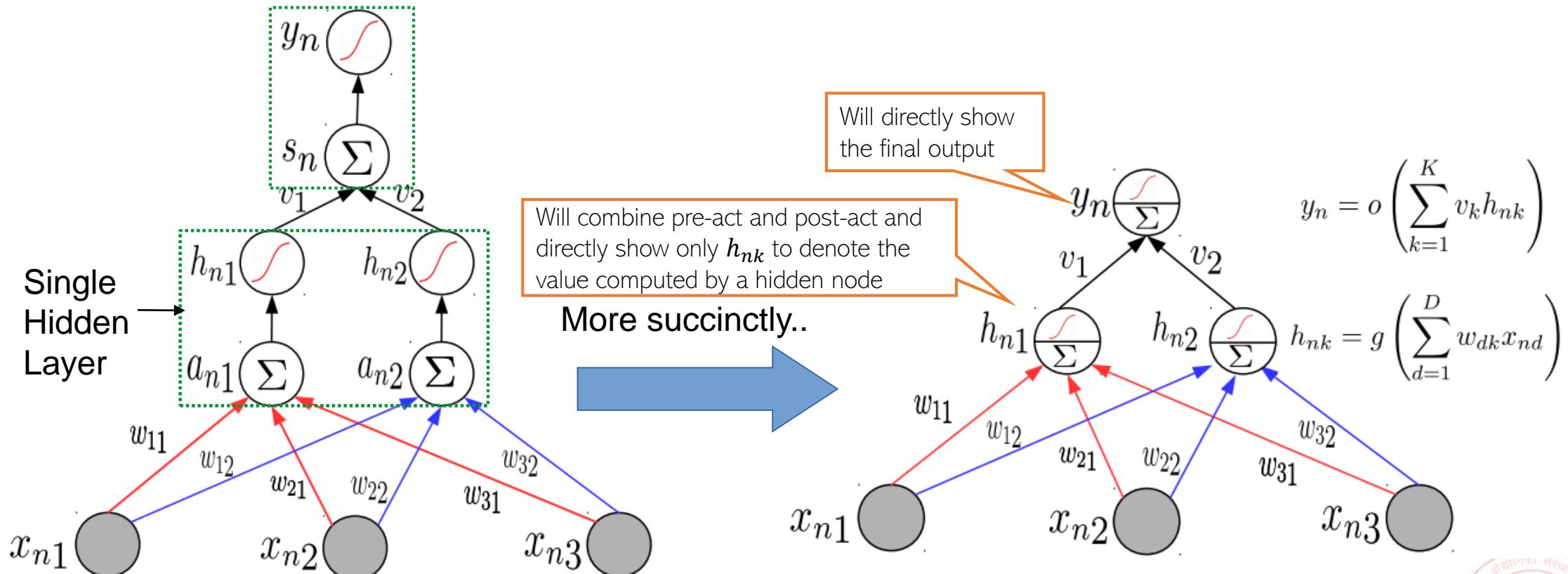


Neural Nets: A Compact Illustration



Will denote a linear combination of inputs followed by a nonlinear operation on the result

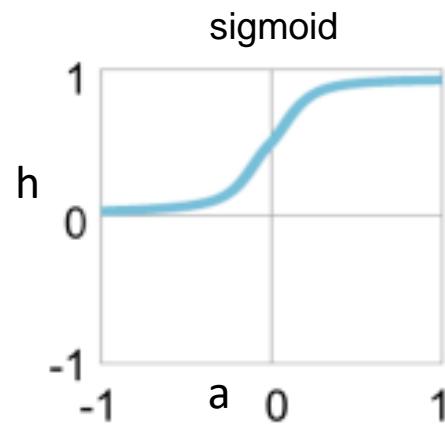
- Note: Hidden layer pre-act a_{nk} and post-act h_{nk} will be shown together for brevity



- Different layers may use different non-linear activations. Output layer may have none.

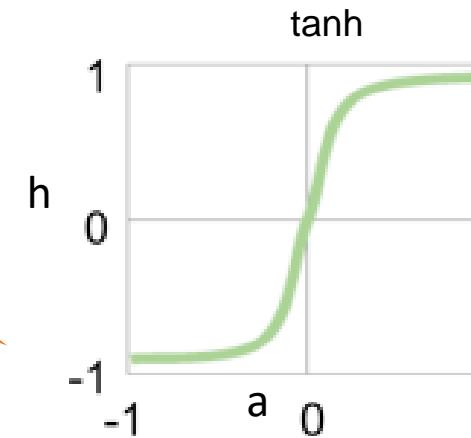


Activation Functions: Some Common Choices



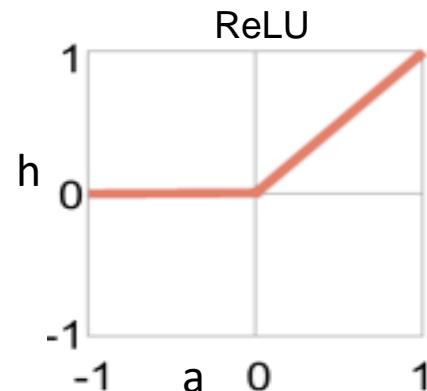
$$\text{Sigmoid: } h = \sigma(a) = \frac{1}{1+\exp(-a)}$$

For sigmoid as well as tanh, gradients saturate (become close to zero as the function tends to its extreme values)



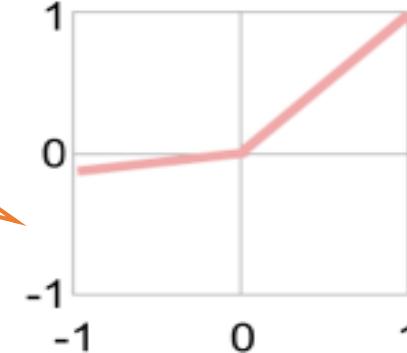
$$\text{tanh (tan hyperbolic): } h = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = 2\sigma(2a) - 1$$

Preferred more than sigmoid. Helps keep the mean of the next layer's inputs close to zero (with sigmoid, it is close to 0.5)



$$\text{ReLU (Rectified Linear Unit): } h = \max(0, a)$$

Helps fix the dead neuron problem of ReLU when a is a negative number



$$\text{Leaky ReLU: } h = \max(\beta a, a) \text{ where } \beta \text{ is a small positive number}$$

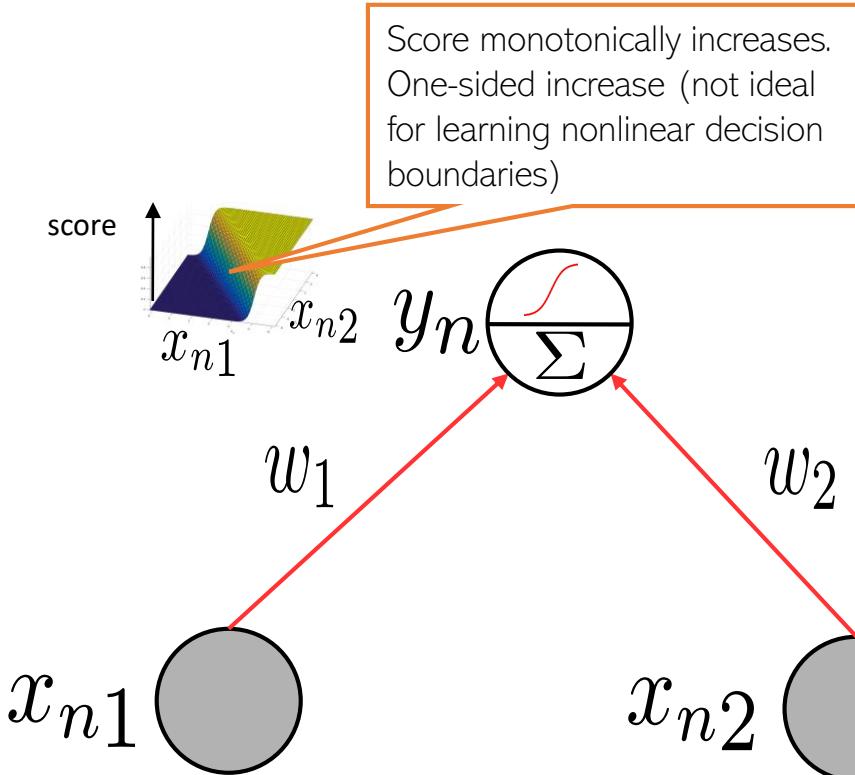
ReLU and Leaky ReLU are among the most popular ones

Without nonlinear activation, a deep neural network is equivalent to a linear model no matter how many layers we use

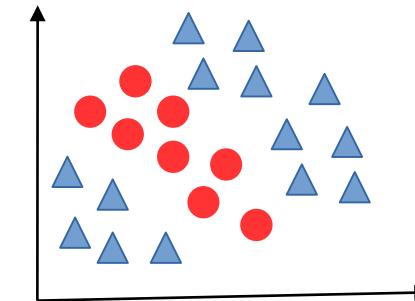


MLP Can Learn Nonlin. Fn: A Brief Justification

- An MLP can be seen as a composition of multiple linear models combined nonlinearly



Obtained by composing the two one-sided increasing score functions (using $v_1 = 1$, and $v_2 = -1$ to “flip” the second one before adding). This can now learn nonlinear decision boundary

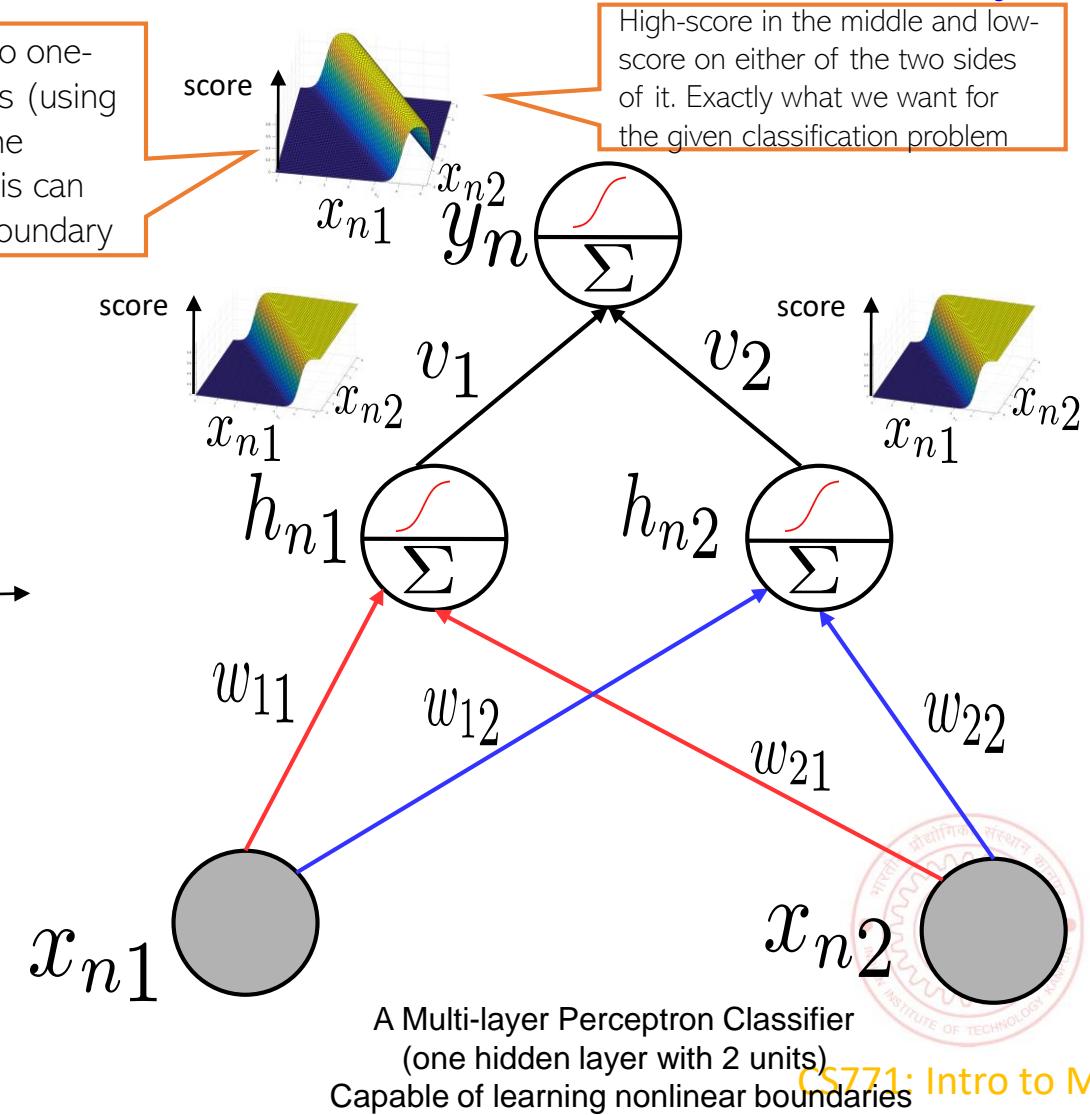


A nonlinear classification problem



Standard Single “Perceptron” Classifier (no hidden units)

A single hidden layer MLP with sufficiently large number of hidden units can approximate any function (Hornik, 1991)



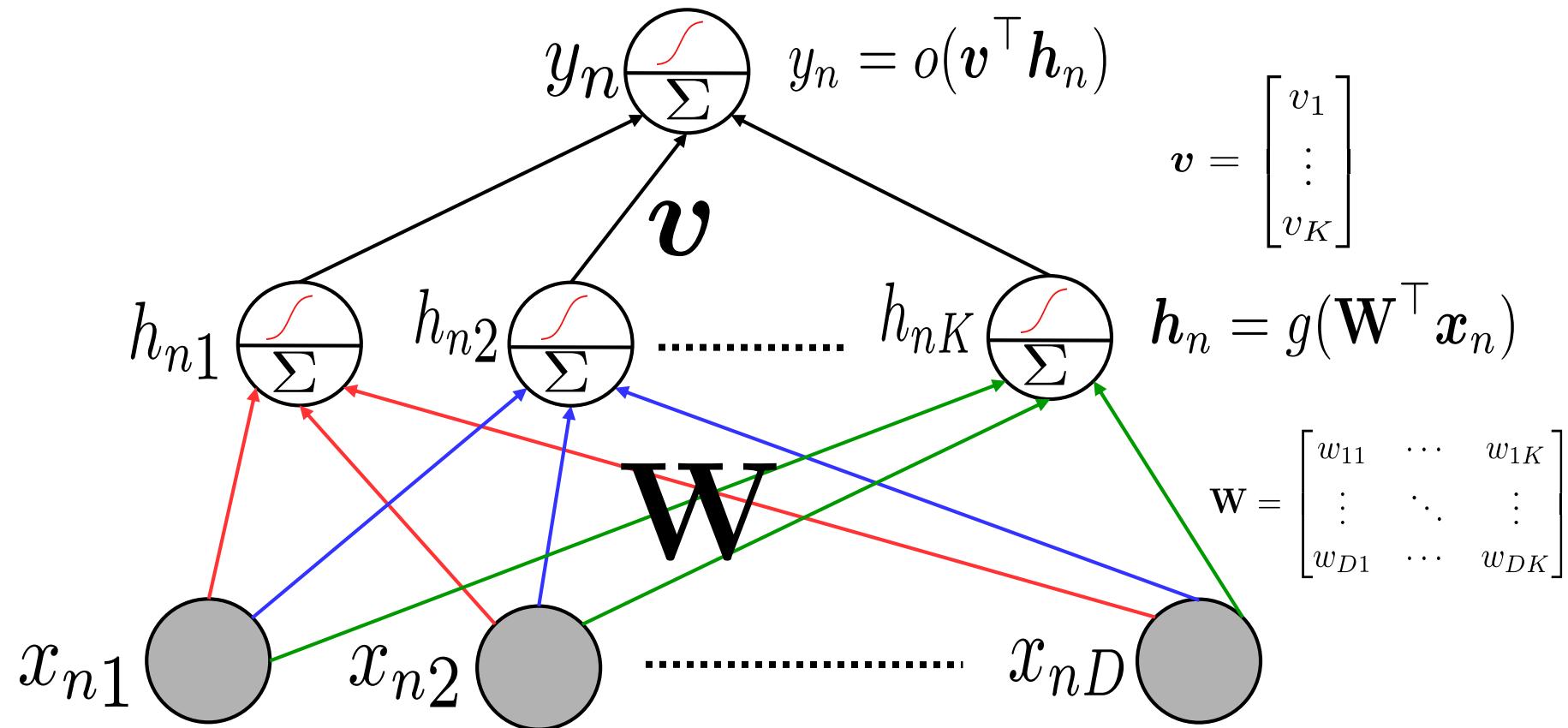
A Multi-layer Perceptron Classifier
(one hidden layer with 2 units)
Capable of learning nonlinear boundaries

Examples of some basic NN/MLP architectures



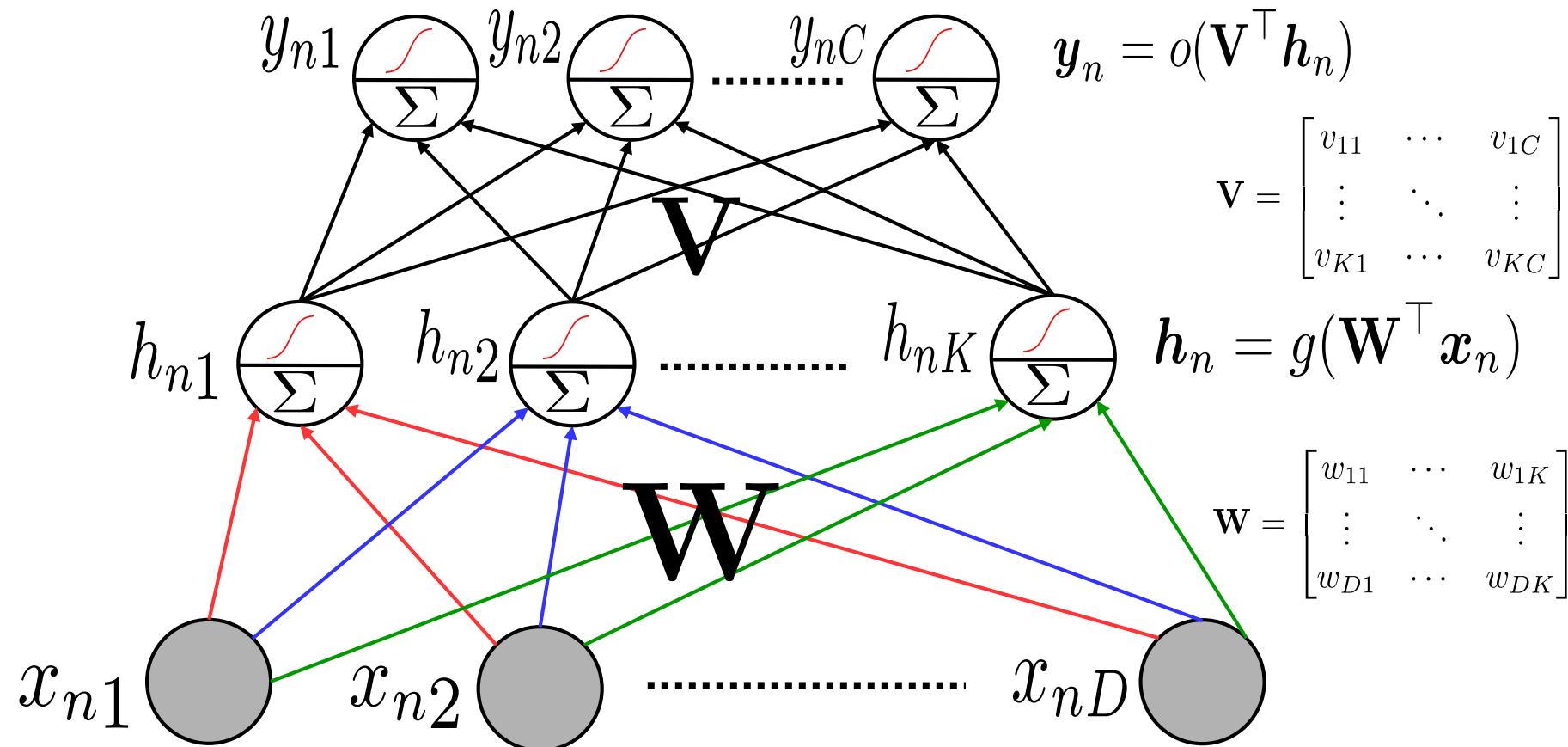
Single Hidden Layer and Single Outputs

- One hidden layer with K nodes and a single output (e.g., scalar-valued regression or binary classification)



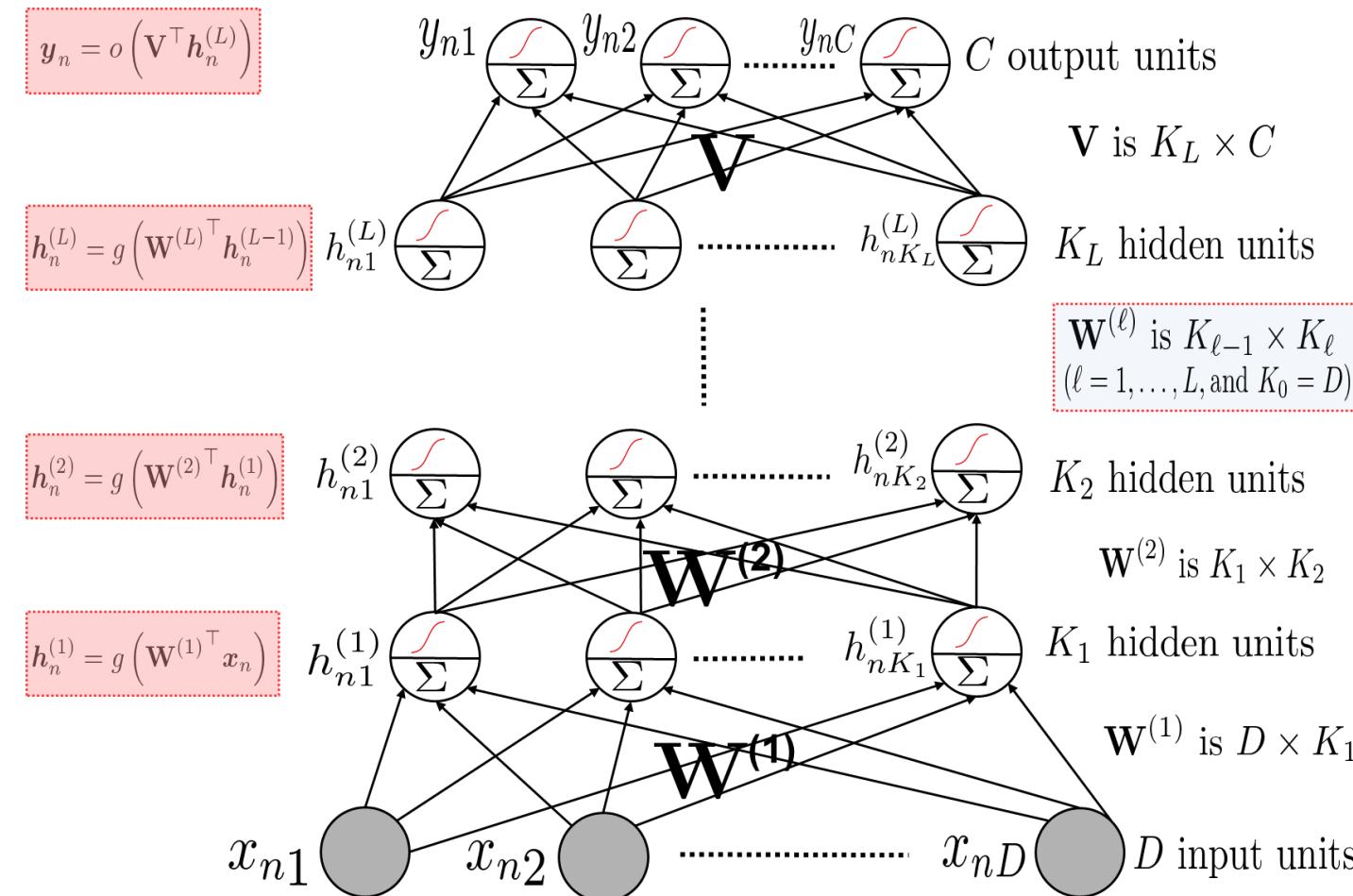
Single Hidden Layer and Multiple Outputs

- One hidden layer with K nodes and a vector of C output (e.g., vector-valued regression or multi-class classification or multi-label classification)



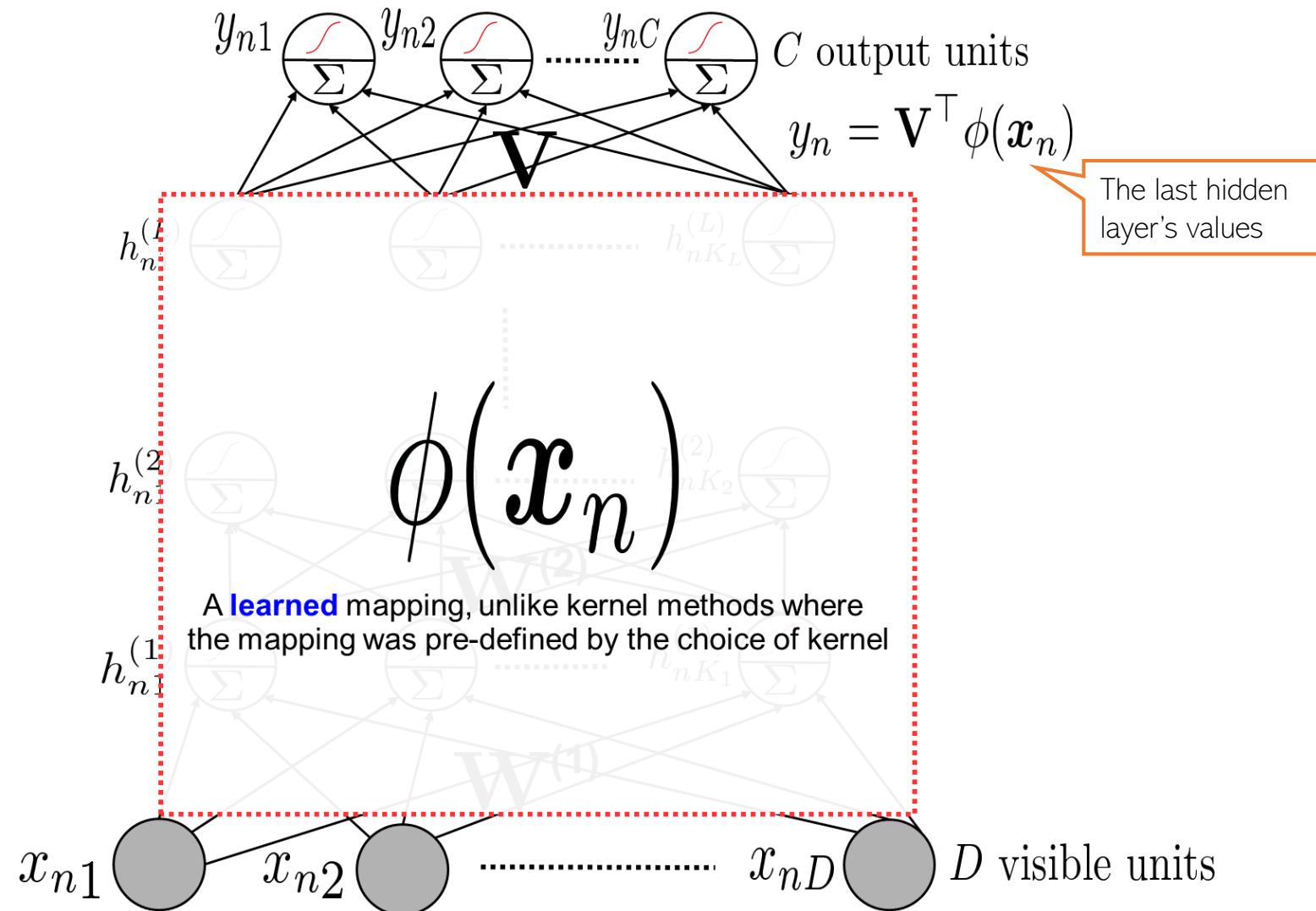
Multiple Hidden Layers (One/Multiple Outputs)

- Most general case: Multiple hidden layers with (with same or different number of hidden nodes in each) and a scalar or vector-valued output



Neural Nets are Feature Learners

- Hidden layers can be seen as learning a feature rep. $\phi(\mathbf{x}_n)$ for each input \mathbf{x}_n



Kernel Methods vs Neural Nets

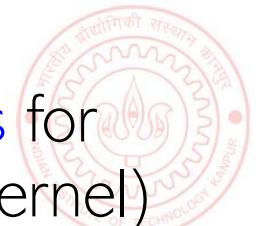
- Recall the prediction rule for a kernel method (e.g., kernel SVM)

$$y = \sum_{n=1}^N \alpha_n k(\mathbf{x}_n, \mathbf{x})$$

- This is analogous to a single hidden layer NN with fixed/pre-defined hidden nodes $\{k(\mathbf{x}_n, \mathbf{x})\}_{n=1}^N$ and output weights $\{\alpha_n\}_{n=1}^N$
- The prediction rule for a deep neural network

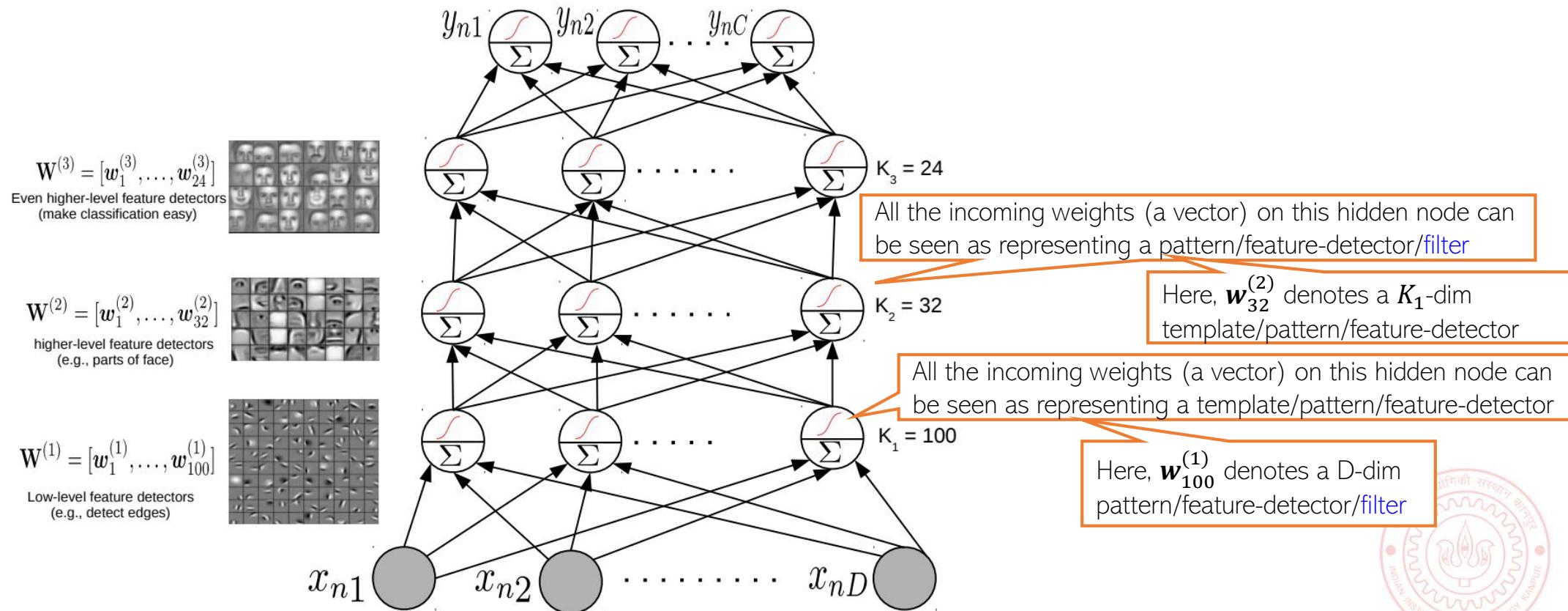
$$y = \sum_{k=1}^K v_k h_k$$

- Here, the h_k 's are learned from data (possibly after multiple layers of nonlinear transformations)
- Both kernel methods and deep NNs be seen as using **nonlinear basis functions** for making predictions. Kernel methods use **fixed basis functions** (defined by the kernel) whereas NN learns the basis functions adaptively from data



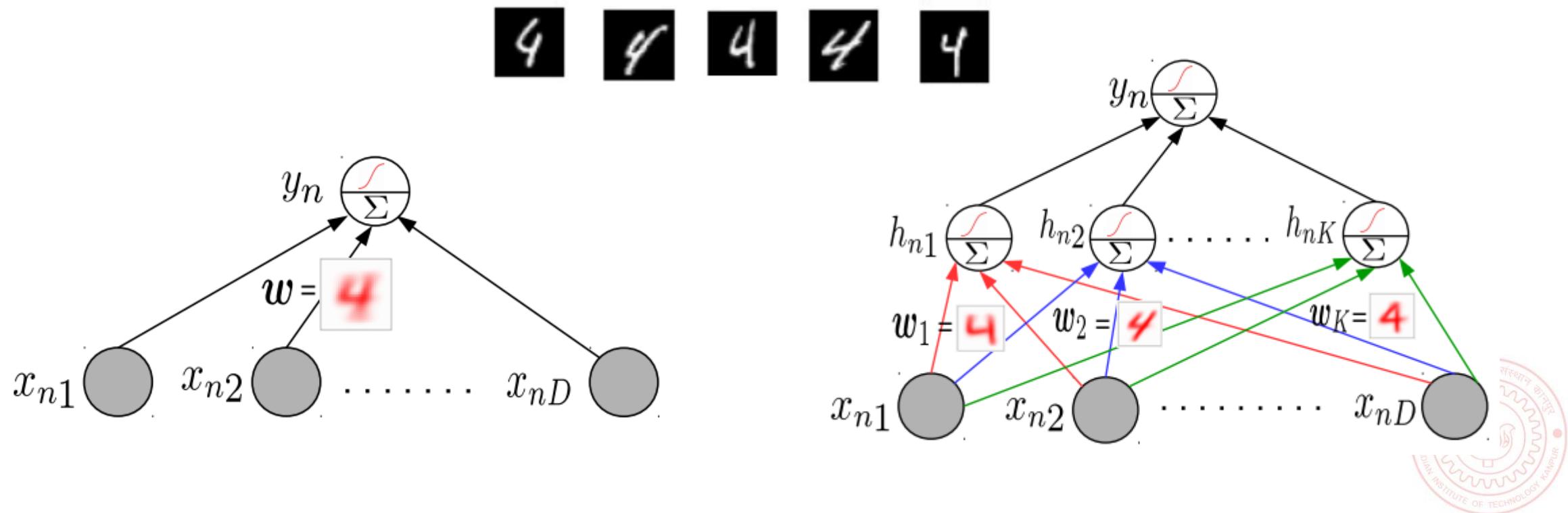
Feature Learned by a Neural Network

- Node values in each hidden layer tell us how much a “learned” feature is active in x_n
- Hidden layer weights are like pattern/feature-detector/filter



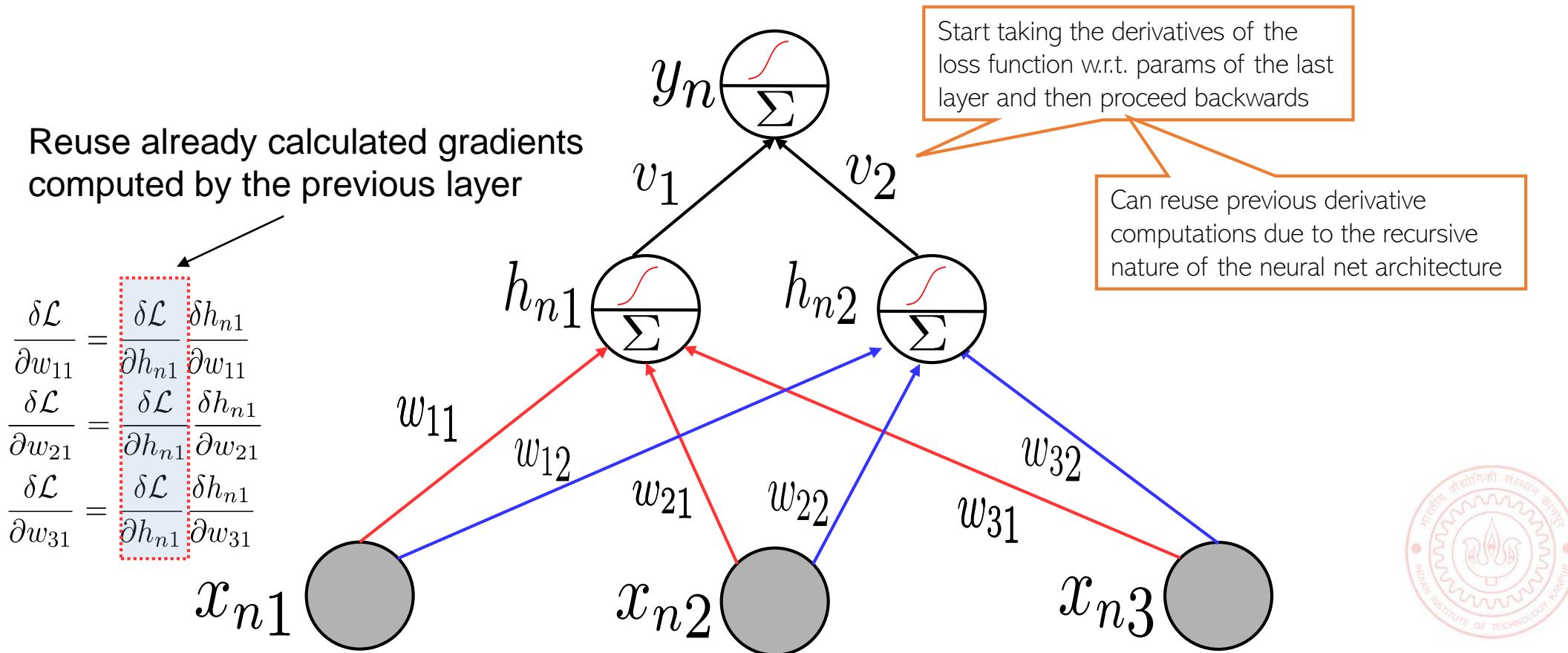
Why Neural Networks Work Better: Another View¹⁵

- Linear models tend to only learn the “average” pattern
- Deep models can learn multiple patterns (each hidden node can learn one pattern)
 - Thus deep models can learn to capture more subtle variations that a simpler linear model



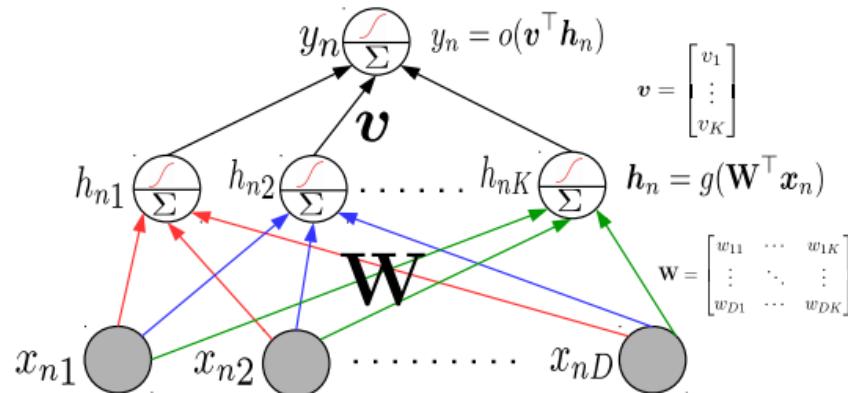
Backpropagation

- Backpropagation = Gradient descent using chain rule of derivatives
- Chain rule of derivatives: Example, if $y = f_1(x)$ and $x = f_2(z)$ then $\frac{\partial y}{\partial z} = \frac{\partial y}{\partial x} \frac{\partial x}{\partial z}$



Backpropagation through an example

Consider a single hidden layer MLP



Assuming regression ($o = \text{identity}$),
the loss function for this model

$$\begin{aligned}\mathcal{L} &= \frac{1}{2} \sum_{n=1}^N (y_n - \mathbf{v}^\top \mathbf{h}_n)^2 \\ &= \frac{1}{2} \sum_{n=1}^N \left(y_n - \sum_{k=1}^K v_k h_{nk} \right)^2 \\ &= \frac{1}{2} \sum_{n=1}^N \left(y_n - \sum_{k=1}^K v_k g(\mathbf{w}_k^\top \mathbf{x}_n) \right)^2\end{aligned}$$

- To use gradient methods for \mathbf{W}, \mathbf{v} , we need gradients.
- Gradient of \mathcal{L} w.r.t. \mathbf{v} is straightforward

$$\frac{\partial \mathcal{L}}{\partial v_k} = - \sum_{n=1}^N \left(y_n - \sum_{k=1}^K v_k g(\mathbf{w}_k^\top \mathbf{x}_n) \right) h_{nk} = \sum_{n=1}^N \mathbf{e}_n h_{nk}$$

- Gradient of \mathcal{L} w.r.t. \mathbf{W} requires chain rule

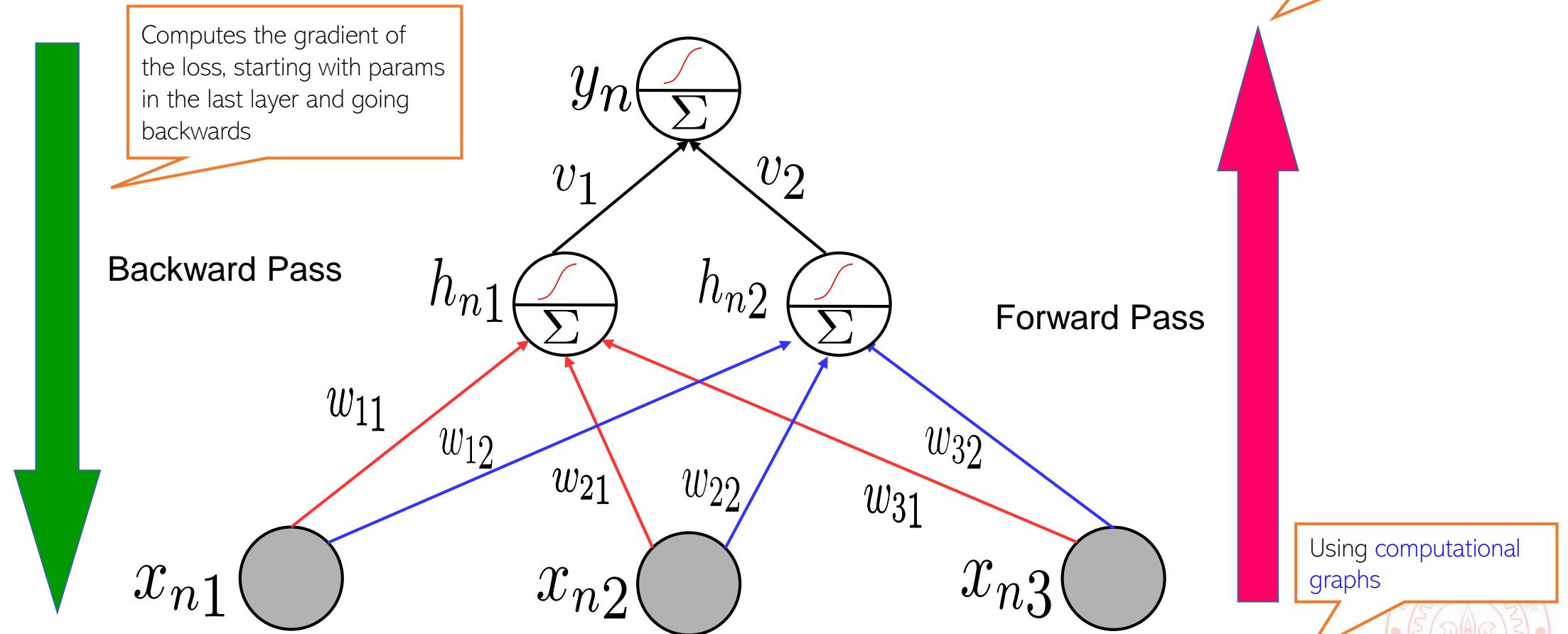
$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{dk}} &= \sum_{n=1}^N \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}} \\ \frac{\partial \mathcal{L}}{\partial h_{nk}} &= -(y_n - \sum_{k=1}^K v_k g(\mathbf{w}_k^\top \mathbf{x}_n)) v_k = -\mathbf{e}_n v_k \\ \frac{\partial h_{nk}}{\partial w_{dk}} &= g'(\mathbf{w}_k^\top \mathbf{x}_n) x_{nd} \quad (\text{note: } h_{nk} = g(\mathbf{w}_k^\top \mathbf{x}_n))\end{aligned}$$

- Forward prop computes errors \mathbf{e}_n using current \mathbf{W}, \mathbf{v} .
Backprop updates NN params \mathbf{W}, \mathbf{v} using grad methods
- Backprop caches many of the calculations for reuse



Backpropagation

- Backprop iterates between a forward pass and a backward pass

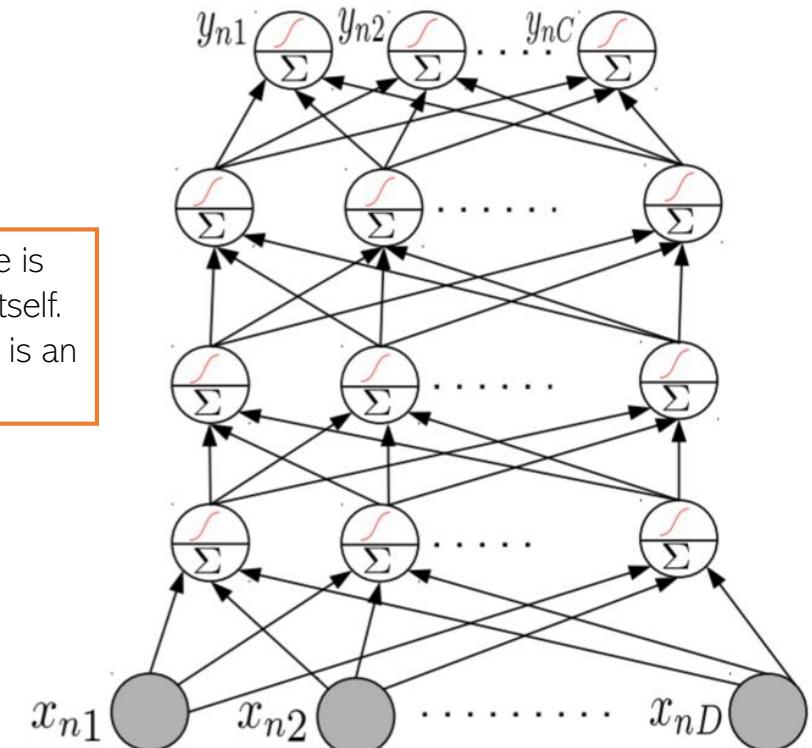


- Software frameworks such as Tensorflow and PyTorch support this already so you don't need to implement it by hand (so no worries of computing derivatives etc)

Neural Nets: Some Aspects

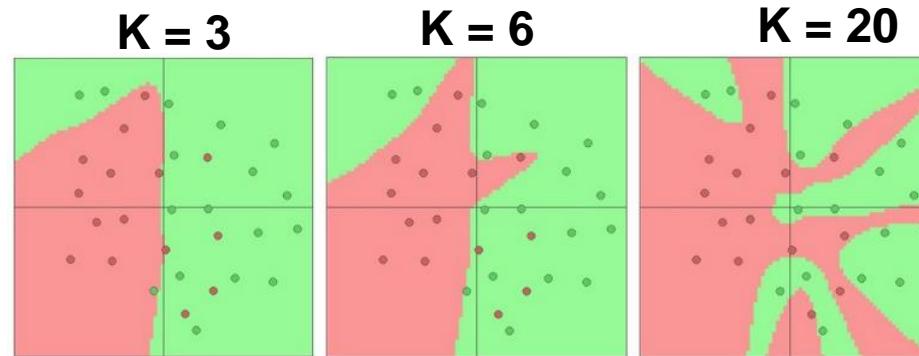
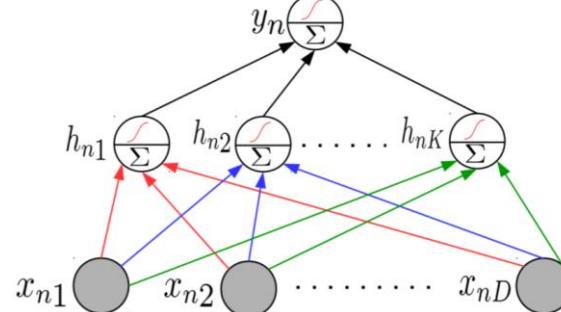
- Much of the magic lies in the hidden layers
- Hidden layers learn and detect good features
- Need to consider a few aspects
 - Number of hidden layers, number of units in each hidden layer
 - Why bother about many hidden layers and not use a single very wide hidden layer (recall Hornik's universal function approximator theorem)?
 - Complex networks (several, very wide hidden layers) or simpler networks (few, moderately wide hidden layers)?
 - Aren't deep neural network prone to overfitting (since they contain a huge number of parameters)?

Choosing the right NN architecture is important and a research area in itself.
[Neural Architecture Search \(NAS\)](#) is an automated technique to do this



Representational Power of Neural Nets

- Consider a single hidden layer neural net with K hidden nodes



- Recall that each hidden unit “adds” a function to the overall function
- Increasing K (number of hidden units) will result in a more complex function
- Very large K seems to overfit (see above fig). Should we instead prefer small K ?
- No! It is better to use large K and regularize well. Reason/justification:
 - Simple NN with small K will have a few local optima, some of which may be bad
 - Complex NN with large K will have many local optimal, all equally good (theoretical results on this)
- We can also use multiple hidden layers (each sufficiently large) and regularize well



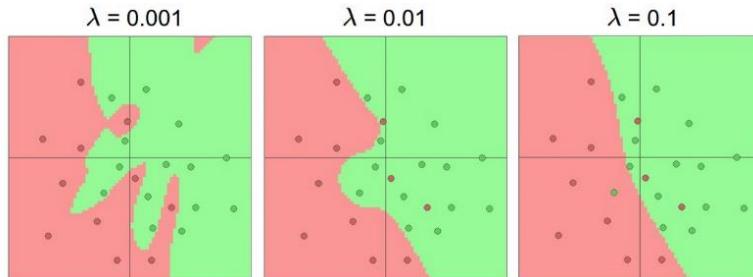
Preventing Overfitting in Neural Nets

Various other tricks, such as weight sharing across different hidden units of the same layer (used in [convolutional neural nets](#) or CNN)

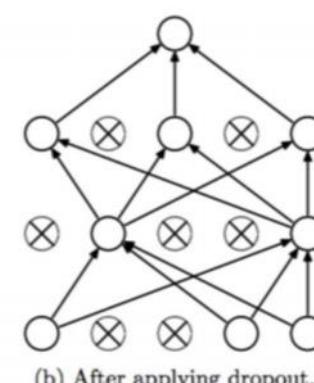
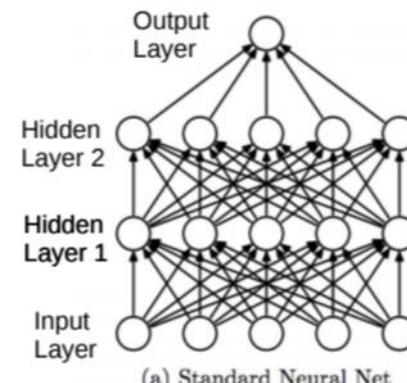
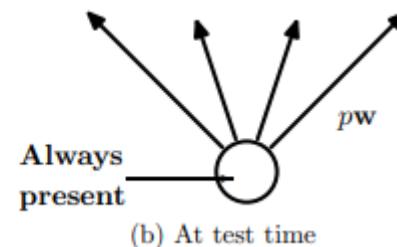
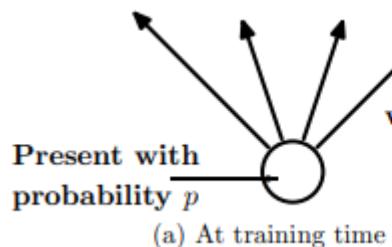


- Neural nets can overfit. Many ways to avoid overfitting, such as
 - Standard regularization on the weights, such as ℓ_2 , ℓ_1 , etc (ℓ_2 reg. is also called [weight decay](#))

Single Hidden Layer NN with K = 20 hidden units and L2 regularization

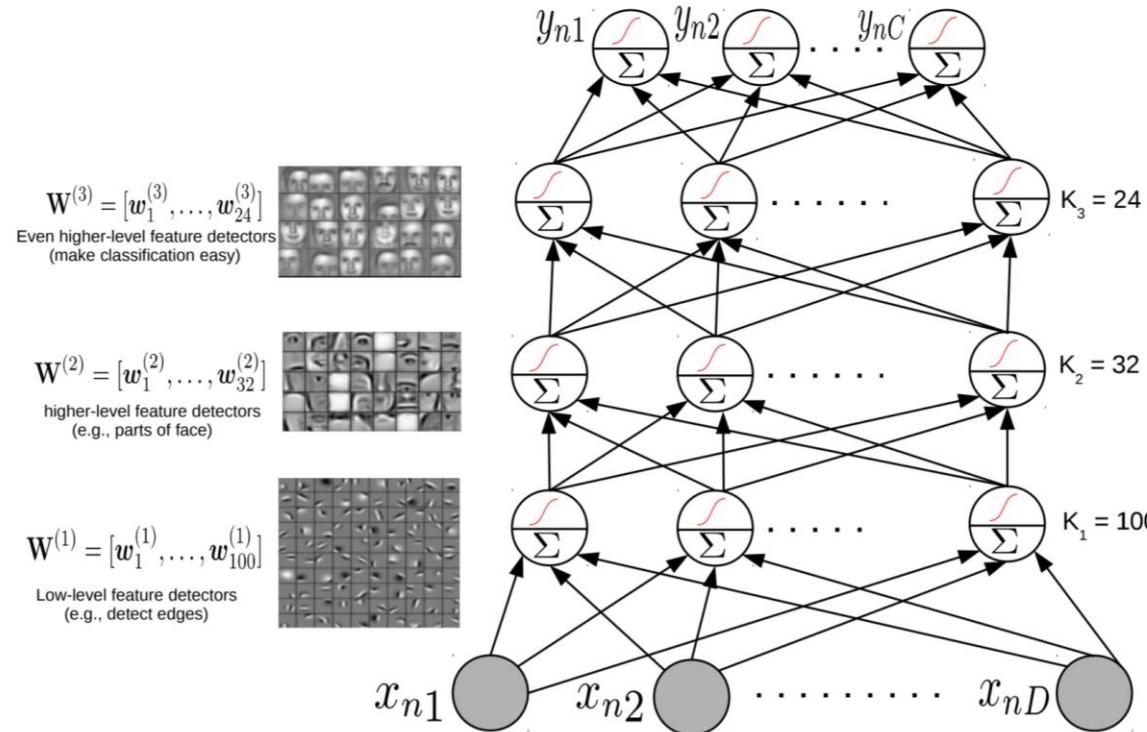


- Early stopping (traditionally used): Stop when validation error starts increasing
- Dropout: Randomly remove units (with some probability $p \in (0,1)$) during training



Wide or Deep?

- While very wide single hidden layer can approx. any function, often we prefer many, less wide, hidden layers

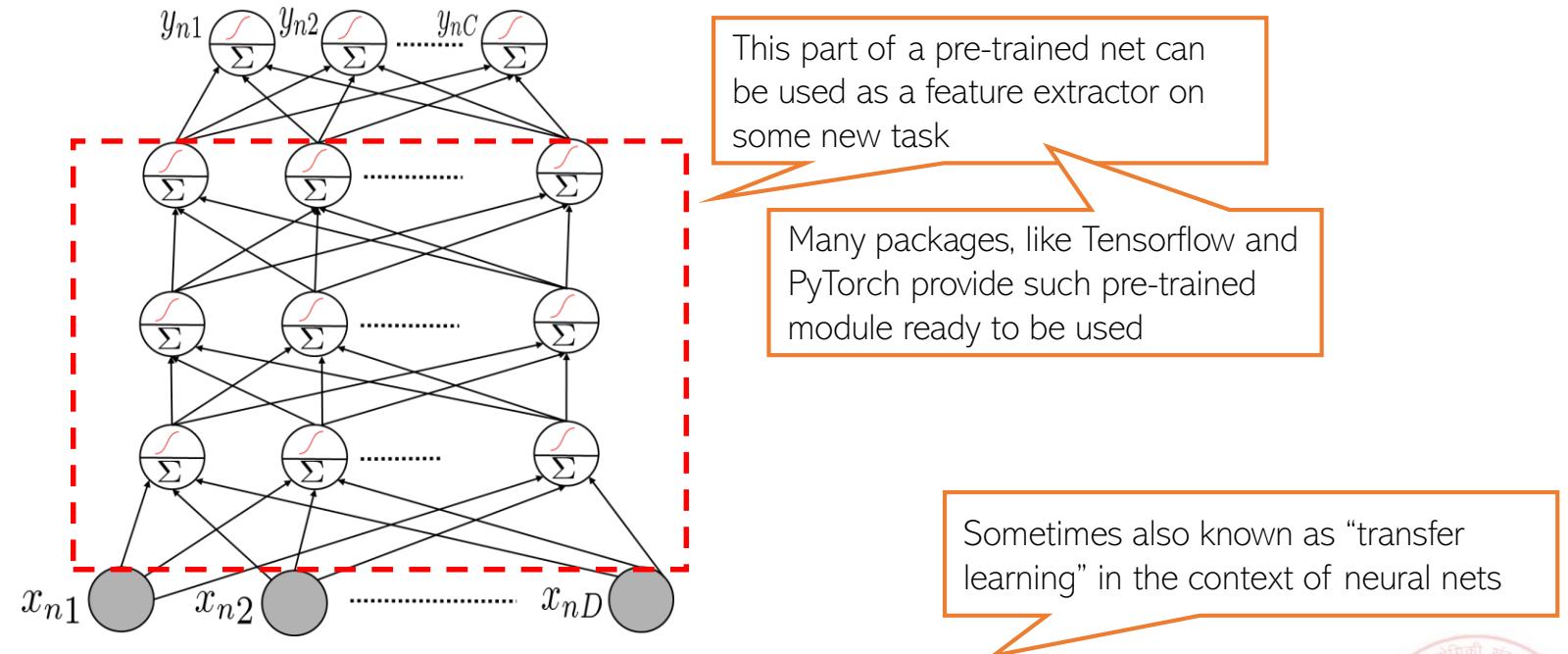


- Higher layers help learn more directly useful/interpretable features (also useful for compressing data using a small number of features)



Using a Pre-trained Network

- A deep NN already trained in some “generic” data can be useful for other tasks, e.g.,
 - **Feature extraction:** Use a pre-trained net, remove the output layer, and use the rest of the network as a feature extractor for a related dataset

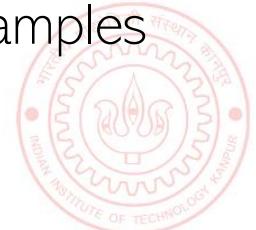


- **Fine-tuning:** Use a pre-trained net, use its weights as initialization to train a deep net for a new but related task (useful when we don't have much training data for the new task)



Deep Neural Nets: Some Comments

- Highly effective in learning good feature rep. from data in an “end-to-end” manner
- The objective functions of these models are **highly non-convex**
 - But fast and robust non-convex opt algos exist for learning such deep networks
- Training these models is computationally very expensive
 - But GPUs can help to speed up many of the computations
- Also useful for unsupervised learning problems (will see some examples)
 - Autoencoders for dimensionality reduction
 - Deep generative models for generating data and (unsupervisedly) learning features – examples include generative adversarial networks (GAN) and variational auto-encoders (VAE)



Coming up next

- Convolutional neural nets
- Neural nets for sequential data
- Neural networks for unsupervised learning and generation



Maths Refresher

Piyush Rai

Introduction to Machine Learning (CS771A)

Basics of Linear Algebra

Vectors and Matrices

- Vectors (column vectors and row vectors) and their transposes

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \quad \mathbf{a}^\top = [a_1 \ a_2 \ a_3], \quad \mathbf{b} = [b_1 \ b_2 \ b_3], \quad \mathbf{b}^\top = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

- We will assume vectors to be column vectors (unless specified otherwise)
- Vector with all 0s except a single 1 is called **elementary vector** (or “one-hot” vector in ML)
- Matrix and its transpose (shown for 3×3 matrices)

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad \mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$$

- For a symmetric matrix (must be square) $\mathbf{A} = \mathbf{A}^\top$
- Diagonal and identity matrices have nonzeros only along the diagonals
- Should know the basic rules of vector addition, matrix addition, etc (won’t list here)

Inner Product

- Inner product (or dot product) of two vectors $\mathbf{a} \in \mathbb{R}^D$ and $\mathbf{b} \in \mathbb{R}^D$ is a scalar

$$c = \mathbf{a}^\top \mathbf{b} = \sum_{d=1}^D a_d b_d$$

- Inner product is a measure of similarity of two vectors
- Inner product is zero if \mathbf{a} and \mathbf{b} are orthogonal to each other
- Inner product of two vector of unit length is the same as cosine similarity
- A more general form of inner product: $c = \mathbf{a}^\top \mathbf{M} \mathbf{b}$ (here \mathbf{M} is $D \times D$)
 - \mathbf{M} can be diagonal or full matrix
 - For identity \mathbf{M} , it becomes the standard inner product
- Euclidean distance between two vectors can be also written in terms of an inner product

$$d(\mathbf{a}, \mathbf{b}) = \sqrt{(\mathbf{a} - \mathbf{b})^\top (\mathbf{a} - \mathbf{b})} = \sqrt{\mathbf{a}^\top \mathbf{a} + \mathbf{b}^\top \mathbf{b} - 2\mathbf{a}^\top \mathbf{b}}$$

Orthogonal/Orthonormal Vectors and Matrices

- A set of vectors $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_N$ is called **orthogonal** if

$$\mathbf{a}_i^\top \mathbf{a}_j = 0 \quad \forall i \neq j$$

- Moreover, a set of orthogonal vectors $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_N$ is called **orthonormal** if

$$\mathbf{a}_i^\top \mathbf{a}_i = 1 \quad \forall i$$

- A **matrix** with orthonormal columns is called **orthogonal**
- For a square orthogonal matrix A , we have $AA^\top = A^\top A = I$

Matrix-Vector/Matrix-Matrix Product as Inner Product

- Important to be conversant with these. Some basic operations worth keeping in mind

- We can 'post-multiply' a matrix by a column vector:

$$Ax = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a_1^T x_1 \\ a_2^T x_2 \\ a_3^T x_3 \end{bmatrix}$$

- We can 'pre-multiply' a matrix by a row vector:

$$x^T A = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} x^T a_1 & x^T a_2 & x^T a_3 \end{bmatrix}$$

- In general, we can multiply matrices A and B when the number of columns in A matches the number of rows in B:

$$AB = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_1^T b_1 & a_1^T b_2 & a_1^T b_3 \\ a_2^T b_1 & a_2^T b_2 & a_2^T b_3 \\ a_3^T b_1 & a_3^T b_2 & a_3^T b_3 \end{bmatrix}$$

- We routinely encounter such operations in many ML problems

Outer Product

- Outer product of two vectors $\mathbf{a} \in \mathbb{R}^D$ and $\mathbf{b} \in \mathbb{R}^D$ is a matrix. For 3-dim vectors, we'll have

$$\mathbf{C} = \mathbf{ab}^\top = \begin{bmatrix} a_1 b_1 & a_1 b_2 & a_1 b_3 \\ a_2 b_1 & a_2 b_2 & a_2 b_3 \\ a_3 b_1 & a_3 b_2 & a_3 b_3 \end{bmatrix} \quad (\text{note: } \mathbf{C} \text{ is a rank-1 matrix})$$

- Matrix rank: Linearly indep. number of rows/columns
- Matrix multiplications can also be written as a sum of outer products (sum of rank-1 matrices)

$$\mathbf{AB}^\top = \sum_{k=1}^K \mathbf{a}_k \mathbf{b}_k^\top$$

where \mathbf{a}_k and \mathbf{b}_k denote the k -th column of A (size: $D \times K$) and B (size: $D \times K$), respectively,

Linear Combination of Vectors as a Matrix-Vector Product

- Linear combination of a set of $D \times 1$ vectors $\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_N$ is another vector of the same size

$$\mathbf{c} = \alpha_1 \mathbf{b}_1 + \alpha_2 \mathbf{b}_2 + \dots + \alpha_N \mathbf{b}_N$$

- The α_n 's are scalar-valued combination weights
- The above can also be compactly written in the [matrix-vector product](#) form $\mathbf{c} = \mathbf{B}\boldsymbol{\alpha}$

The diagram shows a vertical vector \mathbf{c} on the left, followed by an equals sign. To its right is a large gray square labeled \mathbf{B} . To the right of \mathbf{B} is a multiplication symbol $*$. To the right of $*$ is a vertical vector labeled $\boldsymbol{\alpha}$.

where $\mathbf{B} = [\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_N]$ is a $D \times N$ matrix, and $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_N]^\top$ is an $N \times 1$ column vector

- Note that \mathbf{c} can be also seen as a [linear transformation](#) of $\boldsymbol{\alpha}$ using \mathbf{B}
- Such matrix-vector products are very common in ML problems (especially in [linear models](#))

Vector and Matrix Norms

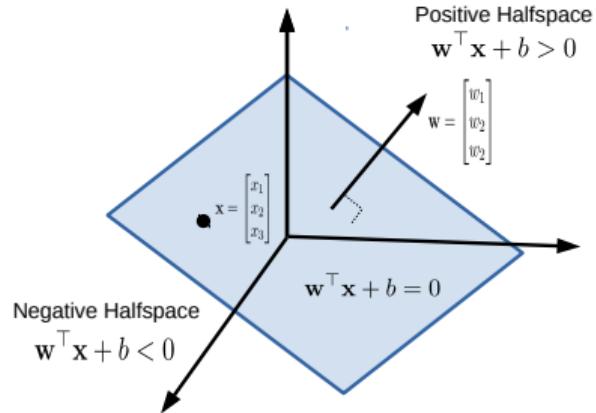
- Roughly speaking, for a vector \mathbf{x} , the norm is its “length”
- Some common norms: ℓ_2 norm (Euclidean norm), ℓ_1 norm, ℓ_∞ norm, ℓ_p norm ($p \geq 1$)

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{n=1}^N x_n^2}, \quad \|\mathbf{x}\|_1 = \sum_{n=1}^N |x_n|, \quad \|\mathbf{x}\|_\infty = \max_{1 \leq n \leq N} |x_n|, \quad \|\mathbf{x}\|_p = \left(\sum_{n=1}^N |x_n|^p \right)^{1/p}$$

- Note: The square of ℓ_2 norm is the inner product of the vector with itself $\|\mathbf{x}\|_2^2 = \mathbf{x}^\top \mathbf{x}$
- Note: $\|\mathbf{x}\|_p$ for $p < 1$ technically not a norm (doesn't satisfy all the formal properties of a norm)
 - Nevertheless it is often used in some ML problems (has some interesting properties)
- Norms for a matrix A (say of size $N \times M$) can also be defined, e.g.,
 - Frobenius norm: $\|A\|_F = \sqrt{\sum_{i=1}^N \sum_{j=1}^M A_{ij}^2} = \sqrt{\text{trace}(A^\top A)}$
 - Many matrix norms can be written in terms of the singular values of A

Hyperplanes

- An important concept in ML, especially for understanding classification problems
- Divides a vector space into two halves (positive and negative halfspaces)



- Assuming 3-dim space, it can be defined by a vector $w = [w_1, w_2, w_3]$ and scalar b
- w is the vector pointing outward to the hyperplane
- b is the real-valued “bias” if the hyperplane doesn’t pass through the origin

Some other things you should know about..

- Eigenvalues, rank, etc. for matrices
- Trace of matrix
- Determinant of matrix (and relation to eigenvalues etc)
- Inverse of matrices
- Positive definite and positive semi-definite matrices (non-negative eigenvalues)
- “Matrix Cookbook” (will provide link) is a nice source of many properties of matrices

Multivariate Calculus and Optimization

Multivariate Calculus and Optimization

- Most of ML problems boil down to solving an optimization problem
- We will usually have to optimize a function $f : \mathbb{R}^D \rightarrow \mathbb{R}$ w.r.t some variable $\mathbf{w} \in \mathbb{R}^D$
- Gradient of f w.r.t. \mathbf{w} denotes the direction of steepest change at \mathbf{w} , and is defined as

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \vdots \\ \frac{\partial f}{\partial w_D} \end{bmatrix} \quad \text{where } [\nabla f]_i = \frac{\partial f}{\partial w_i}$$

- For multivariate functions $f : \mathbb{R}^D \rightarrow \mathbb{R}^M$, we can likewise define the **Jacobian** matrix

$$\mathbf{J}_f = \begin{bmatrix} \frac{\partial f_1}{\partial w_1} & \dots & \frac{\partial f_1}{\partial w_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial w_1} & \dots & \frac{\partial f_M}{\partial w_D} \end{bmatrix} \quad \text{where } [\mathbf{J}_f]_{ij} = \frac{\partial f_i}{\partial w_j}$$

- Can also define second derivatives (called **Hessian**): derivative of gradient/Jacobian

Taking Derivatives

- Optimization in ML problems requires being able to take derivatives (i.e., doing Calculus)
- What makes it tricky is that usually we are no longer doing optimization w.r.t. a single scalar variable but w.r.t. vectors or sometimes even matrices (thus need [vector/matrix calculus](#))
- For some functions, derivatives are easy (can even be done by hand)
- Perhaps the most common, easy ones include derivatives of linear and quadratic functions

$$\nabla_{\mathbf{w}}[\mathbf{x}^T \mathbf{w}] = \mathbf{x}$$

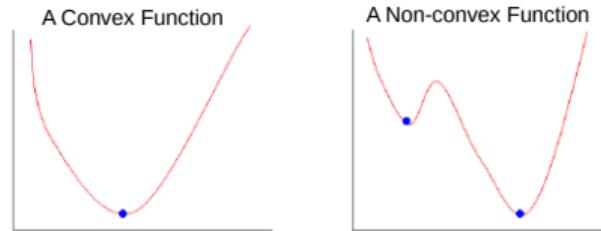
$$\nabla_{\mathbf{w}}[\mathbf{w}^T \mathbf{X} \mathbf{w}] = (\mathbf{X} + \mathbf{X}^T) \mathbf{w} \quad (\text{where } \mathbf{X} \text{ is } D \times D \text{ matrix})$$

$$\nabla_{\mathbf{w}}[\mathbf{w}^T \mathbf{X} \mathbf{w}] = 2\mathbf{X}\mathbf{w} \quad (\text{if } \mathbf{X} \text{ is symmetric matrix})$$

- The “Matrix Cookbook” contains many derivative formulas (you can use that as a reference even if you don’t know how to compute derivative by hand)
- For more complicated functions, thankfully there exist tool that allow [automatic differentiation](#)
- But you should still have a good understanding of derivatives and be familiar with at least some basic results like the above (and some others from the Matrix Cookbook)

Convex Functions

- Convex functions have a unique optima

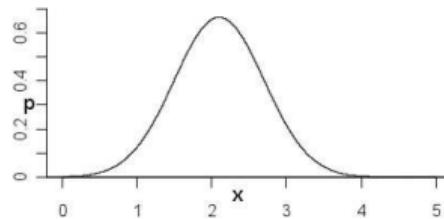
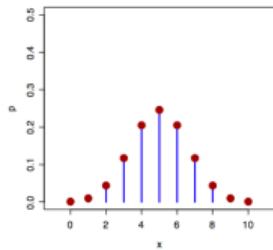


- Optimizing convex functions is usually easier than optimizing non-convex ones
- More on this when we look at optimization for ML later during the semester

Basics of Probability and Probability Distributions

Random Variables

- Informally, a random variable (r.v.) X denotes possible outcomes of an event
- Can be **discrete** (i.e., finite many possible outcomes) or **continuous**

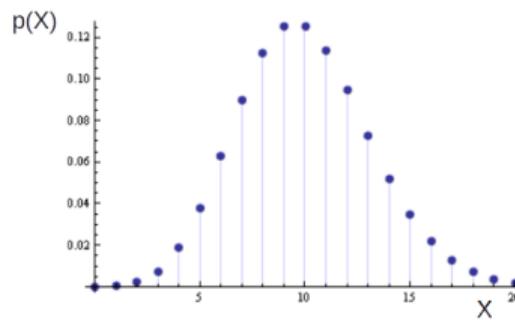


- Some examples of **discrete r.v.**
 - A random variable $X \in \{0, 1\}$ denoting outcomes of a coin-toss
 - A random variable $X \in \{1, 2, \dots, 6\}$ denoting outcome of a dice roll
- Some examples of **continuous r.v.**
 - A random variable $X \in (0, 1)$ denoting the bias of a coin
 - A random variable X denoting heights of students in CS771A
 - A random variable X denoting time to get to your hall from the department

Discrete Random Variables

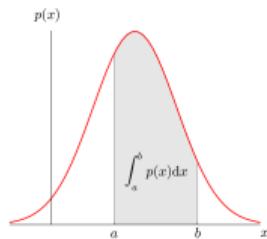
- For a discrete r.v. X , $p(x)$ denotes the probability that $p(X = x)$
- $p(x)$ is called the **probability mass function** (PMF)

$$\begin{aligned} p(x) &\geq 0 \\ p(x) &\leq 1 \\ \sum_x p(x) &= 1 \end{aligned}$$



Continuous Random Variables

- For a continuous r.v. X , a probability $p(X = x)$ is meaningless
- Instead we use $p(X = x)$ or $p(x)$ to denote the probability density at $X = x$
- For a continuous r.v. X , we can only talk about probability within an interval $X \in (x, x + \delta x)$
 - $p(x)\delta x$ is the probability that $X \in (x, x + \delta x)$ as $\delta x \rightarrow 0$



- The probability density $p(x)$ satisfies the following

$$p(x) \geq 0 \quad \text{and} \quad \int_x^{\infty} p(x)dx = 1 \quad (\text{note: for continuous r.v., } p(x) \text{ can be } > 1)$$

A word about notation..

- $p(\cdot)$ can mean different things depending on the context
 - $p(X)$ denotes the distribution (PMF/PDF) of an r.v. X
 - $p(X = x)$ or $p(x)$ denotes the **probability** or **probability density** at point x
- Actual meaning should be clear from the context (but be careful)
- Exercise the same care when $p(\cdot)$ is a specific distribution (Bernoulli, Beta, Gaussian, etc.)
- The following means **drawing a random sample** from the distribution $p(X)$

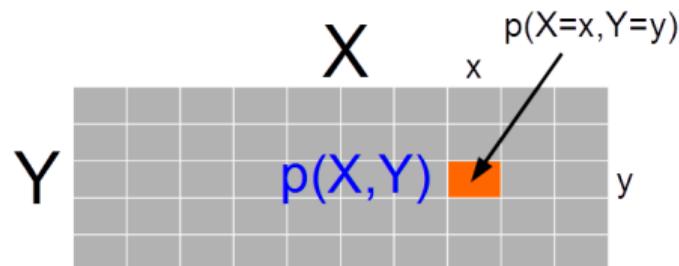
$$x \sim p(X)$$

Joint Probability Distribution

Joint probability distribution $p(X, Y)$ models probability of co-occurrence of two r.v. X, Y

For discrete r.v., the joint PMF $p(X, Y)$ is like a table (that sums to 1)

$$\sum_x \sum_y p(X = x, Y = y) = 1$$

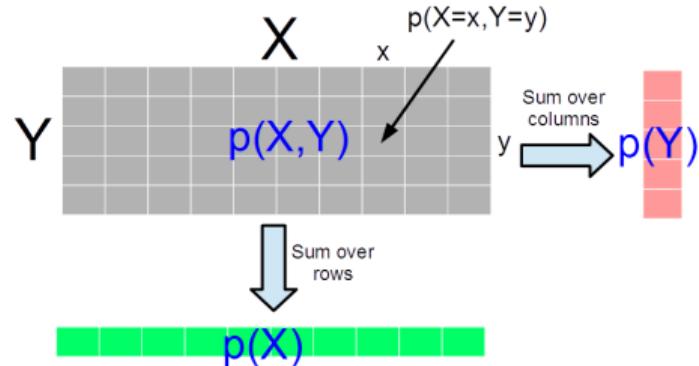


For continuous r.v., we have joint PDF $p(X, Y)$

$$\int_x \int_y p(X = x, Y = y) dx dy = 1$$

Marginal Probability Distribution

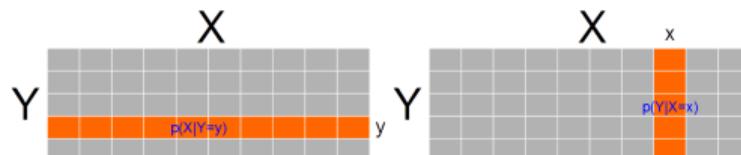
- Intuitively, the probability distribution of one r.v. regardless of the value the other r.v. takes
- For discrete r.v.'s: $p(X) = \sum_y p(X, Y = y)$, $p(Y) = \sum_x p(X = x, Y)$
- For discrete r.v. it is the sum of the PMF table along the rows/columns



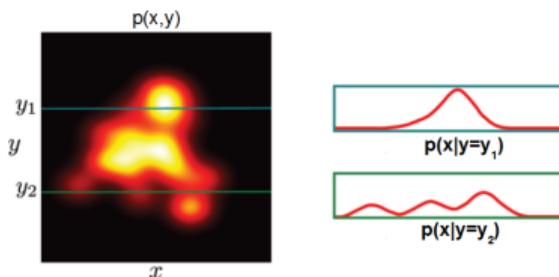
- For continuous r.v.: $p(X) = \int_y p(X, Y = y) dy$, $p(Y) = \int_x p(X = x, Y) dx$
- Note: Marginalization is also called “integrating out” (especially in Bayesian learning)

Conditional Probability Distribution

- Probability distribution of one r.v. given the value of the other r.v.
- Conditional probability $p(X|Y = y)$ or $p(Y|X = x)$: like taking a slice of $p(X, Y)$
- For a discrete distribution:



- For a continuous distribution¹:



¹Picture courtesy: Computer vision: models, learning and inference (Simon Price)

Some Basic Rules

- **Sum rule:** Gives the marginal probability distribution from joint probability distribution
 - For discrete r.v.: $p(X) = \sum_Y p(X, Y)$
 - For continuous r.v.: $p(X) = \int_Y p(X, Y) dY$
- **Product rule:** $p(X, Y) = p(Y|X)p(X) = p(X|Y)p(Y)$
- **Bayes rule:** Gives conditional probability

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}$$

- For discrete r.v.: $p(Y|X) = \frac{p(X|Y)p(Y)}{\sum_Y p(X|Y)p(Y)}$
- For continuous r.v.: $p(Y|X) = \frac{p(X|Y)p(Y)}{\int_Y p(X|Y)p(Y) dY}$
- Also remember the **chain rule**

$$p(X_1, X_2, \dots, X_N) = p(X_1)p(X_2|X_1)\dots p(X_N|X_1, \dots, X_{N-1})$$

CDF and Quantiles

- Cumulative distribution function (CDF): $F(x) = p(X \leq x)$
- $\alpha \leq 1$ quantile is defined as the x_α s.t.

$$p(X \leq x_\alpha) = \alpha$$

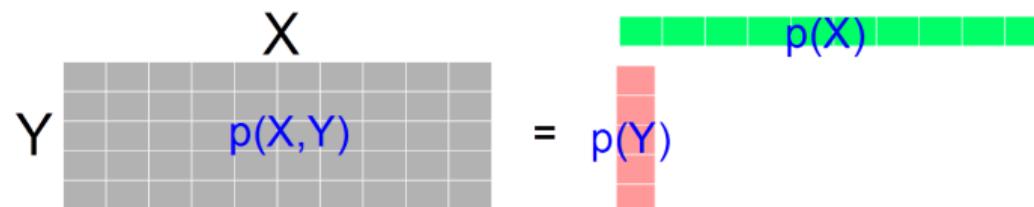
Independence

- X and Y are independent ($X \perp\!\!\!\perp Y$) when knowing one tells nothing about the other

$$p(X|Y = y) = p(X)$$

$$p(Y|X = x) = p(Y)$$

$$p(X, Y) = p(X)p(Y)$$



- $X \perp\!\!\!\perp Y$ is also called **marginal independence**
- **Conditional independence** ($X \perp\!\!\!\perp Y|Z$): independence given the value of another r.v. Z

$$p(X, Y|Z = z) = p(X|Z = z)p(Y|Z = z)$$

Expectation

- **Expectation or mean** μ of an r.v. with PMF/PDF $p(X)$

$$\mathbb{E}[X] = \sum_x xp(x) \quad (\text{for discrete distributions})$$

$$\mathbb{E}[X] = \int_x xp(x)dx \quad (\text{for continuous distributions})$$

- **Note:** The definition applies to **functions of r.v.** too (e.g., $\mathbb{E}[f(X)]$)
- **Note:** Expectations are always w.r.t. the underlying probability distribution of the random variable involved, so sometimes we'll write this explicitly as $\mathbb{E}_{p(\cdot)}[\cdot]$, unless it is clear from the context
- **Linearity of expectation**

$$\mathbb{E}[\alpha f(X) + \beta g(Y)] = \alpha\mathbb{E}[f(X)] + \beta\mathbb{E}[g(Y)]$$

(a very useful property, true even if X and Y are not independent)

- **Rule of iterated/total expectation**

$$\mathbb{E}_{p(X)}[X] = \mathbb{E}_{p(Y)}[\mathbb{E}_{p(X|Y)}[X|Y]]$$

Variance and Covariance

- **Variance** σ^2 (or “spread” around mean μ) of an r.v. with PMF/PDF $p(X)$

$$\text{var}[X] = \mathbb{E}[(X - \mu)^2] = \mathbb{E}[X^2] - \mu^2$$

- **Standard deviation:** $\text{std}[X] = \sqrt{\text{var}[X]} = \sigma$
- For two scalar r.v.'s x and y , the **covariance** is defined by

$$\text{cov}[x, y] = \mathbb{E}[\{x - \mathbb{E}[x]\}\{y - \mathbb{E}[y]\}] = \mathbb{E}[xy] - \mathbb{E}[x]\mathbb{E}[y]$$

- For **vector** r.v. \mathbf{x} and \mathbf{y} , the **covariance matrix** is defined as

$$\text{cov}[\mathbf{x}, \mathbf{y}] = \mathbb{E}[\{\mathbf{x} - \mathbb{E}[\mathbf{x}]\}\{\mathbf{y}^\top - \mathbb{E}[\mathbf{y}^\top]\}] = \mathbb{E}[\mathbf{x}\mathbf{y}^\top] - \mathbb{E}[\mathbf{x}]\mathbb{E}[\mathbf{y}^\top]$$

- Cov. of components of a vector r.v. \mathbf{x} : $\text{cov}[\mathbf{x}] = \text{cov}[\mathbf{x}, \mathbf{x}]$
- **Note:** The definitions apply to functions of r.v. too (e.g., $\text{var}[f(X)]$)
- **Note:** Variance of sum of independent r.v.'s: $\text{var}[X + Y] = \text{var}[X] + \text{var}[Y]$

KL Divergence

- Kullback–Leibler divergence between two probability distributions $p(X)$ and $q(X)$

$$KL(p||q) = \int p(X) \log \frac{p(X)}{q(X)} dX = - \int p(X) \log \frac{q(X)}{p(X)} dX \quad (\text{for continuous distributions})$$

$$KL(p||q) = \sum_{k=1}^K p(X=k) \log \frac{p(X=k)}{q(X=k)} \quad (\text{for discrete distributions})$$

- It is non-negative, i.e., $KL(p||q) \geq 0$, and zero if and only if $p(X)$ and $q(X)$ are the same
- For some distributions, e.g., Gaussians, KL divergence has a closed form expression
- KL divergence is not symmetric, i.e., $KL(p||q) \neq KL(q||p)$

Entropy

- Entropy of a continuous/discrete distribution $p(X)$

$$\begin{aligned} H(p) &= - \int p(X) \log p(X) dX \\ H(p) &= - \sum_{k=1}^K p(X = k) \log p(X = k) \end{aligned}$$

- In general, a peaky distribution would have a smaller entropy than a flat distribution
- Note that the KL divergence can be written in terms of expectation and entropy terms

$$KL(p||q) = \mathbb{E}_{p(X)}[-\log q(X)] - H(p)$$

- Some other definition to keep in mind: conditional entropy, joint entropy, mutual information, etc.

Transformation of Random Variables

Suppose $\mathbf{y} = f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$ be a linear function of an r.v. \mathbf{x}

Suppose $\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu}$ and $\text{cov}[\mathbf{x}] = \boldsymbol{\Sigma}$

- Expectation of \mathbf{y}

$$\mathbb{E}[\mathbf{y}] = \mathbb{E}[\mathbf{A}\mathbf{x} + \mathbf{b}] = \mathbf{A}\boldsymbol{\mu} + \mathbf{b}$$

- Covariance of \mathbf{y}

$$\text{cov}[\mathbf{y}] = \text{cov}[\mathbf{A}\mathbf{x} + \mathbf{b}] = \mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^T$$

Likewise if $y = f(\mathbf{x}) = \mathbf{a}^T \mathbf{x} + b$ is a scalar-valued linear function of an r.v. \mathbf{x} :

- $\mathbb{E}[y] = \mathbb{E}[\mathbf{a}^T \mathbf{x} + b] = \mathbf{a}^T \boldsymbol{\mu} + b$
- $\text{var}[y] = \text{var}[\mathbf{a}^T \mathbf{x} + b] = \mathbf{a}^T \boldsymbol{\Sigma} \mathbf{a}$

Another very useful property worth remembering

Common Probability Distributions

Important: We will use these extensively to model **data** as well as **parameters**

Some **discrete distributions** and what they can model:

- **Bernoulli:** Binary numbers, e.g., outcome (head/tail, 0/1) of a coin toss
- **Binomial:** Bounded non-negative integers, e.g., # of heads in n coin tosses
- **Multinomial:** One of $K (>2)$ possibilities, e.g., outcome of a dice roll
- **Poisson:** Non-negative integers, e.g., # of words in a document
- .. and many others

Some **continuous distributions** and what they can model:

- **Uniform:** numbers defined over a fixed range
- **Beta:** numbers between 0 and 1, e.g., probability of head for a biased coin
- **Gamma:** Positive unbounded real numbers
- **Dirichlet:** vectors that sum of 1 (fraction of data points in different clusters)
- **Gaussian:** real-valued numbers or real-valued vectors
- .. and many others

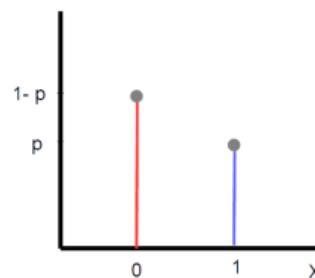
Discrete Distributions

Bernoulli Distribution

- Distribution over a binary r.v. $x \in \{0, 1\}$, like a coin-toss outcome
- Defined by a probability parameter $p \in (0, 1)$

$$P(x = 1) = p$$

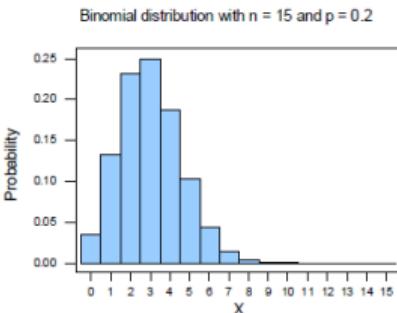
- Distribution defined as: $\text{Bernoulli}(x; p) = p^x(1 - p)^{1-x}$



- Mean: $\mathbb{E}[x] = p$
- Variance: $\text{var}[x] = p(1 - p)$

Binomial Distribution

- Distribution over number of successes m (an r.v.) in a number of trials
- Defined by two parameters: total number of trials (N) and probability of each success $p \in (0, 1)$
- Can think of Binomial as multiple independent Bernoulli trials
- Distribution defined as $\text{Binomial}(m; N, p) = \binom{N}{m} p^m (1 - p)^{N-m}$



- Mean: $\mathbb{E}[m] = Np$
- Variance: $\text{var}[m] = Np(1 - p)$

Multinoulli Distribution

- Also known as the **categorical distribution** (models categorical variables)
- Think of a random assignment of an item to one of K bins - a K dim. binary r.v. \mathbf{x} with single 1 (i.e., $\sum_{k=1}^K x_k = 1$): **Modeled by a multinoulli**

$$\underbrace{[0 \quad 0 \quad 0 \quad \dots 0 \quad 1 \quad 0 \quad 0]}_{\text{length } K}$$

- Let vector $\mathbf{p} = [p_1, p_2, \dots, p_K]$ define the probability of going to each bin
 - $p_k \in (0, 1)$ is the probability that $x_k = 1$ (assigned to bin k)
 - $\sum_{k=1}^K p_k = 1$
- The multinoulli is defined as: $\text{Multinoulli}(\mathbf{x}; \mathbf{p}) = \prod_{k=1}^K p_k^{x_k}$
- Mean: $\mathbb{E}[x_k] = p_k$
- Variance: $\text{var}[x_k] = p_k(1 - p_k)$

Multinomial Distribution

- Think of repeating the Multinoulli N times
- Like distributing N items to K bins. Suppose x_k is count in bin k

$$0 \leq x_k \leq N \quad \forall k = 1, \dots, K, \quad \sum_{k=1}^K x_k = N$$

- Assume probability of going to each bin: $\mathbf{p} = [p_1, p_2, \dots, p_K]$
- Multinomial models the bin allocations via a discrete vector \mathbf{x} of size K

$$[x_1 \quad x_2 \quad \dots x_{k-1} \quad x_k \quad x_{k+1} \dots \quad x_K]$$

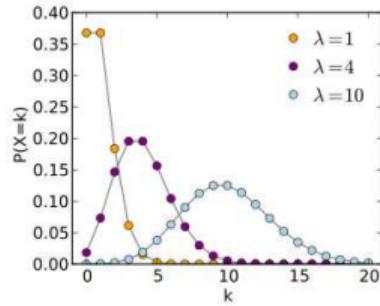
- Distribution defined as

$$\text{Multinomial}(\mathbf{x}; N, \mathbf{p}) = \binom{N}{x_1 x_2 \dots x_K} \prod_{k=1}^K p_k^{x_k}$$

- Mean: $\mathbb{E}[x_k] = Np_k$
- Variance: $\text{var}[x_k] = Np_k(1 - p_k)$
- Note: For $N = 1$, multinomial is the same as multinoulli

Poisson Distribution

- Used to model a non-negative integer (count) r.v. k
- Examples: number of words in a document, number of events in a fixed interval of time, etc.
- Defined by a positive rate parameter λ
- Distribution defined as
$$\text{Poisson}(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \quad k = 0, 1, 2, \dots$$



- Mean: $\mathbb{E}[k] = \lambda$
- Variance: $\text{var}[k] = \lambda$

The Empirical Distribution

- Given a set of points ϕ_1, \dots, ϕ_K , the empirical distribution is a discrete distribution defined as

$$p_{emp}(A) = \frac{1}{K} \sum_{k=1}^K \delta_{\phi_k}(A)$$

where $\delta_\phi(\cdot)$ is the **dirac function** located at ϕ , s.t.

$$\delta_\phi(A) = \begin{cases} 1 & \text{if } \phi \in A \\ 0 & \text{if } \phi \notin A \end{cases}$$

- The “weighted” version of the empirical distribution is

$$p_{emp}(A) = \sum_{k=1}^K w_k \delta_{\phi_k}(A) \quad (\text{where } \sum_{k=1}^K w_k = 1)$$

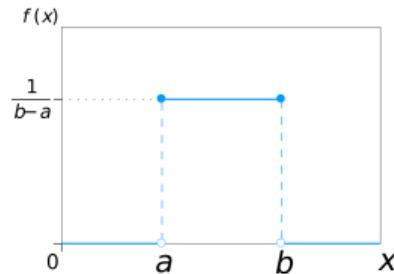
and the weights and points $(w_k, \phi_k)_{k=1}^K$ together define this discrete distribution

Continuous Distributions

Uniform Distribution

- Models a continuous r.v. x distributed uniformly over a finite interval $[a, b]$

$$\text{Uniform}(x; a, b) = \frac{1}{b - a}$$

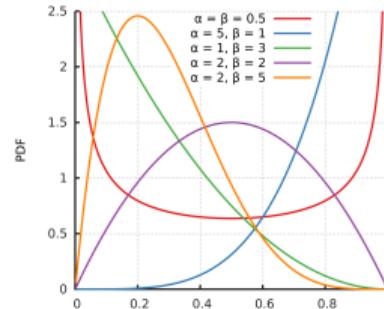


- Mean: $\mathbb{E}[x] = \frac{(b+a)}{2}$
- Variance: $\text{var}[x] = \frac{(b-a)^2}{12}$

Beta Distribution

- Used to model an r.v. p between 0 and 1 (e.g., a probability)
- Defined by two **shape parameters** α and β

$$\text{Beta}(p; \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} p^{\alpha-1} (1-p)^{\beta-1}$$

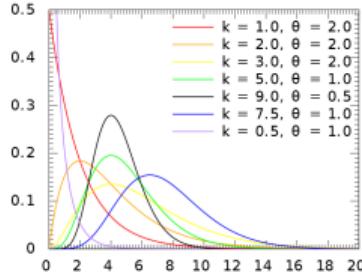


- Mean: $\mathbb{E}[p] = \frac{\alpha}{\alpha+\beta}$
- Variance: $\text{var}[p] = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$
- Often used to model the probability parameter of a Bernoulli or Binomial (also **conjugate** to these distributions)

Gamma Distribution

- Used to model positive real-valued r.v. x
- Defined by a **shape parameters** k and a **scale parameter** θ

$$\text{Gamma}(x; k, \theta) = \frac{x^{k-1} e^{-\frac{x}{\theta}}}{\theta^k \Gamma(k)}$$



- Mean: $\mathbb{E}[x] = k\theta$
- Variance: $\text{var}[x] = k\theta^2$
- Often used to model the rate parameter of Poisson or exponential distribution (conjugate to both), or to model the inverse variance (precision) of a Gaussian (conjugate to Gaussian if mean known)

Note: There is another equivalent parameterization of gamma in terms of **shape** and **rate** parameters (rate = 1/scale). Another related distribution: Inverse gamma.

Dirichlet Distribution

- Used to model non-negative r.v. vectors $\mathbf{p} = [p_1, \dots, p_K]$ that sum to 1

$$0 \leq p_k \leq 1, \quad \forall k = 1, \dots, K \quad \text{and} \quad \sum_{k=1}^K p_k = 1$$

- Equivalent to a distribution over the $K - 1$ dimensional simplex
- Defined by a K size vector $\boldsymbol{\alpha} = [\alpha_1, \dots, \alpha_K]$ of positive reals

- Distribution defined as

$$\text{Dirichlet}(\mathbf{p}; \boldsymbol{\alpha}) = \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\prod_{k=1}^K \Gamma(\alpha_k)} \prod_{k=1}^K p_k^{\alpha_k - 1}$$

- Often used to model the probability vector parameters of Multinoulli/Multinomial distribution
- Dirichlet is conjugate to Multinoulli/Multinomial
- Note:** Dirichlet can be seen as a generalization of the Beta distribution. Normalizing a bunch of Gamma r.v.'s gives an r.v. that is Dirichlet distributed.

Dirichlet Distribution

- For $\mathbf{p} = [p_1, p_2, \dots, p_K]$ drawn from $\text{Dirichlet}(\alpha_1, \alpha_2, \dots, \alpha_K)$

- Mean: $\mathbb{E}[p_k] = \frac{\alpha_k}{\sum_{k=1}^K \alpha_k}$

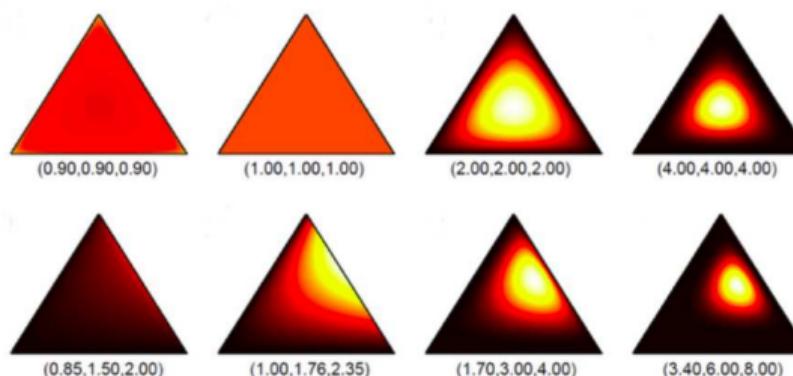
- Variance: $\text{var}[p_k] = \frac{\alpha_k(\alpha_0 - \alpha_k)}{\alpha_0^2(\alpha_0 + 1)}$ where $\alpha_0 = \sum_{k=1}^K \alpha_k$

- Note: \mathbf{p} is a point on $(K - 1)$ -simplex

- Note: $\alpha_0 = \sum_{k=1}^K \alpha_k$ controls how peaked the distribution is

- Note: α_k 's control where the peak(s) occur

Plot of a 3 dim. Dirichlet (2 dim. simplex) for various values of α :

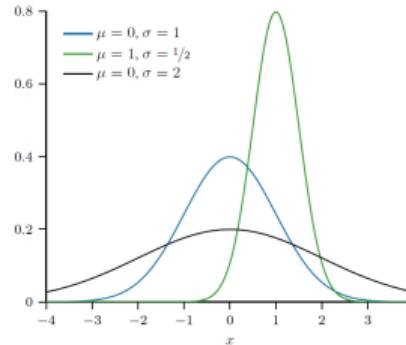


Now comes the
Gaussian (Normal) distribution..

Univariate Gaussian Distribution

- Distribution over real-valued scalar r.v. x
- Defined by a scalar **mean** μ and a scalar **variance** σ^2
- Distribution defined as

$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

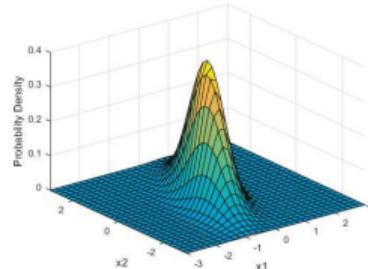


- Mean: $\mathbb{E}[x] = \mu$
- Variance: $\text{var}[x] = \sigma^2$
- Precision (inverse variance) $\beta = 1/\sigma^2$

Multivariate Gaussian Distribution

- Distribution over a multivariate r.v. vector $\mathbf{x} \in \mathbb{R}^D$ of real numbers
- Defined by a **mean vector** $\mu \in \mathbb{R}^D$ and a $D \times D$ **covariance matrix** Σ

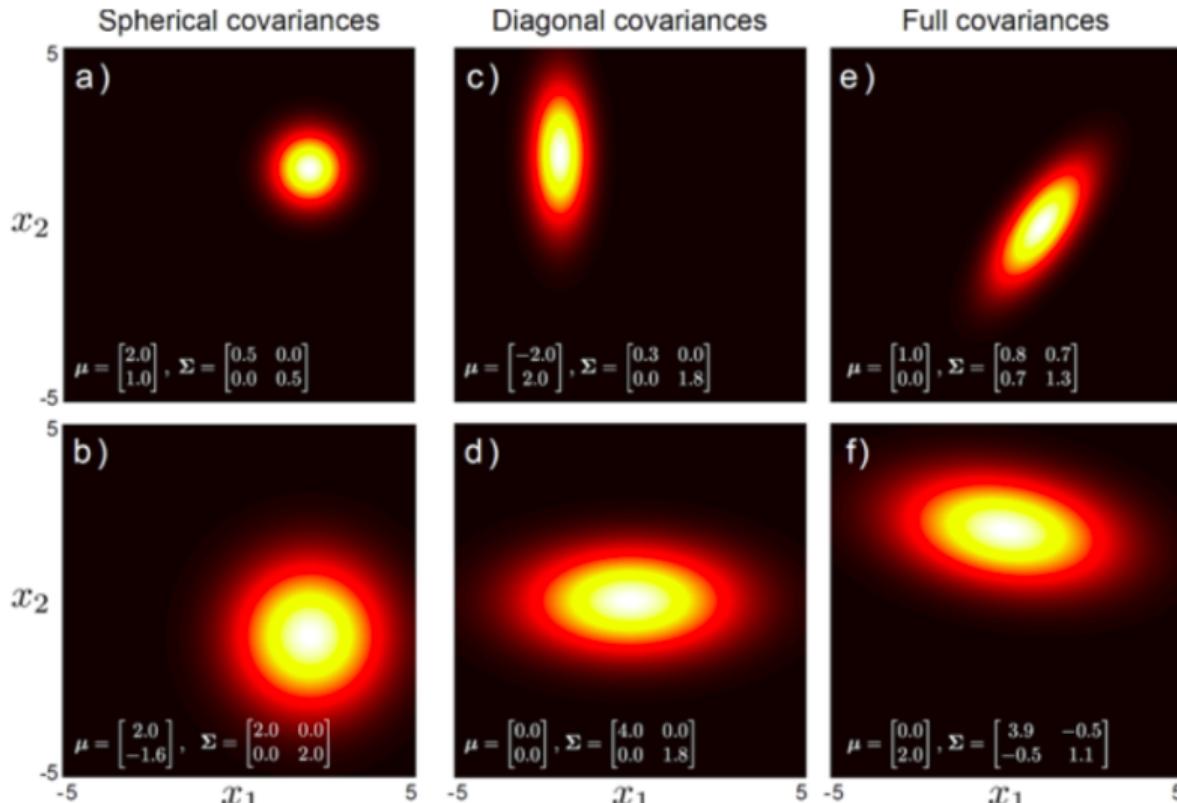
$$\mathcal{N}(\mathbf{x}; \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^D |\Sigma|}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^\top \Sigma^{-1} (\mathbf{x}-\mu)}$$



- The covariance matrix Σ must be symmetric and positive definite
 - All eigenvalues are positive
 - $\mathbf{z}^\top \Sigma \mathbf{z} > 0$ for any real vector \mathbf{z}
- Often we parameterize a multivariate Gaussian using the inverse of the covariance matrix, i.e., the **precision matrix** $\Lambda = \Sigma^{-1}$

Multivariate Gaussian: The Covariance Matrix

The covariance matrix can be spherical, diagonal, or full



Some nice properties of the Gaussian distribution..

Multivariate Gaussian: Marginals and Conditionals

- Given \mathbf{x} having multivariate Gaussian distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \Sigma)$ with $\Lambda = \Sigma^{-1}$. Suppose

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{pmatrix}, \quad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix}$$

$$\Sigma = \begin{pmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{pmatrix}, \quad \Lambda = \begin{pmatrix} \Lambda_{aa} & \Lambda_{ab} \\ \Lambda_{ba} & \Lambda_{bb} \end{pmatrix}$$

- The marginal distribution is simply

$$p(\mathbf{x}_a) = \mathcal{N}(\mathbf{x}_a | \boldsymbol{\mu}_a, \Sigma_{aa})$$

- The conditional distribution is given by

$$p(\mathbf{x}_a | \mathbf{x}_b) = \mathcal{N}(\mathbf{x}_a | \boldsymbol{\mu}_{a|b}, \Lambda_{aa}^{-1})$$

$$\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu}_a - \Lambda_{aa}^{-1} \Lambda_{ab} (\mathbf{x}_b - \boldsymbol{\mu}_b)$$

Thus **marginals and conditionals of Gaussians are Gaussians**

Multivariate Gaussian: Marginals and Conditionals

- Given the conditional of an r.v. \mathbf{y} and marginal of r.v. \mathbf{x} , \mathbf{y} is conditioned on

$$\begin{aligned} p(\mathbf{y}|\mathbf{x}) &= \mathcal{N}(\mathbf{y}|\mathbf{Ax} + \mathbf{b}, \mathbf{L}^{-1}) \\ p(\mathbf{x}) &= \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1}) \end{aligned}$$

- Marginal of \mathbf{y} and “reverse” conditional are given by

$$\begin{aligned} p(\mathbf{x}|\mathbf{y}) &= \mathcal{N}(\mathbf{x}|\boldsymbol{\Sigma}\{\mathbf{A}^T\mathbf{L}(\mathbf{y} - \mathbf{b}) + \boldsymbol{\Lambda}\boldsymbol{\mu}\}, \boldsymbol{\Sigma}) \\ p(\mathbf{y}) &= \mathcal{N}(\mathbf{y}|\mathbf{A}\boldsymbol{\mu} + \mathbf{b}, \mathbf{L}^{-1} + \mathbf{A}\boldsymbol{\Lambda}^{-1}\mathbf{A}^T) \end{aligned}$$

where $\boldsymbol{\Sigma} = (\boldsymbol{\Lambda} + \mathbf{A}^T\mathbf{L}\mathbf{A})^{-1}$

- Note that the “reverse conditional” $p(\mathbf{x}|\mathbf{y})$ is basically the posterior of \mathbf{x} if the prior is $p(\mathbf{x})$
- Also note that the marginal $p(\mathbf{y})$ is the predictive distribution of \mathbf{y} after integrating out \mathbf{x}
- Very useful property for probabilistic models with Gaussian likelihoods and/or priors. Also very handy for computing **marginal likelihoods**.

Gaussians: Product of Gaussians

- Pointwise multiplication of two Gaussians is another (unnormalized) Gaussian

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) \mathcal{N}(\mathbf{x}; \boldsymbol{\nu}, \mathbf{P}) = \frac{1}{Z} \mathcal{N}(\mathbf{x}; \boldsymbol{\omega}, \mathbf{T}),$$

where

$$\mathbf{T} = (\boldsymbol{\Sigma}^{-1} + \mathbf{P}^{-1})^{-1}$$

$$\boldsymbol{\omega} = \mathbf{T}(\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu} + \mathbf{P}^{-1}\boldsymbol{\nu})$$

$$Z^{-1} = \mathcal{N}(\boldsymbol{\mu}; \boldsymbol{\nu}, \boldsymbol{\Sigma} + \mathbf{P}) = \mathcal{N}(\boldsymbol{\nu}; \boldsymbol{\mu}, \boldsymbol{\Sigma} + \mathbf{P})$$

Multivariate Gaussian: Linear Transformations

- Given a $\mathbf{x} \in \mathbb{R}^d$ with a multivariate Gaussian distribution

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$$

- Consider a linear transform of \mathbf{x} into $\mathbf{y} \in \mathbb{R}^D$

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}$$

where \mathbf{A} is $D \times d$ and $\mathbf{b} \in \mathbb{R}^D$

- $\mathbf{y} \in \mathbb{R}^D$ will have a multivariate Gaussian distribution

$$\mathcal{N}(\mathbf{y}; \mathbf{A}\boldsymbol{\mu} + \mathbf{b}, \mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^\top)$$

Some Other Important Distributions

- Wishart Distribution and Inverse Wishart (IW) Distribution: Used to model $D \times D$ p.s.d. matrices
 - Wishart often used as a conjugate prior for modeling precision matrices, IW for covariance matrices
 - For $D = 1$, Wishart is the same as gamma dist., IW is the same as inverse gamma (IG) dist.
- Normal-Wishart Distribution: Used to model mean and precision matrix of a multivar. Gaussian
 - Normal-Inverse Wishart (NIW): : Used to model mean and cov. matrix of a multivar. Gaussian
 - For $D = 1$, the corresponding distr. are Normal-Gamma and Normal-Inverse Gamma (NIG)
- Student-t Distribution (a more robust version of Normal distribution)
 - Can be thought of as a mixture of infinite many Gaussians with different precisions (or a single Gaussian with its precision/precision matrix given a gamma/Wishart prior and integrated out)

Please refer to PRML (Bishop) Chapter 2 + Appendix B, or MLAPP (Murphy) Chapter 2 for more details

Computing the Posterior in Probabilistic Linear Regression

Piyush Rai

CS771 Supplementary Notes/Slides

August 16, 2018



Inferring the Posterior Distribution (fully Bayesian Inference)

- Inferring the full posterior is straightforward if the hyperparams β and λ to be known/fixed
 - Basically, the conjugacy helps here (Gaussian prior is **conjugate** to Gaussian likelihood)
- The posterior over the weight vector \mathbf{w} (with β and λ known)

$$p(\mathbf{w}|\mathbf{X}, \mathbf{y}, \beta, \lambda) = \frac{p(\mathbf{y}|\mathbf{X}, \mathbf{w}, \beta)p(\mathbf{w}|\lambda)}{p(\mathbf{y}|\mathbf{X}, \beta, \lambda)}$$

- Computing $P(\mathbf{w}|\mathbf{X}, \mathbf{y}, \beta, \lambda)$ (like Bernoulli-Beta case, doing it only upto proportionality constant)

$$P(\mathbf{w}|\mathbf{X}, \mathbf{y}, \beta, \lambda) \propto P(\mathbf{w}|\lambda)P(\mathbf{y}|\mathbf{X}, \mathbf{w}, \beta)$$

- After some algebra, this gets simplified into the following (proof on the next two slides)

$$P(\mathbf{w}|\mathbf{X}, \mathbf{y}, \beta, \lambda) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \quad (\text{The posterior must be Gaussian due to conjugacy})$$

$$\text{where } \boldsymbol{\Sigma} = (\beta \sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top + \lambda \mathbf{I}_D)^{-1} = (\beta \mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1}$$

$$\boldsymbol{\mu} = \boldsymbol{\Sigma}(\beta \sum_{n=1}^N \mathbf{y}_n \mathbf{x}_n) = \boldsymbol{\Sigma}(\beta \mathbf{X}^\top \mathbf{y}) = (\mathbf{X}^\top \mathbf{X} + \frac{\lambda}{\beta} \mathbf{I}_D)^{-1} \mathbf{X}^\top \mathbf{y}$$

The “Completing The Square” Trick for Gaussian Posterior

- Plugging in the respective distributions for $p(\mathbf{w}|\lambda)$ and $p(\mathbf{y}|\mathbf{X}, \mathbf{w}, \beta)$, we will get

$$\begin{aligned} p(\mathbf{w}|\mathbf{X}, \mathbf{y}, \beta, \lambda) &\propto p(\mathbf{w}|\lambda) p(\mathbf{y}|\mathbf{X}, \mathbf{w}, \beta) \\ &= \mathcal{N}(\mathbf{w}|\mathbf{0}, \lambda^{-1}\mathbf{I}_D) \mathcal{N}(\mathbf{y}|\mathbf{X}\mathbf{w}, \beta^{-1}\mathbf{I}_N) \\ &\propto \exp\left(-\frac{\lambda}{2}\mathbf{w}^\top \mathbf{w}\right) \exp\left(-\frac{\beta}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})\right) \\ &= \exp\left[-\frac{\lambda}{2}\mathbf{w}^\top \mathbf{w} - \frac{\beta}{2}(\mathbf{y}^\top \mathbf{y} + \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} - 2\mathbf{w}^\top \mathbf{X}^\top \mathbf{y})\right] \\ &\propto \exp\left[-\frac{\lambda}{2}\mathbf{w}^\top \mathbf{w} - \frac{\beta}{2}(\mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} - 2\mathbf{w}^\top \mathbf{X}^\top \mathbf{y})\right] \\ &= \exp\left[-\frac{1}{2}\left(\mathbf{w}^\top (\lambda\mathbf{I}_D + \beta\mathbf{X}^\top \mathbf{X})\mathbf{w} - 2\beta\mathbf{w}^\top \mathbf{X}^\top \mathbf{y}\right)\right] \end{aligned}$$

- We will now try to bring the exponent into a quadratic form to see if it corresponds to some Gaussian. So basically, we will use the “complete the square” trick

The “Completing The Square” Trick for Gaussian Posterior

- So we had.. $p(\mathbf{w}|\mathbf{X}, \mathbf{y}, \beta, \lambda) \propto \exp \left[-\frac{1}{2} \left(\mathbf{w}^\top (\lambda \mathbf{I}_D + \beta \mathbf{X}^\top \mathbf{X}) \mathbf{w} - 2\beta \mathbf{w}^\top \mathbf{X}^\top \mathbf{y} \right) \right]$
- Let's see if we can bring the above posterior into the form of the following Gaussian

$$\mathcal{N}(\mathbf{w}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \propto \exp \left[-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{w} - \boldsymbol{\mu}) \right] = \exp \left[-\frac{1}{2}(\mathbf{w}^\top \boldsymbol{\Sigma}^{-1} \mathbf{w} - 2\mathbf{w}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} + \boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}) \right]$$

- Let's multiply and divide $p(\mathbf{w}|\mathbf{X}, \mathbf{y}, \beta, \lambda) \propto \exp \left[-\frac{1}{2} \left(\mathbf{w}^\top (\lambda \mathbf{I}_D + \beta \mathbf{X}^\top \mathbf{X}) \mathbf{w} - 2\beta \mathbf{w}^\top \mathbf{X}^\top \mathbf{y} \right) \right]$ by $\exp \left[-\frac{1}{2} \boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} \right]$
- This gives the following up to a prop. constant (remember $\boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}$ is constant w.r.t. \mathbf{w}):

$$p(\mathbf{w}|\mathbf{X}, \mathbf{y}, \beta, \lambda) \propto \exp \left[-\frac{1}{2} \left(\mathbf{w}^\top (\lambda \mathbf{I}_D + \beta \mathbf{X}^\top \mathbf{X}) \mathbf{w} - 2\beta \mathbf{w}^\top \mathbf{X}^\top \mathbf{y} + \boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} \right) \right]$$

- Finally comparing with the expression of $\mathcal{N}(\mathbf{w}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ we can see that

$$\begin{aligned}\boldsymbol{\Sigma} &= (\lambda \mathbf{I}_D + \beta \mathbf{X}^\top \mathbf{X})^{-1} \\ \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} &= \beta \mathbf{X}^\top \mathbf{y} \quad \Rightarrow \quad \boldsymbol{\mu} = \boldsymbol{\Sigma}(\beta \mathbf{X}^\top \mathbf{y}) = (\mathbf{X}^\top \mathbf{X} + \frac{\lambda}{\beta} \mathbf{I}_D)^{-1} \mathbf{X}^\top \mathbf{y}\end{aligned}$$

- Note: The above expression for the posterior can also be directly obtained using properties of Gaussian distributions (Refer to the maths refresher slides on “reverse conditionals”, or MLAPP 4.3-4.4)



MLE for Multivariate Gaussian

Piyush Rai

Introduction to Machine Learning (CS771A)

(Supplementary Slides)



Gaussian Distribution

- The (multivariate) Gaussian with mean μ and cov. matrix Σ

$$\begin{aligned}\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \frac{1}{\sqrt{(2\pi)^D |\boldsymbol{\Sigma}|}} \exp \left\{ -\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right\} \\ &= \frac{1}{\sqrt{(2\pi)^D |\boldsymbol{\Sigma}|}} \exp \left\{ -\frac{1}{2} \text{trace} [\boldsymbol{\Sigma}^{-1} \mathbf{S}] \right\} \quad \text{where } \mathbf{S} = (\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top\end{aligned}$$

- An alternate representation: The “information form”

$$\mathcal{N}_c(\mathbf{x}|\boldsymbol{\xi}, \boldsymbol{\Lambda}) = (2\pi)^{-D/2} |\boldsymbol{\Lambda}|^{1/2} \exp \left\{ -\frac{1}{2} \left(\mathbf{x}^\top \boldsymbol{\Lambda} \mathbf{x} + \boldsymbol{\xi}^\top \boldsymbol{\Lambda}^{-1} \boldsymbol{\xi} - 2\mathbf{x}^\top \boldsymbol{\xi} \right) \right\}$$

where $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$ and $\boldsymbol{\xi} = \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}$ are the “natural parameters” (recall exp. family).

- Note that there is a term quadratic in \mathbf{x} (involves $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$) and linear in \mathbf{x} (involves $\boldsymbol{\xi} = \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}$)
- Information form can help recognize $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ of a Gaussian when doing algebraic manipulations

Estimating Parameters of Gaussian: MLE

- Given: N i.i.d. observations $\mathcal{D} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ from a multivariate Gaussian $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$
- Goal: Estimate $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. Simple to do MLE for this task

$$\mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \log p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^N \log p(\mathbf{x}_n|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^N \log \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

- Plugging in $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^D |\boldsymbol{\Sigma}|}} \exp\{-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\}$ and ignoring the constants

$$\begin{aligned}\mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \frac{N}{2} \log |\boldsymbol{\Sigma}|^{-1} - \frac{1}{2} \sum_{n=1}^N (\mathbf{x}_n - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}) \\ &= \frac{N}{2} \log |\boldsymbol{\Sigma}|^{-1} - \frac{1}{2} \sum_{n=1}^N \text{trace}[\boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu})(\mathbf{x}_n - \boldsymbol{\mu})^\top] \\ &= \frac{N}{2} \log |\boldsymbol{\Sigma}|^{-1} - \frac{1}{2} \text{trace}[\boldsymbol{\Sigma}^{-1} \mathbf{S}_\mu] \quad \left[\text{where } \mathbf{S}_\mu = \sum_{n=1}^N (\mathbf{x}_n - \boldsymbol{\mu})(\mathbf{x}_n - \boldsymbol{\mu})^\top \right]\end{aligned}$$

Estimating Parameters of Gaussian: MLE

- Taking (partial) derivatives w.r.t. μ and setting to zero

$$\frac{\partial}{\partial \mu} \mathcal{L}(\mu, \Sigma) = \frac{\partial}{\partial \mu} \left[\frac{N}{2} \log |\Sigma|^{-1} - \frac{1}{2} \sum_{n=1}^N (\mathbf{x}_n - \mu)^\top \Sigma^{-1} (\mathbf{x}_n - \mu) \right] = -\frac{1}{2} \sum_{n=1}^N (\Sigma^{-1} + \Sigma^{-\top}) (\mathbf{x}_n - \mu) = 0$$

which gives the following MLE solution for the multivariate Gaussian's mean

$$\mu_{ML} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n$$

- Taking derivatives w.r.t. $\Lambda = \Sigma^{-1}$ (instead of Σ ; [leads to simpler derivatives](#)) and setting to zero

$$\frac{\partial}{\partial \Lambda} \mathcal{L}(\mu, \Lambda) = \frac{\partial}{\partial \Lambda} \left[\frac{N}{2} \log |\Lambda| - \frac{1}{2} \text{trace}[\Lambda \mathbf{S}_\mu] \right] = \frac{N}{2} \Lambda^{-\top} - \frac{1}{2} \mathbf{S}_\mu^\top = \frac{N}{2} \Lambda^{-1} - \frac{1}{2} \mathbf{S}_\mu = \frac{N}{2} \Sigma - \frac{1}{2} \mathbf{S}_\mu = 0$$

which gives the following MLE solution for the multivariate Gaussian's covariance matrix

$$\Sigma_{ML} = \frac{1}{N} \sum_{n=1}^N (\mathbf{x}_n - \mu_{ML})(\mathbf{x}_n - \mu_{ML})^\top$$



$$\hat{\pi}_k = \frac{1}{N} \sum_{n=1}^N y_{nk}$$

Same as $\frac{N_k}{N}$ where N_k is # of training ex. for which $y_n = k$

$$\hat{\mu}_k = \frac{1}{N} \sum_{n=1}^N y_{nk} \mathbf{x}_n$$

Same as $\frac{1}{N_k} \sum_{n:y_n=k}^N \mathbf{x}_n$

$$\hat{\Sigma}_k = \frac{1}{N} \sum_{n=1}^N y_{nk} (\mathbf{x}_n - \hat{\mu}_k)(\mathbf{x}_n - \hat{\mu}_k)^\top$$

Same as $\frac{1}{N_k} \sum_{n:y_n=k}^N (\mathbf{x}_n - \hat{\mu}_k)(\mathbf{x}_n - \hat{\mu}_k)^\top$