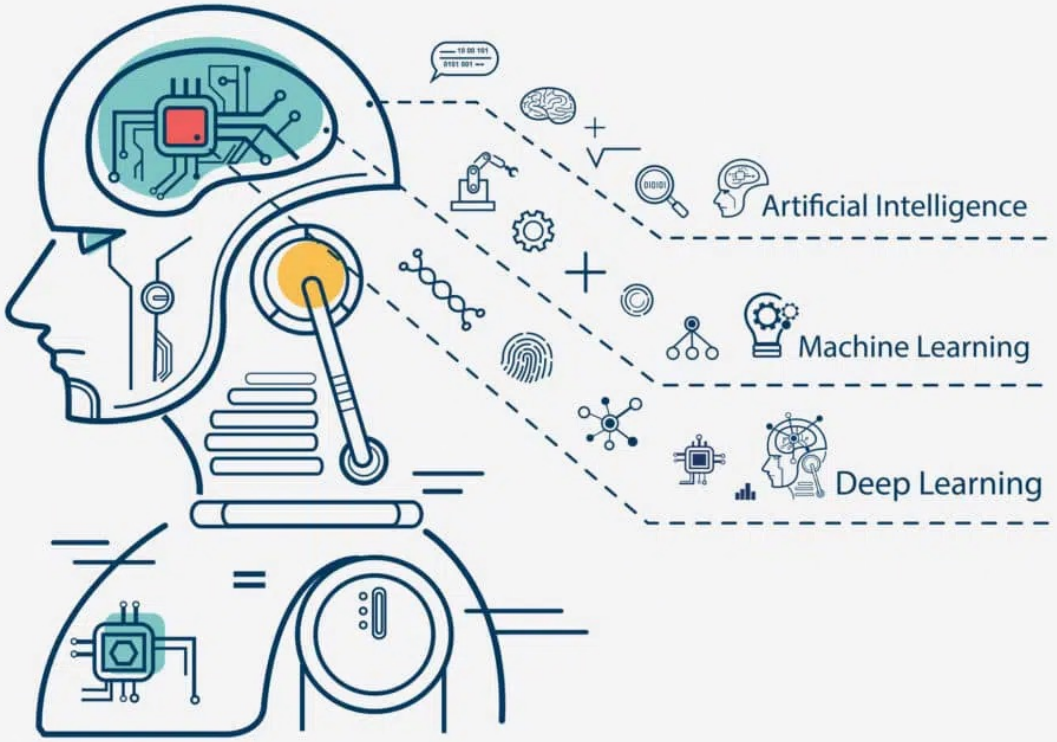


Introduction to Machine Learning

DDA-3020 Notebook

Youthy WANG

Lecture I Introduction

1) Basic Concepts in Machine Learning:

Definition of Machine Learning:

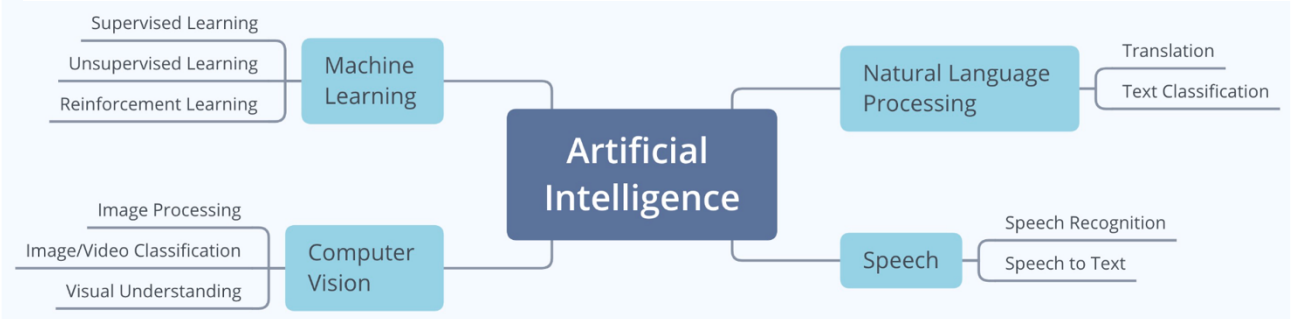
Arthur Samuel: "the field of study that gives computers the *ability to learn without being explicitly programmed*."

Tom Mitchell: "a computer program is said to learn from **experience *E*** concerning some class of **tasks *T*** and **performance measure *P***, if its performance at tasks in *T*, as measured by *P*, *improves* with experience *E*." (To start a ML problem, ***E***, ***P***, ***T*** must be figured out.)

ML as a Branch of Artificial Intelligence:

Artificial intelligence (AI) is intelligence demonstrated by machines (e.g., computers, robots), unlike the natural intelligence displayed by humans and animals, which involves consciousness and emotionality.

AI covers many topics, such as machine learning (ML), computer vision (CV), natural language processing (NLP), and speech processing.



2) Classifications:

Two Basic Paradigms of ML:

Supervised Learning: you are also provided some *human-labeled outputs* $y_1, y_2, y_3, \ldots$, and your task is to *learn a mapping function* from one input data x_i to one output y_i. (Predict something, given some other things.)

- called a **prediction model**
- called **regression** when we predict a real scalar or vector value
- called **classification** when we predict a value from a finite set such as {TRUE, FALSE}.

Unsupervised Learning: your task is to *build/learn a good model* of x, such that some ***data characteristics*** could be revealed, such as **clustering** (discrete), **dimensionality reduction** (continuous), etc.

- called a **data model**

Reinforcement Learning: you can make some actions a_i to change the data x_i (the state of the environment), and you will receive some rewards/punishment r_i. You will gradually learn to *make suitable actions for different data to get more rewards*. It is one of three basic machine learning paradigms, alongside supervised and unsupervised learning.

Discrete & Continuous: (Note that when results are countable, we call it discrete.)

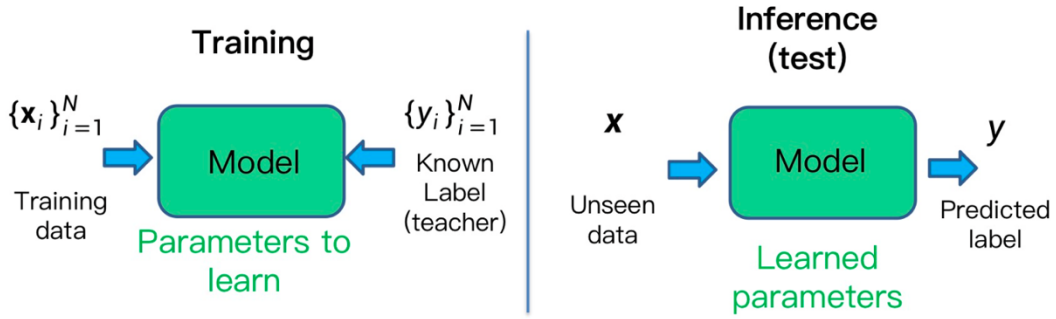
	Supervised Learning (with labels)	Unsupervised Learning (without labels)
Discrete	Classification	Clustering
Continuous	Regression	Dimensionality Reduction

3) Supervised Learning:

In supervised learning, the dataset is the collection of ***labeled examples***, denoted as $\{(\boldsymbol{x_i}, y_i)\}_{i=1}^{N}$, where $\boldsymbol{x_i} = [x_i^{(1)}, \cdots, x_i^{(m)}]$,

Every element $\boldsymbol{x_i}$ is called a ***feature vector***: it is a vector in which each dimension $\boldsymbol{j = 1, \ldots, m}$ contains a value that <u>*describes the example somehow*</u>.

The label $\boldsymbol{y_i}$ can be either an element belonging to a *finite set of classes* ***{1,2,...,C}***, or a *real number*.



General Procedures:

- ***Data collection***: collect $\{(\boldsymbol{x_i}, y_i)\}_{i=1}^{N}$.
- ***Training***: conducting model training on the training data to learn the *model parameters*.
- ***Inference (test)***: Using the trained model to *predict the output* of *unseen data* $\boldsymbol{x}$.

Basic Concepts in Supervised Learning:

- **Data for Supervised Learning**:

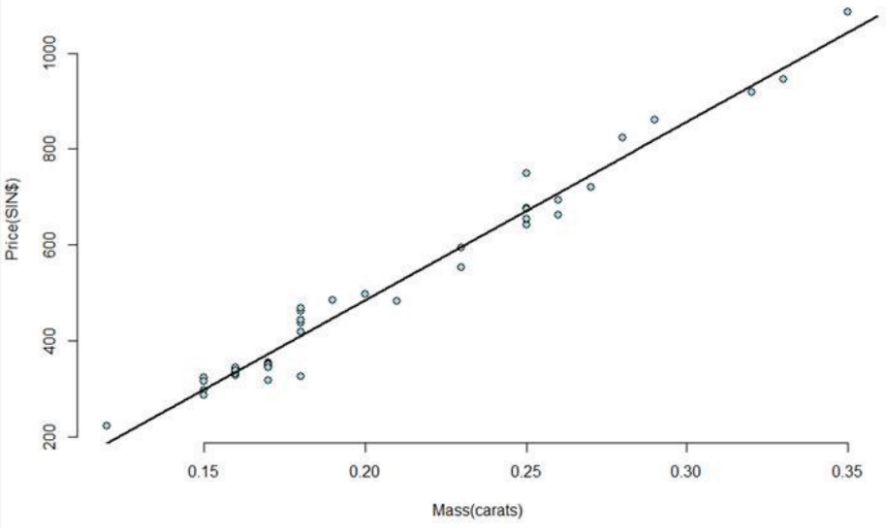
  Training Set: $\boldsymbol{D_{tr}} = \{(\boldsymbol{x_i}, y_i)\}_{i=1}^{N}$, with $\boldsymbol{x_i} \in \boldsymbol{X}$ being the feature presentation (could be scalar, image/video, text sequence, graph, cloud point, etc.), $\boldsymbol{y_i} \in \boldsymbol{Y}$ being the supervision/ground-truth value (could be a discrete label, continuous value, vector, sequence, tensor, etc.).

  Testing set: $\boldsymbol{D_{test}} = \{(\boldsymbol{x_i}, y_i)\}_{i=1}^{M}$ is used to evaluate the performance of the trained model.

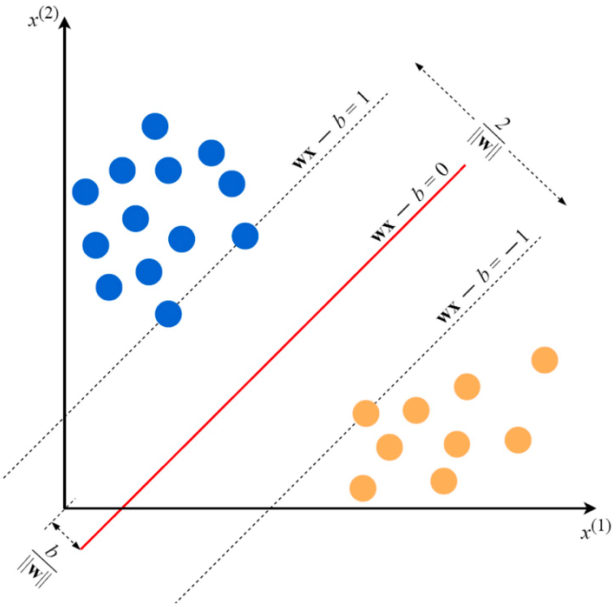
- ***Target function t***: $\boldsymbol{X \rightarrow Y}$: the ground-truth mapping function from the input to the output behind the training/testing data. It is unknown, and our goal is to find it.

- ***Hypothesis h****:* A hypothesis is *a candidate function* that *describes* the unknown target function.
- ***Hypothesis space H***: Hypothesis space is the set of all the possible legal hypotheses.
- ***Cost function:*** a *measure* of how good/wrong the hypothesis is in terms of its ability to estimate the relationship between ***x*** and ***y***, such as the square loss $(h(x) - y)^2$.
- ***Objective function***: the function that we want to optimize (minimize, maximize, or minimax). We may also call it the cost function, loss function, or error function when minimizing it. In this course, we use these terms interchangeably.
- ***Training/learning***: the process of searching for a good hypothesis ***h*** in the hypothesis space ***H***, through optimizing the objective function on a training set D_{tr} using the optimization method, such as $h^* = \arg\min_{h \in \mathcal{H}} \frac{1}{n} \sum_{(\boldsymbol{x}_i, y_i) \in D_{tr}} (h(\boldsymbol{x}_i) - y_i)^2$

Regression Example:



Classification Example:

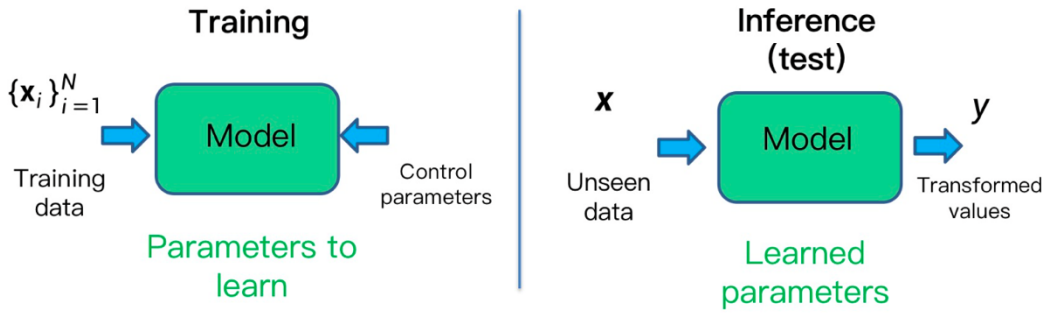


4) Unsupervised Learning:

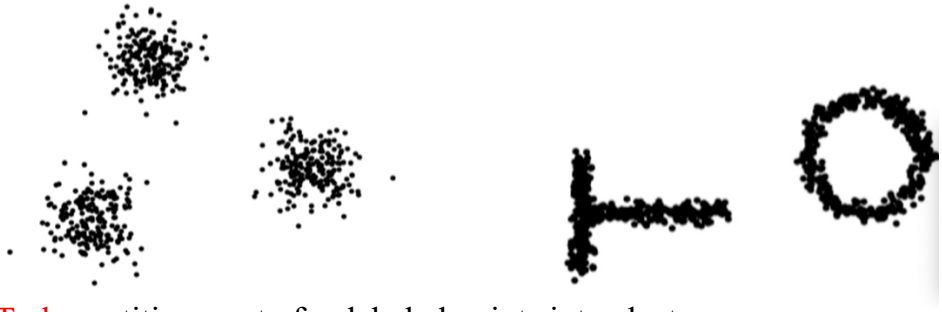
In unsupervised learning, the dataset is a collection of ***unlabeled examples***, i.e., $(\boldsymbol{x_i})_{i=1}^{N}$.

Again, $\boldsymbol{x}$ is a feature vector, and the goal of an unsupervised learning algorithm is to create a model that takes a feature vector $\boldsymbol{x}$ as input and either transforms it into another vector or into a value that can be used to solve a practical problem.

Its main task is to ***analyze the structure*** of data for future inference.



Clustering Example:



Task: partition a set of unlabeled points into clusters.

Performance (for test data):

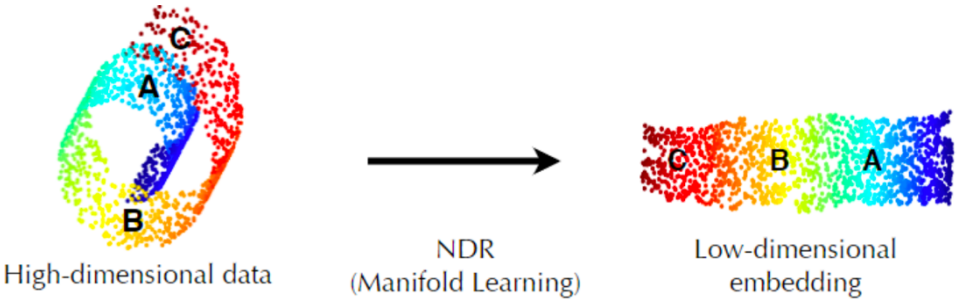
points within the same cluster are close to each other

points from different clusters are far from each other

the clusters have an appropriate coverage of all data

Experience: available data

Dimensionality-Reduction Example:



The purposes of dimensionality reduction:

Data simplification: non-linear → linear

Data visualization: high dimensional → low-dimensional

Reduce noise: some dimensions of the input data may be noises

Variable selection for prediction: learn a sparse model, if there are redundancies among different dimensions

5) A General Machine Learning Workflow:

Independent and Identically Distributed (i.i.d.) Assumption:

Standard machine learning assumes that all the samples are observations/realizations of *independently and identically distributed random variables*, and training and testing sets follow the same distribution.

1. **Collecting data**: Be it the raw data from excel, access, text files, etc., this step (gathering past data) forms the foundation of future learning. The better the variety, density, and volume of relevant data, the better the learning prospects for the machine becomes.
2. **Preprocessing data**: One needs to spend time determining the data quality and then taking steps to fix issues such as missing data and treatment of outliers.
3. **Determine the hypothesis space, objective function, and optimization method**.
4. **Training**: learning the parameters of the hypothesis function through optimizing the objective function.
5. **Testing**: evaluating the learned model on testing data.
6. **Improving the performance**: This step might involve choosing a different model altogether or introducing more variables to augment the efficiency. That's why significant time must be spent on data collection and preparation.

6) Other Learning Paradigms:

- **Reinforcement Learning**: an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward.
- **Semi-supervised Learning**: labeled data + unlabeled data
- **Ensemble Learning**: learning with multiple ML models, and the final prediction is obtained by combining these models. Improve the performance of individual models.
- **Transfer Learning**: source domain data + target domain data, useful especially when the target data is insufficient.
- **Federated Learning**: learning the model at local servers using local data, then uploading locally updated parameters to the central server to obtain the unified parameters. Protecting users' privacy.
- **Machine Unlearning**: erasing the effect of some particular training samples from a trained model. Protecting users' privacy.

Lecture II Probability Recap

1) Recap of Probability Theory:

(Details in relevant STAT notes and MATH notes)

2) Information Theory:

The Measure of Information:

Claude Shannon defined the *measure that quantifies the uncertainty* of an event with a given probability -- ***a bit***.

For a discrete random variable (a source) with a finite alphabet, as follows.

$$X = \{x_0, \cdots, x_k, \cdots, x_S\},$$

Where the probability of each symbol is given by $P(X = x_k) = p_k$.

We define the information as (for a discrete RV)

$$I(x_k) = \log(1/p_k) = -\log(p_k).$$

If the logarithm is *base 2*, information is given in *bits* (*base e – nats*).

Note that $I(x_k) \geq 0$, i.e., non-negative. The equality only holds when $p_k = 1$, which means there is ***no uncertainty, then no information. More considerable surprise/uncertainty, more information/bits.***

event	probability	surprise
one equals one	1	0 bits
wrong guess on a 4-choice question	3/4	0.415 bits
correct guess on true-false question	1/2	1 bit
correct guess on a 4-choice question	1/4	2 bits
seven on a pair of dice	6/36	2.58 bits
win any prize at Euromilhões	1/24	4.585 bits
win Euromilhões Jackpot	$\approx$ 1/76 million	$\approx$ 26 bits
gamma ray burst mass extinction today	$< 2.7 \cdot 10^{-12}$	$>$ 38 bits

Axiomatic Derivations of Uncertainty:

Let Δ_n be the set of all *finite discrete probability distributions*:

$$P = \left\{ (p_1, p_2, \ldots, p_n),\ p_i \geq 0,\ \sum_{i=1}^{n} p_i = 1 \right\}.$$

In general, the ***uncertainty*** associated with the random experiment ***P*** is a mapping ***H(P):*** $\Delta_n \rightarrow \boldsymbol{R}$, where the latter is the set of real numbers. We set the following axiom to be satisfied by the entropy function ***H(P) = H(p_1, p_2,..., p_n)***.

Axiom 1. *We assume that the entropy $H(P)$ is nonnegative, that is, for all $P = (p_1, p_2, \ldots, p_n)$, $H(P) \geq 0$. This is essential for a measure.*

Axiom 2. *We assume that generalized form of entropy function (2.3) is*

$$H(P) = \sum_{i=1}^{n} \phi(p_i). \tag{2.5}$$

Axiom 3. *We assume that the function ϕ is a continuous concave function of its arguments.*

Axiom 4. *We assume the additivity of entropy, that is, for any two statistically independent experiments $P = (p_1, p_2, \ldots, p_n)$ and $Q = (q_1, q_2, \ldots, q_m)$,*

$$H(PQ) = \sum_{j} \sum_{\alpha} \phi(p_j q_\alpha) = \sum_{j} \phi(p_j) + \sum_{\alpha} \phi(q_\alpha). \tag{2.6}$$

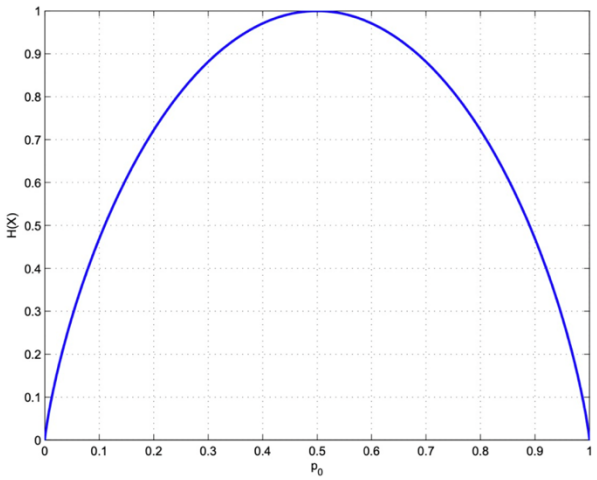
By solving the functional equation provided in Axiom 4, the conclusion is: if the entropy function ***H(P)*** satisfies Axioms 1 to 4, then ***H(P)*** is given by $H(P) = -k \sum_{i=1}^{n} p_i \log p_i$,

Entropy:

Entropy is defined as the ***expected value of information from a source***,

$$\boldsymbol{H_P(X) = E[I(X)] = \sum_k p_k I(x_k)},\ (\textstyle\int_S \boldsymbol{I(x)\mu(dx)} \text{ for } \boldsymbol{differential\ entropy}).$$

(e.g., entropy of binary source –



.)

Cross-entropy builds upon the idea of entropy. It calculates the number of bits required to represent or transmit an average event from one distribution ***P(X)***, compared to another distribution ***Q(X)***,

$$H_{P,Q}(X) = \sum_k p_k I_Q(x_k), \left(\int_S I_Q(x)\mu_P(dx) \text{ for } \textit{differential entropy}\right).$$

Properties of Cross-entropy:

1. ***Non-negativity***: i.e., $H_{P,Q}(X) \geq 0$.
2. $H_{P,Q}(X) \geq H_P(X)$, and the equality holds only when $P = Q$.

The Relative Entropy between two *continuous probability density functions* $p_X(x)$ and $q_X(x)$ is defined as follows

$$D_{P,Q}(X) = \sum_k p_k\left[I_Q(x_k) - I_P(x_k)\right]. \left(\int_S\left[I_Q(x) - I_P(x)\right]\mu_P(dx) \text{ for } \textit{differential entropy}.\right)$$

It is also called ***KL(Kullback-Leibler) divergence***, which measures the distance between two distributions.

Properties of KL Divergence:

1. ***Non-negativity***: $D_{P,Q}(X) \geq 0$
2. ***Asymmetry***: $D_{P,Q}(X) \neq D_{Q,P}(X)$.
3. $H_{P,Q}(X) = H_P(X) + D_{P,Q}(X)$.

(Proofs of above properties.)

[Lemma I] ***Shannon's Inequality on Entropy***

Given $p_k, q_k \geq 0$ for all k, the conclusion is

$$\sum_k p_k \log \frac{p_k}{q_k} \geq \left(\sum_k p_k\right) \log \frac{\sum_k p_k}{\sum_k q_k}.$$

(proof.) For finite sum, *the method of Lagrange multipliers* works. (When $p_i / \sum_k p_k = q_i / \sum_k q_k$, for every i, the equality holds.) Moreover, we can apply *Jensen's inequality* on $\frac{p_i}{\sum_i p_i} \log(x_i)$, where $x_i = q_i / p_i$, also true for the countably infinite case.

For continuous case, again use *Jensen's inequality* on $\varphi = \log$ and f be pdf of P.

Lecture III Linear Regression

1) Functions, Derivatives, and Gradients:

Special Functions and Their Properties

A **vector function**, denoted as $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$, is a function that returns $\boldsymbol{y}$, which can have either a vector argument ($\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$) or a scalar argument ($\boldsymbol{y} = \boldsymbol{f}(x)$).

A function $\boldsymbol{f}: \boldsymbol{R}^d \rightarrow \boldsymbol{R}$ is **linear** if it satisfies the following two properties:

(i) **Homogeneity (with degree 1)**: For any d-vector $\boldsymbol{x}$ and a scalar α, $\boldsymbol{f}(\alpha \boldsymbol{x}) = \alpha \boldsymbol{f}(\boldsymbol{x})$.

(ii) **Additivity**: For any d-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, $\boldsymbol{f}(\boldsymbol{x} + \boldsymbol{y}) = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{f}(\boldsymbol{y})$.

This property is called **superposition** (which consists of homogeneity and additivity).

Derivatives and Gradients

(i) Differentiation of a *scalar* function w.r.t. a *vector*:

$$\frac{df(\mathbf{w})}{d\mathbf{w}} = \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \vdots \\ \frac{\partial f}{\partial w_d} \end{bmatrix}$$

(ii) Differentiation of a *vector* function w.r.t. a *vector*:

$$\frac{d\mathbf{f}(\mathbf{w})}{d\mathbf{w}} = \begin{bmatrix} \frac{\partial f_1}{\partial w_1} & \cdots & \frac{\partial f_h}{\partial w_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_1}{\partial w_d} & \cdots & \frac{\partial f_h}{\partial w_d} \end{bmatrix}$$

(iii) Some Vector-Matrix Differentiation Formulas:

$$\frac{d(\mathbf{X}^\top \mathbf{w})}{d\mathbf{w}} = \mathbf{X}, \text{ where } \mathbf{X} \text{ is not a function of } \mathbf{w}$$

$$\frac{d(\mathbf{y}^\top \mathbf{X}\mathbf{w})}{d\mathbf{w}} = \mathbf{X}^\top \mathbf{y}$$

$$\frac{d(\mathbf{w}^\top \mathbf{X}\mathbf{w})}{d\mathbf{w}} = (\mathbf{X} + \mathbf{X}^\top)\mathbf{w}$$

Note that we adopt the ***denominator layout*** *derivative*. If you use the ***numerator layout*** *derivative*, all the above results will be transposed.

2) Modelling of Linear Regression Questions:

Deterministic Perspective

We want to learn a linear model $f_{w,b}(\boldsymbol{x})$, i.e., linear hypothesis function (w.r.t. $[\boldsymbol{w}; b]$),

$$f_{w,b}(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{w} + b,$$

where $\boldsymbol{w}$ is a d-dimensional vector of parameters, and the ***bias parameter*** b is a real number.

The task $\boldsymbol{T}$ to linear regression is to use the linear model $f_{w,b}(\boldsymbol{x})$ to approximate the *ground-truth target function* t: $\boldsymbol{X} \rightarrow \boldsymbol{Y}$. (Note: If $\boldsymbol{Y}$ is a *finite and discrete set*, then the task corresponds to a *classification* problem; if $\boldsymbol{Y}$ is a *continuous space*, then the task corresponds to a *regression* problem.)

Empirical risk / Cost function (avg. of loss functions): $\frac{1}{m}\sum_{i=1}^{m}(f_{\mathbf{w},b}(\mathbf{x}_i)-y_i)^2$. Notice that $(f_{w,b}(\boldsymbol{x}_i) - y_i)^2$ is called the ***loss function***. All model-based learning algorithms have a loss function and what we do to find the best model is try to minimize the objective known as the cost function.

Probabilistic Perspective ("simple linear model" in STA2002 notes)

We assume that the relationship between the input variable/feature $\boldsymbol{x}$ and the output variable y is

$$y = \boldsymbol{w}^\top\boldsymbol{x} + e, \text{ where } e \sim N(0, \sigma^2),$$

where e is called **observation noise** or **residual error**, and it is independent of any specific input $\boldsymbol{x}$. Under this circumstance, $y|\boldsymbol{w}, \boldsymbol{x} \sim N(\boldsymbol{w}^\top\boldsymbol{x}, \sigma^2)$, MLE can be used to find $\boldsymbol{w}$:

$$\mathbf{w}_{MLE} = \arg\min_{\mathbf{w}} \frac{1}{2}\sum_{i=1}^{m}(y_i - \mathbf{w}^\top\mathbf{x}_i)^2,$$

Analytical Solution $\boldsymbol{X^TX\hat{w} = X^Ty}$ (details in MAT2040 notes)

The *gradient descent method* can be used for updating $\boldsymbol{w}$ if $\boldsymbol{X^TX}$ is not invertible or size of $\boldsymbol{w}$ is way large.

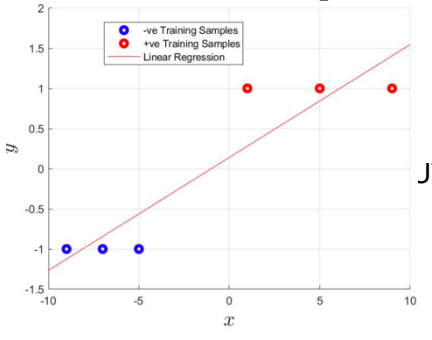
Multiple Outputs

When considering the entire set of data indexed by $i = 1, \ldots, m$, a linear model $\boldsymbol{f}_{w,b}(\boldsymbol{x}) = \boldsymbol{x}^\top\boldsymbol{W} + \boldsymbol{b}^\mathrm{T}$ can be packed as $\boldsymbol{f}_{w,b}(\boldsymbol{x}) = [1; \boldsymbol{x}]^\top[\boldsymbol{b}^\mathrm{T}; \boldsymbol{W}] = \tilde{\boldsymbol{x}}^\top\widetilde{\boldsymbol{W}}$.

Analytically, if $\tilde{\boldsymbol{X}}^\top\tilde{\boldsymbol{X}}$ is invertible, then learning: $\widehat{\widetilde{\boldsymbol{W}}} = (\tilde{\boldsymbol{X}}^\top\tilde{\boldsymbol{X}})^{-1}\tilde{\boldsymbol{X}}^\top\boldsymbol{y}$. Since the *squared-error loss function* can be written compactly using $\boldsymbol{E} = \tilde{\boldsymbol{X}}\widetilde{\boldsymbol{W}} - \boldsymbol{Y}$ and total loss is $\mathrm{tr}(\boldsymbol{E}^\top\boldsymbol{E})$.

3) Linear Regression & Classification

Binary Classification: $\boldsymbol{X^TX\hat{w}} = \boldsymbol{X^Ty}$, where $y_i \in \{-1, +1\}$, i = 1,...,m. But we predict by $\boldsymbol{f}_{w,b}(\boldsymbol{x}_{new}) = \mathrm{sgn}\,(\hat{\boldsymbol{w}}^\top\boldsymbol{x}_{new})$. (NOT satisfactory enough!)



Multi-Category Classification: $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\widehat{\boldsymbol{W}} = \boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}$. But we predict by $\boldsymbol{f}_{w,b}(\boldsymbol{x}_{new})$ = argmax$_{i=1,\ldots,C}$ $(\widehat{\boldsymbol{w}}^{\top}\boldsymbol{x}_{new})$. Add the number of parameters (i.e., size of $\widehat{\boldsymbol{W}}$) to target more precise predictions. Every row of ($i=1,\ldots,m$) in $\boldsymbol{Y}$ has a *one-hot assignment*: e.g., the target for class-2 is labeled as $\boldsymbol{y}_i^{\top}$ = [0, 1, 0, ..., 0] for the i^{th} sample.

4) Variants of Linear Regression:

Ridge Regression

Motivations: (i) We cannot guarantee the matrix $\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}$ is invertible; (ii) Avoid **overfitting**, since overfitting is a fundamental challenge for linear regression (poor performance).

$$\min_{\mathbf{w},b} \sum_{i=1}^{m} \left(f_{\mathbf{w},b}\left(\mathbf{x}_i\right) - y_i\right)^2 + \lambda \bar{\mathbf{w}}^{\top}\bar{\mathbf{w}}, \text{ where } \bar{\mathbf{w}} = \hat{\mathbf{I}}_d\mathbf{w} = [0, w_1, w_2, \ldots, w_d]^{\top},$$

Here, we shall focus on single output y in derivations in the sequel. The bias/offset b is NOT included in the l_2 regularization term, as it affects the function's height, not complexity.

Solve ridge regression, we get $\mathbf{w} = \left(\mathbf{X}^{\top}\mathbf{X} + \lambda\hat{\mathbf{I}}_d\right)^{-1}\mathbf{X}^{\top}\mathbf{y}$.

Under the probabilistic view, we can assume that the parameter $\boldsymbol{w}$ (excluding the bias b) follows a **zero-mean Gaussian prior**, $\boldsymbol{w} \sim \mathrm{N}(\boldsymbol{0}, \boldsymbol{\tau}^2\boldsymbol{I})$ (Bayesian point of view). Utilizing this prior, we obtain the ***maximum a posteriori (MAP) estimation***, $\boldsymbol{w}_{\mathrm{MAP}}$ = argmax $f_{\boldsymbol{W}|\boldsymbol{Y}=\boldsymbol{y},\,\boldsymbol{X}=\boldsymbol{x}}(\boldsymbol{w})$ = argmax $f_{\boldsymbol{Y}=\boldsymbol{y}|\boldsymbol{X}=\boldsymbol{x},\,\boldsymbol{W}}(\boldsymbol{x})\, f_{\boldsymbol{W}}(\boldsymbol{w}) = \arg\min_{\mathbf{w}} \left[\sum_{i=1}^{m}(\mathbf{x}_i^{\top}\mathbf{w} - y_i)^2 + \lambda\|\mathbf{w}\|_2^2\right]$.

The resulting curve will be ***smoother*** when we set a larger λ (penalty), i.e., more weight on the prior.

Polynomial Regression

Some data, such as the classic XOR data, may *not be linearly separated*. Accordingly, we can design *a novel linear regression model*, by adding higher-order terms. The method is called polynomial regression.

$$f_{\mathbf{w},b}(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i x_i + \sum_{i=1}^{d}\sum_{j=1}^{d} w_{ij}x_i x_j + \sum_{i=1}^{d}\sum_{j=1}^{d}\sum_{k=1}^{d} w_{ijk}x_i x_j x_k + \ldots = \phi(\mathbf{x})^{\top}\mathbf{w},$$

The number of polynomial terms becomes explosive for high dimensional d and high polynomial order. Hence, for high dimensional problems, polynomials of order larger than *3* are seldom used.

For Regression Applications

- Learning: $\hat{\mathbf{w}} = (\mathbf{P}^\top \mathbf{P} + \lambda \mathbf{I})^{-1} \mathbf{P}^\top \mathbf{y}$, where $\mathbf{y}$ is continuous
- Prediction: $f_{\mathbf{w},b}(\mathbf{P}(\mathbf{X}_{new})) = \mathbf{P}_{new}\hat{\mathbf{w}}$

For Classification Applications

- Learn **discrete** valued $\mathbf{y}$ (binary) or $\mathbf{Y}$ (one-hot)
- Binary Prediction: $f_{\mathbf{w},b}(\mathbf{P}(\mathbf{X}_{new})) = \mathbf{sgn}(\mathbf{P}_{new}\hat{\mathbf{w}})$ if $y \in \{-1, +1\}$
- Multi-Category Prediction: $f_{\mathbf{w},b}(\mathbf{P}(\mathbf{X}_{new})) = \mathbf{argmax}_{i=1,\ldots,C}(\mathbf{P}_{new}\hat{\mathbf{w}})$

LASSO Regression

We can replace the ***Gaussian prior*** with a ***Laplacian prior***, i.e.,

$$\boldsymbol{w} \sim Lap(\boldsymbol{0}, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|\mathbf{w}|}{\lambda}\right)$$

The combination of the Gaussian distribution of $p(y|\boldsymbol{x}, \boldsymbol{w})$ and the Laplacian prior, leads to $\boldsymbol{w}_{\text{MAP}} = \arg\min_{\mathbf{w}} \left[\sum_{i=1}^{m} (\mathbf{x}_i^\top \mathbf{w} - y_i)^2 + \alpha|\mathbf{w}|\right]$.

Robust Linear Regression

Motivation: Avoid ***the influence of outliers*** since learned parameters $\boldsymbol{w}_{\text{MLE}}$ will be significantly influenced by them, leading to an inferior fit.

We adopt the l_1 loss to replace the l_2 loss *to increase robustness*.

$$J(\mathbf{w}) = \sum_{i=1}^{m} |\mathbf{x}_i^\top \mathbf{w} - y_i|.$$

When the residual is large, the l_1 loss is much smaller than the l_2 loss, so the influence of outliers could be alleviated. The l_1 loss can be converted into linear programmings.

Another way to improve robustness is **iteratively reweighted** least squares method.

$$\min_{\mathbf{w}} \min_{\mu_1,\ldots,\mu_m > 0} \frac{1}{2}\left(\frac{(\mathbf{x}_i^\top \mathbf{w} - y_i)^2}{\mu_i} + \mu_i\right).$$

Probabilistic Perspective

$p(y|\mathbf{x}, \mathbf{w})$	$p(\mathbf{w})$	regression method
Gaussian	Uniform	Least squares
Gaussian	Gaussian	Ridge regression
Gaussian	Laplace	Lasso regression
Laplace	Uniform	Robust regression
Student	Uniform	Robust regression

Lecture IV Logistic Regression

1) Classification:

Threshold Classifier with Linear Regression (the 1st trial)

We assume a linear hypothesis function $f_{w,b}(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{w} + b$

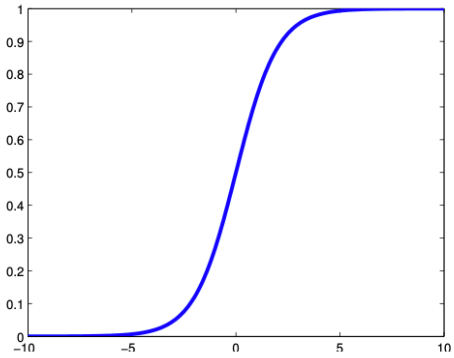
A simple threshold classifier with this hypothesis function is 0.5 (compare $f_{w,b}(\boldsymbol{x})$ with 0.5 for classifications).

But there is still something weird. Our goal is to predict $y \in \{0,1\}$, but the prediction could be $f_{w,b}(\boldsymbol{x}) > 1$ or $f_{w,b}(\boldsymbol{x}) < 0$, which does not serve our purpose.

A desired hypothesis function for this task should be $f_{w,b}(\boldsymbol{x}) \in [0, 1]$.

Hypothesis Representation (improvement)

Introduce ***sigmoid function*** or ***logistic function***: $g(z) = [1 + \exp(-z)]^{-1}$



$f_{w,b}(\boldsymbol{x}) = g(\boldsymbol{x}^\top \boldsymbol{w}) \in [0, 1]$, as the new representation.

Interpretation of sigmoid/logistic function:

$f_{w,b}(\boldsymbol{x})$ = **estimated probability** that $y = 1$ of input $\boldsymbol{x}$, i.e., $f_{w,b}(\boldsymbol{x}) = P(y = 1|\boldsymbol{x};\boldsymbol{w})$.

Suppose that if $f_{w,b}(\boldsymbol{x}) \geqslant 0.5$, then we predict $y = 1$; if $f_{w,b}(\boldsymbol{x}) < 0.5$, then we predict $y = 0$

It determines the ***decision boundary***, which is the curve/hyper-plane corresponding to $f_{w,b}(\boldsymbol{x}) = 0.5$.

2) Logistic Regression:

Cost Function

l_2 loss is *non-convex* w.r.t. $\boldsymbol{w}$ for **logistic regression**.

Recall **cross-entropy**: if we set ***ground-truth posterior probability*** $y(\boldsymbol{x}) = P(y = 1|\boldsymbol{x})$, ***predicted posterior probability*** $f_{w,b}(\boldsymbol{x}) = P(y = 1|\boldsymbol{x};\boldsymbol{w})$, then,

$$loss\,[y(\boldsymbol{x}), f_{w,b}(\boldsymbol{x})] = H\,[y(\boldsymbol{x}), f_{w,b}(\boldsymbol{x})] = \begin{cases} -\log(f_{\mathbf{w},b}(\mathbf{x})), & \text{if } y(\mathbf{x}) = 1 \\ -\log(1 - f_{\mathbf{w},b}(\mathbf{x})), & \text{if } y(\mathbf{x}) = 0 \end{cases}$$

Cost function of logistic regression:

$$J(\boldsymbol{w}) = \frac{1}{\mathrm{m}}\sum_{i=1}^{m} loss[y(\boldsymbol{x}), f_{w,b}(\boldsymbol{x})] = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log(f_{\mathbf{w},b}(\mathbf{x}_i)) + (1 - y_i)\log(1 - f_{\mathbf{w},b}(\mathbf{x}_i))\right].$$

Multi-class Cases

One-vs-all logistic regression:

- Train a binary logistic regression $f_{wj,bj}(\cdot)$ for each class j, by setting all samples of other classes as negative class.
- For a new testing sample $\boldsymbol{x}$, predict its class as $argmax_j\, f_{wj,bj}(\boldsymbol{x})$.

Pros: Easy to implement.

Cons: The *training cost is too high*, and is difficult to scale to tasks with large number of classes.

We use the ***Softmax function*** instead:

$$f_{\mathbf{W},\mathbf{b}}^{(j)}(\mathbf{x}) = \frac{\exp(\mathbf{w}_j^\top \mathbf{x} + b_j)}{\sum_{c=1}^{C}\exp(\mathbf{w}_c^\top \mathbf{x} + b_c)} = P(y = j|\mathbf{x};\mathbf{W},\mathbf{b}),$$

where $\boldsymbol{W} = [\boldsymbol{w}_1,\ldots,\boldsymbol{w}_C]$, $\boldsymbol{b} = [b_1;\ldots;b_C]$ with C being the number of classes. For simplicity, in the following we write $f_{\mathbf{W},\mathbf{b}}^{(j)}(\cdot)$ as $f_{\mathbf{w}_j,b_j}(\cdot)$

Cost function:

$$J(\mathbf{W}) = -\frac{1}{m}\sum_{i}^{m}\sum_{j}^{C}\left[\mathbb{I}(y_i = j)\log(f_{\mathbf{w}_j,b_j}(\mathbf{x}_i))\right],$$

Note: $\{\mathbf{w}_c\}_{c=1}^{C}$ should be updated *in parallel*, rather than *sequentially*.

Regularized Logistic Regression:

Aim: to address the overfitting issue.

Generally, there are two approaches to addressing the overfitting problem, including:

❖ Reducing *the number of features*:

- Feature selection
- Dimensionality reduction (introduced in later lectures)

❖ Regularization:

- Keep all features, but *reduce the magnitude/value of each parameter*, such that each feature contributes a bit to predicting y

In the following, we will focus on the regularization-based approach.

$$\bar{J}(\mathbf{w}) = J(\mathbf{w}) + \frac{\lambda}{2m}\sum_{j=1}^{d} w_j^2$$

$$= -\frac{1}{m}\sum_{i}^{m}\left[y_i \log(f_{\mathbf{w},b}(\mathbf{x}_i)) + (1-y_i)\log(1-f_{\mathbf{w},b}(\mathbf{x}_i))\right] + \frac{\lambda}{2m}\sum_{j=1}^{d} w_j^2.$$

Note: the bias parameter w_0 (or b) is not regularized/penalized.

Probabilistic Perspective

Behind logistic regression, we assume that

$\mu(\boldsymbol{x},\boldsymbol{w})$ = Sigmoid($\boldsymbol{w}^\top\boldsymbol{x}$),

$y|\boldsymbol{x},\boldsymbol{w} \sim$ Bernoulli($\mu(\boldsymbol{x},\boldsymbol{w})$)

l_2-regularized logistic regression: assume prior distribution $\boldsymbol{w} \sim N(\boldsymbol{0}, \sigma^2\boldsymbol{I})$

l_1-regularized logistic regression: assume prior distribution $\boldsymbol{w} \sim Lap(\boldsymbol{0}, b)$

	Linear regression	Logistic regression
Task	regression	classification
Hypothesis $f_{\mathbf{w},b}(\mathbf{x})$	$\mathbf{w}^\top\mathbf{x}+b \in (-\infty,\infty)$	$g(\mathbf{w}^\top\mathbf{x}+b) \in [0,1]$
Objective $J(\mathbf{w})$	$\frac{1}{2m}\sum_i^m (y_i - \mathbf{w}^\top\mathbf{x}_i)^2$	$-\frac{1}{m}\sum_{i=1}^m [y_i \log(f_{\mathbf{w},b}(\mathbf{x}_i)) + (1-y_i)\log(1-f_{\mathbf{w},b}(\mathbf{x}_i))]$
Solution	closed-form or gradient descent	gradient descent

Note that: For each variant of linear/logistic regression, you can derive it from both the deterministic and the probabilistic perspectives.

Own reading: Both linear regression and logistic regression are special cases of generalized linear models. If interested, you can find more details from Section 4 of th book "Pattern Recognition and Machine Learning", Bishop, 2006.

Coding in Python: Basics of PyTorch:

1. Tensors:

Tensor is a specialized data structure that is very similar to arrays and matrices. In PyTorch, we use tensors to *encode the inputs and outputs of a model*, as well as the *model's parameters*.

Tensors are similar to NumPy's ndarrays, except that tensors *can run on GPUs or other hardware accelerators*. In fact, tensors and NumPy arrays can often share the same underlying memory, eliminating the need to copy data. Tensors are also ***optimized for automatic differentiation***.

```
>>> import torch
>>> import numpy as np
```

Initializing a Tensor:

- **Directly from data**:

  Tensors can be created directly from data. The data type is automatically inferred.

```
>>> data = [[1,2],[True,False],[0.5,3.2]]
>>> t_data = torch.tensor(data)
>>> t_data
tensor([[1.0000, 2.0000],
        [1.0000, 0.0000],
        [0.5000, 3.2000]])
```

- **From a NumPy array**:

  Tensors can be created from NumPy arrays (and vice versa).

```
>>> arr = np.array(data)
>>> t_np = torch.from_numpy(arr)
>>> arr
array([[1. , 2. ],
       [1. , 0. ],
       [0.5, 3.2]])
>>> t_np
tensor([[1.0000, 2.0000],
        [1.0000, 0.0000],
        [0.5000, 3.2000]], dtype=torch.float64)
```

- **From another tensor**:

The new tensor *retains the properties (shape, datatype)* of the argument tensor unless explicitly overridden.

```
>>> t_ones = torch.ones_like(t_np,dtype=torch.int) # overrides the datatype
>>> t_rand = torch.rand_like(t_data) # retains the properties of t_data
>>> print(f"Ones Tensor: \n {t_ones} \n")
Ones Tensor:
 tensor([[1, 1],
        [1, 1],
        [1, 1]], dtype=torch.int32)

>>> print(f"Random Numbers Tensor: \n {t_rand} \n")
Random Numbers Tensor:
 tensor([[0.4441, 0.0039],
        [0.3069, 0.0915],
        [0.4122, 0.1474]])
```

- **With random or constant values**:

  `shape` is a *tuple of tensor dimensions*. In the functions below, it determines the dimensionality of the output tensor.

```
>>> shape = (2,3,4)
>>> rand_tensor = torch.rand(shape)
>>> print(f"Random Tensor: \n {rand_tensor} \n")
Random Tensor:
 tensor([[[0.4489, 0.8589, 0.6211, 0.4280],
         [0.6264, 0.1339, 0.5808, 0.1093],
         [0.8480, 0.4389, 0.8267, 0.3324]],

        [[0.4045, 0.0103, 0.6255, 0.4557],
         [0.8855, 0.2574, 0.2799, 0.4438],
         [0.0807, 0.3895, 0.8949, 0.0798]]])
```

Attributes of a Tensor:

Tensor attributes describe their *shape, datatype, and the device* on which they are stored.

```
>>> # use the same tensor as above
>>> print(f"Shape of tensor: {rand_tensor.shape}")
Shape of tensor: torch.Size([2, 3, 4])
>>> print(f"Data Type of tensor: {rand_tensor.dtype}")
Data Type of tensor: torch.float32
>>> print(f"Device where tensor is stored: {rand_tensor.device}")
Device where tensor is stored: cpu
```

Operations on Tensors:

There are over 100 tensor operations, including arithmetic, linear algebra, matrix manipulation (transposing, indexing, slicing), sampling, etc.

Each of these operations can be run on the GPU (at typically higher speeds than on a CPU). If you're using Colab, allocate a GPU by going to

Runtime > Change runtime type > GPU.

By default, tensors are created on the CPU. We need to *explicitly move tensors to the GPU using* `.to` *method* (after checking for GPU availability). Keep in mind that copying large tensors across devices can be expensive in terms of time and memory!

```
>>> if torch.cuda.is_available():
        tensor = tensor.to("cuda")
```

Standard numpy-like indexing and slicing:

```
tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[..., -1]}")
tensor[:,1] = 0
print(tensor)
```

Use `torch.cat` to **concatenate** a sequence of tensors along a given dimension.

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

Arithmetic operations:

```
# This computes the matrix multiplication between two tensors. y1, y2, y3 will have the same value
# ``tensor.T`` returns the transpose of a tensor
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(y1)
torch.matmul(tensor, tensor.T, out=y3)


# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
```

Lecture V Support Vector Machines

1) Derivation of SVM:

Binary Classification:

Given training data set $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^m$, and $\boldsymbol{x}_i \in \boldsymbol{R}^n$, $y_i \in \{-1, +1\}$.

We adopt the sign hypothesis function $y = \text{sgn}[f_{\boldsymbol{w}}(\boldsymbol{x})] = \text{sgn}(\boldsymbol{w}^\top \boldsymbol{x})$.

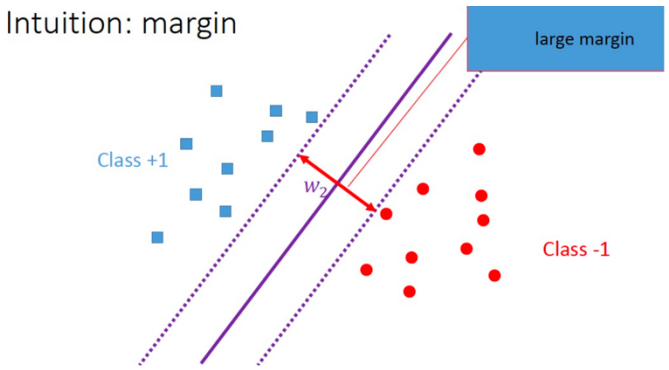
For **standard logistic regression**, the objective function (i.e., cross-entropy loss) is *convex*, rather than *strongly/strictly convex*. Consequently, there are multiple values of parameters that can perfectly fit the training data.

For **regularized logistic regression**, the objective function (i.e., cross-entropy loss + $\lambda \cdot l_2$ regularization) is *strictly convex*, which has a unique optimal solution. However, it depends on the **trade-off hyper-parameter** λ. Use **cross-validation** to use a suitable λ.

SVM provides an elegant way to solve above problem caused by logistic regression.

Large Margin:

We introduce the concept ***margin***: the distance from the *closest point of positive and negative classes* to the decision boundary. ***Support vector machine (SVM)*** chooses the decision boundary with the largest margin, which is also called *large margin classifier*.



Derive the *margin*:

(i) Distance of a vector $\boldsymbol{v}$ to the hyper-plane $\boldsymbol{x}^\top \boldsymbol{w} = a$:

$d^2 = \min_x\{\|\boldsymbol{v} - \boldsymbol{x}\|^2 : \boldsymbol{x}^\top \boldsymbol{w} = a\}$, by the method of Lagrange multipliers, $d = \frac{|a - \boldsymbol{v}^\top \boldsymbol{w}|}{\|\boldsymbol{w}\|}$.

(ii) Due to symmetry, and by ***scaling***, we need to maximize $\frac{y_i f_{\boldsymbol{w},b}(x_i)}{\|\boldsymbol{w}\|}$, which is equivalent to minimizing ½ $\boldsymbol{w}^\top \boldsymbol{w}$.

$$\begin{aligned} &\min_{\mathbf{w},b} && \tfrac{1}{2}\|\mathbf{w}\|^2 \\ &\text{subject to} && y_i\left(\mathbf{w}^\top \mathbf{x}_i + b\right) \geq 1, \forall i \end{aligned}$$

Hinge Loss:

The objective function of *support vector machine*:

$$\frac{1}{m}\sum_{i}^{m}\left[\delta_{y_i=1}\text{cost}_1(\mathbf{w}^\top\mathbf{x}_i+b)+\delta_{y_i=-1}\text{cost}_{-1}(\mathbf{w}^\top\mathbf{x}_i+b)\right]+\frac{\lambda}{2m}\sum_{j=1}^{n}w_j^2$$

Requirements of **cost$_1$** and **cost$_{-1}$** functions:

- If $y_i = +1$, we require that $\boldsymbol{w}^\top\boldsymbol{x}_i+b \geq 1$, **cost$_1$** $(\boldsymbol{w}^\top\boldsymbol{x}_i+b) = 0$.
- If $y_i = -1$, we require that $\boldsymbol{w}^\top\boldsymbol{x}_i+b \leq -1$, **cost$_{-1}$** $(\boldsymbol{w}^\top\boldsymbol{x}_i+b) = 0$.

Thus, we introduce ***Hinge-loss***:

$$\max\{0, 1-y_i(\boldsymbol{w}^\top\boldsymbol{x}_i+b)\} = [1-y_i(\boldsymbol{w}^\top\boldsymbol{x}_i+b)]^+.$$

However, hinge loss is *non-smooth*. We transform the objective function of the support vector machine to minimize a quadratic function under linear constraints.

$$\min_{\mathbf{w},b}\ \frac{1}{2}\|\mathbf{w}\|^2$$
$$s.t.\ y_i(\mathbf{w}^\top\mathbf{x}_i+b)\geq 1, \forall i$$

2) Solve the SVM:

KKT Conditions for SVM:

$$\mathcal{L}(\mathbf{w},b,\boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{w}\|^2+\sum_{i}^{m}\alpha_i\big(1-y_i(\mathbf{w}^\top\mathbf{x}_i+b)\big),$$

- Stationarity:

$$\frac{\partial L}{\partial \mathbf{w}} = 0 \ \Rightarrow\ \mathbf{w} = \sum_{i}^{m}\alpha_i y_i\mathbf{x}_i$$

$$\frac{\partial L}{\partial b} = 0 \ \Rightarrow\ \sum_{i}^{m}\alpha_i y_i = 0$$

- Feasibility:

$$\alpha_i \geq 0,\ 1-y_i(\mathbf{w}^\top\mathbf{x}_i+b)\leq 0,\ \forall i$$

- Complementary slackness:

$$\alpha_i\big(1-y_i(\mathbf{w}^\top\mathbf{x}_i+b)\big) = 0,\ \forall i$$

Replacing the stationary condition into Lagrange function, we have the ***Lagrangian dual***

$$\max_{\boldsymbol{\alpha}}\ \sum_{i}^{m}\alpha_i-\frac{1}{2}\sum_{i,j}^{m}\alpha_i\alpha_j y_i y_j\mathbf{x}_i^\top\mathbf{x}_j,$$
$$s.t.\ \sum_{i}^{m}\alpha_i y_i = 0,\ \alpha_i\geq 0,\ \forall i$$

Then, we replace the solved α back into the stationary condition, thus we obtain the primal solution $\boldsymbol{w} = \sum_i^m \alpha_i y_i \mathbf{x}_i$.

If $\alpha_i = 0$, then it means that $\boldsymbol{x}_i$ doesn't contribute to $\boldsymbol{w}$, i.e., the SVM classifier.

The data points with $\alpha_i > 0$ construct the classifier, and they are called ***support vectors***, which locate at the hyperplanes $y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) = 1$. And, we define the support set as $S = \{i | \alpha_i > 0\}$. This is why we call it a support vector machine.

Determine the bias parameter b:

$$\Rightarrow b = \frac{1}{|\mathcal{S}|} \sum_{j \in \mathcal{S}} \left(y_j - \sum_i^m \alpha_i y_i \mathbf{x}_i^\top \mathbf{x}_j\right)$$

Prediction with SVM:

Given the optimized parameters $\{\boldsymbol{\alpha}, \boldsymbol{w}, b\}$, given a new data $\boldsymbol{x}$, its prediction is

$$\mathbf{w}^\top \mathbf{x} + b = \sum_i^m \alpha_i y_i \mathbf{x}_i^\top \mathbf{x} + \frac{1}{|\mathcal{S}|} \sum_{j \in \mathcal{S}} \left(y_j - \sum_i^m \alpha_i y_i \mathbf{x}_i^\top \mathbf{x}_j\right)$$

(The RHS is always used in prediction because $\boldsymbol{\alpha}$ is a sparse vector).

If $\boldsymbol{w}^\top \boldsymbol{x} + b > 0$, then the predicted class of $\boldsymbol{x}$ is +1, otherwise −1.

If and only if $y(\boldsymbol{w}^\top \boldsymbol{x} + b) > 0$, then your prediction is correct.

SVM with Slack Variables:

However, sometimes samples of different classes are overlapped (i.e., *non-separable*), as shown below. Consequently, some constraints will be violated, and we can not obtain the feasible solution.

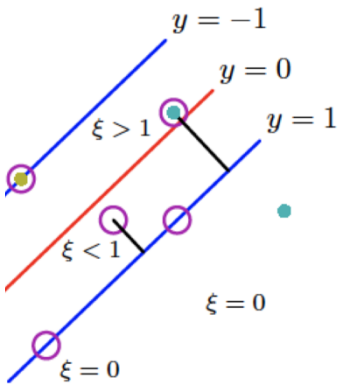
To handle such data, we introduce slack variable $\xi_i \geq 0$. We *allow some errors* for training data, i.e., $y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) \geq 1 - \xi_i$, $\forall i$, and we hope that such errors ξi,∀i are small.

$$\min_{\mathbf{w}, b, \boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + C \sum_i^m \xi_i$$
$$s.t.\ 1 - \xi_i - y_i(\mathbf{w}^\top \mathbf{x}_i + b) \leq 0, -\xi_i \leq 0,\ \forall i$$

By taking the Lagrangian dual, we get

$$\max_{\boldsymbol{\alpha}, \boldsymbol{\mu}} \sum_i^m \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j,$$
$$s.t. \sum_i^m \alpha_i y_i = 0,\ 0 \leq \alpha_i \leq C, \mu_i \geq 0, \alpha_i = C - \mu_i,\ \forall i$$

Interpretation:



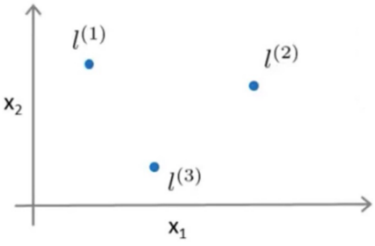
- The solution α_i has three cases: $\alpha_i = 0, 0 < \alpha_i < C, \alpha_i = C$
- $\alpha_i = 0$: the corresponding data are correctly classified and doesn't contribute to the classifier, locating outside of the margin
- $0 < \alpha_i < C$: in this case, $\mu_i > 0$ due to $\alpha_i = C - \mu_i$; Since $\mu_i \xi_i = 0$, then we have $\xi_i = 0$. The corresponding data are correctly classified and contributes to the classifier, locating on the margin
- $\alpha_i = C$: in this case, $\mu_i = 0$; then we have $\xi_i > 0$. The corresponding data contributes to the classifier, locating inside the margin
  - If $\xi_i \leq 1$, then the data is still correctly classified, not crossing decision boundary
  - If $\xi_i > 1$, then the data is incorrectly classified, crossing decision boundary

3) Kernels:

In many real problems, such as image classification, the dimensionality of original features $|\boldsymbol{x}|$ is already very high. Consequently, the dimensionality of the high-order polynomial function will be too high, causing high computational costs or overfitting.

Kernels come to the rescue with SVM for treating cases of non-linear separable data.

Given a new data $\boldsymbol{x}$, compute its new features based on proximity o land-marks $l^{(1)}$, $l^{(2)}$, $l^{(3)}$ (plot right), and here we use the Gaussian kernel, as follows



$$f_1 = \text{similarity}(\mathbf{x}, l^{(1)}) = \exp\left(-\frac{\|\mathbf{x} - l^{(1)}\|^2}{2\sigma^2}\right)$$

$$f_2 = \text{similarity}(\mathbf{x}, l^{(2)}) = \exp\left(-\frac{\|\mathbf{x} - l^{(2)}\|^2}{2\sigma^2}\right)$$

$$f_3 = \text{similarity}(\mathbf{x}, l^{(3)}) = \exp\left(-\frac{\|\mathbf{x} - l^{(3)}\|^2}{2\sigma^2}\right)$$

Then, we *have a new representation* $[f_1; f_2; f_3]$ for the data $\boldsymbol{x}$. We can set all training data points as landmarks.

We firstly define the following ***kernel***

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \varphi(\boldsymbol{x}_i)^{\mathsf{T}} \varphi(\boldsymbol{x}_j),$$

Utilizing this kernel to replace $\boldsymbol{x}_i^{\mathsf{T}} \boldsymbol{x}_j$, we have the following dual problem

$$\max_{\boldsymbol{\alpha}} \sum_i^m \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j), \qquad b = \frac{1}{|\mathcal{S}|} \sum_{j \in \mathcal{S}} \left(y_j - \sum_i^m \alpha_i y_i k(\mathbf{x}_i, \mathbf{x}_j)\right)$$

$$s.t. \sum_i^m \alpha_i y_i = 0, \ \alpha_i \geq 0, \ \forall i \qquad \mathbf{w}^{\mathsf{T}} \mathbf{x} + b = \sum_i^m \alpha_i y_i k(\mathbf{x}_i, \mathbf{x}) + b$$

Widely-used Kernels:

$$\text{Polynomial kernel: } k(\mathbf{x}, \mathbf{x}_i) = \left(1 + \frac{\mathbf{x}^\top \mathbf{x}_i}{\sigma^2}\right)^p, \ p > 0$$

$$\text{Radial Basis Function (RBF) kernel: } k(\mathbf{x}, \mathbf{x}_i) = \exp\left\{-\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{2\sigma^2}\right\}$$

$$\text{Sigmoidal kernel: } k(\mathbf{x}, \mathbf{x}_i) = \frac{1}{1 + \exp^{-\frac{\mathbf{x}^\top \mathbf{x}_i + b}{\sigma^2}}}$$

RBF Kernel is popular because of its ***similarity to K-Nearest Neighborhood Algorithm***.

It has the advantages of *k-NN* and overcomes the space complexity problem as RBF Kernel Support Vector Machines *just need to store the support vectors* during training and not the entire dataset.

Multi-class SVM

- Many SVM packages already have built-in multi-class classification functionality.
- Otherwise, use one-vs.-all method. (Train K SVMs, one to distinguish $y = k$ from the rest, for $k = 1, 2, \ldots, K$), get $(\mathbf{w}^{(1)}, b^{(1)}), \ldots, (\mathbf{w}^{(K)}, b^{(K)})$.
- Predict the label of $\mathbf{x}$ as

$$\underset{k \in \{1,2,\ldots,K\}}{\arg\max} \left(\mathbf{w}^{(k)}\right)^\top \mathbf{x} + b^{(k)}$$

4) Logistic Regression v.s. SVM:

- $n = |\boldsymbol{x}|$ indicates the number of features, and $m = |D_{\text{train}}|$ is the number of training data.
- If n is large (relative to m), then the data is **linearly separable**, one can use **LR** or **SVM without kernel**.
- If n is small, and m is intermediate, then the data may be **non-linearly separable**, one use **SVM with Gaussian kernel**
- If n is small, and m is large, then ***create/add more features*** to make the data more separable, and one can use **LR** or **SVM without kernel**.

Lecture VI Decision Trees & Random Forest

1) Parametric & Non-parametric Models:

Parametric Models:

In training, we define a model (i.e., ***hypothesis function***) over the whole input space, and *learn its parameters* with fixed numbers from all of the training data. In testing, we use the *same model* and *parameter* set for any test input.

Limitations: (i) The trainer should determine the adopted model, which may be far from the ground-truth relationship between the input and the output. (ii) The decision/prediction is difficult to explain/understand; there is no decision procedure.

Non-parametric Models:

Divide the input space into local regions, defined by a metric, and for every input, the corresponding local model computed from the training data in that region is used.

It does not rely on strong assumptions regarding the relationship shape between the variables. Instead, *the data are allowed to speak for themselves* in determining the form of the fitted functions.

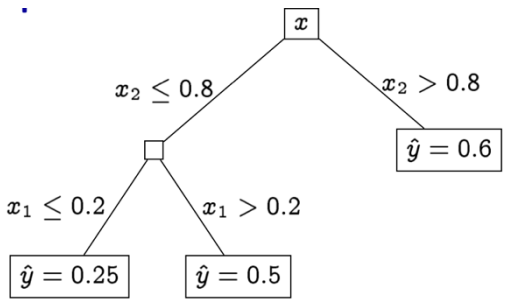
2) Univariate Trees:

Definition:

A ***decision tree*** is a **hierarchical model** for supervised learning whereby the local region is identified in a sequence of recursive splits.

In a ***univariate tree,*** the test uses *only one of the input dimensions* in each internal node.

The *partially developed Boolean tree* represents predictors.



Non-leaf nodes are associated with an ***index*** and ***threshold***.

Advantages: the ***interpretability*** of its decision, i.e., a hierarchical decision process.

Build Decision Trees:

Tree induction (a.k.a. learning or growing) is the tree's construction given a training set.

For a given training set, many trees *code it with no error*. For simplicity, we are interested in finding *the smallest* among them, where tree size is measured as the number of nodes in the tree and the complexity of the decision nodes.

Finding the smallest tree is NP-complete. Thus, we are forced to use local search procedures based on heuristics that give reasonable trees in a reasonable time.

Concrete Procedures:

- *Select an attribute* and *split the data* into its children in a tree.
- *Continue splitting* with available attributes.
- Until (or)
  - Leaf node(s) are pure (only one class remains).
  - A maximum depth is reached.
  - A performance metric is achieved.

Some rules to find the "*best attribute*":

- ***Random***: an attribute is chosen at random.
- ***Least-Values***: the attribute with the smallest number of possible values.
- ***Most-Values***: the attribute with the largest number of possible values.
- ***Impurity Measure***: the attribute that has the largest reduction of *impurity*.

Impurity:

For node m, N_m is the *number of training instances* reaching node m. For the root node, it is N. N_m^i of N_m belonging to class C^i, $i = 1, ..., K$ with $\sum_{i=1}^{K} N_m^i = N_m$.

Given that an instance reaches node m, the estimate for the probability of class C^i is $Pr(C^i \mid x, m) \equiv p_m^i = \frac{N_m^i}{N_m}$. Node m is **pure** if p_m^i for all i are either 0 or 1.

If the node is **pure**, we do not need to split any further and can add a leaf node labeled with the class for which p_m^i is 1.

For the **classification tree**, the *goodness of split/impurity* can be quantified by the ***classification error***. Count the errors, or use probability: $\varphi(p, 1-p) = 1-\max(p, 1-p)$.

We can also use entropy to estimate the error. Entropy for multi-class node

$$I_m = -\sum_{i=1}^{K} p_m^i \log_2 p_m^i.$$

Joint Entropy: $H(X, Y) = \mathbb{E}_{X,Y}[-\log p(x,y)] = -\sum_{x,y} p(x,y) \log p(x,y)$

Conditional Entropy (equivocation):

$$H(X|Y) = \mathbb{E}_Y[H(X|y)] = -\sum_{y\in Y} p(y) \sum_{x\in X} p(x|y) \log p(x|y) = -\sum_{x,y} p(x,y) \log p(x|y).$$

Mutual Information (transinformation): $I(X;Y) = \mathbb{E}_{X,Y}[SI(x,y)] = \sum_{x,u} p(x,y) \log \frac{p(x,y)}{p(x)\,p(y)}$

Relationship: $H(X, Y) = H(X \mid Y) + H(Y) = H(Y \mid X) + H(X) = H(X \mid Y) + I(X; Y) + H(Y \mid X)$

Common Impurity Measures for Binary Problems:

- **Entropy:** $\varphi(p,1-p) = -p\log_2 p - (1-p)\log_2 (1-p)$.
- **Gini Index**: $\varphi(p,1-p) = 2p(1-p)$
- **Misclassification Error**: $\varphi(p, 1-p) = 1-\max(p, 1-p)$.

Common Impurity Measures for Multi-Category Problems:

- **Entropy:** $\varphi = -\sum_{i=1}^{K} p_i \log_2 p_i$.
- **Gini Index**: $\varphi = 1 - \sum_{i=1}^{K} p_i^2$.

Regression Trees

A regression tree is almost the same as a classification tree, except that the impurity measure appropriate for classification is replaced by *a measure appropriate for regression*.

In regression, the goodness of a split is measured by **the mean square error** (MSE) or **the sum of squared errors** (SSE) from the estimated value. The total MSE is $S = \sum_{m\in leaves} e_m$. The prediction for leaf c is $\overline{y}_c$.

Basic Regression-Tree-Growing Algorithm

1. Start with a single node containing all points. Calculate $\bar{y}$ and $\mathcal{S}$.
2. For each node,
   - If all the points in the node have the same value for all the independent variables, **stop**.
   - Otherwise, search over all binary splits of all variables for the one which will reduce $\mathcal{S}$ as much as possible.
     - If the largest decrease in $\mathcal{S}$ would be less than some threshold δ, or one of the resulting nodes would contain less than q points, **stop**.
     - Otherwise, **take that split**, creating two new nodes.
3. In each new node, go back to step 1.

Others

Overfitting and Pruning

Trees *tend to overfit training data*, so the testing data prediction error is likely high.

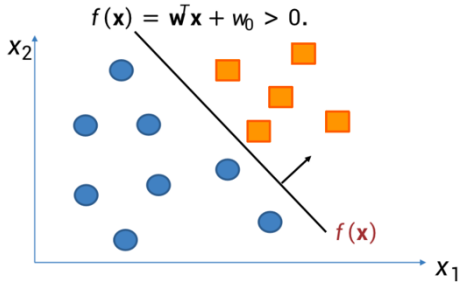
An effective approach to alleviate overfitting is ***pruning*** (剪枝). The general procedure of pruning is as follows:

- Split training data further into training and validation sets
- Grow a deep tree based on the training set
- Before further pruning is harmful:
    - Evaluate the impact on the validation set of pruning each possible node
    - Greedily remove the node that most improve validation set accuracy

Pruning of the decision tree is done by *replacing a whole subtree with a leaf node*. The replacement occurs if a decision rule establishes that the expected error rate in the subtree is greater than in the single leaf.

Multi-variate Trees and Hyper-parameters

In a multivariate tree, at a decision node, *all input dimensions can be used* and thus, it is more general. When all inputs are numeric, a binary linear multivariate node is defined as



Even though the algorithms are considered nonparametric, there are still some (hyper)-parameters used for defining a tree:

- Minimum samples for a node split.
    - Defines the minimum number of samples (or observations) which are required in a node to be considered for splitting.
    - Used to control over-fitting. Too high values can lead to under-fitting
- Minimum samples for a terminal/leaf node.
    - Defines the minimum samples (or observations) required in a terminal node or leaf.
    - Used to control over-fitting.

  - Generally, lower values should be chosen for imbalanced class problems because the regions in which the minority class will be very small.
- Maximum attributes to consider for the split.
  - The number of attributes to consider while searching for the best split. These will be randomly selected.
  - As a thumb-rule, *the square root of the total number of attributes* works great but we should check up to 30-40% of the total number of attributes.
- Maximum depth of the tree (vertical depth).
  - Used to control over-fitting as higher depth will allow the model to learn relations very specific to a particular sample.

3) Ensemble Models:

Bagging (Bootstrap Aggregating)

To address generalization issues of single decision trees, we introduce en- semble models. The basic idea is *constructing many diverse decision trees*, then combine their predictions as the final prediction (majority for classification, or average for regression).

- Step 1: Sample records with replacement (aka ”bootstrap” the training data), meaning that the individual data points can be chosen more than once, to obtain several diverse training data sets (**data-level randomness**)
- Step 2: Fit an over-grown tree to each resampled training data set. Aggregate the predictions of all single trees (**majority or average**).

However, since there are many shared samples between two resampled training data sets, Bagging is likely to produce many correlated trees. The diversity is not large enough.

Random (Decision) Forest

To reduce the correlation among the trees produced by Bagging, we introduce **split-attribute randomization** into the model. It is called ***Random Forests***.

Each time a split is to be performed, the search for the split attribute is limited to a random subset of m of the N attributes. (For regression trees: $m = N/3$; For classification trees: $m=N$).

Random forests introduces randomness into *both the data-level and attribute level*.

Lecture VII Neural Networks

1) Introduction to Neural Networks:

Vector Embedding for Features:

Conventional models for classification default assume that the feature ***x*** is given as a vector. Thus, in the real world, vector embedding is required to put the different field types on an equal footing, i.e., vectors.

Many studies are on *extracting informative features from data*, such as SIFT, HOG, and optical flow features from images/video. All these features are handcrafted, independent of the learning of the classifier.

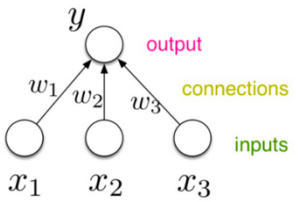
Neural networks combine ***feature learning*** and ***classifier learning***. NNs can be thought of as incorporating aspects of feature engineering into the predictor (and indeed are often used as 'automatic feature engineering').

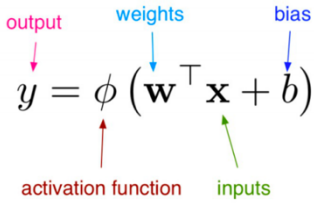
Neuron Model:

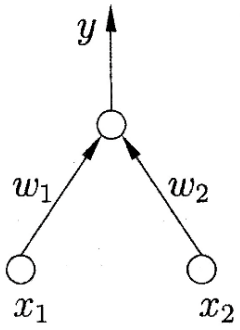
In the biological neuron model, the neurons are connected to others. There are positive and negative electric potentials in each neuron. Suppose the number of positive electric potentials is over one threshold. In that case, it is activated to send chemical substances to other connected neurons, leading to a change in the electric potentials of these neurons.

There are different activation functions:

- Linear
- Rectified Linear Unit (*ReLU*): $(x)_+$
- Soft ReLU: $log\ (1+e^x)$.
- Hard Threshold: ½ $[sgn(x)+1]$
- Sigmoid: $[1 + exp(-x)]^{-1}$
- Hyperbolic Tangent







Perceptron Model:

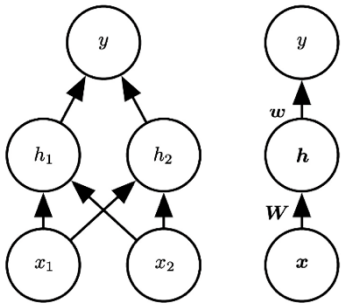
One input layer to receive input signals

One output layer, including one M-P neuron, is called the ***threshold logic unit***.

Formulation: y = $f(\boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b}) = sgn(\boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b})$

Loss function: $J(\boldsymbol{w}) = \frac{1}{2}(y - \hat{y})^2$. Learning by $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta(y - \hat{y})\boldsymbol{x}$, where the gradient of $sgn(\cdot)$ is approximated as 1.

(A perceptron/linear model can model logic 'AND,' 'OR,' and 'NOT' but cannot work for 'Exclusive OR.')



Multi-layer Feedforward Model:

Input layer, hidden layer(s), output layer

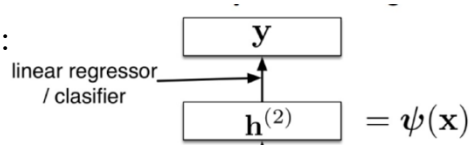
Only on *direction from the input layer to the output layer*

Pairwise connections between two layers

No connections among neurons in the same layer and no connections among neurons in non-adjacent layers (no skip connections).

Formulation: $y = g_1(\boldsymbol{w}^\top \boldsymbol{h} + b)$; $\boldsymbol{h} = g_2(W\boldsymbol{x} + \boldsymbol{c})$.

Neural nets provide modularity: we can implement each layer's computations as a black box. Neural nets can be viewed as a way of learning features:

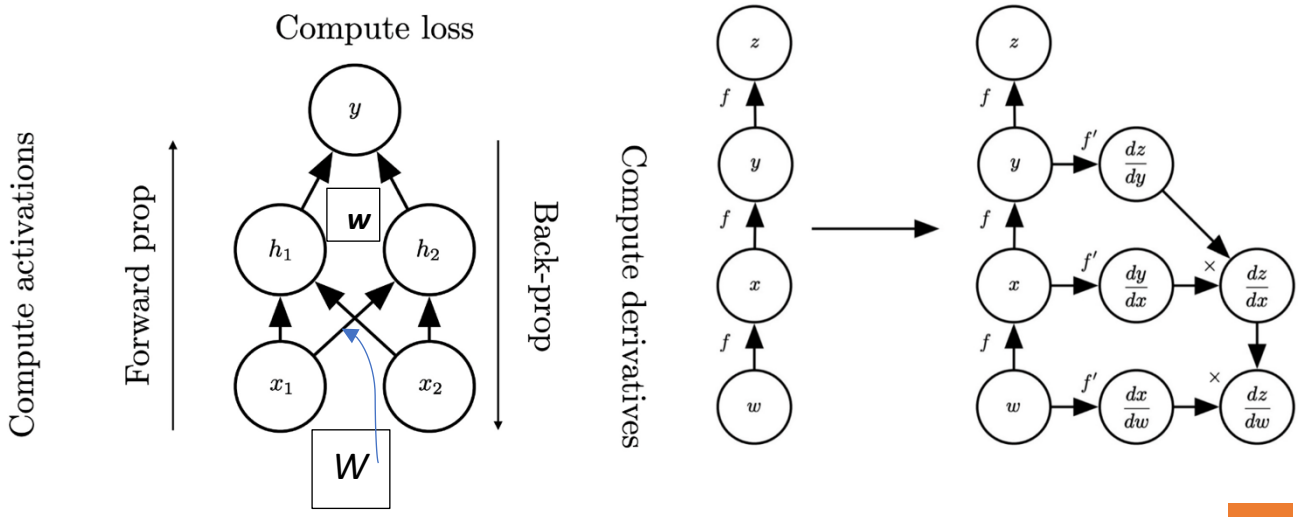


2) Backpropagation:

Generally, one uses chain rule for computing derivatives for the learning of parameters. The goal isn't to obtain closed-form solutions, but to be able to *write a program efficiently computes* the derivatives.

Computational Graph:

A computational graph (CG) is always used to diagram out the computations.



forward pass
$$\left[\begin{array}{l}\text{For } i=1,\ldots,N\\ \quad\text{Compute } v_i \text{ as a function of } \mathrm{Pa}(v_i)\end{array}\right.$$

backward pass
$$\left[\begin{array}{l}\overline{v}_N=1\\ \text{For } i=N-1,\ldots,1\\ \quad \overline{v}_i=\sum_{j\in \mathrm{Ch}(v_i)}\overline{v}_j\cdot\frac{\partial v_j}{\partial v_i}\end{array}\right.$$

Forward pass: $y = g_1(\boldsymbol{w}^\top \boldsymbol{h}+b)$, $\boldsymbol{h} = g_2(W\boldsymbol{x}+\boldsymbol{c})$.

Suppose $W \in \mathrm{R}^{m\times d}$, $\boldsymbol{x} \in \mathrm{R}^{d\times 1}$, $\boldsymbol{w} \in \mathrm{R}^{m\times 1}$, then $O_F = O(md+m)$.

Backward pass: $\frac{dL}{dW} = \frac{dL}{dy}\frac{dy}{dh}\frac{dh}{dW}$; $O_B = O(2m^2d+2m)$.

3) Deep Neural Networks:

Multilayer feed-forward neural nets with *nonlinear activation functions* are universal function approximators: they can approximate any function arbitrary well.

Motivations of Convolutional Neural Networks (CNNs):

Although deeper neural networks have larger representation capability and better generalization, it is difficult to extend the multi-layer feed-forward neural networks to very deep, since every layer is fully connected.

People resort to two tricks: ***Sparse connection***; ***Shared parameters***.

Sparse Connection:

Every input neuron only connects to partial output neurons. Every output neuron only connects to a few neighboring input neurons.

The range of input neurons is called ***receptive field***. Receptive field will increase along the layer goes deeper.

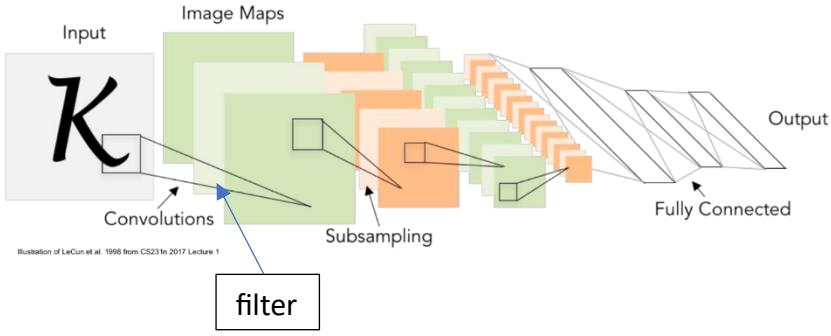
Shared Parameters:

Parameters at different spatial locations are shared.

History of CNNs:

Gradient-based learning applied to document recognition.
[LeCun, Bottou, Bengio, Haffner 1998]

Reinvigorated research in Deep Learning
[Hinton and Salakhutdinov 2006]

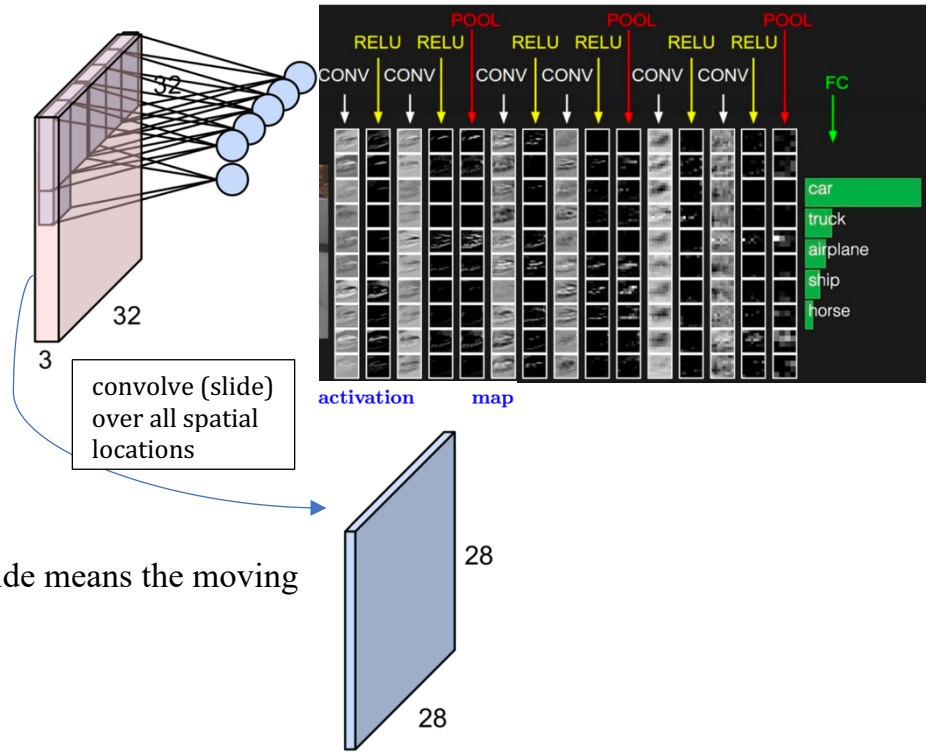


Introduction to CNNs:

ConvNet is a sequence of ***Convolution Layers***, interspersed with *activation functions*.

32×32×3 Image

5×5×3 ***Filter/Kernel*** (the result of taking a convolution between the filter and a small 5×5×3 chunk of the image, $\boldsymbol{w}^T\boldsymbol{x} + b$.)



Output Size:

$(N - F) / \text{stride} + 1$

Where N is the size of the input;

F is the size of the filter, and stride means the moving Size every time.

Output Size with Padding:

$(N + 2P - F) / \text{stride} + 1$

Let's assume the input is $W_1 \times H_1 \times C$ Conv layer needs 4 hyperparameters:

(i) Number of filters K

(ii) The filter size F

(iii) The stride S

(iv) The zero padding P

This will produce an output of $W_2 \times H_2 \times K$ where:

$$W_2 = H_2 = (W_1 - F + 2P) / S + 1$$

Number of parameters: F^2CK and K biases.

Common settings:

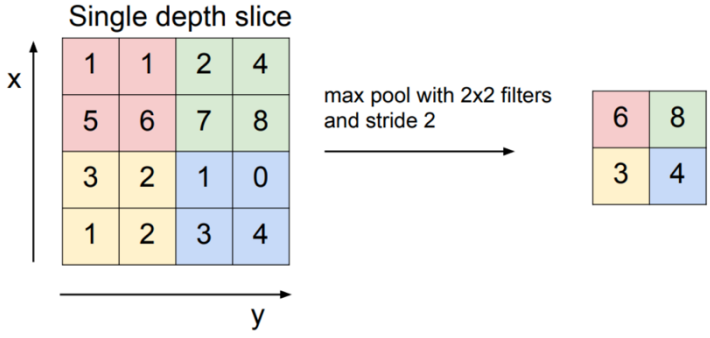
K = (powers of 2, e.g. 32, 64, 128, 512)

– $F = 3, S = 1, P = 1$

– $F = 5, S = 1, P = 2$

– $F = 5, S = 2, P = ?$ (whatever fits)

– $F = 1, S = 1, P = 0$



Pooling Layer:

makes the representations *smaller and more manageable*

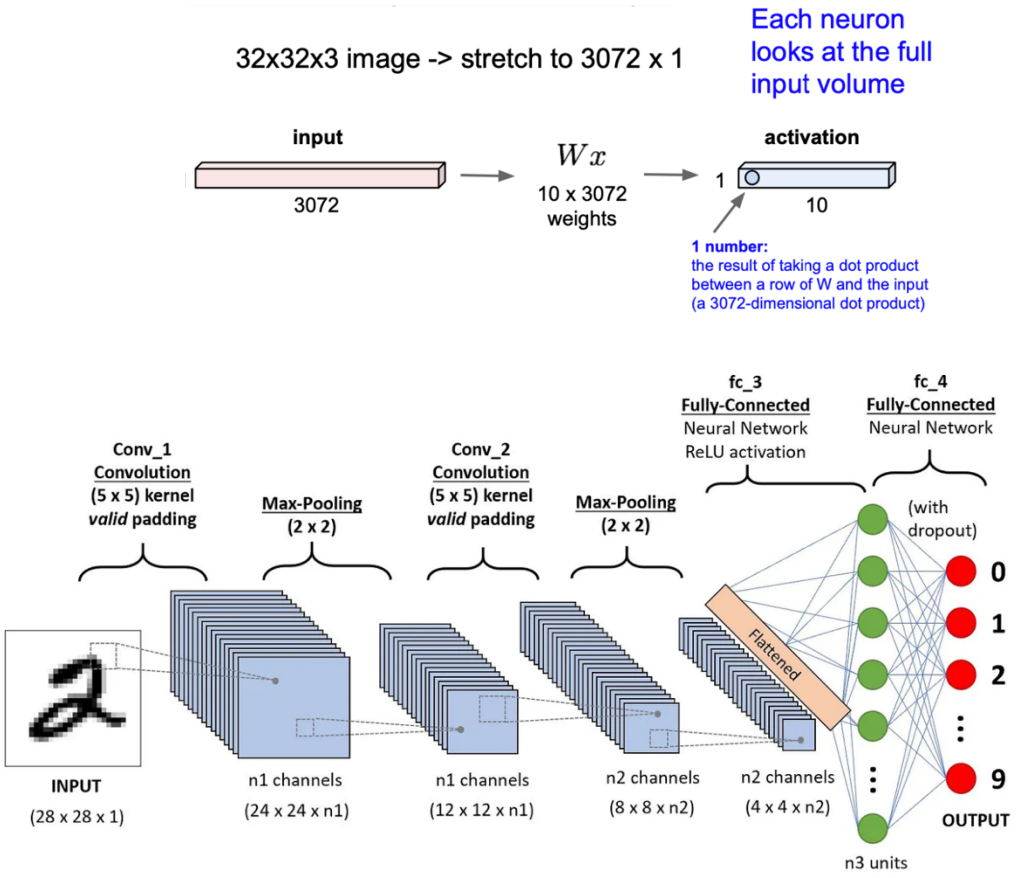
operates over every activation map independently (e.g., max pooling, average pooling)

Max pooling also performs as a ***noise suppressant*** and works better than average pooling.

Fully Connected Layer (FC layer) – Output:

Contains neurons that connect to the entire input volume, as in ordinary Neural Networks.

Adding a Fully-Connected layer is a (usually) cheap way of learning non-linear combinations of the high-level features as represented by the output of the convolutional layer. The Fully-Connected layer is learning a possibly non-linear function in that space.



Lecture VIII Over/Under-Fitting, Bias-Variance Trade-Off

1) Overfitting and Underfitting:

Underfitting:

Underfitting is the *inability* of the model to predict well the labels of the data it was trained on. There could be several reasons for underfitting, the most important of which are:

- your model is too simple for the data (for example, a linear model can often underfit)
- the features you engineered are not informative enough

The solution to the problem of underfitting is to try a *more complex model* or to ***engineer features*** with higher predictive power.

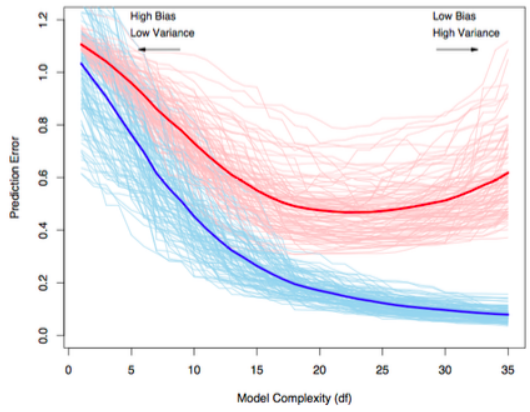
Overfitting:

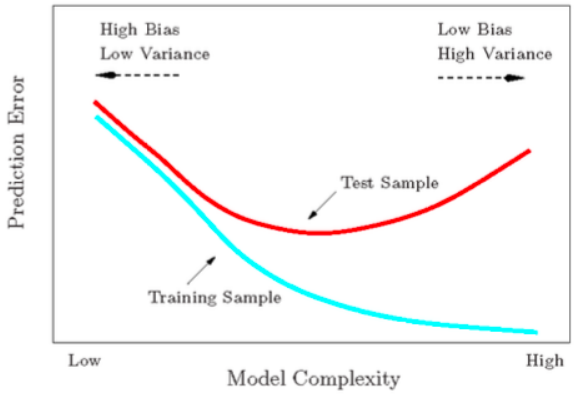
Overfitting is another problem a model can exhibit.

The model that overfits *predicts very well on the training data* but *poorly on the testing data*. Several reasons can lead to overfitting, the most important of which are:

- your model is too complex for the data (for example, a very tall decision tree or a deep or wide neural network often overfits).
- you have too many features but a small number of training examples.

2) Bias-Variance Trade-off:



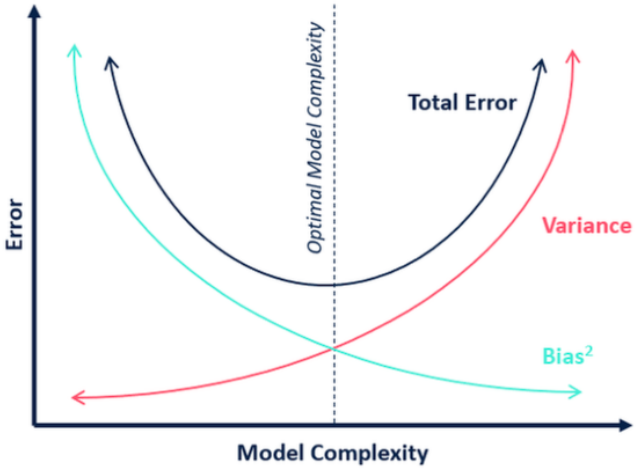


When a lot of experiments were conducted, we observed:

- The training error decreases (towards zero) with the model complexity (e.g., polynomial order).

- In terms of the testing error, for not too big training data, we have the following observations:
  - When the model complexity is low: the test prediction error is high, with *high bias* and *low variance*.
  - When the model complexity is high (e.g., high-order polynomial models): the test prediction error is low and has *low bias* and *high variance*.
  - However, the test prediction error does not decrease after a certain point.

The observation is called bias-variance trade-off. The bias-variance tradeoff is a central problem in supervised learning.



Why Trade-off – Bias-Variance Decomposition:

The goal of machine learning is to learn a ***hypothesis function*** based on the training dataset D using some learning algorithm A (e.g., logistic regression or SVM), i.e., $h_D = A(D)$.

We are provided by a training dataset $D = \{(\boldsymbol{x}_i, y_i)\}$, which is drawn i.i.d. from some distribution $P(X, Y)$. Want: $[y - h_D(x)]^2$ to be small. $(y = h(\boldsymbol{x}) + \epsilon)$

$$\mathbb{E}_{D,\epsilon}[(y - h_D(x))^2] = \mathbb{E}_{D,\epsilon}[(y - h(x))^2] + 2\mathbb{E}_{D,\epsilon}[(y - h(x))(h(x) - h_D(x))] + \mathbb{E}_{D,\epsilon}[(h(x) - h_D(x))^2]$$
$$= \mathbb{E}_{\epsilon}(\epsilon^2) + \{\mathbb{E}_D[h_D(x)] - h(x)\}^2 + \mathrm{Var}_D[h_D(x)]$$

Thus, $E =$ ***noise*** $+$ ***bias***^{2} $+$ ***variance***.

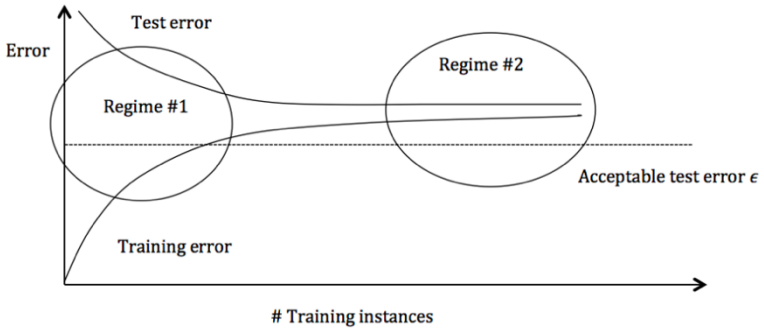
- ***variance***: Captures how much your classifier changes if you train on a different training set, how "over-specialized" is your classifier to a particular training set (overfitting). In other words, variance is due to your fixed training set.
- ***bias***^{2}: What is the inherent error that you obtain from your classifier even with infinite training data? This is due to your classifier being "biased" to a particular kind of solution (e.g., linear classifier). In other words, bias is inherent to your model.
- ***noise***: How big is the data-intrinsic noise? This error measures ambiguity due to your data distribution and feature representation. You can never beat this, it is an aspect of the data.

Finally, MSE loss function (or negative log-likelihood) is obtained by taking the expectation value over $x \sim P$:

$$\mathrm{MSE} = \mathrm{E}_x \left\{ \mathrm{Bias}_D[\hat{f}(x;D)]^2 + \mathrm{Var}_D\left[\hat{f}(x;D)\right] \right\} + \sigma^2.$$

Generally speaking,

- When the model complexity increases, the differences between the models trained on different training sets will increase, i.e., the ***variance increases*** (It is an often made fallacy to assume that complex models must have high variance, since the definition of 'complexity' varies.)
- When the model complexity increases, the average prediction of the models trained on different training sets will be closer to the target value, i.e., the ***bias decreases***
- Thus, the total test error will first decrease and then increase along with the increase in model complexity. It implies that you should choose a model with suitable complexity.



Given the same model, the changes in training and test error, along with the increase in the number of training instances.

Regime #1 (High variance)

In Regime #1, the cause of the poor performance is high variance, i.e., overfitting.

Symptoms:

Training error is much lower than test error

Training error is lower than ε

Test error is above ε

Remedies:

Add more training instances

Reduce model complexity – complex models are prone to high variance

Regime #2 (High bias)

Unlike the first regime, the second regime indicates high bias: the model used is not powerful enough to produce an accurate prediction, i.e., underfitting.

Symptoms:

Training error is higher than ε

Remedies:

Add more features

Use more complex model (e.g., kernel model, non-linear models)

Lecture IX Performance Evaluation

1) Cross-Validation, Tune Hyper-Parameters

Cross-validation aims to tune the ***hyper-parameter***s.

- ***Hyper-parameter***s: they are determined typically *outside the learning algorithms* (e.g., choosing which model, logistic regression or SVM);
- ***Parameter***s: learned by a learning algorithm based on the training set.

IDEA:

1. Split the train data into *K* folds
2. Try each fold as validation, while other folds as train
3. Train the model on the train folds, and *evaluate the performance on the validation fold*
4. Calculate the *average results* on all validation folds across all trials, and pick the hyper-parameters with the best average result.

This is called ***K*-fold cross-validation**. It is a widely used method to tune hyper-parameters. (***K*** = 5 ~ 10, in practice)

fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test

Remark:

1. The *computational cost for cross-validation is large*, which restricts its application to large-scale data. This is why it is rarely used in deep learning, which requires large-scale data.
2. Cross-validation is a *statistical method* of evaluating generalization performance that is more stable and thorough than a split into a training and a test set.

2) Evaluation Metrics for Regression

Mean Square Error **(MSE)**: $\frac{1}{n}\sum_{i=1}^{n}||\hat{y}^i - y^i||_2^2$ (for scalar y, $\frac{1}{n}\sum_{i=1}^{n}(\hat{y}^i - y^i)^2$)

Root Mean Square Error **(RMS)**: $\left(\frac{1}{n}\sum_{i=1}^{n}||\hat{y}^i - y^i||_2^2\right)^{1/2}$

Mean Absolute Error **(MAE)** (for scalar y): $\frac{1}{n}\sum_{i=1}^{n}|\hat{y}^i - y^i|$

Mean Fractional Error (for scalar, positive y, $\hat{y}$): $\frac{1}{n}\sum_{i=1}^{n}\frac{|\hat{y}^i - y^i|}{\min\{\hat{y}^i, y^i\}}$

Median Error (for scalar y): median of $|\hat{y}^i - y^i|$, $i = 1, \ldots, n$

3) Evaluation Metrics for Classification

For *Boolean Classification*:

Error Rate ***E*** is the fraction of errors: $E = \frac{1}{n}\left|\{i \mid \hat{v}^i \neq v^i\}\right|$

The error rate is the most straightforward performance metric for a classifier.

Confusion Matrix and Neyman-Pearson Metric:

Since there are only four possible values for the data pair $(\hat{v}, v)$:

- *true positive* if $\hat{v} = 1$ and $v = 1$
- *true negative* if $\hat{v} = -1$ and $v = -1$
- *false negative* or *type II error* if $\hat{v} = -1$ and $v = 1$
- *false positive* or *type I error* if $\hat{v} = 1$ and $v = -1$

One defines for a predictor and a data set the ***confusion matrix***:

$$C = \begin{bmatrix} \#\text{ true negatives} & \#\text{ false negatives} \\ \#\text{ false positives} & \#\text{ true positives} \end{bmatrix} = \begin{bmatrix} C_{\text{tn}} & C_{\text{fn}} \\ C_{\text{fp}} & C_{\text{tp}} \end{bmatrix}$$

- $C_{\text{tn}} + C_{\text{fn}} + C_{\text{fp}} + C_{\text{tp}} = n$ (total number of examples)
- $N_{\text{n}} = C_{\text{tn}} + C_{\text{fp}}$ is number of negative examples
- $N_{\text{p}} = C_{\text{fn}} + C_{\text{tp}}$ is number of positive examples
- diagonal entries give numbers of correct predictions
- off-diagonal entries give numbers of incorrect predictions of the two types

The ***basic error measures***:

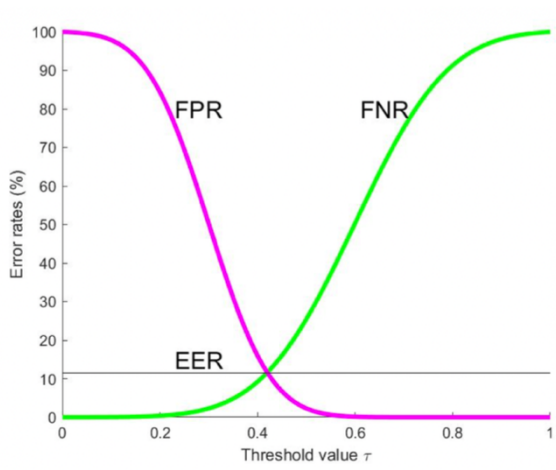
1. *False Positive Rate*: ***FPR*** = C_{fp} / n
2. *False Negative Rate*: ***FNR*** = C_{fn} / n
3. *Error Rate*: ***ER*** = $(C_{\text{fn}} + C_{\text{fp}}) / n$ = ***FPR*** + ***FNR*** (*Error Rate* + *Accuracy* = 1)

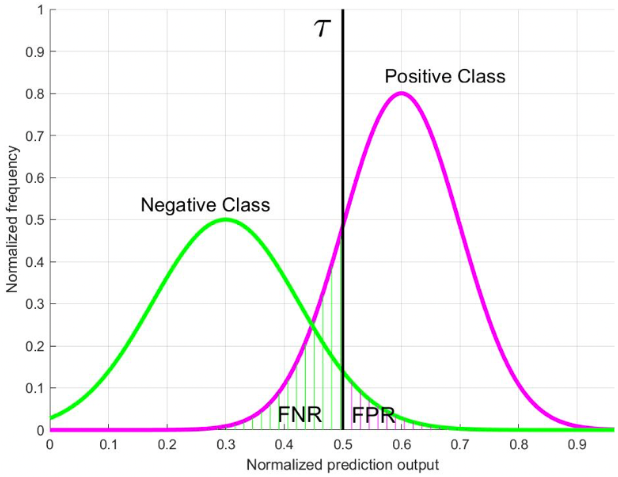
4. ***True Positive Rate / Sensitivity / Recall***: ***TPR*** = C_{tp} / N_p (fraction of true positives we correctly guess)
5. ***False Alarm Rate***: ***FAR*** = C_{fp} / N_n (fraction of true negatives we incorrectly guess as positive)
6. ***True Negative Rate / Specificity***: ***TNR*** = C_{tn} / N_n (fraction of true negatives we correctly guess)
7. ***Precision***: C_{tp} / (C_{tp}+C_{fp}) (fraction of our positive guesses that are positive)

In practice, different classes may have ***different importance***. For example, in the medical test task, the cost of predicting a malignant tumor as benign will be much higher than expecting a benign one as malignant. In such a situation, accuracy is no longer a good metric. For dealing with such a situation, a useful metric is a ***cost-sensitive accuracy***: To compute a cost-sensitive accuracy, you first assign a *cost (a positive number)* to both types of mistakes: $\widetilde{C_{fp}} = W_{fp}C_{fp}$, similar for *tp*, *tn*, *fn*. Use weighted *C* for computing error rate.

An alternative way to ***cost-sensitive accuracy*** is to obtain a single (number) metric; we combine false positives and false negatives with a weight to get the ***Neyman-Pearson metric***: $E^{\mathrm{NP}} = \kappa C_{\mathrm{fn}}/n + C_{\mathrm{fp}}/n$.

The *decision threshold* can be varied, and we get another set of ***FPR*** and ***FNR***. Each *decision threshold setting* is called an ***operating point*** (which has different FNR and FPR values). If one plot for all the operating points by changing the threshold (is τ) from 0 to 1, the *intersection point* of these two curves is when the two error rates are equal (i.e., ***TPR***=***TNR***). This point is called the ***Equal Error Rate*** (***EER***). (***EER*** = 0 → no error)

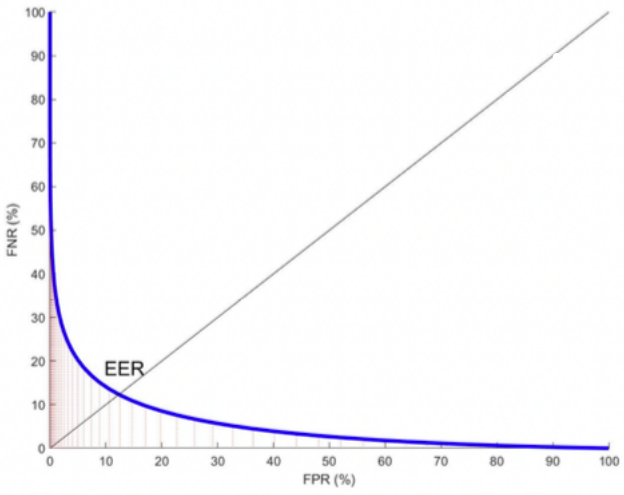


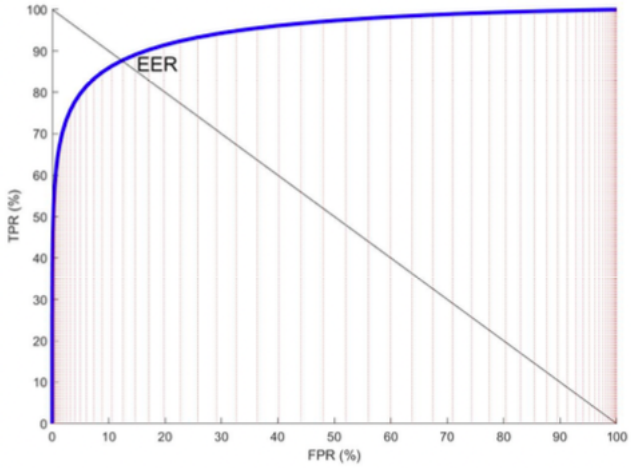


ROC Curve:

When the **(1-TPR)** curve is plotted as the *y-axis*, and the **(1-TNR)** curve as the *x-axis*, the plot is called the **DET** curve (*Detection Error Trade-off curve*). The lower the **DET** (bending to the *bottom-left corner*), the better the classification accuracy.

When the **TPR** -curve is plotted as the *y-axis*, and the **(1-TNR)** curve is planned as the *x-axis*, the plot is called the **ROC** curve (*Receiver Operating Characteristic curve*). The higher the **ROC** (bending to the *top-left corner*), the better the classification accuracy.





Area Under the Curve (ROC curve):

AUC provides an ***aggregate measure*** of performance across all possible classification thresholds. It can be seen as the probability that the model *rank*s a random positive example more highly than a random negative example.

AUC ranges in value from 0 (100% wrong prediction) to 1 (100% correct prediction). It has the following properties:

- ***Scale-invariant***: changing the range of the output will not change the ***AUC*** score;
- ***Classification-threshold-invariant***: Given any threshold, the ***AUC*** score will not be changed.

Consider a set of binary samples indexed by $i = 1, ..., m+$ for positive class; $j = 1, ..., m-$ and for negative class. (Close relationship to ***AUC*** and ***Wilcoxon Rank Test***)

Let $g(\boldsymbol{x})$ be a predictor, and $e_{ij} = g(\boldsymbol{x}_i+) - g(\boldsymbol{x}_j-)$ is the difference in the i, j value of the predictor between a positive-class sample $\boldsymbol{x}_i+$ and a negative-class sample $\boldsymbol{x}_j-$. Use indicator step function $I_{\{>0\}}$ to compute ***AUC***, where $I_{\{>0\}}(0) = 0.5$ (modified).

The Area Under the ROC Curve (AUC) can be expressed as

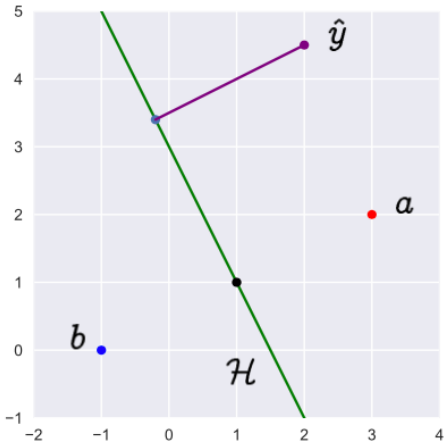
$$\mathrm{AUC} = \frac{1}{m^+ m^-} \sum_{i=1}^{m^+} \sum_{j=1}^{m-} I_{\{>0\}}(e_{ij})$$

For *Multiclass Classification*:

Loss Functions Choice:

The ***signed distance*** $\hat{y}$ of to H is

$$D(\hat{y}, a, b) = \frac{2(b-a)^{\mathsf{T}}\hat{y} - \|a\|_2^2 + \|b\|_2^2}{2\|b-a\|_2}$$



- $D(\hat{y}, a, b) < 0$ when $\hat{y}$ is closer to a than b
- $|D(\hat{y}, a, b)|$ is the distance of $\hat{y}$ to $\mathcal{H}$
- $D(\hat{y}, a, b) = 0$ gives the decision boundary
- $D(\hat{y}, a, b)$ is an affine function of $\hat{y}$
- we need to give the K functions of $\hat{y}$

$$\ell(\hat{y}, \psi_i), \quad i = 1, \ldots, K$$

- $\ell(\hat{y}, \psi_i)$ is how much we dislike predicting $\hat{y}$ when $y = \psi_i$
- loss function $\ell(\hat{y}, \psi_i)$ should be
  - small when $\max_{j \neq i} D_{ij}(\hat{y}) < 0$
  - larger when $\max_{j \neq i} D_{ij}(\hat{y}) \not< 0$

Neyman-Pearson loss:

- Neyman-Pearson loss is

$$\ell(\hat{y}, \psi_i) = \begin{cases} 0 & \max_{j \neq i} D_{ij} < 0 \\ \kappa_i & \text{otherwise} \end{cases}$$

i.e., zero when ψ_i is decoded correctly, κ_i otherwise

Multi-class hinge loss:

- *hinge loss* is

$$\ell(\hat{y}, \psi_i) = \kappa_i \max_{j \neq i} (1 + D_{ij}(\hat{y}))_+$$

Multi-class logistic loss:

- *logistic loss* is

$$\ell(\hat{y}, \psi_i) = \kappa_i \log \left(\sum_{j=1}^{K} \exp(D_{ij}(\hat{y})) \right)$$

(where we take $D_{jj} = 0$)

Confusion Matrix for Multicategory Classification

	$P_{\hat{1}}$ (predicted)	$P_{\hat{2}}$ (predicted)		$P_{\hat{C}}$ (predicted)
P_1 (actual)	$P_{1,\hat{1}}$	$P_{1,\hat{2}}$	⋯	$P_{1,\hat{C}}$
P_2 (actual)	$P_{2,\hat{1}}$	$P_{2,\hat{2}}$	⋯	$P_{2,\hat{C}}$
⋮	⋮	⋮	⋱	⋮
P_C (actual)	$P_{C,\hat{1}}$	$P_{C,\hat{2}}$		$P_{C,\hat{C}}$

Lecture X Unsupervised Learning

1) Motivations and Introduction:

Motivations of Unsupervised Learning:

Humans can partition unlabeled data into some groups (i.e., clustering) such that some functional data structures can be found.

Several reasons make it impossible to obtain labels:

- Labeling is ***expensive***: In some tasks, the labeling should be conducted by experts, such as medical imaging.
- Labeling is ***time-consuming***: supervised learning of deep neural networks requires large-scale labeled databases.

Introduction to Unsupervised Learning:

In unsupervised learning, the dataset is a collection of ***unlabeled examples***, $\{x_i\}_{i=1}^N$.

Again, $\boldsymbol{x}$ is a feature vector, and the goal of an unsupervised learning algorithm is to create a model that takes a feature vector $\boldsymbol{x}$ as input and either *transforms it* into another vector or into a value that can be used to *solve a practical problem* (anomaly detection, data compression, discovery of new species).

The *absence of labels* means the absence of a solid reference point to judge the quality of your model. The evaluation metrics should be defined based on your own task (e.g., clustering, reduction of dimension).

2) Disparate Types of Unsupervised Learning

Clustering:

Clustering is a problem of learning to assign labels to examples by ***leveraging** an unlabeled dataset*.

k-means data model:

Data Model: $\boldsymbol{x}$ is close to one of the k representatives $\theta_1, \cdots, \theta_k \in R^d$.

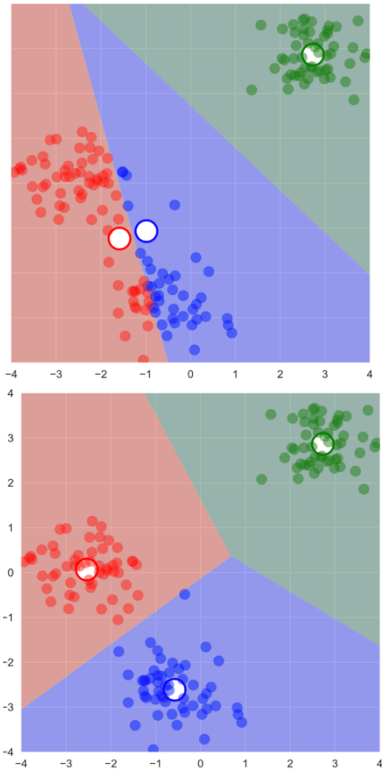
Quantitatively: for our data points x, the quantity

$$\ell_\theta(x) = \min_{i=1,\ldots,k} ||x - \theta_i||_2^2$$

i.e., the ***minimum distance*** squared to the representatives, is small.

$d \times k$ matrix $\Theta = [\theta_1 \cdots \theta_k]$. parametrizes the k-means data model.

k-means algorithm:



- define the *assignment* or *clustering* vector $c \in \mathbf{R}^n$
- c_i is the cluster that data vector x^i is in (so $c_i \in \{1, \ldots, k\}$)
- to minimize

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \min_{j=1,\ldots,k} ||x^i - \theta_j||^2$$

  we minimize $\frac{1}{n} \sum_{i=1}^{n} ||x^i - \theta_{c_i}||^2$ over both c and $\theta_1, \ldots, \theta_k$
- we can minimize over c using $c_i = \operatorname{argmin}_j ||x^i - \theta_j||^2$
- we can minimize over $\theta_1, \ldots, \theta_k$ using θ_i as the average of $\{x^j \mid c_j = i\}$
- k-means algorithm alternates between these two steps
- it is a heuristic for (approximately) minimizing $\mathcal{L}(\theta)$

Dimensionality Reduction:

Principal component analysis (**PCA**): a technique that converts a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables, called principal components.

Density Estimation:

In existing machine learning tasks, we always assume that the training set D is sampled from a distribution $P(\mathrm{X})$. an important task is to estimate the distribution based on some observed data, called ***density estimation***.

The most typical density estimation approach is ***kernel density estimation*** (**KDE**)

We adopt the following kernel model:

$$\hat{f}_b(x) = \frac{1}{Nb} \sum_{i=1}^{N} k\left(\frac{x - x_i}{b}\right), \quad k(z) = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-z^2}{2}\right),$$

where $b > 0$ is a hyper-parameter called ***kernel size***, which can be tuned using K-fold cross-validation.

Autoencoder:

An ***autoencoder*** is a type of artificial neural network used to learn efficient codings of unlabeled data (unsupervised learning).

The following components define an autoencoder:

- Two sets: the space of decoded messages $\mathcal{X}$; the space of encoded messages $\mathcal{Z}$. Almost always, both are Euclidean spaces.
- Two parametrized families of functions: the encoder family $E_\phi: \mathcal{X} \to \mathcal{Z}$, parametrized by ϕ; the decoder family $D_\theta: \mathcal{Z} \to \mathcal{X}$, parametrized by θ.

A task is defined by a reference probability distribution μ_{ref} over $\mathcal{X}$, and a "reconstruction quality" function as a metric on $\mathcal{X}$. With those, we can define the loss function for the autoencoder as $L(\theta, \phi) := \mathbb{E}_{x \sim \mu_{ref}}[d(x, D_\theta(E_\phi(x)))]$.

Self-supervised Learning:

Contrastive self-supervised learning techniques are promising methods that *build representations by learning to* encode what makes two things similar or different.

Advantages of ***self-supervised learning*** (**SSL**):

- *Pre-train large-scale deep neural networks*: in recent 3 years, there is a trend of pre-training very large-scale deep neural networks, which can provide good feature representation for different downstream tasks. Self-supervised learning is a “perfect” approach for the large-scale pre-training task, since it requires very large-scale data, and the pre-trained model should be generalized to different downstream tasks.
- Sometimes, there are noisy labels or malicious labels in training set. The learned model by SSL will not be influenced by such labels. One interesting example is that SSL has been successfully applied to *defend the backdoor attacks* (an emerging topic about AI security).

Lecture XI K-Means Clustering

1) Basic K-Means Clustering:

Definitions:

- A method of ***vector quantization***, originally from signal processing, aims to partition *n* observations/samples into *k* clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.
- Generically, *K*-means clustering minimizes within-cluster variances (squared Euclidean distances).

Optimization Perspective:

Given the data set $\{x_i\}_{i=1}^N$, *K*-means aims to find cluster centers $\boldsymbol{c} = \{c_i\}_{i=1}^N$, and assignments ***r,*** by minimizing the sum of squared distances of data points to their assigned cluster centers. In short, *K*-means will ***minimize the within-cluster variance***, as follows:

$$\min_{\mathbf{c},\mathbf{r}} J(\mathbf{c},\mathbf{r}) = \min_{\mathbf{c},\mathbf{r}} \sum_{i}^{n} \sum_{k}^{K} r_{ik}(\mathbf{x}_i - \mathbf{c}_k)^2,$$

$$\text{Subject to } \ \mathbf{r} \in \{0,1\}^{n\times K}, \ \sum_{k}^{K} r_{ik} = 1,$$

The above problem can be solved by ***coordinate descent algorithm***, i.e., update ***c*** and ***r*** alternatively:

Optimization of K-means clustering:

- Initialization: set K cluster centers $\mathbf{c}$ to random values
- Repeat until convergence (the assignments don't change):
  - Assignment: Given the cluster centers $\mathbf{c}$, update the assignments $\mathbf{r}$ by solving the following sub-problem

$$\min_{\mathbf{r}} \sum_{i}^{n} \sum_{k}^{K} r_{ik}(\mathbf{x}_i - \mathbf{c}_k)^2, \ \text{ subject to } \ \mathbf{r} \in \{0,1\}^{n\times K}, \ \sum_{k}^{K} r_{ik} = 1.$$

Note that the assignment for each data $\mathbf{x}_i$ can be solved independently. It is easy to know that assign $\mathbf{x}_i$ to the closest cluster is the optimal solution.

- Refitting: Given the assignments $\mathbf{r}$, update the cluster centers $\mathbf{c}$:

$$\min_{\mathbf{c}} \sum_{i}^{n} \sum_{k}^{K} r_{ik} (\mathbf{x}_i - \mathbf{c}_k)^2.$$

Note that $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_K$ can be optimized independently. By setting the derivative *w.r.t.* $\mathbf{c}_k$ as 0, it is easy to obtain the optimal solution:

$$\mathbf{c}_k = \frac{\sum_i^n r_{ik} \mathbf{x}_i}{\sum_i^n r_{ik}}.$$

Convergence results:

Since the objective function, *J* is *non-convex*, the coordinate descent on *J* is not guaranteed to converge to the global minimum.

We could run *K*-means with *multiple random initializations* and pick the one with the lowest objective value as the final clustering result.

***Example*: K-means for vector quantization:**

Vector quantization is a classical quantization technique from signal processing used for compressing images.

It works by dividing a large set of points (vectors) into groups having approximately the same number of points closest to them. Each group is represented by its centroid point, as in *K*-means.

2) Soft Clustering

Hard vs Soft Clustering

Hard Clustering:

Each data point can belong to *only one cluster*.

Soft Clustering (also known as **Fuzzy Clustering**):

Each data point can belong to *more than one cluster*. (Disparate percentages belong to multifarious clusters.)

Fuzzy c-means clustering has an objective function formulated as follows:

$$\min_{\mathbf{c},\mathbf{r}} J(\mathbf{c}, \mathbf{r}) = \min_{\mathbf{c},\mathbf{r}} \sum_{i}^{n} \sum_{k}^{K} (r_{ik})^m (\mathbf{x}_i - \mathbf{c}_k)^2,$$

$$\text{Subject to } \mathbf{r} \in [0,1]^{n \times K}, \ \sum_{k}^{K} r_{ik} = 1,$$

The hyper-parameter $m > 1$ is called ***fuzzifier***, and it defines the *level of cluster fuzziness*.

A value of m close to 1 gives a cluster solution that becomes increasingly similar to the solution of hard clustering, such as K-means; whereas a value of m close to infinite leads to *complete fuzziness*.

Under KKT conditions, with $\boldsymbol{c}$ fixed, $\boldsymbol{r}$ can be updated as

$$r_{ik} = \frac{1}{\sum_j^K \left(\frac{1}{d_{ij}^2}\right)^{\frac{1}{m-1}}} \cdot \left(\frac{1}{d_{ik}^2}\right)^{\frac{1}{m-1}} = \frac{1}{\sum_j^K \left(\frac{d_{ik}^2}{d_{ij}^2}\right)^{\frac{1}{m-1}}}. \quad \text{where } d_{ik}^2 = (\mathbf{x}_i - \mathbf{c}_k)^2.$$

Given $\boldsymbol{r}$, $\boldsymbol{c}$ can be updated as

$$\frac{\partial J(\mathbf{c})}{\partial \mathbf{c}_k} = 0 \;\Rightarrow\; \mathbf{c}_k = \frac{\sum_i^n [(r_{ik})^m \mathbf{x}_i]}{\sum_i^n (r_{ik})^m}.$$

3) Variants of K-means Clustering

Constrained K-Means Clustering

In practice, we may know some *additional evidences or preferences* about some data points we want to do clustering, such as:

- ***Must-link constraints***: two points must be partitioned to the same cluster.
- ***Cannot-link constraints***: two points cannot be partitioned to the same cluster.

K-means with must-link/cannot-link constraints is called **constrained *K*-means or semi-supervised *K*-means**.

COP-KMEANS(data set D, must-link constraints $Con_= \subseteq D \times D$, cannot-link constraints $Con_{\neq} \subseteq D \times D$)

1. Let $C_1 \ldots C_k$ be the initial cluster centers.
2. For each point d_i in D, assign it to the closest cluster C_j **such that** VIOLATE-CONSTRAINTS(d_i, C_j, $Con_=$, $Con_{\neq}$) **is false. If no such cluster exists, fail (return {}).**
3. For each cluster C_i, update its center by averaging all of the points d_j that have been assigned to it.
4. Iterate between (2) and (3) until convergence.
5. Return $\{C_1 \ldots C_k\}$.

VIOLATE-CONSTRAINTS(data point d, cluster C, must-link constraints $Con_= \subseteq D \times D$, cannot-link constraints $Con_{\neq} \subseteq D \times D$)

1. For each $(d, d_=) \in Con_=$: If $d_= \notin C$, return true.
2. For each $(d, d_{\neq}) \in Con_{\neq}$: If $d_{\neq} \in C$, return true.
3. Otherwise, return false.

Accelerated K-means Clustering

The computational cost of basic K-means algorithm:

The total cost is $O(T \times (n \times (K+1) \times d))$, with T being the number of iterations, n being the number of points, K being the number of clusters, and d being the feature dimension.

Reduction of Costs:

1. Let $\boldsymbol{x}$ be any data point, and c be the center to which $\boldsymbol{x}$ is currently

assigned, and let $\boldsymbol{c}'$ be any other center $d(\boldsymbol{c}, \boldsymbol{c}') \geq 2d(\boldsymbol{x}, \boldsymbol{c})$, then $d(\boldsymbol{x}, \boldsymbol{c}') \geq d(\boldsymbol{x}, \boldsymbol{c})$, unnecessary to compute $d(\boldsymbol{x}, \boldsymbol{c}')$, leading to the cost reduction.

2. Let $\boldsymbol{x}$ be any data point, $\boldsymbol{b}^{t+1}$ be any center at the $t+1$ iteration, and $\boldsymbol{b}^t$ be the previous version of the same center. Lower bound: Suppose that in the previous iteration t, we knew a lower bound $l(\mathrm{x}, \boldsymbol{b}^t)$ such that $d(\boldsymbol{x}, \boldsymbol{b}^t) \geq l(\boldsymbol{x}, \boldsymbol{b}^t)$, then we can update a new lower bound $l(\boldsymbol{x}, \boldsymbol{b}^{t+1})$ for the current iteration $t+1$.
3. Upper bound: Suppose $u(\boldsymbol{x}) \geq d(\boldsymbol{x}, \boldsymbol{c}^t)$ is an upper bound on the distance between $\boldsymbol{x}$ and its currently assigned centroid $\boldsymbol{c}^t$ (the centroid at iteration t). And, suppose $l(\boldsymbol{x}, (\boldsymbol{c}')^t) \leq d(\boldsymbol{x}, (\boldsymbol{c}')^t)$ is a lower bound on the distance between $\boldsymbol{x}$ and some other center $(\boldsymbol{c}')^t$. If $u(\boldsymbol{x}) \leq l(\boldsymbol{x}, (\boldsymbol{c}')^t)$, then $d(\boldsymbol{x}, \boldsymbol{c}^t) \leq u(\boldsymbol{x}) \leq l(\boldsymbol{x}, (\boldsymbol{c}')^t)) \leq d(\boldsymbol{x}, (\boldsymbol{c}')^t)$. Thus, it is necessary to calculate neither $d(\boldsymbol{x}, \boldsymbol{c}^t)$ nor $d(\boldsymbol{x}, (\boldsymbol{c}')^t)$, leading to <u>*cost reduction*</u>. Note that it will never be necessary to calculate $d(\boldsymbol{x}, (\boldsymbol{c}')^t)$ in the current iteration, but it may be necessary to calculate $d(\boldsymbol{x}, \boldsymbol{c}^t)$ as $u(\boldsymbol{x}) \leq l(\boldsymbol{x}, (\boldsymbol{c}')^t)$ may not true for some other center $\boldsymbol{c}''$. Update the upper bound $u(\boldsymbol{x})$: $u(\boldsymbol{x}) = u(\boldsymbol{x}) + d(\boldsymbol{c}^t, \boldsymbol{c}^{t+1})$.

Putting the observations above together, the accelerated k-means algorithm is as follows.

First, pick initial centers. Set the lower bound $l(x, c) = 0$ for each point x and center c. Assign each x to its closest initial center $c(x) = \text{argmin}_c\, d(x, c)$, using Lemma 1 to avoid redundant distance calculations. Each time $d(x, c)$ is computed, set $l(x, c) = d(x, c)$. Assign upper bounds $u(x) = \min_c d(x, c)$.

Next, repeat until convergence:

1. For all centers c and c', compute $d(c, c')$. For all centers c, compute $s(c) = \frac{1}{2} \min_{c' \neq c} d(c, c')$.
2. Identify all points x such that $u(x) \leq s(c(x))$.
3. For all remaining points x and centers c such that
   (i) $c \neq c(x)$ and
   (ii) $u(x) > l(x, c)$ and
   (iii) $u(x) > \frac{1}{2} d(c(x), c)$:
   3a. If $r(x)$ then compute $d(x, c(x))$ and assign $r(x) =$ false. Otherwise, $d(x, c(x)) = u(x)$.
   3b. If $d(x, c(x)) > l(x, c)$ or $d(x, c(x)) > \frac{1}{2} d(c(x), c)$ then
   Compute $d(x, c)$
   If $d(x, c) < d(x, c(x))$ then assign $c(x) = c$.
4. For each center c, let $m(c)$ be the mean of the points assigned to c.
5. For each point x and center c, assign
   $$l(x, c) = \max\{l(x, c) - d(c, m(c)), 0\}.$$
6. For each point x, assign
   $$u(x) = u(x) + d(m(c(x)), c(x))$$
   $r(x) =$ true.
7. Replace each center c by $m(c)$.

4) Performance Evaluation

There are two types of evaluation metrics for clustering:

- ***Internal evaluation metrics***: Silhouette coefficient.

- ***External evaluation metrics***: These metrics require the knowledge of the ground truth classes, while rarely available in practice or manual assignment by human annotators (as in the supervised learning setting).

Silhouette Coefficient

Given a clustering, we define

a: The mean distance between a point and all other points in the same cluster.

b: The mean distance between a point and all other points in the next nearest cluster.

Silhouette coefficient s for a single sample is formulated as:

$$s = \frac{b-a}{\max(a,b)} \Rightarrow s = \begin{cases} 1 - \frac{a}{b} & \text{if } a < b \\ 0 & \text{if } a = b \\ \frac{b}{a} - 1 & \text{if } a > b \end{cases}$$

It is easy to know that $s \in (-1,1)$, and a larger s value indicates better clustering performance.

Rand Index (RI):

- Given a set of n samples $S = \{o_1, o_2, \ldots, o_n\}$, there are two clusterings/partitions of S to compare, including:
  - $X = \{X_1, X_2, \ldots, X_r\}$ with r clusters
  - $Y = \{Y_1, Y_2, \ldots, Y_s\}$ with s clusters
- We can calculate the following values:
  - a: The number of pairs of elements in S that are in the **same** subset in X and in the **same** subset in Y
  - b: The number of pairs of elements in S that are in the **different** subset in X and in the **different** subset in Y
  - c: The number of pairs of elements in S that are in the **same** subset in X and in the **different** subset in Y
  - d: The number of pairs of elements in S that are in the **different** subset in X and in the **same** subset in Y
- The rand index (RI) can be computed as follows:

$$\text{RI} = \frac{a+b}{a+b+c+d} = \frac{a+b}{\frac{n(n-1)}{2}}$$

Note that $\text{RI} \in [0,1]$, and higher score corresponds higher similarity.

Lecture XII Gaussian Mixture Models

1) Gaussian Mixture Model for Density Estimation:

Mixture Models:

- In statistics, a **mixture model** is a probabilistic model for representing the presence of subpopulations within an overall population, without requiring that an observed data set should identify the sub-population to which an individual observation belongs.
- A typical finite-dimensional mixture model is a ***hierarchical model*** consisting of the following components:
    - N random variables that are observed, each distributed according to a mixture of K components, with the components belonging to the same parametric family of distributions but with different parameters.
    - N random ***latent variables*** specifying the identity of the mixture component of each observation, each distributed according to a K-dimensional categorical distribution.
    - A set of K ***mixture weights***, which are probabilities that sum to 1.
    - A set of K parameters, each specifying the parameter of the corresponding mixture component. In many cases, each "parameter" is actually a set of parameters. For example, if the mixture components are Gaussian distributions, there will be a mean and variance for each component. If the mixture components are categorical distributions (e.g., when each observation is a token from a finite alphabet of size V), there will be a vector of V probabilities summing to 1.

Gaussian Mixture Model (GMM):

A GMM is composed of normal families as its components. GMM is a ***density estimator***. If given enough Gaussian components, GMM is a universal approximator of densities.

$$p(\boldsymbol{x}) = \sum_{i=1}^{K} \phi_i \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \Sigma_k)$$

K	=	number of mixture components
N	=	number of observations
$\theta_{i=1\ldots K}$	=	parameter of distribution of observation associated with component i
$\phi_{i=1\ldots K}$	=	mixture weight, i.e., prior probability of a particular component i
$\boldsymbol{\phi}$	=	K-dimensional vector composed of all the individual $\phi_{1\ldots K}$; must sum to 1
$z_{i=1\ldots N}$	=	component of observation i
$x_{i=1\ldots N}$	=	observation i
$F(x\|\theta)$	=	probability distribution of an observation, parametrized on θ
$z_{i=1\ldots N}$	$\sim$	Categorical($\boldsymbol{\phi}$)
$x_{i=1\ldots N}\|z_{i=1\ldots N}$	$\sim$	$F(\theta_{z_i})$

Maximize the log-likelihood function to find the parameter.

We introduce a hidden (latent) variable ***z***, indicating which Gaussian component generates the observation ***x***, with some probability. ($z \sim$ Categorical(ϕ)).

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(\mathbf{x}, z = k) = \sum_{k=1}^{K} \underbrace{p(z = k)}_{\pi_k} \underbrace{p(\mathbf{x} \mid z = k)}_{\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}$$

Latent Variable Model (LVM):

Variables that are always unobserved are called ***latent variables***, or sometimes ***hidden variables***. A latent variable model is a statistical model that relates a set of observable variables to a set of latent variables.

According to the type of latent variables, there are two types of LVMs,

1. LVM with continuous latent variables, e.g., factor analysis
2. LVM with discrete latent variables, e.g., mixture models

Then, the log-likelihood becomes:

$$\ln L(X; \boldsymbol{\phi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \sum_{j=1}^{K} P(z^{(n)}|\boldsymbol{\phi}) P(\boldsymbol{x}^{(n)}|z^{(n)}, \boldsymbol{\mu}_k, \Sigma_k)$$

with the constraint $1 - \sum_{i=1}^{K} \phi_i = 0$.

KKT conditions:

$$\frac{\partial \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \lambda)}{\partial \boldsymbol{\mu}_k} = \frac{-\partial \sum_{n=1}^{N} 1_{[z^{(n)}=k]} \ln p(\mathbf{x}^{(n)}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\partial \boldsymbol{\mu}_k} = \mathbf{0},$$

$$\frac{\partial \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \lambda)}{\partial \boldsymbol{\Sigma}_k} = \frac{-\partial \sum_{n=1}^{N} 1_{[z^{(n)}=k]} \ln p(\mathbf{x}^{(n)}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\partial \boldsymbol{\Sigma}_k} = \mathbf{0},$$

$$\frac{\partial \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \lambda)}{\partial \phi_k} = \frac{\partial \sum_{n=1}^{N} 1_{[z^{(n)}=k]} \ln \phi_k}{\partial \phi_k} - \lambda = 0,$$

$$1 - \sum_{k=1}^{K} \phi_k = 0.$$

If we know $z^{(n)}$ for every $\boldsymbol{x}^{(n)}$, the maximum log-likelihood problem is easy. The solution can be directly derived from the above KKT conditions:

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} 1_{\left[z^{(n)}=k\right]} \mathbf{x}^{(n)}}{\sum_{n=1}^{N} 1_{\left[z^{(n)}=k\right]}}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_{n=1}^{N} 1_{\left[z^{(n)}=k\right]} \left(\mathbf{x}^{(n)} - \boldsymbol{\mu}_k\right)\left(\mathbf{x}^{(n)} - \boldsymbol{\mu}_k\right)^T}{\sum_{n=1}^{N} 1_{\left[z^{(n)}=k\right]}}$$

$$\phi_k = \frac{1}{N} \sum_{n=1}^{N} 1_{\left[z^{(n)}=k\right]}$$

2) Expectation Maximization for Fitting GMM:

When we don't know $z^{(n)}$ for every $\boldsymbol{x}^{(n)}$, we use the expectation-maximization algorithm**.**

EM Steps:

- ***Expectation Step***: Conditional probability (using Bayes rule) of z given $\boldsymbol{x}$

$$\begin{aligned}\gamma_k = p(z=k \mid \mathbf{x}) &= \frac{p(z=k)p(\mathbf{x} \mid z=k)}{p(\mathbf{x})} \\ &= \frac{p(z=k)p(\mathbf{x} \mid z=k)}{\sum_{j=1}^{K} p(z=j)p(\mathbf{x} \mid z=j)} \\ &= \frac{\phi_k \mathcal{N}\left(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right)}{\sum_{j=1}^{K} \phi_j \mathcal{N}\left(\mathbf{x} \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j\right)}.\end{aligned}$$

γ_k can be viewed as the ***responsibility*** of cluster k towards $\boldsymbol{x}$.

- ***Maximization Step***: Given the posterior probability, update the model parameters, by maximizing the ***expected log-likelihood***.

$$\max_{\Theta} \sum_{n}^{N} \sum_{k}^{K} \gamma_k^{(n)} \ln\left(\phi_k\right) + \sum_{n}^{N} \sum_{k}^{K} \gamma_k^{(n)} \ln\left(\mathcal{N}(\mathbf{x}^{(n)} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right), \text{ s.t. } \sum_{k}^{K} \phi_k = 1.$$

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_k^{(n)} \mathbf{x}^{(n)}$$

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_k^{(n)} \left(\mathbf{x}^{(n)} - \boldsymbol{\mu}_k\right)\left(\mathbf{x}^{(n)} - \boldsymbol{\mu}_k\right)^{\top}$$

$$\phi_k = \frac{N_k}{N}, \text{ with } N_k = \sum_{n=1}^{N} \gamma_k^{(n)}$$

Relation of EM to K-Means:

If fixing the covariance matrices Σ as the identity matrix I for all Gaussian components, EM for GMMs is reduced to a soft version of K-means.

Instead of hard assignments in the E-step, EM does soft assignments based on the softmax of the squared Euclidean distance from each point to each cluster.

Every center is moved by weighted means of the data, with weights given by soft assignments

GMM provides a probabilistic view of clustering - every cluster corresponds to a different Gaussian component.

Lecture XIII Expectation Maximization

1) Expectation-Maximization for Latent Variable Models:

Statistical Perspective of EM

Let $\boldsymbol{X}$ be the notation of observed data and $\boldsymbol{Z}$ denote the unobserved data. Let $g(\boldsymbol{x}|\theta)$ denote the joint pdf of $\boldsymbol{X}$. Let $h(\boldsymbol{x}, z|\theta)$ denote the joint pdf of the observed and the unobserved items. Let $k(z|\theta, \boldsymbol{x})$ denote the conditional pdf of the missing data given the observed data. By the definition of a conditional pdf, we have the identity:

$$k(\mathbf{z}|\theta, \mathbf{x}) = \frac{h(\mathbf{x}, \mathbf{z}|\theta)}{g(\mathbf{x}|\theta)}.$$

The observed likelihood function is $L(\theta|\boldsymbol{X}) = g(\boldsymbol{X}|\theta)$. The complete likelihood function is defined by $L^c(\theta|\mathbf{x}, \mathbf{z}) = h(\mathbf{x}, \mathbf{z}|\theta)$.

Our goal is to maximize the likelihood function $L(\theta|\boldsymbol{X})$ by using the complete likelihood $L^c(\theta|\boldsymbol{X}, \boldsymbol{Z})$ in this process. Then, for any arbitrary but fixed θ_0:

$$\begin{aligned}
\log L(\theta|\mathbf{x}) &= \int \log L(\theta|\mathbf{x}) k(\mathbf{z}|\theta_0, \mathbf{x})\, d\mathbf{z} \\
&= \int \log g(\mathbf{x}|\theta) k(\mathbf{z}|\theta_0, \mathbf{x})\, d\mathbf{z} \\
&= \int [\log h(\mathbf{x}, \mathbf{z}|\theta) - \log k(\mathbf{z}|\theta, \mathbf{x})] k(\mathbf{z}|\theta_0, \mathbf{x})\, d\mathbf{z} \\
&= \int [\log[h(\mathbf{x}, \mathbf{z}|\theta)] k(\mathbf{z}|\theta_0, \mathbf{x})\, d\mathbf{z} - \int [\log[k(\mathbf{z}|\theta, \mathbf{x})] k(\mathbf{z}|\theta_0, \mathbf{x})\, d\mathbf{z} \\
&= E_{\theta_0}[\log L^c(\theta|\mathbf{x}, \mathbf{Z})|\theta_0, \mathbf{x}] - E_{\theta_0}[\log k(\mathbf{Z}|\theta, \mathbf{x})|\theta_0, \mathbf{x}], \qquad (6.6.3)
\end{aligned}$$

Define the first term on the right side of (6.6.3) to be the function

$$Q(\theta|\theta_0, \mathbf{x}) = E_{\theta_0}[\log L^c(\theta|\mathbf{x}, \mathbf{Z})|\theta_0, \mathbf{x}].$$

The expectation which defines the function Q is called the E-Step of the EM algorithm. Recall that we want to maximize $\log L(\theta|\boldsymbol{x})$. As discussed below, we need only maximize Q. This maximization is called the M-Step of the EM algorithm.

Let $H(\theta|\theta_0, \boldsymbol{x}) = -\mathbb{E}_{\theta_0}[\log k(\boldsymbol{Z}|\theta, \boldsymbol{x})|\theta_0, \boldsymbol{x}]$.

Then, due to ***Gibbs' inequality*** $H(\theta|\hat{\theta}^{(m)}, \boldsymbol{x}) \geq H(\hat{\theta}^{(m)}|\hat{\theta}^{(m)}, \boldsymbol{x})$

We finally get $\log L(\theta|\boldsymbol{x}) - \log L(\hat{\theta}^{(m)}|\boldsymbol{x}) \geq Q(\theta|\hat{\theta}^{(m)}, \boldsymbol{x}) - Q(\hat{\theta}^{(m)}|\hat{\theta}^{(m)}, \boldsymbol{x})$

Thus, choosing to improve $Q(\theta|\theta_0, x)$ causes to improve $log\ L\ (\theta|x)$ at least as much.

Algorithm 6.6.1 (EM Algorithm). *Let $\widehat{\theta}^{(m)}$ denote the estimate on the mth step. To compute the estimate on the $(m+1)$st step do:*

1. *Expectation Step: Compute*

$$Q(\theta|\widehat{\theta}^{(m)}, \mathbf{x}) = E_{\widehat{\theta}^{(m)}}[\log L^c(\theta|\mathbf{x}, \mathbf{Z})|\widehat{\theta}_m, \mathbf{x}], \tag{6.6.5}$$

   where the expectation is taken under the conditional pdf $k(\mathbf{z}|\widehat{\theta}^{(m)}, \mathbf{x})$.

2. *Maximization Step: Let*

$$\widehat{\theta}^{(m+1)} = Argmax Q(\theta|\widehat{\theta}^{(m)}, \mathbf{x}). \tag{6.6.6}$$

EM for Latent Variables (as a max–max procedure)

Particularly, for latent variable cases, i.e., z follows a categorical distribution (i.e., in mixture modes), and joint pdf $g(\boldsymbol{x}|\theta) = \prod_{i=1}^{N} p_X(x^{(i)}|\theta)$.

We first introduce a new distribution w.r.t. each latent variable $z^{(n)}$, denoted as $q_n(z^{(n)})$.

$$\begin{aligned}\ln p(\mathbf{x};\boldsymbol{\theta}) &= \mathbb{E}_{q(z)}\left[\ln\left(\frac{p(\mathbf{x};\boldsymbol{\theta})\cdot q(z)}{q(z)}\right)\right] = \mathbb{E}_{q(z)}\left[\ln\left(\frac{p(\mathbf{x},z;\boldsymbol{\theta})}{q(z)}\cdot\frac{q(z)}{p(z|\mathbf{x};\boldsymbol{\theta})}\right)\right] \\ &= \mathbb{E}_{q(z)}\left[\ln\left(\frac{p(\mathbf{x},z;\boldsymbol{\theta})}{q(z)}\right)\right] + \mathbb{E}_{q(z)}\left[\ln\left(\frac{q(z)}{p(z|\mathbf{x};\boldsymbol{\theta})}\right)\right]\end{aligned}$$

It is natural to extend the above decomposition to the log-likelihood of the whole data set D, i.e.,

$$\begin{aligned}\ln p(\mathcal{D};\boldsymbol{\theta}) =& \sum_{n=1}^{N}\mathbb{E}_{q_n(z^{(n)})}\left[\ln\left(\frac{p(\mathbf{x}^{(n)},z^{(n)};\boldsymbol{\theta})}{q_n(z^{(n)})}\right)\right] \\ &+ \sum_{n=1}^{N}\mathbb{E}_{q_n(z^{(n)})}\left[\ln\left(\frac{q_n(z^{(n)})}{p(z^{(n)}|\mathbf{x}^{(n)};\boldsymbol{\theta})}\right)\right] \\ =& \mathcal{L}(\mathbf{q};\boldsymbol{\theta}) + \mathrm{KL}\big(\mathbf{q}(\mathbf{z})||p(\mathbf{z}|\mathcal{D};\boldsymbol{\theta})\big)\end{aligned}$$

Since learning $\boldsymbol{\theta}$ by maximizing ln $\boldsymbol{p}(D; \boldsymbol{\theta})$ is difficult, we resort to maximizing its lower bound $L(\boldsymbol{q}; \boldsymbol{\theta})$ with some auxiliary distribution $\boldsymbol{q}(z)$.

Then, the EM steps become:

- If the current parameter value is $\boldsymbol{\theta}^{\text{old}}$:
  - **E-step**: Given $\boldsymbol{\theta}^{\text{old}}$, we update the auxiliary distribution $\mathbf{q}(\mathbf{z})$ to make the bound tight:

$$\mathbf{q}(\mathbf{z}) = \underset{\mathbf{q}(\mathbf{z})}{\operatorname{argmax}}\ \mathcal{L}(q, \boldsymbol{\theta}^{\text{old}}). \tag{2}$$

    It leads to $q_n\left(z^{(n)}\right) = p\left(z^{(n)} \mid \mathbf{x}^{(n)}; \boldsymbol{\theta}^{\text{old}}\right), \forall n$, and makes

$$\log p\left(\mathcal{D}; \boldsymbol{\theta}^{\text{old}}\right) = \mathcal{L}\left(q; \boldsymbol{\theta}^{\text{old}}\right)$$

  - **M-step**: Given $\mathbf{q}(\mathbf{z})$ updated above, we update $\boldsymbol{\theta}$ by optimizing the lower bound:

$$\begin{aligned}\boldsymbol{\theta}^{\text{new}} &= \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\ \mathcal{L}(q, \boldsymbol{\theta}) \\ &= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{n=1}^{N} \mathbb{E}_{q_n\left(z^{(n)}\right)}\left[\log \frac{p\left(z^{(n)}, \mathbf{x}^{(n)}; \boldsymbol{\theta}\right)}{q_n\left(z^{(n)}\right)}\right]\end{aligned}$$

EM for Gaussian Mixture Models

With the max-max procedures of EM (i.e., EM for latent variables), one can derive EM procedures for the Gaussian mixture models. The results are exhibited in the contents of Chapter 12.

Lecture XIV Principal Component Analysis

1) Dimensionality Reduction:

Dimensionality reduction aims to find a *low-dimensional data vector* representing the original high-dimensional data vector. It can be implemented by either an ***unsupervised learning*** method or a ***supervised learning*** method.

Principal Component Analysis is an unsupervised learning method to attain the reduction of the dimension.

There are several usages of dimensionality reduction, such as

- Visualization
- Alleviate overfitting
- Reduce the computational cost

Dimensionality reduction:

- *Input*s: given a dataset $D = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(N)}\}$ in $\boldsymbol{R}^D$, with D being the original dimension.
- *Goal*: find a K-dimensional ($K << D$) subspace S, which consists of K orthonormal basis $\{\boldsymbol{u}_k\}$. When projecting all points in D onto S, it is desired that the structure or property of the original data is well preserved.
- *Output*s: the basis vectors $\{\boldsymbol{u}_k\}$, and a new representation $D' = \{\boldsymbol{z}^{(1)}, ..., \boldsymbol{z}^{(N)}\} \dashrightarrow \boldsymbol{R}^K$

2) Derivation of PCA:

Maximal Variance

- Given a dataset $\mathcal{D} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\} \in \mathbb{R}^D$, we want to find a K-dimensional ($K < D$) subspace $\mathcal{S}$, which consists of K orthonormal basis vectors $\{\mathbf{u}_k\}_{k=1}^K$, such that the variance of the reconstructions $\tilde{\mathcal{D}} = \{\tilde{\mathbf{x}}^{(1)}, \ldots, \tilde{\mathbf{x}}^{(N)}\}$ is maximal, *i.e.*,

$$\max_{\mathbf{U}, \mathbf{U}^\top \mathbf{U} = \mathbf{I}} \frac{1}{N} \sum_{n=1}^{N} \|\tilde{\mathbf{x}}^{(n)} - \tilde{\boldsymbol{\mu}}\|^2, \tag{3}$$

where $\tilde{\boldsymbol{\mu}} = \frac{1}{N} \sum_{n=1}^{N} \tilde{\mathbf{x}}^{(n)}$ denotes the mean of the reconstructions.

With the property of projections, the original optimization questions can be rewritten as

$$\max_{\mathbf{U},\mathbf{U}^\top\mathbf{U}=\mathbf{I}} \frac{1}{N}\sum_{n=1}^{N} \|\mathbf{U}^\top(\mathbf{x}-\boldsymbol{\mu})\|^2$$

Minimal Reconstruction Errors

- Given a dataset $\mathcal{D}=\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\}\in\mathbb{R}^D$, we want to find a K-dimensional $(K<D)$ subspace $\mathcal{S}$, which consists of K orthonormal basis vectors $\{\mathbf{u}_k\}_{k=1}^K$, such that the reconstruction loss between $\mathbf{x}$ and $\tilde{\mathbf{x}}$ is minimized, *i.e.*,

$$\min_{\mathbf{U},\mathbf{U}^\top\mathbf{U}=\mathbf{I}} \frac{1}{N}\sum_{n=1}^{N} \|\mathbf{x}^{(n)}-\tilde{\mathbf{x}}^{(n)}\|^2. \tag{9}$$

Either to maximize the variance or to minimize the reconstruction errors give the same optimization question!

3) PCA Algorithm and Variant Methods:

PCA Algorithm

By deriving the stationary point of the Lagrangian function, one can get

- The Lagrangian function is formulated as follows:

$$L(\mathbf{U},\boldsymbol{\Lambda}_K)=\mathrm{Trace}\big(\mathbf{U}^\top\boldsymbol{\Sigma}\mathbf{U}\big)+\mathrm{Trace}\big(\boldsymbol{\Lambda}_K^\top(\mathbf{I}-\mathbf{U}^\top\mathbf{U})\big), \tag{14}$$

  where $\boldsymbol{\Lambda}_K=\mathrm{diag}([\hat{\lambda}_1,\ldots,\hat{\lambda}_K])\in\mathbb{R}^{K\times K}$.
- Then, its optimal solution should satisfy

$$\frac{\partial L(\mathbf{U},\boldsymbol{\Lambda}_K)}{\partial\mathbf{U}}=2\boldsymbol{\Sigma}\mathbf{U}-2\mathbf{U}\boldsymbol{\Lambda}_K=\mathbf{0} \tag{15}$$

$$\Rightarrow\ \boldsymbol{\Sigma}\mathbf{u}_k=\hat{\lambda}_k\mathbf{u}_k,\ k=1,\ldots,K. \tag{16}$$

Utilizing SVD decomposition, we have

$$\Sigma = \mathrm{Q}\Lambda_\mathrm{D}\mathrm{Q}^\mathrm{T} = \sum_{\mathrm{i}=1}^{\mathrm{D}} \lambda_\mathrm{i}\, \mathrm{q}_\mathrm{i}\mathrm{q}_\mathrm{i}^\mathrm{T}$$

Substitute into the objective function, we have that it is obvious that we should pick the top-K eigenvalues. Correspondingly, the first K columns of Q should be used as the optimal solution to U.

One eigenvector corresponds to one of the basis vectors of the subspace obtained by PCA.

One eigenvalue corresponds to the *variance of the projected points* on one basis vector (i.e., eigenvector). A larger eigenvalue indicates more information about the original data.

The above derivation of the optimal solution is summarized as the following steps:

- Step 1: Calculate the empirical covariance matrix $\mathbf{\Sigma} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}^{(n)} - \boldsymbol{\mu})(\mathbf{x}^{(n)} - \boldsymbol{\mu})^\top$
- Step 2: Do SVD decomposition of $\mathbf{\Sigma}$ to obtain its D eigenvalues $\{\lambda_i\}_{i=1}^{D}$ and eigenvectors $\{\mathbf{q}_i\}_{i=1}^{D}$, and rank them from large to small according to the eigenvalues.
- Step 3: Pick the top-K eigenvectors to form the matrix $\mathbf{U} = [\mathbf{q}_1, \ldots, \mathbf{q}_K] \in \mathbb{R}^{D\times K}$
- Step 4: The new representation of $\mathbf{x}^{(n)}$ is $\mathbf{U}^\top(\mathbf{x}^{(n)} - \boldsymbol{\mu})$.

The elements of the projection $\boldsymbol{z}$ are uncorrelated if those of $\boldsymbol{x}$ are uncorrelated.

Variant Methods of PCA

Note that PCA is an orthogonal linear transformation method, thus it *cannot handle non-linear data*. There are some interesting variants of PCA, such as

- Kernel PCA
- Probabilistic PCA
- Nonlinear PCA
- Robust PCA

Applications of PCA

PCA is widely used in image recognition, for instance, face figures, digit figures, etc.

Running PCA, we can obtain several ***eigenfaces*** in face recognition. With only top-3 eigenfaces, we achieve 79% accuracy on face/non-face discrimination on test data.