Introduction to Machine Learning

Based upon:

1. The slides of "Introduction to Machine Learning" course, CS faculty, Technion institute, spring semester 2022.
2. The book Understanding Machine Learning, by Shai Shalev-Shwartz and Shai Ben-David.
3. Multiple videos and papers online, links are mentioned.

Created by Nitai Kluger.

Notes:

1. This summary covers most of the course (as studied in the semester mentioned above) but not all of it.
2. I highly recommend using the links "for more information" along this summary. Those are usually animated videos and not tedious papers.

Table of Contents

Introduction

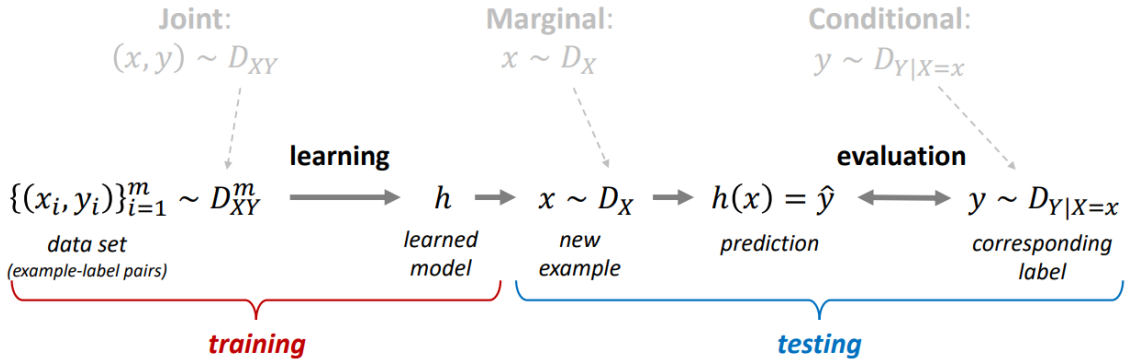
The 3 different probabilistic aspects of using data are:

Joint:
$(x, y) \sim D_{XY}$
(sample input-label pair)

Marginal:
$x \sim D_X$
(sample input)
data uncertainty

Conditional:
$y \sim D_{Y|X=x}$
(sample label for given input)
label uncertaint(ies)

The way we're using those 3 is:



Learning when examples and labels are observed is called supervised learning.

Notations:

- Features: $x \in \mathbb{R}^d$
- Labels: $y \in \{\pm 1\}$
- Data distribution: $(x, y) \sim^{iid} D = D_{xy}$ (supervised)
- Sample set: $S = \{(x_i, y_i)\}_{i=1}^m \sim D^m$
- Model: h: $\mathbb{R}^d \longrightarrow \{\pm 1\}$
- Expected error : $L_D(h) = \mathbb{P}_{(x,y)\sim D}[y \neq h(x)] = \mathbb{E}_{(x,y)\sim D}[1\{y \neq h(x)\}]$
- Empirical error: $L_S(h) = \mathbb{P}_S[y \neq h(x)] = \frac{1}{m}\sum_i 1\{y \neq h(x_i)\}$

The learning problem:

Find a classifier $h(x)$ that's accurate. We will define it as:

$$accuracy(h) = \mathbb{P}_{(x,y)\sim D}[y = h(x)]$$

Or as minimizing the expected error: (0\1 loss)

$$err(h) = 1 - accuracy(h) = \mathbb{E}_{(x,y)\sim D}[1\{y \neq h(x)\}]$$

We want to find:

$$\boldsymbol{h^* \in argmin_{h\in H}\, \mathbb{E}_{(x,y)\sim D}[1\{y \neq h(x)\}]}$$

The real distribution D is unknown. One naïve approach is ERM (Empirical risk minimization):

The output of ERM is a random variable. Questions we should ask ourselves when facing a learning problem:

Optimization:

When can we find the argmin efficiently?

How much time/space does this require?

What are good approximation schemes?

Statistical:

When does low empirical error imply low expected error?

How many samples does this require?

Which model classes can be learned?

Modeling

How can we use our domain knowledge?

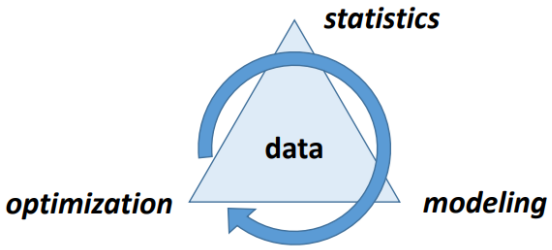
For a given task, what are good models?

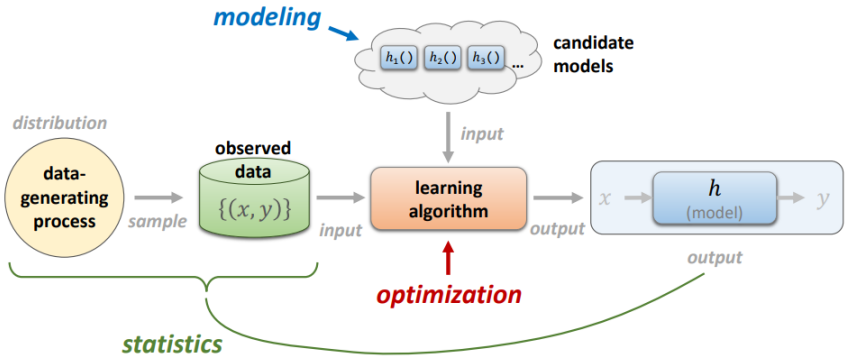
Are we assuming too much? Not enough?

ERM: (Empirical Risk Minimization)

- **Input**:
  - Sample set $S = \{(x_i, y_i)\}_{i=1}^m$
  - Model class H *(candidate classifiers)*
- **Objective**:
  classifier with lowest expected error
  $h^* = \underset{h\in H}{\operatorname{argmin}}\, \mathbb{E}_{(x,y)\sim D}[\mathbb{1}\{y \neq h(x)\}]$
- **Return**:
  classifier with lowest empirical error
  $\hat{h} = \underset{h\in H}{\operatorname{argmin}} \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}\{y_i \neq h(x_i)\}$

foundations of machine learning





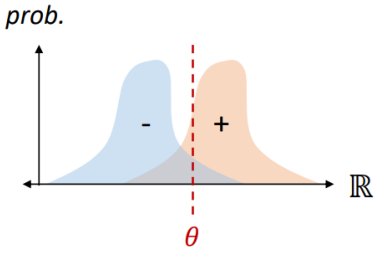
Classification

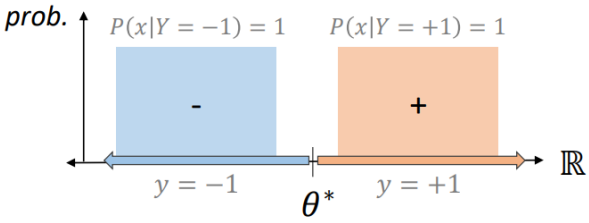
We will focus on three methods of classification:

1. Similarity- based
2. Rule based
3. Linear classifiers

Separable Data: D is separable if exists $\theta^* \in \mathbb{R}\ s.t\ y = 1\ iff\ x > \theta^*$.

We will begin by assuming the data is separable. But we want to also manage data that is almost separable, hence could be overlapping:





Threshold functions could be a good hypothesis class for classification.

$$H_{\text{thresh}} = \{h_\theta(x) = \text{sign}(x \geq \theta) : \theta \in \mathbb{R}\}$$

In that case of the algorithm to the right we get:

$$\hat{h} = h_{\hat{\theta}} = \text{argmin}_{h \in H}\, L_S(h)$$

Revised algorithm:

1. Sort x_i-s
2. For each x_i, compute empirical error "as if" x_i was the threshold
3. Find example with lowest error; call it x_{i^*}
4. Return $\hat{\theta} = x_{i^*}$

Some observations over classification:

1. We can't just assume empirical error ~ expected error.
2. Good results (small empirical error) are relative.

Statistics:

It is helpful to then divide the expected error of the chosen $\hat{h}$ as such:

$$L_D(\hat{h}) = L_D(h^*) + L_D(\hat{h}) - L_D(h^*)$$

Approximation = "how good H is for D" (independent of S)

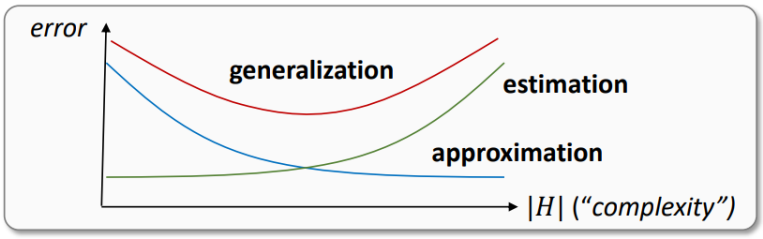
Estimation = "given H, how good is S for D" (empirical error as estimate of expected)

What is the effect of increasing the "size" of H? (Keeping m fixed)

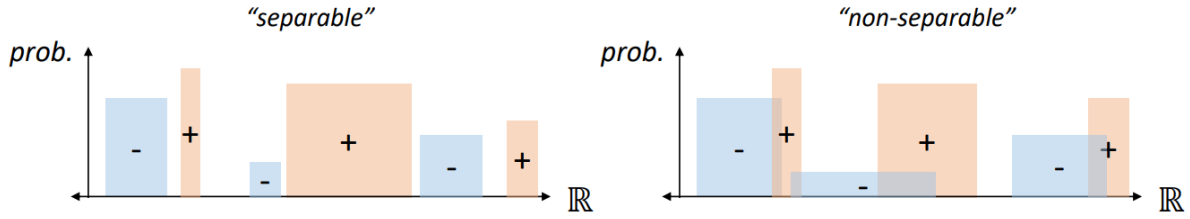
Large H → less bias → lower approximation error → better generalization

Large H→ worse estimation→ higher error→ worst generalization

This is called the bias-complexity tradeoff:



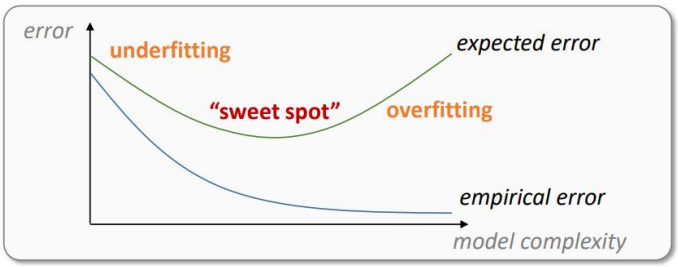
Taking into consideration a different data distribution, say interleaved classes:



Over this dataset, we consider the empirical error of threshold functions as a model class:

- They are statistically performing with low estimation error.
- They may not be best alternative for this dataset – high approximation error.

The model class complexity depends on the expressivity of the functions, and of the number of samples that we have. Increasing complexity can hurt expected error – this concept is called overfitting. Overfitting happens when overly complex models fit noise. Insufficient complexity is called underfitting.



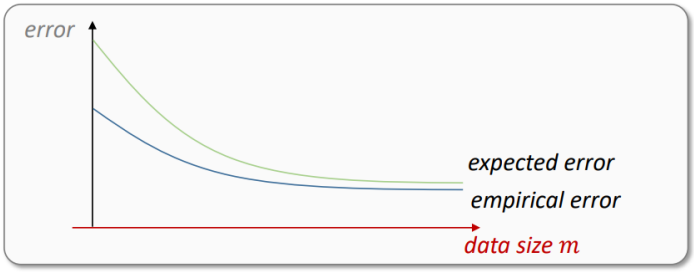
Optimization:

To control this complexity, we could:

1. Restrict hypothesis class complexity
2. Choosing only a subset of our samples

Restricting hypothesis class complexity:

Restriction of the model



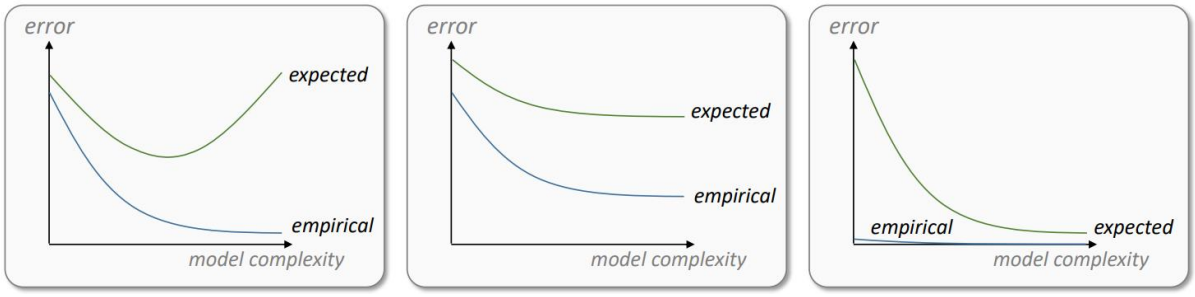
Allows convergence in theory of the expected error to the empirical error. This is helpful only if it happens polynomially-fast.

Note: not all model classes are learnable.

Modeling:

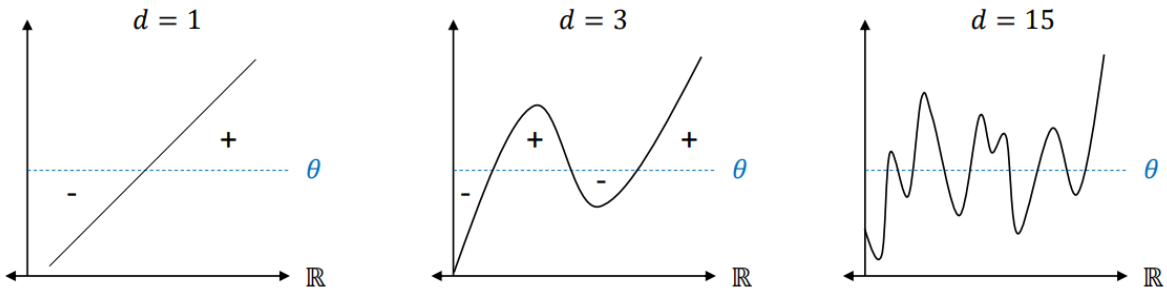
In practice, we have a finite data set (fixed m) and can choose the model class. We will mostly deal with approximate learning algorithms. These are algorithms that find $\hat{h}$ by averaging between multiple sub-datasets, because finding the "best" subset of our samples (the subset that most resembles the distribution D) is a hard combinatorial problem.

Possible scenarios when choosing a model class:



We revisit our second interleaved classes learning problem. We will choose a different model class, thresholds polynomial classifier:

$$h_{p,\theta}(x) = sign(p(x) > \theta)$$



Complexity of this model: number of intervals ⇔ degree of polynomial.

Generalization:

Generalization refers to your model's ability to adapt properly to new, previously unseen data, drawn from the same distribution as the one used to create the model.

Our goal in classification is reducing expected error, by generalizing our learning process, i.e., be wary of over and under fitting. For each method presented below, we will reason generalization by considering:

1. Model complexity
2. Number of samples
3. Dimension of samples

Conclusion:

ERM aims to minimize empirical error, but we care about expected error → our interest is in generalization.

Many subtitles to consider:

- Approximation error and estimation error
- Overfitting and underfitting
- The relation between H and m
- The computational hardness of learning
- How to define “good”

Similarity-based Classification (kNN)

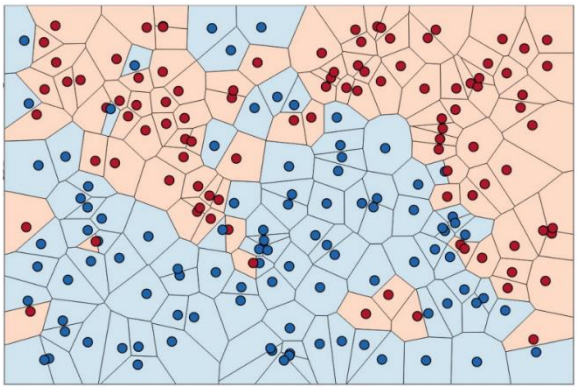
This method assumes that the closest examples of a given example could use to predict its label.

K Nearest Neighbors:

Classification rule:

$$h_k(x;S) = majority\{\, y_i : x_i \in Nei_k(x;S) \,\} = sign(mean\{\{\, y_i : x_i \in Nei_k(x;S) \,\})$$

Model class is complicated and could be very expressive.

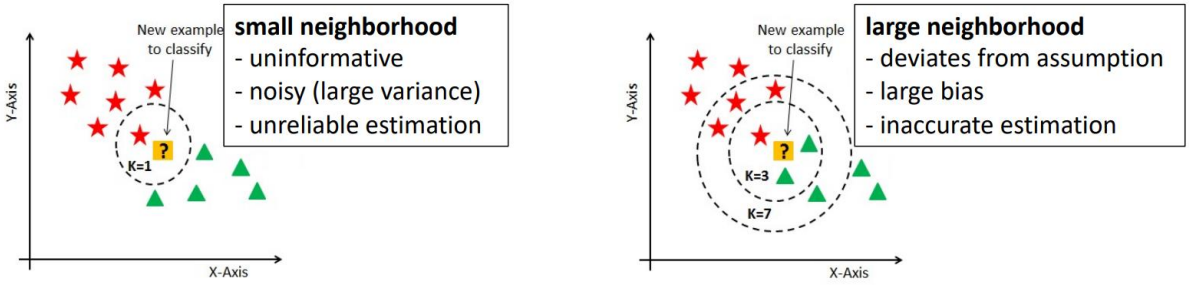


kNN resolves in a complicated function that memorizes the entire training set. This is called instance-based learning. kNN doesn't really have a learning phase, as a sample is predicted by computing during runtime.

kNN requires two modeling choices:

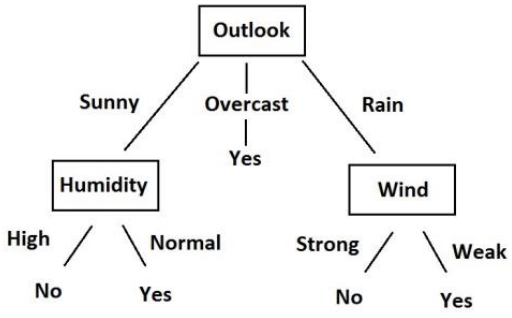
1. Number of neighbors, k
2. Similarity measure

The number of neighbors to consider is a tradeoff between bias and variance:



Scaling (normalizing) the features is necessary in models that consider distances between observations, like kNN for instance. Scaling puts all features into the same range so that they will all have similar effect over the prediction.

Rule based models (Decision trees)



Model class: $H_{DT} = \{h_T(x) : T \text{ is a decision tree}\}$

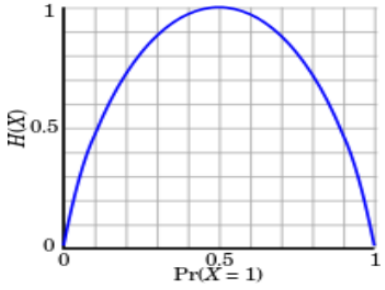
The standard algorithm for creating a decision tree is greedy and recursive. At each step, we locally minimize empirical error.

A node of the tree is called pure if all its samples share the same label. Reasonable measure for impurity is entropy:

$$H = -p_v log p_v - (1-p_v)\log(1-p_v)$$

The greedy step splits the dataset in a way that minimizes entropy, i.e., split to decrease impurity the most.

We will only deal with binary trees.



Splitting criterion:

Entropy before split — Entropy after split, a=F

$$IG(v,a) = H(v) - \frac{|v_{a=T}|}{|v|} H(v_{a=T}) - \frac{|v_{a=F}|}{|v|} H(v_{a=F})$$

Information gain — Entropy after split, a=T

Where a is a binary feature or a target condition, and $v_{a=T} \coloneqq \{ (x,y) \in v \mid x_a = T \}$.

Example of calculating the next best split to predict corona:

Attribute	$\frac{\lvert v_{a=T}\rvert}{\lvert v\rvert}$	$\frac{\lvert v_{a=F}\rvert}{\lvert v\rvert}$	$H(v_{a=T})$	$H(v_{a=F})$	$\mathrm{IG}(v,a) - H(v)$
Fever	$4/7$	$3/7$	$H(3/4)$	$H(1/3)$	$-\frac{4}{7}H(3/4) - \frac{3}{7}H(1/3)$
Cough	$5/7$	$2/7$	$H(3/5)$	$H(1/2)$	$-\frac{5}{7}H(3/5) - \frac{2}{7}H(1/2)$
Smell loss	$3/7$	$4/7$	0	$H(1/4)$	$-\frac{4}{7}H(1/4)$

ID	Fever	Cough	Smell loss	Corona
1	F	T	F	F
2	F	T	F	F
3	F	T	T	T
4	T	F	F	F
5	T	F	T	T
6	T	T	T	T
7	T	T	F	T

Algorithm:

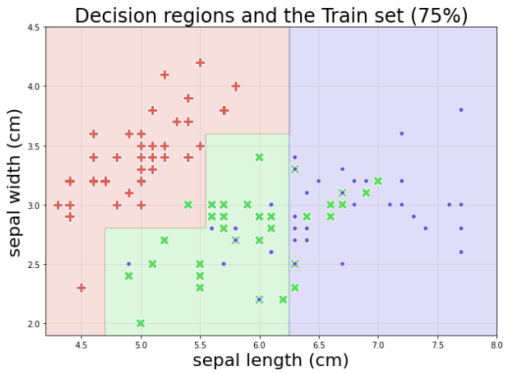
1. Start with root as leaf.
2. For each feature, compute best possible split (using entropy)
3. Create decision node (and split data) using best feature
4. Recurse

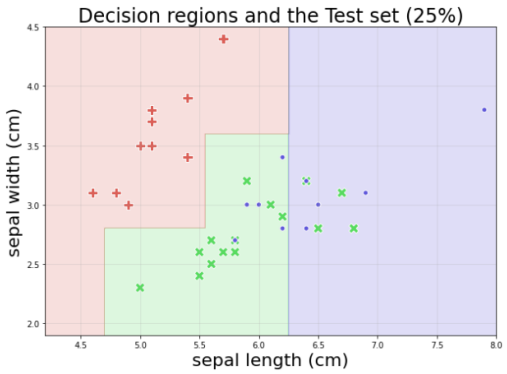
Some parameters of decision trees (many more at sklearn's decision trees):

1. Maximum depth of the tree (not including the root)
2. Minimum number of samples in a leaf
3. Maximum number of leaves
4. Minimum impurity decrease (in every split)

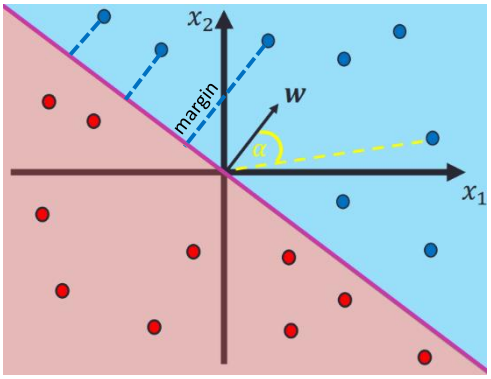
Normalizing the data is not necessary for decision tree making since the algorithm doesn't consider the relations between samples or the relative magnitudes of the features.

Without constraints over the model's complexity, it could always reach perfect training score, i.e., 0 empirical error. Thus, this model is prone to overfit if not adjusted properly (stopping criteria).





Linear Classifiers



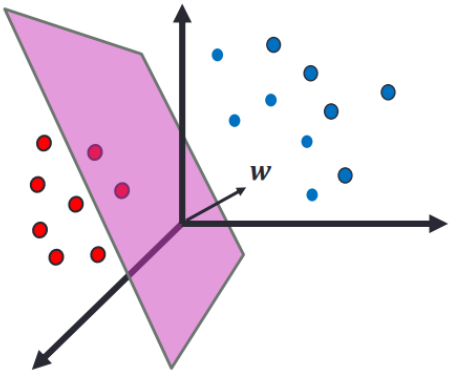
We aim to find a linear separator of a linearly separable dataset. We define any separating line by using a normal w perpendicular to it.

The distance between data points and the separator is called margin. The higher the margin, the more confidence we have of this data point.

$$cos\alpha = \frac{w^T x}{|w||x|} \Rightarrow margin = \frac{w^T x}{|w|}$$

And thus, the linear classifier is:

$$h_w(x) = sign(w^T x) = \begin{cases} 1, & x \text{ is blue} \\ -1, & x \text{ is red} \end{cases}$$



We use the same classifier in higher dimensions but address the separator as a hyperplane.

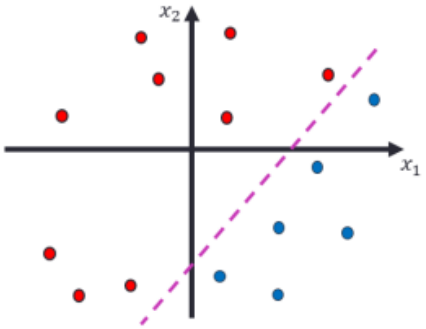
If the data **is not centered,** then we add a bias term to reach a separator that is non-homogeneous:

$$h_{w,b}(x) = sign(w^T x + b) = sign\left(\begin{bmatrix} w \\ b \end{bmatrix}^T \begin{bmatrix} x \\ 1 \end{bmatrix}\right)$$

Therefore, our choices in linear models are:

1. Weights – the vector $w \in \mathbb{R}^d$
2. The bias/offset b

Non-homogeneous linear separator



Linear models have limited expressive power, but this simplicity makes learning traceable (computationally, statistically).

Model class: $H = \{h_w(x) = sign(\langle w, x \rangle) : w \in \mathbb{R}^d\}$

Expected error: $L_D(h) = \mathbb{E}_{(x,y)\sim D}[1\{y \neq sign(w^T x)\}]$

The generalization is finding: $\underset{w\in\mathbb{R}^d}{\operatorname{argmin}}\ \mathbb{E}_{(x,y)\sim D}[1\{y \neq sign(w^T x)\}]$

Scaling is not necessary because loss is invariant to scale:

$$L_D(w) = L_D(\alpha w), \forall \alpha \in \mathbb{R}_+$$

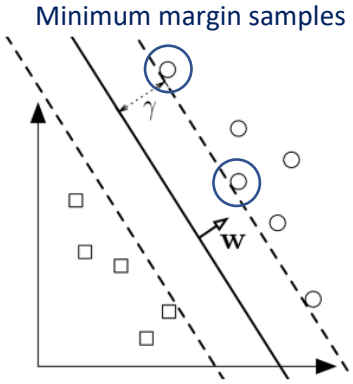
Empirical error: $L_S(h) = \frac{1}{m}\sum_i 1\{y \neq sign(w^T x)\}$

We will find: $\underset{w\in\mathbb{R}^d}{\operatorname{argmin}}\ \frac{1}{m}\sum_i 1\{y \neq sign(w^T x)\}$

Hard SVM

The margin of a linear classifier is the distance to w from the closest point in the dataset:

$$margin(w;S) = \min_{i\in[m]} \frac{|w^T x|}{\|w\|} := \gamma(w;S)$$



We will use **max margin** as our objective of finding the best separator. We will use constraint that imposes that the separator is correct over all samples.

$$\operatorname*{argmax}_w \gamma(w;S) \;\; s.t \;\; y_i \cdot w^T x_i \geq 0 \;\; \forall i \in [m] =$$

$$\operatorname*{argmax}_w \frac{1}{\|w\|} \min_{i\in[m]} |w^T x| \; s.t \;\; y_i \cdot w^T x_i \geq 0 \;\; \forall i \in [m]$$

We will also constraint that $\min_{i\in[m]} |w^T x| = 1$, which guarantees all w-s have a margin of size 1 in the objective.

Now we have:

convex linear

$$= \operatorname*{argmax}_w \frac{1}{\|w\|} \; s.t \;\; y_i \cdot w^T x_i \geq 1 \;\; \forall i \in [m] = \operatorname*{argmin}_w w^T w \; s.t \;\; y_i \cdot w^T x_i \geq 1 \;\; \forall i \in [m]$$

This approach is called Hard Support Vector Machines. It is a convex problem with linear constraints.
Increasing margin = reducing norm, since: $\gamma(w;S) = \frac{1}{\|w\|_2}$

Note: Convex problems are easy to optimize and have a unique solution.

Note: Notice how removing data samples that are not of minimum margin does not affect the model's predicted separator.

Soft SVM

When the data is not separable, the constraints of the hard SVM cannot be satisfied. Instead, we use "soft" constraints; penalize w by how much the constraints are violated:

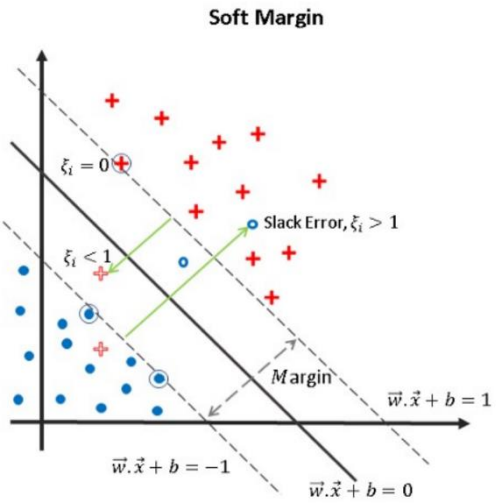
Learning objective: $\operatorname*{argmin}_{w\in\mathbb{R}^d, \zeta\in\mathbb{R}^d} \lambda\|w\|_2^2 + \frac{1}{m}\sum_i \zeta_i \; s.t \;\; y_i \cdot w^T x_i \geq 1 - \zeta_i \;\; \forall i \in [m], \; \zeta_i \geq 0$

Penalties $\zeta_i \geq 0$ are also called slack variables. Separable data has optimal solution of $\zeta_i = 0$.

The penalty is received for points in the inside margin or on the wrong side. We can conclude that:

$$\zeta_i = \begin{cases} 0 & if \;\; y_i \cdot w^T x_i \geq 1 \\ 1 - y_i \cdot w^T x_i & if \;\; y_i \cdot w^T x_i < 1 \end{cases}$$

$$= \mathbf{max}\{\mathbf{0}, \mathbf{1} - \;\; \boldsymbol{y_i \cdot w^T x_i}\}$$



The final soft SVM formula:

$$\operatorname*{argmin}_{w\in\mathbb{R}^d} \lambda\|w\|_2^2 + \frac{1}{m}\sum_i \max\{0, 1 - \;\; y_i \cdot w^T x_i\}$$

Regularization Loss

Another way of forming this formula from the tutorial is:

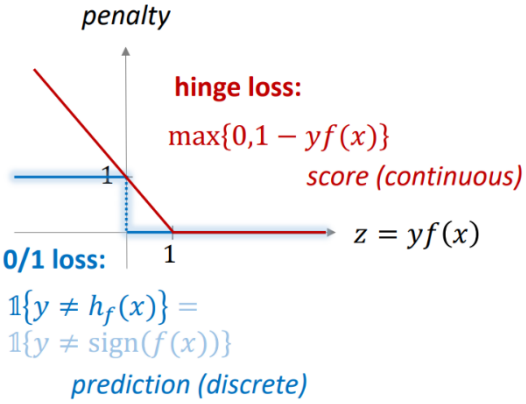
$$\underset{w\in\mathbb{R}^d}{\operatorname{argmin}} \ \|w\|_2^2 + C \cdot \sum_i \max\{0,1 - \ y_i \cdot w^T x_i\}$$

Hinge formulation

$$\operatorname{argmin} regularization + average(\text{hinge loss})$$

This objective is a general template of hinge learning objective.
→We can use other regularizations or other losses.

Loss: penalizes model for being wrong. SVM loss is called hinge loss (hinge = ציר ,מפרק). The loss is plotted for each example and averaged over many of them.



- The hinge loss is convex, continuous, and upper bounds the 0/1 loss.
- It may suffer loss even if correct.
- If wrong, suffers linear loss.

Regularization: The norm restricts model complexity. It encourages w to be small →effectively control the size of the model class →control overfitting.

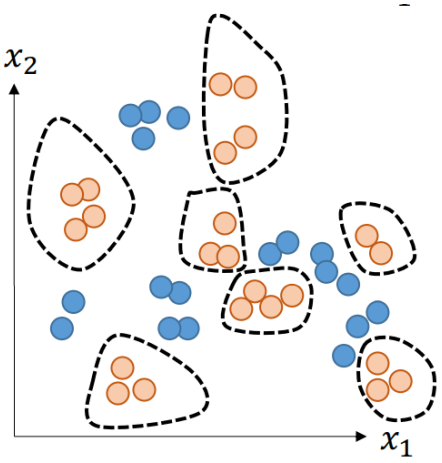
To determine what's more important: large margin vs. small penalty, we use the parameter λ.

Large λ → minimizing $\|w\|_2^2$ is more important → enlarging $\gamma(w;S)$ → large margin→ lower complexity (underfitting?)

Small λ → minimizing $\sum_i \max\{0,1 - \ y_i \cdot w^T x_i\}$ is more important → small penalty (overfitting?)

Dual SVM and Kernels

We can alter the mechanism of SVM and create a linear classifier of non-linear functions. This is helpful when the data is separable but not linearly.



The new Dual SVM objective: $\max_{\alpha\in\mathbb{R}_+^m} \min_{w\in\mathbb{R}^d} \frac{1}{2}\|w\|_2^2 - \sum_i \alpha_i(y_i \cdot w^T x_i - 1)$

We solve for optimal w as function of α, by setting the derivative = 0:

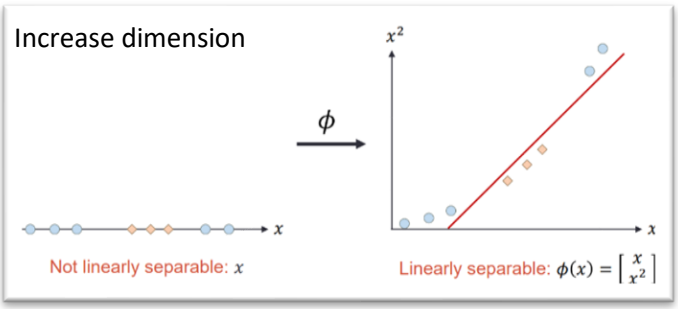
$\frac{\partial L}{\partial w} = w - \sum_i \alpha_i y_i x_i = 0 \ \rightarrow \ w = \sum_i \alpha_i y_i x_i \ \rightarrow \ f_w(x) = w^T x = \sum_{i=1} \alpha_i \, y_i x_i^T x = f_\alpha(x)$

Plug this in and we get:

- **Dual objective** : $\underset{\alpha\in\mathbb{R}_+^m}{\operatorname{argmax}} \ \sum_{i=1} \alpha_i - \frac{1}{2}\sum_{i,j=1} y_i y_j \alpha_i \alpha_j x_i^T x_j$
- **Classifier**: $f_\alpha(x) = \sum_{i=1} \alpha_i \, y_i x_i^T x$

Both the learning objective and the classifier calculate only inner products of our samples. Thus, we can use any feature mapping and the inner products will also map using a kernel function:

1. Feature mapping: $\phi: \mathbb{R}^d \to \mathbb{R}^{d'}$
2. Kernel function: $K: x^T x' \to \phi(x)^T \phi(x')$



Feature mappings could add more features or decrease the dimension of the input.

Some kernel functions could be easily computed. Examples of common kernels with compute time of $O(d)$:

- Polynomials of degree k: $K(x, x'; k) = (x^T x')^k$
- Polynomials of degree $\leq$ k: $K(x, x'; k) = (x^T x')^k$
- RBF/ Gaussian $K(x, x'; \sigma) = \exp\{-\frac{1}{\sigma^2}\|x - x'\|_2^2\}$:
- Sigmoid: $K(x, x'; \eta, v) = \tanh(\eta x^T x' + v)$

Kernels provide a similarity measure via inner products. This provides flexible and expressive modeling power.

Note: This is the same objective as: $\underset{w \in \mathbb{R}^{d'}, b \in \mathbb{R}}{\operatorname{argmin}} \|w\|_2^2 + C \cdot \sum_i \max\{0, 1 - y_i(w^T \phi(x_i) + b)\}$

Well defined kernels:

One way to check if a kernel function is valid is if it follows any of these 8:

1. $K(x, x') = x^T x'$
2. $K(x, x') = x^T A x',\ A\ is\ PSD$
3. $K(x, x') = cK_1(x, x'), c \geq 0$
4. $K(x, x') = K_1(x, x') + K_2(x, x')$
5. $K(x, x') = K_1(x, x') \cdot K_2(x, x')$
6. $K(x, x') = g(K_1(x, x'))\ , g\ polynomial$
7. $K(x, x') = f(x) K_1(x, x') f(x'), f\ is\ a\ function$
8. $K(x, x') = \exp\{K_1(x, x')\}$

Behind any kernel function lies the feature mapping representation ϕ. Example for the RBF kernel:

$$\exp\left(-\frac{1}{2}\|\mathbf{x} - \mathbf{x}'\|^2\right) = \exp\left(\frac{2}{2}\mathbf{x}^\top \mathbf{x}' - \frac{1}{2}\|\mathbf{x}\|^2 - \frac{1}{2}\|\mathbf{x}'\|^2\right)$$

$$= \exp(\mathbf{x}^\top \mathbf{x}') \exp\left(-\frac{1}{2}\|\mathbf{x}\|^2\right) \exp\left(-\frac{1}{2}\|\mathbf{x}'\|^2\right)$$

$$= \sum_{j=0}^{\infty} \frac{(\mathbf{x}^\top \mathbf{x}')^j}{j!} \exp\left(-\frac{1}{2}\|\mathbf{x}\|^2\right) \exp\left(-\frac{1}{2}\|\mathbf{x}'\|^2\right)$$

$$= \sum_{j=0}^{\infty} \sum_{\sum n_i = j} \exp\left(-\frac{1}{2}\|\mathbf{x}\|^2\right) \frac{x_1^{n_1} \cdots x_k^{n_k}}{\sqrt{n_1! \cdots n_k!}} \exp\left(-\frac{1}{2}\|\mathbf{x}'\|^2\right) \frac{x'^{n_1}_1 \cdots x'^{n_k}_k}{\sqrt{n_1! \cdots n_k!}}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{\text{single entry in } \phi(x)} \qquad \underbrace{\qquad\qquad\qquad\qquad}_{\text{single entry in } \phi(x')}$$

Note: the RBF Kernel covers the space with balls of fixed radiuses. It can also be written as: $K(x, x'; \sigma) = \exp\{-\gamma\|x - x'\|_2^2\}$:

Smaller γ → large radius → lower complexity

Larger γ → smaller radius → may lead to overfitting

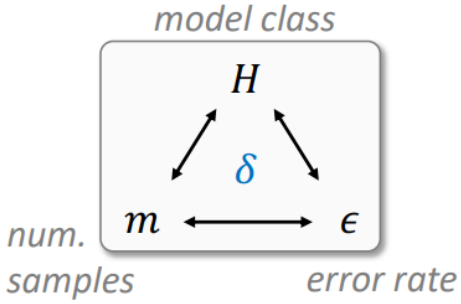
Note: Kernel SVM struggles computationally with higher dimensions and/or many samples.

More on kernel SVM, another video.

Statistical Aspects of Learning

We would like to investigate $L_D(h_S)$. Recall we can only bound or estimate it. Key players in forecasting are model class, num of samples, and error rate: H, m, ϵ.

H is realizable for D if there exists $h^* \in H$ s.t $L_D(h^*) = 0$.



We proved in class that:

a. $P_{S\sim D^m}(L_D(h_S) \geq \epsilon) \leq |H|e^{-\epsilon m}$ for:
   1. $m \geq \frac{\log|H|+\log\left(\frac{1}{\delta}\right)}{\epsilon}$
   2. $\epsilon \geq \frac{\log|H|+\log\left(\frac{1}{\delta}\right)}{m}$

b. $P_{S\sim D^m}(L_D(h_S) - L_D(h^*) \geq \epsilon) \leq 2e^{-2\epsilon^2 m}$ for: (H which is not realizable)
   1. $m \geq \frac{\log 2|H|+\log\left(\frac{1}{\delta}\right)}{2\epsilon^2}$
   2. $\epsilon \geq \sqrt{\frac{\log 2|H|+\log\left(\frac{1}{\delta}\right)}{2m}}$

PAC What generalization guarantees can learning algorithms give?

H is PAC-learnable if $\exists A, \exists\, m_H(\epsilon,\delta) \in poly\left(\frac{1}{\epsilon},\frac{1}{\delta}\right)\ s.t\ \forall D$ (for which H is realizable) and

$\forall \epsilon,\delta \in [0,1], if\ m \geq\ m_H(\epsilon,\delta)\ , then: P_{S\sim D^m}(L_D(h_S) \geq \epsilon) \leq \delta$

PAC theory allows to measure success in learnability in a probabilistic way. The previous bound showed finite $|H|$ are learnable with $|S| \geq \frac{\log\left(\frac{H}{\delta}\right)}{\epsilon}$.

Agnostic PAC

H is Agnostic PAC-learnable if $\exists A, \exists\, m_H(\epsilon,\delta) \in poly\left(\frac{1}{\epsilon},\frac{1}{\delta}\right)\ s.t\ \forall D$ and

$\forall \epsilon,\delta \in [0,1], if\ m \geq\ m_H(\epsilon,\delta)\ , then: P_{S\sim D^m}(L_D(h_S) - L_D(h^*) \geq \epsilon) \leq \delta$

Note: agnostic PAC relies on distributional assumptions (i.i.d).

VC- dimensions

The VC dimension of a model class H is the largest set on which $L_S = 0$ is possible for any labeling.
The result – learning is useless when H can perfectly fit any arbitrary label assignments.

Shattering = Explaining everything = Explaining nothing!

H Shatters C if $\forall\{y_i\} \in \{\pm 1\}^m \ \exists h \in H \ \ s.t. \ h(x_i) = y_i \ \ \forall i \in [m]$

The VC dimension is the size of the largest set that H shatters.

If $VC(H) \leq \infty$, then H is:

1. PAC-learnable with: $m_H(\epsilon, \delta) = \Theta\left(\frac{VC(H)log1/\epsilon + log1/\delta}{\epsilon}\right)$
2. Agnostic PAC-learnable with: $m_H(\epsilon, \delta) = \Theta\left(\frac{VC(H)+log1/\delta}{\epsilon^2}\right)$

"Plug your desired ϵ, δ, and you will need more then m examples"

Note: H is PAC-learnable iff its VC-dimension is finite.

To find the VC dimension of a model class, one should find m such that-

1. Exists C of size m that H shatters
2. H does not shatter all sets C of size $m + 1$

The VC dimension:

- Measures capacity of model classes to express binary patterns
- Is a combinatorial measure – no statistics involved
- Works because binary classification is discrete
- Shows that at full capacity, learning breaks.

Model Selection

Bias-variance decomposition

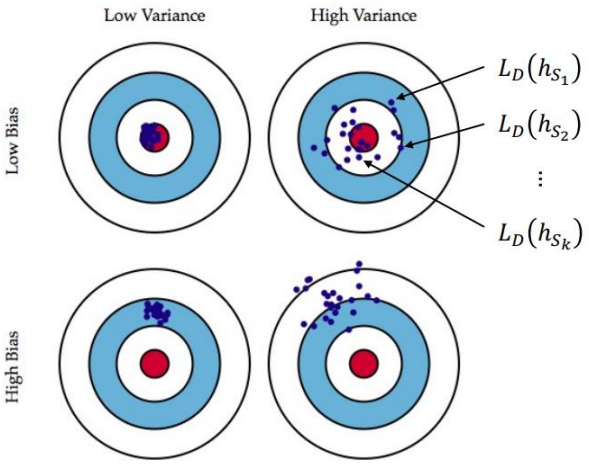
$$\mathbb{E}_{S\sim D^m}\left[L_D^{sqr}(h_s)\right] = \mathbb{E}_{x,y}[\,(\bar{y}(x) - y)^2] + \mathbb{E}_x\left[\left(\bar{h}(x) - y(x)\right)^2\right] + \mathbb{E}_{S,x}\left[\left(h_S(x) - \bar{h}(x)\right)^2\right]$$

$expected\ error$ $noise$ $bias^2$ $variance$

$L_D^{sqr}(h_s) = \mathbb{E}_{(x,y)\sim D}[(h(x) - y)^2]$ = Squared loss

$\bar{y}(x) = \mathbb{E}_{y\sim D_{Y|X=x}}[y] \in [0,1]$ = Expected label

$\bar{h}(x) = \mathbb{E}_{S\sim D^m}[h_S]$ = Expected classifier

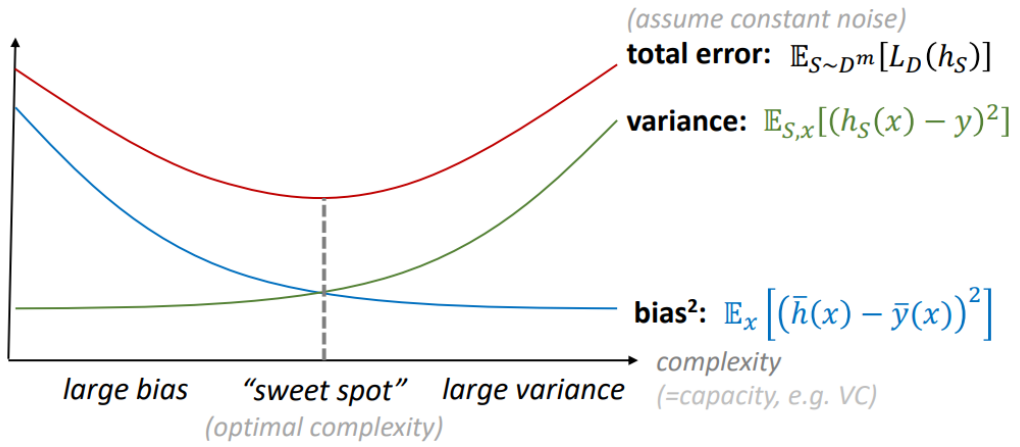


Noise: Property of the data distribution, doesn't depend on the model.

Bias: Quantifies how well our model class fits the data. Does not depend on the sampled data. More complex models→less bias.

Variance: Measures how food output models h_S vary. Uses the average $\bar{h}$ as reference point. Doesn't depend on the real labels. More samples → lower variance. More complex models → higher variance.

Note: There's a tradeoff between bias and variance: to decrease bias we need to increase complexity ⇔ to decrease variance we need to decrease complexity.



SVM in the light of bias and variance

Loss – directly reduces bias.

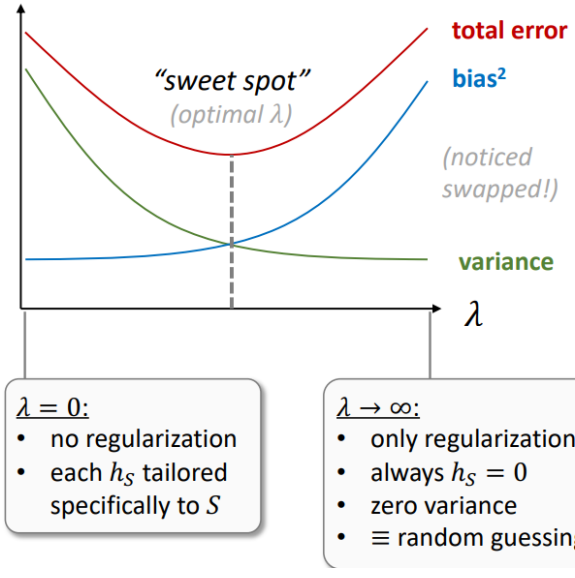
Regularization – indirectly reduces variance.

If we bound w's s.t $H_B = \{w \in \mathbb{R}^d : \|w\|_2 \leq B\}$ then we get that for any m, soft SVM with $\lambda = \sqrt{2/(B^2 m)}$ satisfies:

$$\mathbb{E}_{S\sim D^m}\left[L_D^{hinge}(w_S)\right] \leq \min_{w\in H_B} L_D^{hinge}(w) + B\sqrt{\frac{8}{m}}$$

Restricting norm = control complexity.

How to chooses appropriate λ?



Cross validation

fold 1	**val**	train	train	train	train
fold 2	train	**val**	train	train	train
fold 3	train	train	**val**	train	train
fold 4	train	train	train	**val**	train
fold 5	train	train	train	train	**val**

When modeling we need to carefully control the following:

1. Model class complexity (VC)
2. Hard code into objective (RLM)
3. Regularization (norms)

To do so, we ought to tune the hyperparameters of the model. Tuning is a part of learning, not by ERM. It is practically done by k-fold cross validation. We decide on a number n representing number of folds. Then we divide the dataset into n different subsets. Those subsets are divided to train and validation sets.

One round of cross-validation involves performing the analysis on the training set, and validating the analysis on the validation set. To reduce variability, multiple rounds (folds) of cross-validation are performed using different partitions, and the validation results are combined (e.g., averaged) over the rounds to give an estimate of the model's predictive performance.

Expected error could also be decomposed using the error of the validation set:

$$L_D(h_S) = \big(L_D(h_S) - L_V(h_S)\big) + \big(L_V(h_S) - L_S(h_S)\big) + L_S(h_S)$$

If this term is large, h_S overfits. Options:

- Get more samples
- Lower the capacity
- Change model class

If this term is large, h_S underfits. Options:

- Increase complexity
- Retune hyperparameters
- Change feature mapping
- Change model class

Distribution Shift

Distribution or Dataset shift is a challenging situation where the joint distribution of inputs and outputs differs between the training and test stages. There are multiple manifestations of dataset shift, one of them is the covariate shift.

Covariate shift is the change in the distribution of the covariates specifically, that is, the independent variables. It appears in the case where $\mathbb{P}_{tra}(y|x) = \mathbb{P}_{tst}(y|x)$ and $\mathbb{P}_{tra}(x) \neq \mathbb{P}_{tst}(x)$. This universal marginal satisfies:

$$\mathbb{P}_{trn}(x,y) = \mathbb{P}_{trn}(x)\mathbb{P}(y|x)$$

$$\mathbb{P}_{tst}(x,y) = \mathbb{P}_{tst}(x)\mathbb{P}(y|x)$$

The two most common causes of dataset shift are (1) sample selection bias and (2) non-stationary environments:

Sample selection bias: the discrepancy in distribution is due to training data having been obtained through a biased method, and thus do not represent reliably the operating environment where the classifier is to be deployed (which, in machine learning terms, would constitute the test set).

Non -stationary environments: when the training environment is different from the test one, whether it is due to a temporal or a spatial change.

Covariance shift can cause a lot of problems when performing cross-validation.

Several ways to overcome distribution shift:

1. Feature removal: We can set a boundary on what is deemed an acceptable level of shift, and analyzing individual feature. We can determine which features are most responsible for the shifting and remove these from the dataset.
2. **Reweighting:** Is to change your training data set such that it looks like it was drawn from the test data set. The only thing required for this method is unlabeled examples for the test domain. We define the weight of a feature x as: $w(x) = \frac{p_{tst}(x)}{p_{trn}(x)}$. Since we do not have $p_{tst}(x)$ we estimate it from unlabeled data of both p_{tst} and p_{trn} .
   We learn a classifier to distinguish between data drawn from $p_{tst}(x)$ and data drawn from $p_{trn}(x)$ (a binary target). If it is impossible to distinguish between the two distributions, then it means that the associated instances are equally likely to come from either one of the two distributions. On the other hand, any instances that can be well discriminated should be significantly overweighted or underweighted accordingly. The resulting $h(x_i)$ of this classifier is used to weigh the training data using: $\beta_i = \exp\{h(x_i)\}$.
   Then, we use the β_i weights for training on the original dataset.

Note: This may result in data leakage from the test set.

For more: dataset shift, the MIT press, Understanding Dataset Shift, Correct cov. Shift.

Convex Optimization

Convexity

- C is a convex set if: $\forall c_1, c_2 \in C \ \wedge \forall\, t \in [0,1] : tc_1 + (1-t)c_2 \in C$
- Any intersection of a convex set is a convex set.
- A function $f: C \to \mathbb{R}$ (where C is a convex set) is a convex function if:

$$\forall c_1, c_2 \in C \ \wedge \forall\, t \in [0,1]: f(tc_1 + (1-t)c_2) \leq t \cdot f(c_1) + (1-t) \cdot f(c_2)$$

- Any linear function $g(x) = w^T x + b$ is convex and holds:

$$f(tc_1 + (1-t)c_2) = t \cdot f(c_1) + (1-t) \cdot f(c_2)$$

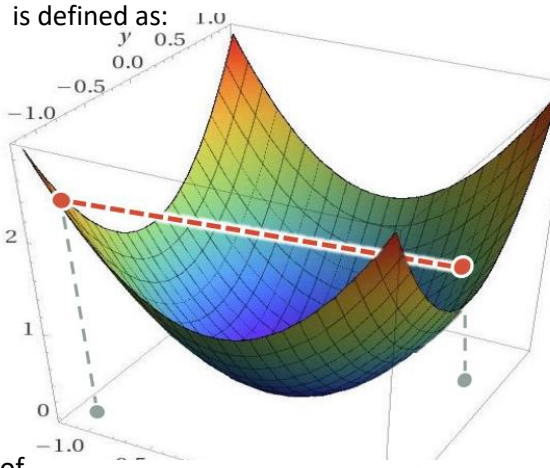
- If C is a convex set and $g, h: C \to \mathbb{R}$ are convex functions, then:
  1. $\forall \alpha \geq 0$: αg is convex.
  2. $g + h$ is convex, and any finite sum of convex functions is convex.

3. If h is linear then $g \circ h$ is convex.

- Any local minimum of a convex function is a global minimum.
- The Hessian matrix $\nabla^2 f$ of a twice differentiable function $f: C \to \mathbb{R}$ is defined as:

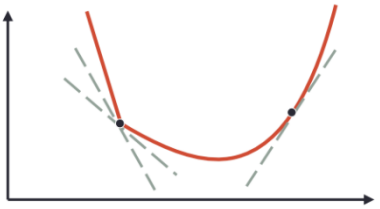
$$\nabla^2 f_{i,j} = \frac{\partial^2 f}{\partial x_i x_j}:$$

$$\nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \frac{\partial^2 f}{\partial x_d \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_d^2} \end{bmatrix}$$



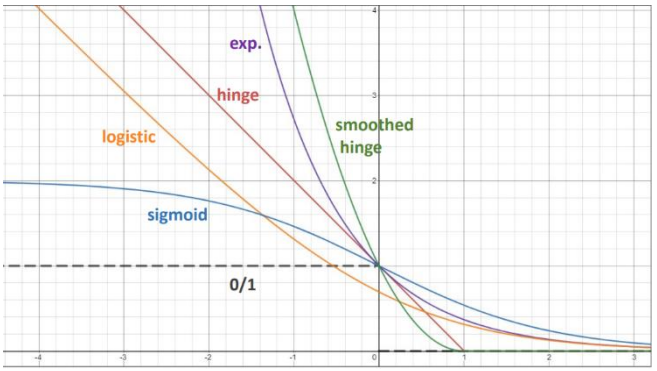
- A twice differentiable function $f: C \to \mathbb{R}$ is convex iff $\nabla^2 f \succcurlyeq 0$
- L_2 norm squared is convex (e.g., $\|w\|_2^2$).
- The soft SVM, hard SVM objectives are convex.
- Let $f: C \to \mathbb{R}$ be a convex function. Denote the set of subgradients of f at point $u \in V$ by $\partial f(u)$, s.t:

  $g \in \partial f(u)$ if $\forall v \in V: f(v) \geq f(u) + \langle g, v - u \rangle$



Loss functions

Loss functions or proxies are the function with which we measure our loss. ERM aims to minimize a specific proxy – the 0/1 loss. This proxy is discrete and not convex. Some options for other proxies are:



1. Hinge loss: $\max\{0, 1 - z\}$
2. Smooth Hinge: $\max\{0, 1 - z\}^2$
3. Ramp loss: $\min\{1, \max\{1 - z\}\}$
4. Logistic: $\ln(1 + e^{-z})$
5. Sigmoid: $\frac{2}{1+e^x}$
6. Exponential: e^{-x}

Gradient Descent

Regularized Risk Minimization (RRM): $\underset{h \in H}{\operatorname{argmin}} \frac{1}{m} \sum_i l(y_i, h(x_i)) + \lambda R(h)$

We want to choose the loss function and the regularization function such that we will have optimization guarantees. This is called to optimize the learning objective. For instance, if we choose our learning objective such that it is convex, we are guaranteed to have a single global minimum.

RRM objective is convex if:

1. $l(\cdot, \cdot)$ is convex
2. $R(\cdot)$ is convex
3. h is linear in w (i.e., $h_w(x) = w^T x$)

The gradient of f w.r.t w is a vector functions of partial derivatives:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial w_1} \\ \vdots \\ \frac{\partial f}{\partial w_d} \end{pmatrix}$$

Gradient Descent Algorithm: (GD)

1. Initialize w_0, η
2. For $t = 1,2 \ldots$
   a. Compute $\nabla f(w_t)$
   b. $w_{t+1} = w_t - \eta \nabla f(w_t)$
3. Until convergence
   (e.g., $\|w_{t+1} - w_t\| \leq \epsilon$)

$\Longleftrightarrow$

Need to determine:

1. How to initialize w_0
2. Step size (learning rate) η
3. When to stop (number of iterations)

When applying GD on a convex objective, the resulting minimum is always the global minimum. In that case, for every starting point w_0, for appropriate η we reach convergence anyways.

When the objective isn't convex, a good method is random initialization. That is to do GD multiple times from different random starting points.

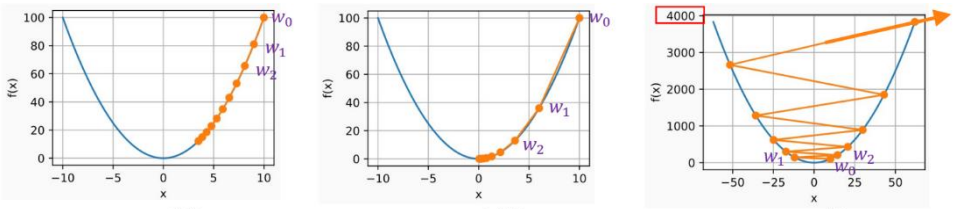
Another possibility is to twist this basic GD idea into stochastic GD, momentum GD or accelerated Gradient. These algorithms have different solutions for rolling through and escaping local minimums, so that the starting point is making less of a difference.

Learning rate:

The learning rate of a GD algorithm is the size of its steps η.

If η is small enough, then $f(w_{t+1}) < f(w)$ for any non-minimum w_t. Therefore→ GD converges to a local minimum (or saddle point) with appropriate η.

Note: A trick is to use time-varying learning rate η_t (for example: $\eta_t = \frac{1}{t}$).



When to stop:

GD converges eventually in theory. In practice, we need to set criteria for stopping. For example, stop when:

- $\|w_{t+1} - w_t\| \leq \epsilon$
- $\|f(w_{t+1}) - f(w_t)\| \leq \epsilon$
- $\|\nabla f(w_t)\| \leq \epsilon$

The convergence rate of GD depends on this criterion, the parameter η and the initialization phase.

GD guarantees convergence to the global minimum if applied over a convex function. But it can still be applied to non-convex objectives (needs differentiability).

Feature scaling (normalization and standardization) is important for GD algorithm since the same η is applied to all dimensions.

Notes:

Normalization is doing something like: $x_i \leftarrow \frac{x_i - \min_i}{\max_i - \min_i} \in [-1,1]$

Standardization is doing something like: $x_i \leftarrow \frac{x_i - \mu_i}{\sigma_i} \approx N(0,1)$

Stochastic Gradient Descent

For running GD we compute gradients of the learning objective:

$$\nabla\left(\frac{1}{\mathrm{m}}\sum_{i\in[m]} l\left(y_i, f_w(x_i)\right)\right) = \frac{1}{\mathrm{m}}\sum_{i\in[m]} \nabla\, l\left(y_i, f_w(x_i)\right)$$

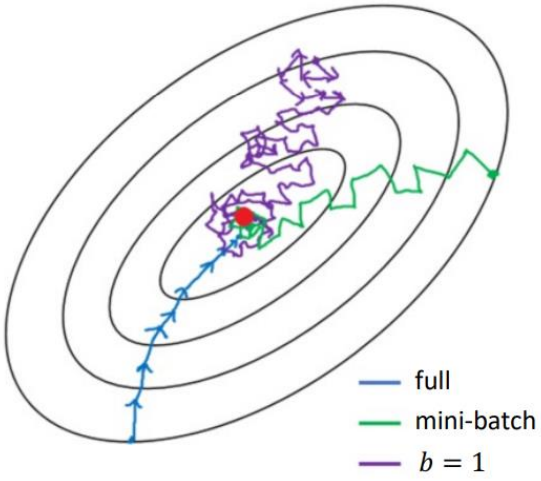
We denote: $\nabla_i = \nabla\, l\left(y_i, f_w(x_i)\right)$. Instead of using all the samples and compute the gradient every step, stochastic gradient descent averages over a mini batch of the samples the following:

Stochastic Gradient Descent Algorithm:

1. Sample small random batch $B \subset S$:
2. Compute avg gradient:

$$\overline{\nabla} = \frac{1}{b}\sum_{i\in B} \nabla_i$$

3. Apply approximate step:
   $x_{t+1} = x_t - \eta\overline{\nabla}$
4. Until convergence



Stochastic GD reduces compute time significantly but adds noise. The size of the batch $|B|$ is the hyperparameter that controls this tradeoff.

Note: experiments show that SGD may perform better (converge after less iterations) than GD. Recent studies show this could happen when the algorithm approaches saddle points. While GD continues to move in the direction of the saddle's plain, SGD manages to find a different path and get out of the it.

Gradient Descent with Momentum

This GD algorithm can be applied over the entire dataset or a batch of it. With every step, this algorithm also computes the "momentum" of this step by looking backwards. That way when approaching wide slopes the momentum increases, and the steps are larger. Respectively, when approaching steep slopes, the momentum decreases to allow convergence with smaller steps.

Gradient Descent with Momentum:

1. Initialize w_0, v_0, η, β
2. Repeat:
   a. $v_{t+1} = \beta v_t - \eta\nabla f(w_t)$ = "momentum/velocity"
   b. $w_{t+1} = w_t + v_{t+1}$
3. Until convergence

Typical β values are 0.9, 0.95, 0.99.

Nesterov's accelerated gradient

Using the momentum GD resolves often with overshooting (going pass the minimum and return) due to generating a lot of momentum down the slope. Nesterov's accelerated gradient is meant to avoid this overshooting by looking ahead.

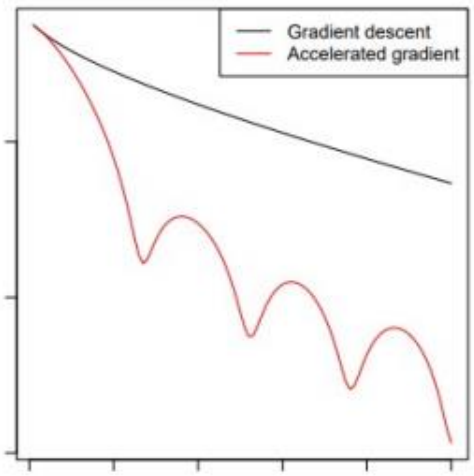
Nesterov's accelerated gradient:

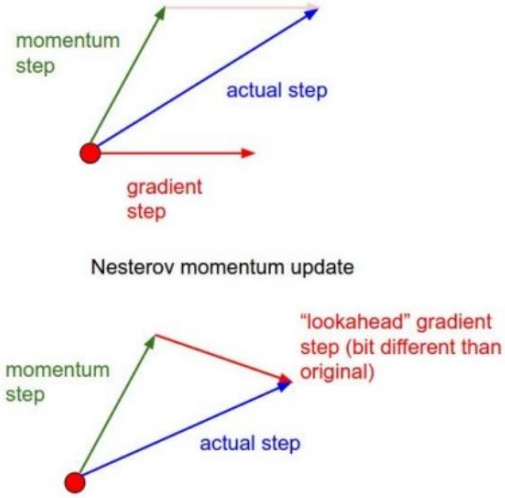
1. Initialize w_0, v_0, η, β
2. Repeat:
   a. $\widetilde{w}_{t+1} = w_t + \beta v_t$
   b. $v_{t+1} = \beta v_t - \eta \nabla f(\widetilde{w}_{t+1})$
   c. $w_{t+1} = w_t + v_{t+1}$
3. Until convergence

Equivalent:

1. Initialize w_0, v_0, η, β
2. Repeat:
   a. $v_{t+1} = \beta v_t - \eta \nabla f(w_t)$
   b. $w_{t+1} = w_t + (1+\beta) v_{t+1} - \beta v_t$
3. Until convergence

This is not a descent method, but it converges faster:





GD and SGD for SVM objective

GD uses all directions: $w_{k+1} = w_k - \eta(\frac{1}{m}\sum_{i\in[m]} \nabla_w l(x_i, y_i, w) + \lambda \nabla_w R(w))$

SGD uses a random set: $w_{k+1} = w_k - \eta\left(\nabla_w l(x_i, y_i, w) + \lambda \nabla_w R(w)\right)$

Note: directions are equal in expectation:

$$\underbrace{\mathbb{E}_{(x_i,y_i)\sim S}\left[\nabla_{\boldsymbol{w}} \ell\left(\boldsymbol{x}_i, y_i, \boldsymbol{w}^{(k)}\right)\right]}_{\text{SGD}} + \lambda \nabla_{\boldsymbol{w}} R\left(\boldsymbol{w}^{(k)}\right) = \underbrace{\frac{1}{m}\sum_{i=1}^{m}\left(\nabla_{\boldsymbol{w}} \ell\left(\boldsymbol{x}_i, y_i, \boldsymbol{w}^{(k)}\right)\right)}_{\text{GD}} + \lambda \nabla_{\boldsymbol{w}} R\left(\boldsymbol{w}^{(k)}\right)$$

Perceptron

First and simplest linear model. Trained iteratively:

Perceptron Algorithm:

1. $w_0 = 0_d$
2. While *didn't separate trainset:*
   a. For $i = 1,2,\ldots m$
      i. $\hat{y}_i = sign(w^T x_i)$
      ii. If $\hat{y}_i \neq y_i$: $w = w + \eta y_i x_i$

Perceptron performs SGD with a hinge-like loss:

$$loss = \max\{0, -y_i w^T x_i\}$$

$$risk = \frac{1}{m}\sum_{i\in[m]} -y_i w^T x_i \cdot 1[y_i \neq sign(w^T x_i)]$$

$$\nabla risk(w) = \frac{1}{m}\sum_{i\in[m]} -y_i x_i \cdot 1[y_i \neq sign(w^T x_i)]$$

Using SGD with batch size = 1 (a single mistake in prediction) the gradient step is:

$$\nabla_{\mathrm{w}}^{batch=1} risk(w) = -y_i x_i$$

Therefore, the step size in perceptron: $w = w - \eta \cdot (-y_i x_i) = w + \eta y_i x_i$ is the same as the calculated SGD step.

Regression

Regression is a classification problem involving a target which is not discrete.

Linear regression

Statistical modeling assumes: $y = f^*(x) + \epsilon,\ f \in F, \epsilon \sim D\_err$

In Linear regression we assume:

1. Linear model class: $F = \{f(x) = w^T x : w \in \mathbb{R}\}$
2. Normal, iid errors: $\epsilon \sim^{iid} N(0, \sigma^2)$

Plug it in and our distributional assumption is:

$$P(y|x; w) = N(w^T x, \sigma^2)\ for\ some\ w$$

$P(y|x; w)$ – given w, what are the chances of observing y given x?

$P(y|x; w)$ – given (x, y), what are the chances of it generated by w?

We define: $P(S; w) = L(w; S)$ as the likelihood. → our goal is learning w that best explains the data.

Maximum Likelihood Estimation (MLE):

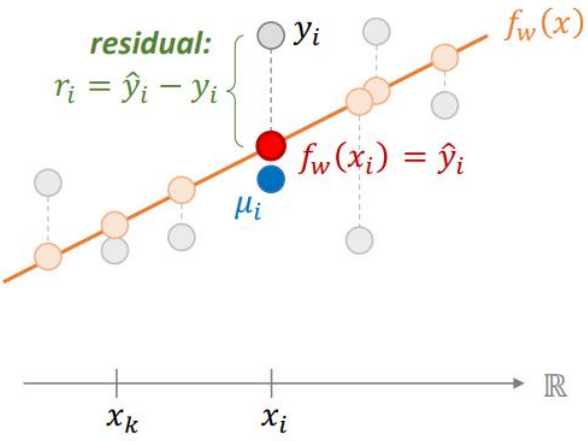
$$w = w_{MLE} = \underset{w\in\mathbb{R}^d}{\operatorname{argmax}} L(w; S) = \underset{w\in\mathbb{R}}{\operatorname{argmin}} \frac{1}{m}\sum_{i\in[m]} (y_i - w^T x_i)^2 = \underset{w\in\mathbb{R}}{\operatorname{argmin}} \frac{1}{m}\|Xw - y\|_2^2$$

This objective is convex, solve using GD or SVD:

- GD: $w_{k+1} = w_k - \eta X^T(Xw - y)$
- SVD: $w_{opt} = (X^T X)^+ X^T y = V\Sigma U^T y$

Linear regression allows us to share label information across examples. Minimizing mean squared errors $(y_i - w_T x_i)^2$ means we aim for the "line" $w^T x$ to pass through all true averages $\bar{y}_i$. This comes from our assumption that $P(y|x; w) = N(w^T x, \sigma^2)$.

Our estimation becomes prediction. Ideally, we'd like that $\mathbb{E}(\hat{y}) = \mathbb{E}(\bar{y})$, and we hope to approach as m increases.



Residual analysis can help us understand if our assumptions were sensible. We define $r_i = \hat{y}_i - \bar{y}_i$, and aim that: $\mathbb{E}(\hat{y}) - \mathbb{E}(\bar{y}) = \mathbb{E}(\hat{y} - \bar{y}) = \mathbb{E}(r) = 0$.

Note: In linear regression we typically don't need a proxy loss and optimize squared loss.

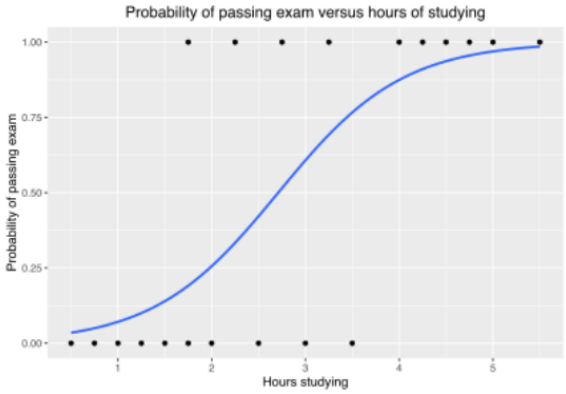
How to interpret performance?

The performance of the linear regression measures how much of the variance the model can explain using the features (how good is our "best guess").

Generalization

VC theory does not apply (it's specific to binary classification). We avoid overfitting by using regularization, as explained below (Ridge/Lasso regression).

Logistic regression



In Logistic regression we assume:

1. $Y|X = x \sim Bernulli(p) \rightarrow P(y_i|x_i) = \begin{cases} p_i, & y_i = 1 \\ 1 - p_i, & y_i = 0 \end{cases}$

Recall Bernulli – $\mathbb{E}[Y|X = x] = p_x$, and we want to model $f(x) = \mathbb{E}[Y|X = x] = p_x$.

We will use $p_x = \sigma(w^T x)$ whereas $w^T x$ is the linear regression computation and the σ applies a sigmoid transformation:
$p_x(w) = P(Y = 1|x; w) = \frac{1}{1+e^{-w^T x}} \coloneqq \sigma(w^T x) \in [0,1]$

Learning w gives us the following predictor:

$$p' = \sigma(w^T x + b) = \frac{1}{1 + e^{-w^T x - b}} \in [0,1]$$

σ is a logistic function. To classify binary targets using logistic regression one can threshold:

$\hat{y} = 1\{p > 0.5\}$. The goal of logistic regression is to estimate the probability of occurrence, not the value of the variable itself.

Ridge Regression

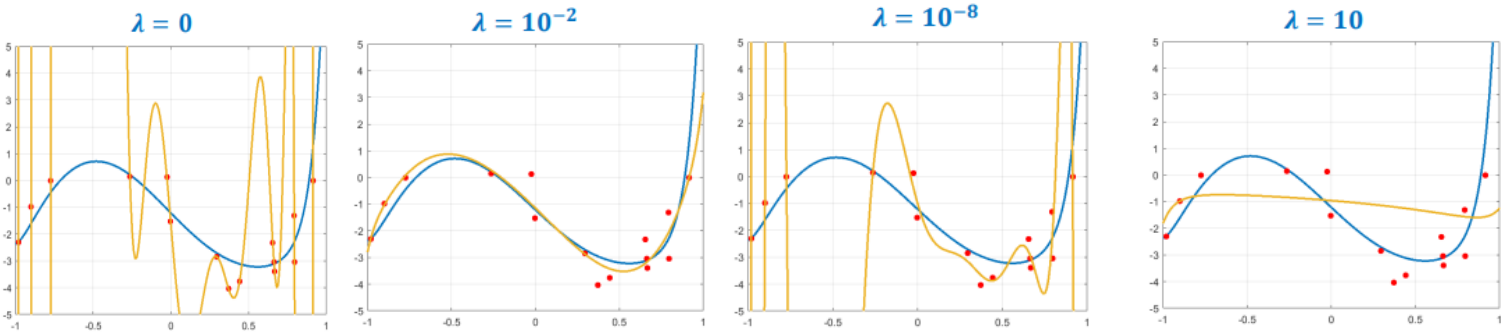
Ridge regression regularize solutions with the L2 norm:

$$w = \underset{w \in \mathbb{R}}{\operatorname{argmin}} \frac{1}{m} \sum_{i \in [m]} (y_i - w_T x_i)^2 + \lambda \|w\|_2^2 = \underset{w \in \mathbb{R}}{\operatorname{argmin}} \frac{1}{m} \|Xw - y\|_2^2 + \lambda \|w\|_2^2$$

Equivalent unregularized objective uses norm constraints:

$$w = \underset{w\in\mathbb{R}}{\operatorname{argmin}} \frac{1}{m}\sum_{i\in[m]}(y_i - w_T x_i)^2 \;\; s.t \;\; \|w\|_2^2 \leq c$$

The loss remains convex and differentiable so GD still applies.



Lasso Regression

Lasso = Least Absolute Shrinkage and Selection Operator

Lasso regression regularize solutions with the L1 norm:

$$w = \underset{w\in\mathbb{R}}{\operatorname{argmin}} \frac{1}{m}\sum_{i\in[m]}(y_i - w_T x_i)^2 + \lambda\|w\|_1 = \underset{w\in\mathbb{R}}{\operatorname{argmin}} \frac{1}{m}\|Xw - y\|_2^2 + \lambda\|w\|_1$$

Equivalent unregularized objective uses norm constraints:

$$w = \underset{w\in\mathbb{R}}{\operatorname{argmin}} \frac{1}{m}\sum_{i\in[m]}(y_i - w_T x_i)^2 \;\; s.t \;\; \|w\|_1 \leq c$$

Loss remains convex but no longer differentiable (could run subgradient-descent). LASSO induces sparser solutions.

Notes:

1. Using subgradient-descent for lasso resolves in a solution very close to the true lasso solution but may not contain exact zeros as weights (very small values instead). This lack of sparsity is one reason not to use subgradient methods for lasso.
2. Both in ridge and lasso regression we don't get the solution with the minimal training mean squared error (MSE).
3. Larger unregularized coefficients aren't necessarily more important!
4. Cool animation by Itay Evron that demonstrates the sparsity of LASSO.

Bagging and Boosting

Bagging and Boosting are meta-learners for ensemble methods. The template of ensemble methods is generally:

1. Base class learner
2. Aggregation rule
3. Meta-learner (Bagging, Boosting)

Bagging

$$\mathbb{E}_{S\sim D^m}\left[L_D^{sqr}(h_S)\right] = \mathbb{E}_{x,y}[\,(\bar{y}(x)-y)^2] + \mathbb{E}_x\left[\left(\bar{h}(x)-y(x)\right)^2\right] + \mathbb{E}_{S,x}\left[\left(h_S(x)-\bar{h}(x)\right)^2\right]$$

$$expected\ error \qquad noise \qquad bias^2 \qquad variance$$

Bagging is an ensemble approach aimed at reducing variance without decreasing complexity. The idea is to learn an H_S that tries to approximate $\bar{h}$. Bagging works well for high-variance, low bias models. Bagging uses linear aggregation of models with uniform weights.

Bagging reduces variance by averaging. According to the law of large numbers:

1. $\bar{h}_k = \frac{1}{k}\sum_{i\in[k]} h_i \to^{k\to\infty} \mu$
2. $Var\left(\bar{h}(x)\right) = \frac{\sigma^2}{k}$

To create many different models h_i we need to simulate many sample sets. Bagging uses Bootstraping for that. Bootstraping = sample uniformly with replacement. Bootstrapping can be applied to any statistic to create a histogram of what might happen if we repeated the experiment many times.

Bagging:

1. Sample $S\sim D^m$
2. Define dist. Uniform over S: $Q\left((x,y)\middle|S\right) = \begin{cases}\frac{1}{m}, & (x,y)\in S\\ 0, & otherwise\end{cases}$
3. Repeat for $i \in [k]$: (k number of models)
   a. Sample m samples: $S_i = Q^m\ iid$
   b. Train $h_i = A(S_i)$
4. Aggregate: $H = agg(h_1,\dots,h_k)$

Bagging = Bootstrap Aggregating

Overall, $P_Q\left(x_i \notin S_j\ after\ m\ tries\right) = \left(1-\frac{1}{m}\right)^m \to e^{-1} = 0.368$ → around $\frac{1}{3}$ of the data is never included, no matter how big m is! → Every h_i is trained over approx. $\frac{2}{3}$ of the data, but of different splits.

Boosting

Mostly aimed at reducing bias. It therefore targets high-bias models and weak base learners.

Boosting uses linear aggregation with varying weights.

The base class H is a γ-weakly learnable if exists A and $m(\delta)$ s.t for all $\delta \in [0,1]$ and all D, for $S \sim D^m$: $P_D\left(err\left(A(S)\right) \leq 0.5 - \gamma\right) \geq 1 - \delta$.

Boosting uses a weak learner to form a strong learner. The algorithm to the right presents the main idea of Boosting. Each h_t compensates for errors made by its predecessors in H_{t-1}.

- **Main idea** – iteratively build aggregate model:
  - for $t = 1 \dots T$
    - learn h_t with weak learner A
    - set α_t
    - update $H_t = H_{t-1} + \alpha_t h_t$
  - return $H = H_T = \sum_{t=1}^{T} \alpha_t h_t$

AdaBoost = adaptive boosting:

1. Initialize $D_0 = \left(\frac{1}{m}, \dots, \frac{1}{m}\right)$ #vector of weights, begins equally
2. For $t = 1 \dots T$:
   a. $h_t = A(S; D_t)$ #Train with weak learner
   b. $\epsilon_t = \sum_i D_{t,i} \cdot 1\{y_i \neq h_t(x_i)\}$ #Compute weighted 0/1 error
   c. $\alpha_t = \frac{1}{2}\log(\frac{1}{\epsilon_t} - 1)$ #Set model coefficients
   d. $\forall i, D_{t+1,i} \propto D_{t,i} \cdot \exp\{-\alpha_t y_i h_t(x_i)\}$ #Update normalized weights
3. Return $H(x) = sign(\sum_{t \in T} \alpha_t h_t)$

Notes:

- AdaBoost's training error after T iterations is bounded by: $L_S(h_S) \leq \exp\{-\gamma^2 T\}$.
- AdaBoost reaches zero training error eventually.
- AdaBoost greedily optimizes the exponential loss.
- Ensemble model class is a linear combination of basis models:
  $\{h_{\alpha.h} = \sum_{t \in [T]} \alpha_t h_t : \alpha_t \in \mathbb{R}, h_t \in \mathcal{H}\}$
- AdaBoost is an instance of Gradient Boosting (below). The loss is exponential and updating D approximates gradient step. α_t is the optimal step size. AdaBoost converges *exponentially* fast.

Bagging vs Boosting

Bagging:

- mostly aimed at reducing variance
- targets high-variance models
  e.g., deep trees
- linear aggregation, uniform weights:

$$H = \sum_{i=1}^{k} \frac{1}{k} h_i$$

- parallel training
  (h_i independent given S)

Boosting:

- mostly aimed at reducing bias
- targets high-bias models
  e.g., stumps (depth-one trees)
- linear aggregation, varying weights:

$$H = \sum_{i=1}^{k} \alpha_i h_i$$

- incremental (greedy) training
  (h_i depends on $h_1, \ldots, h_{i-1}$)

Gradient Boosting

Idea – do GD in *function space*: $H_t = H_{t-1} + \alpha \nabla L_S(H_{t-1})$

This requires the basis model class $\mathcal{H}$ to support inner products and induces a norm. We can now think of GD as learning an "ensemble":

$$\hat{w} = \sum_{t \in [T]} \alpha_t h_t \ , \quad h_t = \nabla L_S(w_t), \quad \alpha_t = -\eta$$

The learned ensemble model is a member of the base class.

Gradient Boosting:

1. Compute "empirical" gradient, with entries only for observed data points:
   $g_i = \nabla L_S(H_{t-1})_{x_i} = \frac{\partial l(y_i, H_{t-1}(x_i))}{\partial H_{t-1}(x_i)}$
2. Find $h_t \in \underset{h \in \mathcal{H}}{\operatorname{argmin}} \sum_i l(g_i, h(x_i))$
3. Take optimal space : $H_t = \underset{h \in \mathcal{H}, \alpha \in \mathbb{R}}{\operatorname{argmin}} (H_{t-1} + \alpha h_t)$

   - **Example**: squared loss, $\ell(y, \hat{y}) = \frac{1}{2}(\hat{y} - y)^2$
   - $g_i = [\nabla L_S(H_{t-1})]_{x_i} = \frac{\partial L_S(H_{t-1})}{\partial H_{t-1}(x_i)} = \frac{\partial \ell(y_i, H_{t-1}(x_i))}{\partial H_{t-1}(x_i)} = \frac{\partial \frac{1}{2}(H_{t-1}(x_i) - y_i)^2}{\partial H_{t-1}(x_i)} = H_{t-1}(x_i) - y_i$
   - **Notice**: $-g_i = y_i - H_{t-1}(x_i) = y_i - \hat{y}_i = r_i$ ← *residual!*

➔ For squared loss, gradient boosting iteratively compensates for residual errors.

Deep Learning

In deep learning we learn both h_i, w_i jointly end-to-end.

Objective:

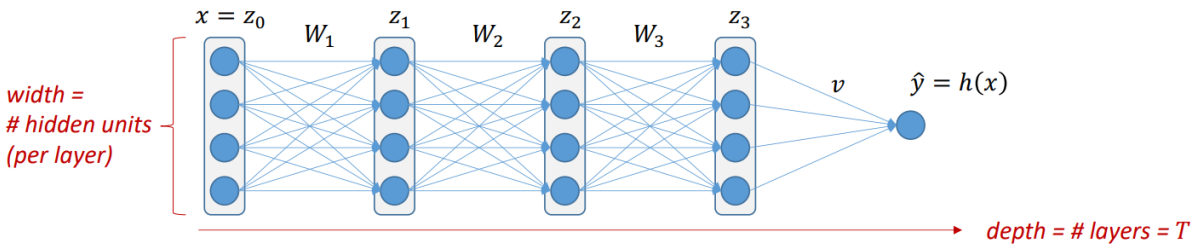
$$\underset{v,W_t}{\operatorname{argmin}} \frac{1}{m}\sum_{i\in[m]} l(y_i, h(x_i)),\ \ where:\ h(x) = \sigma\big(v^T \phi_T(x)\big),\ \ \phi_T(x) = \sigma\big(W_t \phi_{t-1}(x)\big), \phi_0(x) = x$$

- Loss is typically cross-entropy
- Objective is non-convex

$$h(x) = \sigma\big(v^\top \phi_T(x)\big), \qquad \phi_t(x) = \sigma\big(W_t \phi_{t-1}(x)\big), \qquad \phi_0(x) = x$$

- or -

$$h(x) = \sigma(v^\top \sigma(W_T \sigma(W_{T-1} \sigma(\dots \sigma(Wx)))))$$



In the illustration above, each **edge** from one layer to the other is weighted with w_i . Each **vertice** is a scalar of some value between 0 and 1. When resolving a next vertice of the next layer, we multiply previous vertices with the weights of edges connecting to the requested vertice. Then, we pass it through a **sigmoid**/tanh/Relu.

Back propagation

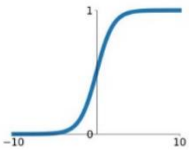
Back Propagation is the algorithm that determines how a single training example "would like to" change the weights of edges so that it'll become closer to its desired label.

Back propagation consists of:

1. Gradient computations
2. Matrix/tensor operations

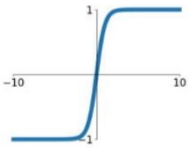
sigmoid:

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



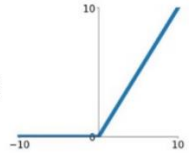
tanh:

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



ReLU:

$$\sigma(x) = \max\{0, x\}$$



backward pass:

- $\frac{\partial \ell(h)}{\partial v} \leftarrow \left(y - v^\top h_{T-1}(x)\right) h_{T-1}(x)$
- compute $\delta_{T-1} = \frac{\partial \ell(h)}{\partial a}$
- for $t = T-1, \ldots, 1$:
  - $\frac{\partial \ell(h)}{\partial W_t} \leftarrow \delta_t h_{t-1}(x)$ # $h_0(x) = x$
  - $\delta_{t-1} = \delta_t \frac{\partial a_t}{\partial a_{t-1}}$

Back prop is usually done over a mini batch to reduce computation time and overcome noise (SGD).

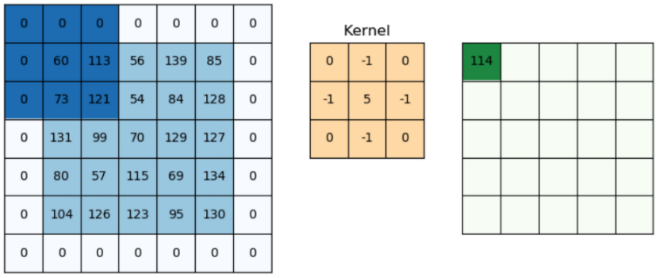
More on backpropagation, and of its chain-rule calculus, another video.

Convolutions

Instead of connecting each neuron to all the previous ones, Convolutional Neural Networks use a fraction of possible connections. These are called convolutional layers. Each of these layers has some size of "filter" or "kernel". This filter is usually a matrix of some size which imposes which neurons to rely upon when calculating the next neurons. This reduces the number of learnable parameters and in a way preserves input structure and invariance.

CNNs are useful for image processing because they detect patterns in an image. The deeper our network, the more complicated those patterns become.

For more: Convolutional Neural Networks.



Generalization

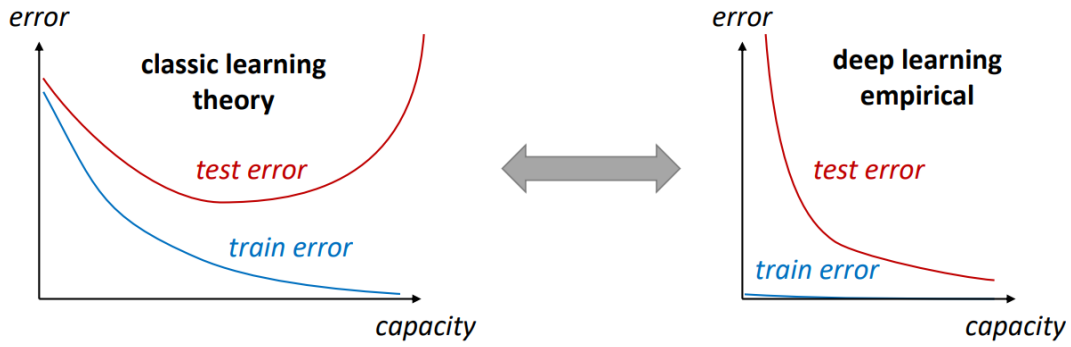
Let H_{NN} be class of sigmoidal-activation neural networks with N units and M weights. Then:

$$M^2 \leq VC(H_{NN}) \leq N^2 M^2$$

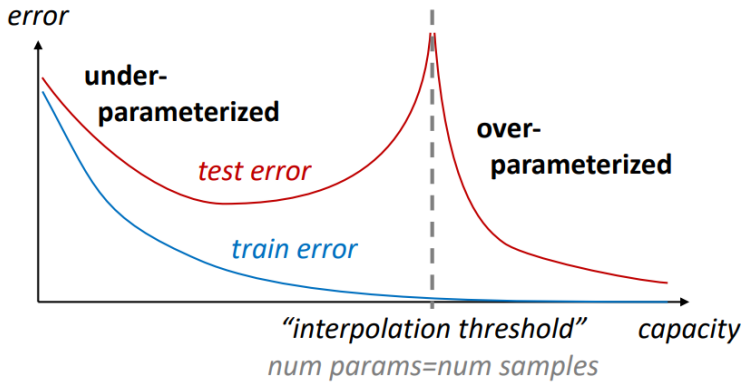
Conclusion: carefully control overfit by:

1. Regularization of weights (not popular)
2. Early stopping
3. Drop-out: at each SGD step, temporarily discard a random subset of weights
   a. Prevents network from relying extensively on specific weights
   b. Prevents network from being tailored to training data
   c. Can be thought of as implicitly training an ensemble of network models.

In practice, neural networks don't necessarily overfit and empirically only getting better. This contradicts theory:



Recent studies suggest maybe:



Unsupervised learning

In unsupervised learning, the labels of the samples are *unobserved/nonexistent.* It is useful for:

- Dimensionally reduction
- Manifold learning
- Data interpolation and generation

Dimensionality Reduction

Principal component analysis (PCA) is a linear process of mapping data to new coordinate system with dimensions ordered from high variance to low. Each dimension is called a principal component. In PCA, the linear mapping must be orthonormal: $\phi(x) = U^T x,\ U\ is\ orthonormal$

Orthogonal – so that no overlap in variance.

Normal – so that variance is same-scaled.

PCA finds linear transformations minimizing the reconstruction MSE, ignoring the labels:

$$\underset{U,V\in\mathbb{R}^{dXd}}{\operatorname{argmin}} \frac{1}{m}\sum_{i\in[m]}(x_i - VU^T x_i)^2$$

Note: For any orthonormal U, $V = U$ is optimal.

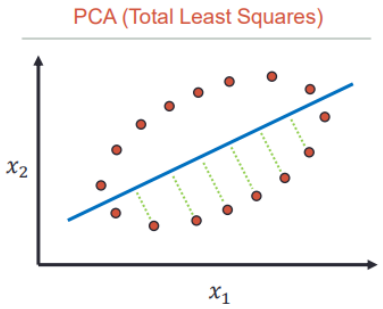
PCA will work when interactions among variables are linear. For nonlinear data, one could use kernel-PCA or LLE instead.

Relation to least squares:

- The PCA formulation:

$$\underset{\mathbf{U},\mathbf{V}\in\mathbb{R}^{d\times k}}{\operatorname{argmin}} \frac{1}{m}\sum_{i=1}^{m} \|\boldsymbol{x}_i - \underbrace{\mathbf{V}\mathbf{U}^\top \boldsymbol{x}_i}_{\hat{\boldsymbol{x}}_i}\|_2^2$$

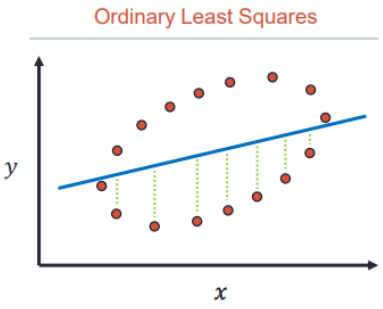
- Tries to compress $\boldsymbol{x}_i$
- Minimizes reconstruction errors



- The least squares formulation:

$$\underset{\boldsymbol{w}\in\mathbb{R}^{d}}{\operatorname{argmin}} \frac{1}{m}\sum_{i=1}^{m} \|y_i - \underbrace{\boldsymbol{w}^\top \boldsymbol{x}_i}_{\hat{y}_i}\|_2^2$$

- Tries to explain y_i using $\boldsymbol{x}_i$
- Minimizes prediction errors



Reconstruction

Reconstruction is the process of returning to the original samples from reduced-dimension ones. This is done by minimizing the same objective but replacing $VU^T x_i$ with nonlinear vector mappings $\psi(\phi(x_i))$. When this is implemented by a neural network, this is called Autoencoder model.

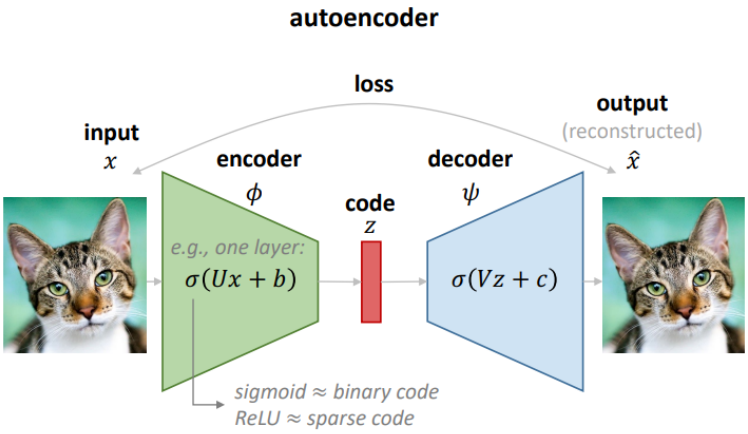
Autoencoder:

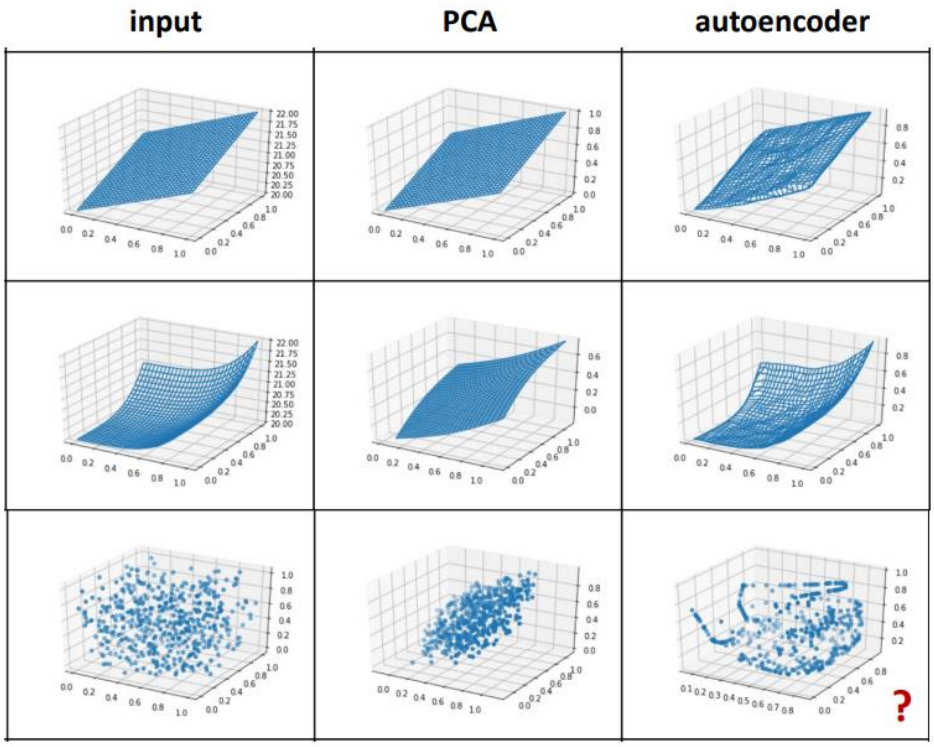
Objective: $\underset{\psi,\phi}{\operatorname{argmin}} \sum_{i\in[m]} \|x_i - \psi(\phi(x_i))\|_2^2$

Manifold: a nonlinear topological space that locally resembles Euclidian space.

Hypothesis: real data lies on a manifold.

Autoencoders can be thought if as learning the non-linear manifold structure.





We can make use of the manifold structure. The idea is:

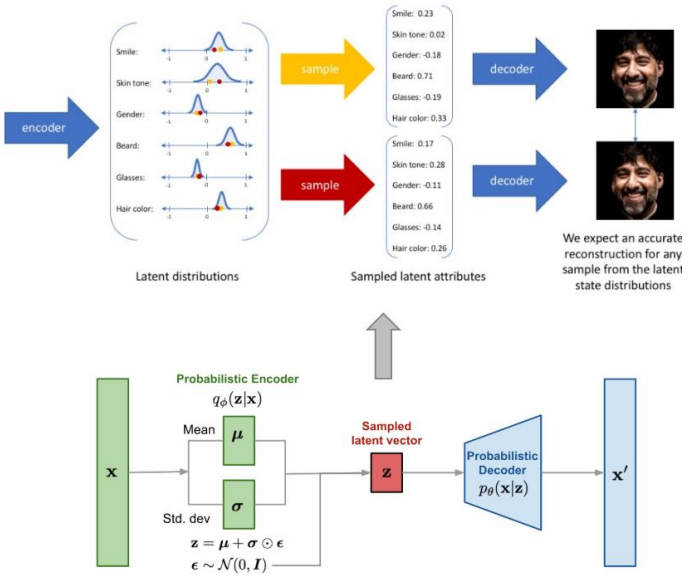
1. Take 2 close points in embedded space z_1, z_2
2. Interpolate: $z = \alpha z_1 + (1-\alpha) z_2$
3. Map back to ambient space: $x' = \psi(\phi(z))$

If $\psi \circ \phi$ captures manifold structure, then x' should be <u>semantically meaningful</u>.

Variational Autoencoders (VAEs):

Replace functions with distributions: $\phi \to q(z|x)$, $\psi \to p(x|z)$.

We learn a deterministic map $x \longmapsto (\mu, \sigma)$ and add noise $\epsilon \sim N(0, I)$.



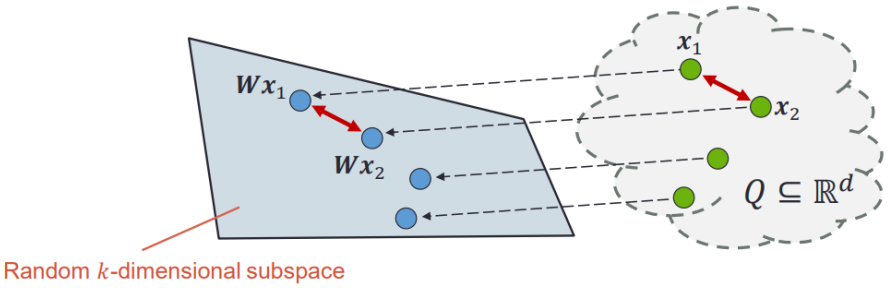
Random projections:

Random projections project onto a random k-dimensional subspace. They are easier to compute, and don't distort Euclidean distances that much:

Johnson-Lindenstrauss lemma suggest roughly: $\forall i \neq j$: $\|Wx_i - Wx_j\|^2 \approx \|x_i - x_j\|^2$

Johnson-Lindenstrauss lemma: more formal

- Sample a matrix $\boldsymbol{W} \in \mathbb{R}^{k \times d}$ such that $\boldsymbol{W}_{i,j} \sim \mathcal{N}(0, 1/k)$.



- Given a set $Q \subseteq \mathbb{R}^d$ and two numbers $\delta \in (0,1)$, $\epsilon = \sqrt{\frac{12}{k}\ln(2|Q|/\delta)} \leq 3$, then with probability of at least $1 - \delta$ over a choice of $\boldsymbol{W}$, we have

$$\forall x_i, x_j \in Q: \ (1-\epsilon)\|x_i - x_j\|^2 \leq \|Wx_i - Wx_j\|^2 \leq (1+\epsilon)\|x_i - x_j\|^2.$$

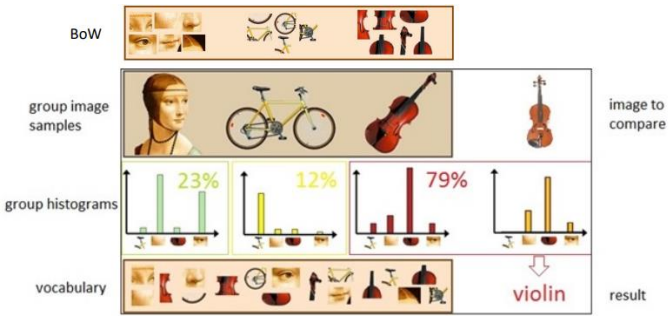
Extras

Objects Representation

Some objects aren't easily defined as vectors. How can we learn anything about these objects?

Bag of Words (BoW): is a simple approach to this issue. The concept is to represent the object (text document, for instance) with a histogram of "word" frequencies.

- Pros:
  - Effectively vectorizes text
  - Utilizes basic statistical linguistic properties
  - Often works well
- Cons:
  - Huge dimension
  - Not utilizes many properties



For images, this method could use a **bag of visual words**.

Set Models: deals with inputs that come in sets (order doesn't matter). Implementing a set function using a vector function requires permutation invariance. The solution is any function of the form: $h(x) = \rho(\sum_i \phi(e_i))$.

Spatial Structure (מבנה מרחבי): deals mostly with graph bases inputs. What are good representations for graphs?

- Kernels: need $\phi: G \to \mathbb{R}^k \; s.t \;\; K(g, G') = \langle \phi(G), \phi(G') \rangle$:
  - Is a valid inner product
  - Captures meaningful similarities between graphs
  - Can be computed efficiently

  Something like:

  - Graphlet kernel
  - Shortest-paths graph kernel
  - Random walk graph kernel
  - WL graph kernel

- Graphs Neural Networks (GNNs):
  - Aim to generalize convolution from grids to graphs
  - Some more info.

Multiclass Classification

Combine multiple learned binary predictors to form a single multiclass predictor that predicts a class distribution.

Model: $h: \mathcal{X} \to \mathcal{Y}$

Loss: $l: \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$

One vs All:

- **One vs. All (1vA):**
  - for $i = 1, \dots, K$
    - construct sample set $S_i = \left\{\left(x_j, y_j^{(i)}\right)\right\}_{j=1}^{m}$ where $y_j^{(i)} = \mathbb{1}\{y_j = i\}$
    - train binary predictor $h_i = A(S_i)$
  - return multiclass predictor $h(x) = \underset{i \in [K]}{\operatorname{argmax}}\, h_i(x)$

One vs all algorithm is learning separate problems over the dataset, classifying binary samples at a time. Eventually it returns the best predictor of those.

For more of multiclass classification using logistic regression.

An alternative:

One vs. One:

- **One vs. One (1v1):**
  - for $i = 1, \dots, K$
    - for $j = i + 1, \dots, K$
      - construct sample set $S_{ij} = \{(x_\ell, y_\ell) : y_\ell \in \{i, j\}\}$
      - train binary predictor $h_{ij} = A(S_{ij})$, set $h_{ji} = -h_{ij}$
  - return multiclass predictor $h(x) = \underset{i \in [K]}{\operatorname{argmax}} \sum_{j \neq i} h_{ij}(x)$

In one vs. one classification, we split the data to pairs of labels, i.e., we use only the samples of these two labels each "round". Each round, we train a model only on this binary target dataset. Doing it for every pair of different targets would mean $\frac{N(N-1)}{2}$ different learning processes (N being number of labels in the class). Eventually, we take the best predictor out of those.

More on one vs one and one vs all.

Cross Entropy:

Measures how one distribution differs from another. We use it to create a loss over class distributions:

$$\ell^{\mathrm{CE}}(\boldsymbol{p}, \hat{\boldsymbol{p}}(\boldsymbol{x})) = H(\boldsymbol{p}, \hat{\boldsymbol{p}}(\boldsymbol{x})) = -\sum_{k=1}^{K} p_k \ln \hat{p}_k$$

True distribution — Predicted distribution given x

- Since each example belongs to one class only, the true distribution is "one hot":

$$\ell^{\mathrm{CE}}(y, \hat{\boldsymbol{p}}) = H(\mathrm{onehot(y)}, \hat{\boldsymbol{p}}) = -\ln \hat{p}_y$$

Multinomial logistic regression

Multinomial regression trains linear separators jointly. It creates a class distribution using softmax and train using cross-entropy loss.

Models the distribution of all classes given x_i:

$$y_i | x_i, w \sim Multinomial(\hat{p}_1, \dots, \hat{p}_k)$$

Note: the multinomial distribution models the probability of counts for each class, using a constant probability $\hat{p}_i$ as the probability of observing a sample of class i → $\sum_{i\in[k]} \hat{p}_i = 1$.

Trains a linear classifier $w_j \in \mathbb{R}^d$ for each class. The multinomial distribution given x_i:

1. score each class using $w_k^T x_i$
2. Turn scores into a normalized distribution (softmax): $\hat{p}_j(x) = \frac{e^{w_j^T x_i}}{\sum_j e^{w_j^T x_i}}$
3. Uses the cross-entropy loss: $\underset{w_1,\dots,w_k}{\operatorname{argmin}} \sum_i -ln\, \hat{p}_{y_i}(x) = \underset{w_1,\dots,w_k}{\operatorname{argmin}} \sum_i -ln \frac{e^{w_j^T x_i}}{\sum_j e^{w_j^T x_i}}$

The gradient of the cross-entropy loss w.r.t each vector w_k is:

$$\nabla_{w_k} loss_{CE}(y_i, \hat{p}(x_i)) = (-[k = y_i] + \hat{p}_k(x_i))x_i$$

Note: the loss is convex w.r.t $w_1, w_2, \dots, w_k$.

More on multinomial logistic regression.

Feature Selection

Our goal is to learn a predictor that only relies on $k \ll d$ features. Predictors that use only a small subset of features require a smaller memory footprint and can be applied faster. Furthermore, in applications such as medical diagnostics, obtaining each possible "feature" (e.g., test result) can be

costly; therefore, a predictor that uses only a small number of features is desirable even at the cost of a small degradation in performance, relative to a predictor that uses more features. Finally, constraining the hypothesis class to use a small subset of features can reduce its estimation error and thus prevent overfitting.

There are multiple ways to decrease the number of features to use in the learning process:

1. **Filters**: Maybe the simplest approach for feature selection is the filter method, in which we assess individual features, independently of other features, according to some quality measure. We can then select the k features that achieve the highest score.
   For instance, applying a linear classifier with only one feature at a time and take the k features with the lowest error rate.
2. **Greedy Selection- forward:** The simplest instance of greedy selection is forward greedy selection. We start with an empty set of features, and then we gradually add one feature at a time to the set of selected features. Given that our current set of selected features is I, we go over all $i \notin I$, and apply the learning algorithm on the set of features $I \cup \{i\}$. Each such application yields a different predictor, and we choose to add the feature that yields the predictor with the smallest risk (on the training set or on a validation set). This process continues until we either select k features, where k is a predefined budget of allowed features, or achieve an accurate enough predictor.
3. **Greedy Selection- backward:** Another popular greedy selection approach is backward elimination. Here, we start with the full set of features, and then we gradually remove one feature at a time from the set of features. Given that our current set of selected features is I, we go over all $i \in I$, and apply the learning algorithm on the set of features $I \backslash \{i\}$. Each such application yields a different predictor, and we choose to remove the feature i for which the predictor obtained from $I \setminus \{i\}$ has the smallest risk (on the training set or on a validation set).
4. **Sparsity-Inducing Norms:** we want w to be sparse, which implies that we only need to measure the features corresponding to nonzero elements of w. As was explained in Lasso regression, the L1 norm resolves in sparser solutions. Since it is also a convex function, learning problems can be solved efficiently if the loss function is convex.