

CS-GY 6923: Lecture 7

Learning Rates, Stochastic Gradient Descent, Taste of Learning Theory, PAC learning

NYU Tandon School of Engineering, Prof. Christopher Musco

First order oracle model: Given a function L to minimize (in our case a loss function), assume we can:

- **Function oracle:** Evaluate $L(\beta)$ for any β .
- **Gradient oracle:** Evaluate $\nabla L(\beta)$ for any β .

Basic Gradient descent algorithm:

- Choose starting point $\beta^{(0)}$.
- For $i = 1, \dots, T$:
 - $\beta^{(i+1)} = \beta^{(i)} - \eta \nabla L(\beta^{(i)})$
- Return $\beta^{(t)}$.

$\eta > 0$ is a step-size parameter. Also called the learning rate.

We did not discuss how to set η in practice, but this is a very important issue in optimization for machine learning.

We have

$$\begin{aligned}\lim_{\eta \rightarrow 0} L(\boldsymbol{\beta} - \eta \mathbf{v}) - L(\boldsymbol{\beta}) &= -\eta \cdot \left(\frac{\partial L}{\partial \beta_1} v_1 + \frac{\partial L}{\partial \beta_2} v_2 + \dots + \frac{\partial L}{\partial \beta_d} v_d \right) \\ &= -\eta \cdot \langle \nabla L(\boldsymbol{\beta}), \mathbf{v} \rangle.\end{aligned}$$

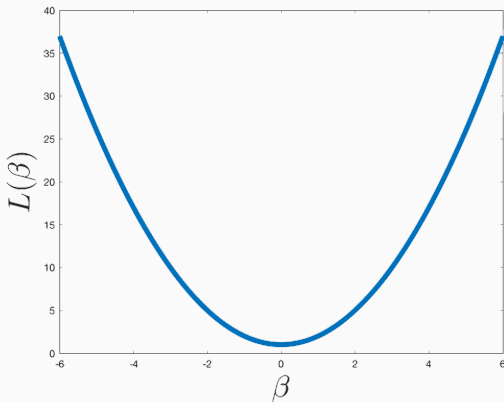
If we set $\mathbf{v} = \nabla L(\boldsymbol{\beta})$, then we make progress.

How to set η in practice?

- Too large, and the above claim doesn't hold, so we don't make progress.
- Too small, and we converge slowly.

LEARNING RATE

Precision in choosing the learning rate η is not super important, but we do need to get it to the right order of magnitude.



CONVERGENCE ANALYSIS FOR CONVEX FUNCTIONS

Assume:

- L is convex.
- Lipschitz function: for all β , $\|\nabla L(\beta)\|_2 \leq G$.
- Starting radius: $\|\beta^* - \beta^{(0)}\|_2 \leq R$.

Gradient descent:

- Choose number of steps T .
- Starting point $\beta^{(0)}$. E.g. $\beta^{(0)} = \mathbf{0}$.
- $\eta = \frac{R}{G\sqrt{T}}$
- For $i = 0, \dots, T$:
 - $\beta^{(i+1)} = \beta^{(i)} - \eta \nabla L(\beta^{(i)})$
- Return $\hat{\beta} = \arg \min_{\beta^{(i)}} L(\beta)$.

This result tells us exactly how to set the learning rate η .

But...

- We don't usually know R or G in advance. We might not even know T .
- Even if we did, setting $\eta = \frac{R}{G\sqrt{T}}$ tends to be a very conservative in practice. The choice 100% leads to convergence, but usually to fairly slow convergence.
- What if L is not convex?

FIRST APPROACH: EXPONENTIAL GRID SEARCH

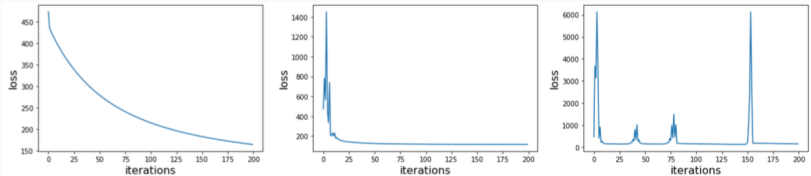
Just as in regularization, search over a grid of possible parameters:

$$\eta = [2^{-5}, 2^{-4}, 2^{-3}, \dots, 2^9, 2^{10}].$$

Can manually check if we are converging too slow or undershooting by plotting the optimization curve.

LEARNING RATE

Plot's of loss vs. number of iterations for three difference choices of step size.



Recall: If we set $\beta^{(i+1)} \leftarrow \beta^{(i)} - \eta \nabla L(\beta^{(i)})$ then:

$$\begin{aligned} L(\beta^{(i+1)}) &\approx L(\beta^{(i)}) - \eta \left\langle \nabla L(\beta^{(i)}), \nabla L(\beta^{(i)}) \right\rangle \\ &= L(\beta^{(i)}) - \eta \|\nabla L(\beta^{(i)})\|_2^2. \end{aligned}$$

Approximation holds for small η . If it holds, maybe we could get away with a larger η . If it doesn't, we should probably reduce η .

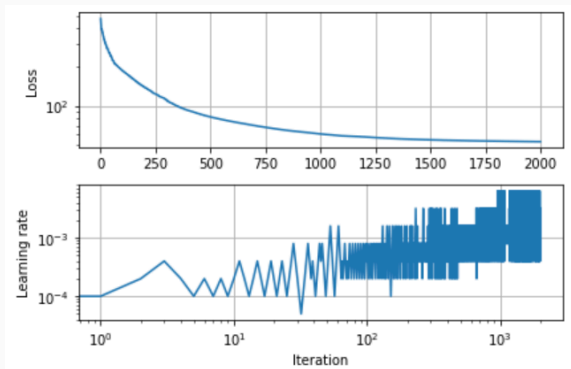
Gradient descent with backtracking line search:

- Choose arbitrary starting point β .
- Choose starting step size η .
- Choose $c < 1$ (typically both $c = 1/2$)
- For $i = 1, \dots, T$:
 - $\beta^{(new)} = \beta - \eta \nabla L(\beta)$
 - If $L(\beta^{(new)}) \leq L(\beta) - c \cdot \eta \nabla L(\beta)$
 - $\beta \leftarrow \beta^{(new)}$
 - $\eta \leftarrow 2\eta$
 - Else
 - $\eta \leftarrow \eta/2$

Always decreases objective value, works very well in practice.

BACKTRACKING LINE SEARCH/ARMIJO RULE

Gradient descent with backtracking line search:



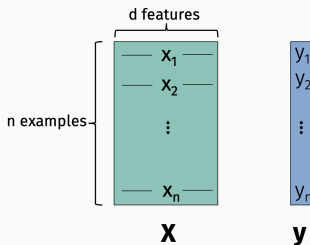
Always decreases objective value, works very well in practice. We will see this in a lab.

COMPLEXITY OF GRADIENT DESCENT

Complexity of computing the gradient will depend on you loss function.

Example 1: Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be a data matrix.

$$L(\beta) = \|\mathbf{X}\beta - \mathbf{y}\|_2^2 \quad \nabla L(\beta) = 2\mathbf{X}^T (\mathbf{X}\beta - \mathbf{y})$$



- Runtime of closed form solution $\beta^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$:
- Runtime of one GD step:

Complexity of computing the gradient will depend on you loss function.

Example 1: Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be a data matrix.

$$L(\boldsymbol{\beta}) = - \sum_{i=1}^n y_i \log(h(\boldsymbol{\beta}^T \mathbf{x}_i)) + (1 - y_i) \log(1 - h(\boldsymbol{\beta}^T \mathbf{x}_i))$$

$$\nabla L(\boldsymbol{\beta}) = \mathbf{X}^T (h(\mathbf{X}\boldsymbol{\beta}) - \mathbf{y})$$

- No closed form solution.
- Runtime of one GD step:

Frequently the complexity is $O(nd)$ if you have n data-points and d parameters in your model. This will also be the case for neural networks.

Not bad, but the dependence on n can be a lot! n might be on the order of thousands, or millions, or trillions.

Stochastic Gradient Descent (SGD).

- Powerful randomized variant of gradient descent used to train machine learning models when n is large and thus computing a full gradient is expensive.

Applies to any loss with finite sum structure:

$$L(\beta) = \sum_{j=1}^n \ell(\beta, \mathbf{x}_j, y_j)$$

Let $L_j(\beta)$ denote $\ell(\beta, \mathbf{x}_j, y_j)$.

Claim: If $j \in 1, \dots, n$ is chosen uniformly at random. Then:

$$\mathbb{E} [n \cdot \nabla L_j(\beta)] = \nabla L(\beta).$$

$\nabla L_j(\beta)$ is called a **stochastic gradient**.

SGD iteration:

- Initialize $\beta^{(0)}$.
- For $i = 0, \dots, T - 1$:
 - Choose j uniformly at random from $\{1, 2, \dots, n\}$.
 - Compute stochastic gradient $\mathbf{g} = \nabla L_j(\beta^{(i)})$.
 - Update $\beta^{(t+1)} = \beta^{(t)} - \eta \cdot n\mathbf{g}$

Move in direction of steepest descent in expectation.

Cost of computing \mathbf{g} is independent of n !

Example: Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be a data matrix.

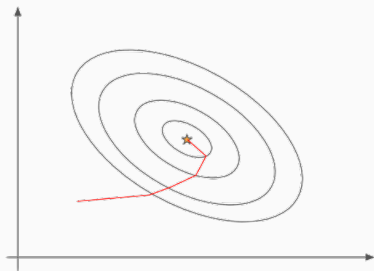
$$L(\boldsymbol{\beta}) = \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 = \sum_{j=1}^n (y_j - \boldsymbol{\beta}^T \mathbf{x}_j)^2$$

- Runtime of one SGD step:

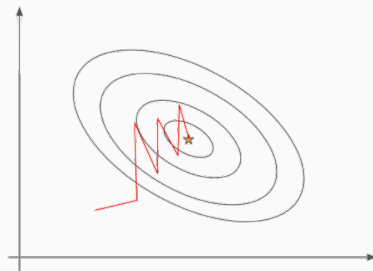
STOCHASTIC GRADIENT DESCENT

Gradient descent: Fewer iterations to converge, higher cost per iteration.

Stochastic Gradient descent: More iterations to converge, lower cost per iteration.



Gradient Descent

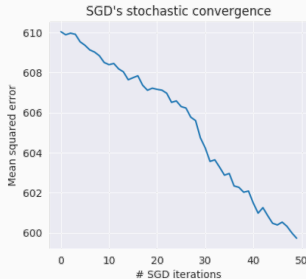
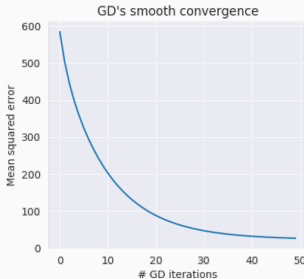


Stochastic Gradient Descent

STOCHASTIC GRADIENT DESCENT

Gradient descent: Fewer iterations to converge, higher cost per iteration.

Stochastic Gradient descent: More iterations to converge, lower cost per iteration.



Typical implementation: Shuffled Gradient Descent.

Instead of choosing j independently at random for each iteration, randomly permute (shuffle) data and set $j = 1, \dots, n$. After every n iterations, reshuffle data and repeat.

- Relatively similar convergence behavior to standard SGD.
- **Important term:** one **epoch** denotes one pass over all training examples: $j = 1, \dots, j = n$.
- Convergence rates for training ML models are often discussed in terms of epochs instead of iterations.

Practical Modification: Mini-batch Gradient Descent.

Observe that for any batch size s ,

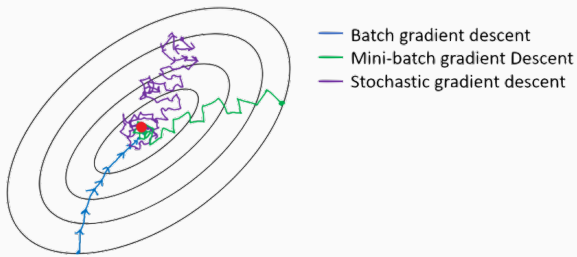
$$\mathbb{E} \left[\frac{n}{s} \sum_{i=1}^s \nabla L_{j_i}(\beta) \right] = \nabla L(\beta).$$

if j_1, \dots, j_s are chosen independently and uniformly at random from $1, \dots, n$.

Instead of computing a full stochastic gradient, compute the average gradient of a small random set (a mini-batch) of training data examples.

Question: Why might we want to do this?

MINI-BATCH GRADIENT DESCENT



- Overall faster convergence (fewer iterations needed).

Practical Mod. 2: Per-parameter adaptive learning rate.

Let $\mathbf{g} = \begin{bmatrix} g_1 \\ \vdots \\ g_p \end{bmatrix}$ be a stochastic or batch stochastic gradient. Our typical parameter update looks like:

$$\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} - \eta \mathbf{g}.$$

We've already seen a simple method for adaptively choosing the learning rate/step size η .

Practical Mod. 2: Per-parameter adaptive learning rate.

In practice, ML lost functions can often be optimized much faster by using “adaptive gradient methods” like Adagrad, Adadelata, RMSProp, and ADAM. These methods make updates of the form:

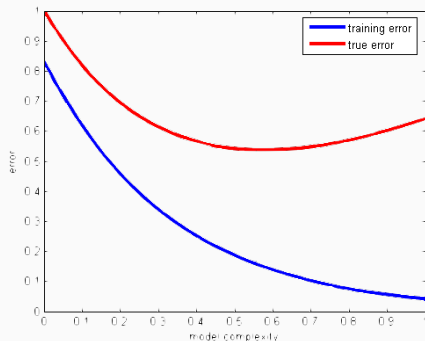
$$\beta_{t+1} = \beta_t - \begin{bmatrix} \eta_1 \cdot g_1 \\ \vdots \\ \eta_d \cdot g_d \end{bmatrix}$$

So we have a separate learning rate for each entry in the gradient (e.g. parameter in the model). And each η_1, \dots, η_p is chosen adaptively.

LEARNING THEORY

THE FUNDAMENTAL CURVE OF ML

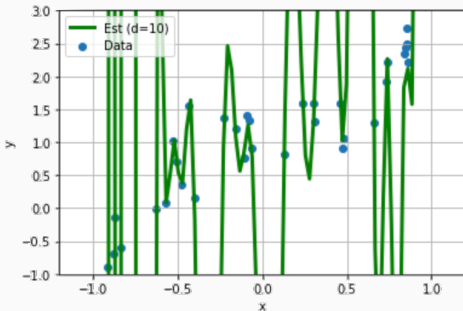
Key Observation: Due to overfitting, more complex models do not always lead to lower test error.



The more complex a model is, the more training data we need to ensure that we do not overfit.

EXAMPLE: POLYNOMIAL REGRESSION

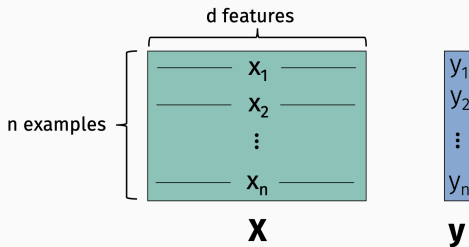
If we want to learn a degree q polynomial model, we will perfectly fit our training data if we have $n \leq q$ examples.



Need $n > q$ samples to ensure good generalization. How much more?

EXAMPLE: LINEAR REGRESSION

If we want to fit a multivariate linear model with d features, we will perfectly fit our training data if we have $n \leq d$ examples.



Need $> d$ samples to ensure good generalization.

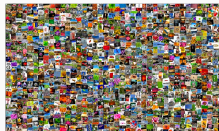
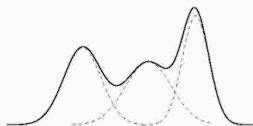
How much more?

Major goal in learning theory:

Formally characterize how much training data is required to ensure good generalization (i.e., good test set performance) when fitting models of varying complexity.

Statistical Learning Model:

- Assume each data example is randomly drawn from some distribution $(\mathbf{x}, y) \sim \mathcal{D}$.



For today: We will only consider classification problems so assume that $y \in \{0, 1\}$.

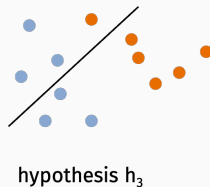
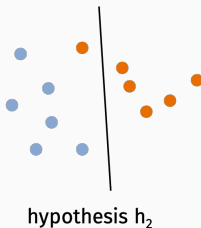
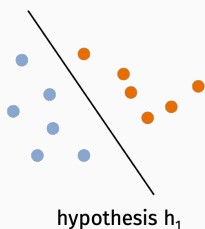
Statistical Learning Model:

- Assume each data example is randomly drawn from some distribution $(\mathbf{x}, y) \sim \mathcal{D}$.
- Assume we want to fit our data with a function h (a “hypothesis”) in some hypothesis class \mathcal{H} . For input \mathbf{x} , $h(\mathbf{x}) \rightarrow \{0, 1\}$.

You can think of h as a model, instantiated with a specific set of parameters. I.e. h is the same as f_{θ} .

EXAMPLE HYPOTHESIS CLASS

Linear threshold functions:



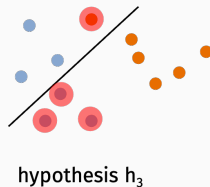
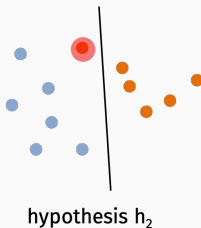
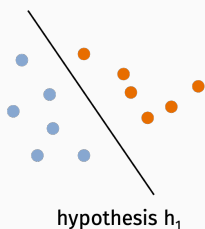
\mathcal{H} contains all functions of the form:

$$h(\mathbf{x}) = \mathbb{1}[\mathbf{x}^T \boldsymbol{\beta} \geq \lambda]$$

.

EXAMPLE HYPOTHESIS CLASS

Linear threshold functions:



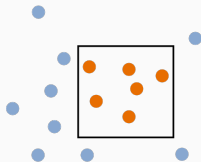
\mathcal{H} contains all functions of the form:

$$h(\mathbf{x}) = \mathbb{1}[\mathbf{x}^T \boldsymbol{\beta} \geq \lambda]$$

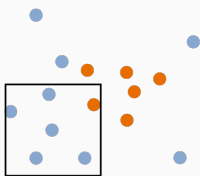
.

EXAMPLE HYPOTHESIS CLASS

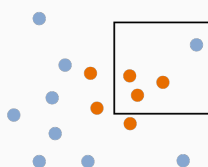
Axis aligned rectangles:



hypothesis h_1



hypothesis h_2



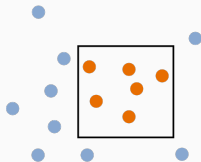
hypothesis h_3

\mathcal{H} contains all functions of the form:

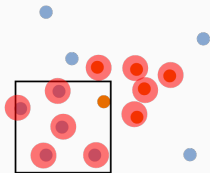
$$h(\mathbf{x}) = \mathbb{1}[l_1 \leq x_1 \leq u_1 \text{ and } l_2 \leq x_2 \leq u_2]$$

EXAMPLE HYPOTHESIS CLASS

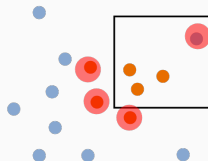
Axis aligned rectangles:



hypothesis h_1



hypothesis h_2



hypothesis h_3

\mathcal{H} contains all functions of the form:

$$h(\mathbf{x}) = \mathbb{1}[l_1 \leq x_1 \leq u_1 \text{ and } l_2 \leq x_2 \leq u_2]$$

Disjunctive Normal Form (DNF) formulas:

Assume $\mathbf{x} \in \{0, 1\}^d$ is binary.

\mathcal{H} contains functions of the form:

$$h(\mathbf{x}) = (x_1 \wedge \bar{x}_5 \wedge x_{10}) \vee (\bar{x}_3 \wedge x_2) \vee \dots \vee (\bar{x}_1 \wedge x_2 \wedge x_{10})$$

\wedge = "and", \vee = "or"

k -DNF: Each conjunction has at most k variables.

Same as “population risk” for the zero one loss:

- Population (“True”) Error:

$$R_{pop}(h) = \Pr_{(\mathbf{x}, y) \sim \mathcal{D}} [h(\mathbf{x}) \neq y]$$

- Empirical Error: Given a set of samples $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m) \sim \mathcal{D}$,

$$R_{emp}(h) = \frac{1}{m} \sum_{i=1}^m \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$$

Goal is to find $h \in \mathcal{H}$ that minimizes population error.

GENERALIZATION

Let $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n) \sim \mathcal{D}$ be our training set and let h_{train} be the empirical error minimizer¹:

$$h_{train} = \arg \min_h \frac{1}{n} \sum_{i=1}^n \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$$

Let h^* be the population error minimizer:

$$h^* = \arg \min_h R_{pop}(h) = \arg \min_h \Pr_{(\mathbf{x}, y) \sim \mathcal{D}} [h(\mathbf{x}) \neq y]$$

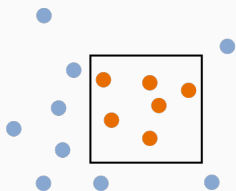
Goal: Ideally, for some small ϵ , $R_{pop}(h_{train}) - R_{pop}(h^*) \leq \epsilon$.

¹Typically we do not *actually* compute h_{train} but rather some approximation based on an easier loss to minimize, e.g. logistic loss.

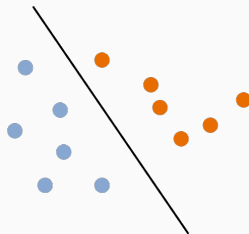
SIMPLIFICATION

Simplification for today: Assume we are in the realizable setting, which means that $R_{pop}(h^*) = 0$. I.e. there is some hypothesis in our class \mathcal{H} that perfectly classifies the data.

Formally, for any (\mathbf{x}, y) such that $\Pr_{\mathcal{D}}[\mathbf{x}, y] > 0$, $h^*(\mathbf{x}) = y$.



hypothesis h^*



hypothesis h^*

Extending to the case when $R_{pop}(h^*) \neq 0$ is not hard, but the math gets a little trickier. And intuition is roughly the same.

Probably Approximately Correct (PAC) Learning (Valiant, 1984):

For a hypothesis class \mathcal{H} , data distribution \mathcal{D} , and training data $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$, let $h_{train} = \arg \min_h \frac{1}{n} \sum_{i=1}^n \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$.

In the realizable setting, how many training samples n are required so that, with probability $1 - \delta$,

$$R_{pop}(h_{train}) \leq \epsilon?$$

The number of samples n will depend on ϵ , δ , and the complexity of the hypothesis class \mathcal{H} . Perhaps surprisingly, it will not depend at all on \mathcal{D} .

Many ways to measure complexity of a hypothesis class. Today we will start with the simplest measure: the number of hypotheses in the class, $|\mathcal{H}|$.

Example: What is the number of hypothesis in the class of 3-DNF formulas on d dimensional inputs

$\mathbf{x} = [x_1, \dots, x_d] \in \{0, 1\}^d$?

$$h(\mathbf{x}) = (x_1 \wedge \bar{x}_5 \wedge x_{10}) \vee (\bar{x}_3 \wedge x_2) \vee \dots \vee (\bar{x}_1 \wedge x_2 \wedge x_{10})$$

Caveat: Many hypothesis classes are infinitely sized. E.g. the set of linear thresholds

$$h(\mathbf{x}) = \mathbb{1}[\mathbf{x}^T \boldsymbol{\beta} \geq \lambda]$$

But you could imagine approximating \mathcal{H} by a finite hypothesis class. E.g. take values in $\boldsymbol{\beta}, \lambda$ to lie on a finite grid of size C . Then how many hypothesis are there?

Formally moving from finite to infinite sized hypothesis classes is a huge area of learning theory (VC theory, Rademacher complexity, etc.)

MAIN RESULT

Consider the realizable setting with hypothesis class \mathcal{H} , data distribution \mathcal{D} , training data set $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$, and $h_{\text{train}} = \arg \min_h \frac{1}{n} \sum_{i=1}^n \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$.

Theorem

If $n \geq \frac{1}{\epsilon} (\log |\mathcal{H}| + \log \frac{1}{\delta})$, then with probability $1 - \delta$,

$$R_{\text{pop}}(h_{\text{train}}) \leq \epsilon.$$

Roughly how many training samples are needed to learn 3-DNF formulas? To learn (discretized) linear threshold functions?

Two ingredients needed for proof:

1. For any $\epsilon \in [0, 1]$, $(1 - \epsilon) \leq e^{-\epsilon}$.
2. **Union bound**. Basic but important inequality about probabilities.

ALGEBRAIC FACT

For any $\epsilon \in [0, 1]$, $(1 - \epsilon) \leq e^{-\epsilon}$.



Raising both sides to $1/\epsilon$, we have the $(1 - \epsilon)^{1/\epsilon} \leq \frac{1}{e} \approx .37$.

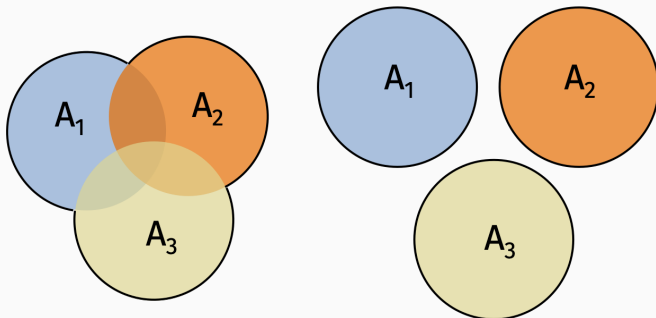
The specific constant here won't be important.

UNION BOUND

Lemma (Union Bound)

For any random events A_1, \dots, A_k :

$$\Pr[A_1 \text{ or } A_2 \text{ or } \dots \text{ or } A_k] \leq \Pr[A_1] + \Pr[A_2] + \dots + \Pr[A_k].$$



Proof by picture.

Union bound is not tight: What is the probability that a dice roll is odd, or that it is ≤ 2 ?

Union bound is tight: What is the probability that a dice roll is 1, or that it is ≥ 4 ?

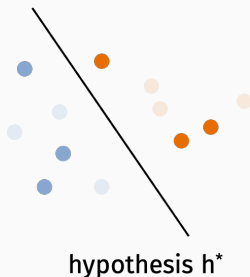
Consider the realizable setting with hypothesis class \mathcal{H} , data distribution \mathcal{D} , training data $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$, and $h_{\text{train}} = \arg \min_h \frac{1}{n} \sum_{i=1}^n \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$.

Theorem

If $n \geq \frac{1}{\epsilon} (\log |\mathcal{H}| + \log \frac{1}{\delta})$, then with probability $1 - \delta$,

$$R_{\text{pop}}(h_{\text{train}}) \leq \epsilon.$$

First observation: Note that because we are in the realizable setting, we always select an h_{train} with $R_{train}(h_{train}) = 0$. There is always at least one $h \in \mathcal{H}$ such that $h(\mathbf{x}_i) = y_i$ for all i .



Proof approach: Show that for any fixed hypothesis h^{bad} with $R_{pop}(h^{bad}) > \epsilon$, it is very unlikely that $R_{train}(h^{bad}) = 0$. So with high probability, we will not choose a bad hypothesis.

Let h^{bad} be a fixed hypothesis with $R_{pop}(h) > \epsilon$. For (\mathbf{x}, y) drawn from \mathcal{D} , what is the probability that $h^{bad}(\mathbf{x}) = y$?

What is the probability that for a training set $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ drawn from \mathcal{D} that $h^{bad}(\mathbf{x}_i) = y_i$ for all i ? I.e. that $R_{train}(h^{bad}) = 0$.

Claim

For any fixed hypothesis h with $R_{pop}(h^{bad}) > \epsilon$, the probability that $R_{train}(h) = 0$ can be bounded by:

$$\Pr[R_{train}(h^{bad}) = 0] < e^{-\epsilon n}.$$

Set $n \geq \frac{1}{\epsilon} \log(|\mathcal{H}|/\delta)$. Then we have that for any fixed hypothesis h^{bad} with $R_{pop}(h^{bad}) > \epsilon$,

$$\Pr[R_{train}(h^{bad}) = 0] < \frac{\delta}{|\mathcal{H}|}.$$

UNION BOUND APPLICATION

Let $h_1^{bad}, \dots, h_m^{bad}$ be all hypothesis in \mathcal{H} with $R_{pop}(h) > \epsilon$.

$$\begin{aligned} & \Pr[R_{train}(h_1^{bad}) = 0 \text{ or } \dots \text{ or } R_{train}(h_m^{bad}) = 0] \\ & \leq \Pr[R_{train}(h_1^{bad}) = 0] + \dots + \Pr[R_{train}(h_m^{bad}) = 0] \\ & < m \cdot \frac{\delta}{\mathcal{H}} \end{aligned}$$

How large can m be? Certainly no more than $|\mathcal{H}|$!

So with probability $1 - \delta$ (high probability) no bad hypotheses have 0 training error. Accordingly, it must be that when we choose a hypothesis with 0 training error, we are choosing a good one. I.e. one with $R_{pop}(h) \leq \epsilon$.

THINGS WE DIDN'T COVER

- How to deal with the non-realizable setting? E.g. where $\min_h R_{pop} \neq 0$?
- How to deal with infinite hypothesis classes (most classes in ML are)?
- How to find $h_{train} = \arg \min_h \frac{1}{n} \sum_{i=1}^n \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$ in a computationally efficient way?

Important take-away as we start working with neural networks and other more complex models:

- We expect the amount of training data required to learn a model to scale logarithmically with the size of the model class being fit, $|\mathcal{H}|$.
- Typically, the size of \mathcal{H} grows exponentially with the number of parameters in the model.
- So overall, our training data size should exceed the number of model parameters (and then some).

I.e., our experience from polynomial regression and linear regression is somewhat universal.