CS-GY 6923: Lecture 7
Learning Rates, Stochastic Gradient Descent, Taste of Learning Theory, PAC Learning

NYU Tandon School of Engineering, Akbar Rafiey
Slides by Prof. Christopher Musco

Recap: first order optimization

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

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

Recap: gradient descent

(Basic) Gradient descent algorithm:

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

$\eta > 0$ is a <u>step-size</u> parameter; also called the <u>learning rate</u>.

Question: How to set η ?

Recap: directional derivatives

We have

$$\lim_{\eta \to 0} L(\boldsymbol{\beta} - \eta \mathbf{v}) - L(\boldsymbol{\beta}) \approx -\eta \cdot \left(\frac{\partial L}{\partial \beta_1} v_1 + \frac{\partial L}{\partial \beta_2} v_2 + \ldots + \frac{\partial L}{\partial \beta_d} v_d \right)$$
$$= -\eta \cdot \langle \nabla L(\boldsymbol{\beta}), \mathbf{v} \rangle.$$

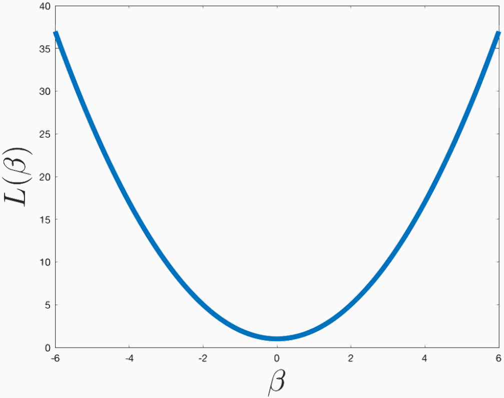
If we set $\mathbf{v} = \nabla L(\boldsymbol{\beta})$, then we make progress.

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 $\boldsymbol{\beta}$, $\|\nabla L(\boldsymbol{\beta})\|_2 \leq G$.
- Starting radius: $\|\boldsymbol{\beta}^* - \boldsymbol{\beta}^{(0)}\|_2 \leq R$.

Gradient descent:

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

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

Setting learning rate

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 (for convex functions), but usually to fairly slow convergence.
- What if L is not convex?

First approach: exponential grid search

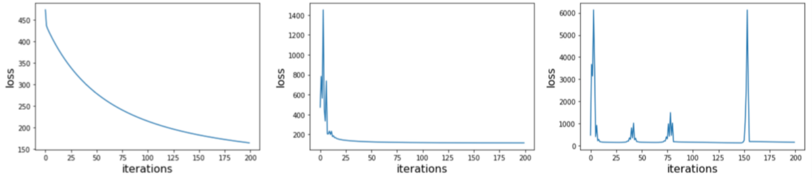
Just as in regularization, search over a grid of possible parameters:

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

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

Learning rate: plotting curves

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



Backtracking line search/Armijo rule

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.

Backtracking line search/Armijo rule

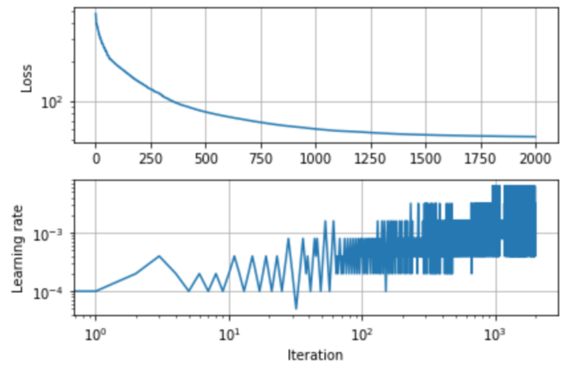
Gradient descent with backtracking line search:

- Choose arbitrary starting point $\boldsymbol{\beta}$.
- Choose starting step size η.
- Choose $c < 1$ (typically both $c = 1/2$)
- For $i = 1, \ldots, T$:
  - $\boldsymbol{\beta}^{(new)} = \boldsymbol{\beta} - \eta \nabla L(\boldsymbol{\beta})$
  - If $L(\boldsymbol{\beta}^{(new)}) \leq L(\boldsymbol{\beta}) - c \cdot \eta \|\nabla L(\boldsymbol{\beta})\|_2^2$
    - $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}^{(new)}$
    - $\eta \leftarrow 2\eta$
  - 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.

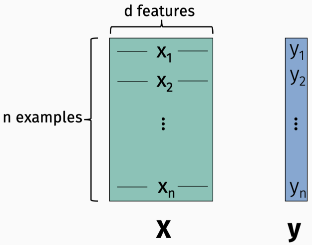
Computationally efficient descent

Complexity of gradient descent

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

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

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



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

Complexity of 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.

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

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

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

Complexity of gradient descent

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:

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

Stochastic gradient descent

- Let $L_j(\boldsymbol{\beta})$ denote $\ell(\boldsymbol{\beta}, \mathbf{x}_j, y_j)$.
- **Claim:** If $j \in 1, \ldots, n$ is chosen uniformly at random. Then:

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

- $\nabla L_j(\boldsymbol{\beta})$ is called a **stochastic gradient**.

Stochastic gradient descent

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 $\beta^{(i+1)} = \beta^{(i)} - \eta \cdot n\mathbf{g}$

Move in direction of steepest descent in expectation.

Cost of computing $\boldsymbol{g}$ is independent of n!

Complexity of stochastic gradient descent

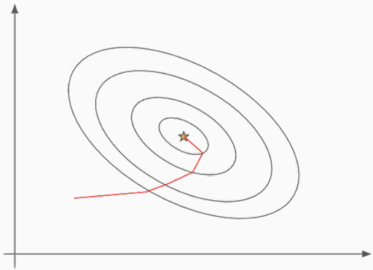
Example: Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be a data matrix.

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

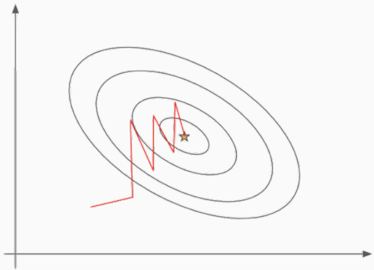
- **Runtime of one SGD step:**

Stochastic gradient descent

- **Gradient descent:** Fewer iterations to converge, higher cost per iteration.
- **Stochastic Gradient descent:** More iterations to converge, lower cost per iteration.



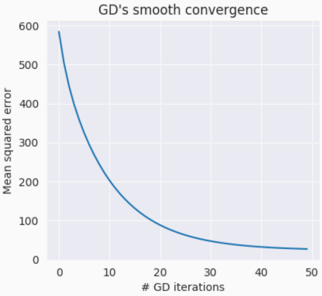
Gradient Descent



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.



Stochastic gradient descent in practice

Typical implementation: **Shuffled Gradient Descent.**

Instead of choosing j independently at random for each iteration, randomly permute (shuffle) data and set $j = 1, \ldots, n$. 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: $j = 1, \ldots, n$.
- Convergence rates for training ML models are often discussed in terms of epochs instead of iterations.

Stochastic gradient descent in practice

Practical Modification: **Mini-batch Gradient Descent.**

Observe that for any batch size s,

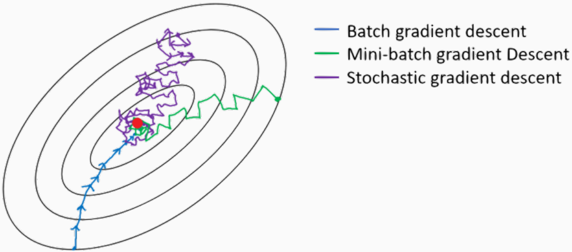
$$\mathbb{E}\left[\frac{n}{s}\sum_{i=1}^{s}\nabla L_{j_i}(\beta)\right] = \nabla L(\beta).$$

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

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).

Stochastic gradient descent in practice

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.

Stochastic gradient descent in practice

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, Adadelta, RMSProp, and ADAM. These methods make updates of the form:

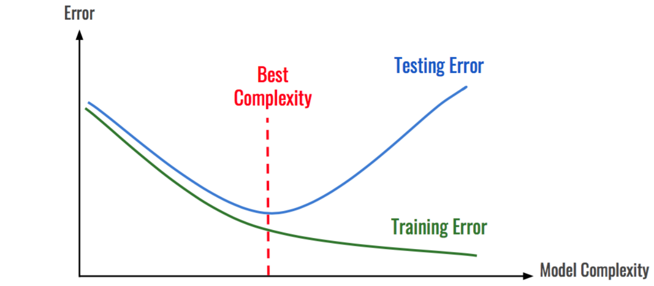
$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t - \begin{bmatrix} \eta_1 \cdot g_1 \\ \vdots \\ \eta_d \cdot g_d \end{bmatrix}$$

So we have a separate learning rate for each entry in the gradient (e.g. parameter in the model); each $\eta_1, \ldots, \eta_p$ is chosen adaptively.

Learning theory

The fundamental curve of ML

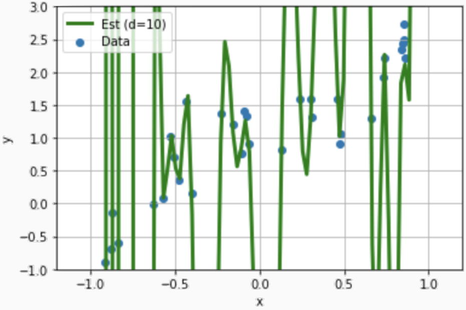
Key Observation: Due to overfitting, more complex models do not always lead to lower test error.



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.

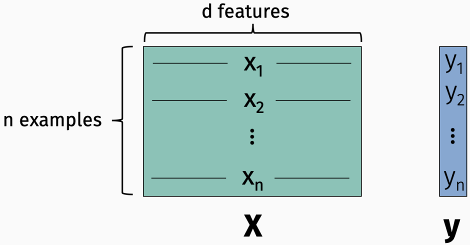


We need $n > q$ samples to ensure good generalization.

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.



We need $n > d$ samples to ensure good generalization.

How much more?

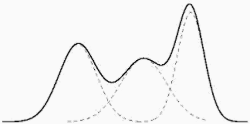
Major goal in statistical learning theory

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

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 $y \in \{0, 1\}$.

Setup

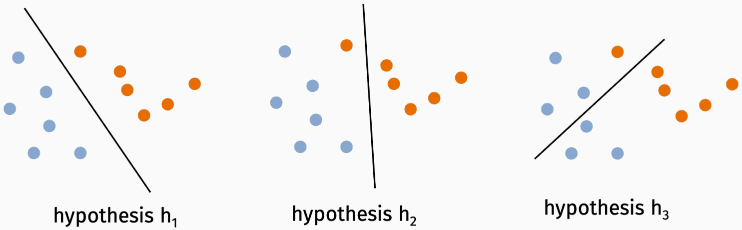
Statistical Learning Model:

- 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:



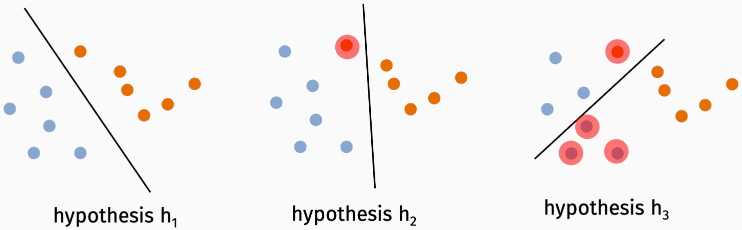
$\mathcal{H}$ contains all functions of the form:

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

.

Example hypothesis class

Linear threshold functions:



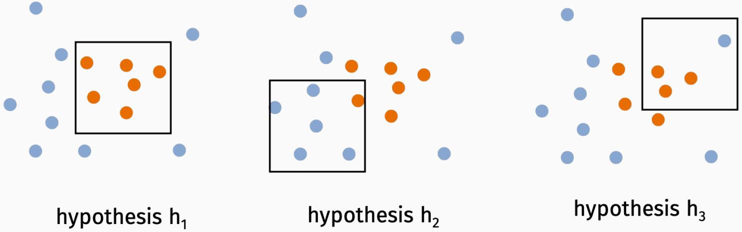
$\mathcal{H}$ contains all functions of the form:

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

.

Example hypothesis class

Axis aligned rectangles:

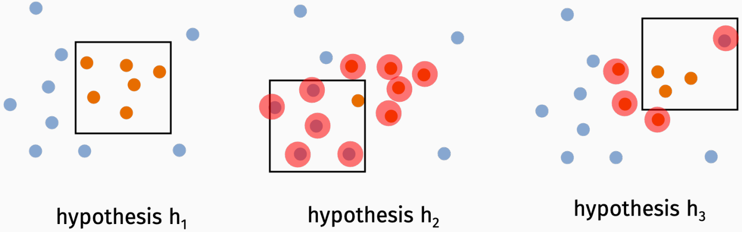


$\mathcal{H}$ contains all functions of the form:

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

Example hypothesis class

Axis aligned rectangles:



$\mathcal{H}$ 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

Disjunctive Normal Form (DNF) formulas:

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

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

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

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

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

Population and empirical error

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 $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n) \sim \mathcal{D}$,

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

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

Generalization

Let $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n) \sim \mathcal{D}$ be our training set and let h_{train} be the empirical error minimizer[1]:

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

Let h^* be the population error minimizer:

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

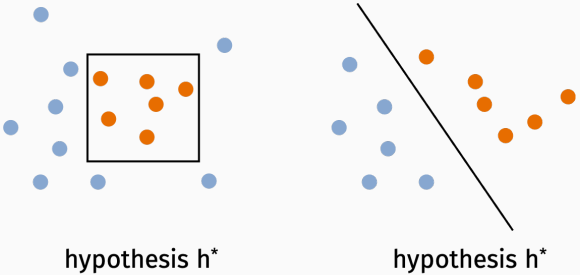
Goal: Ideally, for some small ϵ, $R_{pop}(h_{train}) - R_{pop}(h^*) \leq \epsilon$.

[1]Typically 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 $\Pr_{\mathcal{D}}[\mathbf{x}, y] > 0$, $h^*(\mathbf{x}) = y$.



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.

PAC learning

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

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

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

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

PAC learning

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

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

Some intuitions:

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

Complexity of hypothesis class

Question: How to measure complexity of a hypothesis class ?

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

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

Complexity of hypothesis class

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

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

- We 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 $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$, and $h_{train} = \arg\min_h \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$.

Theorem

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

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

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

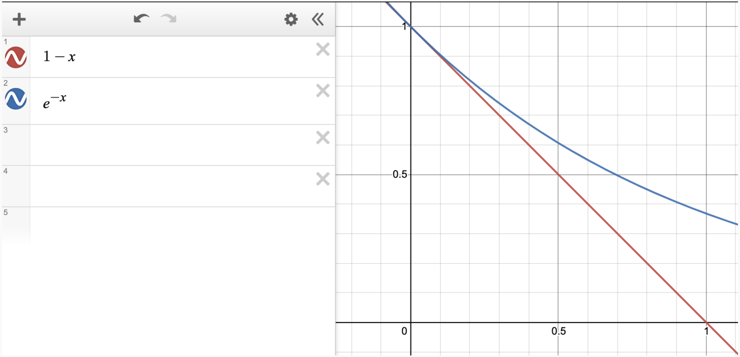
Tools

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 $(1 - \epsilon)^{1/\epsilon} \leq \frac{1}{e} \approx .37$.

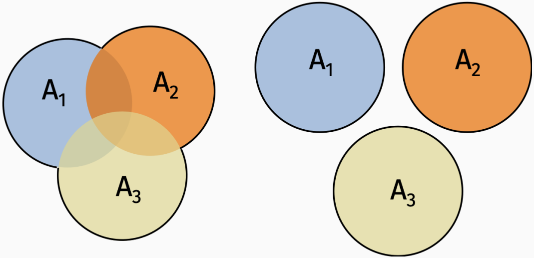
The specific constant here won't be imporatnt.

Union bound

Lemma (Union Bound)

For <u>any</u> random events $A_1, \ldots, A_k$*:*

$$\Pr[A_1 \textit{ or } A_2 \textit{ or } \ldots \textit{ or } A_k] \leq \Pr[A_1] + \Pr[A_2] + \ldots + \Pr[A_k].$$



Proof by picture.

Sometimes written as $\Pr[A_1 \cup A_2 \cup \ldots \cup A_k]$.

Union bound

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?

Main result

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

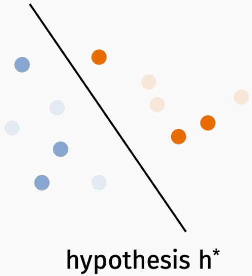
Theorem

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

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

Proof

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

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.

Proof

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 $h^{bad}(\mathbf{x}) = y$?

- at most $(1 - \epsilon)$.

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

- at most $(1 - \epsilon)^n$.

Proof

Claim

For any fixed hypothesis h^{bad} 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 $h_1^{bad}, \ldots, h_m^{bad}$ be all hypthesis in $\mathcal{H}$ with $R_{pop}(h) > \epsilon$.

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

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?

Take away

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.

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

Infinite hypothesis classes

Ideally we would like to give formal results for infinite hypothesis classes (e.g., any class with real valued parameters) without resorting to discretization. One of the most important tools for doing so is the Vapnik–Chervonenkis (VC) dimension.

Theorem

Let $\mathcal{H}$ be a hypothesis class with VC dimension V. If $n \geq \frac{2\log(1/\epsilon)}{\epsilon}\left(\log V + \log\frac{2}{\delta}\right)$, then with probability $1-\delta$,

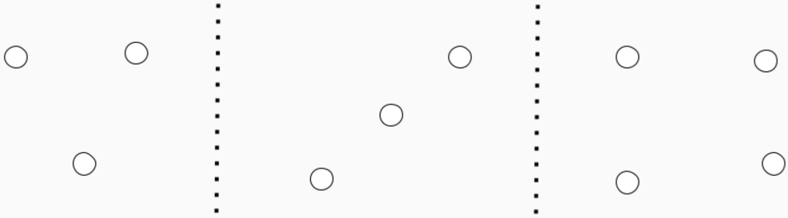
$$R_{pop}(h_{train}) \leq \epsilon.$$

Essentially the same bound as earlier, but $|\mathcal{H}|$ replaced with VC dimension, V.

Shattering

We say a hypothesis class $\mathcal{H}$ shatters a set of points $\mathbf{x}_1, \ldots, \mathbf{x}_q \in \mathbb{R}^d$ if there is some hypothesis $h \in \mathcal{H}$ that matches any possible labeling of the data.

Example: Linear classifiers in $d = 2$ dimensions.



VC dimension

Definition (VC dimension)

The VC dimension of a hypothesis class $\mathcal{H}$ over points in $\mathbb{R}^d$ is the size of the largest point set that $\mathcal{H}$ shatters.

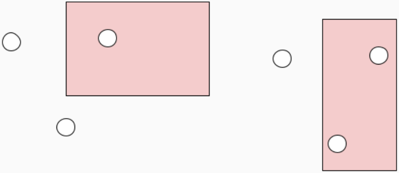
What is the VC dimension of the set of linear classifiers in $d = 2$ dimensions?

VC dimension

Definition (VC dimension)

The VC dimension of a hypothesis class $\mathcal{H}$ over points in $\mathbb{R}^d$ is the size of the largest point set that $\mathcal{H}$ shatters.

What about axis aligned rectangles?



Other important topics

- Generalization of VC dimension to multi-class classification.
- Generalization to regression.
- Tighter bounds that take the distribution $\mathcal{D}$ into account (e.g., via Rademacher complexity).

At the end of the day, the main value of these tools is to improve our understanding of the complexity of different modes/hypothesis classes.

In practice, train/test split is still the major tool for determining if we are overfitting and need more data.