CSCI699: Machine Learning Theory Fall 2023

Lecture 9: SGD, Boosting, Kernels

Instructor: Vatsal Sharan

These lecture notes are based on scribe notes by Di Zhang, Navid Hashemi and Sai Anuroop Kesanapalli.

In this lecture, we will continue to talk about convex optimization. We will then talk about two more powerful algorithmic frameworks in machine learning, boosting and kernels.

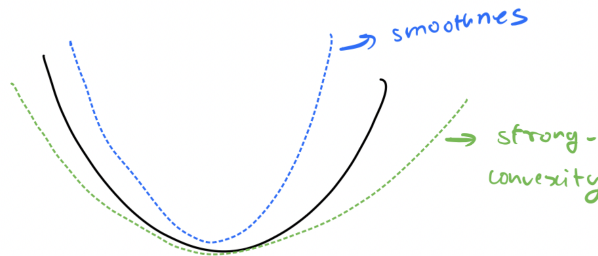
1 Strong convexity

Last time we stated the following bound for smooth, strongly convex functions.

Theorem 1. *Let f be a convex function that is β-smooth and λ-strongly convex. Let $\kappa = \beta/\lambda \geq 1$ be the condition number of the function. Then GD with step size $\eta = \dfrac{2}{\beta+\lambda}$ satisfies*

$$f(w_T) - f(w^*) \leq \exp\left(-\frac{T}{\kappa}\right)\frac{B^2\beta}{2T}.$$

Smoothness and strong convexity give upper and lower bounds on f, respectively, as shown in the figure above. This means that $\beta \geq \lambda$.



In this case, error ϵ can be achieved in $T \geq \kappa \log\left(\dfrac{B^2\beta}{2\epsilon}\right)$ or $\mathcal{O}\left(\kappa \log\left(\dfrac{1}{\epsilon}\right)\right)$ iterations. This is known as a linear convergence in literature since when if we plot the error ϵ on a logarithmic scale, the error reduces linearly with iterations. Note that without strong convexity, the convergence rates that we obtained for GD were $1/\operatorname{poly}(\epsilon)$ ($1/\epsilon$ for smooth, convex functions and $1/\epsilon^2$ for Lipschitz, convex functions). The $\log(1/\epsilon)$ rate is exponentially faster in terms of its dependence on ϵ.

Let us understand this further for quadratic problems (which is really the problem of solving a linear system). In fact, linear systems are always a good canonical setting to try out an optimization algorithm and to get intuition for them.

Example 2. *Consider the following quadratic optimization problem,*

$$\min_{x\in\mathbb{R}^d}\left(\frac{1}{2}x^TAx - bx\right).$$

Note that this function is convex if $A \succcurlyeq 0$, i.e. $v^T A v \geq 0 \; \forall \; v \in \mathbb{R}^d$. The function is strongly-convex if $A \succ 0$, i.e. $v^T A v > 0 \; \forall \; v \in \mathbb{R}^d$. In this case the function also has a simple closed form solution that you can verify by taking the gradient and setting it to 0, and the optimal value for x is $A^{-1}b$. Consider a simple case where A is a diagonal matrix with positive entries, sorted from largest to smallest for simplicity,

$$A = \begin{pmatrix} \lambda_{max} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \lambda_{min} \end{pmatrix}.$$

In this case, the smoothness parameter $\beta = \lambda_{max}$, and the strong convexity parameter $\lambda = \lambda_{min}$.

If $\kappa \approx 1$, this means that the quadratic function has similar curvature in every direction. This is the case where gradient descent will have the fastest rate of convergence. If κ is very large, then this means that there are certain directions which are much flatter than other directions. In this case, gradient descent will take many steps to converge along these flat directions, and the convergence rate will be slow (if we try to increase the step size to speed up convergence along these flat directions, then that step size will be too large for the steeper directions and gradient descent will not converge).

This finishes our discussion for gradient descent. You may wonder if the convergence rates that we showed for gradient descent can be improved. Our analysis of the GD algorithm is tight, but there are modifications which get faster convergence. *Accelerated gradient descent* improves the convergence rate from $O(1/\epsilon) \rightarrow O(1/\sqrt{\epsilon})$ for smooth functions, and $O(\kappa \log(1/\epsilon)) \rightarrow O(\sqrt{\kappa} \log(1/\epsilon))$ for smooth, strongly convex functions. There are also lower bound known for gradient based methods [1], and it is known that accelerated gradient descent is optimal among algorithms whose convergence rates do not depend on the dimensionality d of the problem. How about rates which can depend on the dimensionality? Using various *cutting-plane methods* such as the ellipsoid method, it is possible to get a $O(d \log(1/\epsilon))$ rate for convex, Lipschitz functions. This could be much better than the $O(1/\epsilon^2)$ rate for gradient descent for small error ϵ (in particular, if $\epsilon \ll 1/\sqrt{d}$). However, these cutting plane based methods are much more expensive than our simple and efficient gradient descent algorithm. There is recent work which shows that any algorithm which significantly improves on the convergence rate of gradient descent in this error regime must be much more expensive computationally, with memory as the metric of computational cost [2].

It is also known that we cannot improve on the $O(\sqrt{\kappa} \log(1/\epsilon))$ convergence of accelerated GD for strongly convex problems with only gradient information. *Second-order methods* such as Newton's method can get a much better polylog(κ) dependence on the condition number κ by using second-derivative information. Intuitively, second-derivative information allows the algorithm to correct for the curvature of the problem. These methods can be more expensive computationally since they need to second-derivative (for e.g. this needs $O(d^2)$ memory to store in d dimension since the second-derivative matrix has size $d \times d$ vs. only $O(d)$ memory for gradient information). There are techniques which aim to use some of this second-derivative information without this computational cost, variants of these such as Adagrad [3] and Adam [4] are very popular in practice.

2 Stochastic Gradient Descent (SGD)

Stochastic gradient descent is a simple modification of GD, where we use an unbiased estimate of the gradient instead of the exact gradient.

Algorithm 1: SGD

1 Initialize w_1
2 **for** t=*1,2,...,T* **do**
3 Choose any v_t s.t. $\mathbb{E}(v_t|w_t) = \nabla f(w_t)$
4 $w_{t+1} \leftarrow w_t - \eta v_t$

Step 3 requires that v_t is a valid gradient in expectation. If we are minimizing training loss over n points, we can just sample one point (x_i, y_i) uniformly at random and take gradient at (x_i, y_i), instead of computing gradient for the entire training set. Since

$$\nabla_w \left(\frac{1}{n} \sum_{i=1}^{n} \ell(w, z_i) \right) = \frac{1}{n} \sum_{i=1}^{n} \nabla_w \ell(w, z_i),$$

There if we sample z_i uniformly at random from the datapoints,

$$\mathbb{E}_{z_i \sim \text{Unif}(S)} \left[\nabla_w \ell(w, z_i) \right] = \nabla_w \left(\frac{1}{n} \sum_{i=1}^{n} \ell(w, z_i) \right)$$

Therefore this gives an unbiased estimate of the overall gradient.

We have a similar convergence guarantee for SGD as our guarantee for GD.

Theorem 3 (convergence of SGD)**.** *Let f be a convex, ρ-Lipschitz function. Let $w^* = \arg\min_{w \in \mathbb{R}^d} f(x)$ and $\|w^* - w_1\| = B$ (where w_1 is the initialization). Suppose we run SGD for T steps with step size $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$ and $\|v_t\| \leq \rho$ for all t. Let $\overline{w} = \frac{1}{T} \sum_{t=1}^{T} w_t$. Then,*

$$\mathbb{E}[f(\overline{w})] - f(w^*) \leq \frac{B\rho}{\sqrt{T}}.$$

Proof. Let $v_{1:t}$ denote the sequence $v_1, \ldots, v_t$. By convexity,

$$\mathop{\mathbb{E}}_{v_{1:T}} [f(\overline{w}) - f(w^*)] \leq \mathop{\mathbb{E}}_{v_{1:T}} \left[\frac{1}{T} \sum_{t=1}^{T} f(w_t) - f(w^*) \right].$$

By the iterative update lemma from the proof of GD,

$$\mathop{\mathbb{E}}_{v_{1:T}} \left[\frac{1}{T} \sum_{t=1}^{T} \langle w_t - w^*, v_t \rangle \right] \leq \frac{B\rho}{\sqrt{T}}.$$

So we only need to show

$$\mathop{\mathbb{E}}_{v_{1:T}}\left[\frac{1}{T}\sum_{t=1}^{T} f(w_t) - f(w^*)\right] \le \mathop{\mathbb{E}}_{v_{1:T}}\left[\frac{1}{T}\sum_{t=1}^{T} \langle w_t - w^*, v_t\rangle\right].$$

$$\begin{aligned}\mathop{\mathbb{E}}_{v_{1:T}}\left[\frac{1}{T}\sum_{t=1}^{T} f(w_t) - f(w^*)\right] &= \frac{1}{T}\sum_{t=1}^{T}\mathop{\mathbb{E}}_{v_{1:T}}\left[f(w_t) - f(w^*)\right] \\ &= \frac{1}{T}\sum_{t=1}^{T}\mathop{\mathbb{E}}_{v_{1:t-1}}\left[f(w_t) - f(w^*)\right].\end{aligned}$$

SGD requires $\mathop{\mathbb{E}}_{v_t}[v_t \mid w_t] = \nabla f(w_t)$.

Since w_t only depends on $v_{1:t-1}$,

$$\mathop{\mathbb{E}}_{v_t}[v_t \mid v_{1:t-1}] = \nabla f(w_t).$$

$$\begin{aligned}\mathop{\mathbb{E}}_{v_{1:T}}\left[\frac{1}{T}\sum_{t=1}^{T} f(w_t) - f(w^*)\right] &\le \frac{1}{T}\sum_{t=1}^{T}\mathop{\mathbb{E}}_{v_{1:t-1}}\left[\left\langle w_t - w^*, \mathop{\mathbb{E}}_{v_t}[v_t \mid v_{1:t-1}]\right\rangle\right] \\ &= \frac{1}{T}\sum_{t=1}^{T}\mathop{\mathbb{E}}_{v_{1:t-1}}\left[\mathop{\mathbb{E}}_{v_t}\left[\langle w_t - w^*, v_t\rangle \mid v_{1:t-1}\right]\right] \\ &= \frac{1}{T}\sum_{t=1}^{T}\mathop{\mathbb{E}}_{v_{1:t-1}}\left[\langle w_t - w^*, v_t\rangle\right] \text{(law of iterated expectations)} \\ &= \frac{1}{T}\sum_{t=1}^{T}\mathop{\mathbb{E}}_{v_{1:T}}\left[\langle w_t - w^*, v_t\rangle\right] \\ &= \mathop{\mathbb{E}}_{v_{1:T}}\left[\frac{1}{T}\sum_{t=1}^{T}\langle w_t - w^*, v_t\rangle\right],\end{aligned}$$

and we are done. ■

Some more points about the convergence rate,

- Same convergence rate as GD (but in expectation).
- The convergence rate does not improve with the β-smoothness assumption.
- For strongly convex functions, the convergence rate goes from $\frac{1}{\sqrt{T}}$ to $\frac{1}{T}$, but we cannot get the $e^{-T/K}$ rates that we got for GD.

Even though the convergence rate maybe slower for SGD compared to GD, each iteration can be much faster. For example, for the ERM problem we only need to find the gradient at one randomly sampled datapoint to get a stochastic gradient, instead of taking the gradient over every example. Therefore, there is some tradeoff in terms of cost per iteration and the number of iterations.

3 Learning with SGD, Discussion

Suppose we are interested in minimizing the risk

$$R(w) = \mathop{\mathbb{E}}_{z \sim D}[\ell(w, z)].$$

We have seen one way of solving this, we sample n datapoints, minimize training error, and then use generalization bounds to ensure small test error.

SGD gives a different way to look at this. Note that

$$\nabla R(w_t) = \nabla \mathop{\mathbb{E}}_{z \sim D}[\ell(w_t, z)] = \mathop{\mathbb{E}}_{z \sim D}[\underbrace{\nabla \ell(w_t, z)}_{\text{gradient at } z}].$$

If we set $v_t = \nabla \ell(w_t, z)$ (for $z \sim D$),

$$\mathbb{E}[v_t] = \nabla R(w_t).$$

Therefore we can directly do SGD on the true risk, and get a direct bound on the risk of the estimate.

Algorithm 2: SGD for minimizing $R(w)$

1. Initialize w_1
2. **for** $t = 1, \ldots, T$ **do**
3. $\quad z \leftarrow \mathrm{EX}(\mathcal{C}, D)$
4. $\quad$ get $v_t = \nabla \ell(w_t, z)$
5. $\quad$ update $w_{t+1} = w_t - \eta v_t$
6. Output $\bar{w} = \frac{1}{T} \sum_{t=1}^{T} w_t$

Corollary 4. *Consider a convex, ρ-Lipschitz function $\ell(w, z)$. Let $w^* = \arg\min_{w \in \mathcal{H}} R(w)$. Let $\|w^* - w_1\| \leq B$. Then if we run SGD for T iterations with $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$ where $T \geq \frac{B^2 \rho^2}{\epsilon^2}$, then the output $\overline{w}$ satisfies,*

$$\mathbb{E}[R(\overline{w})] \leq \min_{w \in \mathcal{H}} R(w) + \epsilon.$$

We conclude with some remarks on SGD:

1. SGD is implementable in the SQ model (we need to extend the SQ model to allow real-valued queries).
2. SGD is a stable algorithm [5], and hence we can get some guarantees for its generalization gap.

4 Boosting

We will discuss a new algorithmic framework now, boosting. We start by recalling our definition of weak learning from the lecture on SQ learnability.

Definition 5 (Weak Learning)**.** *An algorithm A is a weak learner with advantage γ for class $\mathcal{C}$ if: for any dist. D and any target $c \in \mathcal{C}$, given access to $EX(c, D)$, w.p. $(1-\delta)$, produces a hypotheses with error$(h; c, D) \leq \gamma$.*

If A runs in time poly$(d, 1/\delta)$ and $\gamma \geq \frac{1}{\text{poly}(d)}$, then $\mathcal{C}$ is efficiently weakly PAC-learnable.

In [6], Kearns and Valiant asked if weak PAC learning implies PAC learning. This was answered positively by Freund and Schapire in [7], and the resulting Adaboost algorithm has been highly influential both in theory and practice.

Theorem 6. *If $\mathcal{C}$ is weakly PAC-learnable (efficiently) the $\mathcal{C}$ is PAC-learnable (efficiently).*

Proof. The proof relies on the AdaBoost algorithm due to Freund and Schapire. We begin by recalling our setup.

There is a training set $\{(x_1, y_1), \cdots, (x_n, y_n)\}$, where $x_i \in \mathcal{X}, y_i \in \{-1, 1\}$. Note that by realizability,

$$\exists\ c \in \mathcal{C}\ s.t. \forall i\ y_i = c(x_i).$$

We assume there exists weak learning algorithm (WL), for $\mathcal{C}$.

Algorithm 3: AdaBoost

---

1 $D_1(i) = \frac{1}{n}\ \forall\ i \in [n]$

2 **for** $t = 1, \ldots, T$ **do**

3 Use Weak Learner (WL) on distribution D_t to get h_t

4 Let $\epsilon_t = \mathbb{P}_{x \sim D_t}[h_t(x) = y]$ ($\epsilon_t \leq 1 - \gamma$ with probability $1 - \delta$)

5 Choose $\alpha_t = \frac{1}{2} \log \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$

6 $D_{t+1}(i) = \frac{D_t(i)}{z_{t+1}} = \begin{cases} e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\ e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i \end{cases} = \frac{D_t(i)}{z_{t+1}} e^{-\alpha_t h_t(x_i) y_i}$, where z_{t+1} is normalizing constant

7 Output sign$(H(x))$ where $H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$

Some remarks about the algorithm:

1. We can assume $\epsilon_t \leq \frac{1}{2} - \gamma$ (by a union bound over all time steps).

2. We can emulate the example oracle $EX(c, D_t)$, because D_t has finite support.

As an aside, AdaBoost fits in the "Multiplicative Weight Update" framework [8]. This is a general framework with many algorithmic applications, all of which take the form of maintaining a distribution over a certain set and use the multiplicative update rule (such as in AdaBoost) to iteratively change these weights.

Though AdaBoost might initially seem very different from the algorithms we have seen so far in the class, it can be shown that it is actually minimizing the exponential loss as a surrogate for 0/1 loss in a greedy manner. Recall that for convex surrogate function $\phi : \mathbb{R} \to \mathbb{R}$, instead of minimizing the 0/1 loss we minimize the loss on the surrogate,

$$\min \phi(yh(x))$$

With AdaBoost, we solve $\min \sum_{i=1}^{n} \exp(-y_i H(x_i))$ in a greedy manner.

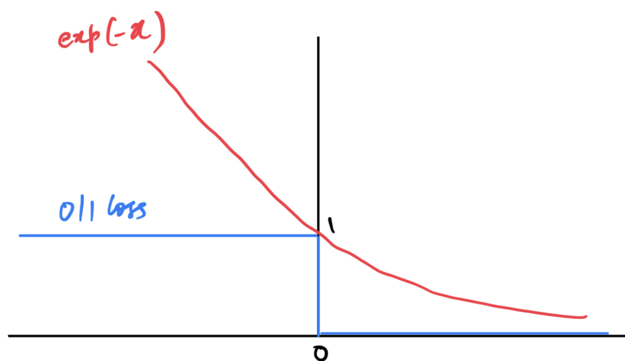


Figure 1: $\exp(-x)$ is convex surrogate for 0/1 loss

Suppose we have found $f_t(x) = \sum_{\tau=1}^{t} \alpha_\tau h_\tau(x)$. AdaBoost adds a function $h_{t+1}(x)$ to $f_t(x)$ which is a solution of the following problem,

$$\min_{h_{t+1}(x)} \sum_{i=1}^{n} \exp\Big(-y_i f_{t+1}(x)\Big).$$

Try to verify that solving this problem yields the AdaBoost algorithm.

We now go back to proving the guarantee for AdaBoost. We will first show that AdaBoost gets zero training error, and then show that it generalizes.

Lemma 7. *For* $T = \frac{1}{2\gamma^2}\log(2n)$*, the training error is* 0.

Proof. Note that the training error is given by,

$$\mathbb{P}_{D_1}(\text{sign}(H(x) \neq y)) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}(\text{sign}(H(x) \neq y_i)).$$

Note that

$$\mathbb{1}(\text{sign}(H(x) \neq y)) \leq e^{-yH(x)}.$$

Define $H_t(x) = \sum_{s=t}^{T} \alpha_s h_s(x)$. $H_t(x)$ has the following recursive form,

$$H_t(x) = \alpha_t h_t(x) + H_{t+1}(x)$$
$$H_1(x) = H(x)$$

Therefore we can write the training error as,

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}(\text{sign}(H(x_i) \neq y_i)) &\leq \sum_{i=1}^{n} D_1(i) e^{-y_i H_1(x_i)} \\
&= \sum_{i=1}^{n} D_1(i) e^{-d_1 y_i h_1(x_i) - y_i H_2(x_i)} \\
&= z_2 \sum_{i=1}^{n} D_2(i) e^{-y_i H_2(x_i)} \\
&= z_2 \sum_{i=1}^{n} D_2(i) e^{-\alpha_2 y_i h_2(x_i) - y_i H_3(x_i)} \\
&= z_2 z_3 \sum_{i=1}^{n} D_3(i) e^{-y_i H_3(x_i)} \\
&\vdots \\
&= \prod_{t=2}^{T+1} z_t
\end{aligned}$$

We now need to bound z_{t+1}.

$$\begin{aligned}
z_{t+1} &= \sum_{i=1}^{n} D_t(i) e^{-\alpha_t y_i h_t(x_i)} \\
&= \sum_{i: y_i = h_t(x_i)} D_t(i) e^{-\alpha_t} + \sum_{i: y_i \neq h_t(x_i)} D_t(i) e^{\alpha_t} \\
&= (1-\epsilon_t) e^{-\alpha_t} + \epsilon_t e^{\alpha_t} \qquad \text{Recall: } \alpha_t = \frac{1}{2}\log(\frac{1-\epsilon_t}{\epsilon_t}) \\
z_{t+1} &= (1-\epsilon_t)\sqrt{\frac{\epsilon_t}{1-\epsilon_t}} + \epsilon_t \sqrt{\frac{1-\epsilon_t}{\epsilon_t}} - 2\sqrt{\epsilon_t(1-\epsilon_t)}
\end{aligned}$$

Define $\gamma_t = \frac{1}{2} - \epsilon_t$ Note that $\gamma_t \geq \gamma$:

$$\begin{aligned}
z_{t+1} &= 2\sqrt{(1/2 - \gamma_t)(1/2 + \gamma_t)} \\
&= \sqrt{1 - 4\gamma_t^2} \\
&\leq (e^{-4\gamma_t^2})^{\frac{1}{2}} = e^{2\gamma_t^2} \leq e^{-2\gamma^2}
\end{aligned}$$

$$\text{Therefore } \prod_{t=1}^{T+1} z_t \leq e^{-2T\gamma^2} \leq \frac{1}{2n} \text{ if } T \geq \frac{1}{2\gamma^2}\log(2n)$$

■

This shows that the training error is small. What can we say about test error?

We will use our generalization bound to show that the test error is also small. Suppose that WL(weak learning algo) always outputs a hypothesis from some class $\mathcal{H}$ whose VC-dim is d. Let

$$LC(\mathcal{H}, T) = \left\{ \text{sign}(\sum_{i=1}^{T} \alpha_i h_i(x)) : \; h_i \in \mathcal{H} \; d_i \in \mathbb{R} \right\}$$

Exercise: Show that VC-dim($LC(\mathcal{H}, T)$) $\leq c.T.d. \log(T)$ for some constant c. (Hint: As in HW1, Compute the growth function and use Sauer's Lemma.)

Therefore, due to the VC-theorem,
If $n \geq \frac{c}{\epsilon}\left(\log(\frac{1}{\delta}) + Td\log(T)\log(1/\epsilon)\right)$ then the hypothesis produced by Ada Boost has **error** $\leq \epsilon$, with probability $(1-\delta)$.
Putting in the bound of T, if $n \geq \frac{c}{\epsilon}\left(\log(1/\delta) + \frac{1}{2\gamma^2}\log(2n)d\log\left(\frac{1}{2\gamma^2}\log(2n)\right)\log(1/\epsilon)\right)$, we will get error $\leq \epsilon$ w.p. $1-\delta$.
This is satisfied for $n \geq \frac{c}{\epsilon}\left(\frac{d}{\gamma^2}\text{poly}\left(\log(d, \frac{1}{\gamma^2}, \frac{1}{\epsilon})\right) + \log(1/\delta)\right)$ (in general, $n \geq a\log(n)$ is satisfied for $n \geq O(a\log(a))$). This completes the proof of our theorem. ■

"Overfitting" and AdaBoost

In the proof of the previous theorem, the VC bound degrades with the number of steps T, i.e. we expect poor generalization if we continue with boosting . However, in practice what is often observed is that that the test error for boosting can go down even after training error is already zero [9].

Figure 2: Training and test errors of AdaBoost

This has subsequently been explained, and it has been shows that AdaBoost maximizes some notion of "margin" after getting perfect accuracy on training set. Informally, in the linear case the margin is the minimum distance of any datapoint from the separating hyperplane.

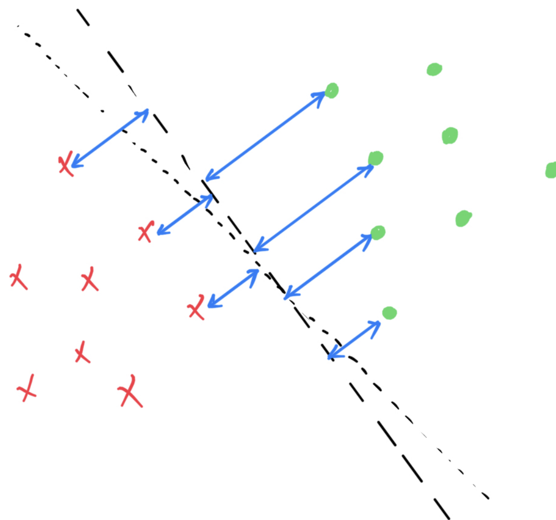


Figure 3: Margin of a classifier. Adaboost prefers classifiers with larger margin, among classifiers which get similar accuracy.

The training/test curves in Fig. 2 are also seen with deep neural networks. Recent work aims to explain this by showing that neural networks also maximize margin [10, 11].

5 Kernels: A Brief Introduction

We consider one final algorithmic framework, kernels. To motivate kernels, let us consider linear regression with ℓ_2 regularization:

$$\min_{w \in \mathbb{R}^d} \sum_{i=1}^{n} \left(w^T x_i - y_i\right)^2 + \lambda ||w||_2^2.$$

A linear model on the original datapoints may not always be expressive enough. Therefore, we may choose to worker with a richer set of features, which we can accomplish by mapping each data point $x \in \mathbb{R}^d$ to features $\phi(x) \in \mathbb{R}^m$. For example,

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \phi(x) = \begin{pmatrix} x_1^2 \\ \sqrt{2} x_1 x_2 \\ x_2^2 \end{pmatrix}.$$

The regression problem then becomes,

$$\min_{w \in \mathbb{R}^m} \sum_{i=1}^{n} \left(w^T \phi(x_i) - y_i\right)^2 + \lambda ||w||_2^2.$$

Let us consider two different algorithms for solving this.

1. Gradient descent: The gradient is given by
   $\nabla_w(SRM) = 2\sum_{i=1}^{n}\Big(w^T\phi(x_i) - y_i\Big)\phi(x_i) + 2\lambda w.$
   Therefore GD steps take the form,

   $$w_{t+1} = w_t - \eta\Bigg(2\sum_{i=1}^{n}\Big(w^T\phi(x_i) - y_i\Big)\phi(x_i) + 2\lambda w\Bigg).$$

   If we start with initialization $w_1 = 0$, then at any t, $w_t = \sum_{i=1}^{n}\alpha_i\phi(x_i)$ for some $\alpha_i \in \mathbb{R}$. Therefore, **the predictor is a linear combination of data points**.In addition, note that the prediction at any point x is given by,

   $$\langle w_t, \phi(x)\rangle = \sum_{i=1}^{n}\alpha_i\langle\phi(x_i), \phi(x)\rangle.$$

   Therefore, **the prediction only depends on the inner products between feature vectors**.

2. Suppose we find closed form solution instead of doing GD. Let

   $$\Phi_{\in\mathbb{R}^{n\times m}} = \begin{pmatrix} \text{---} & \phi(x_1) & \text{---} \\ \text{---} & \phi(x_2) & \text{---} \\ & \vdots & \\ \text{---} & \phi(x_n) & \text{---} \end{pmatrix}, y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}.$$

   Then the objective is,

   $$\min_{w\in\mathcal{R}^d} ||\Phi w - y||_2^2 + \lambda||w||_2^2$$

   $$\nabla_w|_{w=w^*}\Big((\Phi w - y)^T(\Phi w - y) + \lambda w^T w\Big) = 0.$$

   Taking the gradient and setting it to zero to find the solution,

   $$2\Phi^T\Big(\Phi w^* - y\Big) + 2\lambda w^* = 0$$

   $$w^* = \Big(-\frac{1}{\lambda}\Big)\Phi^T\Big(\Phi w^* - y\Big)$$

   $$\implies w^* = \Phi^T\alpha$$

   for some $\alpha \in \mathbb{R}^n$, i.e. $\Big(w^* = \sum_{i=1}^{n}\alpha_i\phi(x_i)\Big)$, as with GD before. We can also find α by plugging $w = \Phi^T d$ back into the SRM objective,

   $$\min_{\alpha} ||\Phi\Phi^T\alpha - y||_2^2 + \lambda||\Phi^T\alpha||_2^2$$

   Let $K = \Phi\Phi^T \in \mathbb{R}^{n\times n}$, given by

   $$\begin{pmatrix} \text{---} & \phi(x_1) & \text{---} \\ \text{---} & \phi(x_2) & \text{---} \\ & \vdots & \\ \text{---} & \phi(x_n) & \text{---} \end{pmatrix}\begin{pmatrix} | & | & \cdots & | \\ \phi(x_1) & \phi(x_2) & \cdots & \phi(x_n) \\ | & | & \cdots & | \end{pmatrix}.$$

Therefore the objective becomes,

$$\min_{\alpha} ||K\alpha - y||_2^2 + \lambda\alpha^T K\alpha$$

We can solve for α as follows,

$$\begin{aligned}
&\nabla_\alpha(||K\alpha - y||_2^2 + \lambda\alpha^T K\alpha) = 0 \\
\iff &2K^T(K\alpha - y) + 2\lambda K\alpha = 0 \\
\iff &K^T((K + \lambda I)\alpha - y) = 0 \\
\iff &(K + \lambda I)\alpha = y \\
\iff &\alpha = (K + \lambda I)^{-1} y \\
\implies &w^* = \Phi^T\alpha = \Phi^T(K + \lambda I)^{-1} y.
\end{aligned}$$

Note that compared to finding the regularized least squares solution $(\Phi^T\Phi + \lambda I)^{-1}\Phi^T y$, it could be more computationally efficient to find the above solution. This is because the above solution requires finding $(K + \lambda I)^{-1} \in \mathbb{R}^{n\times n}$ which takes $O(n^3)$ time, whereas the regularized least squares solution requires finding $(\Phi^T\Phi + \lambda I)^{-1} \in \mathbb{R}^{m\times m} \to \mathcal{O}(m^3)$ which takes $O(m^3)$ time. For $m \gg d$, the regularized least squares solution is more expensive to compute.

Therefore, for both these algorithms, we have the property that the predictor is a linear combination of the feature vectors of the datapoints, and because of this the predictions only depend on the inner product between the feature vectors. A kernel is a function which is defined as the inner product of two feature vectors.

Definition 8 (Kernel). *A function $k : \mathbb{R}^d \to \mathbb{R}$ is called a kernel function if and only if there exists $\phi : \mathbb{R}^d \to \mathbb{R}^m$, such that for any $x, x' \in \mathbb{R}^d$,*

$$k(x, x') = \phi(x)^T\phi(x')$$

.

Let us see some examples of kernels.

Example 9. *Examples of kernel functions:*

1. *For* $\phi(x) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$
   $\phi(x)^t\phi(x') = x_1^2x_1'^2 + 2x_1x_2x_1'x_2' + x_2^2x_2'^2 = (x_1x_1' + x_2x_2')^2 = (x^Tx')^2$
   Therefore $k(x, x') = (\langle x, x'\rangle)^2$, *just the square of the original inner product.*

2. *As a different type of example, for two strings s_1 and s_2 let*

   $$k(s_1, s_2) = |\{\textit{all characters which appear at least once in both } s_1 \textit{and } s_2\}|.$$

   For example if s_1 = "university" and s_2 = "california", then $k(s_1, s_2) = 2$. This function is a kernel, which we can verify via the following feature mapping,

   $$\phi(s_1) = \begin{pmatrix} \mathbf{1}(\text{'a' appears in } s_1) \\ \mathbf{1}(\text{'b' appears in } s_1) \\ \vdots \\ \mathbf{1}(\text{'z' appears in } s_1) \end{pmatrix}, \phi(s_2) = \begin{pmatrix} \mathbf{1}(\text{'a' appears in } s_2) \\ \mathbf{1}(\text{'b' appears in } s_2) \\ \vdots \\ \mathbf{1}(\text{'z' appears in } s_2) \end{pmatrix}.$$

For many interesting feature maps, it can be much more efficient to compute the kernel function rather than explicitly computing the inner products between the feature vectors. This will even be true for the string similarity kernel above, if the set of possible characters is very large but the length of the words is small. Later, we will see that the feature map corresponding to some kernel functions may even be infinite. Therefore, kernels can allow us to efficiently work with rich feature maps.

The following theorem provides an equivalent characterization of kernels, and can be useful for showing that a function is/is not a kernel.

Theorem 10 (Mercer's theorem)**.** *Any function k is a kernel if and only if the matrix K, $K_{ij} = k(x_i, x_j)$, is positive semi-definite ($K \succcurlyeq 0$) for any n data points $x_1, \ldots, x_n$ (and any n). (Recall that $K \succcurlyeq 0$ if and only if $v^T K v \geq 0 \;\forall\; v$).*

The following properties can also be useful to verify that a function is a kernel.

Theorem 11. *Properties of kernels:*

1. *For any $f : \mathbb{R}^d \to \mathbb{R}$, $k(x, x') = f(x)f(x')$ is a kernel*
2. *If $k_1(\cdot,\cdot)$ and $k_2(\cdot,\cdot)$ are kernels then the following are also kernels,*
   a) $\alpha k_1(\cdot,\cdot) + \beta k_2(\cdot,\cdot)$ if $\alpha, \beta \geq 0$
   b) $k_1(\cdot,\cdot) \cdot k_2(\cdot,\cdot)$

Try to verify these properties by constructing corresponding feature mappings.

6 Further reading

The discussion for SGD closely mirrors Chapter 14 of [12], and boosting is discussed in Chapter 10. There are many resources online for kernels, for example you can refer to Percy Liang's lecture notes.

References

[1] Arkadij Semenovič Nemirovskij and David Borisovich Yudin. Problem complexity and method efficiency in optimization. 1983.

[2] Annie Marsden, Vatsal Sharan, Aaron Sidford, and Gregory Valiant. Efficient convex optimization requires superlinear memory. In *Conference on Learning Theory*, pages 2390–2430. PMLR, 2022.

[3] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[4] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[5] Moritz Hardt and Benjamin Recht. Patterns, predictions, and actions: A story about machine learning. *arXiv preprint arXiv:2102.05242*, 2021.

[6] Michael Kearns and Leslie Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM (JACM)*, 41(1):67–95, 1994.

[7] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[8] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of computing*, 8(1):121–164, 2012.

[9] Peter Bartlett, Yoav Freund, Wee Sun Lee, and Robert E Schapire. Boosting the margin: A new explanation for the effectiveness of voting methods. *The annals of statistics*, 26(5):1651–1686, 1998.

[10] Daniel Soudry, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. The implicit bias of gradient descent on separable data. *The Journal of Machine Learning Research*, 19(1):2822–2878, 2018.

[11] Ziwei Ji and Matus Telgarsky. Directional convergence and alignment in deep learning. *Advances in Neural Information Processing Systems*, 33:17176–17186, 2020.

[12] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.