Contents

1 Optimization Methods

Beginning of Aug. 25, 2022

1.1 Loss Function & Risk Function

Suppose we are given a **loss function** $\ell(f(x), y)$. A frequent example is the squared loss for $y = \mathbb{R}$ defined by $\ell(f(x), y) = (f(x) - y)^2$.

A big question of interest in ML is *what* to minimize this function over? For example, if we were to minimize loss over some distribution D over all (x, y), we want to minimize the **risk function** (risk of prediction $f(x)$ defined by

$$R(f) := \mathbb{E}_{(x,y)\sim D}[\ell f(x), y] = \sum_{(\tilde{x},\tilde{y})} \mathbb{P}((x, y) = (\tilde{x}, \tilde{y}))\ell(f(\tilde{x}), \tilde{y}).$$

Challenges we naturally encounter:

(1) i.i.d. assumption. We assume that we have a set of labelled instances drawn identically and independently from a distribution D but often this assumption is too idealized.

(2) Theoretical abstraction – often useful.

Minimizing Risks

Definition: Empirical Risk

Given a set of labelled data points $S = \{(x_i, y_i)\}_{i=1}^n$, we define the **empirical risk** of any predictor f with respect to S to be

$$\hat{R}_S(f) := \frac{1}{n}\sum_{i=1}^{n} \ell(f(x_i), y_i).$$

Note that this risk function coincides with the more general one under the discrete uniform distribution.

Definition: Empirical Risk Minimizer (ERM)

Given a function class (i.e., a collection of functions) $\mathcal{F} : \{f : \mathcal{X} \to \mathcal{Y}\}$ and a set S of labelled data points, we define the **empirical risk minimizer** to be

$$\min_{f \in \mathcal{F}} \hat{R}_S(f) = \min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i).$$

Generalizing Risks

What we *really* want to do is to generalize the results to beyond data points we already have. Given a function f, a data set S, the following is a trivial tautology:

$$R(f) = \underbrace{\hat{R}_S(f)}_{\text{emp. risk}} + \underbrace{(R(f) - \hat{R}_S(f))}_{\text{generalization gap}}.$$

That is, to minimize $R(f)$, it suffices to minimize both the empirical risk $\hat{R}_S(f)$ and the remaining term $R(f) - \hat{R}_S(f)$, known as the **generalization gap**, a quantifier of how well our prediction generalizes to unseen examples.

Measuring Generalization: Training / Test Paradigm

In theory, we derive *generalization bounds* based on complexity of the model to obtain upper bounds for the generalization gap. In practice we conduct **empirical evaluation** — we divide the data into two parts, the **training set**, a subset of data points on which we train our model, and a **test set**, another subset on which we test the model. Ideally, we only use test set once or a few times. Our major concern is that our algorithm does well on training set only because it has memorized everything rather than actually doing prediction. A good algorithm, on the other hand, should have a small generalization gap.

Supervised Learning in a Nutshell

(1) Loss function: what is the right loss function for the task?

(2) Representation: what class of functions should we use?

- Inductive bias: *no model can do well on every task. "All models are wrong, but some are useful."*

(3) Optimization: how can we efficiently find the ERM?

(4) Generalization: will the predictions of our model transfer gracefully to unseen examples?

- Note we can have trivial models that have zero generalization gap by outputting a constant, but such model violates the optimization criteria in almost all cases.

1.2 Formal Setup of Linear Regression

- Input: $x \in \mathbb{R}^d$.

- Output: $y \in \mathbb{R}$.
- Training data: $S = \{(x_i, y_i)\}_{i=1}^n$.
- Linear model: $f : \mathbb{R}^d \to \mathbb{R}$ with $f(x) = w_0 + w \cdot x$ where $w := (w_1, ..., w_d) \in \mathbb{R}^d$ is the **weight factor**. For convenience, we define $\tilde{x} := (1, x) \in \mathbb{R}^{d+1}$ and $\tilde{w} := (w_0, w_1, ..., w_d)$. By doing so, $f(x)$ can be re-written as $f(x) := \tilde{w}^T \tilde{x}$.

Our goal is to minimize total squared error,

$$\hat{R}_S(\tilde{w}) = \frac{1}{n} \sum_{i=1}^n (f(x_i) - y_i)^2 = \frac{1}{n} \sum_{i=1}^n (\tilde{x}_i^T \tilde{w} - y_i)^2.$$

We define the **residual sum of squares**, RSS, to be

$$\text{RSS}(\tilde{w}) := n\hat{R}_S(\tilde{w}) = \sum_{i=1}^n (\tilde{x}_i^T \tilde{w} - y_i)^2.$$

Note that under such notation, ERM is identical to finding

$$\tilde{w}^* := \operatorname*{argmin}_{\tilde{w} \in \mathbb{R}^{d+1}} \text{RSS}(\tilde{w}),$$

and the solution is known as the **least squares solution**.

Example: $d = 0$. Let $d = 0$ so we are trying to find the best constant function $f(x) = w_0$ that predicts a set of data. In this case,

$$\text{RSS}(w_0) = \sum_{i=1}^n (w_0 - y_i)^2 = nw_0^2 - 2w_0 \sum_{i=1}^n y_i + C = n\Big(w_0 - \frac{1}{n} \sum_{i=1}^n y_i\Big)^2 + \tilde{C}$$

where $C, \tilde{C}$ are constants of little interest. It follows that w_0^* is simply the mean of the y_i's.

Example: $d = 1$. Now let us consider $d = 1$ so that

$$\text{RSS}(\tilde{w}) = \sum_{i=1}^n (w_0 + w_1 x_i - y_i)^2. \tag{*}$$

Taking gradient gives

$$\frac{\partial}{\partial w_0} \text{RSS}(\tilde{w}) \propto \sum_{i=1}^n (w_0 + w_1 x_i - y_i)$$

and

$$\frac{\partial}{\partial w_1} \text{RSS}(\tilde{w}) \propto \sum_{i=1}^n x_i (w_0 + w_1 x_i - y_i).$$

Setting both to 0, we obtain a linear system

$$\begin{bmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \end{bmatrix},$$

whose solution would be (assuming the 2×2 matrix is invertible)

$$\begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = \begin{bmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum y_i \\ \sum x_i y_i \end{bmatrix}.$$

Since (*) is a convex function of both arguments, a stationary point is guaranteed to be a (global) minimum.

Example: General Case. Now we generalize to $\mathbb{R}^d$. Here,

$$\text{RSS}(\tilde{w}) = \sum_{i=1}^{n} (\tilde{x}_i^T \tilde{w} - y_i)^2.$$

Setting $\nabla \text{RSS}(\tilde{w})$ to $0 \in \mathbb{R}^d$, we have

$$\begin{aligned}\nabla \text{RSS}(\tilde{w}) &= 2 \sum_{i=1}^{n} \tilde{x}_i (\tilde{x}_i^T \tilde{w} - y_i) \propto \tilde{w} \sum_{i=1}^{n} (\tilde{x}_i^T \tilde{x}_i) - \sum_{i=1}^{n} \tilde{x}_i y_i \\ &= (\tilde{x}^T \tilde{x}) \tilde{w} - \tilde{x}^T y.\end{aligned}$$

The general solution is (assuming $\tilde{x}^T \tilde{x}$ is invertible) $\tilde{w}^* = (\tilde{x}^T \tilde{x})^{-1} \tilde{x}^T y$.
In particular, suppose that $\tilde{x}^T \tilde{x} = I$ so that $\tilde{w}^* = \tilde{x}^T y$.
Alternate approach:

$$\begin{aligned}\text{RSS}(\tilde{w}) &= \sum_{i=1}^{n} (\tilde{w}^T \tilde{x}_i - y_i)^2 = \|\tilde{x}\tilde{w} - y\|_2^2 \\ &= (\tilde{x}\tilde{w} - y)^T (\tilde{x}\tilde{w} - y) \\ &= \tilde{w}^T \tilde{x}^T \tilde{x} \tilde{w} - y^T \tilde{x} \tilde{w} - \tilde{w}^T \tilde{x}^T y + C \\ \text{[completion of squares]} &= (\tilde{w} - (\tilde{x}^T \tilde{x})^{-1} \tilde{x}^T y)^T (\tilde{x}^T \tilde{x}) (\tilde{x} - (\tilde{x}^T \tilde{x})^{-1} \tilde{x}^T y) + C.\end{aligned}$$

It remains to notice that $u^T (\tilde{x}^T \tilde{x}) u = \|\tilde{x}u\|_2^2 \geqslant 0$ and $= 0$ iff $u = 0$. Hence the minimizer of RSS takes place when $\tilde{w}^* = (\tilde{x}^T \tilde{x})^{-1} \tilde{x}^T y$.

1.3 Gradient Descent

The bottleneck of computing

$$\tilde{w}^* = (\tilde{x}^T \tilde{x})^{-1} \tilde{x}^T y$$

is to invert the matrix $\tilde{x}^T \tilde{x} \in \mathbb{R}^{(d+1)^2}$ which takes $\mathcal{O}(d^3)$ time (faster algorithms exist in theory but may not be practical).
Problem: minimize a function $F(w)$.
Gradient descent: start with some $w^{(0)}$; for $t \in \{0, 1, ..., T\}$, define $w^{(t+1)} := w^{(t)} - \eta \nabla F(w^{(t)})$ where η is called the step size / learning rate.

Example. Let $w = (w_1, w_2) \in \mathbb{R}^2$ and define

$$F(w) := 0.5(w_1^2 - w_2)^2 + 0.5(w_1 - 1)^2.$$

The gradient is

$$\frac{\partial F}{\partial w_1} = 2(w_1^2 - w_2) w_1 + w_1 - 1 \qquad \text{and} \qquad \frac{\partial F}{\partial w_2} = -(w_1^2 - w_2).$$

For GD, we initialize $w^{(0)} = (w_1^{(0)}, w_2^{(0)})$ (maybe $(0, 0)$ or randomly) and $t = 0$. We set η as well. Then we

iteratively set

$$w_1^{(t+1)} \leftarrow w_1^{(t)} - \eta[2((w_1^{(t)})^2 - w_2^{(t)})w_1^{(t)} + w_1^{(t)} - 1]$$
$$w_2^{(t+1)} \leftarrow w_2^{(t)} - \eta[(w_1^{(t)})^2 - w_2^{(t)}]$$
$$t \leftarrow t + 1.$$

We stop either when w converges or when t reached a prescribed number.

Why GD?

Intuition: we consider the first-order Taylor approximation

$$F(w) \approx F(w^{(t)}) + \nabla F(w^{(t)})^T (w - w^{(t)}).$$

Consequently

$$F(w^{(t+1)}) \approx F(w^{(t)}) - \eta \|\nabla F(w^{(t)})\|_2^2 \leqslant F(w^{(t)}),$$

so GD never increases function value, assuming we have a good choice of η (so we don't trave l too far and actually move away from the minima).

Convergence Guarantees for GD

For *convex objectives*, given ϵ there exists a lower bound for t such that $F(w^{(t)}) - F(w^*) < \epsilon$ for large t (so we eventually converge to the minima). Even for non-convex objectives, some guarantees still exist, e.g., how many iterations $t = t(\epsilon)$ are needed to achieve $\|\nabla F(w^{(t)})\| < \epsilon$ and approximate a **stationary point**.

It is known mathematically that a stationary point for a convex objective is a global minimizer.

Beginning of Sept. 1, 2022

For non-convex objectives, however, a stationary point may be a saddle point, and there exist functions and saddle points around which GD will get stuck. Even worse, it is *NP-hard* to distinguish saddle points from local minimum.

Stochastic Gradient Descent

Instead of always moving in the negative gradient direction, we consider an algorithm that moves in the *noisy* negative gradient direction given by

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \tilde{\nabla} F(w^{(t)})$$

where $\tilde{\nabla} F(w^{(t)})$ is an unbiased random variable called the **stochastic gradient** such that

$$\mathbb{E}[\tilde{\nabla} F(w^{(t)})] = \nabla F(w^{(t)}).$$

SGD usually needs more iterations to obtain convergence but each iteration takes less time. In some examples SGD can be significantly faster, e.g. when GD fails due to "bad" saddle points.

Second-Order Methods

Taylor approximation gives

$$F(w) = F(w^{(t)}) + \nabla F(w^{(t)})^T (w - w^{(d)}) + \frac{1}{2}(w - w^{(t)})^T H_t (w - w^{(t)}) + \mathcal{O}(\|w - w^{(t)}\|)^3$$

where H_t is the Hessian $\left\{\frac{\partial^2 F(w)}{\partial w_i \partial w_j}\right\}_{i,j}$ evaluated at $w = w^{(t)}$. We define the second-order approximation

$$\tilde{F}(w) := F(w^{(t)}) + \nabla F(w)^T (w - w^{(t)}) + \frac{1}{2}(w - w^{(t)})^T H_t (w - w^{(t)}).$$

Differentiating $\tilde{F}$ gives

$$\nabla \tilde{F}(w) = \nabla F(w^{(t)}) + H_t w - H_t w^{(t)}/2 - H_t w^{(t)}/2.$$

Setting this to zero we obtain

$$H^{(t)} w = H_t w^{(t)} - \nabla F(w^{(t)})$$

and so

$$w = w^{(t)} - H_t^{-1} \nabla F(w^{(t)}).$$

This iteration ($w \leftarrow w - H_t^{-1}\nabla F(w) = w - (\nabla^2 F(w))^{-1}\nabla F(w)$) is called **Newton's method**.

Newton's Method v.s. Gradient Descent

- Step size: Newton's method doesn't have any but GD requires one.
- Rate/order of convergence: Newton's method converges much faster (quadratic convergence).
- Complexity: Newton's method requires inversion of Hessians which is of $\mathcal{O}(d^3)$, but GD only takes $\mathcal{O}(d)$.

2 Linear Classifiers

Recall we have:

- input $x \in \mathbb{R}^d$,
- output (label): $y \in [C] := \{1, 2, ...C\}$, and
- goal: a mapping $f : \mathbb{R}^d \to [C]$ that predicts reliable outcomes based on input.

We first look at **binary classification**, where $C = 2$.

2.1 Binary Classification

Let $\mathcal{F}$ be the class of linear functions. Conveniently when $C = 2$ we make use of the sign function.

Definition: Separating Hyperplanes & Linear Predictors

In $\mathbb{R}^d$, we define the function class of **separating hyperplanes** to be

$$\mathcal{F} := \{f : \mathbb{R}^d \to \{-1, 1\} : f(x) = \text{sgn}(w^T x) \text{ where } w \in \mathbb{R}^d\}.$$

For notational simplicity we denote it as $\{f(x) = \text{sgn}(w^T x) : w \in \mathbb{R}^d\}$ and similarly onward.

Picking a Loss Function

With this setup, we now need to choose an appropriate loss function. The most common choice for binary classification is

$$\ell(f(x), y) \coloneqq \mathbf{1}[f(x) \neq y]$$

(i.e. the inficator function). The **0/1 loss function** is

$$\ell_{0-1}(yw^T x) \coloneqq \mathbf{1}[(yw^T x \leqslant 0)].$$

The function returns 1 when $w^T x \neq y$ (i.e., 1 and -1 or reversed). However, such function is non-convex and inconvenient mathematically and computationally. We instead consider a **convex surrogate loss.**

- The **perceptron loss** is defined as $\ell(z) \coloneqq \max\{0, -z\}$ (used in Perceptron),
- The **hinge loss** is defined as $\ell(z) \coloneqq \max\{0, 1 - z\}$ (used in SVM and many others), and
- The **logistic loss** $\ell(z) \coloneqq \log(1 + \exp(-z))$ (used in logistic regression).

GD on Perceptron Loss

We now try to find the ERM

$$w^* = \operatorname*{argmin}_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i w^T x_i)$$

where ℓ is a convex surrogate loss. In general there will not be a closed form solution but we have numerical toolboxes.

Perceptron Loss and the Perceptron Algorithm

We first focus on Perceptron loss:

$$F(w) = \frac{1}{n} \sum_{i=1}^{n} \max\{0, -y_i w^T x_i\}. \tag{*}$$

The gradient of $\max\{0, -z\}$ is 0 if $z \geqslant 0$ and -1 otherwise.
Applying GD to (*) gives

$$\nabla F(w) = -\frac{1}{n} \sum_{i=1}^{n} \mathbf{1}[y_i w^T x_i \leqslant 0] y_i x_i,$$

so we iteratively define

$$w \leftarrow w + \frac{\eta}{n} \sum_{i=1}^{n} \mathbf{1}[y_i w^T x_i \leqslant 0] y_i x_i. \tag{GD.1}$$

After every iteration, we need to update the entire training set, which can be costly.
Alternatively we can consider SGD, as follows:

- Pick one index $i \in [n]$ uniformly at random, and let

$$\tilde{\nabla} F(w^{(t)}) = -\mathbf{1}[y_i w^T x_i \leqslant 0] y_i x_i.$$

- Use $\tilde{\nabla}$ as the stochastic gradient.

- To see it is unbiased:
$$\mathbb{E}[\tilde{\nabla} F(w^{(t)})] = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}[y_i w^T x_i \leqslant 0] y_i x_i = \nabla F(w^{(t)}).$$
- We update the SGD by
$$w \leftarrow w + \eta \mathbf{1}[y_i w^T x_i \leqslant 0] y_i x_i. \tag{SGD.1}$$

In comparison, (SGD.1) is significantly faster than (GD.1) as each update only changes one data point at most.
The **Perceptron algorithm** is simply (SDG.1) with $\eta = 1$. That is, we initialize $w = 0$, and repeat the following until convergence:

- Pick $x_i \sim$ unif$(x_1, ..., x_n)$,
- If $\operatorname{sgn}(w^T x_i) \neq y_i$:
$$w \leftarrow w + y_i x_i.$$

Intuition. Say our w makes mistake on (x_i, y_i), i.e., $y_i w^T x_i < 0$ or equivalently $\operatorname{sgn}(w^T x_i) \neq y_i$. We consider $w' := w + y_i x_i$. At least for this one data point,
$$y_i (w')^T x_i = y_i w^T x_i + y_i^2 x_i^T x = y_i w^T x_i + y_i^2 \|x_i\|^2.$$
If $x_i \neq 0$ then $\|x_i\| > 0$ and $y_i (w')^T x_i > y_i w^T x_i$. That is, our Perceptron algorithm pushes $y_i w^T x_i$ towards the positive numbers, which we want.

Logistic Loss

We now look at logistic loss, which is given by
$$F(w) = \frac{1}{n}\sum_{i=1}^{n} \log(1 + \exp(-y_i w^T x_i)).$$
Instead of $\{\pm 1\}$, we "predict the probability", i.e., regress on probability. One way to model probability using the sigmoid function is
$$\mathbb{P}(y = \pm 1 \mid x; w) = \sigma(w^T x) \qquad \text{where} \qquad \sigma(z) = \frac{1}{1 + e^{-z}}.$$
Some immediate properties of the sigmoid function:

- $\lim_{x \to -\infty} \sigma(x) = 0$ and $\lim_{x \to \infty} \sigma(x) = 1$. Monotone increasing and differentiable; good for probability.
- $\sigma(w^T x) \gtrless 0$ if and only if $w^T x \gtrless 0$, consistent with predicting the label with $\operatorname{sgn}(w^T x)$.
- The larger $w^T x$ is, the larger $\sigma(w^T x)$ is, and thus the higher *confidence* in labelling the point as 1.
- $\sigma(z) + \sigma(-z) = 1$ for all z, corresponding to
$$\mathbb{P}(y = -1 \mid x; w) + \mathbb{P}(y = 1 \mid x; w) = \sigma(-w^T x) + \sigma(w^T x) = 1.$$

Because of these, it is natural to model $\mathbb{P}(y \mid x; w)$ by
$$\mathbb{P}(y \mid x; w) := \sigma(y w^T x) = \frac{1}{1 + \exp(-y w^T x)}.$$

MLE on Sigmoid v.s. Minimizing Logistic Loss

What we observe as labels, *not* probabilities. Taking a probabilistic view, we consider "what is the probability of seeing labels $y_1, ..., y_n$ given $x_1, ..., x_n$, when all of these are generated by some w? The **maximum likelihood estimator**, MLE, maximizes the following probability

$$\mathbb{P}(w) = \prod_{i=1}^{N} \mathbb{P}(y_i \mid x_i; w).$$

Some math using monotonicity of $\log$:

$$\begin{aligned} w^* &= \operatorname*{argmax}_w P(w) = \operatorname*{argmax}_w \prod_{i=1}^{N} \mathbb{P}(y_i \mid x_i; w) \\ &= \operatorname*{argmax}_w \sum_{i=1}^{n} \log \mathbb{P}(y_i \mid x_i; w) \\ &= \operatorname*{argmin}_w \sum_{i=1}^{n} -\log \mathbb{P}(y_i \mid x_i; w) \\ &= \operatorname*{argmin}_w \sum_{i=1}^{n} \log(1 + \exp(-y_i w^T x_i)) \\ &= \operatorname*{argmin}_w \sum_{i=1}^{n} \ell_{\text{logistic}}(y_i w^T x_i) = \operatorname*{argmin}_w F(w). \end{aligned}$$

That is, minimizing the logistic loss is *exactly* finding MLE for the sigmoid model.

SGD on Logistic Loss

Recall we can sample one index uniformly at random and make the corresponding loss function on $y_i w^T x_i$ the stochastic gradient. That is,

$$\begin{aligned} w &\leftarrow w - \eta \tilde{\nabla} F(w) \\ &= w - \eta \nabla_w \ell_{\text{logistic}}(y_i w^T x_i) \\ &= w - \eta \cdot \left[\frac{\partial \ell_{\text{logistic}}(z)}{\partial z} \bigg|_{z = y_i w^T x_i} \right] y_i x_i \\ &= w - \eta \cdot \left[\frac{-e^{-z}}{1 + e^{-z}} \bigg|_{z = y_i w^T x_i} \right] y_i x_i \\ &= w + \eta \sigma(-y_i w^T x_i) y_i x_i \\ &= w + \eta \mathbb{P}(-y_i \mid x_i; w) y_i x_i \end{aligned}$$

using our previously defined probability model for $\mathbb{P}(y \mid x; w)$. Note that this is a "soft version" of Perceptron (where $\mathbf{1}[\text{sgn}(w^T x_i) \neq y_i]$ is discontinuous, $\mathbb{P}(-y_i \mid x_i; w)$ is smooth; for the normal Perceptron we either move a lot or stay, whereas in this SGD we always move, though sometimes very little).

Beginning of Sept.8, 2022

2.2 Generalizing ERM

We first make some assumptions before making generalizations:

- The function class $\mathcal{F}$ is assumed to have finite cardinally.
- (Realizability) There exists $f^* \in \mathcal{F}$ such that $y = f^*(x)$ for all $x \in \mathcal{X}$ (e.g. all given labels are indeed separated by a hyperplane).

In some cases, even with the same distribution, we may unfortunately get samples with terrible but working ERMs. For example consider a unit square where the realizable function is a horizontal function cutting the square into two parts. A terrible sample could consist only of possible points on NE and negative points on SW, of which $y = 1 - x$ is also an ERM, although it differs significantly from the actual function. The following theorem addresses such issue and bounds its probability of occurring.

Theorem

Let $\mathcal{F}$ be a function class with $|\mathcal{F}| < \infty$. Let $y = f^*(x)$ for some $f^* \in \mathcal{F}$. Suppose we get a training set, $S = \{(x_i, y_i)\}_{i=1}^n$, drawn i.i.d. from the data distribution D. Define

$$f_S^{\text{ERM}} := \operatorname*{argmin}_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^n \ell(f(x_i), y_i).$$

Then if $n > \log(|\mathcal{F}|/\delta)/\epsilon$, with probability $> 1 - \delta$ over $\{(x_i, y_i)\}_{i=1}^n$, we can bound the risk of f_S^{ERM} by ϵ.

Proof. Note that there exists $f^* \in \mathcal{F}$ with $R(f^*) = 0$ by the realizability assumption. Let

$$\mathcal{F}_{\text{bad}} := \{f \in \mathcal{F} : R(f) \geqslant \epsilon\}.$$

(1) What is the probability of "getting tricked" and mistakenly setting $f \in \mathcal{F}_{\text{bad}}$ as the ERM?

Fix such $f' \in \mathcal{F}_{\text{bad}}$. Consider the following probability:

$$\begin{aligned}
\mathbb{P}_{S \in D^n}[f' \text{ is an ERM}] &= \mathbb{P}_{S \sim D^n}[f^* \text{has zero empirical risk}] \\
&= \mathbb{P}_{S \sim D^n}[f'(x_i) = f^*(x_i) \text{ for all } i \in [n]] \\
\text{(i.i.d. assumption)} &= \prod_{i=1}^n \mathbb{P}_{(x_i, y_i) \sim D}[f'(x_i) = f^*(x_i)] \leqslant (1 - \epsilon)^n
\end{aligned}$$

where in the last step we used the fact that $R(f') \geqslant \epsilon$.

Using $1 - x \leqslant e^{-x}$, we bound the probability by

$$\mathbb{P}_{S \in D^n}[f' \text{ is an ERM}] \leqslant e^{-n\epsilon}.$$

(2) What is the probability of being "tricked" by any $f \in \mathcal{F}_{\text{bad}}$?

By the union bound,

$$\mathbb{P}_{S \sim D^n}\Big[\bigcup_{f \in \mathcal{F}_{\text{bad}}} \{f \text{ is an ERM}\}\Big] \leqslant \sum_{f \in \mathcal{F}_{\text{bad}}} \mathbb{P}(\{f \text{ is an ERM}\}) \leqslant |\mathcal{F}_{\text{bad}}| e^{-n\epsilon} \leqslant |\mathcal{F}| e^{-n\epsilon}.$$

(3) Therefore, if

$$n > \epsilon^{-1}(\log|\mathcal{F}| + \log(1/\delta)) = \frac{\log(|\mathcal{F}|/\delta)}{\epsilon}$$

then

$$\mathbb{P}\Big[\bigcup_{f \in \mathcal{F}_{\text{bad}}} \{f \text{ is an ERM}\}\Big] < \delta,$$

so with probability $> 1 - \delta$, the chosen ERM does not fall into $\mathcal{F}_{\text{bad}}$, in which case $R(f_S^{\text{ERM}}) < \epsilon$. □

Remark. Both assumptions (finite function class cardinality and realizability) can be loosened. We can prove similar theorems which guarantee small generalization gap without realizability (with ϵ^2 as denominator). This is called *agonistic learning*.

Rule of Thumb for Generalization

Suppose the functions f in our function class $\mathcal{F}$ have d parameters to be set. Assume we discretize there parameters so they take W possible values each. In this case, $|\mathcal{F}| = W^d$, so the generalization gap is at most ϵ with $n \geqslant \log(W^d/\delta)/\epsilon = d\log(W/\delta)/\epsilon$ samples.

Remark. To guarantee generalization, the training data size n should be at least linear in d, the number of free parameters.

3 Nonlinear Classifiers

What if our data does not fit a linear model well?
One solution: use a nonlinear mapping $\varphi(x) : \mathbb{R}^r \to \mathbb{R}^m$ to transform the data into a more complicated space then apply linear regression, hoping a linear model would perform better in the new space.
Example: $y = 1 - x^2, x \in [0,1]$ clearly has a parabolic graph. Now we define $y := w^T\varphi(x)$ where

$$w = (1,0,-1) \qquad \text{and} \qquad (1,x,x^2)^T \in \mathbb{R}^3.$$

We claim y is indeed a linear function in $\mathbb{R}^3$.

3.1 Regression with Nonlinear Basis

Model: like mentioned above, we consider $f(x) = w^T\varphi(x)$ where $w \in \mathbb{R}^m$ and $\varphi : \mathbb{R}^d \to \mathbb{R}^m$. Our objective function is

$$\text{RSS}(w) := \sum_{i=1}^{n}(w^T\varphi(x_i) - y_i)^2.$$

Like computed before, the least square solution is similar:

$$w^* = (\varphi^T\varphi)^{-1}\varphi^T y \qquad \text{where} \qquad \varphi = \begin{bmatrix} \varphi(x_1)^T \\ \vdots \\ \varphi(x_n)^T \end{bmatrix} \in \mathbb{R}^{n\times m}.$$

A basic example with $d = 1$ and the polynomial basis for m^{th} order polynomials:

$$\varphi(x) = (1,x,x^2,...,x^m)^T \qquad \text{and so} \qquad f(x)w_0 + \sum_{i=1}^{m} w_m x^m.$$

Remark. What about a fancy linear feature map? Yes, but it's useless since it's equivalent to using a

standard basis:

$$\min_{w\in\mathbb{R}^m}\sum(w^TAx_i-y_i)^2=\min_{w'\in\text{Im}(A^T)\subset\mathbb{R}^d}\sum(w'^Tx_i-y_i)^2.$$

3.2 Overfitting and Regularization

Suppose we have a sine function with a random noise and we have training sample of 10 data points. It is obviously a terrible idea to use a degree 9 polynomial to fit the points even though doing so would result in 0 error on the data points (training error), since doing so will result in huge test error (on points other than the ones in the training set).

In this same example, let m denote the order of polynomial we use to fit the data. If $m\leqslant 2$ we **underfit** the data and have both large training error and test error. On the other hand, if $m=0$, we **overfit** the data in the sense that we have small training error but large test error. In general, the more complicated the model is, the larger the gap between training and test error is.

Question: how to prevent overfitting?

(1) More data! However, this is sometimes costly.

(2) Control model complexity; use **cross-validation** to determine the degree m we want. Rough idea: do a three-way split of training, test, and *validation* set.

(3) **Regularized linear regression**. Instead of the old objective RSS, we now consider

$$G(w)=\text{RSS}(w)+\lambda\psi(w)$$

while also trying to find the minimizer w^*.

Here $\psi:\mathbb{R}^d\to\mathbb{R}^+$, the **regulized**, calculates the complexity of a model and penalizes complex ones. Common choices include $\|w\|_2^2$, $\|w\|_1$, and so on. Also, $\lambda>0$ is the **regularization coefficient**. If $\lambda=0$ there is no regularization. As $\lambda\to\infty$, we have $w^*\to\operatorname{argmin}\psi(w)$.

As a consequence of adding regularization, eventually the gap between training and test error will go down.

Why is regularization useful?

Regularization helps with generalization. In general, we should fit a more expressive model if possible. However, if we don't have sufficient data to fit a more expressive model, then ERM *will* overfit. Regularization addresses this issue and helps us find a balance between "finding a more expressive data" and overfitting.

3.3 ℓ_2 Regularization

We begin by considering **ℓ_2 regularization** (or **ridge regression**) with $\psi(w):=\|w\|_2^2$. In this case,

$$G(w)=\text{RSS}(w)+\lambda\|w\|_2^2=\|Xw-y\|_2^2+\lambda\|w\|_2^2.$$

The gradient is

$$\nabla G(w)=2(X^TXw-X^Ty)+2\lambda w.$$

Setting it to 0 we obtain

$$w^* = (X^T X + \lambda I)^{-1} X^T y.$$

By convexity, this critical point is the global minimizer of G.

When $X^T X$ is not invertible, the least square solution $(X^T X)^{-1} X^T Y$ is undefined. This happens when

(1) There are infinitely many w's with $Xw = Y$, or

(2) No solution exists for $Xw = Y$.

We will focus on the first case here. It corresponds to not having enough data (so $X^T X$ is not full rank), namely when sample size $n < d$. With regularization,

$$G(w) = \|Xw - y\|_2^2 + \lambda \|w\|_2^2 \geqslant \lambda \|w\|_2^2,$$

we pick the w with the smallest 2-norm such that $Xw = Y$.
To be put in a linear algebraic context, doing the eigenvalue decomposition of $X^T X + \lambda I$,

$$X^T X + \lambda I = U^T \{\text{diag}(\lambda_i + \lambda)\} U,$$

we have

$$(X^T X + \lambda I)^{-1} = U^T \{\text{diag}(1/(\lambda_i + \lambda)\} U.$$

A Bayesian View of ℓ_2 Regularization

The **Maximum a Posteriori Probability** estimation (MAP) is a Bayesian generalization of MLE.
Suppose we are given a training set $\{(x_i, y_i)\}_{i=1}^n$ where $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$. We suppose for each i,

$$y_i = w_x^T x_i + \epsilon_i$$

where the noise is Gaussian, $\epsilon_i \in \mathcal{N}(0, \sigma^2)$, so that

$$\mathbb{P}(y \mid x_i, w_x, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y - w_x^T x_i)^2}{2\sigma^2}\right).$$

Taking log-likelihood, we obtain

$$\log \mathbb{P}(y_i) = -\frac{(y_i - w^T x_i)^2}{2\sigma^2} + \text{ constant}.$$

By differentiating, the minimizer agrees with what we have shown before.

The Bayeisan View:
uppose we suspect w will not deviate far from the origin. In particular we assume the prior for w to be $\mathcal{N}(0, \gamma^2 I)$.
Now we find the model which maximizes the posterior probability

$$\text{Posterior}(w) \propto \underbrace{\prod_{i=1}^d \exp\left(-\frac{(w_i)^2}{2\gamma^2}\right)}_{w \sim \mathcal{N}(0, \gamma^2 I)} \prod_{i=1}^n \exp\left(-\frac{(y_i - w^T x_i)^2}{2\sigma^2}\right)$$

and the log posterior is

$$\log \text{Posterior}(w) = -\frac{\|w\|^2}{2\gamma^2} - \sum_{i=1}^n (y_i - w^T x_i)^2.$$

From this, we see that maximizing log of posterior is equivalent to minimizing $G(w)$ for $\varphi(w) = \|w\|^2$.

The Frequentist View:

Instead of using a prior to infer the model, we constrain the model, assuming that such constraints will lead to optimization / the true model:

$$\text{Find } w^* = \operatorname*{argmin}_w \text{RSS}(w) \qquad \text{subject to } \psi(w) \leqslant \beta.$$

3.4 Encouraging Sparsity: the ℓ_0 Regularization

We define the **sparsity** of $w \in \mathbb{R}^n$ to be the number of non-zero coefficients in w, or equivalently in math

$$\|w\|_0 := \lim_{p \to 0} \|w\|_p.$$

Advantage:

- In many applications, we have numerous possible features, only some of which have relationship with our label. The others should have weight 0. Example: genes related to a disease — only a selected few genes may have impact; the others should be ignored while creating a linear model.
- Sparse models can be more interpretable — they narrow down a small number of features which carry a lot of signal.
- Data required to learn sparse model may be significantly less than to learn dense model.

Beginning of Sept.15, 2022

The Good, the Bad, & the Ugly

We choose $\varphi(w) = \|w\|_0$ as stated. The objective is to minimize

$$G(w) = \sum_{i=1}^{n} (w^T x_i - y_i)^2 + \lambda \|w\|_0.$$

- **The good**: "information theoretically" nice — need less data to learn.
  - Suppose the weight in w are in $\{-w, -w+1, ..., 0, ..., w\}$. There are $(2w)^s \binom{d}{s}$ s-sparse (s non-zero entries) vectors in d dimensions.
  - Recall form previous lectures that we need approximately $\log(|\mathcal{F}|)$ (about this magnitude) to learn:
  $$\log\left((2w)^s \binom{d}{s}\right) \sim \log\left((2w)^s \left(\frac{d}{s}\right)^s\right) = s\log(d/s) + s\log(2w).$$
  - How many free parameters?
  - We first choose the nonzero s-coordinates: need about $\log d$ bits per coordinate, so $s \log d$ in total.
  - We need to now choose the value for each non-zero coordinates: fixing s values results in $s \log w$ in total.
  - In contrast, without s-sparsity, we need about d samples in d-dimensions. Therefore, when $s \ll d$, with s-sparsity, we need much less data to learn.
- **The bad**: $\|w\|_0$ is not convex. (Any $\|\cdot\|_p$ for $0 < p < 1$ is non-convex too.) Furthermore, minimizing $G(w)$ is NP-hard, so no guaranteed efficient minimization algorithm.
- **The ugly**: $\|w\|_0$ is highly discontinuous (for example on $\mathbb{R}$), so GD does not work.

3.5 ℓ_1 Regularization as a Proxy of ℓ_0

We now instead consider $\varphi(w) = \|w\|_1$:

$$G(w) = \sum_{i=1}^{n}(w^T x_i - y_i)^2 + \lambda\|w\|_1.$$

Note $\|w\|_1$ is convex, and we can use GD/SGD. Furthermore, to minimize $\|w\|_1$, it often suffices to minimize $\|w\|_0$, as stated in the following theorem:

Theorem

Given n vectors $x_i \in \mathbb{R}^d$, $i \in [n]$, drawn i.i.d. from $\mathcal{N}(0, I)$, let $y_i = (w^*)^T x$ for some w^* with $\|w^*\|_0 = s$. Then for some fixed $C > 0$, the minimizer of $G(w)$ with ℓ_1 regularization will be w^* as long as $n > C \cdot s \log d$ (with high probability over the randomness in the training data points x_i).

3.6 Isotropic ℓ_1 and ℓ_2 Regularization

We assume that $X^T X = I$ (isotropic assumption), meaning informally that

- all features are uncorrelated,
- all features have mean 0, and
- all features have variance 1.

Let us consider $\|\cdot\|_2$ first:

$$G(w) = \sum_{i=1}^{n}(x_i^T w - y_i)^2 + \lambda\|w\|_2^2 \implies w^* = (X^T X + \lambda I)^{-1} X^T y.$$

With our isotropic assumption $X^T X = I$, we have

$$w^* = \frac{1}{1+\lambda} X^T y$$

so the i^{th} coordinate of w^* iks simply $X_i^T y/(1+\lambda)$, namely $1/(1+\lambda)$ times the correlation of the i^{th} feature with the label. We see that, in this special case, ℓ_2 regularization shrinks the estimated parameters.
More generally, when the features have unequal variance, ℓ_2 regularization applies similar shrinkage to all of them so scaling features can be important.

Now we consider $\|\cdot\|_1$:

$$G(w) = \sum_{i=1}^{n}(x_i^T w - y_i)^2 + \lambda\|w\|_1.$$

The gradient is (ignoring zero components of w)

$$\begin{aligned}
\frac{\partial G(w)}{\partial w_j} &= 2\sum_{i=1}^{n}(x_i^T w - y_i)^2 x_i^{(j)} + \lambda\,\mathrm{sgn}(w_j) \\
&= 2\sum_{i=1}^{n}(x_i^{(j)} x_i^T w) - 2\sum_{i=1}^{n} x_i^{(j)} y_i + \lambda\,\mathrm{sgn}(w_j) \\
&= 2\sum_{i=1}^{n} x_i^{(j)} x_i^T w - 2x_i^T y + \lambda\,\mathrm{sgn}(w_j) \\
&= 2w_j - 2x_i^T y + \lambda\,\mathrm{sgn}(w_j)
\end{aligned}$$

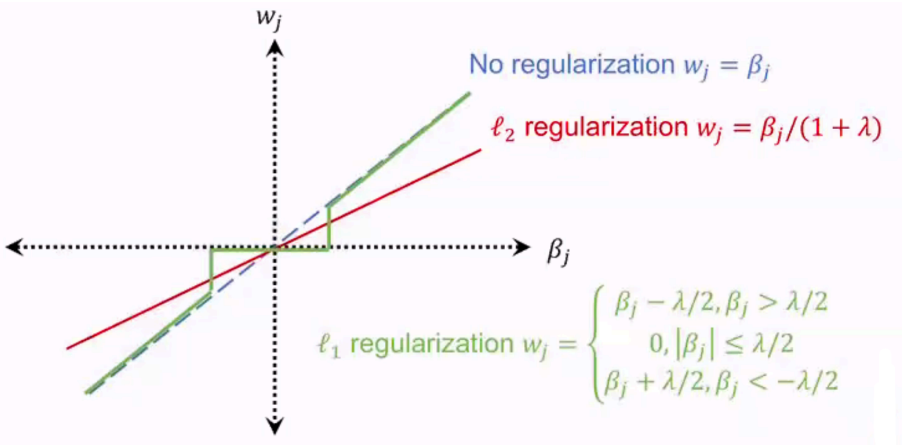
where $x_i^{(j)}$ is the j^{th} component of x_i. For GD, we literately define

$$w_j \leftarrow w_j - \eta(2(w_j - x_i^T y) + \lambda \operatorname{sgn}(w_j)).$$

Without regularization we would have

$$w_j \leftarrow w_j - 2\eta(w_j - x_i^T y).$$

Here, as $x_i^T y$ approaches w_j, the change of each iteration becomes very small. However, with $\eta\lambda \operatorname{sgn}(w_j)$ present, when $w_j > 0$, this extra term pushes w_j towards the negative side and vice versa, until w_j becomes 0.
A geometric interpretation of no regularization, ℓ_1, and ℓ_2 regularizations is shown below. Note how large, positive w_j under ℓ_2 is slightly above the blue line and how large (magnitude), negative w_j under ℓ_2 regularization is slightly under the blue line. Also note when $|\beta_j|$ is sufficiently small, ℓ_1 regularization forces w_j to be 0, thereby giving us sparsity as ideally used in $\|\cdot\|_0$.



Bias-Variance Tradeoff

The phenomenon of underfitting and overfitting is often referred to as the **bias-variance tradeoff**. A model whose complexity is too small will underfit and cause a large bias because the model's accuracy will not improve with addition of data. Conversely, a model with overly high complexity will overfit and create high variance, because the model's predictions vary significantly with the randomness in its training data.

4 Support Vector Machines, SVMs

4.1 The Kernel Trick

Recall our method of mapping a nonlinear function map to a linear regression. The **kernel methods** give a way to choose and efficiently work with the nonlinear map $\varphi : \mathbb{R}^d \to \mathbb{R}^m$.
Recall the regularized least squares: we minimize

$$w^* = \operatorname*{argmin}_w F(w) = \operatorname*{argmin}_w (\|\Phi w - y\|_2^2 + \lambda \|w\|_2^2) = (\Phi^T \Phi + \lambda I)^{-1} \Phi^T y$$

where

$$\Phi = \begin{bmatrix} \varphi(x_1)^T \\ \vdots \\ \varphi(x_n)^T \end{bmatrix} \in \mathbb{R}^{nM} \qquad \text{and} \qquad y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \in \mathbb{R}^n.$$

Looking back at ℓ_2 regularization LS,

$$F(w) = \|\Phi w - y\|_2^2 + \lambda \|w\|_2^2,$$

with

$$\nabla F(w) = \Phi^T(\Phi w^* - y) + \lambda w^* \implies w^* = \lambda^{-1}\Phi^T(y - \Phi w^*) =: \Phi^T \alpha = \sum_{i=1}^n \alpha_i \varphi(x_i).$$

That is, w^* is a linear combination of $\varphi(x_i)$. (Note that we do not know what α is from this computation though.)

Suppose we know α. Then the prediction of w^* on a new example x is

$$(w^*)^T \varphi(x) = \sum_{i=1}^n (\alpha_i \varphi(x_i))^T \varphi(x) = \sum_{i=1}^n \alpha_i \varphi(x_I^T \varphi(x).$$

Therefore, only inner products in the new feature space matter — we can compute inner products without explicitly computing φ.

But of course, doing so requires us to know what α is.

Step 1: kernel matrix. Plugging in $w = \Phi^T \alpha$ into $F(w)$ gives

$$\begin{aligned} H(\alpha) = F(\Phi^T \alpha) &= \|\Phi\Phi^T \alpha - y\|_2^2 + \lambda \|\Phi^T \alpha\|_2^2 \\ &= \|Ka - y\|_2^2 + \lambda \alpha^T \Phi\Phi^T \alpha = \|K\alpha - y\|_2^2 + \lambda \alpha^T K \alpha. \qquad [K = \Phi\Phi^T \in \mathbb{R}^{n^2}] \end{aligned}$$

K is called the **Gram matrix** with $K_{i,j} = \varphi(x_i)^T \varphi(x_j)$.

Note this is different from the covariance matrix (the $M \times M$ matrix $\Phi^T\Phi$), but both are indeed symmetric and PSD.

Step 2: minimize the dual information

$$H(\alpha) = \|K\alpha - y\|_2^2 + \lambda \alpha^T K \alpha.$$

Setting the derivative to 0 we have

$$0 = (K^2 + \lambda K)\alpha - Ky = K((K + \lambda I)\alpha - y).$$

Therefore $\alpha = (K + \lambda I)^{-1} y$ is a minimizer of H and we obtain

$$\boxed{w^* = \Phi^T \alpha = \Phi^T(\Phi\Phi^T + \lambda I)^{-1} y.}$$

We now have found two forms of minimizers of $F(w)$, w^*:

- Minimizing $F(w)$ gives $w^* = (\Phi^T\Phi + \lambda I)^{-1}\Phi^T y$.
- Minimizing $H(\alpha)$ gives $w^* = \Phi^T(\Phi\Phi^T + \lambda)^{-1} y$ (different dimension of I).

These two are indeed identical (and they have to, since F is convex), since

$$\begin{aligned} (\Phi^T\Phi + \lambda I)^{-1}\Phi^T y &= (\Phi^T\Phi + \lambda I)^{-1}\Phi^T(\Phi\Phi^T + \lambda I)(\Phi\Phi^T + \lambda I)^{-1} y \\ &= (\Phi^T\Phi + \lambda I)^{-1}(\Phi^T\Phi\Phi^T + \lambda\Phi^T)(\Phi\Phi^T + \lambda I)^{-1} y \\ &= (\Phi^T\Phi + \lambda I)^{-1}(\Phi^T\Phi + \lambda I)\Phi^T(\Phi\Phi^T + \lambda I)^{-1} y \\ &= \Phi^T(\Phi\Phi^T + \lambda I)^{-1} y. \end{aligned}$$

The Kernel trick:

- Computing $(\Phi\Phi^T + \lambda I)^{-1}$ is more efficient than computing $(\Phi^T\Phi + \lambda I)^{-1}$ when $n \leqslant M$: since $\Phi\Phi^T$ is $n \times n$, inversion takes $\mathcal{O}(n^3)$ time as opposed to inverting $\Phi^T\Phi$, which takes $\mathcal{O}(M^3)$.
- More importantly, computing $\alpha = (K + \lambda I)^{-1} y$ only requires computing inner products in the new feature space. That is,
- The exact form of $\varphi(\cdot)$ is not essential. All we need is the inner products $\varphi(x)^T\varphi(x')$.
- For some φ it is indeed possible to compute this quantity without knowing what φ looks like. This is the **kernel trick**.

Example. Consider $\varphi : \mathbb{R}^2 \to \mathbb{R}^3$ by

$$\varphi(x_1, x_2) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}.$$

The inner product of $\varphi(x)$ and $\varphi(y)$ is

$$\varphi(x)^T\varphi(x') = x_1^2y_1^2 + 2x_1x_2y_1y_2 + x_2^2y_2^2 = (x_1y_1 + x_2y_2)^2 = (x^Ty)^2.$$

That is, the inner product in the new space ($\mathbb{R}^3$) is s imply a function of the inner product in the original space ($\mathbb{R}^2$).

Example. Consider $\varphi : \mathbb{R}^d \to \mathbb{R}^{2d}$ parametrized by θ:

$$\varphi_\theta(x) = [\cos(\theta x_1), \sin(\theta x_1), ..., \cos(\theta x_d), \sin(\theta x_d)]^T.$$

We have

$$\varphi_\theta(x)^T\varphi_\theta(y) = \sum_{i=1}^{d}[\cos(\theta x_i)\cos(\theta y_i) + \sin(\theta x_i)\sin(\theta y_i)] = \sum_{i=1}^{d}\cos(\theta(x_i - y_i)).$$

In this example, the inner product in the new space is a simple function of features in the original space.

Example. Consider $\varphi_L : \mathbb{R}^d \to \mathbb{R}^{2d(L+1)}$ for some integer L:

$$\varphi_L(x) = \begin{bmatrix} \varphi_0(x) \\ \varphi_{2\pi/L}(x) \\ \varphi_{4\pi/L}(x) \\ \vdots \\ \varphi_{2\pi L/L}(x) \end{bmatrix}.$$

Here

$$\varphi_L(x)^T\varphi_L(y) = \sum_{\ell=0}^{L}\sum_{i=1}^{d}\cos\left(\frac{2\pi\ell}{L}(x_i - y_i)\right).$$

Example. As $L \to \infty$, we cannot compute $\varphi(x)$, but the inner product still makes sense:

$$\varphi_\infty(x)^T \varphi_\infty(y) = \int_0^{2\pi} \sum_{i=1}^{d} \cos(\theta(x_i - y_i)) \, \mathrm{d}\theta = \sum_{i=1}^{d} \frac{\sin(2\pi)(x_i - y_i)}{x_i - y_i},$$

again, a function of the original features.

With these illustrations, we now formally define the kernel functions:

Definition: Kernel Functions

A function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is called a **kernel function** if there exists a function $\varphi : \mathbb{R}^d \to \mathbb{R}$ such that for any $x, y \in \mathbb{R}^d$,

$$k(x, y) = \varphi(x)^T \varphi(y).$$

Now, choosing a nonlinear basis φ becomes equivalent to choosing a kernel function. The Gram matrix now becomes

$$K = \Phi\Phi^T = \{\varphi(x_i)^T \varphi(x_j)\} = \{k(x_i, x_j)\}.$$

Mercer's theorem states that k is a kernal if and only if K is PSD for any n and any $x_1, ..., x_n$. The contrapositive is often used to prove something is not a kernel.

Example. The function $k(x, y) = \|x - y\|_2^2$ is not a kernel, because the matrix

$$K = \begin{bmatrix} k(x,x) & k(x,y) \\ k(y,x) & k(y,y) \end{bmatrix} = \begin{bmatrix} 0 & \|x_1 - x_2\|_2^2 \\ \|x_1 - x_2\|_2^2 & 0 \end{bmatrix}$$

is not PSD. (For example $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ is not because $\begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = -2$.)

Properties of kernels:

- For any $f : \mathbb{R}^d \to \mathbb{R}$, $k(x, y) = f(x)f(y)$ is a kernel (with feature map $\varphi \equiv f$).
- If k_1, k_2 are kernels, the following are also kernels (prove by using feature maps):
  - $\alpha k_1 + \beta k_2$ for $\alpha, \beta \geqslant 0$,
  - $k_1 k_2$, and
  - $\exp(k_1)$ and $\exp(k_2)$.

Popular Kernels

The **polynomial kernel** is given by

$$k(x, y) = (x^T y + c)^M$$

for some $c \geqslant 0$ and positive integer M. We've seen the feature map for $c = 0$ and $M = 2$ before.

The **Gaussian kernel** or the **Radial basis function (RBF) kernel** is

$$k(x,y) = \exp\left(-\frac{\|x-y\|_2^2}{2\sigma^2}\right) \qquad \text{for some } \sigma > 0.$$

Note that

$$k(x,y) = \exp\left(-\frac{\|x\|_2^2}{2\sigma^2}\right)\exp\left(-\frac{\|y\|_2^2}{2\sigma^2}\right)\exp\left(\frac{x^Ty}{\sigma^2}\right)$$

where the product of the first two terms can be seen as $f(x)f(y)$ where $f(x) \coloneqq \exp(-\|x\|_2^2/(2\sigma^2))$. So, without the third term, k would have been a kernel with feature map f.
Now we look at the last term using $e^x = 1 + x + x^2/2! + x^3/3! + \ldots$:

$$\exp\left(\frac{x^Ty}{\sigma^2}\right) = 1 + \frac{x^Ty}{\sigma^2} + \frac{(x^Ty)^2}{2!(\sigma^2)^2} + \frac{(x^Ty)^3}{3!(\sigma^2)^3} + \ldots$$

Each term on the RHS is a polynomial, and we claim that this infinite sum also results in a kernel. Then, the original $k(x,y)$ is again the product of two kernels and is therefore a kernel.

Prediction with Kernels

As long as $w^* = \sum_{i=1}^n \alpha_i \varphi(x_i)$, the prediction on a new example x becomes

$$(w^*)^T\varphi(x) = \sum_{i=1}^n \alpha_i \varphi(x_i)^T\varphi(x) = \sum_{i=1}^n \alpha_i k(x_i,x).$$

This is known as a **non-parametric** method. Informally speaking, this means that there is no fixed set of parameters that the model is trying to learn, since the RHS is free of w^*.

Beginning of Sept. 22, 2022

4.2 Support Vector Machines, Separable Case

Why SVM?

- It is one of the most commonly used classification algorithms.
- It allows us to explore the concept of *margins* in classification.
- It works well with the kernel trick.
- It has strong theoretical guarantees.

We will again focus on binary classification here. The function class for SVMs is a function on a feature map φ applied to the data points, namely $\operatorname{sgn}(w^T\varphi(x) + b)$. The bias term b is taken separately for SVMs, the reason of which will be explained later.

Margins: Geometric Intuition

When data is linearly separable, there are infinitely many hyperplanes with zero training error. Back to the same old question — which one should we choose? We claim that the further away the separating hyperplane is from the data points, the better. To this end, we define the **margin** for linearly separable data to be the distance from the hyperplane the closest point.

Some math first: given x and a hyperplane $\{x : w^T x + b = 0\}$, we compute the distance as follows:

- Orthogonally project x onto the hyperplane and obtain x'.
- Since w is orthogonal to the hyperplane, x' is of form $x' = x - \beta w/\|w\|_2$.
- Find β using $w^T x' + b = 0$, namely

$$0 = w^T(x - \beta w/\|w\|_2) + b = w^T x - b\|w\| + b \implies \beta = \frac{w^T x + b}{\|w\|_2}.$$

- The distance is then $\|x - x'\|_2 = |\beta| = |w^T x + b|/\|w\|_2$.
- More generally, if the hyperplane classifies (x, y), then $\operatorname{sgn}(w^T x + b) = y$, so the distance becomes

$$\frac{y(w^T x + b)}{\|w\|_2}.$$

Margins: Functional Motivation

Recall from previous lectures that we used the signoid function to show the probability of a data point getting 1 or 0 by

$$\mathbb{P}(y = 1 \mid x, w) = \sigma(y(w^T x + b)) = \frac{1}{1 + \exp(-y(w^T x + b))}.$$

Here, if $y = 1$, we want $w^T x + b \gg 0$ and if $y = -1$, we want $w^T x + b \ll 0$. Hence we want $y(w^T x + b) \gg 0$. However, we can easily "cheat" by making w large. To offset this potential effect, we normalize the quantity and instead try to maie $(y(w^T x + b))/\|w\|$ as large as possible.

Maximizing Margin

The formal definition of a margin distance from all trainig points is

$$\min_i \frac{y_i(w^T \varphi(x_i) + b)}{\|w\|_2} \qquad \text{for data points} \qquad (x_i, y_i).$$

Since we want to maximize the smallest distance among all data points, this translates to solving

$$\max_{w,b} \min_i \frac{y_i(w^T \varphi(x_i) + b)}{\|w\|_2} = \max_{w,b} \frac{1}{\|w\|_2} \min_i y_i(w^T \varphi(x_i) + b).$$

Note that if we rescale (w, b), multiplying both by some scalar, the hyperplane remains the same, i.e.,

$$\{x : w^T \varphi(x) + b = 0\} = \{x : kw^T \varphi(x) + kb = 0\}.$$

We can pick the appropriate quantity so that $\min_i y_i(w^T \varphi(x_i) + b) = 1$. More concretely, this scalar is $\dfrac{1}{\min_i y_i(w^T \varphi(x_i) + b)}$. After rescaling, the margin simply becomes $1/\|w\|_2$.

SVM for Separable Data: "Primal" Formulation

From above, on a separable training set, lour goal is to solve

$$\max_{w,b} \frac{1}{\|w\|_2} \qquad \text{subject to} \qquad \min_i y_i(w^T\varphi(x_i)+b) = 1.$$

This is equivalent to solving

$$\min_{w,b} \frac{1}{2}\|w\|_2^2 \qquad \text{subject to} \qquad y_i(w^T\varphi(x_i)+b) \geqslant 1 \text{ for all } i \in [n]. \tag{SVM1}$$

In doing so, we transformed a non-convex objective into a convex one! Because of the new formulation, SVM is also called the **max-margin** classifier. The constraints above are called **hard-margin constraints**.

4.3 Support Vector Machines: General Non-Separable Case

If the data are not linearly separable, the previous constraint

$$y_i(w^T\varphi(x_i)+b) \geqslant 1 \qquad \text{for all} \qquad 1 \leqslant i \leqslant n$$

is obviously not feasible.

In fact, even in the separable case, sometimes it is *not* the best idea to completely separate them. Consider, for example, a rectangle, in which all blue dots are in bottom right. Almost all red points are in top left, so an "ideal" classifier would separate the rectangle diagonally. But if we have an extra red point just on top of the blue ones, this forces a horizontal separating line, and clearly this is not what we want. (Though we successfully classify all training data.)

To deal with this issue, we relax the constraints to ℓ_1 **norm soft-margin** constraints:

$$y_i(w^T\varphi(x_i)+b) \geqslant 1-\xi_i \iff 1-y_i(w^T\varphi(x_i)+b) \leqslant \xi_i, \qquad i \in [n].$$

Here we introduce the **slack variables** $\xi_i \geqslant 0$. (This should ring a bell on hinge loss $\ell_{\text{hinge}}(z) = \max\{0, 1-z\}$ with $z = (w^T\varphi(x)+b)$.)

But Why ℓ_1 Penalization?

Functions like squared hinge loss really penalizes a misclassification as $\max(0,1-z)^2$ grows very quickly as z gets increasing negative. Hinge loss itself, however, is much more robust compared to squared loss when facing outliers in data.

Example: A one-dimensional example. Suppose we have $x_1, x_2, ..., x_n$. We know

$$w^*_{\ell_2} = \operatorname*{argmin}_w \sum_{i=1}^n (x_i - w)^2 = \frac{1}{n}\sum_{i=1}^n x_i.$$

On the other hand,

$$w^*_{\ell_1} = \operatorname*{argmin}_w \sum_{i=1}^n |x_i - w| = \text{median of } \{x_1, ..., x_n\}.$$

It is obvious that the median is much more robust to outliers — one significant outlier shifts the mean significantly but has little impact on the median.

Example: For one-dimensional regression. Suppose the data points are nice enough to satisfy $y_i = 10x_i + \epsilon_i$ (ϵ for noise). Consider

$$w^*_{\ell_2} = \operatorname*{argmin}_w \sum_{i=1}^n (y_i - wx_i)^2 \qquad \text{and} \qquad w^*_{\ell_1} = \operatorname*{argmin}_w \sum_{i=1}^n |y_i - wx_i|.$$

What if we add an outlier now? $w^*_{\ell_2}$ will shift much more than $w^*_{\ell_1}$ does.

SVM: General Primal Formulation

Ideally, we want ξ_i to be as small as possible. The objective therefore becomes

$$\min_{w,b,\{x_i\}} \frac{1}{2}\|w\|_2^2 + C\sum_i \xi_i \qquad \text{subject to} \qquad y_i(w^T\varphi(x_i) + b) \geqslant 1 - \xi_i$$
$$\xi_i \geqslant 0. \tag{SVM2}$$

- When $\xi_i = 0$, the data is classified correctly.
- When $\xi_i < 1$, the data is classified correctly but does not satisfy the large margin constraint.
- When $\xi_i > 1$, the data is misclassified.

Primal Formulation: Another View

In a nutshell: SVM can be thought of as a linear model with ℓ_2 regularized hinge loss. We claim (SVM2) is equivalent to finding

$$\min_{w,b,\{\xi_i\}} \frac{1}{2}\|w\|_2^2 + C\sum_{i=1}^n \xi_i \qquad \text{subject to} \qquad \max\{0, 1 - y_i(w^T\varphi(x_i) + b)\} = \xi_i \text{ for } i \in [n]. \tag{SVM3}$$

In order to minimize $\sum \xi_i$, we should set ξ_i to be as small as possible. When if $\xi_i = 0$ we want $y_i(w^T\varphi(x_i) + b) \geqslant 1$; when $\xi_i > 0$ we want it to be precisely $1 - y_i(w^T\varphi(x_i) - b)$). But then we can rewrite (SVM3) as

$$\min_{w,b} C\sum_{i=1}^n \max\{0, 1 - y_i(w^T\varphi(x_i) + b)\} + \frac{1}{2}\|w\|_2^2$$

which, after setting $\lambda = 1/C$, gives the form identical to minimizing ℓ_2 regularized hinge loss:

$$\min_{w,b} \sum_{i=1}^n \max\{0, 1 - y_i(w^T\varphi(x_i) + b)\} + \frac{\lambda}{2}\|w\|_2^2. \tag{SVM4}$$

4.4 Optimizing / Kernelizing SVM

We now go back to (SVM2), the convex objective. We can apply any convex optimization algorithms (e.g. SGD), but usually we apply kernel trick, which requires solving the **dual problem**. It suffices to show that w^*, the solution, is a linear combination of the feature vectors $\varphi(x_i)$.

Proposition

We claim that, for SVM, w^* *is* a linear combination of $\varphi(x_i)$, i.e., $w^* = \sum_{i=1}^n \alpha_i y_i \varphi(x_i)$ for some α_i's.

Informal proof. We first formulate the SVM question as a linear model $F(w)$ with ℓ_2 regularized hinge loss as in (SVM4). Since the function is convex, GD is guaranteed to find a minimizer with any initialization (with appropriate learning rate).
Taking derivatives,

$$\frac{\partial F(w)}{\partial w} = \sum_{i=1}^n \left(\frac{\partial \ell_{\text{hinge}}(z)}{\partial z}\bigg|_{z=y_i(w^T\varphi(x_i)+b)} \cdot \overbrace{(-y_i\varphi(x_i))}^{\text{chain rule}} \right) + \lambda w. \tag{Δ}$$

Therefore, we can initialize $w^{(0)}$ as 0 and iteratively define

$$w^{(t+1)} \leftarrow w^{(t)} - \eta(\Delta).$$

Observe that $w^{(t)}$ is always in the span of $y_i\varphi(x_i)$: every time when we update, we are adding a linear combination of the $y_i\varphi(x_i)$'s. Taking the limit we see w^* also lies in the span. □

Having shown $w^* = \sum_i \alpha_i y_i \varphi(x_i)$, in the separable case, minimizing $\|w\|_2^2/2$ becomes

$$\min_{\alpha,t} \frac{1}{2}\Big\|\sum_{i=1}^n \alpha_i y_i \varphi(x_i)\Big\|_2^2 \qquad \text{subject to} \qquad y_i\Big(\sum_{i=1}^n \alpha_i y_i \varphi(x_i)^T \varphi(x_i) + b\Big) \geqslant 1.$$

This is equivalent to

$$\min_{\alpha_i,b} \frac{1}{2}\sum_{i,j} \alpha_i\alpha_j y_i y_j \varphi(x_i)^T\varphi(x_j) \qquad \text{subject to} \qquad y_i\Big(\sum_{i=1}^n \alpha_i y_i \varphi(x_i)^T \varphi(x_j) + b\Big) \geqslant 1.$$

Using **Lagrange duality** (not covered), for the separable case, the objective above is equivalent to

$$\max_{\alpha_i} \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i\alpha_j y_i y_j \varphi(x_i)^T\varphi(x_j) \qquad \text{subject to} \qquad \sum_{i=1}^n \alpha_i y_i = 0 \text{ and } \alpha_i \geqslant 0.$$

In particular, we no longer need to compute $\varphi(x)$, and the objective is quadratic.
For the general case, the dual is identical except the extra requirement that $0 \leqslant \alpha_i \leqslant C$.

4.5 Prediction Using SVM

How do we predict, given the solution $\{\alpha_i^*\}$ to the optimization problem?
Remember

$$w^* = \sum_{i=1}^n \alpha_i^* y_i \varphi(x_i) = \sum_{i:\alpha_i^*>0} \alpha_i^* y_i \varphi(x_i).$$

That is, we ignore indices with $\alpha_i = 0$. A point $\varphi(x_i)$ with $\alpha_i^* > 0$ is called a **support vector** hence the name SVM.

To make a prediction on any data point x:

$$\begin{aligned}\operatorname{sgn}(w^{*T}\varphi(x) - b^*) &= \operatorname{sgn}\left(\sum_{\alpha_i^*>0} \alpha_i^* y_i \varphi(x_i)^T \varphi(x) - b^*\right) \\ &= \operatorname{sgn}\left(\sum_{\alpha_i^*>0} \alpha_i^* y_i k(x_i, x) - b^*\right)\end{aligned}$$

with the help of kernels.

Bias Term b^*

It can be shown that, in the separable case, the support vectors lie on the margin.
In this case, for any i with $\alpha_i^* > 0$,

$$y_i^2(w^{*T}\varphi(x_i) + b^*) = y_i \implies w^{*T}\varphi(x_i) + b^* = y_i \implies b^* = y_i - w^{*T}\varphi(x_i).$$

In the general case, for any support vector $\varphi(x_i)$ with $0 < \alpha_i^* < C$, it can be shown that $y_i(w^{*T}\varphi(x_i) + b^*) = 1$ so

$$b^* = y_i - w^{*T}\varphi(x_i) = y_i - \sum_{j=1}^{n} \alpha_j^* y_j k(x_i, x_j).$$

With α^* and b^* known, we can make a prediction on any data point as stated above.

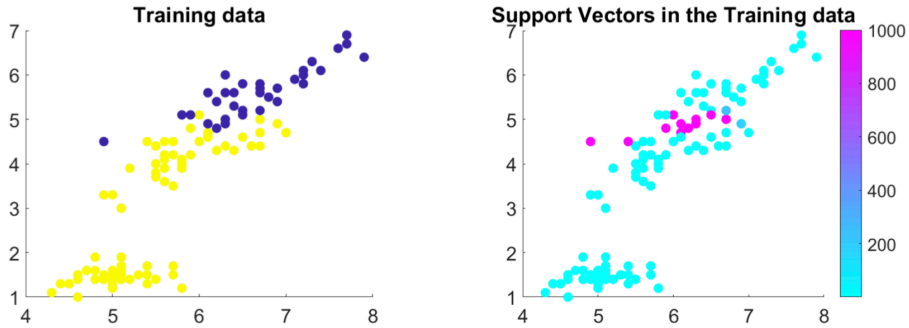
4.6 Understanding SVM

Straight from definition, support vectors are $\varphi(x_i)$'s with $\alpha_i^* > 0$. We can show that they are precisely the points satisfying one of the following:

- They lie on the large margin boundary. Consequently $\xi_i^* = 0$, so $y_i(w^{*T}\varphi(x_i) + b^*) = 1$ and the point is precisely $1/\|w^*\|_2$ away from the hyperplane.
- They do not satisfy the large margin constraint. This is when $\xi_i^* < 1$ (still classified correctly).
- They are misclassified, corresponding to $\xi_i^* > 1$.

All other points (i.e., those classified correctly and not on the margin boundary) have $\alpha_i^* = 0$ and are therefore not support vectors.

A potential drawback of the kernel method is that it's non-parametric and needs to sometimes keep track of all training data. SVM, however, usually have $|\{i : \alpha_i^* > 0\}| \ll n$, thereby avoiding this issue.



Beginning of Sept. 29, 2022

5 Multiclass Classification

The setup:

- Input: feature vectors from $x \in \mathbb{R}^d$
- Output: labels, $y \in [C]$
- Goal: learn a mapping $f : \mathbb{R}^d \to [C]$.
- Examples: recognizing digits ($C = 10$) or letters ($C = 26$); predicting weathers.

5.1 Linear models: Binary to Multiclass

- **Step 1**. What should a linear model look like for multiclass tasks?

  Note that linear model for binary tasks (we previously used $\{\pm 1\}$; now use $\{1, 2\}$)

  $$f(x) = \begin{cases} 1 & \text{if } w^T x \geqslant 0 \\ 2 & \text{otherwise} \end{cases}$$

  can be re-written as

  $$f(x) = \begin{cases} 1 & \text{if } w_1^T x \geqslant w_2^T x \\ 2 & \text{if } w_1^T x < w_2^T x \end{cases} = \operatorname*{argmax}_{k=1,2} w_k^T x$$

  for any w_1, w_2 such that $w = w_1, w_2$. We can think of $w_k^T x$ as a **score** for class k.

- **Step 2**. Using the transformed notation above, we can easily add another vector, w_3, and define the label using

  $$f(x) = \operatorname*{argmax}_{1 \leqslant k \leqslant 3} w_k^T x.$$

- **Step 3**. More generally: the function class is given by

  $$\mathcal{F} = \{f(x) = \operatorname*{argmax}_{k \in [C]} w_k^T x : w_k \in \mathbb{R}^d\} = \{f(x) = \operatorname*{argmax}_{k \in [C]} (Wx)_k : W \in \mathbb{R}^{C \times d}\}.$$

5.2 Generalizing Logistic Loss to Multiclass

First note that, with $w = w_1, w_2$, we have

$$\mathbb{P}(y = 1 \mid x; w) = \sigma(w^T x) = \frac{1}{1 + e^{-w^T x}} = \frac{e^{w_1^T x}}{e^{w_1^T x} + e^{w_2^T x}} \propto e^{w_1^T x}.$$

Similarly,

$$\mathbb{P}(y = 2 \mid x; w) \propto e^{w_2^T x}.$$

(Proportionality is because the denominator for all w_k's are the same.)

More generally, with C labels,

$$\mathbb{P}(Y = k \mid x; W) = \frac{\exp(w_k^T x)}{\sum_{j\in[C]} \exp(w_j^T x)} \propto \exp(w_k^T)x.$$

This function $w_k^T x \mapsto \mathbb{P}(y = k \mid x; w)$ is called the **softmax function**.

Now, to find the MLE, given labels $y_1, ..., y_n$ and data points $x_1, ..., x_n$,

$$\mathbb{P}(W) = \prod_{i=1}^{n} \mathbb{P}(y_i \mid x_i; W) = \prod_{i=1}^{n} \frac{\exp w_{y_i}^T x_i}{\sum_{k\in[C]} \exp(w_k^T x_i)}.$$

Taking *negative* log, maximizing above is equivalent to minimizing the **multiclass logistic loss**

$$F(W) = \sum_{u=1}^{n} \log\left(\frac{\sum_{k\in[C]} \exp(w_k^T x_i)}{\exp(w_{y_i}^T x_i)}\right) = \sum_{k=1}^{n} \log\left(1 + \sum_{k\neq y_i} \exp((w_k - w_{y_I})^T x_i\right). \tag{*}$$

This is an upper bound for the **0 – 1 misclassification loss**:

$$\mathbf{1}[f(x) \neq y] \leqslant \log_2\left(1 + \sum_{k\neq y} \exp((w_k - w_y)^T x)\right).$$

In particular, for $C = 2$, if $y_i = 1$, this becomes $\log(1 + \exp(-(w_2 - w_1)^T x_i))$, and if $y_i = 2$, this becomes $\log(1 + \exp(-(w_1 - w_2)^T x_i))$. Then, for $w = w_1 - w_2$, after transforming labels fro $\{1, 2\}$ to $\{1, -1\}$, we obtain

$$F(w) = \sum_{i=1}^{n} \log(1 + \exp(-y_i w^T x_i)).$$

5.3 Optimizing Logistic Loss

As usual, we apply SGD to $F(W)$ as defined in (*). It is a $C \times d$ matrix. We first focus on the k^{th} row:
If $k \neq y$,

$$\nabla_{w_k^T} F(W) = \frac{\exp(w_k - w_{y_i})^T x_i}{1 + \sum_{k\neq y_i} \exp((w_k - w_{y_i})^T x_i)} x_i^T = \frac{\exp(w_k^T x_i)}{\exp(w_{y_i}^T x_i) + \sum_{k\neq y_i} \exp(w_k^T x_i)} x_i^T = \mathbb{P}(y = k \mid x_i; W) x_i^T,$$

and if $k = y_i$,

$$\nabla_{w_k^T} F(W) = \frac{-\sum_{k\neq y_i} \exp((w_k - w_{y_i})^T x_i)}{1 + \sum_{k\neq y_i} \exp((w_k - w_{y_i})^T x_i)} x_i^T = \frac{-\sum_{k\neq y_i} \exp(w_k^T x_i)}{\exp(w_{y_i}^T x_i) + \sum_{k\neq y_i} \exp(w_k^T x_i)} x_i = (\mathbb{P}(y = k = y_i \mid x_i; W) - 1) x_i^T.$$

Step for SGD on multinomial logistic regression:
Initialize $W = 0$ or randomly. Repeat:

(1) Pick $i \in [n]$ randomly, and

(2) Update the parameters

$$W \leftarrow W - \eta \begin{bmatrix} \mathbb{P}(Y = 1 \mid x_i; W) \\ \vdots \\ \mathbb{P}(y = y_i \mid x_i; W) - 1 \\ \vdots \\ \mathbb{P}(y = c \mid x_i; W) \end{bmatrix} x_i^T.$$

Intuition: if we are sure that $y = y_i$, then $\mathbb{P}(y = y_i) - 1$ is zero, namely, we are not moving anything at all. Conversely, if we are very sure $y \neq y_i$, we shift by a large amount towards the correct one.
Having learned W, we can either (i) make a *deterministic* prediction $\operatorname{argmax}_{k\in[C]} w_k^T x$, or we can (ii) make a *randomized* prediction according to $\mathbb{P}(y = k \mid x; W) \propto \exp(w_k^T x)$.

5.4 Beyond Linear Models

Suppose we have any model f (instead of inner product $w_k^T x$ as given before), not necessarily linear, which gives the score $f_k(x)$ for a data oint having the k^{th} label. We can invoke softmax again and transform the scores into probabilities using

$$\tilde{f}_k(x) = \mathbb{P}(y = k \mid x; f) = \frac{\exp(f_k(x))}{\sum_{j\in[C]} \exp(f_k(x))} \propto \exp(f_k(x)).$$

Once we obtain the probability estimates, we can use **log loss** or **cross-entropy loss**.

Log Loss: Binary Case

We first begin with binary classification using models predicting $\tilde{f}(x)$ as the probability of label being 1 for labelled data point (x, y). The **log loss** is defined as

$$\begin{aligned}\text{LogLoss} &:= \mathbf{1}y = 1]\log(1/\tilde{f}(x)) + \mathbf{1}[y = -1]\log(1/(1 - \tilde{f}(x))) \\ &= -\mathbf{1}[y = 1]\log(\tilde{f}(x)) - \mathbf{1}[y = -1]\log(1 - \tilde{f}(x)).\end{aligned}$$

The reason: if $y = 1$, we want to maximize $\log(\tilde{f}(x))$ or equivalently minimize $\log(1/\tilde{f}(x)) = -\log(\tilde{f}(x))$.
For a linear model, $\tilde{f}(x) = \sigma(w^T x) = 1/(1 + \exp(-w^T x))$, so the log loss is

$$-\mathbf{1}[y = 1]\log\left(\frac{1}{1 + e^{-w^T x}}\right) - \mathbf{1}[y = -1]\log\left(\frac{1}{1 + e^{w^T x}}\right) = \log(1 + e^{-yw^T x}).$$

This easily generalizes to the multiclass case: if we let $\tilde{f}_k(x)$ be the predicted probability of label k, then

$$\text{LogLoss} = \sum_{k\in[C]} \mathbf{1}[y = k]\log(1/\tilde{f}_k(x)) = -\sum_{k\in[C]} \mathbf{1}[y = k]\log(\tilde{f}_k(x)).$$

Log Loss: Multiclass Case

By combining softmax and log loss, we obtain the following loss function which we use to train a multiclass classification model assigning scores $f_k(x)$ to the k^{th} class. This is an easy generalization of the above binary case:

$$\ell(f(x), y) = -\sum_{k\in[C]} \mathbf{1}[y = k]\log(\tilde{f}_k(x)) = \log\left(\frac{\sum_{k\in[C]} \exp(f_k(x))}{\exp(f_y(x))}\right) = \log\left(1 + \sum_{k\neq y} \exp(f_k(x) - f_y(x))\right).$$

(Note the similarity between this and the previous multiclass logistic loss.)

Multiclass Logistic Loss: Another View

Recall that we can make predictions using $\operatorname{argmax}_k f_k(x)$ when f_k is the inner product.

$$\ell(f(x), y) = \log\left(\sum_k \exp(f_k(x))\right) - \log\left(\exp(f_y(x))\right) = \log\left(\sum_k \exp(f_k(x))\right) - f_y(x).$$

It can be shown (a property of logsum) that

$$\max_{k\in[C]} f_k(x) \leqslant \log\left(\sum_{k\in[c]} \exp(f_k(x))\right) \leqslant \max_{k\in[C]} f_k(x) + \log C$$

so

$$\ell(f(x), y) \approx \max_{k\in[C]} f_k(x) - f_y(x).$$

In particular, if the maximum is labelled wrong, we pay extra loss.

5.5 Other Techniques for Multiclass Classification

One-versus-all

Idea: train C binary classifiers to learn "is it class k or no?" for each k.

Training: for each $k \in C$,

- Relabel examples with class k as 1 and all other classes as -1, namely, binary classification.
- Training a binary classifier h_k using this new data set.

Output: for a new example x,

- Ask each h_k, does this new point below to class k?
- Randomly pick among all k's with $h_k(x) = 1$.

In a perfect data set, each data point should be labeled positive exactly once. However, when one h_k makes error, the output will probably contain mistakes.

One-versus-one

Idea: train $\binom{C}{2}$ binary classifiers to learn "is it class k or k'" for each pair k, k'.

Training:

- Relabel examples with class k as $+1$ and k' as -1.
- Discard all other examples.
- Training a binary classifier $h_{k,k'}$ using this new data set.

Prediction: for a new example x,

- Ask each classifier to vote for either class k or k'.
- Predict the class with the most votes, break tie in some way.

This is clearly more robust than one-versus-all, but slower in prediction. Other techniques such as *tree-based methods* and *error-correcting codes* can achieve intermediate tradeoffs.

6 Neural Networks

Linear models aren't always enough. As previously discussed, sometimes we can use a nonlinear mapping and learn a linear model in the feature space $\varphi(x) : x \in \mathbb{R}^d \to z \in \mathbb{R}^m$.

But can we just learn the nonlinear mapping itself?

For model which makes predictions $f(x)$ on labeled data point (x, y), we can use the following losses.

Regression:

$$\ell(f(x), y) = (f(x) - y)^2,$$

and classification:

$$\ell(f(x), y) = \log\left(\frac{\sum_{k\in[C]} \exp(f_k(x))}{\exp(f_y(x))}\right) = \log\left(1 + \sum_{k\neq y} \exp(f_k(x) - f_y(x))\right).$$

These are the most popular loss functions for supervised learning.

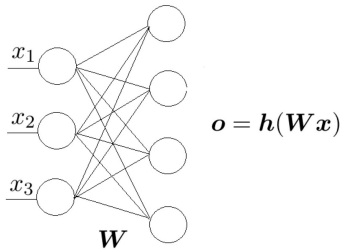
6.1 Representation: Defining Neural Networks

Linear model as a one-layer neural network: given x_1, x_2, x_3, we assign them weights w_1, w_2, w_3 and compute $o = h(w^T x)$.

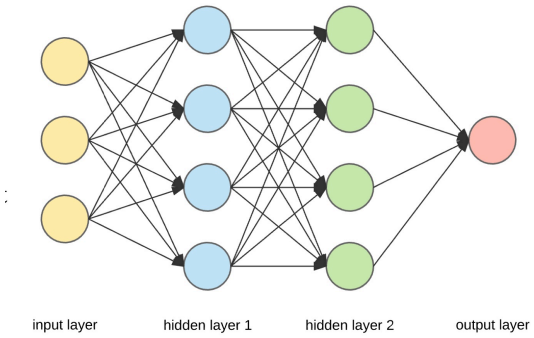
For non-linearity, we can use some nonlinear functions:

- Rectified linear unit ReLU: $h(a) = \max\{0, a\}$,
- Sigmoid: $h(a) = 1/(1 + \exp(-a))$,
- Tanh: $h(a) = (e^a - e^{-a})/(e^a + e^{-a})$, and so on.

Now we add a whole layer. Consider $W \in \mathbb{R}^{4\times 3}$, $h : \mathbb{R}^4 \to \mathbb{R}^4$ so that $h(a) = (h_1(a_1), ..., h_4(a_4))$. For convenience, we write $h(a)$ simply as $(h(a_1), ..., h(a_4)$. We can think of this as a nonlinear mapping $\varphi(x) = \varphi(x_1, x_2, x_3) = h(Wx)$.



What if we add more layers? We obtain a **neural network**.



Looking at the network:

- Each node is called a **neuron**.
- h is called the **activation function**.
  - We can use $h(a) = 1$ for one neuron in each layer to incorporate bias.
  - Output neuron can use $h(a) = a$.
- Usually, number of layers refers to the number of *hidden* layers (plus 1 or 2 for input and/or output layers).
- Deep neural nets can have many layers and millions of parameters.
- The above is a **feedforward**, **fully connected** neural net; there are many variants.

More formally —

Definition: Neural Network

An ***L*-layer neural net** can be written as

$$f(x) = h_L(W_l h_{L-1}(W_{L-1} \cdots h_1(W_1 x))).$$

Here, we define:

- $W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ as the weights between layers $\ell - 1$ and ℓ,
- $d_0 = d, d_1, ..., d_L$ are the number of neurons at each later,
- $a_\ell \in \mathbb{R}^{d_\ell}$ the input to layer ℓ,
- $o_\ell \in \mathbb{R}^{d_\ell}$ the output of layer ℓ, and
- $h_\ell : \mathbb{R}^{d_\ell} \to \mathbb{R}^{d_\ell}$ the activation functions at layer ℓ.

Now, for a given x, we have the recursive relation:

$$o_0 = x, \; a_\ell = W_\ell o_{\ell-1}, \; o_\ell = h_\ell(a_\ell), \qquad \text{for } \ell \in [L].$$

Optimizing a Neural Network

Our objective to minimize is

$$F(W_1, ..., W_L) = \frac{1}{n} \sum_{i=1}^{n} F_i(W_1, ..., W_L)$$

where

$$F_i(W_1, ..., W_L) = \begin{cases} \|f(x_i) - y_i\|_2^2 & \text{for regression} \\ \log(1 + \sum_{k \neq y_i} \exp(f_k(x_i) - f_{y_i}(x_i))) & \text{for classification.} \end{cases}$$

As usual, we apply SGD to approximating the minimum. To compute the gradient efficiently, we use **back propagation**, which we will cover later.

Similarly, we can also apply regularization: e.g. ℓ_2 regularization attempts to minimize

$$G(W_1, ..., W_K) = F(W_1, ..., W_L) + \lambda \sum_{\text{all weights}} w^2.$$

Also, the function class is very powerful!

Theorem: Universal Approximation Theorem

A feedforward neural net with a single hidden layer can approximate any continuous function.

We might need a large numbers of neurons and a deep neural net, but eventually it can approximate.

Obviously, it is important to decide the network architecture: the number of hidden layers, the number of neurons at each layer, activation functions, and so on.

6.2 Optimization: Backprop

The naive way to compute gradients: for a univariate parameter w,

$$\frac{\mathrm{d}F(w)}{\mathrm{d}w} = \lim_{\epsilon \to 0} \frac{F(w+\epsilon) - F(w-\epsilon)}{2\epsilon}.$$

Backprop in a nutshell: chain rule.

For a composite function $f(g(w))$, we have

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial w}.$$

More generally, for a composition function $f(g_1(w), ..., g_d(w))$,

$$\frac{\partial f}{\partial w} = \sum_{i=1}^{d} \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial w}.$$

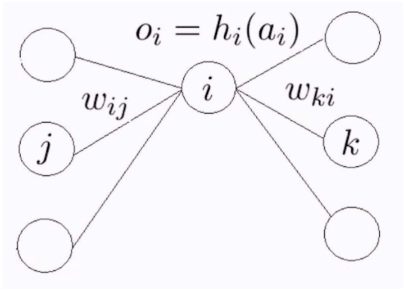
Main idea of backprop: we reuse computation by storing gradient w.r.t. the inputs to each layer (a_ℓ) because it *will* be reused many times, by the chain rule.

Moving on to neural networks, we drop the layer ℓ to make notation simpler. For this derivation, let F_m be the loss function (not F_i since i will be used for dummy variable). The derivative of F_m w.r.t. $w_{,j}$ is

$$\frac{\partial F_m}{\partial w_{i,j}} = \frac{\partial F_m}{\partial a_i}\frac{\partial a_i}{\partial w_{i,j}} = \frac{\partial F_m}{\partial a_i}\frac{\partial (w_{i,j} o_j)}{\partial w_{i,j}} = \frac{\partial F_m}{\partial a_i} o_j$$

and

$$\frac{\partial F_m}{\partial a_i} = \frac{\partial F_m}{\partial o_i}\frac{\partial o_i}{\partial a_i}\left(\sum_k \frac{\partial F_m}{\partial a_k}\frac{\partial a_k}{\partial o_i} h_i'(a_i)\right) = \left(\sum_k \frac{\partial F_m}{\partial a_k} w_{k,i}\right) h_i'(a_i).$$



Adding the subscripts back:

$$\frac{\partial F_m}{\partial (W_\ell)_{i,j}} = \frac{\partial F_m}{\partial (a_\ell)_i}(o_{\ell-1})_j \qquad \text{and} \qquad \frac{\partial F_m}{\partial (a_\ell)_i} = \left(\sum_k \frac{\partial F_m}{\partial (a_{\ell+1})_k}(w_{\ell+1})_{k,i}\right)(h_\ell)_i'((a_\ell)_i).$$

For the last layer,

$$\frac{\partial F_m}{\partial (a_L)_i} = \frac{\partial ((h_L)_i((a_L)_i - (y_n)_i)^2}{\partial (a_L)_i} = 2[(h_L)_i((a_L)_i - (y_n)_i](h_L)_i'((a_L)_i).$$

To make everything more concise, we use matrix notation:

$$\frac{\partial F_m}{\partial W_\ell} = \frac{\partial F_m}{\partial a_\ell} o_{\ell-1}^T \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$$

where

$$\frac{\partial F_m}{\partial a_\ell} = \begin{cases} \left[W_{\ell+1}^T \frac{\partial F_m}{\partial a_{\ell+1}}\right] \circ h_\ell'(a_\ell) & \text{if } \ell < L \\ 2(h_L(a_L) - y_n) \circ h_L'(a_L) & \text{if } \ell = L \end{cases} \tag{Δ}$$

where $v_1 \circ v_2$ is the **Hadamard product** / element-wise product $((v_1)_1(v_2)_1, ..., (v_1)_d(v_2)_d)$.

The Algorithm

Initialize $W_1, ..., W_L$ randomly. Repeat:

(1) Random pick $i \in [n]$.

(2) **Forward propagation**: for each $\ell = 1, ..., L$,

- Compute $a_\ell = W_\ell o_{\ell-1}$ and $o_\ell = h_\ell(a_\ell)$.

(3) **Backward propagation**: for $\ell = L, ..., 1$,

- Compute $\frac{\partial F_i}{\partial a_\ell}$ as in (Δ).
- Update weights by

$$W_\ell \leftarrow W_\ell - \eta \frac{\partial F_i}{\partial W_\ell} = W_\ell - \eta \frac{\partial F_i}{\partial a_\ell} o_{\ell-1}^T.$$

Mini-Batch

Consider $F(w) = \sum_{i=1}^{n} F_i(w)$ where $F_i(w)$ is the loss function for the i^{th} data point. Recall any $\nabla \tilde{F}(w)$ is a stochastic gradient of $F(w)$ if

$$\mathbb{E}[\nabla \tilde{F}(w)] = \nabla F(w).$$

In particular, this gives rise to the **mini-batch SGD**: sample $S \subset \{1, 2, ..., n\}$ at random and estimate the average gradient over these batch of $|S|$ samples:

$$\nabla \tilde{F}(w) := \frac{1}{|S|} \sum_{j \in S} \nabla F_j(w).$$

If batch size $S = 1$ then we get SGD. Common batch sides: $32, 64, 128$, and so on.

Adaptive Learning Rate Tuning

Another common variant on SDG is by choosing a different learning rate for each parameter (and vary this across iterations) based on the magnitude of the previous gradients for that parameter.

Momentum

Yet another variant: add a "momentum" term to encourage model to continue along previous gradient directions. "Move faster along directions that were previously good and slow down along directions where the gradient has suddenly changed, just like a ball rolling downhill." (The momentum helps dampen the oscillation caused by SGD.)

Algorithm: first initialize w_0 and $v = 0$ (velocity). For $t = 1, 2, ...$

- Estimate a stochastic gradient g_t,
- Update $v \leftarrow \alpha v + g_t$ for some discount factor $\alpha \in (0, 1)$. If $\alpha = 0$ this is just SGD.
- Update weight $w_t \leftarrow w_{t-1} - \eta v$.

Update for the first few rounds:

- $w_1 = w_0 - \eta g_1$,
- $w_2 = w_1 - \alpha\eta g_1 - \eta g_2$,
- $w_3 = w_2 - \alpha^2\eta g_1 - \alpha\eta g_2 - \eta g_3$, and so on.

6.3 Generalization: Preventing Overfitting

The best way to prevent overfitting? Get more samples.
What if we cannot get access to more samples?

- One way is data augmentation, to *exploit* prior knowledge to add more training data, e.g., flip the image, add noise, randomly translate, and so on.
- Regularization: for example, ℓ_2 regularization as stated above by adding λw^2 summed over all weights w in the network.
- Dropout: we independently delete each neuron with a fixed probability after each iteration of backprop (only for training). Not sure why this works but it is very effective and popular in practice.
- Early stopping: stop training when performance on validation set stops improving (the training error may still decrease due to overfitting but this effect does not carry over to other sets).

Beginning of Oct. 20, 2022

6.4 Convolutional Neural Networks

See the professor's lecture notes. Taking live notes on CNN is an absolute nightmare. No way I am doing that.

6.5 Sequence Prediction

Main question: given observations $x_1, x_2, ..., x_{t-1}$, what is x_t? Example: if I type a bunch of words, how to predict my next word?
In this lecture, we will mostly focus on text data (language modelling): what word comes next?

More formally, in a **language model**, let X_i be the random variable for the i^{th} word in the sentence and let x_i be the value taken by the random varaible. The goal is to compute

$$\mathbb{P}(X_{t+1} \mid X_i = s_i, 1 \leqslant i \leqslant t).$$

By property of conditionals,

$$\mathbb{P}(X_i = x_i, 1 \leqslant i \leqslant T) = \prod_{t=1}^{T} \mathbb{P}(X_t = x_t \mid X_i = x_i, 1 \leqslant i \leqslant t-1).$$

The n-gram Language Models

Question: how to learn a language model?
Answer: learn an n-gram language model.
We say an ***n*-gram** is a chunk of n consecutive words. Consider the sentence "the students opened their ..."

- Unigrams: "the", "students", "opened", "their"
- Bigrams: "the students", "students opened", "opened their"
- Trigrams: "the students opened", "students opened their"
- Four-gram: "the students opened their"

Idea: collect statistics about how frequent different n-grams are, and use these to predict the next word.

In order to do so, we need to use **Markov chains**. Stated formally, a **Markov model/chain** is a sequence of random variables $X_1, X_2, ...$ with the **Markov property**

$$\mathbb{P}(X_{t+1} \mid X_{1:t}) := \mathbb{P}(X_{t+1} \mid X_1, ..., X_t) = \mathbb{P}(X_{t+1} \mid X_t).$$

That is, the current state depends *only* on the most recent one. This is a bigram model.
We will consider the following setting:

- All X_i's take values from a *discrete* set $\{1, ..., S\}$, corresponding to different words.
- $\mathbb{P}(X_{t+1} = s' \mid X_t = s) := a_{s,s'}$, the *transition probability*, showing the probability of s' following s. For example, if $[S]$ is a dictionary,
$$a_{\text{ice, cream}} = \mathbb{P}(X_{t+1} = \text{cream} \mid X_t = \text{ice}).$$
- $\mathbb{P}(X_1 = s) := \pi_s$, the initial probability.
- $(\{\pi_s\}, \{a_{s,s'}\}) := \{\pi, A\}$, the *parameters* of the model.

We can nicely translate the Markov model into a directed graph, where an edge from the starting state to any other state s is π_s, and the edge from state s_1 to s_2 is a_{s_1,s_2}, namely, the probability of transitioning from state s_1 into s_2.
Having set up the model, now suppose we have observed n sequence of examples, all of length T:

- $x_{1,1}, ..., x_{1,T}$,
- ...
- $x_{n,1}, ..., x_{n,T}$.

Question: how do we learn the model parameters (π, A) from the observations? Most intuitively, we use MLE. The log likelihood of a sequence $x_1, ..., x_T$ is

$$\begin{aligned}\log \mathbb{P}(X_{1:T} = x_{1:T}) &= \sum_{t=1}^{T} \log(\mathbb{P}(X_t = x_t \mid X_{1:t-1} = x_{1:t-1}) \\ [\text{Markov}] &= \sum_{t=1}^{T} \log \mathbb{P}(X_t = x_t \mid X_{t-1} = x_{t-1}) \\ &= \log \pi_{x_1} + \sum_{t=2}^{T} \log a_{x_{t-1}, x_t} \\ &= \sum_s \mathbf{1}[x_1 = s] \log \pi_s + \sum_{s,s'} \left(\sum_{t=2}^{T} \mathbf{1}[x_{t-1} = s, x_t = s'] \right) \log a_{s,s'}.\end{aligned}$$

Therefore, the MLE is

$$\operatorname*{argmax}_{\pi, A} \sum_s (\#\text{ initial states with value } s \log \pi_s) + \sum_{s,s'} (\#\text{ transitions } s \to s') \log a_{s,s'}.$$

The closed form solution is

$$\pi_s = \frac{\#\text{ initial states } s}{\#\text{ initial states}} \qquad \text{and} \qquad a_{s,s'} = \frac{\#\text{ transitions } s \to s'}{\#\text{ transitions } s \to \text{any}}.$$

Looking from another perspective: by the Markov assumption, along with conditional probability,

$$\mathbb{P}(X_{t+1} = x_{t+1} \mid X_t = x_t) = \frac{\mathbb{P}(X_{t+1} = x_{t+1}, X_t = x_t)}{\mathbb{P}(X_t = x_t)}.$$

Therefore, using data,

$$\frac{\mathbb{P}(X_{t+1} = x_{t+1}, X_t = x + t)}{\mathbb{P}(X_t = x_t)} \approx \frac{\#\text{ times } (x_t, x_{t+1}) \text{ appears}}{\#\text{ times } x_t \text{ appears and is not last state}}.$$

Similarly, $\mathbb{P}(X_1 = s) \approx (\#\text{ times } s \text{ is first state})/(\#\text{ states})$.

More generally, we can have **higher-order Markov models**, where the conditional depends on the more than one recent states. Second-order Markov models, for example, are trigram, with

$$\mathbb{P}(X_{t+1} \mid X_1, ..., X_t) = \mathbb{P}(X_{t+1} \mid X_{t-1}, X_t).$$

The higher the order, the more expensive the model. Instead of counting the fraction of number of $s \to s'$ over $s \to$ any state, we now need to check the fraction of number of (previous k states to s') over (previous k states to any state).

Example: 4-gram language model. Consider the sentence "as the proctor started the clock, the students opened their ___."

In an (naive) 4-gram model,

$$\mathbb{P}(\textbf{word} \mid \text{students opened their}) = \frac{\#\text{ students opened their } \textbf{word}}{\#\text{ students opened their}},$$

and we ignored all the early contents, namely, "as the proctor started the clock, the."

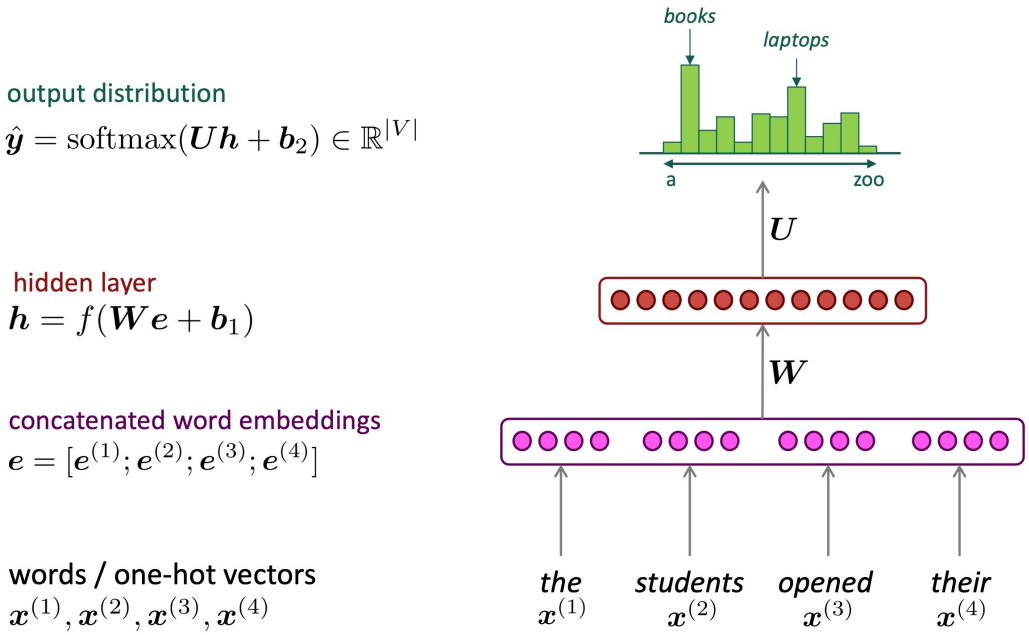
But should we disregard the context? In an exam, it is more likely that the students opened their exams instead of books, whereas in most other scenarios, "book" is more likely.

When using autocomplete, we may have seen how the algorithm outputs chunks of words that seem somehow grammatically correct but meaningless paragraphs. With insufficient grams, this will happen, and language modelling requires the model to consider at least three words for consistent performance.

6.6 Recurrent Neural Network & Language Modelling

Based on previous dissection, given inputs, a sequence of words $x^{(1)}, x^{(2)}, ..., x^{(t)}$, our goal is to output a probability distribution of the next word $\mathbb{P}(x^{(t+1)} \mid x^{(t)}, ..., x^{(1)})$, using a fixed-window method (i.e., only keeping track of the previous k words).

We can use **word embeddings / word vectors**. This is a mapping from words to vector representations of the words such that words with similar meanings have representations close to each other in the vector space.



Problems with this model:

- Fixed window size may be too small, but larger windows will make the weight matrix also larger.
- *No symmetry*: different inputs are multiplied by completely different weights in W.

7 Decision Trees

Beginning of Oct. 27, 2022

In a nutshell: decision tree is another popular model for classification. They

- are **nonlinear** in general,
- work for both classification and regression; we focus on classification,
- have good **interpretability**, and
- are effective because of their similarity to trees.

7.1 Basics

Again, tons of trees to draw, so... check link here instead.

7.2 Measures of Uncertainty

Clearly, it is impractical to do exact ERM on node parameters (runtime is exponential), and gradient descent cannot be done on this many discrete parameters. Instead, we consider a **greedy top-down approach**. But how do we determine how to split the root.

Intuitively, the **uncertainty** of a node should be a function of the distribution of classes within the node. For example, a node with 2 positive and 4 negative examples can be summarized by a distribution $\mathbb{P}$ with $1/3$ chance of being 1 and $2/3$ of being -1. One classic measure is the **Shannon entropy**

$$H(\mathbb{P}) = -\sum_{k=1}^{C} \mathbb{P}(Y = k) \log \mathbb{P}(Y = k)$$

summed over all classes C. Important properties:

- $H(\mathbb{P}) = \mathbb{E} \log(1/P(Y))$ drawn over $Y \sim \mathbb{P}$. It measures how unlikely an outcome occurs. Heuristically,H is the "average unlikeliness" when we sample outcomes from distribution $\mathbb{P}$.
- Log usually can be of base $2, e,$ or 10.
- Always non-negative.
- It is *the smallest codeword length to encode symbols drawn from* $\mathbb{P}$.
- It is maximized if $\mathbb{P}$ is uniform, i.e., *most uncertain* case (and it takes value $\log C$).
- It is minimized when $\mathbb{P}$ is constant, i.e., *most certain* case. (Here we define $0 \log 0 := \lim_{z \to 0+} z \log z = 0$).

Intuitively, for a node, we take the "weighted average" of entropy:

$$\begin{aligned} H(Y \mid A) &= \sum_a \mathbb{P}(A = a) H(Y \mid A = a) \\ &= \sum_a \mathbb{P}(A = a) \left(-\sum_{k=1}^{C} \mathbb{P}(Y \mid A = a) \log \mathbb{P}(Y \mid A = a) \right) \\ &= \sum_a \text{"fraction of examples at node } A = a\text{"} \times \text{"entropy at node } A = a\text{"}, \end{aligned}$$

and we pick the feature that leads to the smallest conditional entropy.

We split the root as such, and for each child that is not already completely split, we recursively find the "optimal" feature and use it to split the child.

Variant: the **Gini impurity** is defined by

$$G(\mathbb{P}) = \sum_{k=1}^{C} \mathbb{P}(Y = k)(1 - \mathbb{P}(Y = k)),$$

which describes how often a chosen example would be incorrectly classified if we predict according to another randomly picked example.

Regularization: if the dataset has no contradiction (same x always correspond to same y), the training error is always zero, and the model can overfit. Ways to prevent overfitting:

- restrict the depth or the number of nodes,
- do not split a node if the examples at the node is not enough, or
- other approaches, using make use of a validation set to turne the parameters.

7.3 Ensemble Methods - Bagging

Main idea: combine multiple classifiers to form a learner with better performance than any of them individually.
Why? Individual decision trees can be fast and robust to data variations, but they are unstable, and small variations in data can lead to different trees (since differences can propagate). They are **high variance models**, which can overfit.
The process is known as **bagging** (Bootstrap Aggregating). Bootstrap sampling – getting different subsets of the data, and aggregating – averaging.
Procedure:

- Get multiple random splits / subsets of the data.
- Train a given procedure (e.g. decision tree) on each subset.
- Average the predictions of all trees to make predictions on test data.

More formally, we collect T subsets of size m by sampling with replacement from the training data (**bootstrap**). Let $f_t(x)$ be a classifier obtained on the subset $t \in \{1, ..., T\}$. Then the aggregated classifier f is defined by

$$f(x) := \begin{cases} T^{-1} \sum_{t=1}^{T} f_t(x) & \text{for regression} \\ \operatorname{sgn}(T^{-1} \sum_{t=1}^{T} f_t(x)) = \text{majority vote} & \text{for classification} \end{cases}$$

(For multiclass classification, majority vote generalizes to taking the most popular vote.)
Majority vote works very well — suppose $X_1, ..., X_{100}$ are i.i.d. with $\mathbb{P}(X_i = 1) = 0.6$. Then,

$$\mathbb{P}(\sum_{i=1}^{100} 1_{X_i} > 50) = \mathbb{P}(\text{majority votes } 1) \approx 0.97.$$

That is, if each individual has a 0.6 probability of being correct, bagging over 100 of them, this number drastically increases to 0.97.
To sum up:

- Bagging reduces overfitting / variance.
- Bagging works with any type of classifier (specifically, trees).
- Easy to parallelize (can train multiple trees in parallel).
- However, we indeed lose on interpretability to single decision tree, as do all ensemble methods.

7.4 Ensemble methods - Random Forest

Issue with bagging: bagged trees may still be too correlated, as they are still sharing many similar data.
How to decorrelate the trees further? When growing a tree on bootstrapped dataset, before each split, select $k \leqslant d$ of the d input random variables at random as candidates for splitting. If $k = d$ this is bagging, otherwise we get **random forests**.
Issues:

- When we have a large number of features but a small number of *relevant* features, $\mathbb{P}$ of selecting a relevant feature in this way is small.
- Lacks expressive power compared to other ensemble methods.

7.5 Boosting

Recall we learn our classifier by $f(x) = \operatorname{sgn}(T^{-1}\sum_{i=1}^{T} f_i(x))$. Instead of training $\{f_t\}_{t=1}^{T}$ in parallel, what if we sequentially learn which models to use for the function class $\mathcal{F}$ so they are together as accurate as possible?

Boosting is a **meta-algorithm**, meaning it takes a **base algorithm** as input and **boosts** its accuracy.
Main idea: combine "weak rules of thumb" (e.g. 51% accuracy) to form a *highly accurate predictor*.

Example: Boosting: example. Suppose we are given a training set labelling messages as spam or not spam.

- Let the base algorithm be given, for example a bad one like (containing "money") $\Rightarrow$ spam.
- We *reweigh* the example so that *difficult* ones get more attention, e.g., on the spams that don't contain the word "money."
- Obtain another classifier by applying the same base algorithm: for example (empty "to address") $\Rightarrow$ spam.
- Repeat. And finally take (weighted) majority vote.

More formally: a **base algorithm** $\mathcal{A}$ (also called a **weak learning algorithm**) takes a training set S weighted by D as input and outputs a classifier $f = \mathcal{A}(S, D)$.
Booting: idea. The boosted predictor is of form $f_b(x) = \operatorname{sgn}(h(x))$ where

$$h(x) = \sum_{t=1}^{T} \beta_t f_t(x) \qquad \text{for } \beta_t \geqslant 0 \text{ and } f_t \in \mathcal{F}.$$

The goal is to minimize $\ell(h(x), y)$ w.r.t. some loss function ℓ.
We find β_t one by one for $t \in [1, T]$ using a greedy approach. In particular, let $h_t = \sum_{i=1}^{t} \beta_i f_i(x)$. If we have already found $h_{t-1}(x)$, we need to find $\beta_t, f_t(x)$ such that they minimize $\ell(h_t(x), y)$. Different ℓ gives different boosting algorithms:

- If $\ell(h(x), y) = (h(x) - y)^2$, we have **least squares boosting**.
- If $\ell(h(x), y) = \exp(-h(x)y)$, we have **AdaBoost**.

7.6 AdaBoost

Recall $h_t(x) = \sum_{j=1}^{t} \beta_j f_j(x)$. Suppose we have found h_{t-1}. Greedily we want to find $\beta_t, f_t(x)$ to minimize

$$\sum_{i=1}^{\infty} \exp(-y_i h_t(x_i)) = \sum_{i=1}^{n} \exp(-y_i h_{t-1}(x_i)) \exp(-y_i \beta_t f_t(x_i)) = C_t \cdot \sum_{i=1}^{n} D_t(i) \exp(-y_i \beta_t f_t(x_i))$$

where the last step is by defining $D_t(i) := \exp(-y_i h_{t-1}(x_i))/C_t$, normalized so that $\sum_{i=1}^{n} D_t(i) = 1$. Then, we want to maximize (keeping in mind we are doing binary classification here, so $y_i, f_t(x_i) \in \{\pm 1\}$)

$$\begin{aligned}\sum_{i=1}^{n} D_i(i) \exp(-y_i \beta_t f_t(x_i)) &= \sum_{i: y_i \neq f_t(x_i)} D_t(i) e^{\beta_t} + \sum_{y: y_i = f_t(x_i)} D_t(i) e^{-\beta_t} \\ &= \epsilon_t e^{\beta_t} + (1 - \epsilon_t) e^{-\beta_t} = \epsilon_t (e^{\beta_t} - e^{-\beta_t}) + e^{-\beta_t},\end{aligned}$$

where $\epsilon_t = \sum_{y_i \neq f_t(x_i)} D_t(i)$ is the **weighted error** of f_t. This is a two-step process: first set f_t such that ϵ_t is minimized; then, given this ϵ_t, minimize the rest.
With f_t and ϵ_t fixed, the above quantity is minimized by

$$\beta_t = \frac{1}{2}\log((1-\epsilon_t)/\epsilon_t).$$

Then we need to update $D_{t+1}(i)$ by

$$D_{t+1}(i) = \exp(-y_i h_t(x_i))/C_{t+1} = \left(D_t(i)\frac{C_t}{C_{t+1}}\right)\exp(-y_i\beta_t f_t(x_i)).$$

That is,

$$D_{t+1}(i) \propto D_t(i)\exp(-\beta_t y_i f_t(x_i)) = \begin{cases} D_t(i)e^{-\beta_t} \text{ if } f_t(x_i) = y_i \\ D_t(i)e^{\beta_t} \text{ else.} \end{cases}$$

Intuitively, for those already correctly classified, we lower the weight since the next classifier need not to worry too much about; conversely, the misclassified need more attention.

Theorem: AdaBoost, full algorithm

Given a training set S and a base algorithm $\mathcal{A}$, initialize D_1 to uniform.
For $t = 1, ..., T$:

- Obtain a weak classifier $f_t(x) = \mathcal{A}(S, D_t)$.
- Calculate the weight β_t of $f_t(x)$ as

  $$\beta_t = 0.5\log((1-\epsilon_t)/\epsilon_t)$$

  where $\epsilon_t = \sum_{i:y_i \neq f_t(x_i)} D_t(i)$, the weighted error of $f_t(x)$.
- Update distributions by

  $$D_{t+1}(i) \propto \begin{cases} D_t(i)e^{-\beta_t} \text{ if } f_t(x_i) = y_i \\ D_t(i)e^{\beta_t} \text{ else.} \end{cases}$$

Output the final classifier $f_b = \text{sgn}(\sum_{t=1T}\beta_t f_t(x).)$

Beginning of Nov. 3, 2022

7.7 Ensemble Methods - Gradient Boosting

Recall $h_t(x) = \sum_{j=1}^{t} b_j f_j(x)$. In Adabost we chose the lose function as $\exp(-h(x)y)$. We now have a more general method for any loss function $\ell(h(x), y)$:

- For all training data points (x_i, y_i), find the gradient

  $$r_i = \left[\frac{\partial \ell(h(x_i), y_i)}{\partial h(x_i)}\right]_{h(x_i)=h_{t-1}(x_i)}.$$

  "How should predictions change 'locally' to reduce loss?"

- Use this weak learner to find f_t which fits (x_i, r_i) as well as possible:

$$f_t = \operatorname*{argmin}_{f \in \mathcal{F}} \sum_{i=1}^{n} (r_i - f(x_i))^2.$$

  r_i: what should be added to bring the loss down.

- Update $h_t(x) = h_{t-1}(x) + \eta f_t(x)$ for some prescribed η.

As usual, we can add regularization terms to prevent overfitting. In general, gradient boosting is extremely successful.

8 Unsupervised Learning: PCA

8.1 Introduction

Recall we have various kinds of learnings:

- supervised: aim to predict outputs of future datapoints,
- reinforcement: aim to make sequential decisions, and
- **unsupervised**: aim to discover hidden patterns and explore data.

Goal of dimensional reduction: reduce the dimensionality of a dataset so that

- it is easier to visualize and discover patterns,
- it takes less time and space to process,
- noise is reduced, and so on.

High-level Goal

Suppose we have a dataset of n d-dimensional vectors $x_1, ..., x_n$. The high level goal of PCA is to find a set of k **principal components** (PCs) $v_1, ..., v_k \in \mathbb{R}^d$ such that for each x_i,

$$x_i \approx \sum_{j=1}^{k} \alpha_{i,j} v_j,$$

for some coefficients $\alpha_{i,j} \in \mathbb{R}$.

Processing the Data

- Before applying PCA, we preprocess the data by centering them:

$$x_i \leftarrow x_i - \frac{1}{n} \sum_{i=1}^{n} x_i$$

  so that $\sum x_i = 0$.

- In many applications we also scale w.r.t. each component/coordinate: for each $j \in [d]$, divide the j^{th} coordinate of each data point by $\left(\sum_{i=1}^{n} x_{i,j}^2\right)^{1/2}$.

Objective Function of PCA

Key difference from supervised learning: no labels are given, so there is no ground trug to measure the quality of the answer (loss functions previously defined no longer work!).
However, we can still write an optimization problem based on our high level goal. For example, the special case of $k = 1$ reduces to an optimization problem for finding the first PC v_1.

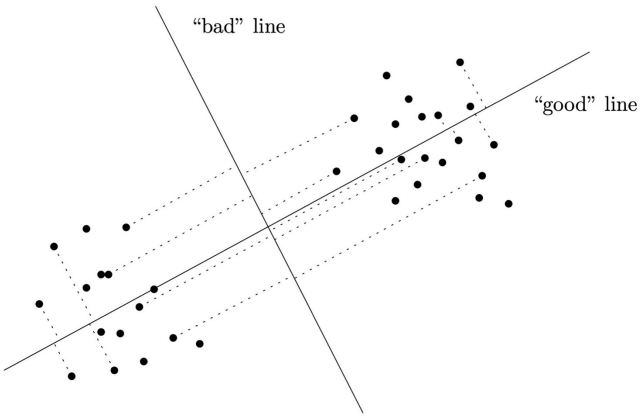


Figure 5: For the good line, the projection of the points onto the line keeps the two clusters separated, while the projection onto the bad line merges the two clusters.

Therefore, the original objective

$$\operatorname*{argmin}_{\|v\|=1} \frac{1}{m} \sum_{i=1}^{m} (\text{distance between } x_i \text{ and the line spanned by } v)^2$$

is equivalent to finding

$$v_1 = \operatorname*{argmax}_{\|v\|=1} \sum_{i=1}^{n} \langle x_i, v \rangle^2 ,$$

since distance^2 + $\langle x_i, v \rangle^2 = \|x_i\|^2$, which is fixed. That is, the data points have as much projection as possible onto v_1.
For general k, the objective is to find

$$S = \operatorname*{argmin}_{k\text{-dim subspace } S} \sum_{i=1}^{n} (\text{distance between } x_i \text{ and } S)^2 = \operatorname{argmax} \sum_{i=1}^{n} (\text{length of } x_i's \text{ projection onto } S)^2.$$

Linear Algebra Recap

If $v_1, v_2, ..., v_k$ are orthonormal, then the projection of x onto the span of v_i is

$$\sum_{j=1}^{k} \langle x_i, x_j \rangle x_i,$$

whose norm is $\sum_{j=1}^{k} \langle x_i, x_j \rangle^2$. Therefore, we obtain the formal statement of PCA:

Definition: PCA Objective

Given $x_1, ..., x_n \in \mathbb{R}^d$ and a parameter $k \geqslant 1$, find a set of orthonormal vectors $v_1, ..., v_k \in \mathbb{R}^d$ to maximize

$$\sum_{i=1}^{n} \sum_{j=1}^{k} \langle x_i, v_j \rangle^2 .$$

Using PCA for Data Compression & Visualization

Input: n data points $x_1, ..., x_n \in \mathbb{R}^d$, and k, number of components we want.

Step 1. Perform PCA to get the top k principal components $v_1, ..., v_k \in \mathbb{R}^d$.

Step 2. For each dataset x_i define its v_j-coordinate to be $\langle x_i, v_j \rangle$.

Step 3. We now obtain a compresssed data set where data points are k-, not kd-, dimensional. Plot them!

8.2 PCA Optimization

Consider $k = 1$, in which case we have

$$v_1 = \operatorname*{argmax}_{\|v\|=1} \sum_{i=1}^{n} \langle x_i, v \rangle^2 .$$

Writing all vectors x_i in one big matrix X, and defining $A := X^T X$ (a $d \times d$ matrix), we obtain an equivalent objective

$$v_1 = \operatorname*{argmax}_{\|v\|=1} v^T A v.$$

Note that

$$A_{11} = \sum_{i=1}^{n} x_{i,1}^2 = \text{variance of first coordinate}$$

and

$$A_{12} = \sum_{i=1}^{n} x_{i,1} x_{i,2} = \text{covariance between the first and second coordinates.}$$

The same token holds for general i, j.

Now we move on to actual linear algebra.

If A is diagonal, then $v^T A v = \|Dv\|^2$ where $D_{i,i} = A_{i,i}^{1/2}$. Therefore, it is clear that in order to maximize the norm, we need to "stress" the component j on which $A_{j,j}$ is the largest among all diagonal entries, i.e., $\operatorname{argmax}_v v^T A v = e_j = (0, ..., 0, 1, 0, ..., 0)$, where 1 appears on the j^{th} component.

More generally, if we diagonalize A as QDQ^T where Q is orthogonal and D diagonal with diagonal entries $\lambda_1 \geqslant \lambda_2 \geqslant ... \geqslant \lambda_d \geqslant 0$, then $v^T A v = v^T Q D Q^T v = v^T Q D^{1/2} D^{1/2} Q^T v = \|D^{1/2} Q^T v\|$. Similar to above, we want $Q^T v = e_1$, since λ_1 is the largest eigenvalue of A, which implies we need $v = Q^{-T} e_1 = Q e_1$, and this vector is stretched by a factor of $\lambda_1^{1/2}$.

Upshot. Compute $X^T X$ and diagonalize as QDQ^T (which is doable since $X^T X$ is symmetric). Then compute Qe_1.

How many PCs to use?

For visualization, we choose k small $(1, 2, 3)$ and plot them out.

In other applications such as compression, it is a good idea to plot the eigenvalues first. If the data is close to being low ranks, the eigenvalues may decay and become small.
We can also decide based on a variance threshold: if we want to capture 90% of the data, pick k such that

$$\frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{d} \lambda_j} = \frac{\sum_{j=1}^{k} \lambda_j}{\text{tr}(X^T X)} \geqslant 0.9.$$

When does PCA Fail?

(1) When data is not properly scaled / normalized.

(2) Not many orthogonal components in the data which are interpretable.

(3) Nonlinear structure of data (e.g. polar).