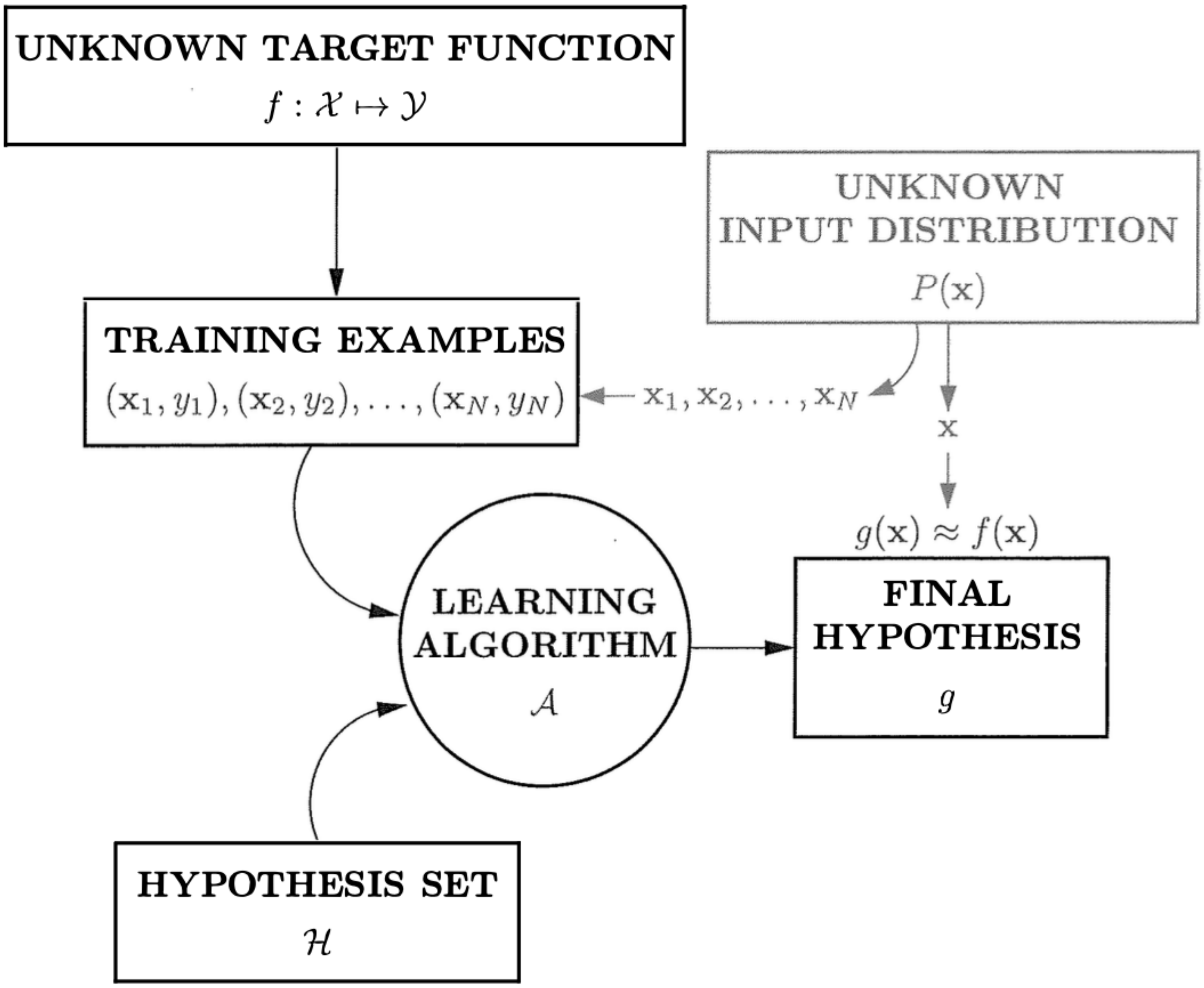
HTML 2023

Machine Learning: using data to compute hypothesis g that approximates target f


UNKNOWN TARGET FUNCTION
$f : \mathcal{X} \mapsto \mathcal{Y}$
UNKNOWN INPUT DISTRIBUTION
$P(\mathbf{x})$
TRAINING EXAMPLES
$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)$
$\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$
$\mathbf{x}$
$g(\mathbf{x}) \approx f(\mathbf{x})$
LEARNING ALGORITHM
$\mathcal{A}$
FINAL HYPOTHESIS
g
HYPOTHESIS SET
$\mathcal{H}$

Perception Learning Algorithm

Given a data set $\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_N, y_N)\}$, and a hypothesis set $\mathcal{H}$, we choose from $\mathcal{H}$ a function $g \in \mathcal{H} = \{h_k\}$ such that $g \approx f$.

For $\mathbf{x} = (x_1, x_2, \cdots, x_d)$, we compute a weighted score $\sum_{i=1}^{d} w_i x_i$ and determine y according to

$$h(x) = \text{sign}\,\big(\sum_{i=1}^{d} w_i x_i - \text{threshold}\big)$$

define $(w_0, x_0) = (-\text{threshold}, 1)$, thus

$$h(x) = \text{sign}\,\big(\sum_{i=0}^{d} w_i x_i\big) = \text{sign}(\mathbf{w}^T \mathbf{x})$$

The function $\mathbf{w}^T\mathbf{x} = 0$ is a hyperplane in R^d space, and we optimize $\mathbf{w}$ to fit all data. If we find a mistake in the data set, namely $h(x_n) \neq y_n$, we correct this by adjusting $\mathbf{w}$:

$$w_{t+1} \leftarrow w_t + x_n y_n$$

and we repeat until all data is correctly classified. This is the perception learning algorithm. If $\mathcal{D}$ is linear separable, then there exists w_f such that

$$y_{n(t)}\mathbf{w}_f^T\mathbf{x}_{n(t)} \geq \min y_n \mathbf{w}_f^T\mathbf{x}_n > 0$$

so

$$\mathbf{w}_f^T\mathbf{w}_{t+1} = \mathbf{w}_f^T(\mathbf{w}_t + x_n y_n) \geq \mathbf{w}_f^T\mathbf{w}_t$$

w_t approaches w_f. The length of w_{t+1} is given by

$$||\mathbf{w}_{t+1}||^2 = ||\mathbf{w}_t||^2 + 2y_n\mathbf{w}_t^T\mathbf{x}_n + ||y_n(t)\mathbf{x}_n(t)||^2 \leq ||\mathbf{w}_t||^2 + \max ||\mathbf{x}_n||^2$$

where $2y_n\mathbf{w}_t^T\mathbf{x}_n < 0$. Therefore

$$||\mathbf{w}_t||^2 \leq ||\mathbf{w}_0||^2 + tR^2$$

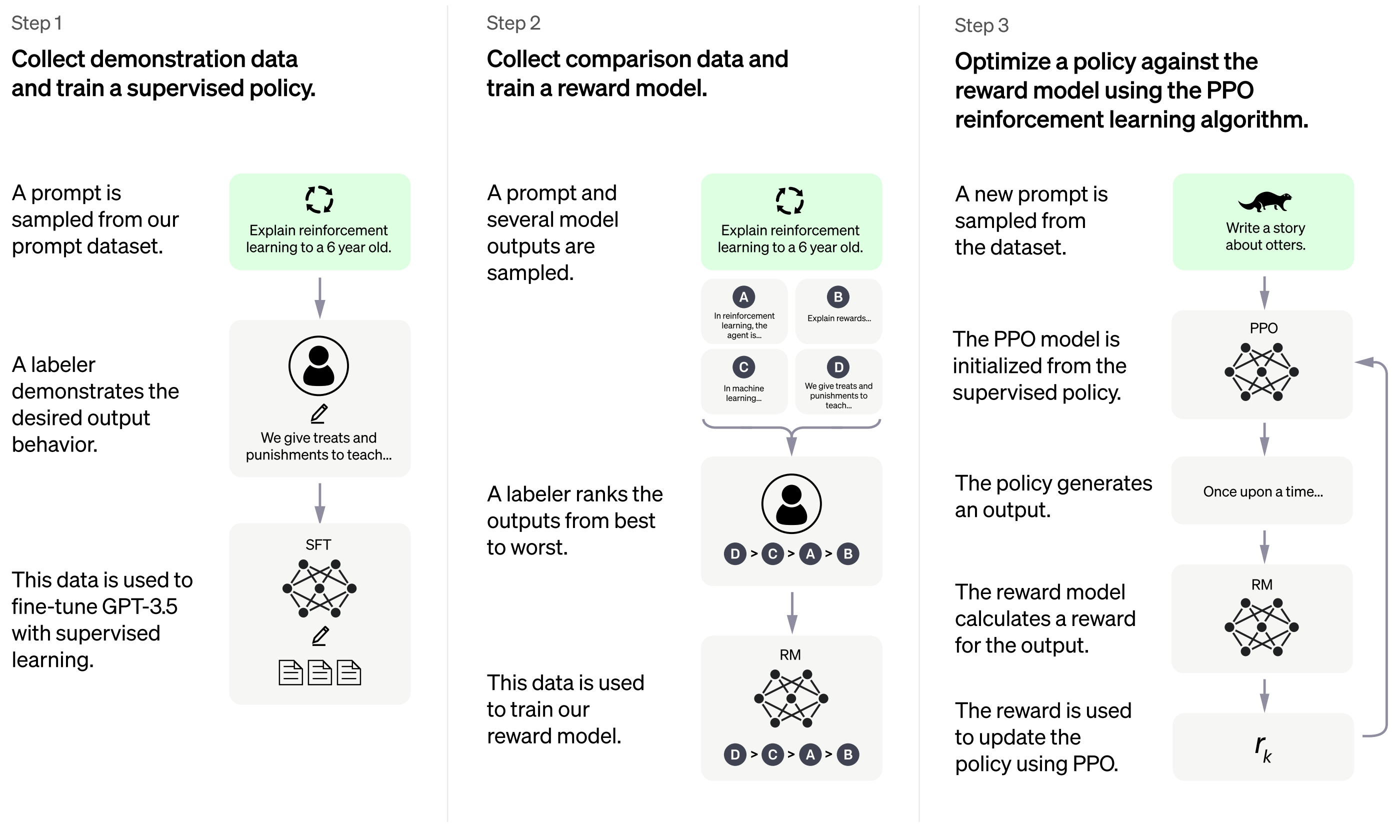
$$\mathbf{w}_f^T\mathbf{w}_t \geq \mathbf{w}_f^T\mathbf{w}_0 + t\rho$$

where $R = \max ||\mathbf{x}_n||$ and $\rho = \min y_n \mathbf{w}_f^T\mathbf{x}_n$. Starting from $\mathbf{w}_0 = \mathbf{0}$:

$$1 \leq \frac{\mathbf{w}_f^T\mathbf{w}_t}{||\mathbf{w}_f||||\mathbf{w}_t||} \leq \frac{T\rho}{\sqrt{T}R} \quad \Rightarrow \quad T \leq (\frac{R}{\rho})^2$$

This puts an upper bound on the number of iterations. Here the magnitude of $||\mathbf{w}_f||$ is chosen to be 1, but the choice can be arbitrary.

ChatGPT


Step 1
Collect demonstration data and train a supervised policy.
A prompt is sampled from our prompt dataset.
Explain reinforcement learning to a 6 year old.
A labeler demonstrates the desired output behavior.
We give treats and punishments to teach...
This data is used to fine-tune GPT-3.5 with supervised learning.
SFT
Step 2
Collect comparison data and train a reward model.
A prompt and several model outputs are sampled.
Explain reinforcement learning to a 6 year old.
A In reinforcement learning, the agent is...
B Explain rewards...
C In machine learning...
D We give treats and punishments to teach...
A labeler ranks the outputs from best to worst.
D > C > A > B
This data is used to train our reward model.
RM
D > C > A > B
Step 3
Optimize a policy against the reward model using the PPO reinforcement learning algorithm.
A new prompt is sampled from the dataset.
Write a story about otters.
The PPO model is initialized from the supervised policy.
PPO
The policy generates an output.
Once upon a time...
The reward model calculates a reward for the output.
RM
The reward is used to update the policy using PPO.
r_k

Classification of Learning

Protocols

- **Batch** : Learning from all data at once
- **Online** : hypothesis improves through receiving data instances sequentially
- **Active** : improve hypothesis with fewer labels by asking questions strategically

Input Space

- **Concrete** : each dimension of $\mathcal{X}$ represents ‘sophisticated physical meaning’ .
- **Raw** : simple physical meanings
- **Abstract** : no physical meaning

Deep Learning

capture the most important features of the data, and conversion from raw to concrete data.

Hoeffding's Inequality

Given a sample of size N selected out of a pool with probability μ , the distribution of the in-sample probability ν is related by

$$P[|\nu-\mu|>\epsilon] \leq 2\exp(-2\epsilon^2 N)$$

In the case of learning, the out-of-sample error is determined by whether the hypthoesis set is equal to the target function $h(x)\neq f(x)$, which is infered by the in sample error $h(x)\neq y_n$. define

$$E_{\text{in}}(h)=\frac{1}{N}\sum_{n=1}^{N}[h(x_n)\neq y_n]$$

$$E_{\text{out}}(h)=\mathcal{E}_{\mathbf{x}\sim P}[h(x_n)\neq y_n]$$

then the Hoeffding's Inequality becomes

$$P[|E_{\text{in}}(h)-E_{\text{out}}(h)|>\epsilon] \leq 2\exp(-2\epsilon^2 N)$$

for any fixed h, when the data is large enough

$$E_{\text{in}}(h)\approx E_{\text{out}}(h)$$

Given a data set $\mathcal{D}$ and hypothesis h_m, The chance that the in-sample error deviates from the expected value is

$$P(\mathcal{D},h_m)\leq 2\exp(-2\epsilon^2 N)$$

The chance that "bad" data exists for all M hypothesis is

$$P\leq P(\mathcal{D},h_1)+P(\mathcal{D},h_2)+\cdots+P(\mathcal{D},h_M)\leq 2M\exp(-2\epsilon^2 N)$$

Growth Function

In the case where $M\to\infty$, the constraint equation is no longer sufficient, and we must look to find a quantity the captures how different the hypotheses in $\mathcal{H}$ are, and hence how much overlap the different events have. Assume each hypothesis set $h\in\mathcal{H}$ maps $\mathcal{X}$ to $\{-1,+1\}$. Given a finite number of sample points $\mathbf{x}_1,\cdots,\mathbf{x}_N\in\mathcal{X}$. We define a dichotomy as

$$\mathcal{H}(\mathbf{x}_1,\cdots,\mathbf{x}_N)=\{\,(h_1,\cdots,h_N)\,|\,h\in\mathcal{H}\}$$

And the growth function defined for $\mathcal{H}$ by

$$m_{\mathcal{H}}(N)=\max|\,\mathcal{H}(\mathbf{x}_1,\cdots,\mathbf{x}_N)\,|$$

In words, $m_{\mathcal{H}}(N)$ is the maximum number of dichotomies that can be generated by $\mathcal{H}$ on any N points. Like M, $m_{\mathcal{H}}(N)$ is a measure of the number of hypotheses in $\mathcal{H}$, except that a hypothesis is now considered on N points instead of the entire $\mathcal{X}$. The value of $m_{\mathcal{H}}(N)$ is bounded by

$$m_{\mathcal{H}}(N) \le 2^N$$

If $\mathcal{H}$ is capable of generating all possible dichotomies on $\mathbf{x}_1, \cdots, \mathbf{x}_N$ then we say that $\mathcal{H}$ can $shatter$ $\mathbf{x}_1, \cdots, \mathbf{x}_N$. This signifies that $\mathcal{H}$ is as diverse as can be on this sample.
If no data set of size k can be shattered by $\mathcal{H}$, then k is said to be a breakpoint for $\mathcal{H}$. In the case of two-dimensional perceptrons:

k	$m_{\mathcal{H}}$	2^k
1	2	2
2	4	4
3	8	8
4	14	16

Because $m_{\mathcal{H}} < 2^k$, $k = 4$ is considered a break point for two-dimensional perceptrons. Similarly, $k+1, k+2, \cdots$ are also break points.

The Vapnik-Chervonenkis Bound

The Hoeffding inequality now needs to be modified when replacing M with $m_{\mathcal{H}}$. The correct form is

$$P_{\mathcal{D}}[\ \exists h \in \mathcal{H} \text{ s.t. } |E_{\text{in}}(h) - E_{\text{out}}(h)| > \epsilon\ ] \le 4 m_{\mathcal{H}}(2N) \exp(-\epsilon^2 N/8) \le 4(2N)^{d_{VC}} \exp(-\epsilon^2 N/8)$$

The VC bound keeps track of overlaps, so it estimates the total area of bad generalization to be relatively small.

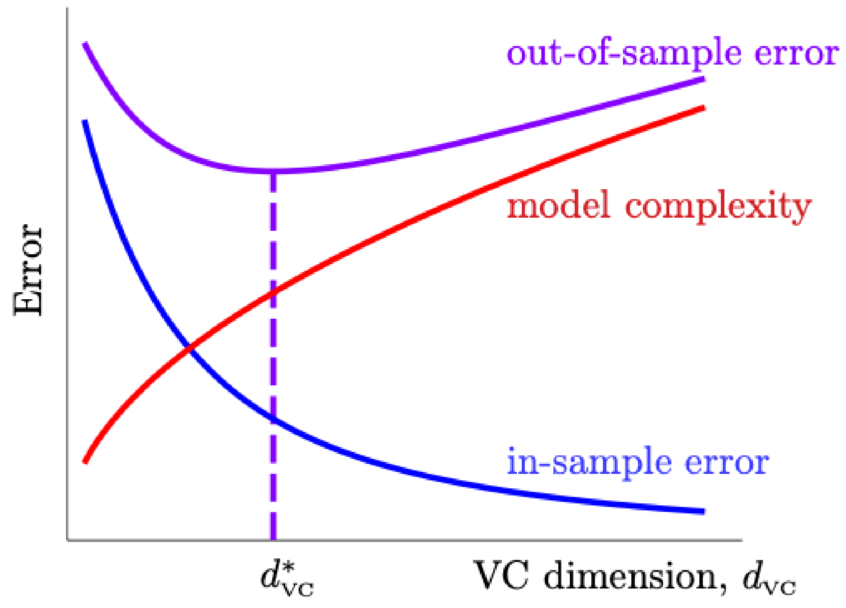
The VC Dimension

The VC dimension d_{VC} is defined as the largest value of N that satisfies $m_{\mathcal{H}}(N) = 2^N$, with $k = d_{VC} + 1$ being the break point. For perceptrons of d-dimensions $d_{VC} = d + 1$

In conclusion, the VC Bound tells us that, with a high probaility,

$$E_{\text{out}} \le E_{\text{in}} + \sqrt{\frac{8}{N} \ln\left(\frac{4(2N)^{d_{vc}}}{\delta}\right)}$$

where $\Omega(N, \mathcal{H}, \delta) = \sqrt{\frac{8}{N} \ln\left(\frac{4(2N)^{d_{vc}}}{\delta}\right)}$ is the model complexity.



Linear Regression

In the case of linear regression, the hypothesis is generated by a weighted sum,

$$y \approx h(x) = \mathbf{w}^T\mathbf{x}$$

The in-sample error is defined as

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(h(x_n) - y_n)^2 = \frac{1}{N}||\mathbf{X}\mathbf{w} - \mathbf{y}||^2$$

Here $\mathbf{X}$ is an $N \times (d+1)$ matrix. To minimize $E_{\text{in}}(\mathbf{w})$, take the gradient of $E_{\text{in}}(\mathbf{w})$ to be 0.

$$\nabla E_{\text{in}}(\mathbf{w}) = \frac{2}{N}(\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y})$$

For invertible $\mathbf{X}^T\mathbf{X}$:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

The in and out of sample error is related to the noise level σ^2:

$$\langle E_{\text{out}}\rangle = \sigma^2\big(1 + \frac{d+1}{N}\big)$$

$$\langle E_{\text{in}}\rangle = \sigma^2\big(1 - \frac{d+1}{N}\big)$$

Logistic Regression

Logistic regression is a case of soft binary classification. The target function is represented by a probability distribution,

$$f(\mathbf{x}) = P(+1|\,\mathbf{x}) \in [0,1]$$

and data is generated by taking a sample of the distribution. Therefore, the goal would be to maximize the probability of the given sample. Contrary to hard binary classification, where

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x})$$

The hypothesis of the logistic regression is in the form

$$h(\mathbf{x}) = \theta(\mathbf{w}^T\mathbf{x})$$

where $\theta(s)$ satisfies the condition

$$\theta(-\infty) = 0 \quad \theta(0) = \frac{1}{2} \quad \theta(\infty) = 1$$

We choose

$$\theta(s) = \frac{1}{1+e^{-s}}$$

thus we use

$$h(\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$$

to approximate target function.

Cross-Entropy Error

The logistic function has the property

$$h(x) + h(-x) = 1$$

For the target function, the given sample has a probability of

$$P(\mathbf{x}_1)f(\mathbf{x}_1) \times P(\mathbf{x}_2)(1 - f(\mathbf{x}_2)) \times \cdots \times P(\mathbf{x}_N)(1 - f(\mathbf{x}_N))$$

We would like to approximate the above with $h(x)$:

$$P(\mathbf{x}_1)h(\mathbf{x}_1) \times P(\mathbf{x}_2)h(-\mathbf{x}_2) \times \cdots \times P(\mathbf{x}_N)h(-\mathbf{x}_N) \propto \prod_{n=1}^{N} h(y_n\mathbf{x}_n)$$

The method of maximum likelihood selects the hypothesis h which maximizes this probability

$$\max_{\mathbf{w}} \ln \prod_{n=1}^{N} \theta(y_n\mathbf{w}^T\mathbf{x}_n) = \min_{\mathbf{w}} \sum_{i=1}^{N} -\ln\theta(y_n\mathbf{w}^T\mathbf{x}_n)$$

Substituting

$$\min_{\mathbf{w}} \sum_{i=1}^{N} -\ln\theta(y_n\mathbf{w}^T\mathbf{x}_n) = \min_{\mathbf{w}} \sum_{i=1}^{N} \ln\left(1+\exp(-y_n\mathbf{w}^T\mathbf{x}_n)\right)$$

The fact that we are minimizing this quantity allows us to treat it as an 'error measure'. Define

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \ln\left(1+e^{-y_n\mathbf{w}^T\mathbf{x}_n}\right)$$

The error is small when the quantity $y_n\mathbf{w}^T\mathbf{x}_n$ is positive.

Gradient Descent

We want to derive the $\mathbf{w}$ which minimizes the error, therefore

$$\nabla E_{\text{in}}(\mathbf{w}) = 0$$

For each of $\mathbf{w}$'s component we take its partial derivative

$$\frac{\partial E_{\text{in}}(\mathbf{w})}{\partial w_i} = -\frac{1}{N}\sum_{i=1}^{N} \frac{1}{1+e^{y_n\mathbf{w}^T\mathbf{x}_n}} y_n x_{ni} = 0$$

A explicit solution can not be obtained, instead, we approach the problem using iterative optimization.

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta\mathbf{v}$$

where the method is defined by the stopping condition and the choice of $\eta, \mathbf{v}$. In the case of gradient descent, we generally choose $\eta = 0.1$ and have the iteration stop at large t.

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta\frac{\nabla E_{\text{in}}(\mathbf{w}_t)}{||\nabla E_{\text{in}}(\mathbf{w}_t)||}$$

To select η of suitable magnitude, set $\eta_t = \eta\,||\nabla E_{\text{in}}(\mathbf{w}_t)||$, leading to the fixed learning rate gradient descent algorithm.

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t\nabla E_{\text{in}}(\mathbf{w}_t)$$

The version of gradient descent we have described so far is known as batch gradient descent the gradient is computed for the error on the whole data set before a weight update is done. A sequential version of gradient descent known as stochastic gradient descent. We pick one data point at a time randomly.

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta\frac{y_n\mathbf{x}_n}{1+e^{y_n\mathbf{w}_t^T\mathbf{x}_n}}$$

Over a series of iteration, the expected value is

$$\eta\frac{1}{N}\sum_{i=1}^{N}\frac{y_n\mathbf{x}_n}{1+e^{y_n\mathbf{w}^T\mathbf{x}_n}}$$

This is exactly the same as the deterministic weight change from the batch gradient descent weight update, but much more efficient.

Non-linear Models

Linear perceptrons take the form

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x})$$

For non-linear cases, we make the transformation

$$\mathbf{x} \in \mathcal{X} \quad \rightarrow \quad \mathbf{z} = \Phi(\mathbf{x}) \in \mathcal{Z}$$

maybe the model is a circular perceptron

$$\mathbf{z} = \Phi(\mathbf{x}) = (1, x_1^2, x_2^2)$$

In the general quadratic case:

$$\Phi_2(\mathbf{x}) = (1, x_1, x_2, x_1^2, x_1x_2, x_2^2)$$

and

$$h(\mathbf{x}) = \tilde{h}(\mathbf{z}) = \text{sign}(\tilde{\mathbf{w}}^T\Phi(\mathbf{x}))$$

Polynomial transform for large dimensions can be difficult to store and compute,

$$\Phi_Q(\mathbf{x}) = (1, x_1, \cdots, x_d, x_1^2, \cdots, x_1^Q, \cdots, x_d^Q)$$

There are

$$\tilde{d} = \binom{Q+d}{Q} = O(Q^d)$$

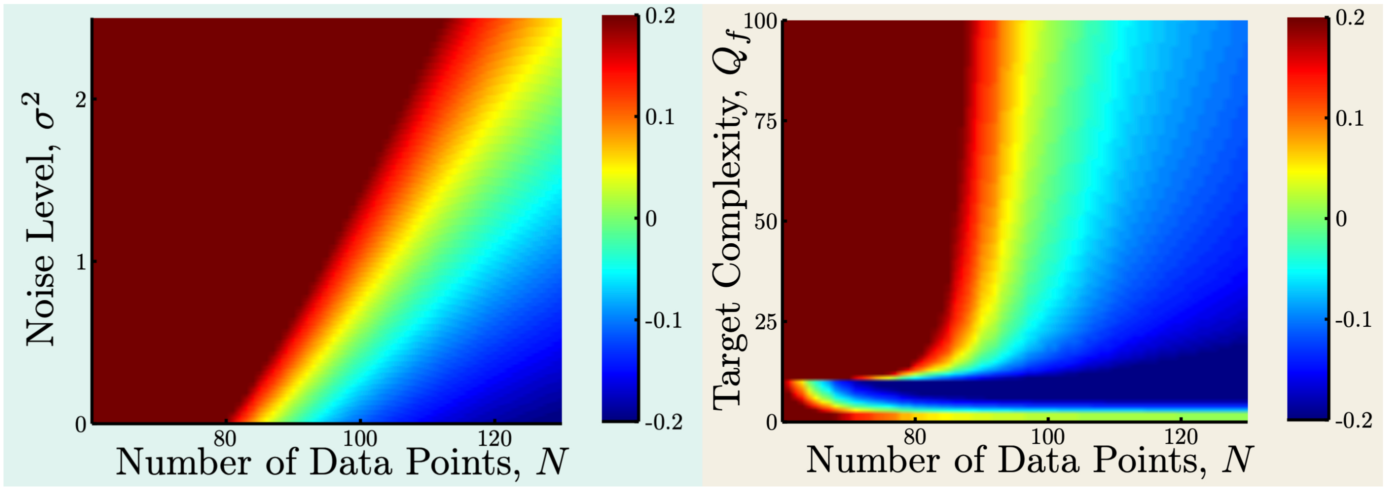
possible combinations, and $\tilde{d} + 1 \approx d_{VC}$ free parameters.

Overfitting

Overfitting is the phenomenon where fitting the observed facts (data) well no longer indicates that we will get a decent out-of-sample error, and may actually lead to the opposite effect. The main case of overfitting is when you pick the hypothesis with lower E_{in}, and it results in higher E_{out}. This means that E_{in} alone is no longer a good guide for learning. Let us start by identifying the cause of overfitting. We demonstrate the phenomenon of overfitting in the following experiment. First, generate the inputs from the input space $\mathcal{X} = [-1, 1]$ with a uniform probability distribution $P(x) = \dfrac{1}{2}$. The target function $f(x)$ is a degree-Q_f polynomial with coefficients selected at random. Additional random noise $\sigma\epsilon_n$ os added to the output

$$y_n = f(x_n) + \sigma\epsilon_n$$

and ϵ_n are iid (independent and identically distributed) standard Normal random variates. g_2 and g_10 are the best git hypotheses for 2 and 10 degree polynomials.
Plot the overfit measure $E_{\text{out}}(g_{10}) - E_{\text{out}}(g_2)$ with respect to Q_f, N, σ^2.



Deterministic Noise

Deterministic noise refers to the discrepancy or error that arises due to the inherent limitations of the hypothesis class in capturing the true underlying relationship between the input features and the target variable. It is different from stochastic noise in that it depends on $\mathcal{H}$ and it is fixed for a given set of inputs.

Regularization

In regularization, we constraint the polynomial transform such that $w_i = 0$ for higher order terms,

$$\mathcal{H}_2 \equiv \{\mathbf{w} \in \mathbb{R}^{10+1},\ w_3 = \cdots = w_{10} = 0\}$$

It is also possible to constrain

$$\mathcal{H}_2' \equiv \{\mathbf{w} \in \mathbb{R}^{10+1},\quad \sum_{q=0}^{10} [[\, w_q \neq 0\,]] \leq 3\}$$

where $\mathcal{H}_2 \subset \mathcal{H}_2' \subset \mathcal{H}_{10}$. Yet another constraint is

$$\mathcal{H}(C) \equiv \{\, \mathbf{w} \in \mathbb{R}^{10+1},\quad \sum_{q=0}^{10} w_q^2 \leq C\,\}$$

We can then determine $\mathbf{w}$ by minimizing E_{in}:

$$\min_{\mathbf{w}\in\mathbb{R}^{Q+1}} E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}^T\mathbf{z}_n - y_n)^2$$

This is a optimization problem with a constraint, therefore we use the Lagrange multipliers.

$$\nabla E_{\text{in}}(\mathbf{w}) + \frac{2\lambda}{N}\mathbf{w} = \frac{2}{N}(\mathbf{Z}^T\mathbf{Z}\mathbf{w} - \mathbf{Z}^T\mathbf{y}) + \frac{2\lambda}{N}\mathbf{w} = 0$$

The optimal $\mathbf{w}$ is

$$\mathbf{w} \leftarrow (\mathbf{Z}^T\mathbf{Z} + \lambda\mathbf{I})^{-1}\mathbf{Z}^T\mathbf{y}$$

This is called the ridge regression in Statistics. We define the Augmented error as

$$E_{\text{aug}}(\mathbf{w}) = E_{\text{in}}(\mathbf{w}) + \frac{\lambda}{N}\mathbf{w}^T\mathbf{w}$$

we denote the general regularizer $\Omega(\mathbf{w})$.

- L_2 **Regularizers** : $\Omega(\mathbf{w}) = \sum_{q=0}^{Q} w_q^2 = ||\mathbf{w}||_2^2$
  It is convex, differentiable everywhere and easy to optimize.

- L_1 **Regularizers** : $\Omega(\mathbf{w}) = \sum_{q=0}^{Q} |w_q| = ||\mathbf{w}||_1$
  It is not differentiable at the corners, and due to its monotonous nature, solutions are often sparse and at the corner.

Validation

The idea of a validation set is almost identical to that of a test set. We remove a subset from the data; this subset is not used in training. We then use this held-out subset to estimate the out-of-sample error. The held-out set is effectively out-of-sample, because it has not been used during the learning.
We partition the data set $\mathcal{D}$ into the training set $\mathcal{D}_{\text{train}}$ of size $N - K$ and the validation set $\mathcal{D}_{\text{val}}$ of size K. Run the learning algorithm using the training set to obtain the final hypothesis, from which we compute the validation error

$$E_{\text{val}}(g^-) = \frac{1}{K} \sum_{\mathbf{x}_n \in \mathcal{D}_{\text{val}}} e(g^-(\mathbf{x}_n), y_n)$$

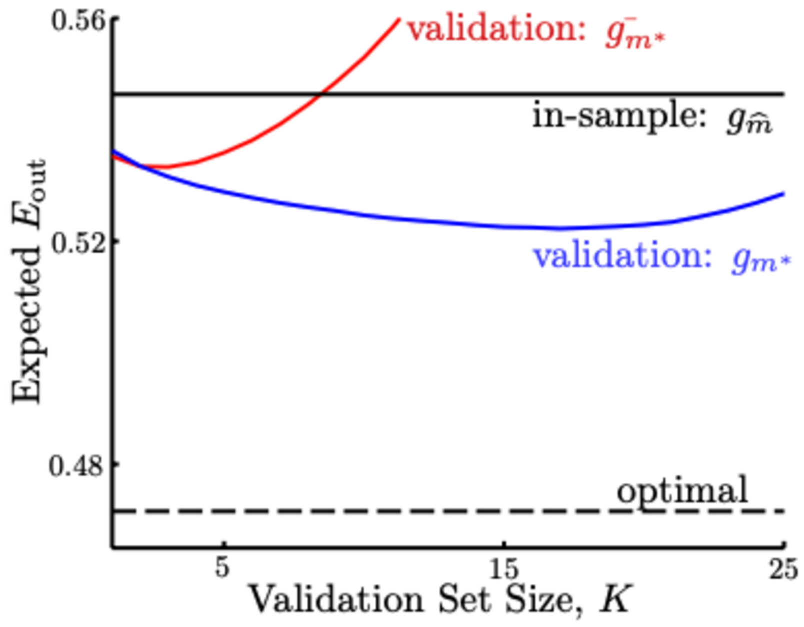
This is an unbiased estimate of the out-of-sample error as their expected value is identical

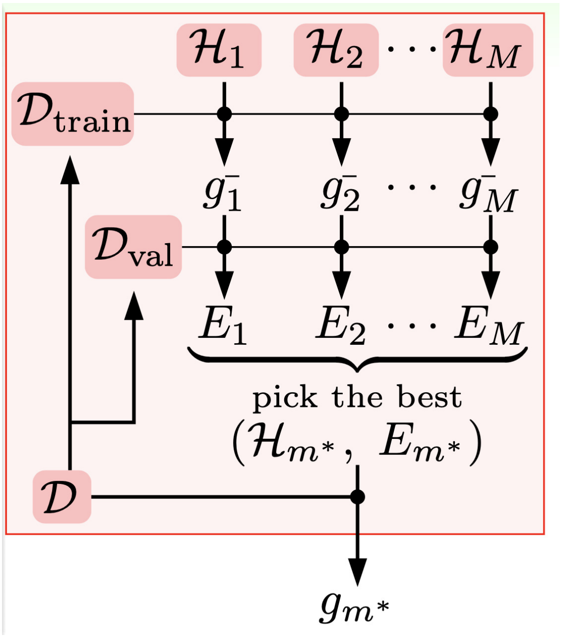
$$\mathbb{E}_{\mathcal{D}_{\text{val}}}[E_{\text{val}}(g^-)] = E_{\text{out}}(g^-)$$

Now, K has to be big enough to be close to E_{out} but not too big as it limits the size of the training data. A rule of thumb in practice is to set $K = N/5$.

Model Selection

Model selection based solely on in-sample error can lead to overfitting, which occurs when a model learns the noise in the training data rather than the underlying patterns. Models with low in-sample error may perform very well on the training data but fail to generalize to new, unseen data. In addition, since models with higher VC dimension is usually favored, leading to a larger VC bound.





In the above figure, we see that validation error is clearly superior to the in-sample error. Validation relies on the following chain of reasoning,

$$E_{\text{out}}(g) \approx E_{\text{out}}(g^-) \approx E_{\text{val}}(g)$$

But the former occurs for small K while the latter for large K. This leads to the idea of the leave-one-out cross validation, for which $K = 1$.

$$\mathcal{D}_{\text{val}} = \{(\mathbf{x}_n, y_n)\} \qquad E_{\text{val}}(g^-) = \text{err}(g^-(\mathbf{x}_n), y_n) = e_n$$

repeat the process for N times

$$E_{\mathrm{cv}} = \frac{1}{N}\sum_{n=1}^{N} e_n$$

The expected value of the cross validation error is

$$\mathbb{E}_{\mathcal{D}}[E_{\mathrm{cv}}] = \mathbb{E}_{\mathcal{D}}[E_{\mathrm{out}}(g^-)] = \bar{E}_{\mathrm{out}}(N-1)$$

where $\bar{E}_{\mathrm{out}}(N-1)$ is the expectation over data sets $\mathcal{D}$ of size $N-1$ of the out-of-sample error produced by the model. It goes without saying this algorithm places a signifacant burden on computaional power. Training is performed N times for every model, and may not be feasible in practice. A popular derivative of leave one-out cross validation is V-fold cross validation. In V-fold cross validation, the data are partitioned into V disjoint sets (or folds) of size approximately N/V. Take 1 for validation and $V-1$ for training. Leave-one-out cross validation is the same as N-fold cross validation. But for $V \ll N$ it lessens the computational needs. The drawback is that you will be estimating E_{out} for a hypothesis g trained on less data (as compared with leave-one-out) and so the discrepancy between $E_{\mathrm{out}}(g^)$ and $E_{\mathrm{out}}(g^-)$ will be larger. Common choices of K is 10 or 5 .

Support Vector Machine

In linear classification, the hypothesis is given by

$$h(\mathbf{x}) = \mathrm{sign}(\mathbf{w}^T\mathbf{x})$$

The hyperplane $\mathbf{w}^T\mathbf{x}$ separates the inputs with different labels. Define the margin of $\mathbf{w}$ as the distance of the closest point to the hyperplane.

$$\mathrm{margin}(\mathbf{w}) = \min_n \ \mathrm{distance}(\mathbf{x}_n, \mathbf{w}) = \min_n \frac{|\mathbf{w}^T\mathbf{x}_n + b|}{||\mathbf{w}||}$$

where we have separated the constant component b from the weight vector. For every n we have $y_n(\mathbf{w}^T x + b) > 0$, so

$$\mathrm{margin}(\mathbf{w}) = \min_n \frac{1}{||\mathbf{w}||} y_n(\mathbf{w}^T\mathbf{x}_n + b)$$

In SVM, we would like to maximize the margin subject to a contraint

$$\min_n y_n(\mathbf{w}^T\mathbf{x}_n + b) = 1$$

The optimization problem is now

$$\max_{b,\mathbf{w}} \frac{1}{||\mathbf{w}||}, \ \text{subject to } \ \min_n y_n(\mathbf{w}^T\mathbf{x}_n + b) = 1$$

or

$$\min_{b,\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{w}, \ \text{subject to } \ \min_n y_n(\mathbf{w}^T\mathbf{x}_n + b) = 1$$

The problem involves minimizing a quadratic function subject to linear constraints, and can be solved using quadratic programming.

$$\min_{\mathbf{u}} \frac{1}{2}\mathbf{u}^T Q\mathbf{u} + \mathbf{p}^T\mathbf{u}, \ \text{subject to } \mathbf{a}_m^T\mathbf{u} \geq c_m$$

where

$$u = \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix}; \quad Q = \begin{bmatrix} 0 & \mathbf{0}_d^T \\ \mathbf{0}_d^T & \mathbf{I}_d \end{bmatrix}; \quad \mathbf{p} = \mathbf{0}_d^T$$

$$\mathbf{a}_n^T = y_n \, [1 \quad \mathbf{x}_n^T]; \quad c_n = 1$$

What we are doing here is minimizing $\mathbf{w}^T\mathbf{w}$ under the constraint $E_{\mathrm{in}} = 0$, whereas in L_2 regularization we minimize E_{in} while $\mathbf{w}^T\mathbf{w} \leq C$.

Non-Linear SVM

In linear SVM, the problem is to optimize $\tilde{d}+1$ variables satisfying N constraints. This would often be difficult to compute when dealing with complex model, we would thus like to change the problem into the optimization of $N+1$ variables satisfying N constraints. First, we start with the non-linear SVM:

$$\min_{b,\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{w}, \quad \text{s.t.} \quad y_n(\mathbf{w}^T\mathbf{z}_n + b) \geq 1$$

Using the Lagrange multipliers, the Lagrange function is given by

$$\mathcal{L}(b,\mathbf{w},\alpha) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{n=1}^{N}\alpha_n(1 - y_n(\mathbf{w}^T\mathbf{z}_n + b))$$

Next, consider the quantity

$$\max_{\alpha_n \geq 0} \mathcal{L}(b,\mathbf{w},\alpha) = \begin{cases} \infty \ , & \text{if constraint violated} \\ \frac{1}{2}\mathbf{w}^T\mathbf{w} \ , & \text{else} \end{cases}$$

Hence, if we consider the minimization problem

$$\min_{b,\mathbf{w}} \left(\max_{\alpha_n \geq 0} \mathcal{L}(b,\mathbf{w},\alpha)\right)$$

we see that it is the same problem as the original. Now,

$$\min_{b,\mathbf{w}} \left(\max_{\alpha_n \geq 0} \mathcal{L}(b,\mathbf{w},\alpha)\right) = \max_{\alpha_n \geq 0} \left(\min_{b,\mathbf{w}} \mathcal{L}(b,\mathbf{w},\alpha)\right)$$

The partial derivative with respect to b is

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{n=1}^{N}\alpha_n y_n = 0$$

for $\mathbf{w}$

$$\frac{\partial \mathcal{L}}{\partial w_i} = w_i - \sum_{n=1}^{N}\alpha_n y_n z_{ni} = 0$$

so we have two new constraints

$$\sum_{n=1}^{N}\alpha_n y_n = 0 \qquad \mathbf{w} = \sum_{n=1}^{N}\alpha_n y_n \mathbf{z}_n$$

plugging these into the original expression:

$$\max\left(-\frac{1}{2}||\sum_{n=1}^{N}\alpha_n y_n \mathbf{z}_n||^2 + \sum_{n=1}^{N}\alpha_n\right) = \max\left(-\frac{1}{2}\sum_{n,m=1}^{N}\alpha_n\alpha_m y_n y_m \mathbf{z}_n^T\mathbf{z}_m + \sum_{n=1}^{N}\alpha_n\right)$$

The constraints are

1. $y_n(\mathbf{w}^T\mathbf{z}_n + b) \geq 1$
2. $\alpha_n \geq 0$
3. $\sum_{n=1}^{N}\alpha_n y_n = 0$; $\mathbf{w} = \sum_{n=1}^{N}\alpha_n y_n \mathbf{z}_n$
4. $\alpha_n(1 - y_n(\mathbf{w}^T\mathbf{z}_n + b)) = 0$
   called the Karush-Kuhn-Tucker (KKT) conditions. The last one is called the KKT dual complementarity condition, or complementary slackness. Since the optimization problem minimizes $\alpha_n(1 - y_n(\mathbf{w}^T\mathbf{z}_n + b))$, so $\alpha_n = 0$ if $y_n(\mathbf{w}^T\mathbf{z}_n + b) > 1$ (condition 1). Under these conditions, the optimal parameters are

$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n y_n \mathbf{z}_n \qquad b = y_n - \mathbf{w}^T \mathbf{z}_n$$

Kernel

The kernel provides a faster way to compute the inner-product of feature transforms, in the case of a 2nd order polynomial transform

$$\Phi_2(\mathbf{x}) = (1, x_1, x_2, \cdots, x_d, x_1^2, \cdots, x_d^2, x_1 x_2, \cdots)$$

The norm is

$$K_{\Phi_2}(\mathbf{x}, \mathbf{x}') = \Phi_2(\mathbf{x})^T \Phi_2(\mathbf{x}') = 1 + \sum_{i=1}^{d} x_i x_i' + \sum_{i=1}^{d} x_i x_i' \sum_{j=1}^{d} x_j x_j' = 1 + \mathbf{x}^T \mathbf{x}' + (\mathbf{x}^T \mathbf{x}')^2$$

we see that the product can be done in $O(d)$ instead of $O(d^2)$. The general polynomial kernel is

$$K_Q(\mathbf{x}, \mathbf{x}') = (\xi + \gamma \mathbf{x}^T \mathbf{x}')^Q \quad \text{with } \gamma > 0,\ \xi \geq 0$$

The Gaussian kernel is a special kernel with infinite dimension.

$$K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma ||\mathbf{x} - \mathbf{x}'||^2)$$

using the transform

$$\Phi(\mathbf{x}) = \exp(-x^2) \cdot (1, \sqrt{2}\mathbf{x}, \cdots, \sqrt{\frac{2^n}{n!}} x^n, \cdots)$$

Soft-Margin SVM

A soft margin support vector machine records the *margin violation* ξ_n for each point $\mathbf{x}_n$ tp distinguish different magnitudes of error. The optimization problem is modified to

$$\min_{b, \mathbf{w}} \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{n=1}^{N} \xi_n, \quad \text{s.t.} \ \ y_n(\mathbf{w}^T \mathbf{z}_n + b) \geq 1 - \xi_n \text{ and } \xi_n \geq 0$$

using the Lagrange multipliers,

$$\mathcal{L} = \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{n=1}^{N} \xi_n + \sum_{n=1}^{N} \alpha_n (1 - \xi_n - y_n(\mathbf{w}^T \mathbf{z}_n + b)) + \sum_{n=1}^{N} \beta_n (-\xi_n)$$

And maximize it under the constrain $\alpha_n, \beta_n \geq 0$. The partial derivative with respect to ξ is

$$\frac{\partial \mathcal{L}}{\partial \xi_n} = C - \alpha_n - \beta_n = 0$$

Thus the problem becomes

$$\max_{0 \leq \alpha_n \leq C} \left(\min_{b, \mathbf{w}} \frac{1}{2} \mathbf{w}^T \mathbf{w} + \sum_{n=1}^{N} \alpha_n (1 - y_n(\mathbf{w}^T \mathbf{z}_n + b)) \right)$$

We already know the optimal $\mathbf{w}$ is

$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n y_n \mathbf{z}_n$$

The condition of complementary slackness forces the Lagrange multiplier terms to be zero:

$$\alpha_n (1 - \xi_n - y_n(\mathbf{w}^T \mathbf{z}_n + b)) = 0 \qquad \beta_n \xi_n = (C - \alpha_n) \xi_n = 0$$

For support vectors(vectors within/on the margin) $\alpha_n > 0$, so

$$b = y_n(1-\xi_n) - \mathbf{w}^T\mathbf{z}_n$$

In particular, support vectors on the boundary satisfies

$$\xi_n = 0 \text{ and } 0 < \alpha_n < C$$

while support vectors with the margin satisfies

$$\xi_n > 0 \text{ and } \alpha_n = C$$

For non-support vectors:

$$\alpha_n = \xi_n = 0$$

The margin violation is thus

$$\xi_n = \max(1 - y_n(\mathbf{w}^T\mathbf{x}_n + b), 0)$$

The unconstrained from of soft-margin SVM is

$$\min_{b,\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{n=1}^{N} \max(1 - y_n(\mathbf{w}^T\mathbf{x}_n + b), 0)$$

Aggregation

Aggregation refers to the process of combining multiple models to improve performance. There are multiple kinds of aggregation

$$G(\mathbf{x}) = \operatorname{sign}\left(\sum_{t=1}^{T} q_t(\mathbf{x})g_t(\mathbf{x})\right)$$

including uniform aggregation, in which $q_t(\mathbf{x}) = 1$, or non-uniform aggregation, in which $q_t(\mathbf{x}) = \alpha_t$, where α_t are constants. Uniform blending for regression is given by

$$G(\mathbf{x}) = \frac{1}{T}\sum_{t=1}^{T} g_t(\mathbf{x})$$

It can be shown that the out-of-sample error of $G(\mathbf{x})$ is always lesser or equal to the average error of $g_t(\mathbf{x})$. For non-uniform blending, the problem is similar to linear regression after a feature transformation.

$$\min_{\alpha_t \geq 0} \frac{1}{N}\sum_{n=1}^{N}\left(y_n - \sum_{t=1}^{T}\alpha_t g_t(\mathbf{x})\right) \quad \leftrightarrow \quad \min_{\mathbf{w}_t} \frac{1}{N}\sum_{n=1}^{N}\left(y_n - \sum_{i=1}^{\tilde{d}}\mathbf{w}_i\phi_i(\mathbf{x})\right)$$

Another technique of aggregation called stacking involves training a meta-model that learns to combine the predictions of multiple base models. The base models make predictions on the input data, and the meta-model is trained to learn how to best combine these predictions to make the final prediction. Given $g_1^-, g_2^-, \cdots, g_T^-$ obtained from the training data $\mathcal{D}_{\text{train}}$, transform the data from the validation set $\mathcal{D}_{\text{val}}$ into $\Phi^-(\mathbf{x}) = (g_1^-, g_2^-, \cdots, g_T^-)$, from which compute $G(\mathbf{x}) = \tilde{g}(\Phi(\mathbf{x}))$.

Bagging and Boosting

A random sampling from a data set $\mathcal{D}$ of size N to generate a new data set $\tilde{\mathcal{D}}_t$ of size N' is called bootstrapping. We then obtain g_t by training with $\tilde{\mathcal{D}}_t$, repeat this for T times. The final algorithm G is the uniform blending of g_t. This process is called bootstrap aggregation (BAGging). Data randomness is important to the performance of bagging, because small changes in the training data can lead to significantly different models or predictions, creating diversity in g_t. Bootstrapping can be represented mathematically by adding a weight vector to the error

$$E_{\text{in}}^{\mathbf{u}} = \frac{1}{N}\sum_{n=1}^{N} u_n^{(t)} \cdot \text{err}(y_n, h(\mathbf{x}_n))$$

where

$$\sum_{n=1}^{N} u_n = N'$$

Consider the case of binary classification. We obtain g_t from

$$g_t \quad \leftarrow \quad \operatorname{argmin}\left(\sum_{n=1}^{N} u_n^{(t)} \cdot [|y_n \neq h(\mathbf{x}_n)|]\right)$$

The goal now is to find $\mathbf{u}^{(t+1)}$ such that g_{t+1} differs from g_t.

$$\frac{\sum_n u_n^{(t+1)}[|y_n \neq g_t(\mathbf{x}_n)|]}{\sum_n u_n^{(t+1)}} = \frac{1}{2}$$

This is equivalent to

$$\sum_{\text{correct}} u_n^{(t+1)} = \sum_{\text{incorrect}} u_n^{(t+1)}$$

Define

$$\epsilon_t = \frac{\sum_{\text{incorrect}} u_n^{(t)}}{\sum_n u_n^{(t)}}$$

Therefore the update rule is

$$u_n^{(t+1)} \quad \leftarrow \quad u_n^{(t)}\sqrt{\frac{1-\epsilon_t}{\epsilon_t}}$$

For incorrectly classified n. And

$$u_n^{(t+1)} \quad \leftarrow \quad u_n^{(t)}\sqrt{\frac{\epsilon_t}{1-\epsilon_t}}$$

For correctly classified n. We can rewrite the above as

$$u_n^{(t+1)} = u_n^{(t)}\left(\frac{1-\epsilon_t}{\epsilon_t}\right)^{-y_n g_t(\mathbf{x}_n)/2} = u_n^{(t)}\exp(-y_n\alpha_t g_t(\mathbf{x}_n))\$\$where\$\$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)\$\$Theinitialweightvectorisusually$$

u_n^{(1)} =\frac{1}{N}

$$Therefore$$

u*n^{(T+1)}=\frac{1}{N}\exp\bigg(-y_n\sum*{t=1}^T\alpha_tg_t(\mathbf{x}_n)\bigg)

$$The final hypothesis is the voting score on\$\mathbf{x}_n : \$\$G(\mathbf{x}) = \operatorname{sign}\left(\sum_{t=1}^{T}\alpha_t g_t(\mathbf{x})\right)\$. Thisiscalledtheadaptiveboostingalgorithm(AdaBoost).$$

h_{s,i,\theta}(\mathbf{x})=s\cdot\operatorname{sign}(x_i-\theta)

$$considerthesumofall\$u_n^{(t)}\$:$$

\sum*{n=1}^Nu_n^{(t)}=\frac{1}{N}\sum*{n=1}^N\exp\bigg(-y*n\sum*{t=1}^T\alpha_tg_t(\mathbf{x}_n)\bigg)

$$sincethesumof\$U_N = \sum_{n=1}^{N} u_n^{(t)}\$decreaseswitheveryiteration,$$

U_{T+1}=2U_T\sqrt{\epsilon_T(1-\epsilon_T)}\leq U_T

$$WecandefinetheAdaboosterrorfunction :$$

\text{err}_{\text{ADA}}(s,y)=\exp(-ys)

$where\$s = \sum_t \alpha_t g_t(\mathbf{x}_n)\$isthelinearscore.Thegoalnowistominimizetheerrorasafunctionofthenewhypothesisanditscoefficient.$

\min*{h,\eta}\frac{1}{N}\sum*{n=1}^N\exp\bigg(-y*n\bigg(\sum*{t=1}^T\alpha*tg_t(\mathbf{x}_n)+\eta ,h(\mathbf{x}_n)\bigg)\bigg)=\sum*{n=1}^Nu_n^{(t)}\exp({-y_n\eta h(\mathbf{x}_n)})

$$approximatingwithtaylorexpansion: \$\$ \approx \sum_{n=1}^N u_n^{(t)} - \eta \sum_{n=1}^N u_n^{(t)} y_n h(\mathbf{x}_n)\$\$ In particular, thetermontherightcanberewrittenas$$

\sum*{n=1}^Nu_n^{(t)}y_n h(\mathbf{x}_n)=\sum*{n=1}^Nu*n^{(t)}-2NE*\text{in}^\mathbf{u}(h(\mathbf{x}))
$\$Theoptimalvalueof\$\eta$ is determined by

$$\min_\eta \sum_{n=1}^N u_n^{(t)} \exp(-y_n \eta h(\mathbf{x}_n)) = \min_\eta \sum_{n=1}^N u_n^{(t)} \left((1-\epsilon_t)e^{-\eta} + \epsilon_t e^{\eta}\right)$$

From which we obtain

$$\eta_t = \alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

And the process repeats itself for the next iteration.

Decision Tree

The decision tree is an example of conditional aggregation,

$$G(\mathbf{x}) = \sum_{t=1}^T q_t(\mathbf{x}) g_t(\mathbf{x})$$

The base hypothesis $g(\mathbf{x})$ is located at the leaf of each path, and q_t depends on if $\mathbf{x}$ is on the path t. From a recursive perspective, we have

$$G(\mathbf{x}) = \sum_{c=1}^C [|b(\mathbf{x}) = c|] \cdot G_c(\mathbf{x})$$

where $b(\mathbf{x})$ is the branching criteria and $G_c(\mathbf{x})$ is sub-tree hypothesis at
the c-th branch. At each node, if the terminate criteria is met, the base hypothesis is returned. If not the data is split into C parts based on the condition of each branch, then repeat until the terminate criteria is met. After which the model return the base hypothesis. The branching criteria is determined such that the *impurity* of the data set is minimized.

$$b(\mathbf{x}) = \text{argmin} \sum_c |\mathcal{D}_c| \cdot \text{impurity}(\mathcal{D}_c)$$

some common impurity functions includes

- **regression error**:

$$\text{impurity}(\mathcal{D}) = \frac{1}{N}\sum_{n=1}^N (y_n - \bar{y})^2$$

- **classification error**:

$$\text{impurity}(\mathcal{D}) = \frac{1}{N}\sum_{n=1}^N [|y_n \neq y^*|]$$

  with y^* being the most common value for y .
- **Gini index**:

$$\text{impurity}(\mathcal{D}) = 1 - \frac{1}{N^2}\sum_{k=1}^K \left(\sum_{n=1}^N [|y_n = k|]\right)^2$$

This is most popular with classification.

Random Forest

Random Forest combines multiple decision trees by bootstrapping and feature expansion, where d' features is sampled from the input data and combined at random.

$$\Phi(\mathbf{x}) = P\mathbf{x}$$

Recall that in bagging, data of size N' is sampled by bootstrapping with $\mathcal{D}$. It is likely that there are examples that are not sampled at all. They are called the out-of-bag examples. They are of size

$$N\big(1 - \frac{1}{N}\big)^{N'} \approx N\big(1 - \frac{1}{N}\big)^{N} \approx \frac{1}{e}N$$

These can be used as validation data when training a random forest, but only for trees in which the data has not been used for training.

$$E_{\text{oob}} = \frac{1}{N}\sum_{n=1}^{N} \text{err}\big(y_n, G_n^-(\mathbf{x}_n)\big)$$

Here G_n^- is the ensemble of decision trees that did not use $\mathbf{x}_n$ for training.

Gradient Boosting

Gradient boosting extends the idea of adaboost to arbitrary error functions

$$\min_{\eta,h} \frac{1}{N}\sum_{n=1}^{N} \text{err}\left(\sum_{\tau=1}^{t-1} \alpha_\tau g_\tau(\mathbf{x}_n) + \eta h(\mathbf{x}_n)\,,\, y_n\right).$$

Take the mean squared error for example,

$$\min_{h} \frac{1}{N}\sum_{n=1}^{N} \left(\sum_{\tau=1}^{t-1} \alpha_\tau g_\tau(\mathbf{x}_n) + \eta h(\mathbf{x}_n) - y_n\right)^2.$$

Taking its taylor expansion

$$\min_{h} \frac{2\eta}{N}\sum_{n=1}^{N} h(\mathbf{x}_n)\left(s_n - y_n\right) + \eta h(\mathbf{x}_n)^2.$$

where

$$s_n = \sum_{\tau=1}^{t-1} \alpha_\tau g_\tau(\mathbf{x}_n)\,.$$

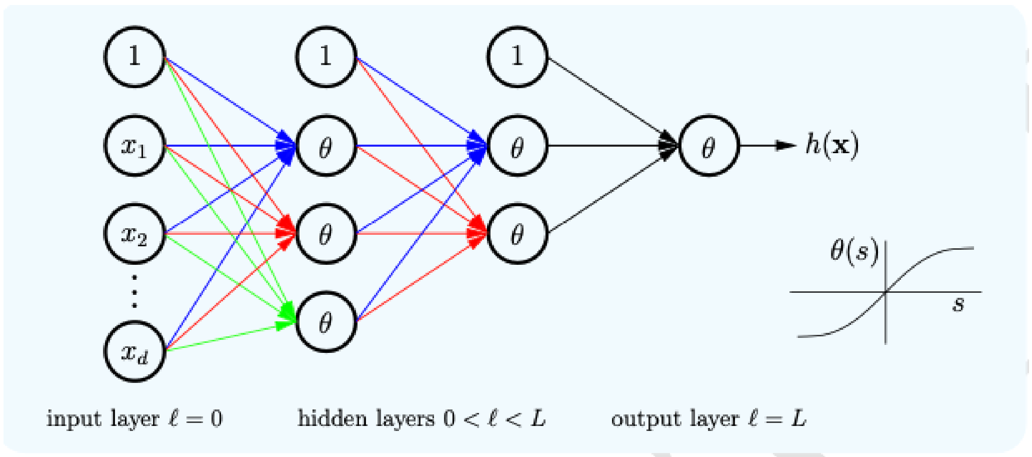
if we remove the η inside the summation, we have equivalently

$$g_t = \arg\min_{h} \frac{\eta}{N}\sum_{n=1}^{N} \big(h(\mathbf{x}_n) - (y_n - s_n)\big)^2.$$

We can determine the optimal g_t by performing regression on $\{(\mathbf{x}_n, y_n - s_n)\}$. From which we can find the blending weight,

$$\alpha_t = \arg\min_{\eta} \frac{1}{N}\sum_{n=1}^{N} \big(\eta g_t(\mathbf{x}_n) - (y_n - s_n)\big)^2.$$

Neural Networks



A neural network consists of layers labeld by $\ell = 0, 1, \cdots L$, with the L-th layer being the output layer. The layers in between are the hidden layers. Each layer ℓ has dimension $d^{(\ell)}$, meaning it has $d^{(\ell)} + 1$ nodes, including the bias node with output 1. Every node has an activation function $\theta(s)$, which transfroms the input to output

$$\mathbf{x}^{(\ell)} = \begin{bmatrix} 1 \\ \theta(\mathbf{s}^{(\ell)}) \end{bmatrix},$$

where the input vector is given by the weighted sum of the previous output

$$\mathbf{s}^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{ij}^{(\ell)} x_i^{(\ell-1)} = (\mathbf{w}^{(\ell)})^T \mathbf{x}^{(\ell-1)} .$$

Some common activation functions include

$$\theta(s) = \tanh(s) ,$$

as well as the rectified linear unit, or ReLU, defined as

$$\theta(s) = \max(s, 0)$$

ReLU has gradient 1 for $s > 0$, so it solves the problem of vanishing gradient during backpropagation. Other variants of the ReLU includes the leaky ReLU

$$\theta(s) = \max(s, 0.01s) = \max(s, 0) + 0.01 \min(s, 0) ,$$

and the parametric ReLU

$$\theta(\alpha, s) = \max(s, \alpha s) .$$

The optimization problem now involves, assuming we are using the squared error

$$e_n = (y_n - s^{(L)})^2 = (y_n - s^{(L)})^2.$$

Backpropagation

To describe the algorithm, we define the sensitivity vector for layer ℓ, which is the sensitivity of the error e with respect to the input signal $s^{(\ell)}$ that goes into layer ℓ. We denote the sensitivity by $\delta^{(\ell)}$,

$$\delta_j^{(\ell)} = \frac{\partial e}{\partial s_j^{(\ell)}}.$$

From chain rule, it follows that

$$\frac{\partial e}{\partial w_{ij}^{(\ell)}} = \delta_j^{(\ell)} x_i^{(\ell-1)} .$$

Now we expand the expression of the sensitivity,

$$\delta_j^{(\ell)} = \sum_{k=1}^{d^{(\ell+1)}} \frac{\partial e}{\partial s_k^{(\ell+1)}} \frac{\partial s_k^{(\ell+1)}}{\partial x_j^{(\ell)}} \frac{\partial x_j^{(\ell)}}{\partial s_j^{(\ell)}} = \sum_{k=1}^{d^{(\ell+1)}} \delta_k^{(\ell+1)} w_{jk}^{(\ell+1)} \theta'(s_j^{(\ell)})\,.$$

The sensitivity in layer ℓ cna be computed backwards from layer $\ell+1$. After we have computed $\delta_j^{(\ell)}$, we can perform gradient descent.

$$w_{ij}^{(\ell)} \;\leftarrow\; w_{ij}^{(\ell)} - \eta\,\delta_j^{(\ell)} x_i^{(\ell-1)}\,.$$

For neural networks using parametric ReLU, the α parameter is also optimized. Problems frequently encountered in neural networks includes the vanishing gradient problem, where the gradient of each layer approaches zero. This issue is more prevalent in deep neural networks, particularly those with activation functions that saturate, such as the hyperbolic functions

$$\theta(s) = \tanh(s)\,.$$

Another problem is the dead neuron problem, where the output s of a network is consistently below zero, resulting in zero gradient. In some ways, This problem is similar to dropout methods.

Optimization methods

The normal schotastic gradient descent methods can be generalized to running averages of the last M iterations, that is

$$\mathbf{w}_t \;\leftarrow\; \mathbf{w}_{t-1} - \eta\,\mathbf{v}_t\,,$$

with

$$\mathbf{v}_t = \frac{1}{M} \sum_{m=t-M+1}^{t} \Delta_m\,.$$

This "moving window" average uniformly weights the gradient of each iteration. For non-uniform weighting, we introduce a constant β called the decay rate such that

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\Delta_t = \sum_{m=1}^{t} \beta^{t-m}(1-\beta)\Delta_t\,.$$

This is said to introduce "momentum" into optimization. This method removes some variance in th e gradient descent, dampens oscillations between ravines, and helps escape local extremas. we can also introduce different learning rates for each components,

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \odot \mathbf{v}_t\,,$$

where the $\odot$ operator denotes element-wise multiplication. One of the choices for the learning rate vector is

$$(\eta_t)_i = \frac{1}{(\Delta_t)_i^2}$$

This reduces the step size for larger gradient. Other optimization methods includes the RMSProp

$$\mathbf{u}_t = \beta \mathbf{u}_{t-1} + (1-\beta)\Delta_t \odot \Delta_t\,,$$

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{u}_t} + \epsilon} \odot \Delta_t\,,$$

and Adam, or adaptive moment estimation, which combines both RMSProp and momentum.

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{u}_t} + \epsilon} \odot \mathbf{v}_t\,,$$