MACHINE LEARNING

PROBABILISTIC PERSPECTIVE

VICTOR

Index

22. Miscellaneous Machine Learning Terms

Hypothesis space (H)

Let the data be D = {(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)}

where y_i is the label of the feature vector x_i.

A hypothesis function h is a function that maps the feature vector x to the label y, ie

$$h : X \to Y$$

The hypothesis space is the set of all possible hypothesis functions, denoted as

$$H = \{h | h : X \to Y\}$$

During the learning process, we try to find the best hypothesis function h from the hypothesis space H that minimizes the error between the predicted label and the true label.

Loss function (L)

Let the data be D = {(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)}

where y_i is the label of the feature vector x_i.

Let the hypothesis function be $h : X \to Y$.

Let $\hat{y_i} = h(x_i)$ be the prediction of this hypothesis function for the feature vector x_i, whereas the true label is y_i.

Then the loss function $L(y_i, \hat{y_i})$ is a function that measures the error between the predicted label and the true label.

Desirable properties

1. Non-negative: $L(y_i, \hat{y_i}) \geq 0$
2. Zero if and only if the prediction is correct: $L(y_i, \hat{y_i}) = 0$ if and only if $y_i = \hat{y_i}$
3. Continous and differentiable for all y_i and $\hat{y_i}$ for smooth optimization using gradient descent.

Examples of loss functions

0-1 loss function

$$L(y_i, \hat{y_i}) = \begin{cases} 0 & \text{if } y_i = \hat{y_i} \\ 1 & \text{if } y_i \neq \hat{y_i} \end{cases}$$

Square loss function

$$L(y_i, \hat{y_i}) = (y_i - \hat{y_i})^2$$

Cross-entropy loss function

The cross-entropy loss function is specifically used for classification tasks where y_i and $\hat{y}_i$ represent probabilities. It is defined as:

$$L(y_i, \hat{y}_i) = -y_i \log(\hat{y}_i) - (1 - y_i) \log(1 - \hat{y}_i)$$

Risk function

Recall that the loss function $L(y_i, \hat{y}_i)$ measures the error between the predicted label and the true label for a single data point.

The risk function $R(h)$ is the expected loss over all data points in the dataset. It is defined as:

$$R(h) = \mathbb{E}_{(x,y)\sim P}[L(y, h(x))]$$

where P is the true data distribution.

Conditional risk

The conditional risk $R(h|x)$ is the expected loss for a given input x. It is defined as:

$$R(h(x)|x) = \mathbb{E}_{y\sim P(y|x)}[L(y, h(x))]$$

Then total risk is the expected value of the conditional risk:

$$R(h) = \mathbb{E}_{x\sim P(x)}[R(h(x)|x)]$$

Empirical risk

We do not know the true data distribution P and we have access to a dataset D sampled from P. Hence we approximate the risk function by the empirical risk:

$$R(h) \approx \frac{1}{n}\sum_{i=1}^{n} L(y_i, h(x_i))$$

where $\{(x_i, y_i)\}_{i=1}^{n}$ is the dataset.

Learning problem

We are given a dataset $D = \{(x_i, y_i)\}_{i=1}^{n}$ sampled from an unknown distribution P.

We want to find a hypothesis function h out of the complete hypothesis space H that minimizes the risk function $R(h)$.

Bayes classifier

Let the data be D = {(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)}

where y_i is the label of the feature vector x_i.

Let the hypothesis space be $H = \{h | h : X \to Y\}$.

The Bayes classifier is defined as:

$$h_B(x) = \begin{cases} 1 & \text{if } P(y = 1|x = x_i) > P(y = 0|x = x_i) \\ 0 & \text{if } P(y = 1|x = x_i) \leq P(y = 0|x = x_i) \end{cases}$$

Notations

- $P(y)$ is also called the prior probability of class y or class prior probability.
- $P(x|y)$ is also called the likelihood of x given class y.
- $P(y|x)$ is also called the posterior probability of class y given x.
- $P(x)$ is also called the evidence.

Bayes classifier is the best classifier for 0-1 loss function

To prove that the Bayes classifier is the best classifier for the 0-1 loss function, we need to show that it minimizes the expected risk (error) for any given input x.

Let's define the 0-1 loss function:

$$L(y, h(x)) = \begin{cases} 0 & \text{if } y = h(x) \\ 1 & \text{if } y \neq h(x) \end{cases}$$

The expected risk for a classifier h is:

$$R(h) = E[L(y, h(x))] = \int L(y, h(x))P(x, y)\, dx\, dy$$

For a given x, the conditional risk is:

$$R(h|x) = E[L(y, h(x))|x] = P(y = 0|x)L(0, h(x)) + P(y = 1|x)L(1, h(x))$$

Now, let's consider two cases:

1. If $h(x) = 0$:

$$R(h|x) = P(y = 1|x)$$

2. If $h(x) = 1$:

$$R(h|x) = P(y = 0|x)$$

The Bayes classifier chooses the class that minimizes this conditional risk:

$$h_B(x) = \arg\min_{y \in \{0,1\}} R(h|x)$$

This means:

- If $P(y = 1|x) > P(y = 0|x)$, then $h_B(x) = 1$
- If $P(y = 1|x) \leq P(y = 0|x)$, then $h_B(x) = 0$

This is exactly the definition of the Bayes classifier we started with.

Since the Bayes classifier minimizes the conditional risk for every x, it also minimizes the overall expected risk $R(h)$. Therefore, the Bayes classifier is the optimal classifier for the 0-1 loss function.

KL Divergence - Kullback-Leibler Divergence

KL divergence is a measure of how one probability distribution diverges from second, probability distribution.

Mathematically, for two probability distributions $P(x)$ and $Q(x)$, the KL divergence from Q to P is defined as:

$D_{KL}(p||q) = \Sigma p(x) * log\left(\frac{p(x)}{q(x)}\right)$ for discrete distributions

$D_{KL}(p||q) = \int p(x) * log\left(\frac{p(x)}{q(x)}\right) dx$ for continuous distributions

where the sum/integral is over all possible events x. And $p(x)$ and $q(x)$ are the probability density functions of distributions $P(x)$ and $Q(x)$ respectively.

Intuition

KL divergence is a measure of how one probability distribution diverges from another. It is a measure of the information lost when Q is used to approximate P.

Properties

- KL divergence is not symmetric: $D_{KL}(P||Q) \neq D_{KL}(Q||P)$
- KL divergence is always non-negative: $D_{KL}(P||Q) \geq 0$
- KL divergence is 0 if and only if P and Q are the same distribution

Minimizing KL Divergence is Equivalent to Maxmimizing Likelihood

For a typical ML problem, all we have are samples from the true distribution $P(x)$ ie $data = \{(x_i)\}_{i=1}^{N}$ where $x_i \in \mathbb{R}^d$ are data points. We do not know the distribution $P(x)$ explicitly.

We try our best to estimate the true distribution $P(x)$ by $Q(x;\theta)$ where θ are the parameters of the model.

We want to know how well our model $Q(x;\theta)$ is performing. We can do this by calculating the KL divergence between the true distribution $P(x)$ and the estimated distribution $Q(x;\theta)$.

$$D_{KL}(P||Q) = \int p(x) * log\left(\frac{p(x)}{q(x;\theta)}\right) dx$$

$$D_{KL}(P||Q) = E_{x\sim p(x)}[log\left(\frac{p(x)}{q(x;\theta)}\right)]$$

$$D_{KL}(P||Q) = E_{x\sim p(x)}[log\left(p(x)\right)] - E_{x\sim p(x)}[log\left(q(x;\theta)\right)]$$

We are trying to find the parameters θ that minimize the KL divergence between $p(x)$ and $q(x;\theta)$.

hence $\theta^* = \underset{\theta}{argmin}\ D_{KL}(p||q(x;\theta))$

$$\theta^* = \underset{\theta}{argmin}\ E_{x\sim p(x)}[log\left(p(x)\right)] - E_{x\sim p(x)}[log\left(q(x;\theta)\right)]$$

because $E_{x\sim p(x)}[log\left(p(x)\right)]$ is constant with respect to θ, we can ignore it.

$$\theta^* = \underset{\theta}{argmax}\ E_{x\sim p(x)}[log\left(q(x;\theta)\right)]$$

$E_{x\sim p(x)}[log\left(q(x;\theta)\right)]$ is called the **Expected Log Likelihood**,

By the law of large numbers, we can approximate the expected log likelihood by the average log likelihood of the data:

$$E_{x\sim p(x)}[log(q(x;\theta))] \approx \frac{1}{N}\sum_{i=1}^{N} log(q(x_i;\theta))$$

Therefore, our optimization problem becomes:

$$\theta^* = \underset{\theta}{argmax}\ \frac{1}{N} \sum_{i=1}^{N} log(q(x_i;\theta))$$

This is equivalent to maximizing the log likelihood of the data.

$$\theta^* = \underset{\theta}{argmax}\ \frac{1}{N} \sum_{i=1}^{N} log(q(x_i;\theta))$$

hence θ is also called the **maximum log likelihood estimate(MLE)**.

Expectation Maximization (EM) algorithm

Let the data be $D = \{t_i\}_{i=1}^N$ where $t_i \in \mathbb{R}^d$ are iid $p_t(t)$ data points.

Latent variable models

Let each data point x_i be associated with a latent variable z_i that is not observed. z_i is a random variable that takes values in some finite set $1, \ldots, K$ and represents the membership of x_i to one of K clusters.

Hence the data can be represented as $D = \{(t_i, z_i)\}_{i=1}^N$ where t_i is the observed data and z_i is the latent variable and (t_i, z_i) are iid p_{tz}.

The marginal distribution of the observed data is given by:

$$p_t(t) = \sum_z p_{tz}(t, z)$$

Variational inference

We want to maximize the log-likelihood of the observed data:

$$\ell(\theta) = \log \sum_z p_{tz}^\theta(t, z)$$

Let $q(z)$ be an arbitrary distribution over z.

$$\ell(\theta) = \log \sum_z q(z) \frac{p_{tz}^\theta(t, z)}{q(z)}$$

$$\ell(\theta) = \log \mathbb{E}_{q(z)} \left[\frac{p_{tz}^\theta(t, z)}{q(z)} \right]$$

By Jensen's inequality, we have:

$$\log \mathbb{E}_{q(z)} \left[\frac{p_{tz}^\theta(t, z)}{q(z)} \right] \geq \mathbb{E}_{q(z)} \left[\log \frac{p_{tz}^\theta(t, z)}{q(z)} \right]$$

Hence, we have:

$$\ell(\theta) \geq \mathbb{E}_{q(z)}\left[\log \frac{p_{tz}^{\theta}(t,z)}{q(z)}\right]$$

$\ell(\theta)$ is the log-likelihood of the observed data it is also called the evidence.

$\mathbb{E}_{q(z)}\left[\log \frac{p_{tz}^{\theta}(t,z)}{q(z)}\right]$ is the lower bound of the log-likelihood and is also called the **evidence lower bound (ELBO)**.

Note that the ELBO is a function of $q(z)$ and θ.

Optimizing ELBO

Instead of maximizing the evidence, we maximize the evidence lower bound (ELBO). We do it by maximizing the ELBO with respect to $q(z)$ and θ.

Making ELBO tight

To make the ELBO tight, we consider the difference between the evidence and the ELBO:

$$\ell(\theta) - \mathrm{ELBO}(q,\theta) = \log p_t^{\theta} - \mathbb{E}_{q(z)}\left[\log \frac{p_{tz}^{\theta}(t,z)}{q(z)}\right]$$

$$\ell(\theta) - \mathrm{ELBO}(q,\theta) = \log p_t^{\theta} - \mathbb{E}_{q(z)}\left[\log \frac{p_{z|t}^{\theta}(z|t)p_t^{\theta}(t)}{q(z)}\right]$$

$$\ell(\theta) - \mathrm{ELBO}(q,\theta) = \log p_t^{\theta} - \mathbb{E}_{q(z)}\left[\log p_{z|t}^{\theta}(z|t) + \log p_t^{\theta}(t) - \log q(z)\right]$$

$$\ell(\theta) - \text{ELBO}(q, \theta) = \log p_t^\theta - \mathbb{E}_{q(z)}\left[\log p_{z|t}^\theta(z|t)\right] - \mathbb{E}_{q(z)}\left[\log p_t^\theta(t)\right] + \mathbb{E}_{q(z)}\left[\log q(z)\right]$$

$$\ell(\theta) - \text{ELBO}(q, \theta) = \log p_t^\theta - \mathbb{E}_{q(z)}\left[\log p_{z|t}^\theta(z|t)\right] - \log p_t^\theta(t) + \mathbb{E}_{q(z)}\left[\log q(z)\right]$$

$$\ell(\theta) - \text{ELBO}(q, \theta) = -\mathbb{E}_{q(z)}\left[\log p_{z|t}^\theta(z|t)\right] + \mathbb{E}_{q(z)}\left[\log q(z)\right]$$

$$\ell(\theta) - \text{ELBO}(q, \theta) = \mathbb{E}_{q(z)}\left[\log q(z) - \log p_{z|t}^\theta(z|t)\right]$$

$$\ell(\theta) - \text{ELBO}(q, \theta) = \mathbb{E}_{q(z)}\left[\log \frac{q(z)}{p_{z|t}^\theta(z|t)}\right]$$

$$\ell(\theta) - \text{ELBO}(q, \theta) = KL(q(z)||p_{z|t}^\theta(z|t))$$

The difference between the evidence and the ELBO is the Kullback-Leibler (KL) divergence between $q(z)$ and $p_{z|t}^\theta(z|t)$. To make the ELBO tight, we need to minimize this KL divergence.

The KL divergence is always non-negative and equals zero if and only if the two distributions are identical. Therefore, to make the ELBO tight, we set:

$$q(z) = p_{z|t}^\theta(z|t)$$

This choice of $q(z)$ makes the ELBO equal to the evidence, achieving the tightest possible bound.

EM Algorithm

The Expectation-Maximization (EM) algorithm is an iterative method to find maximum likelihood estimates of parameters in statistical models with latent variables. It consists of two main steps:

1. **E-step (Expectation)**: Compute the expected value of the log-likelihood function with respect to the conditional distribution of z given t under the current estimate of the parameters θ.

2. **M-step (Maximization)**: Find the parameter that maximizes this expected log-likelihood.

Formally, the EM algorithm can be described as follows:

1. Initialize $\theta^{(0)}$

2. Repeat until convergence:

- E-step: Compute $q^{(t)}(z) = p_{z|t}^{\theta^{(t-1)}}(z|t)$
- M-step:

1. $\theta^{(t)} = \arg\max_\theta \mathbb{E}_{q^{(t)}(z)} \left[\log \frac{p_{tz}^\theta(t,z)}{q(z)}\right]$
2. $\theta^{(t)} = \arg\max_\theta \mathbb{E}_{q^{(t)}(z)} \left[\log p_{tz}^\theta(t,z)\right] - \mathbb{E}_{q^{(t)}(z)} \left[\log q(z)\right]$
3. $\theta^{(t)} = \arg\max_\theta \mathbb{E}_{q^{(t)}(z)} \left[\log p_{tz}^\theta(t,z)\right]$ as second term is constant wrt θ

The EM algorithm guarantees that the likelihood increases at each iteration and converges to a local maximum.

EM algorithm for GMM

Let's apply the EM algorithm to the Gaussian Mixture Model (GMM) we discussed earlier. Recall that in a GMM, we have:

- Observed data points: $\mathbf{x} = (x_1, ..., x_N)$
- Latent variables: $\mathbf{z}$, where $z_i \in \{1, ..., m\}$ indicates gaussian component
- $p_t(t) = \sum_{j=1}^m \alpha_j \mathcal{N}(t; \mu_j, \xi_j)$
- α_j are mixing coefficients, $\sum_{j=1}^m \alpha_j = 1$
- μ_j are mean vectors
- ξ_j are covariance matrices
- $p(t_i|z_i = k) = \mathcal{N}(\mathbf{t_i}; \mu_k, \Sigma_k)$
- Parameters: $\theta = (\alpha_1, ..., \alpha_m, \mu_1, ..., \mu_m, \xi_1, ..., \xi_m)$

The EM algorithm for GMM proceeds as follows:

1. **Initialization**:

Choose initial values for the parameters $\theta = (\alpha_1, ..., \alpha_m, \mu_1, ..., \mu_m, \xi_1, ..., \xi_m)$.

2. **E-step**:

Compute the posterior probabilities (responsibilities) for each data point and each Gaussian component:

$$q^{t+1}(z=s) = p^{\theta^k}_{z|t}(z=s|t=t_i) = \frac{p^{\theta^k}_{tz}(z)}{p^{\theta^k}_{t}(t)} = \frac{N(t;t_i,\mu_s,\xi_s)\alpha_s}{\sum_{j=1}^{m}\alpha_j N(t_j;\mu_j,\xi_j)}$$

3. **M-step**:

Update the parameters:

$$\theta^{k+1} = \arg\max_{\theta} ELBO(q,\theta)$$

$$\theta^{k+1} = \arg\max_{\theta}$$

$$= \arg\max_{\theta}(E_q \log N(t,\mu_\xi,\xi)\alpha_s)$$

The EM algorithm for GMM alternates between these steps until convergence, effectively maximizing the likelihood of the observed data under the Gaussian mixture model.

Linear Discriminant Analysis(LDA) from bayesian perspective

Let the data be $D = \{(x_i, y_i)\}_{i=1}^{N}$ where $x_i \in \mathbb{R}^d$ and $y_i \in \{1, 2\}$

Let's assume the parametric form of the conditional density $p(x|y)$ is:

$$p(x|y = 1) \sim N(x; \mu_1, \Sigma)$$

$$p(x|y = 0) \sim N(x; \mu_2, \Sigma)$$

where $N(x; \mu, \Sigma)$ denotes a multivariate Gaussian distribution with mean μ and covariance matrix Σ. Note that we assume the covariance matrix Σ is the same for both classes, which is a key assumption in LDA.

In simple words, we are assuming that the data is distributed as Gaussian in each class with different means but shared covariance matrix.

Also, let's assume that the prior probabilities $P(y = 1)$ and $P(y = 0)$ are same ie $1/2$.

Derivation

Let's derive the decision boundary for LDA using the Bayes classifier.

1. Bayes Classifier:

The Bayes classifier is defined as:

$$h_B(x) = \begin{cases} 1 & \text{if } P(y = 1|x = x_i) > P(y = 0|x = x_i) \\ 0 & \text{if } P(y = 1|x = x_i) \leq P(y = 0|x = x_i) \end{cases}$$

2. Using Bayes' Rule:

$$P(y = k|x) = \frac{P(x|y = k) \cdot P(y = k)}{P(x)}$$

3. Decision Rule:

Choose class 1 if:

$$P(x|y = 1) \cdot P(y = 1) > P(x|y = 0) \cdot P(y = 0)$$

4. Taking logarithms (monotonic transformation):

$$\log(P(x|y=1)) + \log(P(y=1)) > \log(P(x|y=0)) + \log(P(y=0))$$

5. Substituting Gaussian densities:

$$-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1) - \frac{d}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma| + \log(P(y=1)) >$$

$$-\frac{1}{2}(x-\mu_2)^T\Sigma^{-1}(x-\mu_2) - \frac{d}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma| + \log(P(y=0))$$

6. Simplifying:

$$-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1) + \log(P(y=1)) > -\frac{1}{2}(x-\mu_2)^T\Sigma^{-1}(x-\mu_2) + \log(P(y=0))$$

7. Expanding the quadratic terms:

$$-\frac{1}{2}(x^T\Sigma^{-1}x - 2\mu_1^T\Sigma^{-1}x + \mu_1^T\Sigma^{-1}\mu_1) + \log(P(y=1)) >$$

$$-\frac{1}{2}(x^T\Sigma^{-1}x - 2\mu_2^T\Sigma^{-1}x + \mu_2^T\Sigma^{-1}\mu_2) + \log(P(y=0))$$

8. Cancelling out common terms:

$$\mu_1^T\Sigma^{-1}x - \frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \log(P(y=1)) >$$

$$\mu_2^T\Sigma^{-1}x - \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \log(P(y=0))$$

9. Rearranging:

$$(\mu_1-\mu_2)^T\Sigma^{-1}x > \frac{1}{2}(\mu_1^T\Sigma^{-1}\mu_1 - \mu_2^T\Sigma^{-1}\mu_2) + \log(P(y=0)/P(y=1))$$

10. Final decision boundary:

$$h_B(x) = \begin{cases} 1 & \text{if } w^Tx + w_0 > 0 \\ 0 & \text{otherwise} \end{cases}$$

where:

$$w = \Sigma^{-1}(\mu_1 - \mu_2)$$

$$w_0 = -\frac{1}{2}(\mu_1^T\Sigma^{-1}\mu_1 - \mu_2^T\Sigma^{-1}\mu_2) - \log(P(y=0)/P(y=1))$$

Note

- The decision boundary is linear in x. This is the reason it is called Linear Discriminant Analysis.
- The decision boundary $w^T x + w_0 = 0$ is a hyperplane.
- The decision boundary will not be linear if the covariance matrices are not the same for both classes.

Linear regression

Let the data be (x_i, y_i) for $i = 1, 2, ..., n$ where $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$.

then we can model y_i as:

$$Y = \beta_0 + \beta_1 X + \varepsilon$$

Where:

- $Y \in \mathbb{R}$ is the dependent variable
- $X \in \mathbb{R}^d$ is the independent variable (or feature vector in higher dimensions)
- β_0 is the y-intercept (bias term)
- β_1 is the slope (or coefficient vector in higher dimensions)
- ε is the error term

Ideal regressor

For mean squared error loss, the ideal regressor is defined as:

$$h^*(x) = E[Y|X = x]$$

Where $E[Y|X = x]$ is the conditional expectation of Y given $X = x$.

Derivation of ideal regressor

To derive the ideal regressor for squared error loss, we need to find the function h(x) that minimizes the expected squared error:

$$h^* = \arg\min_h E[(Y - h(X))^2]$$

Let's expand this expectation:

$$E[(Y - h(X))^2] = E[Y^2] - 2E[Yh(X)] + E[h(X)^2]$$

To minimize this, we differentiate with respect to h(x) and set it to zero:

$$\frac{\partial}{\partial h(x)} E[(Y - h(X))^2] = -2E[Y|X = x] + 2h(x) = 0$$

Solving for $h^*(x)$:

$$h^*(x) = E[Y|X = x]$$

This shows that the ideal regressor for squared error loss is indeed the conditional expectation of Y given $X = x$.

Now, let's derive this further using our linear model:

$$E[Y|X = x] = E[\beta_0 + \beta_1 X + \varepsilon|X = x]$$

Using the linearity of expectation:

$$E[\beta_0 + \beta_1 X + \varepsilon|X = x] = E[\beta_0|X = x] + E[\beta_1 X|X = x] + E[\varepsilon|X = x]$$

Simplifying:

1. $E[\beta_0|X = x] = \beta_0$ (since β_0 is a constant)
2. $E[\beta_1 X|X = x] = \beta_1 x$ (since X is fixed at x)
3. $E[\varepsilon|X = x] = 0$ (assuming the error term has zero mean and is independent of X)

Therefore:

$$h^*(x) = \beta_0 + \beta_1 x$$

Interpretation

The ideal regressor $h^*(x) = \beta_0 + \beta_1 x$ minimizes the expected squared error. It represents the best possible prediction of Y given $X = x$ under the squared error loss, assuming the linear model is correct. This function provides the average value of Y for each value of X, effectively capturing the underlying linear relationship between the variables while averaging out the random noise (represented by ε).

Empirical Risk Minimization

In practice, we don't have access to the true distribution of the data, so we can't directly minimize the expected risk. Instead, we use the empirical risk as an approximation:

$$\hat{R}(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\beta_0 + \beta_1 x_i))^2$$

where n is the number of samples in our dataset.

Note that in this formulation, we don't explicitly see the error term ε that was present in our original model $Y = \beta_0 + \beta_1 X + \varepsilon$. This is because the empirical risk is calculated using the observed y_i values, which already incorporate the random error.

To find the optimal parameters β^*, we minimize this empirical risk:

$$\beta^* = \arg\min_{\beta} \hat{R}(\beta)$$

This optimization problem has a closed-form solution, which can be derived using linear algebra:

$$\beta^* = (X^T X)^{-1} X^T Y$$

Here, X is the design matrix:

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1d} \\ 1 & x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nd} \end{bmatrix}_{n \times (d+1)}$$

This matrix has n rows (one for each data point) and $d + 1$ columns. The first column is all 1s (for the intercept term), and the remaining d columns contain the feature values of our data points.

And Y is the vector of target values:

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}_{n \times 1}$$

This is a column vector with n rows, containing the scalar y values of our data points.

The solution β^* is a $(d+1) \times 1$ vector:

$$\beta^* = \begin{bmatrix} \beta_0^* \\ \beta_1^* \\ \vdots \\ \beta_d^* \end{bmatrix}_{(d+1)\times 1}$$

To retrieve the individual β_1^* values from β^*:

1. β_0^* is the first element of β^*, i.e., $\beta^*[0]$
2. $\beta_1^* = \begin{bmatrix} \beta_1^* \\ \vdots \\ \beta_d^* \end{bmatrix}_{(d)\times 1}$

This solution is known as the Ordinary Least Squares (OLS) estimator.

Logistic regression also known as logit regression or binary classification

Let the data be D = {(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)}

where $y_i \in \{0, 1\}$ is the binary label of the feature vector x_i.

Then the logistic regression model is defined as:

$$P(y_i = 1|x_i) = \frac{1}{1 + e^{-w^T x_i + b}}$$

$$P(y_i = 0|x_i) = \frac{1}{1 + e^{w^T x_i + b}}$$

where w is the parameter vector.

Note that

$$P(y_i = 1|x_i) + P(y_i = 0|x_i) = 1$$

Why define the model like this?

because consider:

$$p(y = 1|x) = \frac{p(x|y = 1)}{p(x|y = 1) + p(x|y = 0)}$$

$$p(y = 1|x) = \frac{1}{1 + \frac{p(x|y=0)}{p(x|y=1)}}$$

if:

- $y_i \in \{0, 1\}$ and the priors ie $P(y_i = 1)$ and $P(y_i = 0)$ are equal
- And $p(x_i|y_i = 0)$ and $p(x_i|y_i = 1)$ follows an gaussian distribution with different means and equal covariance matrix

then this becomes:

$$P(y_i = 1|x_i) = \frac{1}{1 + e^{-w^T x_i + b}}$$

$$P(y_i = 0|x_i) = \frac{1}{1 + e^{w^T x_i + b}}$$

How to find the parameters w and b?

For the binary classification problem, use logistic regression as hypothesis function and cross-entropy loss function as the loss function and perform gradient descent to find the parameters w and b.

Why not use hard labels (0 or 1)?

Using hard labels (0 or 1) directly in backpropagation can lead to several issues, hence we use soft labels. Soft labels provide a continuous probability distribution over the classes, allowing for smoother gradients and more stable training. This approach enables the model to capture uncertainty and learn more nuanced decision boundaries compared to hard binary classifications.

Softmax regression also known multiclass regression

Let the data be D = {(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)}

where $y_i \in \{1, 2, ..., K\}$ is the multiclass label of the feature vector x_i, and K is the number of classes.

Then the softmax regression model is defined as:

$$P(y_i = k|x_i) = \frac{e^{w_k^T x_i + b_k}}{\sum_{j=1}^{K} e^{w_j^T x_i + b_j}}$$

where w_k is the parameter vector for class k, and b_k is the bias term for class k.

Note that

$$\sum_{k=1}^{K} P(y_i = k|x_i) = 1$$

This ensures that the probabilities for all classes sum up to 1, providing a valid probability distribution over the K classes.

The reason for using softmax regression instead of hard labels and the method to find the parameters w_k and b_k is the same as the reason for using logistic regression.

How to find the parameters w_k and b_k?

For the multiclass classification problem, we use softmax regression as the hypothesis function and cross-entropy loss function as the loss function. We then perform gradient descent to find the parameters w_k and b_k for each class k.

Why use softmax instead of hard labels?

Using softmax instead of hard labels (e.g., one-hot encoding) in multiclass classification offers several advantages:

1. Smooth gradients: Softmax provides a continuous, differentiable output, allowing for smoother gradients during backpropagation. This leads to more stable and efficient training.

2. Probability interpretation: Softmax outputs can be interpreted as probabilities, giving a measure of the model's confidence in its predictions for each class.

Maximum aposteriori estimate(MAP)

The Maximum A Posteriori (MAP) estimate is defined as:

$$\theta^*_{MAP} = \arg\max_{\theta} p(\theta|V) = \arg\max_{\theta} p(V|\theta)p(\theta)$$

where

- θ^*_{MAP} is the MAP estimate of the parameter θ
- $p(\theta|V)$ is the posterior probability of the parameter θ given the observed data V
- $p(V|\theta)$ is the likelihood of the observed data V given the parameter θ
- $p(\theta)$ is the prior probability of the parameter θ

Note that:

- Unlike MLE, MAP estimation incorporates prior knowledge about the parameter θ and observed data V where as MLE only depends on the observed data V.
- MAP provides a balance between the likelihood of the data and the prior beliefs, leading to more robust estimates, especially when the data is limited.

Conjugate prior

A conjugate prior is a type of prior distribution $p(\theta)$ such that when multiplied with the likelihood function $p(V|\theta)$ the posterior distribution $p(\theta|V)$ that we get is in the same form as the prior $p(\theta)$.

In simple terms, the posterior $p(\theta)$ belongs to the same family of distributions as the prior $p(\theta|V)$.

Non parametric density estimation

What is parametric density estimation?

In parametric density estimation, we assume that the data is generated from a known distribution, such as the normal distribution, and we estimate the parameters of the distribution using various methods like maximum likelihood estimation or risk minimization.

What is non-parametric density estimation?

However, in non-parametric density estimation, we make no assumptions about the form of the distribution and estimate the density directly from the data.

The basic idea behind non-parametric density estimation is to estimate the probability density function (PDF) directly from the data without assuming a specific functional form. One way to approach this is by considering the probability of a data point falling within a certain region.

Let $D = \{x_1, x_2, ..., x_n\}$ be our dataset. The probability of a data point falling within a region R can be estimated as:

$$P(\text{data point in region } R) = \frac{\text{number of data points in region } R}{\text{total number of data points}} = \frac{k}{n}$$

where k is the number of data points in region R, and n is the total number of data points.

Also

$$P(\text{data point in region } R) = p(x) * V$$

where $p(x)$ is the probability density at point x and V is the volume of the region R.

Combining the above two equations, we get:

$$\frac{k}{n} = p(x) * V$$

Rearranging this equation, we get:

$$p(x) = \frac{k}{nV}$$

This forms the basis for various non-parametric density estimation techniques.

Parzen window estimate also known as the kernel density estimate

Basic idea

Recall that we had derived the following equation for non-parametric density estimation:

$$p(x) = \frac{k}{nV}$$

where:

- k is the number of data points in region R
- n is the total number of data points in the dataset D
- V is the volume of the region R

In Parzen window estimate, we fix the volume V and count k by using a window function.

Problem setting

Given a set of data points $D = \{x_1, x_2, ..., x_n\}$, we want to estimate the probability density function $p(x)$ at a given point x ie model the distribution of data points in the dataset.

Formulation using uniform kernel (rectangular kernel)

Let's define the formulation for the Parzen window estimate:

1. Define the volume V_n:

$$V_n = (h_n)^d$$

where:

- h_n is the length of the hypercube in $\mathbb{R}^d$
- d is the dimension of the data
- V_n is the volume of the hypercube in $\mathbb{R}^d$

2. Define the window function $\phi(u)$:

$$\phi(u) = \begin{cases} 1 & \text{if } |u_j| \leq \frac{1}{2}, j = 1, \ldots, d \\ 0 & \text{otherwise} \end{cases}$$

where:

- $\phi(u)$ returns 1 if the point u is within the unit hypercube centered at the origin, and 0 otherwise.
- u_j is the jth coordinate of the point u.

3. Then the window function centered at a data point x_i is:

$$\phi(\frac{x - x_i}{h_n}) = \begin{cases} 1 & \text{if } x \text{ is in the hypercube centered at } x_i \text{ of side } h_n \\ 0 & \text{otherwise} \end{cases}$$

4. Count k_n, the number of points in the hypercube centered at x_i of side h_n:

$$k_n = \sum_{i=1}^{n} \phi(\frac{x - x_i}{h_n})$$

5. The Parzen window estimate:

$$p(x) = \frac{k}{nV} = \frac{\sum_{i=1}^{n} \phi(\frac{x - x_i}{h_n})}{n(h_n)^d}$$

Note that:

- Here, h_n is a hyperparameter that controls the width of the window.
- As $n \to \infty$, if $h_n \to 0$ and $nh_n^d \to \infty$, then the estimate converges to the true density.

Parzen window with Gaussian kernel

1. Gaussian kernel:

$$\phi(u) = \frac{1}{(2\pi)^{d/2}} e^{-\frac{1}{2}u^2}$$

2. The window function centered at a data point x_i is:

$$\phi(\frac{x - x_i}{h_n}) = \frac{1}{(2\pi)^{d/2}} e^{-\frac{1}{2}(\frac{x-x_i}{h_n})^2}$$

3. The Parzen window estimate:

$$p(x) = \frac{k}{nV} = \frac{\sum_{i=1}^{n} \phi(\frac{x-x_i}{h_n})}{n(h_n)^d}$$

Algorithm

1. Choose a value for h_n

2. For each test data point x_i, find the number of points in the hypercube centered at x_i of side h_n

3. Calculate the Parzen window estimate $p(x)$ using the number of points found in the previous step

K Nearest neighbour (KNN)

Basic idea

Recall that we had derived the following equation for non-parametric density estimation:

$$p(x) = \frac{k}{nV}$$

where:

- k is the number of data points in region R
- n is the total number of data points in the dataset D
- V is the volume of the region R

In K-Nearest Neighbour (KNN) method, we fix the volume V and count k

Problem setting

Given a set of data points $D = \{x_1, y_1\}, \{x_2, y_2\}, ..., \{x_n, y_n\}$, where $x_i \in \mathbb{R}^d$ is the feature vector and $y_i \in \{1, 2, ..., C\}$ is the class label.

Formulation

The posterior probability of class c given input x_i can be estimated as:

$$p(y = c|x_i) = \frac{p(x_i, y = c)}{p(x_i, y = 1) + p(x_i, y = 2) + ... + p(x_i, y = C)}$$

$$p(y = c|x_i) = \frac{\frac{k_i}{nV}}{\sum_{j=1}^{C} \frac{k_j}{nV}} = \frac{k_i}{\sum_{j=1}^{C} k_j}$$

Now we can use bayes classification rule to assign label to the new data point x_i.

Algorithm

1. Choose a value for k
2. For each test data point x_i, find the k nearest neighbours in the training data D
3. Assign the class label to x_i based on the majority class label of the k nearest neighbours

Bias variance tradeoff

It can be shown that average risk for square error loss can be decomposed into three components:

$$R_{\text{avg}}(h) = \mathbb{E}_{P_D}\mathbb{E}_{P_{x,y}}\left[\left(h_D(x) - \hat{h}(x)\right)^2\right] \quad // \text{ variance (sensitivity to dataset)}$$

$$+\mathbb{E}_{P_{x,y}}\left[\left(\hat{h}(x) - h^*(x)\right)^2\right] \quad // \text{ bias (how different is avg classifier from optimum classifier)}$$

$$+\mathbb{E}_{P_{x,y}}\left[\left(h^*(x) - y\right)^2\right] \quad // \text{ irreducible noise (nothing can be done about this)}$$

where:

- $R_{\text{avg}}(h)$ is the average risk.
- p_D is the distribution of the training data. In simple words,$p_D(D = D_i)$ is the probability of observing the training data D_i.
- $\mathbb{E}_{P_D}$ is the expectation over the distribution of the training data.
- $p_{x,y}$ is the distribution of the input-output pairs.
- $\mathbb{E}_{P_{x,y}}$ is the expectation over the input-output pairs.
- $h_D(x)$ is classifier trained on the training data D.
- $\hat{h}(x)$ is the average classifier ie $\hat{h}(x) = \mathbb{E}_{P_D}(h_D(x))$
- $h^*(x)$ is the optimal classifier

Term breakdown

Variance:

$$\mathbb{E}_{P_D}\mathbb{E}_{P_{x,y}}\left[\left(h_D(x) - \hat{h}(x)\right)^2\right]$$

Measures how sensitive the learned classifier $h_D(x)$ is to different training datasets D. High variance means the model changes significantly with different training sets, making it unstable.

Bias:

$$\mathbb{E}_{P_{x,y}}\left[\left(\hat{h}(x)-h^*(x)\right)^2\right]$$

Measures how much the average learned classifier $\hat{h}(x)$ deviates from the optimal classifier $h^*(x)$ that minimizes the error. High bias means the model is systematically inaccurate or underfits.

Irreducible Noise:

$$\mathbb{E}_{P_{x,y}}\left[\left(h^*(x)-y\right)^2\right]$$

This term captures the inherent noise in the data y. No model can reduce this part, as it reflects randomness or variability in the data that is not related to the features x.

Bias variance tradeoff

1. **Relationship**:

- As we decrease bias by making our model more complex (e.g., using more features or a more flexible model), we often increase variance. This means the model may fit the training data very well but perform poorly on unseen data due to overfitting.
- Conversely, if we increase bias by simplifying the model (e.g., using fewer features or a more rigid model), we may reduce variance, but at the cost of underfitting the training data.

2. **Optimal Point**:

- The goal is to find a balance where both bias and variance are minimized, leading to the lowest possible total error. This is often visualized as a U-shaped curve where the total error is minimized at a certain level of model complexity.

Regularization

Regularization is a technique to increase model bias to reduce variance by constraining empirical risk minimization.

regularized empirical risk minimization:

$$\text{Reg } ERM = \min_{h \in H} \hat{R}(h_\theta) \quad \text{s.t. } \Omega(h_\theta) < k$$

here $\Omega(h_\theta) < k$ is the regularization function which is a design choice.

This can be solved using the method of Lagrange multipliers.

$h(x) = \arg\min_{h_\theta \in H} \left(\hat{R}(h_\theta) + \lambda \Omega(\theta) \right)$

here λ is the regularization parameter which is a design choice.

Norm based regularization

When $\Omega(\theta)$ is taken to be the p-norm, i.e., $||\theta||_p$:

- If $p = 1$, then we have L1 (lasso) regularization.
- If $p = 2$, then we have L2 (ridge) regularization.

Importance of regularization

It can be shown that:

$$MLE \approx ERM$$

$$MAP \approx \text{reg } ERM$$

Support vector machine(SVM) when data is linearly separable

Dataset is linearly separable

Dataset $D = \{(x_i, y_i)\}_{i=1}^{n}$ where $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$ is said to be linearly separable if there exists a hyperplane that can separate the two classes of data points with zero training error.

Mathematically, a dataset is linearly separable if there exist weights w and bias b such that:

$$w^T x_i + b > 1 \text{ for } y_i = 1$$

$$w^T x_i + b < -1 \text{ for } y_i = -1$$

This can be rewritten as:

$$y_i(w^T x_i + b) \geq 1 \text{ for all } i$$

In other words, there does not exist any x_i such that $-1 \leq w^T x_i + b \leq 1$.

The distance between the margins is $\frac{2}{\|w\|}$.

Optimization problem

Hence the SVM optimization problem is:

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

$$\text{s.t. } y_i(w^T x_i + b) \geq 1 \text{ for all } i$$

Solution to the optimization problem

We can solve this optimization problem using the method of Lagrange multipliers.

The Lagrangian for the SVM optimization problem can be formulated as follows:

$$L(w, b, \mu) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \mu_i \left(y_i(w^T x_i + b) - 1\right)$$

where $\mu_i \geq 0$ are the Lagrange multipliers.

To find the optimal solution, we take the partial derivatives of the Lagrangian with respect to w, b, and μ_i, and set them to zero:

1. **Gradient with respect to (w)**:

$$\frac{\partial L}{\partial w} = w - \sum_{i=1}^{n} \mu_i y_i x_i = 0$$

2. **Gradient with respect to (b)**:

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{n} \mu_i y_i = 0$$

3. **Complementary slackness condition**:

$$\mu_i(y_i(w^T x_i + b) - 1) = 0$$

The solution can be obtained by solving these equations, which leads to the optimal weights w and bias b that define the separating hyperplane.

From the first equation, we can express w in terms of μ_i:

$$w = \sum_{i=1}^{n} \mu_i y_i x_i$$

The second equation gives us a constraint on μ_i:

$$\sum_{i=1}^{n} \mu_i y_i = 0$$

Substituting these back into the Lagrangian and simplifying, we get the dual formulation:

Dual Formulation

The dual formulation of the SVM problem can be expressed as:

$$\max_{\mu} \sum_{i=1}^{n} \mu_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \mu_i \mu_j y_i y_j (x_i^T x_j)$$

subject to:

$$\sum_{i=1}^{n} \mu_i y_i = 0$$

$$\mu_i \geq 0 \quad \text{for all } i$$

Finding w

The dual problem can be solved using quadratic programming techniques. Once we have the optimal μ_i, we can recover w using the equation:

$$w = \sum_{i=1}^{n} \mu_i y_i x_i$$

Finding b

To find b, we can use any support vector (a point where $\mu_i > 0$) and the fact that for these points, $y_i(w^T x_i + b) = 1$.

Decision Function

The decision function for classifying new points becomes:

$$f(x) = \text{sign}\left(\sum_{i=1}^{n} \mu_i y_i (x_i^T x) + b\right)$$

where only the support vectors (points with $\mu_i > 0$) contribute to the sum.

Support vector machine(SVM) when data is not linearly separable

Dataset is not linearly separable

Unlike the case when the dataset is linearly separable, we cannot find a hyperplane that separates the two classes of data points with zero training error.

Soft margin

To handle this case, we introduce a slack variable ξ_i for each data point x_i to allow some points to be on the wrong side of the margin or even in the wrong class.

Optimization problem

The optimization problem becomes:

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i$$

$$\text{s.t. } y_i(w^T x_i + b) \geq 1 - \xi_i \text{ for all } i$$

$$\xi_i \geq 0 \text{ for all } i$$

Solution to the optimization problem

The Lagrangian for the SVM optimization problem can be formulated as follows:

$$L(w, b, \xi, \mu, \nu) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \mu_i \left(y_i(w^T x_i + b) - 1 + \xi_i\right) - \sum_{i=1}^{n} \nu_i \xi_i$$

where $\mu_i \geq 0$ and $\nu_i \geq 0$ are the Lagrange multipliers.

To find the optimal solution, we take the partial derivatives of the Lagrangian with respect to w, b, ξ_i, μ_i, and ν_i, and set them to zero:

$$\frac{\partial L}{\partial w} = 0 \implies w = \sum_{i=1}^{n} \mu_i y_i x_i$$

$$\frac{\partial L}{\partial b} = 0 \implies \sum_{i=1}^{n} \mu_i y_i = 0$$

$$\frac{\partial L}{\partial \xi_i} = 0 \implies C - \mu_i - \nu_i = 0$$

$$\frac{\partial L}{\partial \mu_i} = 0 \implies y_i(w^T x_i + b) - 1 + \xi_i \geq 0$$

$$\frac{\partial L}{\partial \nu_i} = 0 \implies \xi_i \geq 0$$

The complementary slackness conditions are:

$$\mu_i(y_i(w^T x_i + b) - 1 + \xi_i) = 0$$

$$\nu_i \xi_i = 0$$

From these conditions, we can deduce:

1. If $0 < \mu_i < C$, then $\xi_i = 0$ and $y_i(w^T x_i + b) = 1$
2. If $\mu_i = 0$, then $y_i(w^T x_i + b) \geq 1$
3. If $\mu_i = C$, then $y_i(w^T x_i + b) \leq 1$

Dual Formulation

Substituting these back into the Lagrangian and simplifying, we get the dual formulation:

$$\max_{\mu} \sum_{i=1}^{n} \mu_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \mu_i \mu_j y_i y_j (x_i^T x_j)$$

subject to:

$$\sum_{i=1}^{n} \mu_i y_i = 0$$

$$0 \leq \mu_i \leq C \quad \text{for all } i$$

Finding w and b

Once we solve the dual problem and obtain the optimal μ_i, we can find w and b:

$$w = \sum_{i=1}^{n} \mu_i y_i x_i$$

To find b, we can use any support vector (a point where $0 < \mu_i < C$) and the fact that for these points, $y_i(w^T x_i + b) = 1$.

Decision Function

The decision function for classifying new points remains the same as in the linearly separable case:

$$f(x) = \text{sign}\left(\sum_{i=1}^{n} \mu_i y_i (x_i^T x) + b\right)$$

where only the support vectors (points with $\mu_i > 0$) contribute to the sum.

The main difference from the linearly separable case is the upper bound C on the Lagrange multipliers μ_i, which allows for some misclassifications in the training set while still finding the optimal separating hyperplane.

SVM with kernel

Prerequisites - Kernel trick

Kernel function $k(x, y) = \phi(x)^T \phi(y)$ where $\phi(x)$ is a feature mapping that maps the data points to a higher dimensional space without actually computing the feature mapping, i.e., $\phi(x)$.

In other words, we can compute the kernel function $k(x, y)$ without actually computing the feature mapping $\phi(x)$.

Examples of kernel functions:

- Polynomial kernel: $k(x_1, x_2) = \phi(x_1)^T \phi(x_2) = (1 + x_1^T x_2)^p$
- Sigmoid kernel: $k_s(x_1, x_2) = \frac{1}{1+\exp(a x_1^T x_2)}$
- RBF/Gaussian kernel: $k(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|^2}{\sigma^2}\right)$

Motivation

If the dataset $D = \{(x_i, y_i)\}_{i=1}^n$ where $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$ is not linearly separable in the original d-dimensional space, we can use a feature mapping $\phi(x)$ to map the data points to a higher dimensional space where they become linearly separable and then use the SVM optimization problem to find the optimal hyperplane.

Hence the new dataset $D' = \{(x_i', y_i)\}_{i=1}^n$ where $x_i' = \phi(x_i)$ and $y_i \in \{-1, 1\}$.

Optimization problem

The optimization problem for SVM with kernel can be formulated as:

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^n \xi_i$$

subject to:

$$y_i(w^T\phi(x_i) + b) \geq 1 - \xi_i, \quad i = 1, 2, 3, ..., n$$

$$\xi_i \geq 0$$

Solution to the optimization problem

The Lagrangian for the SVM optimization problem with kernel can be formulated as follows:

$$L(w, b, \xi, \alpha, \beta) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i\left(y_i(w^T\phi(x_i) + b) - 1 + \xi_i\right) - \sum_{i=1}^{n}\beta_i\xi_i$$

where $\alpha_i \geq 0$ and $\beta_i \geq 0$ are the Lagrange multipliers.

To find the optimal solution, we take the partial derivatives of the Lagrangian with respect to w, b, ξ_i, α_i, and β_i, and set them to zero:

$$\frac{\partial L}{\partial w} = 0 \implies w = \sum_{i=1}^{n}\alpha_i y_i \phi(x_i)$$

$$\frac{\partial L}{\partial b} = 0 \implies \sum_{i=1}^{n}\alpha_i y_i = 0$$

$$\frac{\partial L}{\partial \xi_i} = 0 \implies C - \alpha_i - \beta_i = 0$$

$$\frac{\partial L}{\partial \alpha_i} = 0 \implies y_i(w^T\phi(x_i) + b) - 1 + \xi_i \geq 0$$

$$\frac{\partial L}{\partial \beta_i} = 0 \implies \xi_i \geq 0$$

The complementary slackness conditions are:

$$\alpha_i(y_i(w^T\phi(x_i) + b) - 1 + \xi_i) = 0$$

$$\beta_i\xi_i = 0$$

From these conditions, we can deduce:

1. If $0 < \alpha_i < C$, then $\xi_i = 0$ and $y_i(w^T\phi(x_i) + b) = 1$

2. If $\alpha_i = 0$, then $y_i(w^T\phi(x_i) + b) \geq 1$

3. If $\alpha_i = C$, then $y_i(w^T\phi(x_i) + b) \leq 1$

Dual Formulation

Substituting these back into the Lagrangian and simplifying, we get the dual formulation:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j k(x_i, x_j)$$

subject to:

$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

$$0 \leq \alpha_i \leq C \quad \text{for all } i$$

where $k(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel function.

Finding w

Once we solve the dual problem and obtain the optimal α_i, we can find w and b:

$$w = \sum_{i=1}^{n} \alpha_i y_i \phi(x_i)$$

but we don't need to compute $\phi(x_i)$ explicitly. Hence we can use the kernel function $k(x_i, x)$ to compute the decision function:

$$w^T\phi(x) = \left(\sum_{i=1}^{n} \alpha_i y_i \phi(x_i) \right)^T \phi(x) = \sum_{i=1}^{n} \alpha_i y_i \phi(x_i)^T \phi(x) = \sum_{i=1}^{n} \alpha_i y_i k(x_i, x)$$

This is often referred to as the "kernel trick", which allows us to work in high-dimensional feature spaces without explicitly computing the feature vectors.

Finding b

To find b, we can use any support vector (a point where $0 < \alpha_i < C$) and the fact that for these points, $y_i(w^T \phi(x_i) + b) = 1$.

Decision Function

The decision function for classifying new points becomes:

$$f(x) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i k(x_i, x) + b\right)$$

where only the support vectors (points with $\alpha_i > 0$) contribute to the sum.

The final classifier is:

$$f(x) = \begin{cases} 1 & \text{if } f(x) > 0 \\ -1 & \text{if } f(x) < 0 \end{cases}$$

The main difference from the non-kernel SVM is the use of the kernel function $k(x_i, x)$ instead of the dot product $x_i^T x$. This allows the SVM to find non-linear decision boundaries in the original input space by implicitly working in a higher-dimensional feature space.

Support measure machine (SCM) as Empirical Risk Minimization (ERM)

Recall that in support measure machine (SCM), we were trying to maximize the margin between the two classes of data points.

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i$$

$$\text{s.t. } y_i(w^T x_i + b) \geq 1 - \xi_i \text{ for all } i$$

$$\xi_i \geq 0 \text{ for all } i$$

SCM as ERM

This is an optimization problem that can be formulated as a empirical risk minimization (ERM) problem.

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

This is an ERM problem with:

- the loss function $l(y_i, f(x_i)) = \max(0, 1 - y_i f(x_i))$, is known as the hinge loss.
- the regularization term $\frac{1}{2}w^T w$.

Neural network

A neural network is a computational model inspired by the structure and function of biological neural networks. Mathematically, it can be defined as a series of function compositions:

$$f(x) = f_L(f_{L-1}(...f_2(f_1(x))))$$

where L is the number of layers in the network, and each function f_i represents a layer operation.

For a single layer, the operation can be expressed as:

$$f_i(x) = \sigma(W_i x + b_i)$$

where:

- W_i is the weight matrix for layer i
- b_i is the bias vector for layer i
- σ is a non-linear activation function

The complete neural network can then be written as:

$$f(x) = \sigma_L(W_L\sigma_{L-1}(W_{L-1}...\sigma_2(W_2\sigma_1(W_1 x + b_1) + b_2)... + b_{L-1}) + b_L)$$

This formulation allows the network to learn complex, non-linear mappings from inputs to outputs through the composition of simpler functions and the application of non-linear activations.

Non-linear activation functions σ

- Sigmoid or logistic:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Sign function:

$$\text{sign}(x)$$

- Hyperbolic tangent:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Rectified Linear Unit (ReLU):

$$\text{ReLU}(x) = \max(0, x)$$

Why do we need non-linear activation functions?

- Without non-linear activation functions, the neural network would be equivalent to a linear model. Let's derive this:

Consider a neural network with two layers and no activation function:

1. First layer: $y = W_1 x + b_1$
2. Second layer: $z = W_2 y + b_2$

Substituting y into the second layer:

$$z = W_2(W_1 x + b_1) + b_2$$

$$z = W_2 W_1 x + W_2 b_1 + b_2$$

This can be simplified to:

$$z = W x + b$$

Where:

$$W = W_2 W_1$$

$$b = W_2 b_1 + b_2$$

This is the equation of a linear model. Therefore, without non-linear activation functions, regardless of the number of layers, a neural network will always produce a linear transformation of the input.

- Non-linear activation functions introduce non-linearity into the network, allowing it to learn and represent complex, non-linear relationships in the data.

Backpropagation

Notations

- L: number of layers in the network.
- w_{jk}^l: weight connecting k^{th} neuron of $(l-1)^{th}$ layer to j^{th} neuron of l^{th} layer
- b_j^l: bias of j^{th} neuron in l^{th} layer
- a_j^l: output of j^{th} neuron in l^{th} layer
- z_j^l: preactivation output of j^{th} neuron in l^{th} layer

Here:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

Also:

$$a_j^l = \sigma(z_j^l)$$

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l)$$

Derivation

Let's assume we use squared loss function:

$$L = \frac{1}{2}\|a^L - y\|^2$$

We define the risk R as the expected loss over the data distribution:

$$R = \mathbb{E}[L]$$

Risk derivative with respect to the output layer:

$$\frac{\partial R}{\partial a_j^L} = \frac{\partial}{\partial a_j^L}\mathbb{E}[\frac{1}{2}\|a^L - y\|^2] = \mathbb{E}[(a_j^L - y_j)]$$

Let $\delta_j^L = \frac{\partial R}{\partial z_j^L}$ then:

$$\delta_j^L = \frac{\partial R}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} = \mathbb{E}[(a_j^L - y_j)] \cdot \sigma'(z_j^L)$$

Error term for the output layer (using element-wise product $\odot$):

$$\delta^L = \nabla_a R \odot \sigma'(z^L)$$

Error term for hidden layers:

$$\frac{\partial R}{\partial z_j^l} = \sum_k \frac{\partial R}{\partial z_k^{l+1}} \cdot \frac{\partial z_k^{l+1}}{\partial a_j^l} = \sum_k \delta_k^{l+1} \cdot w_{jk}^{l+1} \cdot \sigma'(z_j^l)$$

because there are k neurons in $l + 1^{th}$ layer.

This can be written in vector form as:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

Gradient of the risk with respect to weights:

$$\frac{\partial R}{\partial w_{jk}^l} = \frac{\partial R}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \cdot a_k^{l-1}$$

Gradient of the risk with respect to biases:

$$\frac{\partial R}{\partial b_j^l} = \frac{\partial R}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l$$

These derivatives form the basis of the backpropagation algorithm, allowing us to compute the gradients needed for updating the weights and biases in the neural network.

Miscellaneous Machine Learning Terms

Epoch

One complete pass of the entire training dataset for training the model.

Batch Size

Risk is defined as

$$L = \frac{1}{N} \sum_{i=1}^{N} l(y_i, \hat{y}_i)$$

Where $l(y_i, \hat{y}_i)$ is a general loss function that measures the discrepancy between the true value y_i and the predicted value $\hat{y}_i$ for each sample.

Instead of calculating the risk over the entire dataset, however calculating the risk over complete dataset is computationally expensive. Hence we calculate the risk over a small subset of the dataset, called a batch to perform back propogation.

Gradient Descent Variants

1. Batch Gradient Descent

- Full dataset: Computes the gradient of the risk function using the entire training dataset.
- Update frequency: Weights are updated after evaluating the entire dataset in one go.
- Efficiency: Can be slow for large datasets as it requires calculating gradients over the whole dataset before updating weights.
- Convergence: More stable gradient leads to smoother convergence.

Mathematically:

$$\theta = \theta - \eta \nabla_\theta R(\theta)$$

where θ are the model parameters, η is the learning rate, and $R(\theta)$ is the risk function.

2. Stochastic Gradient Descent (SGD)

- Single sample: Updates weights using one randomly chosen sample from the dataset at a time.
- Update frequency: Weights are updated after each individual sample, leading to more frequent updates.
- Efficiency: Faster and more efficient for large datasets as each update only requires computing the gradient for one sample.
- Convergence: Can have a noisier path to convergence but may help escape local minima due to randomness.

Mathematically:

$$\theta = \theta - \eta \nabla_\theta R(\theta; x^{(i)}, y^{(i)})$$

where $(x^{(i)}, y^{(i)})$ is a single training example.

3. Mini-batch Gradient Descent

- Batch of samples: Computes the gradient over a small batch of samples (between full dataset and single sample).
- Update frequency: Weights are updated after evaluating the risk on each mini-batch.
- Efficiency: Faster than batch gradient descent but less noisy than stochastic gradient descent.
- Convergence: Provides a balance between the efficiency of SGD and the stability of batch gradient descent.

Mathematically:

$$\theta = \theta - \eta \nabla_\theta R(\theta; x^{(i:i+n)}, y^{(i:i+n)})$$

where $(x^{(i:i+n)}, y^{(i:i+n)})$ represents a mini-batch of n training examples.

Batch Normalization

Normalizes the input layer by adjusting and scaling the activations.

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Where:

- x_i is the input
- μ_B is the mini-batch mean
- σ_B^2 is the mini-batch variance
- ϵ is a small constant added for numerical stability

Batch Normalization is typically implemented as a non-learnable layer in neural networks, with fixed parameters during inference. During inference, the layer uses the moving averages of mean and variance computed during training, rather than calculating batch statistics, to normalize the inputs.

Layer Normalization

Layer Normalization normalizes the inputs across the features for each sample in a batch, rather than across the batch for each feature.

Mathematically, for an input x with H features:

$$\mu = \frac{1}{H} \sum_{i=1}^{H} x_i$$

$$\sigma^2 = \frac{1}{H} \sum_{i=1}^{H} (x_i - \mu)^2$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

Where:

- μ is the mean of the features for a single sample
- σ^2 is the variance of the features for a single sample
- ϵ is a small constant for numerical stability
- γ and β are learnable parameters for scaling and shifting

Implementation:

1. Compute mean and variance across features for each sample
2. Normalize each feature using the computed statistics
3. Scale and shift the normalized values with learnable parameters

Unlike Batch Normalization, Layer Normalization's behavior is the same during training and inference, as it doesn't depend on batch statistics.

Dropout

- A regularization technique
- Also implemented as a layer in neural networks
- During training:
- Each neuron is "switched on" with probability p
- When doing backpropagation, remember if neuron was on or off and update weights accordingly
- The value of p is generally same across all neurons of same layer but can be different for different layers
- During inference/classification:

- All neurons are active
- Output of each neuron is multiplied by probability p to get the output

N-Fold Cross Validation

- A resampling technique used to evaluate machine learning models
- Dataset is split into N equal parts
- N-1 parts are used for training and 1 part is used for validation
- This process is repeated N times, with each part being used for validation once
- Finally, the performance of the model is evaluated by averaging the results from all N folds

Decision Trees

How to descion tree works

- At each node, a question is asked about the data that splits the data into two or more non-overlapping subsets.

$$\text{Question: } x_j \leq \theta$$

then the data is split into two subsets, one where $x_j \leq \theta$ and one where $x_j > \theta$.

- The process is repeated till we reach leaf node, that classifies the datapoint to a region of the feature space.
- We can do regression and classification of the test datapoint based on the trainig datapoints that lie in the region.

Growing a Decision Tree

Choose a data dimension j and a threshold θ to split the data that minimises a metric eg minimize gini impurity for classification or minimize mean squared error for regression. Do this recursively.

Gini Impurity

Gini Impurity is a measure of impurity or disorder in a set of data points. It's used to determine the quality of a split in a decision tree. The goal is to minimize the Gini Impurity when growing the tree.

The Gini Impurity is calculated as:

$$G(set) = \sum_{i=1}^{K} \sum_{j \neq i} p_i p_j$$

$$G(set) = 1 - \sum_{i=1}^{K} p_i^2$$

Where:

- K is the total number of classes
- p_i is the probability of picking a data point with class i(p_j also defined similarly) if you randomly choose from the set

$$p_i = \frac{n_i}{N}$$

Where:

- n_i is the number of datapoints of class i in the set
- N is the total number of datapoints in the set

To evaluate a potential binary split, we calculate the weighted average of the Gini Impurities for the resulting subsets:

$$G_{split} = \frac{n_{left}}{n} G_{left} + \frac{n_{right}}{n} G_{right}$$

Where:

- n_{left} and n_{right} are the number of instances in the left and right subsets
- n is the total number of instances
- G_{left} and G_{right} are the Gini Impurities of the left and right subsets

The split with the lowest G_{split} is chosen as the best split for that node.

Mean Squared Error

For regression tasks, Mean Squared Error (MSE) is commonly used as the splitting criterion. MSE measures the average squared difference between the predicted and actual values.

The Mean Squared Error for a set of data points is calculated as:

$$MSE(set) = \frac{1}{|set|} \sum_{i \in set} (y_i - \hat{y})^2$$

Where:

- $|set|$ is the number of data points in the set
- y_i is the actual value of the i-th data point
- $\hat{y}$ is the mean of the target values in the set, calculated as:

$$\hat{y} = \frac{1}{|set|} \sum_{i \in set} y_i$$

To evaluate a potential binary split, we calculate the weighted average of the MSEs for the resulting subsets:

$$MSE_{split} = \frac{n_{left}}{n} MSE_{left} + \frac{n_{right}}{n} MSE_{right}$$

Where:

- n_{left} and n_{right} are the number of instances in the left and right subsets
- n is the total number of instances
- MSE_{left} and MSE_{right} are the Mean Squared Errors of the left and right subsets

The split with the lowest MSE_{split} is chosen as the best split for that node in the regression tree.

Pruning a Decision Tree

Pruning is the process of removing branches from a decision tree to prevent overfitting. Overfitting occurs when the tree is too complex and fits the training data too closely, capturing noise and details that are specific to the training data rather than generalizing to new, unseen data.

Pruning helps to simplify the tree, making it more robust and reducing its complexity. This can lead to better generalization to new data.

There are two main types of pruning:

1. Pre-pruning (Early Stopping):

- This method stops the growth of the tree before it fully fits the training data.
- It uses stopping criteria such as:
- Maximum depth of the tree
- Minimum number of samples required to split an internal node
- Minimum number of samples required to be at a leaf node
- Pre-pruning is computationally efficient but may result in underfitting if the stopping criteria are too strict.

2. Post-pruning (Reduced Error Pruning):

- This method first grows a full tree and then removes branches that do not provide significant predictive power.
- The process typically involves:

1. Grow a full tree on the training data

2. For each node:

- Calculate the accuracy of the tree with and without the node
- If removing the node increases accuracy, prune it
- Post-pruning often results in better performance but is more computationally expensive than pre-pruning.

Ensemble learning

Ensemble learning is a machine learning technique that combines multiple models to improve the overall performance and robustness of the prediction. It is based on the idea that by aggregating the predictions of several models, we can achieve better results than any single model alone.

Bagging

- Let the original data be D.
- Create n new datasets by sampling with replacement from D.
- Each new dataset will have the same number of samples as the original dataset, but some samples will be repeated, and some will be excluded.
- Train a model on each of the new datasets.
- The final prediction is made by aggregating the predictions of all models:
- For classification, the final prediction is made by taking the majority vote of the predictions of all models.
- For regression, the final prediction is made by taking the average of the predictions of all models.

Random Forest - Bagged descion trees

- Bagging of decision trees.
- Each decision tree is trained on a random subset of the features.
- The final prediction is made by taking the majority vote of the predictions of all decision trees.

Boosting

In bagging, we train each model independently on a random subset of the data. In boosting, we train each model sequentially on the same data, with each subsequent model focusing on correcting the errors of combined previous model by increasing weights of misclassified datapoints

Unlike bagging, each model depends on the previous ones and its contribution to the final prediction is weighted differently.

Mathematical Formulation of Boosting

Let's define the ensemble model $H_T(x)$ as:

$$H_T(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$$

where:

- $h_t(x)$ is the t-th weak learner
- $\alpha_t \in [0, 1]$ is the weight of the t-th weak learner
- T is the total number of weak learners

The risk (or error) of the ensemble model H is defined as:

$$R(H) = \frac{1}{n} \sum_{i=1}^{n} L(H(x_i), y_i)$$

where:

- L is the loss function
- (x_i, y_i) are the input-output pairs in the dataset
- n is the number of samples

Our goal is to minimize $R(H)$ with respect to H_T. We do this by gradient descent over functions:

$$h_T = \arg\min_{h \in H} R(H_{T-1} + \alpha h)$$

Using Taylor expansion, this can be approximated as:

$$h_T = \arg\min_{h \in H} R(H_{T-1}) + \alpha \langle \nabla R(H_{T-1}), h \rangle$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product.

Because $R(H_{T-1})$ is fixed, we can ignore it:

$$h_T = \arg\min_{h \in H} \alpha \cdot \langle \nabla R(H_{T-1}), h \rangle$$

Expanding this further:

$$h_T = \arg\min_{h \in H} \sum_{i=1}^{n} \frac{\partial R(H_{T-1}(x_i))}{\partial H_{T-1}(x_i)} \cdot h(x_i)$$

We call $r_i = -\frac{\partial R(H_{T-1}(x_i))}{\partial H_{T-1}(x_i)}$ the pseudo-residuals.

Thus, the problem reduces to:

$$h_T = \arg\min_{h \in H} \sum_{i=1}^{n} r_i \cdot h(x_i)$$

This formulation shows that each new weak learner h_T is fit to the pseudo-residuals of the previous ensemble model, effectively focusing on the errors of the combined previous models.

XGBoost - Gradient boosted regression tree

In XGBoost, the weak learners are decision trees.

The algorithm builds these trees sequentially, with each new tree aiming to correct the errors of the combined previous trees.

1. Recall that $r_i = -\frac{\partial R(H_{T-1}(x_i))}{\partial H_{T-1}(x_i)}$ are the pseudo-residuals.

2. XGBoost formulates the optimization problem for finding the next weak learner as:

$$h_T = \arg\min_{h \in H} \sum_{i=1}^{n} (r_i \cdot h(x_i) + \frac{1}{2} h(x_i)^2)$$

3. We can rewrite this by defining $\hat{y}_i = -r_i$:

$$h_T = \arg\min_{h \in H} \sum_{i=1}^{n} (-\hat{y}_i \cdot h(x_i) + \frac{1}{2} h(x_i)^2)$$

where $\hat{y}_i = \frac{\partial R(H_{T-1}(x_i))}{\partial H_{T-1}(x_i)}$

4. This formulation is equivalent to:

$$h_T = \arg\min_{h \in H} \sum_{i=1}^{n} (h(x_i) - \hat{y}_i)^2$$

which is the standard squared error regression problem with new labels $\hat{y}_i$.

5. In practice, $\hat{y}_i$ is approximated as:

$$\hat{y}_i = H_{T-1}(x_i) - y_i$$

where y_i is the true label and $H_{T-1}(x_i)$ is the prediction of the ensemble up to the previous iteration. This approximation is derived from the first-order Taylor expansion of the gradient when using the squared error loss $L(y_i, H(x_i)) = \frac{1}{2}(y_i - H(x_i))^2$.

Algorithm

The XGBoost algorithm can be summarized in the following steps:

1. Initialize the model with a constant value:

$$H_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$$

2. For t = 1 to T (number of trees):

a. Update labels:

$$\hat{y}_i = H_{T-1}(x_i) - y_i$$

for $i = 1, ..., n$

b. Fit a regression tree to the updated labels, giving terminal regions R_j^t, $j = 1, ..., J_t$

c. For each terminal region R_j^t, compute:

$$\gamma_{jt} = \arg\min_{\gamma} \sum_{x_i \in R_{jt}} L(y_i, H_{t-1}(x_i) + \gamma)$$

This step computes the optimal value $\gamma_j t$ (the adjustment) for each leaf region R_j^t, minimizing the loss function L for observations in that region. This "boost" is added to the current model predictions in the next step.

d. Update the model:

$$H_t(x) = H_{t-1}(x) + \nu \sum_{j=1}^{J_t} \gamma_{jt} I(x \in R_{jt})$$

where ν is the learning rate (0 < ν ≤ 1) that controls how much of the new tree's contribution is added to the current model, γ_{jt} is the prediction adjustment for each terminal region R_{jt} of the tree, and $I(x \in R_{jt})$ is an indicator function that equals 1 if x belongs to region R_{jt} and 0 otherwise.

3. Output the final model:

$$H(x) = H_T(x) = \sum_{t=0}^{T} \nu \sum_{j=1}^{J_t} \gamma_{jt} I(x \in R_{jt})$$

The final model is the sum of all the contributions from the individual trees, with each tree's contribution scaled by the learning rate ν.

Adaboost

Recall

Recall from our discussion on boosting we defined the ensemble model $H_T(x)$ as:

$$H_T(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$$

the problem of finding the next weak learner reduces to:

$$h_T = \arg\min_{h \in H} \sum_{i=1}^{n} r_i \cdot h(x_i)$$

where

$$r_i = -\frac{\partial R(H_{T-1}(x_i))}{\partial H_{T-1}(x_i)}$$

Problem setting

Let the dataset be $D = \{(x_i, y_i)\}_{i=1}^{n}$ where $y_i \in \{-1, 1\}$.

The loss function is given by:

$$L(H(x_i), y_i) = \exp(-y_i H(x_i))$$

The risk of the ensemble model H is given by:

$$R(H) = \frac{1}{n} \sum_{i=1}^{n} \exp(-y_i H(x_i)).$$

We want to find the model H_T that minimizes this risk.

Solution

We know:

$$r_i = -\frac{\partial R(H_{T-1}(x_i))}{\partial H_{T-1}(x_i)}$$

Let's calculate the pseudo-residuals:

$$r_i = -\frac{\partial R(H_{T-1}(x_i))}{\partial H_{T-1}(x_i)} = \frac{\partial}{\partial H_{T-1}(x_i)} \left(\frac{1}{n} \sum_{j=1}^{n} \exp(-y_j H_{T-1}(x_j)) \right)$$

$$= \frac{1}{n} \frac{\partial}{\partial H_{T-1}(x_i)} \exp(-y_i H_{T-1}(x_i))$$

$$= \frac{1}{n} \exp(-y_i H_{T-1}(x_i))(-y_i)$$

$$= y_i \exp(-y_i H_{T-1}(x_i))$$

Now, let's define weights w_i for each data point:

$$w_i = \frac{\exp(-y_i H_{T-1}(x_i))}{\sum_{j=1}^{n} \exp(-y_j H_{T-1}(x_j))}$$

Note that $\sum_{i=1}^{n} w_i = 1$.

Using these weights, we can rewrite our optimization problem:

$$h_T = \arg\min_{h \in H} \sum_{i=1}^{n} r_i \cdot h(x_i)$$

$$= \arg\min_{h \in H} \sum_{i=1}^{n} y_i \exp(-y_i H_{T-1}(x_i)) \cdot h(x_i)$$

$$= \arg\min_{h \in H} -\sum_{i=1}^{n} w_i y_i h(x_i)$$

This is equivalent to:

$$h_T = \arg\max_{h \in H} \sum_{i=1}^{n} w_i y_i h(x_i)$$

The weak learner h_T is chosen to maximize the weighted correlation between its predictions and the true labels.

In practice, we take any h say MLP, weight the losses for each datapoint i by w_i and then optimize for the weights that minimize the weighted loss.

Finding step size

After finding h_T, we need to determine its weight α_T in the ensemble:

To find the step size α_{t+1}, we minimize the risk with respect to α:

$$\alpha_{t+1} = \arg\min_{\alpha} R(H_t + \alpha h)$$

$$\alpha_{t+1} = \arg\min_{\alpha} \sum_{i=1}^{n} e^{-y_i(H_t(x_i)+\alpha h(x_i))}$$

Differentiating with respect to α and setting to zero, we get:

$$\alpha_{t+1} = \frac{1}{2} \log \frac{1-\epsilon}{\epsilon}$$

where $\epsilon = \sum_{i:h(x_i)\neq y_i} w_i$ is the weighted error of h.

We then update the ensemble:

$$H_{t+1}(x) = H_t(x) + \alpha_{t+1} h(x)$$

And update the weights for the next iteration:

$$w_i^{(t+1)} = w_i^{(t)} e^{-\alpha_{t+1} y_i h(x_i)}$$

These weights are then normalized to sum to 1.

Algorithm

The AdaBoost algorithm can be summarized in the following steps:

1. Initialize the weights for each training example:

$$w_i^{(1)} = \frac{1}{n} \text{ for } i = 1, ..., n$$

2. For t = 1 to T (number of weak learners):

a. Train a weak learner h_t using the weighted training data:

- Choose any suitable model as the weak learner, such as a decision tree, MLP, or logistic regression.
- For each training example i, weight its contribution to the loss function by $w_i^{(t)}$.
- If using a decision tree:
- Modify the splitting criterion to account for sample weights.
- When calculating impurity measures (e.g., Gini index or entropy), use weighted sums.
- If using an MLP or logistic regression:
- Modify the loss function to include sample weights. For example, with binary cross-entropy loss:

$$L = -\sum_{i=1}^{n} w_i^{(t)} [y_i \log(h_t(x_i)) + (1 - y_i) \log(1 - h_t(x_i))]$$

- Optimize the parameters of h_t to minimize this weighted loss function.
- The resulting h_t will focus more on correctly classifying examples with higher weights.

b. Calculate the weighted error of h_t:

$$\epsilon_t = \sum_{i:h_t(x_i) \neq y_i} w_i^{(t)}$$

c. Compute the weight of the weak learner:

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

d. Update the ensemble:

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

e. Update the weights for the next iteration:

$$w_i^{(t+1)} = w_i^{(t)} \exp(-\alpha_t y_i h_t(x_i))$$

f. Normalize the weights:

$$w_i^{(t+1)} = \frac{w_i^{(t+1)}}{\sum_{j=1}^{n} w_j^{(t+1)}}$$

3. Output the final ensemble:

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

This algorithm iteratively builds an ensemble of weak learners, each time focusing more on the examples that were misclassified in previous iterations. The final classifier is a weighted combination of all the weak learners, where the weights are determined by the performance of each weak learner.

Unsupervised learning

In unsupervised learning, we don't have labeled data. We want to find structure in the data.

Hence the data is of the form $X = \{x_1, ..., x_N\}$ where $x_i \in \mathbb{R}^d$.