Machine Learning

Probability and Statistics

The set of all possible outcomes, denoted Ω. The collection of all sets of outcomes (events), denoted A. And a probability measure P. Specification of the triple (Ω,A, P) defines the probability space which models a real-world measurement or experimental process.

Two event are **mutually exclusive**, if $A \wedge B = \phi$

Conditional Probability: Given two events, A and B, the probability that A will occur given that B has occurred.

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}, P(B) \neq 0$$

Two events are said to be independent if $P(A|B) = P(A)$. Equivalently, A and B are independent if $P(A \wedge B) = P(A) \cdot P(B)$

Bayes' Rule

$$P(A|B) = P\frac{(A \wedge B)}{P(B)}$$

$$P(B|A) = P\frac{(A \wedge B)}{P(A)}$$

$$\Rightarrow P(B|A) = \frac{P(A|B) \cdot P(B)}{P(A)}$$

A **random variable** X is a mapping of Ω to the real or complex numbers.

Continuous Random Variable: Uniform, Gaussian.

Discrete Random Variable: Binomial, Poisson, Bernoulli.

Expectation

$$E[X] = \int_{-\infty}^{\infty} x f_X(x) dx \qquad \text{or} \qquad E[X] = \sum_j x_j P_X(x_j)$$

$E[E[X|Y]] = E[X]$

Multivariate Normal Distribtion

$$X \sim N(\mu, \sigma)$$

$$P_X(x) = \frac{1}{(2\pi)^{(n/2)}|R|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)R^{-1}(x-\mu)\right)$$

Statistical Learning Framework

Data

Domain set $\mathcal{X}$

Label set $\mathcal{Y}$

A joint probability distribution $\mathcal{P}$ on of training data $\mathcal{X} \times \mathcal{Y}$, which consists of n points (x_1, y_1), $(x_2, y_2), \ldots, (x_n, y_n) \overset{iid}{\sim} P$

Model

A prediction function $f : \mathcal{X} \rightarrow \mathcal{Y}$

Measure of success

A loss function $\ell : \mathcal{Y} \times \mathcal{Y} \rightarrow \Re$

Given the prediction function *f*, and the loss function ℓ, the loss of an example is $\ell(f(x), y)$
Training error/ empirical error of

$$\hat{L}(f) \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^{n} l(f(x_i), y_i)$$

Generalization error/ risk/ true error

$$L(f) \stackrel{\text{def}}{=} E_{(x,y)\sim P}[l(f(x), y)] = \int l(f(x), y)\, p(x, y)\, dx\, dy$$

Bayes Classifier

Notations:
X feature space Usually $\Re^n$
Y Set of class labels
$\mathbf{X} = (X_1, X_2, \ldots X_n)^T$ feature vector
Classifier h : $\mathsf{X} \rightarrow \mathsf{Y}$
f_0 f_1 – Class Conditional Densities
p_0, p_1 – Prior probabilities
q_0, q_1 – Posterior probabilities
The Bayes Classifier

$$h_B(\boldsymbol{x}) = 0 \quad if\ q_0(\boldsymbol{x}) > q_1(\boldsymbol{x})$$
$$= 1 \quad otherwisw$$

Optimality
Let $R_i(h) = \{x \in X : h(x) = i\}, i = 0, 1.$ That is $R_0(h)$ is the set of all feature vectors that get classified as Class-0 by the classifier h.
Let F(h) denote the probability of error for h

$$\begin{aligned} F(h) &= P[\boldsymbol{X} \in R_1(h), \boldsymbol{X} \in C_0] + P[\boldsymbol{X} \in R_0(h), \boldsymbol{X} \in C_1] \\ &= P[X \in C_0] P[\boldsymbol{X} \in R_1(h) | \boldsymbol{X} \in C_0] + P[X \in C_1] P[\boldsymbol{X} \in R_0(h) | \boldsymbol{X} \in C_1] \\ &= p_0 P[\boldsymbol{X} \in R_1(h) | \boldsymbol{X} \in C_0] + p_1 P[\boldsymbol{X} \in R_0(h) | \boldsymbol{X} \in C_1] \\ &= p_0 \int_{R_1(h)} f_0(\boldsymbol{x})\, d\boldsymbol{x} + p_1 \int_{R_0(h)} f_1(\boldsymbol{x})\, d\boldsymbol{x} \end{aligned}$$

Bayes Error :

$$\begin{aligned} F(h_B) &= \int_{R_1(h_B)} p_0 f_0(\boldsymbol{x})\, d\boldsymbol{x} + \int_{R_0(h_B)} p_1 f_1(\boldsymbol{x})\, d\boldsymbol{x} \\ &= \int_X min(p_0 f_0(\boldsymbol{x}), p_1 f_1(\boldsymbol{x}))\, d\boldsymbol{x} \end{aligned}$$

For a Bayes classifier with 'M' classes and arbitrary loss function, we have $h_B(X) = \alpha_i$, if

$$\sum_{j-0}^{M-1} L(\alpha_i, C_j) q_j(\boldsymbol{X}) \leq \sum_{j=0}^{M-1} L(\alpha_k, C_j) q_j(\boldsymbol{X}) \quad \forall k$$

To implementing Bayes Classifier , we need class conditional densities and prior probabilities.(Prior can be calculated as fraction of examples)
To estimate densities , there are two main approaches :
1) **Parametric** : We assume we have iid realizations of a random variable X whose distribution is known except for values of a parameter vector. We estimate the parameters of the density using the samples available. [Maximum likelihood, Bayesian Estimation]
2)**Non-parametric**: We do not assume form of density. It is often model as a convex combination of some densities using the samples. [Parzen window]

Maximum Likelihood Estimation

Define Likelihood function by, $L(\theta|\boldsymbol{x})=\prod_{j=1}^{n} f(x_j|\theta)$ where x_j are known values (as given by data).

For convenience we often take log likelihood $l(\theta|\boldsymbol{x})=\log L(\theta|\boldsymbol{x})=\sum_{j=1}^{n} \log f(x_j|\theta)$. ML estimate is the maximizer of log likelihood or likelihood function.

$$\hat{\theta}_n = arg\,max_\theta \sum_{i=1}^{n} \log\left(\log f(x_i|\theta)\right)$$

MLEs are a very important type of estimator for the following reasons:

- The MLE is often simple and easy to compute.
- MLEs are invariant under reparameterization.
- MLEs often have asymptotic optimal properties (e.g. consistency $MSE \to 0\ as\ n \to \infty$)
- MLE occurs naturally in composite hypothesis testing.

Maximum Likelihood can be thought as minimizing KL divergence.

$$KL(f_{data}\|f_\theta) = -\sum_{i=1}^{n} f_{data}(x_i)\ln\left(\frac{f_\theta(x_i)}{f_{data}(x_i)}\right)$$

$$= -\sum_{i=1}^{n} f_{data}(x_i)\ln\left(f_\theta(x_i)\right) + const$$

Since data is all we have, we can say that $f_{data}(x_i) = 1/n$, So Minimizing KL divergence as same as maximizing $\sum_{i=1}^{n} \frac{1}{n}\ln\left(f_\theta(x_i)\right)$ that is the ML estimate.

Theorem: In-variance of the MLE

Let $\hat{\theta}$ denote the MLE of θ, Then $\hat{g}=G(\hat{\theta})$ is the MLE of $g=G(\theta)$

Mixture density Estimate and EM algorithm

Density Model

$$f(x)=\sum_{k=1}^{K} \lambda_k f_k(x), \quad \lambda_k \geq 0, \quad \sum_{k=1}^{K} \lambda_k = 1$$

The log likelihood is $l(\theta|D)=\sum_{i=1}^{n} \ln\left[\sum_{k=1}^{K} \lambda_k f_k(x_i)\right]$

There is sum inside log, so maximizing log likelihood becomes a difficult optimization algorithm.

EM Algorithm Let Z_{ij} , i = 1, 2, … n ; j =1, 2, … K denote information of which component density each sample come from. Z_{ij} = 1 if x_i came from j^{th} component density.

$$P(Z_{ij} = 1) = \lambda_j \qquad f(x_i \mid Z_{ij} = 1) = \phi(x_i \mid \theta_j)$$

Z_{ij} is the missing information. Z_i denote vector with component Z_{ij}

Denote $D^c = \{(x_1,Z_1), \ldots (x_n,Z_n)\}$ (complete data) but we only have x_i, i =1, 2, … n (incomplete data).

The complete data log likelihood is :

$$l(\theta|D^c)=\ln\left(\prod_{i=1}^{n} f(x_i,Z_i|\theta)\right)$$

denoted by $\ln(f(\mathbf{x},\mathbf{Z}|\theta)$

Two steps of EM algorithm are as follows:

E-step Compute $Q(\theta, \theta^{(k)})$ which is expectation of the complete data log likelihood w.r.t. the conditional distribution of hidden variables conditioned on incomplete data and current value of θ as $\theta^{(k)}$.

$$Q(\theta,\theta^{(k)}) = E_{(\mathbf{Z}|\mathbf{x},\theta^{(k)})} \ln(f(\boldsymbol{x},\boldsymbol{Z}|\theta)) = \int \ln(f(\boldsymbol{x},\boldsymbol{Z}|\theta)) f(\boldsymbol{z}|\boldsymbol{X},\theta^{(k)})\, dz$$

M-Step Compute next value θ as $\theta^{(k+1)}$ by maximizing $Q(\theta, \theta^{(k)})$ over θ.

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

Bayesian Estimation

In Bayesian Estimation we think parameter itself as random variable. We have some knowledge (subjective beliefs) about this which we called prior density and then data transforms our prior density into posterior density. Using Bayes Theorem , we have

$$f(\theta|D) = \frac{f(D|\theta) f(\theta)}{\int f(D|\theta) f(\theta)\, d\theta}$$

Here we have estimate of parameter as density , rather than a single output. So we can either take the maximum aposteriori probability(MAP) estimate i.e. $arg\,max_\theta\, f(\theta|D)$ or we can take the expected value of the resultant density i.e. $\int \theta f(\theta|D)\, d\theta$

The form of prior density that results in the same form of density for the posterior(or that of data likelihood) is called **conjugate prior**.

Parzen Window

Define

$$\phi(u) = 1 \quad if\ |u_i| \le 0.5, i = 1,2,\ldots d$$
$$= 0 \quad otherwise$$

This defines a unit hypercube in $\Re^d$ centered at origin. $\phi\left(\frac{u-u_0}{h}\right)$ be hypercube of length h centered at u_0. The number of data points falling in a hypercube of side h centered at x is

$$k = \sum_{i=1}^{n} \phi\left(\frac{x-x_i}{h}\right)$$

So we can write our density estimate as $\hat{f}(x) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{h^d}\phi\left(\frac{x-x_i}{h}\right)$

Now instead of a hypercube, we can use multivariate Gaussian (known as kernel density estimate), so our density estimate will be :

$$\hat{f}(x) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{(h\sqrt{2\pi})^d} \exp\left[\frac{-\|x-x_i\|^2}{2h^2}\right]$$

This Gaussian kernel gives a smoother density estimate.

For implementing Bayes, we need class conditional densities. Class conditional densities can be estimated,using parametrically or non-parametrically. Bayes classifier is optimal when we exactly know the posterior probabilities. When we estimate densities, there would be inaccuracies. This results in the non-optimality of implemented Bayes classifier. In general, it is not easy to relate estimation errors to classification errors. (This is simple approach for generative models, one can also look for other generative models or Graphical models)

Now we look at look at Discriminative models:

Linear Classifiers/ Regression

In a regression/Classification problem, the training set is $\{(X_i, y_i), i = 1,\ldots,n\}$ with $X_i \in \Re^d$ and $y_i \in \Re\ , \forall i$ The goal is to learn a function, $f: \Re^d \rightarrow \Re$, that captures the relationship between X and y.

$$f(X)=\sum_{j=1}^{d} w_i x_i + w_0$$

We can assume X to be augmented i.e. X=(1, x_1, x_2, … ,x_n) similarly assuming W=(w_0, w_1, …, w_n)

$$f(X)=W^T X$$

Linear Least Squares Regression

We want to find a W such that $\hat{y}(X) = f(X) = W^T X$ is a good fit for the training data. Define a function $J:\Re^{d+1}\rightarrow\Re$ by

$$J(W)=\frac{1}{2}\sum_{j=1}^{n}(X_i^T W-y_i)^2$$

We can assume A be our data matrix $A\in\Re^{n\times d+1}$ whose i^{th} row is X_i and $Y\in\Re^n$ whose i^{th} element is y_i. Hence we have

$$J(W)=\frac{1}{2}\sum_{j=1}^{n}(X_i^T W-y_i)^2=\frac{1}{2}(AW-Y)^T(AW-Y)$$

Simple Solution to above problem is $W = (A^TA)^{-1}A^TY = A^{\dagger}Y$ (If A^TA is invertible). Even if A^TA is not invertible one can solve for $(A^TA)W = A^TY$ (Using SVD or Eigen value decompistion)

Statistical viewpoint of Linear Regression

The least square error criterion is same as minimizing:

$$J(W)=\frac{1}{2}\frac{1}{n}\sum_{j=1}^{n}(X_i^T W-y_i)^2$$

Assuming the training examples to be drawn iid, the above is a good approximation of

$$J(W)=\frac{1}{2}E[(X^T W-Y)^2]$$

Equating the gradient of J(W) to zero we get

$$E[X(X^T W-y)]=0\Rightarrow E[XX^T]W=E[Xy]$$

This gives us the optimal W^* as

$$W^*=(E[XX^T])^{-1}(E[Xy])$$

The earlier expression we have for W^*, would be same as this if we approximate expectations by sample averages.

Since rows of A are X^i , we have

$$A^T A=\sum_{i=1}^{n} X_i X_i^T\approx nE[XX^T]$$

Similarly $A^T Y=\sum_{i=1}^{n} X_i y_i\approx nE[Xy]$

Thus we have

$$(A^T A)^{-1}A^T Y\approx(nE[XX^T])^{-1}(nE[Xy])$$

The problem of approximating a random variable y as a function of another random variable X in the sense of least mean square error:

$$min\ E[(f(X)-y)^2]$$

If f^* is the optimal function, then $f^*(X)$ is called the regression function of y on X. One can show that this f^* is given by $f^*(X) = E[y|X]$.

In a 2-class classification problem, suppose we learnt W to minimize

$$J(W)=\frac{1}{2}\sum_{i=1}^{n}(X_i^T W-y_i)^2$$

If we had $y \in \{0, 1\}$, then we learn a best linear approximation to the posterior probability, $q_1(X)$. Linear least squares is learning the conditional distribution of y given X. If we had $y \in \{-1, 1\}$ then we learn a good linear approximation to $2q_1(X)-1$.
Suppose X, y are related by $y = W^T X + \xi$ where ξ is a zero mean noise. Then we expect linear least squares method to easily learn W. The final mean sqare error would be variance of ξ. Using this idea, we can think of the least squares method as an ML estimation procedure under a reasonable probability model. We take the probabilty model for y as

$$f(y|X, W, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\frac{(y - W^T X)^2}{\sigma^2}\right]$$

where W and σ are the parameters. Let $D=\{y_1(X_1),\dots, y_n(X_n)\}$ be the iid data. We want to derive the ML estimate for the parameters.
Data Likelihood:

$$L(W, \sigma|D) = \prod_{i=1}^{n} f(y_i|X_i, W, \sigma)$$

$$L(W, \sigma|D) = \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(y_i - X_i^T W)^2}{\sigma^2}\right)$$

Log Likelihood:

$$l(W, \sigma|D) = n \ln\frac{1}{\sigma\sqrt{2\pi}} - \frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(y_i - X_i^T W\right)^2$$

Maximizing log-likelihood is same as minimizing squares of errors. Thus linear least squares is the ML estimate (for the discriminative model) under the assumption of Gaussian noise

Logistic Regression

We know that if we want to predict 'y' as a function of 'X' to minimize $E[(f(X)-y)^2]$, then the optimal function is $f^*(X) = E[y \mid X]$. Suppose $y \in \{0, 1\}$. Then $f^*(X) = E[y|X] = P[y=1|X] = q_1(X)$. We can ask what would be a reasonable model for posterior probability.
By Bayes rule

$$\begin{aligned} q_0(X) &= \frac{f_0(X)p_0}{f_0(X)p_0 + f_1(X)p_1} \\ &= \frac{1}{1 + \frac{f_1(X)p_1}{f_0(X)p_0}} \\ &= \frac{1}{1+\exp(-\xi)}, \quad \textit{where} \\ \xi &= -\ln\left(\frac{f_1(X)p_1}{f_0(X)p_0}\right) = \ln\left(\frac{f_0(X)p_0}{f_1(X)p_1}\right) \end{aligned}$$

The logistic function is a good model for posterior probability.
We want to learn a probability model:

$$f_{(y|X)}(1|X) = P[y=1|X] = \frac{1}{1+\exp(-W^T X)}$$

Ridge Regression

In linear least squares regression, we often choose regularization term $\Omega(W) = \frac{1}{2}\|W\|^2$, called Tikhanov regularization.

Now the criterion is

$$J(W)=\frac{1}{2}\sum_{i=1}^{n}(W^T X_i-y_i)^2+\frac{\lambda}{2}W^T W$$
$$=\frac{1}{2}(AW-Y)^T(AW-Y)+\frac{\lambda}{2}W^T W$$

Equating the gradient of J to zero, we get $A^T(AW-Y)+\lambda W=0$. This gives us

$$(A^T A+\lambda I)W=A^T Y \;\Rightarrow\; W=(A^T A+\lambda I)^{-1}A^T Y$$

($A^TA + \lambda I$) would have all eigen values above λ and hence is always invertible. So, regularization helps the condition number of the linear equations and thus the learning is more 'stable'.

Another way to look at regularized least squares is from a Bayesian framework. As earlier, take the probability model as

$$f(y|X,W,\sigma)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{2}\frac{(y-W^T X)^2}{\sigma^2}\right)$$

We take the prior density of W as $f(W)=\left(\frac{1}{\alpha\sqrt{2\pi}}\right)^d\exp\left(\frac{-W^T W}{2\alpha^2}\right)$

Now the posterior density is given by

$$f(W|Y)\propto\prod_{i=1}^{n}f(y_i|X_i,W,\sigma)f(W)$$
$$\propto\exp\left(-\sum_{i=1}^{n}\frac{(y_i-W^T X_i)^2}{2\sigma^2}-\frac{1}{2\alpha^2}W^T W\right)$$

To find the MAP estimate we need to maximize the posterior density (or og of posterior)

$$\ln(f(W|Y))=-\frac{1}{2}\sum_{i=1}^{n}(y_i-W^T X_i)^2-\lambda W^T W+K$$

Hence the MAP estimate is the regularized least squares solution.

Fisher Linear Discriminant

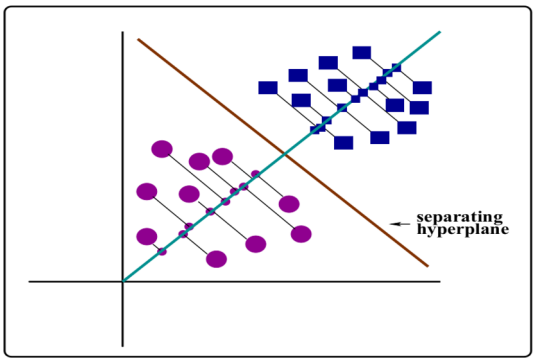
We project the data along the direction W . Hence one can think of the best W as the direction along which the two classes are well separated.

Let $\{(X_i,y_i), i=1,\ldots,n\}$ be the data. Let $y_i\in\{0, 1\}$. Let C_0 and C_1 denote the two classes. Let n_0 and n_1 denote the number of examples of each class.

$(n=n_0 + n_1)$

For any W , let $z_i = W^TX_i$. z_i are the one dimensional data that we get after projection.

Let M_0 and M_1 be the means of data from the two classes:



$$M_0=\frac{1}{n_0}\sum_{X_i\in C_0}X_i$$
$$M_1=\frac{1}{n_1}\sum_{X_i\in C_1}X_i$$

The corresponding means of the projected data would be $m_0 = W^TM_0$ and $m_1 = W^TM_1$.

The difference $(m_0 - m_1)$ gives us an idea of the separation between samples of the two classes after projecting the data onto the direction W . Hence, we may want a W that maximizes $(m_0 - m_1)^2$.

However, we have to make this scale independent. Define

$$s_0^2 = \sum_{X_i \in C0} (W^T X_i - m_0)^2 \;; \qquad s_1^2 = \sum_{X_i \in C_1} (W^T X_i - m_1)^2$$

These give us the variances (up to a factor) of the two classes in the projected data. We want large separation between m_0 and m_1 relative to the variances. Hence we can take our objective to be to

$$maximize\, J(W) = \frac{(m_1 - m_0)^2}{s_0^2 + s_1^2}$$

We now rewrite J into a more convenient form. We have

$$\begin{aligned}(m_1 - m_0)^2 &= (W^T M_1 - W^T M_0)^2 \\ &= [W^T (M_1 - M_0)][W^T (M_1 - M_0)]^T \\ &= W^T (M_1 - M_0)(M_1 - M_0)^T W\end{aligned}$$

Denote $S_B = (M_1 - M_0)(M_1 - M_0)^T$ (called between class scatter matrix).
Similarly we have

$$\begin{aligned}s_0^2 &= \sum_{X_i \in C_0} (W^T X_i - W^T M_0)^2 \\ &= \sum_{X_i \in C_0} [W^T (X_i - M_0)] \\ &= \sum_{X_i \in C_0} W^T (X_i - M_0)(X_i - M_0)^T W \\ s_0^2 &= W^T \left[\sum_{X_i \in C_0} (X_i - M_0)(X_i - M_0)^T \right] W\end{aligned}$$

Similarly, we get

$$s_1^2 = W^T \left[\sum_{X_i \in C_1} (X_i - M_1)(X_i - M_1)^T \right] W$$

Thus we can write $s_0^2 + s_1^2 = W^T S_w W$, where

$$S_w = \sum_{X_i \in C_0} (X_i - M_0)(X_i - M_0)^T + (X_i - M_1)(X_i - M_1)^T$$

S_w is called within class scatter matrix.
Hence we can now write J as

$$J(W) = \frac{W^T S_B W}{W^T S_w W}$$

We want to find a W that maximizes J(W). Differentiating w.r.t. W and equating to zero, we get

$$\frac{2 S_B W}{W^T S_w W} - \frac{W^T S_B W}{(W^T S_w W)^2} 2 S_w W = 0$$

Implies, $S_B W$ is in the same direction as $S_w W$. Thus, any maximizer of J(W) has to satisfy for some λ

$$S_w W = \lambda S_B W$$

Learning and Generalization

The problem of designing a classifier is essentially one of learning from examples. Given training data, we want to find an appropriate classifier. It amounts to searching over a family of classifiers to find one that minimizes ‘error’ over training set. Performance on training set is not the ultimate objective. We would like the learnt classifier to perform well on new data. This is the issue of **generalization**.
Notations:
i) X – input space; often $\Re^d$ (feature space)
ii) $\mathsf{Y} = \{0, 1\}$ – output space (set of class labels)
iii) $\mathsf{C} \subset 2^{\mathsf{X}}$– concept space (family of classifiers)
Each $C \in \mathsf{C}$ is a subset of X .

(iv) S = {(X_i , y_i), i = 1,…, n} – the set of examples, X_i are drawn iid according to some distribution P_x on X. $y_i = C^*(X_i)$ for some $C^* \in \mathsf{C}$. C^* is called target concept.

Probably Approximately Correct Learning

Let C_n denote the concept or classifier output by the learning algorithm after it processes 'n' iid examples. For correctness of the learning algorithm we want C_n to be 'close' to C^* as n becomes large. Define error of C_n by

$$err(C_n) = P_x(C_n \Delta C^*) = P_x((\bar{C}_n \cap C^*) \cap (C_n \cap \bar{C}^*))$$
$$= Prob[\{X : C_n(X) \neq C^*(X)\}]$$

The err(C_n) is the probability that on a random sample,drawn according to P_x , the classification of C_n and C^* differ. We want err(C_n) to become zero as $n \to \infty$

We say a learning algorithm **Probably Approximately Correctly (PAC)** learns a concept class C if given any $\epsilon, \delta > 0, \exists N < \infty$ such that

$$Prob[err(C_n) > \epsilon] < \delta$$

for all n > N and for any distribution P_x and any C^*.

However, PAC learnability deals with ideal learning situations. We assume there is a ('god-given') C^* and that it is in our C. Also, we assume that examples are noise free and are perfectly classified.

Now we consider a new framework.

X – input space; (as earlier, Feature space)

Y – Output space (as earlier, Set of class labels)

H – hypothesis space (family of classifiers)

Each $h \in \mathsf{H}$ is a function $h : \mathsf{X} \to \mathsf{A}$ where A is called action space. Training data: {(X^i , y^i), i =1,..., n} drawn iid according to some distribution P_{xy} on $\mathsf{X} \times \mathsf{Y}$.

Define Loss function: $L : \mathsf{Y} \times \mathsf{A} \to \Re^+$

Define the risk function, $R : \mathsf{H} \to \Re^+$, by

$$R(h) = E[L(y, h(X))] = \int L(y, h(X))\, dP_{xy}$$

We want to find h with low risk $h^* = arg\, min_{h \in H} R(h)$

Define the empirical risk function, $\Re^n : \mathsf{H} \to \Re^+$, by

$$\hat{R}_n(h) = \frac{1}{n} \sum_{i-1}^{n} L(y_i, h(X_i))$$

This is the sample mean estimator of risk obtained from n iid samples. Let $\hat{h}_n^*$ be the global minimizer of empirical risk, $\hat{R}_n$

$$\hat{h}_n^* = arg\, min_{h \in H} \hat{R}_n(h)$$

Consistency of Empirical Risk Minimization

We would like the algorithm to satisfy : $\forall \epsilon, \delta > 0 \; \exists N < \infty$, such that

$$Prob\left[\left|R(\hat{h}_n^*) - R(h^*)\right| > \epsilon\right] \leq \delta, \; \forall n \geq N$$

In addition, we would also like to have

$$Prob\left[\left|\hat{R}_n(\hat{h}_n^*) - R(h^*)\right| > \epsilon\right] \leq \delta, \; \forall n \geq N$$

For empirical risk minimization to be effective, we need $R(\hat{h}_n^*)$ to converge in probability to $R(h^*)$. This will happen if $\hat{R}_n(h)$ converges to R(h) uniformly over H. (H is the family of classifiers over which we are minimizing empirical risk). The needed uniform convergence holds if H has finite VC-dimension.

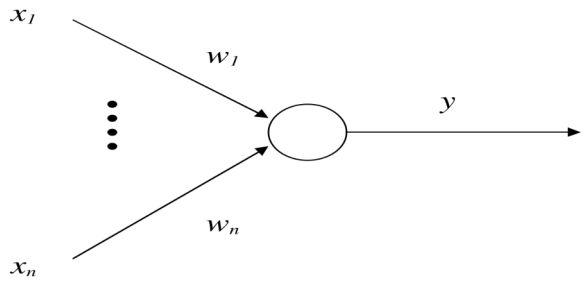
The VC-dimension also gives us an idea of the complexity of the family H. If the VC-dimension is high then we need correspondingly larger number of examples to have confidence that low empirical risk means low true risk.

Now we look at some Non-Linear Models

Neural Network Models

Single Neuron Models

x_i are inputs into the (artificial) neuron and w_i are the corresponding weights. y is the output of the neuron



Net input : $\eta = \sum_{i=1}^{n} x_i w_i$

Output: y = f(η), where f (.) is called activation function (Perceptron, AdaLinE are such models).

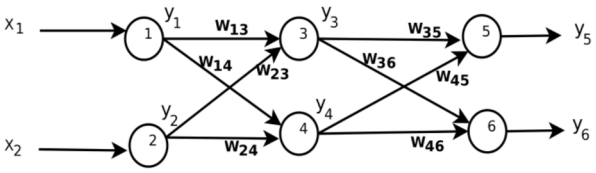
Networks in Neurons

Notation:

y_j Output of j^{th} neuron;

w_{ij} Weight of connection from neuron 'i' to neuron 'j'

f_i Activation function for i^{th} neuron.



$$y_5 = f_5(w_{35} y_3 + w_{45} y_4)$$
$$= f_5(w_{35} f_3(w_{13} y_1 + w_{23} y_2) + w_{45} f_4(w_{14} y_1 + w_{24} y_2))$$

Typical Activation function

1) Hard limiter

$$f(x) = 1 \quad if\ x > \tau$$
$$= 0 \quad otherwise$$

2) Sigmoid function:

$$f(x) = \frac{1}{1 + \exp(-bx)}; \quad a, b > 0$$

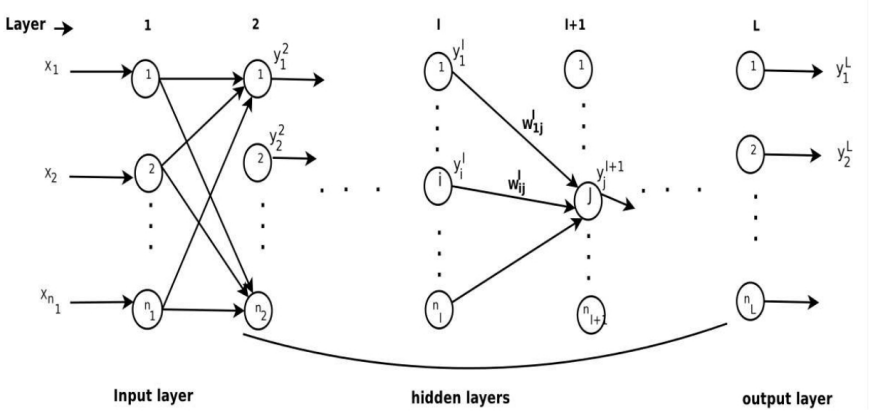
3) tanh

$$f(x) = a \tanh(bx); \quad a, b > 0$$

4) ReLU (Rectified Linear Unit)

$$f(x) = max(0, x)$$

Multilayer feed-forward Network



Notation :

L : Number of layers

n_l : Number of nodes in layer l, l = 1,…, L.

y_i^l : Output of i[th] node in layer l, i=1,…,n_l, l=1,…,L.

w_{ij}^l : Weight of connection from node-i, layer-l to node-j, layer-(l + 1).

η_i^l : Net input of node-i in layer-l

Network represents a function from $\Re^{n_1} \rightarrow \Re^{n_L}$

The outputs of a typical unit is computed as

$$\eta_j^l = \sum_{i=1}^{n_{l-1}} w_{ij}^{l-1} i_i^{l-1}$$

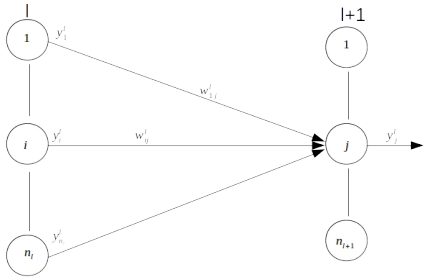
$$y_j^l = f(\eta_j^l)$$

(can also include a bias term)

To get a specific function we need to learn appropriate weights

Backpropagation Algorithm

Any weight w_{ij}^l can affect loss function J only by affecting the final output of the network. In a layered network, the weight w_{ij}^l can affect the final output only through its affect on η_j^{l+1}



Hence, using the chain rule of differentiation, we have

$$\frac{\partial j}{\partial w_{ij}^l} = \frac{\partial J}{\partial \eta_j^{l+1}} \frac{\partial \eta_j^{l+1}}{\partial w_{ij}^l}$$

We know, $\eta_j^{l+1} = \sum_{s=1}^{n_l} w_{sj}^l y_s^l \Rightarrow \frac{\partial \eta_j^{l+1}}{\partial w_{ij}^l} = y_i^l$

Define, $\delta_j^l = \frac{\partial J}{\partial \eta_j^l}$

Now we get

$$\frac{\partial J}{\partial w_{ij}^l} = \delta_j^{l+1} y_i^l$$

We can get all the needed partial derivatives if we calculate δ_j^l for all nodes. We can compute δ_j^l recursively:

$$\begin{aligned}
\delta_j^l = \frac{\partial J}{\partial \eta_j^l} &= \sum_{s=1}^{n_{l+1}} \frac{\partial J}{\partial \eta_s^{l+1}} \frac{\partial \eta_s^{l+1}}{\partial \eta_j^l} \\
&= \sum_{s=1}^{n_{l+1}} \frac{\partial J}{\partial \eta_s^{l+1}} \frac{\partial \eta_s^{l+1}}{\partial y_j^l} \frac{\partial y_j^l}{\partial \eta_j^l} \\
&= \sum_{s=1}^{n_{l+1}} \delta_s^{l+1} w_{js}^l f'(\eta_j^l)
\end{aligned}$$

We need to first compute δ_j^L

By definition,

$$\delta_j^L = \frac{\partial J}{\partial \eta_j^L}$$

We have

$$J = \frac{1}{2} \sum_{i=1}^{n_L} (y_j^L - d_j)^2$$

Hence we have

$$\delta_j^L = \frac{\partial J}{\partial \eta_j^L} = \frac{\partial J}{\partial y_j^L}\frac{\partial y_j^L}{\partial \eta_j^L}$$
$$= (y_j^L - d_j) f'(\eta_j^L)$$

Then we can compute δ_j^l , j = 1, · · · , n_l for l = (L−1),…, 2, recursively, using

$$\delta_j^l = \left(\sum_{s=1}^{n_{l+1}} \delta_s^{l+1} w_{js}^l\right) f'(\eta_j^l)$$

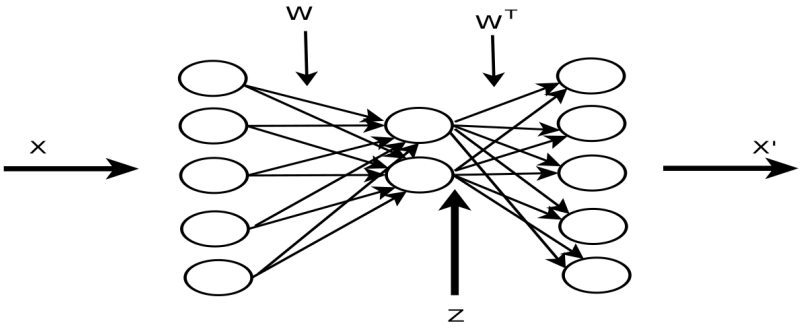
Then we compute all partial derivatives with respect to weights as

$$\frac{\partial J}{\partial w_{ij}^l} = \delta_j^{l+1} y_i^l$$

Hence can update the weights using the gradient descent procedure.

Auto-Encoder

Given data, {X_1,… ,X_m}, we want to learn through backpropagation. This is unsupervised learning. We learn a ‘compressed’ representation. The representation may be ‘good’ because we can recreate original X.



Denoising Autoencoder

We would give noise-corrupted X at input but want X′ to be the ‘clean’ X. We can add independent noise to each component of X. But, what is often done is to make a few randomly selected components of X zero.

If we can learn W to create X at output, then that W can capture dependences among components of X. Hence, W is a good set of weights to transform X into a useful representation.

Sparse Autoencoder

We are learning a ‘compressed’ representation by having only few nodes in hidden layer. Alternately, only a few of the hidden nodes should be ‘active’ for any given X. Then we need not have any restriction on the number of hidden layer nodes. By making representation ‘sparse’ we achieve similar ’compression’. Sparsity can be incorporated into the objective function.

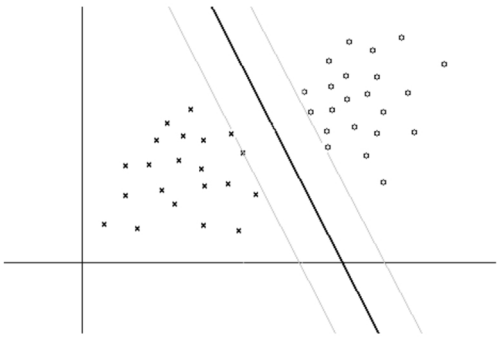
Support Vector Machines(SVM)

The idea is to map the feature vectors nonlinearly into another space and learn a linear classifier there. The linear classifier in this new space would be an appropriate nonlinear classifier in the original space. There are two major issues in naively using this idea. One is computational and the other is statistical If we want, e.g., p^{th} degree polynomial discriminant function in the original feature space $(\Re^m)$, then the transformed feature vector, Z, has dimension $O(m^p)$. Results in huge computational cost both for learning and and final operation of the classifier. We need to learn $O(m^p)$ parameters rather than O(m) parameters. Hence may need much larger number of examples for achieving proper generalization. SVM offers an elegant solution to both.

First we see **Optimal Hyperplane** i.e. **Linear SVM**

Training set:

$\{(X_i, y_i), i = 1, \ldots, n\}$, $X_i \in \Re^m$, $y_i \in \{+1, -1\}$



Assume training set is linearly separable i.e., $\exists W \in \Re^m$ and $b \in \Re$ such that

$$W^T X_i + b > 0, \quad \forall i \, s.t. \, y_i = +1$$
$$W^T X_i + b < 0, \quad \forall i \, s.t. \, y_i = -1$$

$W^T X + b = 0$ A separating hyperplane. (Infinitely many separating hyperplanes exist).
Since the training set is finite, $\exists \epsilon_1, \epsilon_2 > 0 \; s.t.$

$$W^T X_i + b \geq \epsilon_1, \forall i \quad s.t. \, y_i = +1$$
$$W^T X_i + b \leq -\epsilon_2, \forall i \quad s.t. \, y_i = -1$$

By dividing by min{ϵ_1, ϵ_2},

$$\bar{W}^T X_i + \bar{b} \geq +1 \forall i \quad s.t. \, y_i = +1$$
$$\bar{W}^T X_i + \bar{b} \leq -1 \forall i \quad s.t. \, y_i = -1$$

Hence, when training set is linearly separable, we can scale W, b such that

$$W^T X_i + b \geq +1 \forall i \quad s.t. \, y_i = +1$$
$$W^T X_i + b \leq -1 \forall i \quad s.t. \, y_i = -1$$

or, equivalently

$$y_i (W^T X_i + b) \geq 1, \forall i$$

Distance between these two hyperplanes is: $\frac{2}{\|W\|}$,called margin of the separating hyperplane. The distance between the hyperplane and the closest pattern is $\frac{1}{\|W\|}$ More the margin, better is the chance of correct classification of new patterns. We need a separating hyperplane with maximum margin.
The optimal hyperplane is a solution to the following optimization problem.
Find $W \in \Re^m, b \in \Re$ to

$$minimize \quad \frac{1}{2} W^T W$$
$$s.t. \;\; y^i (W^T X^i + b) \geq 1, i = 1, \ldots, n$$

Since the problem is a Convex Program, KKT conditions are necessary and sufficient,
The Lagrangian is given by

$$L(W, b, \mu) = \frac{1}{2} W^T W + \sum_{i=1}^{n} \mu_i \left[1 - y_i (W^T X_i + b) \right]$$

The Kuhn-Tucker conditions give

$$\nabla_W L = 0 \Rightarrow W^* = \sum_{i=1}^{n} \mu_i^* y_i X_i$$
$$\frac{\partial L}{\partial b} = 0 \Rightarrow \sum_{i=1}^{n} \mu_i^* y_i = 0$$
$$1 - y_i (X_i^T W^* + b^*) \leq 0, \forall i$$
$$\mu_i^* \geq 0 \forall i$$
$$\mu_i^* \left[1 - y_i (X_i^T W^* + b^*) \right] = 0, \forall i$$

Let S = {i | $\mu_i^* > 0$} . From Kuhn-Tucker conditions, we have

$$\mu_i^* \left[1 - y_i (X_i^T W^* + b^*) \right] = 0$$

Hence $i \in S \Rightarrow \mu_i^* > 0 \Rightarrow y_i (X_i^T W^* + b^*) = 1$ Implies X^i is closest to separating hyperplane.
{X_i | i ∈ S} are called Support vectors. We have

$$W^* = \sum_i \mu_i^* y_i X_i = \sum_{\mu_i^* > 0} \mu_i^* y_i X_i = \sum_{i \in S} \mu_i^* y_i X_i$$

Optimal W is a linear combination of Support vectors. Support vectors constitute a very useful output of the method.

$$b^* = y_j - X_j^T W^*, \quad j \ s.t. \ \mu_j > 0$$

We use the dual of the optimization problem to get μ_i^*

The dual function is

$$q(\mu) = inf_{W,b} \left\{ \frac{1}{2} W^T W \sum_{i=1}^{n} \mu_i \left[1 - y_i (W^T X_i + b) \right] \right\}$$

We obtain the dual by substituting $W = \sum \mu_i y_i X_i$ and imposing $\sum \mu_i y_i = 0$, Hence we have

$$\begin{aligned} q(\mu) &= \frac{1}{2} W^T W + \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \mu_i y_i (W^T X_i + b) \\ &= \frac{1}{2} \left(\sum_i \mu_i y_i X_i \right)^T \sum_j \mu_j y_j X_j + \sum_i \mu_i - \sum_i \mu_i y_i X_i^T \left(\sum_j \mu_j y_j X_j \right) \\ &= \sum_i \mu_i - \frac{1}{2} \sum_i \sum_j \mu_i y_i \mu_j y_j X_i^T X_j \end{aligned}$$

The dual problem is :

$$max_{\mu} \quad q(\mu) = \sum_i \mu_i - \frac{1}{2} \sum_i \sum_j \mu_i y_i \mu_j y_j X_i^T X_j$$

$$s.t. \quad \mu_i \geq 0 \ \ i = 1, \ldots, n; \quad \sum_{i=1}^{n} y_i \mu_i = 0$$

The optimal hyperplane is a solution of

$$minimize \quad \frac{1}{2} W^T W$$

$$s.t. \ \ y_i (W^T X_i + b) \geq 1, \ \ i = 1 \ldots n$$

We solve the above dual. Then the final solution is:

$$W^* = \sum \mu_i^* y_i X_i, \quad b^* = y_j - X_j^T W^*, \quad j \ s.t. \ \mu_j > 0$$

Using Slack Variable

When data are not linearly separable, we can try:

$$minimize \frac{1}{2} W^T W + C \sum_{i=1}^{n} \xi_i$$

$$s.t. \ \ y_i (W^T X_i + b) \geq 1 - \xi_i, i = 1, \ldots, n$$

$$\xi_i \geq 0, \ i = 1, \ldots, n$$

The Lagrangian is

$$L(W, b, \xi, \mu, \lambda) = \frac{1}{2} W^T W + C \sum_{i=1}^{n} \xi_i + \sum_{i=1}^{n} \mu_i (1 - \xi_i - y_i (W^T X_i + b)) - \sum_{i=1}^{n} \lambda_i \xi_i$$

The KKT conditions gives us,

$$\nabla_W L=0 \Rightarrow W^*=\sum_{i=1}^{l} \mu_i^* y_i X_i$$

$$\frac{\partial l}{\partial b}=0 \Rightarrow \sum \mu_i^* y_i=0$$

$$\frac{\partial l}{\partial \xi_i}=0 \Rightarrow \mu_i^*+\lambda_i^*=C, \ \forall i$$

$$1-\xi_i-y_i(W^T X_i+b)\le 0; \ \xi_i\ge 0 \ \ \forall i$$

$$\mu_i\ge 0; \ \ \lambda_i\ge 0 \ \ \forall i$$

$$\mu_i(1-\xi_i y_i(W^T x_i+b))=0; \ \lambda_i \xi_i=0 \ \ \forall i$$

The W^* is given by the same expression. We also have $0 \le \mu_i + \lambda_i = C, \forall$ i.
If $0 < \mu_i < C$, then, $\lambda_i > 0$ which implies $\xi_i = 0$. Now the complementary slackness condition, we have

$$1 - y^i(W^T X_i+b)=0$$

Thus we get b^* as :

$$b^*=y_j - X_j^T W^*, \ j \quad s.t. \ 0<\mu_j<C$$

The dual problem will be now:

$$max_\mu q(\mu)=\sum_{i=1}^{n} \mu_i-\frac{1}{2}\sum_{i,j=1}^{n} \mu_i\mu_j y_i y_j X_i^T X_j$$

$$s.t. 0\le\mu_i\le C. i=1,...n; \sum_{i=1}^{n} y_i\mu_i=0$$

Non Linear Classifier

In general, we can use a mapping, $\phi: \Re^m \rightarrow \Re^{m'}$
In $\Re^m$, the training set is $\{(Z_i,y_i), i = 1,\ldots, l\}$, $Z_i = \phi(X_i)$. We can find optimal hyperplane by solving the dual (replacing $X_i^T X_j$ with $Z_i^T Z^j$).
The dual problem now would be the following.

$$max_\mu q(\mu)=\sum_{i=1}^{n} \mu_i-\frac{1}{2}\sum_{i,j=1}^{n} \mu_i\mu_j y^i y^j \phi(X_i)^T \phi(X_j)$$

$$s.t. 0\le\mu_i\le C. i=1,...n; \sum_{i=1}^{n} y_i\mu_i=0$$

This is an optimization problem over R (with quadratic cost function & linear constraints) irrespective of ϕ and m^l.

Kernel function

Suppose we have a function, $K:\Re^m\times\Re^m\rightarrow\Re$,(called Kernel function), such that

$$K(X_i, X_j)=\phi(X_i)^T \phi(X_j)$$

Computation of $K(X_i,X_j)$ is about as expensive as that of $X_i^T X_j$. Replacing $Z_i^T Z_j$ by $K(X_i,X_j)$, we can solve dual without ever computing any $\phi(X_i)$.
Let μ_i^* be soln of Dual. Then $W^*=\mu_i^* y_i \phi(X_i)$ .Then we have

$$b^* = y_j - \phi(X_j)^T W^* = (y_j - \sum_i \mu_i^* y_i \phi(X_i)^T \phi(X_j))$$

Given a new pattern X, we only need to compute

$$\begin{aligned} f(X) &= Z^T W^* + b^* = \phi(X)^T W^* + b^* \\ &= \mu_i^* y_i \phi(X_i)^T \phi(X) + b^* \\ &= \sum_i \mu_i^* y_i K(X_i, X) + (y_j - \sum_i \mu_i^* y_i K(X_i, X_j)) \end{aligned}$$

This is an interesting way of learning nonlinear classifiers. We solve the dual whose dimension is that of n, number of examples. All we need to store are:

- Non-zero Lagrange multipliers: $\mu_i^* > 0$
- Support vectors: $X_i,\ i \quad s.t. \mu_i^* > 0$

Then we compute

$$f(X) = \sum_i \mu_i^* y_i K(X_i, X) + (y_j - \sum_i \mu_i^* y_i K(X_i, X_j))$$

and classify 'X' based on sign of f(X). Never need to enter 'ϕ(X)' space!!

Popular Kernels:

1) Polynomial Kernel $K_p(X_1, X_2) = (1 + X_1^T X_2)^p$

2)Gaussian Kernel $K_G(X_1, X_2) = \exp\left(-\frac{\|X_1 - X_2\|^2}{\sigma^2}\right)$

Example of Kernel function

Consider a Kernel function in $\Re^2$ such that $K(X_i, X_j) = [1 + \phi(X_i)^T \phi(X_j)]^2$. Let $X_i = (x_{i1}, x_{i2})^T$ ans similarly for X_j. Then ,

$$K(X_i, X_j) = (1 + x_{i1} x_{j1} + x_{i2} x_{j2})^2$$

We only need to show that there exist a mapping $\phi : K(X_i, X_j) = \phi(X_i)^T \phi(X_j)$.

Consider $\phi: \Re^2 \rightarrow \Re^6$ given by

$$Z = \phi(X) = [1 \quad x_1^2 \quad x_2^2 \quad \sqrt{2} x_1 \quad \sqrt{2} x_2 \quad \sqrt{2} x_1 x_2]$$

Here $X = (x_1, x_2)^T \in \Re^2$

We can easily show that $K(X_i, X_j) = (1 + X_i^T X_j)^2 = Z_i^T Z_j = \phi(X_i)^T \phi(X_j)$

We have

$$\begin{aligned} Z_i^T Z_j &= 1 + x_{i1}^2 x_{j1}^2 + x_{i2}^2 x_{j2}^2 + 2 x_{i1} x_{j1} + 2 x_{i2} x_{j2} + 2 x_{i1} x_{i2} x_{j1} x_{j2} \\ &= (1 + x_{i1} x_{j1} + x_{i2} x_{j2})^2 \\ &= (1 + X_i^T X_j)^2 \\ &= K(X_i, X_j) \end{aligned}$$

Mercer Theorem:

Given a symmetric function $K: \Re^m \times \Re^m \rightarrow \Re$, there exists an inner product space H and a mapping $\phi: \Re^m \rightarrow H$ so that $K(X_1, X_2) = \phi(X_1)^T \phi(X_2)$ if for all square-integrable functions g,

$$\int K(X_1, X_2) g(X_1) g(X_2) dX_1 dX_2 \geq 0$$

Using Mercers's Theorem we can show that functions of the form (given below) are kernels

$$K(X_1, X_2) = \sum_{j=0}^{p} a_j (X_1^T X_2)^j; \quad a_j \geq 0$$

Some theoretical results with SVM

Let P_{err}^n be the error rate on a test set for an SVM trained with 'n' random examples. Then we can show (for SVM with no slack variables)

$$E[P_{err}^n] \leq min\left(\frac{s}{n}, \frac{[R^2 \|W\|^2]}{n}, \frac{m}{n}\right)$$

's' is number of support vectors

'R' is the radius of smallest sphere enclosing all examples

$\|W\|^{-2}$ is the margin of the maximum margin hyperplane (in the feature space of dimension m).

Solving the SVM optimization problem

The optimization problem to be solved is

$$max_{\mu}\ \ q(\mu) = \sum_{i=1}^{n} \mu_i - \frac{1}{w} \sum_{i,j=1}^{n} \mu_i \mu_j y_i y_j K(X_i, X_j)$$
$$s.t.\ \ 0 \le \mu_i \le C, \quad i = 1, \ldots, n$$
$$\sum_{i=1}^{n} \mu_i y_i = 0$$

A quadratic programming (QP) problem with interesting structure.
One interesting idea to optimize – Chunking
We optimize on only a few variables at a time. Dimensionality of the optimization problem is controlled. We keep randomly choosing the subset of variables. Gave rise to the first specialized algorithm for SVM – SVM Light.

SMO Algorithm (Sequential Minimal Optimization)
Taking chunking to extreme level – what is the smallest set of variables we can optimize on?
We need to consider at least two variables because there is an equality constraint. SMO works on optimizing two variables at a time. The algorithm (heuristically) decides which two we consider in each iteration.

Support Vector Regression

Given training data $\{(X_1, y_1), \ldots, (X_n, y_n)\}, X_i \in \Re^m, y_i \in \Re$, we want to find 'best' function to predict y given X. We search in a parameterized class of functions

$$g(X, W) = w_1 \phi_1(X) + \ldots + w_{m'} \phi_{m'}(X) + b$$
$$= W^T \Phi(X) + b$$

where $\phi_i : \Re^m \rightarrow \Re$ are some chosen functions.
We want to formulate the problem so that we can use the Kernel idea
For this we use ϵ-insensitive loss, which is given by

$$L_\epsilon(y_i, g(X_i, W)) = 0 \qquad if \ \ |y_i - g(X_i, W)| < \epsilon$$
$$= |y_i - g(X_i, W)| - \epsilon \qquad otherwise$$

Empirical risk minimization under the ϵ-insensitive loss function would minimize

$$\sum_{i=1}^{n} max\left(|y_i - \Phi(X_i)^T W - b| - \epsilon, 0\right)$$

We can pose the problem as follows.

$$min_{W, b, \xi, \xi'} \quad \sum_{i=1}^{n} \xi_i + \sum_{i=1}^{n} \xi_i'$$
$$s.t. \quad y_i - W^T \Phi(X_i) - b \le \epsilon + \xi_i, \quad i = 1, \ldots, n$$
$$W^T \Phi(X_i) + b - y_i \le \epsilon + \xi_i', \quad i = 1, \ldots, n$$
$$\xi_i \ge 0, \quad \xi_i' \ge 0 \qquad i = 1, \ldots, n$$

But this does not give a dual with the structure we want. So, we reformulate the optimization problem.
Find W, b, ξ_i, ξ_i' to

$$minimize \quad \frac{1}{2} W^T W + C\left(\sum_{i=1}^{n} \xi_i + \sum_{i=1}^{n} \xi_i'\right)$$
$$s.t. \quad y_i - W^T \Phi(X_i) - b \le \epsilon + \xi_i, \quad i = 1, \ldots, n$$
$$W^T \Phi(X_i) + b - y_i \le \epsilon + \xi_i', \quad i = 1, \ldots, n$$
$$\xi_i \ge 0, \quad \xi_i' \ge 0 \qquad i = 1, \ldots, n$$

We have added the term W^TW in the objective function. This is like model complexity in a regularization context.
Given that this problem is similar to the earlier one, we would get W^* in terms of the optimal lagrange multipliers as earlier. Essentially, the Lagrange multipliers corresponding to the inequality constraints on the errors would be the determining factors.
The dual of the problem is :

$$max_{\alpha,\alpha'} \quad \sum_{i=1}^{n} y_i(\alpha_i - \alpha_i') - \epsilon\sum_{i=1}^{n}(\alpha_i + \alpha_i') - \frac{1}{2}\sum_{i,j=1}^{n}(\alpha_i + \alpha_i')(\alpha_j - \alpha_j')\Phi(X_i)^T\Phi(X_j)$$

$$s.t. \quad \sum_{i=1}^{n}(\alpha_i - \alpha_i') = 0$$

$$0 \le \alpha_i, \alpha_i' \le C, i = 1,..,n$$

Here α_i, α_i' are the Lagrange multipliers corresponding to the first two inequality in primal.
Using KKT conditions, we get the final optimal values of 'W' and 'b' , which gives:

$$W^* = \sum_{i=1}^{n}(\alpha_i^* - \alpha_i^{*\prime})\Phi(X_i)$$

$$b^* = y_j - \Phi(X_j)^T W^* + \epsilon, \quad j \ s.t. 0 < \alpha_j^* < C/n$$

Note that we have $\alpha_i^* \alpha_i^{*\prime} = 0$. Also, $\alpha_i^*, \alpha_i^{*\prime}$ are zero for examples where error is less than ϵ. The final W is a linear combination of some of the examples – the support vectors.
Also the dual and the final solution are such that we can use the kernel trick
In our formulation, we added W^TW term in the objective function i.e. we are minimizing

$$\frac{1}{2}W^TW + C\sum_{i=1}^{n} max\left(\left|y_i - \Phi(X_i)Z^TW - b\right| - \epsilon, 0\right)$$

The W^TW is the model complexity term which is intended to favour learning of 'smoother' models.
Why W^TW is a good term to capture degree of smoothness in case of linear models ?
Let $f: \Re^m \rightarrow \Re$ be a continuous function. Continuity means we can make $|f(X) - f(X')|$ as small as we want by taking ||X − X′|| sufficiently small. There are ways to characterize the 'degree of continuity' of a function. We consider one such measure now.
The ϵ-margin of a function, $f: \Re^n \rightarrow \Re$ defined as,

$$m_\epsilon(f) = inf\{\|X - X'\| : |f(X) - f(X')| \ge 2\epsilon\}$$

The intuitive idea is: How small can ||X − X ′ || be, still keeping |f(X) − f (X′)| 'large'
The larger $m_\epsilon(f)$, the smoother is the function.
Higher margin would mean the function is 'slowly varying' and hence is a 'smoother' model.
Consider case of linear models

$$|f(X) - f(X')| = |W^T(X - X')|$$

For all X, X′ with $|W^T(X - X')| \ge 2\epsilon$, we want the smallest ||X − X′|| , it would be smallest if $|W^T(X - X')| = 2\epsilon$ and (X − X′) is parallel to W .
That is, $X - X' = \pm\frac{2\epsilon W}{W^TW}$

$$m_\epsilon(f) = \left\|\pm\frac{2\epsilon W}{W^TW}\right\| = \frac{2\epsilon}{\|W\|}$$

Thus in our optimization problem adding the term W^TW promotes learning of smoother models. As we have seen linear regression models use this as the regularization term.
ν- SVM

The primal problem for SVM with slack variables is

$$\begin{gathered} minimize \frac{1}{2} W^T W + C \sum_{i=1}^{n} \xi_i \\ s.t. \;\; y_i (W^T X_i + b) \geq 1 - \xi_i, i = 1, \ldots, n \\ \xi_i \geq 0, \; i = 1, \ldots., n \end{gathered}$$

In this, we do not know how to choose C. Consider a changed optimization problem

$$\begin{gathered} minimize \frac{1}{2} W^T W - \nu\rho + \frac{1}{n} \sum_{i=1}^{n} \xi_i \\ s.t. \;\; y_i (W^T X_i + b) \geq \rho - \xi_i \\ \xi_i \geq 0 \end{gathered}$$

where ν is a user-chosen constant. Note that W,b,ρ,ξ_i = 0 is a feasible solution.
The Lagrangian for this problem is

$$L(W, b, \xi, \rho, \eta, \mu) = \frac{1}{2} W^T W - \nu\rho + \frac{1}{n} \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \eta_i \xi_i + \sum_{i=1}^{n} \mu_i (\rho - \xi_i - y_i [W^T \Phi(X_i)] + b)$$

The μ_i are the Lagrange multipliers for the separability constraints and η_i are the Lagrange multipliers for the constraints $\xi_i \geq 0$.
The Kuhn-Tucker conditions give us

$$\begin{gathered} \nabla_W L = 0 \Rightarrow W = \sum_{i=1} \mu_i y_i \phi(X_i) \\ \frac{\partial L}{\partial b} = 0 \Rightarrow \sum_i \mu_i y_i = 0 \\ \frac{\partial L}{\partial \xi_i} = 0 \Rightarrow \mu_i + \eta_i = \frac{1}{n}, \;\; \forall i \\ \frac{\partial L}{\partial \rho} = 0 \Rightarrow \sum_i \mu_i = \nu \\ \rho - \xi_i - y_i (W^T \phi(X_i) + b) \leq 0; \;\; \xi_i \geq 0; \;\; \forall i \\ \mu_i \geq 0; \;\; \eta_i \geq 0, \;\; \forall i \\ \mu_i (\rho - \xi_i - y_i (W^T \phi(X_i) + b)) = 0, \;\; \forall i \\ \eta_i \xi_i = 0, \;\; \forall i \end{gathered}$$

Suppose $\xi_i > 0$ for some 'i' , then we have $\eta_i = 0$ and $\mu_i = 1/n$, Hence

$$\begin{aligned} \nu &= \sum_{i=1}^{n} \mu_i = \sum_{i: \xi_i > 0} \mu_i + \sum_{i: \xi_i = 0} \mu_i \\ &\geq \sum_{i: \xi_i > 0} \mu_i \\ &= \frac{\left| \{ i : \xi_i > 0 \} \right|}{n} \end{aligned}$$

ν is upper bounded on the fractions of 'margin error'

$$\begin{aligned} \nu &= \sum_{i=1}^{n} \mu_i = \sum_{i: \mu_i > 0} \mu_i + \sum_{i: \mu_i = 0} \mu_i \\ &\leq \sum_{i: \mu_i > 0} \mu_i \leq \frac{\left| \{ i : \mu_i > 0 \} \right|}{n} \end{aligned}$$

ν is a lower bound on fraction of support vectors.
If for the chosen ν, the problem has a solution with ρ > 0 , then both the bounds would be met.

The dual for the ν-SVM turns out to be

$$max_{\mu} \;\; q(\mu) = -\frac{1}{2}\sum_{i,j=1}^{n} \mu_i \mu_j y_i y_j K(X_i, X_j)$$

$$s.t. \quad 0 \le \mu_i \le \frac{1}{n}, \;\; \forall i$$

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

$$\sum_{i=1}^{n} \mu_i = \nu$$

This a simple optimization problem similar to that of 'C-SVM'. It can be shown that if we have a solution for ν-SVM then if we choose C = 1/ρn, we get the same solution with 'C-SVM'

ν-SVR

The idea of ν-SVM can be extended to the regression problem also. In SVR, we had two user defined constants: ϵ and C. The ϵ specifies the 'tolerable error' and it is difficult to know what value to choose for it. We can reformulate SVR so that we can optimize on ϵ also.

In SVR , we have

$$minimize_{W,b,\xi,\xi'} \quad \frac{1}{2}W^T W + C\left(\sum_{i=1}^{n}\xi_i + \sum_{i=1}^{n}\xi_i'\right)$$

$$s.t. \quad y_i - W^T\Phi(X_i) - b \le \epsilon + \xi_i, \;\; i = 1,\ldots,n$$

$$W^T\Phi(X_i) + b - y_i \le \epsilon + \xi_i', \;\; i = 1,\ldots,n$$

$$\xi_i \ge 0, \quad \xi_i' \ge 0 \qquad i = 1,\ldots,n$$

We change the optimization problem to the following:

$$minimize_{W,b,\epsilon,\xi,\xi'} \quad \frac{1}{2}W^T W + C\left(\nu\epsilon + \frac{1}{n}\sum_{i=1}^{n}(\xi_i + \xi_i')\right)$$

$$s.t. \quad y_i - W^T\Phi(X_i) - b \le \epsilon + \xi_i, \;\; i = 1,\ldots,n$$

$$W^T\Phi(X_i) + b - y_i \le \epsilon + \xi_i', \;\; i = 1,\ldots,n$$

$$\xi_i \ge 0, \quad \xi_i' \ge 0 \qquad i = 1,\ldots,n$$

$$\epsilon \ge 0$$

Positive definite kernels

Let K be a $n \times n$ matrix with $K_{i,j} = K(X_i, X_j)$. A positive definite kernel is the function K such that K is positive semi-definite for all 'n' and all data sets $\{X_1,\ldots, X_n\}$.

That is, given any 'n', and any feature vectors $X_1,\ldots, X_n$, we have for all scalars $c_1,\ldots,c_n$

$$\sum_{i,j=1}^{n} c_i c_j K(X_i, X_j) \ge 0$$

Given any 'n' points, $X_1,\ldots, X_n \in \mathsf{X}$, the n × n matrix with (i,j) element as $K(X_i,X_j)$ is called the Gram matrix of K. If K is positive definite if the Gram matrix ispositive semi-definite for all n and all $X_1,\ldots, X_n$.

All positive definite kernels are inner-products on some appropriate space. This space is called the Reproducing Kernel Hilbert Space(RKHS) associated with the Kernel (K).

Let $\Re^X$ be the set of all real-valued functions on X . Let K be a positive definite kernel.

For any $X \in \mathsf{X}$, let $K(\cdot, X) \in \Re^X$ denote the function that maps $X' \in \mathsf{X}$ to $K(X',X) \in \Re$. That is, $K(\cdot, X)(X') = K(X, X')$.

Consider the set of functions $\mathsf{H}_1 = \{K(\cdot, X) : X \in \mathsf{X}\}$.

Let H be the set of all functions that are finite linear combinations of functions in H_1 .
Any f(·) ∈ H can be written as

$$f(\cdot)=\sum_{i=1}^{n} \alpha_i K(\cdot, X_i) \qquad \text{for some 'n' ; Xi} \in \mathsf{X} \; ; \; \alpha_i \in \Re$$

It is easy to see that if $f, g \in \mathsf{H}$ then $f+g \in \mathsf{H}$ and $\alpha f \in \mathsf{H}$ for $\alpha \in \Re$ Thus, H is a vector space. Let,

$$f(\cdot)=\sum_{i=1}^{n} \alpha_i K(\cdot, X_i) \;\; ; \;\; g(\cdot)=\sum_{j=1}^{n'} \beta_i K(\cdot, X_j{}')$$

Define the inner product as

$$\langle f, g \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n'} \alpha_i \beta_j K(X_i, X_j{}')$$

One can show this is well defined and satisfy all the properties of inner product.
Reproducing Kernel property : for any f ∈ H, then

$$\langle f, K(,X) \rangle = \sum_{i=1}^{n} \alpha_i K(X, X_i) = f(X)$$

Assessing Learnt Models

We know no algorithm is inherently superior like this. If a method does better on some PR problems then it would do worse than average on some others.

Notation

Let D denote training set of n examples and let F denote the target function. Let $P_k(h(X)|D)$ denote the probability that the classifier learnt by the k[th] algorithm, with training data as D, would say h(X) on a pattern X.
Let $J_k(F, D)$ be the expected error of classifier learnt by k[th] algorithm using data D with target being F .

$$J_k(F, D) = \sum_{h} \sum_{X \neq D} P(X)\left[1 - \delta(F(X), h(X))\right] P_k(h(X)|D)$$

where $\delta(a, b)=1 \quad if\ a=b$
$=0 \quad otherwise$

(If we assume the learning algorithm to be deterministic, then, for a given D, P k (h|D) is non zero only for one h. Then we can omit the first summation.)

'No Free Lunch' Theorem

For any two learning algorithms,

$$\sum_{F} \sum_{D} Prob[D|F](J_1(F, D) - J_2(F, D)) = 0$$

(we take |D| = n fixed)
Averaged over all size-n training sets and uniformly averaged over all targets, F , the difference in expected errors of any two algorithms is zero.
The second part of theorem says that this is true, even if we fix D:

$$\sum_{F} (J_1(F, D) - J_2(F, D)) = 0$$

If we uniformly average over all problems (i.e., target functions, F), all algorithms are the same!

Bias-Variance Trade-off

Let Y denote the target random variable. Assume that Y = f(X) + ϵ where ϵ is a zero mean independent noise. Let $\hat{f}$ denote the function output by the learning algorithm.
Expected error in the prediction by our learning algorithm at some point x_0 will be \

$$
\begin{aligned}
error(x_0) &= E\left[(Y-\hat{f}(x_0))^2 \middle| X=x_0\right] \\
&= E\left[(f(x_0)+\epsilon-\hat{f}(x_0))^2\right] \\
&= E[\epsilon^2]+E\left[(f(x_0)-\hat{f}(x_0))^2\right] \\
&= \sigma_\epsilon^2+E\left[\{(f(x_0)-E[\hat{f}(x_0)])+(E[\hat{f}(x_0)]-\hat{f}(x_0))\}^2\right] \\
&= \sigma_\epsilon^2+(f(x_0)-E[\hat{f}(x_0)])^2+E\left[(\hat{f}(x_0)-E[\hat{f}(x_0)])^2\right]
\end{aligned}
$$

because,

$$E\left[2(f(x_0)-E[\hat{f}(x_0)])(\hat{f}(x_0)-E[\hat{f}(x_0)])\right] = 2(f(x_0)-E[\hat{f}(x_0)])\,E\left[(\hat{f}(x_0)-E[\hat{f}(x_0)])\right] = 0$$

Hence,

$$
\begin{aligned}
error(x_0) &= \sigma_\epsilon^2+(f(x_0)-E[\hat{f}(x_0)])^2+E\left[(\hat{f}(x_0)-E[\hat{f}(x_0)])^2\right] \\
&= \sigma_\epsilon^2+bias^2(\hat{f}(x_0))+Var(\hat{f}(x_0))
\end{aligned}
$$

Bias is about how well, on the average, the learnt function captures the true underlying function. The variance is about how close the predictions of the learnt function would be (over different random training sets).

Low model complexity can lead to high bias. We can say that high bias indicates ‘poor estimate’.

High model complexity can lead to high variance. We can say high variance indicates ‘weak estimate’.

But this is not a ’zero-sum game’. If we increase model complexity and also increase number of examples, we can simultaneously decrease bias and variance.

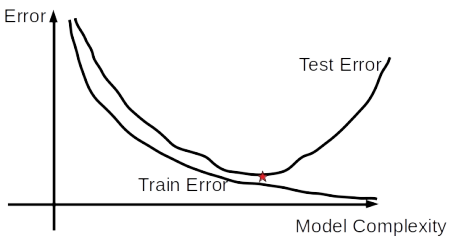
We are given a set of iid examples. We learn a classifier (or a regression function) using these examples. This essentially involves two steps:

- Selecting a suitable class of models (Model selection)
- Learning a specific model (a classifier) in this class (Model estimation)

One method of assessing learnt models is the use of so called validation set

Hold out Validation Set

We keep some part of the training data separately (validation set). We then learn our model using only the remaining training data. Then we estimate the true risk of the learnt model as the sample mean estimator on the validation set. We can use the validation set to choose between different models(i.e., to fix the hyperparameters). One can often recognize overfitting by looking at the plot of training and validation error versus the complexity of model class. This is often called the model selection curve(figure on right).



K-fold Cross-Validation

For cross-validation, we divide the data into K (roughly) equal sized parts. Then we run the model learning algorithm K times. On the i^{th} time, we use for training all the parts but the i^{th} one, and use the i^{th} part to estimate error of the learnt model. The final error estimate is an average of all these errors. This is referred to as K-fold cross validation.

Let $\{(X_1,y_1),\ldots,(X_n,y_n)\}$ be the data. Let $\rho : \{1,\ldots, n\} \rightarrow \{1,\ldots, K\}$ denote a index function that tells which data sample is in which part. Let $\hat{f}^{-k}$ be the model learnt when we leave part k for testing and use all the other parts for training. Thus, $\hat{f}^{-\rho(i)}(X_i)$ would be the prediction on X_i for the model which is learnt with training data that does not contain X_i .

The final cross-validation error estimate is

$$e_{cv} = \frac{1}{n}\sum_{i=1}^{n} L(\hat{f}^{-\rho(i)} X_i, y_i)$$

Bootstrap Methods

The idea in bootstrap is to generate many training sets by sampling with replacement from the given data. Given the original data of n points, we generate B number of training sets, each of size n, by randomly sampling from the given data set. Then we learn a model on each of the B training sets. The final error estimate could be the average of errors of all the models.

Let $\hat{f}_b$ denote the model learnt using the b[th] bootstrap sample, b=1,…,B. The final bootstrap estimate of error is

$$e^1_{boot} = \frac{1}{B}\sum_{b=1}^{B}\left[\frac{1}{n}\sum_{i=1}^{n} L(\hat{f}_b(X_i), y_i)\right]$$

Here we are using the original data set as test data while for each b, the $\hat{f}_b$ is learnt using some of the same data. Hence this bootstrap error estimate would not be very good.

Example Consider a problem where the class label is independent of the feature vector. Then the true error rate is 0.5 (under 0–1 loss). Suppose we use the 1-nearest neighbour as our classifier. Then on any X_i , the classification by $\hat{f}_b$ would be correct if X_i is in the b[th] bootstrap sample. Otherwise, it would be correct with probability 0.5

Now,

$$\begin{aligned} P[X_i \in bootstrap\ sample\ b] &= 1 - \left(1 - \frac{1}{n}\right)^n \\ &\approx 1 - e^{-1} \\ &= 0.632 \end{aligned}$$

Thus expected value of our e^1_{boot} is about 0.5 × 0.368 = 0.184, which is much less than the true value of 0.5. We can reduce this bias by doing what we did in cross-validation – for each model use only those data samples which are not used in learning it. So, we define the error estimate as

$$e_{boot} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{|C^{-i}|}\sum_{b\in C^{-i}} L(\hat{f}_b(X_i), y_i)$$

where $C^{-i} = \{b : X_i \notin b^{th} bootstrap\ data\ set\}$. (We leave out any ‘i’ for which C^{-i} is null)

In cross validation we divide data into at most n parts and hence can learn only n different models.

In bootstrap we can choose B as high as we want and still have proper size of training sets. However, all training sets here are similar. Each bootstrap sample would have about 0.632n distinct samples. Hence, it roughly behaves like a 3-fold or 2-fold cross validation.

Bagging Estimate

In bootstrap, we are learning B number of models, $\hat{f}^b$, b = 1, · · · , B. Can we use all these to improve prediction of final model? This is called Bagging.

Bagging stands for Bootstrap Aggregation. It averages over the predictions of all models.Given all the bootstrap models learnt, at a point X, the bagging model prediction is (Often bagging can improve performance of the final model)

$$\hat{f}_{bag}(X) = \frac{1}{B}\sum_{i=1}^{B}\hat{f}^b(X)$$

Combining Classifier

Problem : If all the different classifiers are trained on the same data set we may not get much improvement.

We want different classifiers in the ensemble to ‘complement’ each other. We have only one training set of data. Somehow we need to create multiple data sets (with appropriate variations) to train different classifiers. We can also think of introducing variability in the way a classifier is learnt.

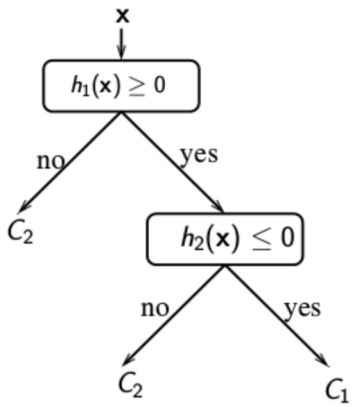
Another issue is how to combine the decisions of all the classifiers for prediction on new data.
The individual classifiers are called base classifiers. Let us call a classifier 'weak' if it gives only a little more than 50% accuracy. (In a 2-class case). The idea of classifier ensembles is to combine many 'weak' classifiers so that what we get is a 'strong' one. In bagging, one relies on randomness to introduce variability in classifiers.
A popular method is the so called random forests (here the base classifiers are decision trees).
A popular boosting algorithm is the so called Adaboost.
Example (Designing classifier ensembles) Suppose we want to create an ensemble with three component classifiers in a 2-class problem. Let D denote the given training data set with n points. We can begin by randomly selecting $n_1 < n$ points from D, putting them in a set D_1 . We can learn our first component classifier h 1 using the training set D_1 . For learning the second component classifier h_2 ,we want a data set D_2 that is most informative given h_1 in the sense that the performance of h_1 on D_2 would be no better than pure guessing. So, we want roughly only half the points in D_2 to be correctly classified by h_1. We learn classifier h_2 using training set D_2 .We can think of the third component classifier to be a kind of tie-breaker of h_1 and h_2. So, we can take all patterns in D where the classification by h_1 and h_2 differ and put it in a set D_3 . Now learn the third component classifier h_3 using training set D_3. We use this classifier ensemble as follows :
Given a new pattern X, if $h_1(X) = h_2(X)$ then that is what we output; otherwise we output $h_3(X)$.

Random Forest

Random forests is a method of learning a classifier ensemble using bagging. The base classifiers are decision trees. We learn many decision trees on randomly generated bootstrap data sets The decision tree learning is also randomized here. A majority rule is used for combining classifiers.

Decision Trees

A decision tree is a piece-wise linear classification method. Each (non-leaf) node is labeled with a so called 'split rule'. All leaf nodes are labeled with a class label. Given a feature vector, at each node we go to one of the children based on the value of split rule on this feature vector until we reach a leaf that gives the classification of the feature vector.



If the split rule depends on only one of the features then the tree is termed axis-parallel; otherwise called oblique. If a feature takes only finitely many values, then the axis-parallel tree need not be binary.
Learning a decision tree involves learning 'best' split rule at each node. Most decision tree algorithms learn the tree in a top-down fashion under a greedy heuristic
A top-down learning algorithm for decision trees learns the tree starting from the root. We start with the given training data. According to a chosen criterion we find the best split rule for this data. Then we make the root node with this split rule. We divide the data based on this split rule into the two sets that go into the left and right child nodes of the root. Now recursively we do the same thing at the two children nodes. We need a stopping criterion which decides when we decide to make a leaf node for the data.

AdaBoost

In AdaBoost we assign (non-negative) weights to points in the data set. The algorithm iteratively keeps learning new classifiers. In each iteration, we learn a classifier to minimize weighted error on training data. We assume we can find a classifier with weighted error less than 50%. After learning the current classifier, we increase the (relative) weights of data points that are misclassified by the current classifier. We learn another classifier with the modified weights. The variability in the classifier

ensemble comes from the modification of weights. The final classifier is a weighted majority voting by all the classifiers.
Let $\{(X_1,y_1),\ldots,(X_n,y_n)\}$ be the data. We take $y_i \in \{-1,+1\}$. Let $w_i(k)$ denote the weight for the i_{th} data point at k_{th} iteration. Let h_k denote the classifier learned at k^{th} iteration; we take $h_k(X) \in \{-1,+1\}$.
We assume that the error rate of each classifier on its training data is less than 0.5 , and I_A be the indicator function for event 'A'

AdaBoost Algorithm

1. Initialize: $w_i(1)=\frac{1}{n}, \forall i$.
2. For m = 1 to M do
   a. Learn classifier h_m to minimize $\sum_{i=1}^{n} w_i(m)\, I_{[y_i \neq h_m(X_i)]}$
   b. Let $\xi_m=\sum_{i=1}^{n} w_i(m)\, I_{[y_i \neq h_m(X_i)]}$ (We assume ξ m # < 0.5).
   c. Set $\alpha_m=\frac{1}{2}\ln\left(\frac{1-\xi_m}{\xi_m}\right)$ (We have $\alpha_m > 0$)
   d. Update the weights by
   $$w_i'(m+1)=w_i(m)\exp(-\alpha_m y_i h_m(X i))$$
   $$w i(m+1)=\frac{w_i'(m+1)}{\sum_i w_i'(m+1)}$$
3. Output the final classifier:
$$h(X)=sgn\left(\sum_{m=1}^{M} \alpha_m h_m(X)\right)$$

If $h_m(X_i)\neq y_i$, then $w_i(m+1)>w_i(m)$. If the current classifier misclassifies a pattern, its weight for the next iteration is increased. The weight update scheme of AdaBoost ensures

$$\sum_{i:h_m(X_i)\neq y_i} w_i(m+1) \;=\; \sum_{i:h_m(X_i)=y_i} w_i(m+1) \;=\; \frac{1}{2}$$

We have, $\xi_m=\sum_{i:h_m(X_i)\neq y_i} w_i(m)$ and the weights are normalized , we have $\sum_{i:h_m(X_i)=y_i} w_i(m)=1-\xi_m$ and by definition $\exp(\alpha_m)=\sqrt{\frac{1-\xi_m}{\xi_m}}$ Hence we have

$$\sum_i w_i'(m+1)=\sum w_i(m)\exp(-\alpha_m y_i h_m(X_i))$$
$$= \sum_{h_m(x_i)\neq y_i} w_i(m) e^{\alpha_m} + \sum_{h_m(x_i)=y_i} w_i(m) e^{-\alpha_m}$$
$$= e^{\alpha_m} \sum_{h_m(x_i)\neq y_i} w_i(m) + e^{-\alpha_m} \sum_{h_m(x_i)=y_i} w_i(m)$$
$$\sum_i w_i'(m+1) = \xi_m \sqrt{\frac{1-\xi_m}{\xi_m}} + (1-\xi_m)\sqrt{\frac{\xi_m}{1-\xi_m}}$$
$$= 2\sqrt{(1-\xi_m)\,\xi_m}$$

We also have,

$$\sum_{h_m(X_i)\neq y_i} w_i'(m+1) = \sum_{h_m(X_i)\neq y_i} w_i(m)\exp(-\alpha_m y_i h_m(X_i))$$
$$= \exp(\alpha_m) \sum_{h_m(X_i)\neq y_i} w_i(m)$$
$$= \exp(\alpha_m)\xi_m$$
$$= \sqrt{(1-\xi_m)\xi_m}$$

Using this, we get

$$\sum_{h_m(X_i)\neq y_i} w_i(m+1) = \sum_{h_m(X_i)\neq y_i} \frac{w_i'(m+1)}{\sum_i w_i'(m+1)}$$
$$= \frac{\sum_{h_m(X_i)\neq y_i} w_i'(m+1)}{\sum_i w_i'(m+1)}$$
$$= \frac{\sqrt{(1-\xi_m)\xi_m}}{2\sqrt{(1-\xi_m)\xi_m}}$$
$$= \frac{1}{2}$$

The weight update is such that at the next iteration, half the total weight is for patterns misclassified by the current classifier and the remaining half is for patterns correctly classified by the current classifier

How to design classifiers to minimize weighted errors?

One way is to generate different random training sets for each iteration. At m^{th} iteration, we generate a random set of examples by sampling with replacement from the original training set using the distribution $\{w_i(m)\}$. Now an algorithm minimizing errors on this random training set (approximately) minimizes the weighted error as needed. This is one of the standard ways of using AdaBoost

Feature Selection

Feature selection refers to learning of best subset of features. If we totally have n features then there are 2^n possible subsets of features. We can think of feature selection as being similar to model selection. So, we can use holdout validation or cross validation for this. However, generally n is large and hence this is not computationally feasible. So, we choose among only some of the subsets.

Here is a simple filter-based method for feature selection. We assign a score to each feature based on how well it correlates with class label. The score can be mutual information between x, the feature, and y, the class label. Given two random variables, x, y, the mutual information is the KL divergence between the joint distribution f (x, y) and the product of the marginals f (x)f (y):

$$MI(x,y) = \sum_{x,y} f(x,y)\ln\left(\frac{f(x,y)}{f(x)f(y)}\right)$$

Then we choose the K best features where K itself is chosen through, e.g., cross-validation.

We are looking for techniques to transform the original feature vector into a new feature vector. We want to determine this transformation based on a set of data points given (Unsupervised). We may want to do this to reduce the dimensionality of the feature vector without loosing too much information. We may want to do this to improve the features (e.g., make them uncorrelated). One such technique is the Principal Component Analysis (PCA). This is a general-purpose method useful in many problems of data analysis and machine learning.

Principal Component Analysis

We can think of PCA as a useful linear transformation of the feature vector. For dimensionality reduction, we essentially want to project the data onto a lower dimensional subspace. We can define PCA as projection onto a subspace such that

- variance of projected data is maximized, or
- mean-square error (in approximating a feature vector with its projection) is minimized

PCA as dimensionality reduction

Let $\{X_1, \ldots, X_n\}, X_i \in \Re^d$ be the given data. Suppose we want to project it onto an m-dimensional subspace. Let $U_1,\ldots,U_m$ denote an orthonormal basis for the m-dimensional subspace.
Let $U_1,\ldots, U_m, U_{m+1},\ldots, U_d$ denote the extension of this basis to whole of $\Re^d$.

$$X_i = \sum_{j=1}^{d} (X_i^T U_j) U_j$$

Let $\widetilde{X}_i$ (in the m-dimensional subspace), denote the approximation of X_i .

$$\widetilde{X}_i = \sum_{j=1}^{m} z_{ij} U_j + \sum_{j=m+1}^{d} \beta_j U_j$$

(Note that the second term does not depend on i)
We need to find the z_{ij} and β_j to get an approximation with least mean-square error.Hence we want z_{ij} and β_j to minimize :

$$\begin{aligned} J &= \frac{1}{n}\sum_{i=1}^{n} \|X_i - \widetilde{X}_i\|^2 \\ &= \frac{1}{n}\sum_{i=1}^{n} \left\| \sum_{j=1}^{d} (X_i^T U_j) U_j - \sum_{j=1}^{d} z_{ij} U_j - \sum_{j=m+1}^{d} \beta_j U_j \right\|^2 \\ &= \frac{1}{n}\sum_{i=1}^{n} \left\| \sum_{j=1}^{d} (X_i^T U_j - z_{ij}) U_j - \sum_{j=m+1}^{d} (X_i^T U_j - \beta_j) U_j \right\|^2 \\ &= \frac{1}{n}\sum_{i=1}^{n} \left[\sum_{j=1}^{d} (X_i^T U_j - z_{ij})^2 - \sum_{j=m+1}^{d} (X_i^T U_j - \beta_j)^2 \right] \end{aligned}$$

For any indices s and t ($1 \le s \le n$, $1 \le t \le m$),

$$\frac{\partial J}{\partial z_{st}} = 0 \;\Rightarrow\; 2(X_s^T U_t - z_{st}) = 0 \;\Rightarrow\; z_{st} = X_s^T U_t$$

Similarly, for any index t, $t \ge m + 1$,

$$\begin{aligned} \frac{\partial J}{\partial \beta_t} = 0 &\Rightarrow \frac{1}{n}\sum_{i=1}^{n} 2(X_i^T U_t - \beta_t) = 0 \\ &\Rightarrow \beta_t = \left(\frac{1}{n}\sum_{i=1}^{n} X_i \right)^T U_t = \bar{X}^T U_t \end{aligned}$$

where $\bar{X}$ is the mean of the data vectors.
Thus, for a given basis $\{U_j\}$, we get

$$\begin{aligned} \widetilde{X}_i &= \sum_{j=1}^{m} (X_i^T U_j) U_j + \sum_{j=m+1}^{d} (\bar{X}^T U_j) U_j \\ &= \sum_{j=1}^{m} (X_i^T U_j - \bar{X}^T U_j) U_j + \sum_{j=1}^{d} (\bar{X}^T U_j) U_j \\ \widetilde{X}_i &= \sum_{j=1}^{m} ((X_i - \bar{X})^T U_j) U_j + \bar{X} \end{aligned}$$

We now need to find the subspace that minimizes the error.

We have

$$\widetilde{X}_i = \sum_{j=1}^{m} (X_i^T U_j) U_j + \sum_{j=m+1}^{d} (\bar{X}^T U_j) U_j$$

Hence,

$$X_i - \widetilde{X}_i = \sum_{j=m+1}^{d} (X_i^T U_j - \bar{X}^T U_j) U_j$$

$$\|X_i - \widetilde{X}_i\|^2 = \sum_{j=m+1}^{d} ((X_i - \bar{X})^T U_j)^2$$

Hence we have,

$$J = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=m+1}^{d} ((X_i - \bar{X})^T U_j)^2$$
$$= \sum_{j=m+1}^{d} \frac{1}{n} \sum_{i=1}^{n} U_j^T (X_i - \bar{X})(X_i - \bar{X})^T U_j$$
$$= \sum_{j=m+1}^{d} U_j^T \left(\frac{1}{n} \sum_{i=1}^{n} (X_i - \bar{X})(X_i - \bar{X})^T \right) U_j$$

Let

$$S = \frac{1}{n} \sum_{i=1}^{n} (X_i - \bar{X})(X_i - \bar{X})^T$$

'S' is the data covariance matrix. Our mean square error is $J = \sum_{j=m+1}^{d} U_j^T S U_j$

Suppose we want to find vector U , to

$$min_U \ \ U^T S U$$
$$s.t. \ \ U^T U = 1$$

The lagrangian for the problem is $U^T SU + \lambda(1 - U^T U)$. Equating the gradient of Lagrangian to zero, $SU = \lambda U$. This means U should be an eigen vector of S. The corresponding value is: $U^T SU = U^T \lambda U = \lambda$. The minimizing U is eigen vector corresponding to least eigen value.

We want to minimize $J = \sum_{j=m+1}^{d} U_j^T S U_j$.

Since S is real symmetric, all its eigen values are real and it would have a set of orthonormal eigen vectors that span the space. So, to minimize J, we should choose, $U_{m+1},\ldots,U_d$ to be the (d−m) eigen vectors corresponding to the least (d−m) eigen values.

The vectors $U_{m+1},\ldots,U_d$ span the orthogonal complement of our desired m-dimensional space, that space has orthonormal basis $U_1,\ldots,U_m$ which are the remaining eigen vectors of S.

If $U_1,\ldots,U_d$ be the orthogonal set of eigen vectors of S, arranged in decreasing order of the corresponding eigen values.

Now our approximation is

$$\widetilde{X}_i = \sum_{j=1}^{m} (X_i^T U_j) U_j + \sum_{j=m+1}^{d} (\bar{X}^T U_j) U_j$$

We can rewrite this as

$$\widetilde{X}_i = \sum_{j=1}^{m} (X_i^T U_j - \bar{X}^T U_j) U_j + \sum_{j=1}^{d} (\bar{X}^T U_j) U_j$$
$$= \sum_{j=1}^{m} (X_i^T U_j - \bar{X}^T U_j) U_j + \bar{X}$$
$$\widetilde{X}_i - \bar{X} = \sum_{j=1}^{m} (X_i^T U_j - \bar{X}^T U_j) U_j$$

WLOG, Assume $\bar{X} = 0$ (because we can always work with mean-subtracted data).
In our representation, $\widetilde{X}_i$ is a d-dimensional vector which is in an m-dimensional subspace of $\Re^d$. We can write it as a m-component vector as

$$\widetilde{X}_i = [U_1 \ldots U_m]^T X_i = A^T X_i$$

where A is a d × m matrix whose columns are $U_1,\ldots,U_m$. We are projecting X_i onto the space spanned by the eigen vectors $U_1,\ldots,U_m$. The projections are called the principal components.

PCA and Whitening Transform

We can use PCA to find a linear transform of the feature vector so that the transformed features are uncorrelated. As earlier, let $S = \frac{1}{n}\sum_{i=1}^{n} (X_i - \bar{X})(Xi - \bar{X})^T$ be the data covariance matrix. Let $\lambda_1,\ldots,\lambda_d$ be the eigenvalues of S arranged in a decreasing order. Let $U_1,\ldots, U_d$ be the corresponding eigen vectors. Let L be a diagonal matrix with λ_i being the diagonal entries. Let $\widetilde{U}$ be a d × d matrix whose columns are U_j. We have $S\widetilde{U} = \widetilde{U} L$ Define a transformation of the data vectors given by

$$Z_i = L^{-0.5} \widetilde{U}^T (X_i - \bar{X})$$

Here we are essentially scaling each principal component by square root of the corresponding eigen value. It is easy to see that mean of Z_i is zero. The covariance matrix for the data Z_i is now given by :

$$S_Z = \frac{1}{n}\sum_{i=1}^{n} Z_i Z_i^T$$
$$= \frac{1}{n}\sum_{i=1}^{n} L^{-0.5} \widetilde{U}^T (X_i - \bar{X})(X_i - \bar{X})^T \widetilde{U} L^{-0.5}$$
$$= L^{-0.5} \widetilde{U}^T S \widetilde{U} L^{-0.5}$$
$$= L^{-0.5} \widetilde{U}^T \widetilde{U} L L^{-0.5}$$
$$= I$$

Thus, the transformed data are zero-mean, unit variance and uncorrelated.
This is the transformation we saw earlier for normalizing inputs to a neural network. This idea comes from PCA. This is called PCA whitening.

ZCA Whitening

Assume mean of X is zero.. The PCA whitening transform is $Z = L^{-0.5} \widetilde{U}^T X$. If R is any orthogonal matrix then Z_1 = RZ would also be a whitening transform:

$$E[Z_1 Z_1^T] = E[R Z Z^T R^T] = R I R^T = I$$

Thus the PCA whitening transform is not unique. If we take $R = \widetilde{U}$ then it is called ZCA whitening. PCA transform is $\widetilde{X} = \widetilde{U}^T X$. So, PCA whitening is $L^{-0.5}\widetilde{X}$. This need not be close to X in mean-square sense. Suppose we are looking for a transformation $\widetilde{X} = A^T X$ such that the output is whitened and, in addition, it is close to the original data in mean-square sense. That transform is ZCA whitening. The ZCA whitening is more useful when data are images. The ZCA whitened images look similar to the original but the PCA whitened ones may not.

High dimensional PCA

To implement PCA, we need to find eigen vectors of 'S' which is an d × d matrix. There can be situations where the feature vector dimension, d, is large (and n < d). For example, image-based pattern recognition. Here we may have d >> n. When d is large, finding eigen vectors of S can be computationally expensive. In such situations we can reformulate PCA so that we find eigen vectors of only a n × n matrix.

Recall

$$S=\frac{1}{n}\sum_{i=1}^{n}(X_i-\bar{X})(X_i-\bar{X})^T$$

Since S is sum of n rank-1 matrices, its rank can not exceed 'n'. Thus, anyway, d−n eigen values of S would be zero and hence we do not need those eigen vectors Let A be the n × d matrix whose i th row is $(X_i-\bar{X})^T$. Then, we have $S=\frac{1}{n}A^TA$.

Let U_i be an eigen vector of S for eigen value $\lambda_i > 0$.

$$\frac{1}{n}A^TAU_i=\lambda_iU_i$$

This implies

$$\lambda_i(AU_i)=A\left(\frac{1}{n}A^TAU_i\right)=\frac{1}{n}AA^T(AU_i)$$

Thus, λ_i is also an eigen value of the n × n matrix $\frac{1}{n}AA^T$ and AU_i is its corresponding eigenvector

Thus, for PCA now, we do not need to find the eigenvectors of S. It is enough to find eigenvectors of the n × n matrix $\frac{1}{n}AA^T$.

Specifically, let V_i be the eigenvector for an eigenvalue λ_i of the n × n matrix $\frac{1}{n}AA^T$. Then we have $\frac{1}{n}AA^TV_i=\lambda_iV_i$, and hence

$$\lambda_i(A^TV_i)=A^T\left(\frac{1}{n}AA^TV_i\right)=\left(\frac{1}{n}A^TA\right)(A^TV_i)$$

Thus, we get all required eigenvectors of S as A^TV_i.

Kernel PCA

PCA is essentially a linear transform of the data. We can think of a non-linear analogue through the kernel trick.

Suppose we use $\phi:\Re^d\rightarrow\Re^M$ to map the data to a new (possibly high dimensional) feature space.

Suppose we want to do PCA in $\Re^M$. We can use the Kernel trick for that.

Our data in the new space is: $\phi(X_i), i=1,\ldots,n$. Assume $\sum_i\phi(X_i)=0$, and

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

The data covariance matrix now is

$$C=\frac{1}{n}\phi(X_i)\,\phi(X_i)^T$$

Our task is to find eigen vectors of C without working in the new feature space.

The eigen vectors of C satisfy:

$$CV_i=\frac{1}{n}\sum_{j=1}^{n}\phi(X_j)(\phi(X_j)^TV_i)=\lambda_iV_i$$

Implies V_i is a linear combination of $\phi(X_j)$.
Let

$$V_i = \sum_{j=1}^{n} a_{ij}\,\phi(X_j), \quad i = 1, \ldots, M$$

Substituting for C and V_i in the eigen value equation:

$$CV_i = \lambda_i V_i$$

$$\frac{1}{n}\sum_{j=1}^{n} \phi(X_j)\,\phi(X_j)^T \sum_{p=1}^{n} a_{ip}\,\phi(X_p) = \lambda_i \sum_{p=1}^{n} a_{ip}\,\phi(X_p)$$

Premultiplying both sides by $\phi(X_l)^T$ for some l,

$$\sum_{j=1}^{n} \phi(X_l)^T\,\phi(X_j)\,\phi(X_j)^T \sum_{p=1}^{n} a_{ip}\,\phi(X_p) = n\,\lambda_i\,\phi(X_l)^T \sum_{p=1}^{n} a_{ip}\,\phi(X_p)$$

Hence

$$\sum_{j=1}^{n} \phi(X_l)^T\,\phi(X_j) \sum_{p=1}^{n} a_{ip}\,\phi(X_j)^T\,\phi(X_p) = n\,\lambda_i\,\phi(X_l)^T \sum_{p=1}^{n} a_{ip}\,\phi(X_p)$$

Now, using the kernel function we get

$$\sum_{j=1}^{n} k(X_l, X_j) \sum_{p=1}^{n} a_{ip}\,k(X_j, X_p) = n\,\lambda_i \sum_{p=1}^{n} a_{ip}\,k(X_l, X_p)$$

$$\sum_{p=1}^{n} \left(\sum_{j=1}^{n} k(X_l, X_j)\,k(X_j, X_p) \right) a_{ip} = n\,\lambda_i \sum_{p=1}^{n} a_{ip}\,k(X_l, X_p)$$

Let $K = [K_{ij}]$ where $K_{ij} = k(X_i, X_j)$. Then

$$(K^2)_{lm} = \sum_{j} K_{lj} k_{jm}$$

So we have

$$\sum_{p=1}^{n} K_{lp}^2 a_{ip} = n\,\lambda_i \sum_{p=1}^{n} K_{lp}\,a_{ip}, \forall\, l$$

We can write this in matrix notation as

$$K^2 a_i = n\,\lambda_i K\,a_i, i = 1, \ldots, n$$

where a_i is a column vector whose p^{th} component is a_{ip}.
If we know all λ_i and a_i then we have eigen values and eigen vectors of C. So, we need to find all λ_i and a_i satisfying the above.
Suppose a_i , λ_i satisfy

$$K\,a_i = n\lambda_i\,a_i, \quad i = 1, \ldots, n$$

These will then satisfy

$$K^2 a_i = n\,\lambda_i K\,a_i, \quad i = 1, \ldots, n$$

Here the difference is essentially in terms of eigen vectors of K having zero eigen values.
We can find the relevant a_i by solving

$$K\,a_i = n\,\lambda_i\,a_i$$

which is same as finding eigen values and eigen vectors of K.
From the eigen spectrum of K, we can find all the needed λ_i and a_i. That is, if b is an eigen vector of K corresponding to eigen value μ then Kb = μb, and hence one of the λ_i is μ/n and the corresponding a_i is b.
We know V_i are linear combination of $\phi(X_j)$ with weights given by a_{ij}. Hence, once we get all a_i, we can calculate all V_i.

Since a_i are eigenvectors of K, we need to know how to normalize it. This is determined by the requirement that $V_i^T V_i = 1$.
We normalize a_i using

$$1 = V_i^T V_i = \sum_{j,p=1}^{n} a_{ij} a_{ip} \phi(X_j)^T \phi(X_p) = a_i^T K a_i = n \lambda_i a_i^T a_i$$

We cannot do this normalization for eigenvectors corresponding to zero eigen value. (But we can ignore them)
For PCA in the new feature space we do not need V_i explicitly. Given an X we only need $\phi(X)^T V_i$. This is given by

$$\phi(X)^T V_i = \sum_{j=1}^{n} a_{ij} \phi(X)^T \phi(X_j) = \sum_{j=1}^{n} a_{ij} k(X, X_j)$$

Thus we can compute the PCA in the new feature space without ever needing to evaluate $\phi(X)$.
However, there is one issue that we still need to take care, In the analysis presented we have assumed that mean of $\phi(X_i)$ is zero which, in general, is not true. We cannot do a 'mean subtraction' because we do not want to compute $\phi(X_i)$. So, we need a trick for that too.
Define

$$\tilde{\phi}(X_i) = \phi(X_i) - \frac{1}{n} \sum_{j=1}^{n} \phi(X_j)$$

Let $\tilde{K}$ be the gram matrix corresponding to 'tilde' variables. We need the eigen vectors of this matrix. But we can calculate only K the gram matrix of original variables. So, we need $\tilde{K}$ in terms of K.
We have

$$\tilde{K}_{jm} = \tilde{\phi}(X_j)^T \tilde{\phi}(X_m)$$

By substituting for $\tilde{\phi}$ we get

$$\tilde{K}_{jm} = \left(\phi(X_j) - \frac{1}{n} \sum_{s=1}^{n} \phi(X_s) \right)^T \left(\phi(X_m) - \frac{1}{n} \sum_{s=1}^{n} \phi(X_s) \right)$$

Algebraic simplification gives

$$\tilde{K}_{jm} = K_{jm} - \frac{1}{n} \sum_{p=1}^{n} K_{jp} - \frac{1}{n} \sum_{l=1}^{n} K_{ml} + \frac{1}{n^2} \sum_{p,l=1}^{n} K_{lp}$$

We can write this in matrix notation as

$$\tilde{K} = K - A_n K - K A_n + A_n K A_n$$

where A_n is a n × n matrix all of whose entries are $\frac{1}{n}$.
Thus, we can calculate $\tilde{K}$ from the data gram matrix K. From the eigen vectors of $\tilde{K}$ we can now find the PCA in the new feature space.