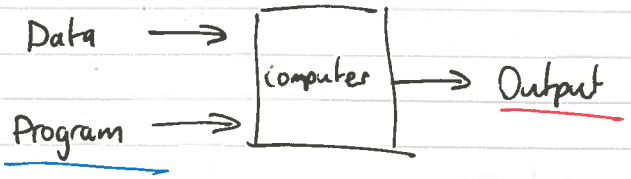
MACHINE LEARNING FOR INTELLIGENT SYSTEMS

CORNELL UNIVERSITY

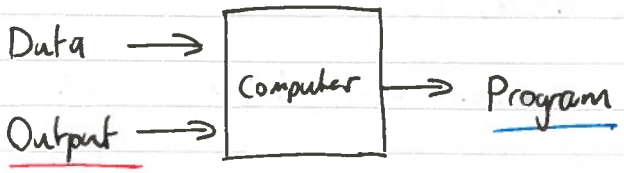
TABLE OF CONTENTS

LECTURE 1 - SUPERVISED LEARNING SETUP

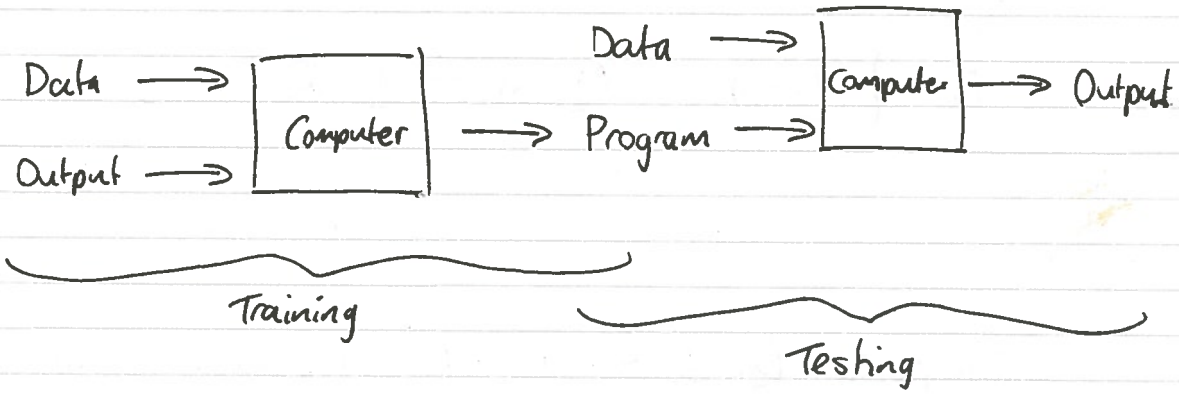
Traditional Computer Science:



Machine Learning:



Setup:



Machine Learning formally defined in 1997, summarised as "Algorithms that improve on some task with experience."

Difference between machine learning (ML) and AI
- ML is bottom up, AI is top down.
- ML focuses on practical smaller goals,
- ML based on statistics and optimisation, not logic.

Applications:

- Spam filter
- Web search

Types of Learning:

- Supervised: given labeled examples, find the correct prediction of an unlabeled example.
- Unsupervised: given data, try to discover similar patterns, structure, sub-spaces.
- Reinforced: try to learn from delayed feedback.

Setup:

$$D = \{(\underline{x}_1, y_1), \ldots, (\underline{x}_n, y_n)\} \subseteq \mathbb{R}^d \times C$$

($\subseteq$: is a subset or identical set)

- features: $\underline{x}_i$
- output: y_i
- $\mathbb{R}^d$: d-dimensional feature space
- C: label space

- data points $(\underline{x}_i, y_i)$ are drawn from some unknown distribution $P(x,y)$
- We want to learn a function h such that for a new pair $(\underline{x}, y) \sim P(x,y)$ $h(\underline{x}) = y$ with a high probability.

Examples of Label Space

Binary classification	$C = \{0,1\}$ or $\{-1,1\}$	→ Yes or No, Male or Female
Multi-class	$C = \{1,2,3,\ldots,k\}$	→ Science, Sport, Politics ...
Regression	$C = \mathbb{R}$	→ Temperature, height.

Examples of feature vectors

- Patient Data in a hospital. eg.

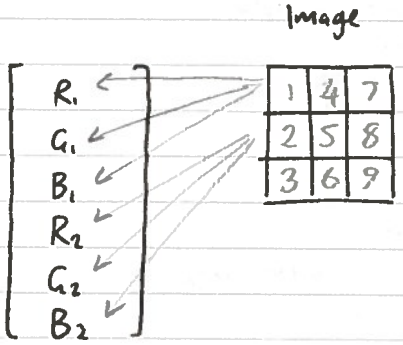
$$\underline{x}_i = \begin{bmatrix} 1 \\ 62 \\ 183 \\ \dots \\ \dots \end{bmatrix} \quad \begin{matrix} \text{male/female} \\ \text{age} \\ \text{height in cm} \end{matrix}$$

- Text Document in bag-of-words format. Count the frequency of occuring words in a text document.

$$\underline{x}_i = \begin{bmatrix} 1 \\ \dots \\ 0 \end{bmatrix} \quad \begin{matrix} \text{"ant" frequency} \\ \\ \text{"zebra" frequency} \end{matrix}$$

- sparse feature sparse

- Images



- for 7 megapixel camera, this gives 21 million dimensions
- Very dense feature space

Hypothesis Class

Before we can find a function h, we need to what type of function we are looking.

By choosing the hypothesis class, we are encoding important assumptions about the type of problem we are trying to learn.

There is no best algorithm, there are trade offs for everything!

Loss Functions

Lower is Better, Always non-negative

Zero/One Loss: $L_{0/1}(h, D) = \frac{1}{n} \sum_{(x_i, y_i) D} \delta[h(x_i) \neq y_i]$

counts the number of times the function misclassifies.

1 if $h(x_i) \neq y$
0 otherwise

D = dataset

Squared Loss: $L_{sq}(h, D) = \frac{1}{n} \sum_{i} (h(x_i) - y_i)^2$

Absolute Loss: $L_{Ab}(h, D) = \frac{1}{n} \sum_{i} |h(x_i) - y_i|$

Choosing between squared loss and absolute loss is important. With squared loss, the loss grows quadratically with absolute mispredicted amount. Absolute loss is more suited for noisy data when some mispredictions are unavoidable and shouldn't dominate the loss.

Generalisation

$h \in H$ s.t. $\forall (\underline{x}, y) \sim P \quad h(x) \approx y$

we want a function h from a set of all functions H, such that any x and y drawn from distribution P gives $h(x) \approx y$

Given our function h, what loss would we expect to find if we apply it to a new dataset given values are from the same distribution?

$E[L(h_i(\underline{x}, y))]_{(\underline{x}, y) \sim P}$ ← the problem is: we don't know $P(x, y)$

The best function will find the solution to...

$h = \text{argmin}_{h \in H} L(h)$

Train / Test Splits

How do we estimate the expected loss?

As soon as we get our data, we split it into a TEST and TRAINING data set. We keep the TEST data set seperate until we have a model. Both TRAINING and TEST are from the same distribution $P \sim (x, y)$

Then we can calculated the expected loss on the TEST set.

LECTURE 2 - K-NEAREST NEIGHBOURS

we have a function $h(\underline{x}_i) = y_i$,

where the loss function is described as $l(h_i, D)$

if we test this function on a new dataset where information is from the same distribution, we calculate the expected loss as

$E[l(h_i(\underline{x}_i, y_i))]_{(x,y)\sim P}$ if this is low, that's great!

But we can't compute this as we don't know what P is. We have to estimate from TEST dataset.

TRAINING = D_{TR}, TEST = D_{TE}

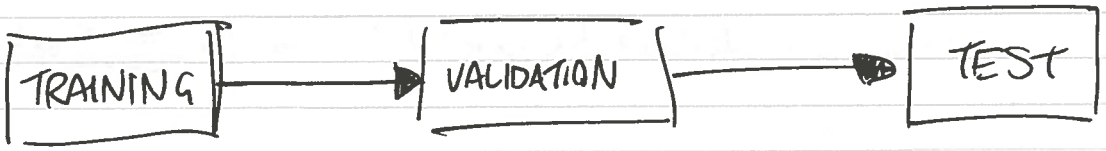
How to split data into training and test datasets?

- If there is a temporal aspect, divide by time.
- If data is IID (independent and randomly distributed), they can be shuffled randomly.

If you get this right, a test dataset is a great way to estimate the expected loss.

Sometimes a training set is split further into a VALIDATION set.

A validation dataset is useful if you want to test multiple models, but don't want the TEST dataset to influence the decision on the best one.



Test all models on validation, and choose best.

Apply best to TEST dataset.

This avoids training to the test set, which is meant to remain independent and unseen until an algorithm is decided

You can only test your model on the test set ONCE!

Loss function on test set $L(h, D_{TE})$

$$= \frac{1}{n} \sum_{i=1}^{n} l(h; (x_i, y_i))$$

is equal to $E\left[l(h, (x, y))\right]_{(x,y) \sim P}$ as $n \to \infty$

This is because of the "Weak Law of Large Numbers" or "Bernoulli's Theorem".

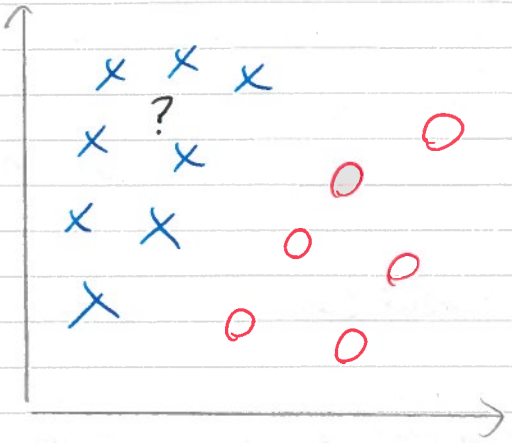
This states that as the number of samples converges towards infinity of a IID, the sample means converges to the population mean".

How do we work out what h to use from H?

- We have to make some assumptions in order to create a model.
- Each algorithm makes different assumptions.
- There is no <u>one</u> algorithm which works for all problems. "NO FREE LUNCH" theorem.

- Assumptions should be made to best suit the data.
- Which algorithm leverages these assumptions?

K-Nearest Neighbours



This graph describes a classification problem; crosses and circles. Two arbitrary features are plotted, one on the x and one on the y.

If a point is placed at the question mark, what is it likely to be? Answer: most likely a cross because its close to other crosses. This is the basis of k-nearest neighbour, and the assumption its making: "data points that are similar, have similar labels".

Algorithm:

- Define parameter n which describes the number of surrounding points to consider.
- Find the n closest points to our new datapoint
- Count how many from each class are in the list of n closest points.
- The class with the most points wins!

Formally:

test point x

$S_x \in D$ s.t. $|S_x| = k$ ← S_x is a subset of D with k elements

$\forall (x', y') \in D \setminus S_x$ ← for every point that is in D but not S_x, the distance between from test point x must be greater than any distance between x and any point in S_x.

$$dist(x, x') \geq \max_{(x'', y'') \in S_x} dist(x, x'')$$

k-nearest neighbours is only as good as the distance metric.

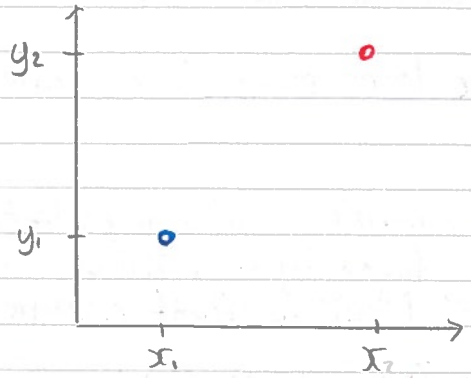
Minkowski Distance $dist(x,y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$

p can be tuned to change how distance is calculated, ∴ very general and flexible.

$p = 1 \Rightarrow$ Manhattan Distance

$p = 2 \Rightarrow$ Euclidean Distance

$p \to \infty \Rightarrow$ Chebyshev Distance



$p=2$ $\sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}$

$p=1$ $|x_1-x_2| + |y_1-y_2|$

$p \to \infty$ $\max(|x_1-x_2|, |y_1-y_2|)$

Bayes Optimal Classifier

Try to determine the most probable hypothesis given the data.

Suppose our hypothesis space H has 3 functions h_1, h_2, h_3

$P(h_1|D) = 0.4$, $P(h_2, D) = 0.3$, $P(h_3, D) = 0.3$

- What is the MAXIMUM A POSTERIOR estimation?

  Answer: h_1 because it has the greatest individual probability

- for x, suppose $h_1(x) = +1$, $h_2(x) = -1$, $h_3(x) = -1$. What is the most probable classification of x?

  -1 because $P(+1|x) = \underline{0.4}$ but $P(-1|x) = 0.3 + 0.3 = \underline{0.6}$

Therefore the most probable classification is not the same as the prediction of the MAP hypothesis. So which do we use?

The Bayes optimal classifer is defined as the label produced by the most probable classifer

$$\underset{y}{\operatorname{argmax}} \sum_{h_i \in H} P(y|h_i) P(h_i|D)$$

However, computing this is "HOPELESSLY INEFFICIENT".

And yet an interesting theoretical concept because no other classification method can outperform this method (on average) using the same hypothesis space and prior knowledge.

∴ This provides a highly informative lower bound of error rate.

Additionally, the upper bound of error rate is also important to know. This upper bound would be found by a classifier which essentially always predicts the same label (constant classifier). independent of any features.

The best constant in classification is the most common label in the training set. This is equivalent to the k-NN classifier as $k = N$ where n is the number of points in the training set.

The "Best Constant Classifier" is important for debugging as we should always be able to show that our model outperforms this on the TEST set.

The error is written as

$$\epsilon_{\text{bayesOpt}} = 1 - P(h_{opt}(x)|x) = 1 - P(y^*|x)$$

1 - NN Convergence Proof

Described by Cover and Hart in 1967:

> "As $n \to \infty$, the 1-NN error is no more than twice the error rate of the Bayes Optimum Classifier."

Let x_{NN} be the nearest neighbour to our test point x_T.
As $n \to \infty$, $dist(x_{NN}, x_T) \to 0$, i.e $x_{NN} \to x_T$.

We ask our model to return the value of x_{NN}. What is the probability that we get an incorrect classification?

$$\epsilon_{NN} = P(y^*|x_T)(1 - P(y^*|x_{NN})) + P(y^*|x_{NN})(1 - P(y^*|x_T))$$

Probability that x_T has the label y^*, but k-NN classifies as NOT y^*.

Probability that x_T doesn't have the label y^*, but kNN classifies as y^*

as $n \to \infty$, $x_T \to x_{NN}$ $\therefore$ $P(y^*|x_T) = P(y^*|x_{NN})$
as they are probabilities, $P(y^*|x_T) \leq 1$ and $P(y^*|x_{NN}) \leq 1$

~~$\epsilon_{NN} = P(y^*|x_T) - P(y^*|x_T)(P(y^*|x_{NN}) + P(y^*|x_{NN})$~~
~~$- P(y^*|x_{NN})P(y^*|x_T)$~~

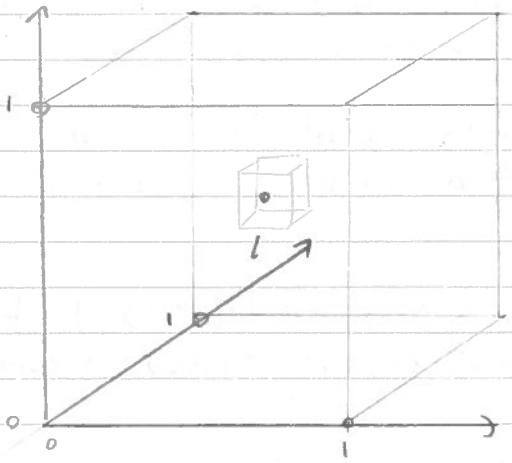
$$\epsilon_{NN} = \underbrace{P(y^*|x_T)}_{\leq 1}(1 - P(y^*|x_{NN})) + \underbrace{P(y^*|x_{NN})}_{\leq 1}(1 - P(y^*|x_T))$$

$$\epsilon_{NN} \leq (1 - P(y^*|x_{NN})) + (1 - P(y^*|x_T))$$
$$\leq 2(1 - P(y^*|x_T))$$
$$\leq 2\epsilon_{Bayes\ Opt}$$

Curse of Dimensionality

- The kNN classifier makes the assumption that similar points share similar labels.
- In high dimensional space, points drawn from a probability distribution tend not to be close together.

Suppose we have a unit cube and we draw points inside uniformly at random. We will investigate how much space the k nearest neighbours of a test point inside this cube will take up.



Formally: imagine a unit cube $[0,1]^d$ where d is the number of dimensions.

All training data is sampled uniformly within the cube $\forall i, x_i \in [0,1]^d$, and we are considering the 10 nearest neighbours to such a test point

Let l be the edge length of the smallest hyper-cube that contains all k-Nearest Neighbours to a test point.

$l^d \approx \frac{k}{n}$ and $l \approx \left(\frac{k}{n}\right)^d$.

So as $d \gg 0$, almost the entire space is needed to find the 10 nearest neighbours.

Distances to hyperplanes

- The distance between 2 randomly chosen data points increases drastically with dimensionality.

In 2-dimensions, a point can have distances of Δx and Δy from a test point $\therefore$ distance $= \sqrt{\Delta x^2 + \Delta y^2}$

In 3-dimensions, a point can have distances of Δx, Δy and Δz from a test point $\therefore$ distance $= \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$

therefore $\text{distance}_{2D} \leq \text{distance}_{3D}$

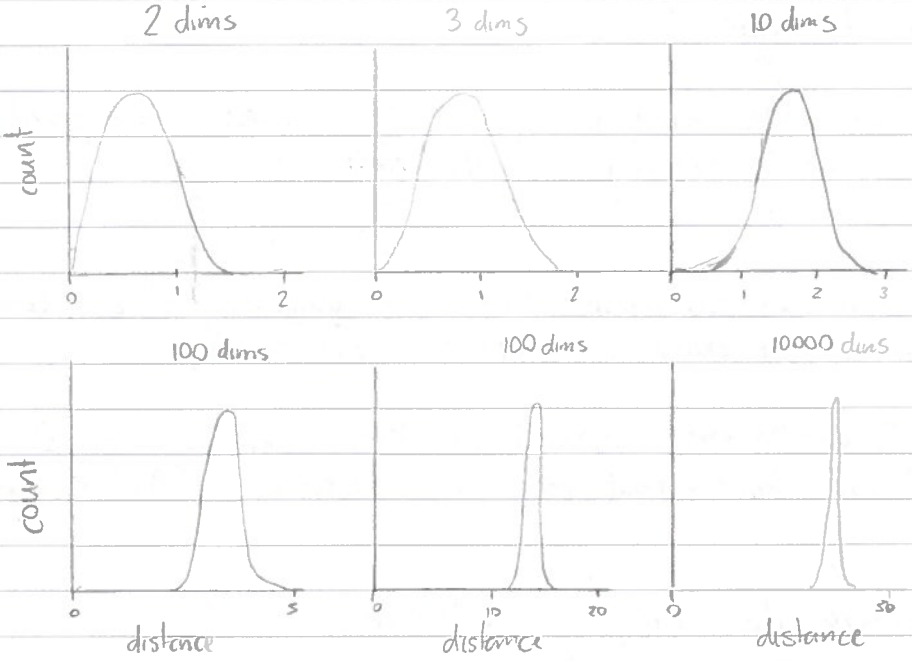
- However, distances of a point to a hyperplane remains unchanged as dimensionality increases.

This is because the hyperplane is orthogonal to the next higher dimension.

Data with low dimensional structure

- Although a dataset may have a high dimensionality, not all is lost. One example is face recognition from images.
- Although an image of a face could contain millions of pixels, the face could be described with less than 50 attributes; eg. male/female, blond/dark hair, etc.
- "Low intrinsic dimensionality" data confined to smaller sub-space or sub-manifold

A manifold is a surface which is locally euclidian.

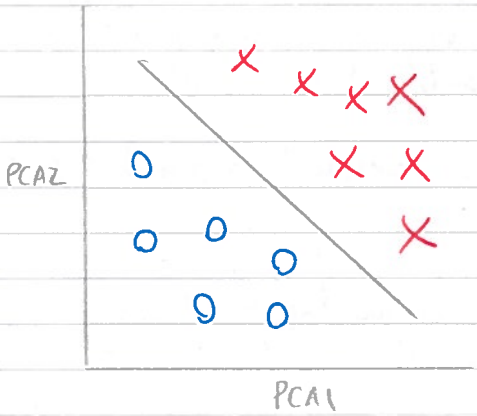


As dimensionality gets too high, there are essentially no neighbours that are close, which is the opposite of the assumption required for k-NN to work

LABELS CHANGE QUICKER THAN SAMPLE DENSITY

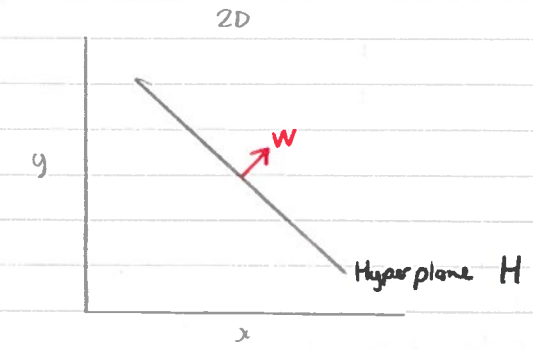
LECTURE 3 - THE PERCEPTRON

The problem of k-Nearest Neighbour is the computation time when n gets very large, as the distance needs to be calculated between the test point and the training points.



- Binary classification
- Assumes data is linearly seperable, which is highly likely in high dimensions.
- Find the function for this HYPER PLANE.

How do we define a hyper-plane?



We use vector $\underline{w}$ and offset B.

$$H = \{x : w^T x + B = 0\}$$

B is a bias term, which shifts the hyperplanes such that it doesn't always go through the origin

Given a new random point, we can determine which side of the hyper-plane the point lies and then classify. The computational time is the same for any number of training points.

$$h(x_i) = \text{sign}(\underline{w}^T x_i + B)$$

Learning both $\underline{w}$ and B is a pain. But there is a trick to absorb it into the equation.

Instead of x_i, use $\begin{pmatrix} x_i \\ 1 \end{pmatrix}$

Instead of $\underline{w}$, use $\begin{pmatrix} \underline{w} \\ B \end{pmatrix}$

therefore $H = \{x : \underline{w}^T x = 0\}$ no B!

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

- Our classes are $+1$ and -1
- If our point is on the right of the hyper plane, we want the class to be $+1$
- If our point is on the left of the plane, we want our class to be -1

$$\left.\begin{array}{lll} y = +1 & \Rightarrow & w^T x > 0 \\ y = -1 & \Rightarrow & w^T x < 0 \end{array}\right\} \quad yw^T x > 0 \Leftrightarrow \text{classified correctly}$$

[This only works if our binary class is labelled -1 and 1, not 0 and 1.]

Perceptron Convergence

- Data must have binary classes
- Binary classes must be linearly separable.

$$\exists w^* \text{ s.t. } y_i(x^T w^*) > 0 \quad \forall (x_i, y_i) \in D$$

"There exists a value of w^* such that for every x-y pair, $y_i(x^T w^*) > 0$."

If there exists one hyperplane to separate the data, there exists an infinite amount of hyperplanes. The hyperplane can be scale by any positive value to give the same classification.

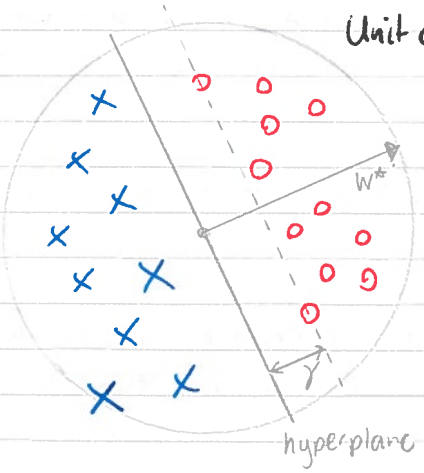
$$w^{*1} = \alpha w^*$$

however, we can limit the number to one by insisting $\|w^*\| = 1$

We can also scale x by α to cause no difference. So we must insist that...

$$\forall_i \ \|x_i\| \leq 1$$

From these rules we can illustrate the problem as



This means all data points can fit inside a unit circle because
- $\|w^*\| = 1$
- we divided a x vectors by the maximum distance.

γ is known as the margin, which is the distance between the hyperplane and the closest point. $\gamma = \min_{(x_i, y_i) \in D} |x_i^T w^*| > 0$

Theorem: If the above is all true, then the perceptron algorithm makes at most $1/\gamma^2$ mistakes.

Proof: w is vector during optimisation
w^* is the hyperplane

During an update, we look at $w^T w^*$ and $w^T w$.

$w^T w^*$ can grow if w is simply scaled, but we want it to grow because it converges to w^*. How?

We also look at $w^T w$, which also grows as w is scaled, but doesn't grow it is converging to w^*.

Therefore is $w^T w^*$ grows, but $w^T w$ doesn't, then we are converging to w^*.

Update $w \Leftarrow (w + yx)$ because we misclassified eg. $yw^T x \leq 0$

$\boxed{w^T w^*}$ $\quad (w+yx)^T w^* = w^T w^* + yw^{T*} x$

W^{*T} is the correct hyperplane, so it alway correctly classifies
$\therefore yw^{*T} x_m > 0$ and $\geq \gamma$ because.

$W^{*T} x \quad \underbrace{x^T w^*}_{\gamma} = W^{*T} x$

$\therefore \quad w^T w^* + yw^{*T} x \geq w^T w^* + \gamma$

Thus means that if we make an update, the inner product of of the hyperplane we're looking for with the hyper plane we want to find grows by at least γ.

$\boxed{w^T w}$ $\quad (w+yx)^T (w+yx) = w^T w + \underbrace{2yw^T x}_{<0} + \underbrace{y^2}_{1} \underbrace{x^T x}_{\leq 1}$

$\leq w^T w + 1$

Thus means that for each update, $w^T w$ grows by at most 1

After M updates... Cauchy Schwartz inequality

$$M\gamma \leq w^T w^* = |w^T w^*| \leq \|w\| \underbrace{\|w^*\|}_{=1} = \|w\| = \sqrt{w^T w} \leq \sqrt{M}$$

$$\therefore M\gamma \leq \sqrt{M} \Rightarrow M^2\gamma^2 \leq M \Rightarrow M \leq \frac{1}{\gamma^2}$$

Therefore, the number of updates is bound. Finite number of steps, and is only dependent on the distance of true hyperplane to closest point.

LECTURE 4 ESTIMATING PROBABILITIES FROM DATA

Remember Bayes optimal classifier: If we are provided with $P(x,y)$, we can predict the most likely label for x, formally $\text{argmax}_y (P(y|x))$. It is worth considering if we can estimate $P(x,y)$ from the training data.

If this is possible, we could the use the Bayes optimal classifier in practise on our estimate of $P(x,y)$

Many supervised learning algorithms can be viewed as estimating $P(x,y)$. Generally they fall into two categories

- Estimating $P(x,y) = P(x|y)P(y)$ GENERATIVE
- Estimating $P(y|x)$ DISCRIMINATIVE

COIN TOSS

You find a coin and its valuable. What is the probability that it comes up heads when you toss it? It is tossed n times and you get the sequence

$$D = \{H, T, T, H, H, H, T, T, T, T\}$$

Based on this sample, how would we estimate $P(H)$?

$n_H = 4$, $n_T = 6$ so intuitively

$$P(H) \approx \frac{n_H}{n_H + n_T} = \frac{4}{10} = 0.4$$

Can we derive this more formally?

MAXIMUM LIKLIHOOD ESTIMATION (MLE)

Two assumptions are made for MLE:

- Make an explicit assumption about what type of distribution your data was sampled from.
- Set the parameters of this distribution so that the data you observed is as likely as possible

For the coin flipping example, we assume the distribution is binomial, and it has 2 parameters: n and θ

n = number of independent random events (coin tosses)

θ = probability of positive outcome (coin lands heads)

We can then write the coin toss example formally as

$$P(D|\theta) = \binom{n_H + n_T}{n_H} \theta^{n_H} (1-\theta)^{n_T}$$

This computes the probability that we would observe exactly n_H heads and n_T tails if a coin was tossed $n = n_H + n_T$ times, and its probability of coming up heads.

So we want to find

$$\hat{\theta}_{MLE} = \underset{\theta}{\operatorname{argmax}}\, P(D;\theta)$$

This is basically saying, what parameters will give us the best chances of observing what we observe from the data.

We can attempt to solve this maximisation problem with a simple 2 step procedure...

- Plug in all terms for the distribution and take the log.
- Compute it's derivative and equate to zero.

Because log is an increasing function, we can find the argmax of the log of the function. This makes maths easier.

$$\underset{\theta}{\text{argmax}} \ \log\left[\binom{n_H+n_T}{n_H}\theta^{n_H}(1-\theta)^{n_T}\right]$$

Taking the log also helps when adding and subtracting small probabilities. Truncation and floating point errors are reduced NUMERICAL STABILITY

$$\underset{\theta}{\text{argmax}} \ \underbrace{\log\left[\binom{n_H+n_T}{n_H}\right]}_{\text{vanishes in next step.}} + n_H \log\theta + n_T \log(1-\theta)$$

differentiate with respect to θ and set to zero

$$\frac{n_H}{\theta} - \frac{n_T}{(1-\theta)} = 0 \Rightarrow \frac{n_H}{\theta} = \frac{n_T}{(1-\theta)}$$

$$\theta = \frac{n_H}{n_H+n_T}$$

- MLE gives the explanation of the data you observed
- If n is large and your model / distribution is correct, MLE will find the true parameters
- MLE can overfit if the data is small
- If you have the incorrect model MLE will be very wrong!

Coin Toss with Prior Knowledge

Assume you have a hunch that θ is close to 0.5, but your sample size is small so you don't trust your earlier MLE estimate.

We can add m imaginary throws that would result in θ' (eg. $\theta = 0.5$). Add m heads and m tails to the data

$$\hat{\theta} = \frac{n_H + m}{n_H + n_T + 2m}$$

For large n, the change is insignificant, but for small n it is incorporating a prior belief of what θ should be.

+1 smoothing simply sets $m = 1$

This helps avoid degenerative case where things can get divided by zero.

By doing +1 smoothing, we change from MLE to Maximum A Posteriori estimation (MAP) (ie. Frequentist vs. Bayesian)

$P(D;\theta)$ is a frequentist notation. θ in this case is a parameter.

$P(D|\theta)$ is Bayesian notation. θ in this case is now a random variable, and as such can be drawn from a distribution $P(\theta)$

$P(D|\theta)$ = likelihood ← THIS WHAT MLE ESTIMATES

$P(\theta)$ = Prior

$P(\theta|D)$ = Posterior ← THIS IS WHAT MAP ESTIMATES

Likelihood $P(D|\theta)$ which parameter makes our data most likely?

Posterior $P(\theta|D)$ given we have our data what parameter is likely?

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$$ Bayes Rule

A natural choice for the prior is the Beta distribution, as this cannot be negative and gives values between 0 and 1

$$P(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}$$

B is just a normalisation constant to make sure everything sums to one.

~~$P(\theta|D) \propto P(D|\theta)P(\theta)$~~

Find the value of θ that maximises the posterior distribution $P(\theta|D)$

$$\hat{\theta}_{MAP} = \underset{\theta}{\text{argmax}}\ P(\theta|D)$$

$$= \underset{\theta}{\text{argmax}}\ \log(P(D|\theta)) + \log(P(\theta))$$

In our coin flipping example...

$$\hat{\theta}_{MAP} = \underset{\theta}{\text{argmax}}\ \underbrace{n_H \log\theta + n_T \log(1-\theta)}_{\log(P(D|\theta))} + (\alpha-1)\log\theta + (\beta-1)\log(1-\theta)$$

we have removed the normalisation term $B(\alpha,\beta)$

$$= \underset{\theta}{\text{argmax}} \; (n_H + \alpha - 1)\log\theta + (n_T + \beta - 1)\log(1-\theta)$$

differentiating and setting equal to 0 gets

$$\hat{\theta}_{MAP} = \frac{n_H + \alpha - 1}{n_H + n_T + \alpha + \beta - 2}$$

This shows the bayesian approach does the same thing as smoothing.

- MAP is the same thing as MLE with $\alpha-1$ imaginary heads and $\beta-1$ imaginary tails
- As $n \to \infty$, $\hat{\theta}_{MAP} \to \hat{\theta}_{MLE}$, $\alpha-1$ and $\beta-1$ became negligable
- MAP is a great estimator if a prior belief is available
- If n is small, MAP can be very wrong if the belief is wrong.

A "True" Bayesian approach would be to use the Posterior Predictive distribution directly to make a prediction about label Y of a test sample with features X

$$P(Y|D,X) = \int_\theta P(Y,\theta|D,X)\,d\theta = \int_\theta P(Y|\theta,D,X)P(\theta|D)\,d\theta$$

MLE: $\theta = \underset{\theta}{\operatorname{argmax}}\ P_\theta(D)$ θ = parameter

MAP: $\theta = \underset{\theta}{\operatorname{argmax}}\ P(\theta \mid D)$ θ = random variable

Our training data consists of the set

$$D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$$

drawn from some unknown distribution $P(x,y)$.

Because all pairs are sampled I.I.D, we obtain

$$P(D) = P((x_1,y_1), \ldots, (x_n,y_n)) = \prod_{\alpha=1}^{n} P(x_\alpha, y_\alpha)$$

If we do not have enough data, we could estimate $P(X,Y)$.

If we are primarily interested in predicting the label y from the features x, we may estimate $P(Y \mid X)$ directly instead of $P(X,Y)$. Then we can use the Bayes Optimal Classifier for a specific $\hat{P}(y \mid x)$ to make predictions

How can we estimate $\hat{P}(y \mid x)$?

$$\hat{P}(y) = \frac{\sum_{i=1}^{n} I(y_i = y)}{n} \qquad \hat{P}(x) = \frac{\sum_{i=1}^{n} I(x_i = x)}{n}$$

$$\hat{P}(y,x) = \frac{\sum_{i=1}^{n} I(x_i = x \wedge y_i = y)}{n} \qquad \text{therefore}$$

$$\hat{P}(y \mid x) = \frac{\hat{P}(y,x)}{P(x)} = \frac{\sum_{i=1}^{n} I(x_i = x \wedge y_i = y)}{\sum_{i=1}^{n} I(x_i = x)}$$

There is a big problem with this method. The MLE estimate is only good if there are many training vectors with the same identical features as x. In high dimensional space (or continuous x), this never happens!

NAÏVE BAYES

We can approach this dillema using a simple trick, and an assumption. The trick is to estimate $P(y)$ and $P(x|y)$ instead

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

We can estimate $P(x)$ and $P(y)$, but how do we get $P(x|y)$.

We make the assumption that feature values are independent given the label.

$$P(x|y) = \prod_{\alpha=1}^{d} P(x_\alpha|y)$$

No feature value is dependent on any other feature value. This not the best assumption, but it allows us to solve it.

The bayes classifier is written as ...

$$h(x) = \underset{y}{\operatorname{argmax}} \left(P(y|x)\right)$$

$$= \underset{y}{\operatorname{argmax}} \frac{P(x|y)P(y)}{P(x)} = \underset{y}{\operatorname{argmax}} P(x|y)P(y)$$

$$= \underset{y}{\operatorname{argmax}} \prod_{\alpha=1}^{d} \left[P(x_\alpha|y)\right]P(y)$$

$$= \underset{y}{\operatorname{argmax}} \sum_{\alpha=1}^{d} \log\left(P(x_\alpha|y)\right) + \log(P(y))$$

This is easy to estimate as we only need to consider 1 dimension.

Estimating $P([x]_\alpha | y)$

There are three cases we can consider.

Case ① : Categorical features

$$[x]_\alpha \in \{f_1, f_2, f_3, \ldots, f_{K_\alpha}\}$$

Each feature falls into one of K_α categories. If this were a binary case, $K_\alpha = 2$.

Model of $P(x_\alpha | y)$:

$$P(x_\alpha = j | y = c) = [\theta_{jc}]_\alpha \quad \text{and.}$$
$$\sum_{j=1}^{K_\alpha} [\theta_{jc}]_\alpha = 1$$

where $[\theta_{jc}]$ is the probability of feature α having the value j, given that it has label c.

We can use MLE to estimate this parameter.

$$[\theta_{jc}] = \frac{\sum_{i=1}^{n} I(y_i = c) I(x_{i\alpha} = j) + l}{\sum_{i=1}^{n} I(y_i = c) + l K_\alpha}$$

Smoothing

where $x_{i\alpha} = [x_i]_\alpha$ and l is a smoothing parameter. If $l > 0$, this is MAP. If not, this is MLE. If $l = +1$, we get a special case called LaPlace smoothing.

Case ② : Multinomial Features

If features don't represent categories but counts, we need to use a different model.

$$x_\alpha \in \{0, 1, 2, 3, \ldots, m\} \text{ and } m = \sum_{\alpha=1}^{d} x_\alpha$$

each feature represents a count and m is the length of the sequence.

Model

$$P(x \mid m, y=c) = \frac{m!}{x_1! x_2! \ldots x_d!} \prod_{\alpha=1}^{d} (\theta_{\alpha c})^{x_\alpha}$$

where $\theta_{\alpha c}$ is the probability of selecting x_α and $\sum_{\alpha=1}^{d} \theta_{\alpha c} = 1$

we can estimate $\theta_{\alpha c}$ using

$$\hat{\theta}_{\alpha c} = \frac{\sum_{i=1}^{n} I(y_i = c) x_{i\alpha}}{\sum_{i=1}^{n} I(y_i = c) \sum_{\beta=1}^{d} x_{i\beta}} \rightarrow m$$

where $\sum_{\beta=1}^{d} x_\beta$ is the nuber of words in a document.

The numberator sums up all counts for feature x_α and the denominator sums up all counts of all features across multiple data points. eg

$$\frac{\#\text{ of times word } \alpha \text{ appears in all spam emails}}{\#\text{ of words in all spam emails combined}}$$

We can also add smoothing to estimate parameter $\hat{\theta}_{\alpha c}$

$$\hat{\theta}_{\alpha c} = \frac{\sum_{i=1}^{n} I(y_i = c) x_{i\alpha} + 1}{\sum_{i=1}^{n} I(y_i = c) \sum_{\beta=1}^{d} (x_i)_\beta + d}$$

where d is the number of words in the dictionary. This assumes we see each word at least once.

Case ③ : Continuous Features

$$x_\alpha \in \mathbb{R}$$

each feature can take on any real value

Model:

$$P(x_\alpha | y) = N(\mu_{\alpha c}, \sigma^2_{\alpha c}) = \frac{1}{\sqrt{2\pi\sigma^2_{\alpha c}}} e^{-\left(\frac{(x_\alpha - \mu_{\alpha c})^2}{2\sigma_{\alpha c}^2}\right)}$$

This is the Gaussian distribution. We estimate the parameters of the distributions for each dimension and class independently.

Gaussian distributions only have two parameters, the mean and variance. The mean $\mu_{\alpha c}$ is estimated by the average feature value of dimension α from all samples with label y.

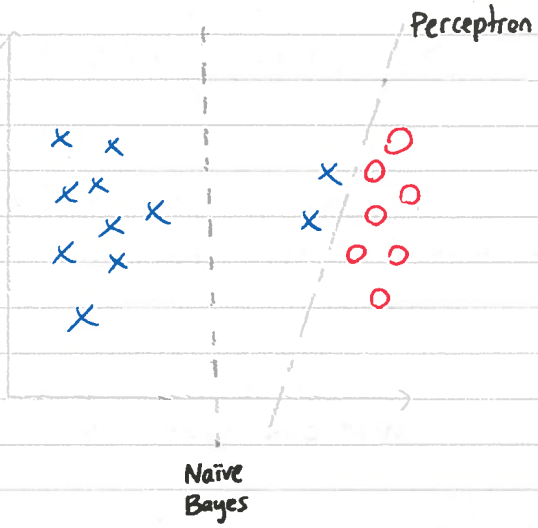
$$\mu_{\alpha c} \leftarrow \frac{1}{n_c} \sum_{i=1}^{n} I(y_i = c) x_{i\alpha}$$

$$\sigma^2_{\alpha c} \leftarrow \frac{1}{n_c} \sum_{i=1}^{n} I(y_i = c)(x_{i\alpha} - \mu_{\alpha c})^2$$

$$n_c = \sum_{i=1}^{n} I(y_i = c)$$

Naïve Bayes is a linear classifier, and therefore leads to a linear decision boundary in many cases.

We found that the Perceptron will always find a separating hyperplane if one exists. However, with Naive bayes this is not always the case.



Above is an example where a linear classifier exists to achieve perfect separation; the Perceptron can find it but the Naive Bayes doesn't. This is because the two blue X's are more likely to come from the cluster of red circles if the points are projected onto the x-axis.

Showing Naive Bayes is a linear classifier...

$$P(y|x) \propto P(x|y)P(y)$$

Suppose we have a multinomial with two classes: +1 and -1

$$\text{If } P(y=+1|x) > P(y=-1|x)$$

then a +1 class is predicted. Using the equation above, we can rewrite this as...

$$P(x|y=+1)P(y=+1) > P(x|y=-1)P(y=-1)$$

We can use the form of $P(x|y)$ that was given for a multinomial distribution.

$$P(y=+1)\underbrace{\frac{m!}{x_1!x_2!\dots x_d!}}_{\text{cancel}}\prod_{\alpha=1}^{d}\theta_{\alpha(+1)}^{x_\alpha} > P(y=-1)\underbrace{\frac{m!}{x_1!x_2!\dots x_d!}}_{\text{cancel}}\prod_{\alpha=1}^{d}\theta_{\alpha(-1)}^{x_\alpha}$$

$$P(y=+1)\prod_{\alpha=1}^{d}\theta_{\alpha(+1)}^{x_\alpha} > P(y=-1)\prod_{\alpha=1}^{d}\theta_{\alpha(-1)}^{x_\alpha}$$

Now we take the log to allow easier calculation.

$$\log(P(y=+1)) + \sum_{\alpha=1}^{d} x_\alpha \log(\theta_{\alpha(+1)}) > \log(P(y=-1)) + \sum_{\alpha=1}^{d} x_\alpha \log(\theta_{\alpha(-1)})$$

Collect the terms on the left hand side...

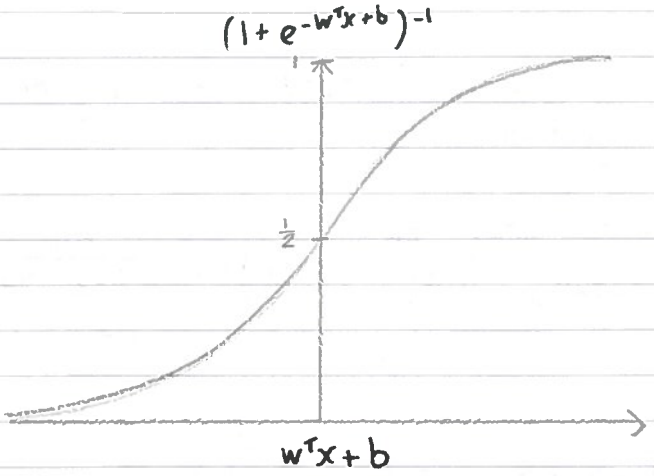
$$\underbrace{\log(P(y=+1)) - \log(P(y=-1))}_{b} + \sum_{\alpha=1}^{d} x_\alpha \underbrace{\left(\log\theta_{\alpha(+1)} - \log\theta_{\alpha(-1)}\right)}_{W_\alpha} > 0$$

$$\Rightarrow W^T x + b > 0 \qquad \text{LINEAR CLASSIFIER}$$

LOGISTIC REGRESSION

We can show that if the Naive bayes approach is used for a Gaussian model, we get

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



This is demonstrating the probability of a point falling into class +1 given the distance from the hyperplane. If w^Tx+b is very positive, it means that the point is well away from the hyperplane on the +1 side. If it is very negative, it means that the point is well away from the hyperplane on the -1 side, giving a low probability.

The Naive bayes approach first estimates the distribution of the data, in order to then estimate the hyperplane w^Tx+b. But if every gaussian distribution collapses to the formular $(1+e^{-w^Tx+b})^{-1}$, can we just predict w^Tx+b directly? This approach is called Logistic Regression, and is the discriminative counterpart to Naive Bayes.

We make little assumptions on $P(x_i|y)$, eg. it could be Gaussian or multinomial. Ultimately, it doesn't matter because we estimate w and b directly with MLE or MAP to maximise the conditional likelihood of $\prod_i P(y_i|x_i; w,b)$.

From here we assume that b is absorbed into w (as in page 15).

Maximum Likelihood Estimation

In MLE, we choose parameters that maximise the conditional likelihood. The conditional data likelihood $P(y|X,w)$ is the probability of the observed values $y \in \mathbb{R}^n$ in the training data conditioned on the feature values X. ($X = [x_1, x_2, x_3, \ldots, x_n] \in \mathbb{R}^{d \times n}$) We choose parameters that maximise this function and we assume that all y_i's are independent given the input features X and w.

$$P(y|X,w) = \prod_{i=1}^{n} P(y_i|x_i,w)$$

we want to find w such that.

$$w = \arg\max_w \prod_{i=1}^{n} P(y_i|x_i,w)$$

take the log...

$$= \arg\max_w \sum_{i=1}^{n} \log P_w(y_i|x_i,w)$$

$$= \arg\max_w -\sum_{i=1}^{n} \log\left(1+e^{-y_i w^T x}\right)$$

$$= \arg\min_w \sum_{i=1}^{n} \log\left(1+e^{-y_i w^T x}\right)$$

remove negative sign and rephrase to minimisation

To solve this, we can try setting the derivative to 0 and solving for w, but it turns out there is no closed form solution to this. To solve we need to use a hill climbing / gradient descent method.

Maximum A Posteriori

In MAP, we treat w as a random variable and we can specify a prior belief over this. We may use $w \sim \mathcal{N}(0,\sigma^2 I)$ → Gaussian.

Our goal in map is to find the most ~~take~~ likely model parameters given our data, i.e. parameters that maximise the posterior.

$$P(w|D) = P(w|X,y) \propto \underbrace{P(y|X,w)}_{\text{MLE}} P(w)$$

$$\hat{W}_{MAP} = \underset{w}{\operatorname{argmax}} \log\left(P(y|X,w)P(w)\right)$$

$$= \underset{w}{\operatorname{argmin}} \left(\sum_{i=1}^{n} \log\left(1 + e^{-y_i w^T x_i}\right) + \lambda w^T w \right)$$

where $\lambda = 1/2\sigma^2$.

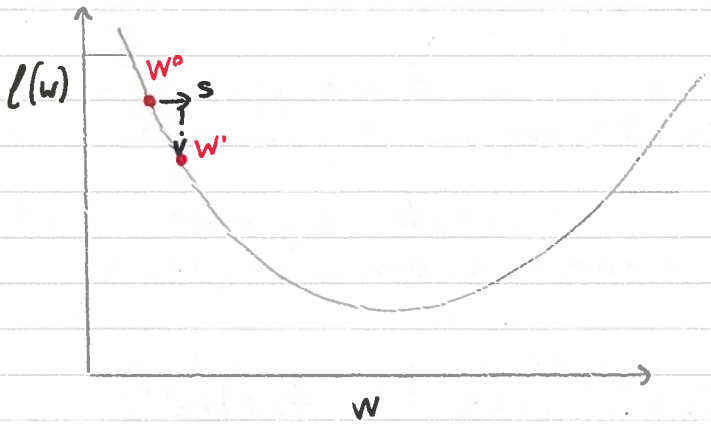
Gradient Descent

$$\text{MLE: } \underset{w}{\operatorname{argmax}} \prod_i P(y_i|x_i) = \underbrace{\underset{w}{\operatorname{argmin}} \sum_{i=1}^{n} \log\left(1+e^{-w^T x y_i}\right)}_{\ell(w) \text{ loss function}}$$

We want to minimise a convex, continuous and differentiable loss function $\ell(w)$

$\Rightarrow \underset{w}{\operatorname{argmin}}\, \ell(w)$

We don't know much about this function other than it looks roughly like this...



Choose initial w value (w_0) and step distance s away.

If $\ell(w_0+s)$ is lower, we are going in the right direction to minimise.

$\ell(w+s)$ can be quite tricky to solve, so we assume it is much more simple than it really is. We use Taylor expansion, provided that the norm $\|s\|_2$ is small (ie. $w+s$ is very close to w) we can approximate $\ell(w+s)$ by its first and second derivatives...

$$\ell(w+s) \approx \ell(w) + \underbrace{g(w)^T s}_{\text{gradient at } w,\ \nabla\ell(w)} + \frac{1}{2} s^T \underbrace{H(w)}_{\nabla^2\ell(w),\ \text{Hessian}} s$$

If only using one derivative, approximates a local straight line
If only using two derivatives, approximates a local parabola.

More derivatives $\rightarrow$ more expensive calculations

In gradient descent we only use the gradient (first derivative) This means we assume that the function ℓ around w is linear and behaves like $\ell(w) + g(w)^T s$. Our goal is to find an s that minimises this function.

$$s = -\alpha g(w)$$

for some small $\alpha > 0$. We can show that $\ell(w+s) < \ell(w)$

$$\ell(w + \underbrace{(-\alpha g(w))}_{s}) \approx \ell(w) - \underbrace{\alpha g(w)^T g(w)}_{s} < \ell(w)$$

$\underbrace{\qquad\qquad}_{>0}$

Setting the learning rate $\alpha > 0$ is not trivial. Only if it is sufficiently small will the gradient descent converge. If it is too large, the algorithm will easily diverge out of control.

A safe (but slow) choice is $\alpha = \frac{t_0}{t}$ where t is the number of updates already taken and t_0 is some constant.

Adagrad

This is a method for choosing α, and it will be different for every feature.

Adagrad keeps a running average of the squared magnitude gradient and sets a small learning rate for features with a large gradient and large learning rate for features with small gradient.

Setting different learning rates for different features is important for features that vary in scale or frequency. For example, word counts.

$s = 0$

FOR

$$g = \frac{\partial L}{\partial w}$$

$$s \leftarrow s + g.^2$$ ← element-wise squaring

$$w \leftarrow w - \frac{\alpha g}{\sqrt{s+\varepsilon}}$$

stops division by 0 if $s=0$

Once the difference between w after updates is less than some δ, stop the algorithm.

Newton's Method - 2nd Order Approximation

Newton's method assumes that the loss function L is twice differentiable, and uses the approximation with the Hessian.

The Hessian matrix contains all second order partial derivatives and is defined as...

$$\begin{bmatrix} \frac{\partial^2 \ell}{\partial w_1^2} & \frac{\partial^2 \ell}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 \ell}{\partial w_1 \partial w_n} \\ \frac{\partial^2 \ell}{\partial w_2 \partial w_1} & \frac{\partial^2 \ell}{\partial w_2^2} & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \ell}{\partial w_n \partial w_1} & \cdots & \cdots & \frac{\partial^2 \ell}{\partial w_n^2} \end{bmatrix} = H(w)$$

and because of the convexity of ℓ, it always has a symmetric square matrix and positive semi-definite.

NOTE A symmetric matrix M is positive semi-definite if it has only none-negative eigenvalues or equivalently, for any vector x we must have $x^T M x \geq 0$.

It follows that the approximation

$$\ell(w+s) \approx \ell(w) + g(w)^T s + \frac{1}{2} s^T H(w) s$$

describes a ~~complex parabola~~ convex parabola, and we can find it's minimum by solving the following optimization problem.

$$\underset{s}{\operatorname{argmin}} \; \ell(w) + g(w)^T s + \frac{1}{2} s^T H(w) s$$

To find the minimum of the objective, we take it's first derivative with respect to s, equate it to 0, and solve for s.

$$g(w) + H(w)s = 0$$

$$\Rightarrow s = -[H(w)]^{-1} g(w)$$

This choice of s converges extremely fast if the approximation is sufficiently accurate and the resulting step is sufficiently small. Otherwise it can diverge. Divergence often happens if the function is flat or almost flat with respect to some dimension. In that case, the second derivatives are close to 0, and their inverses become very large - resulting in gigantic steps. Different from gradient descent, here there is no step size that guarantees that the steps are small and local. As the taylor approximation is only accurate locally, large steps can move the current estimates far from regions where the Taylor approximation is accurate.

Best Practices

1. The matrix $H(w)$ scales $d \times d$ and is expensive to compute. A good approximation is to only take the diagonal components and update with a small step size. This is then essentially a hybrid between gradient descent and Newton's method.

2. To avoid divergence, a good approach is to start with gradient descent and finish with Newtons method.

   Gradient descent will get close quickly, but take many steps to then find the minimum accurately.

   Newtons method works quickly if its starting point is already close to the minimum, but often fails if far away.

LINEAR REGRESSION OLS: Ordinary Least Squares

Data assumption $y_i \in \mathbb{R}$

Model assumption $y_i = w^T x_i + \epsilon_i$ where $\epsilon_i = \mathcal{N}(0, \sigma^2)$ (Gaussian, 0 mean)

y_i is drawn from a continuous distribution

we assume the correspondence between x and y is linear, but there is an additional noise model, ϵ_i. (Gaussian)

$$\Rightarrow y_i | x_i = \mathcal{N}(w^T x_i, \sigma^2)$$

$$P(y_i | x_i, w) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(w^T x_i - y_i)^2}{2\sigma^2}}$$

Estimating w with MLE

$$= \operatorname{argmax}_w \prod_{i=1}^{n} P(y_i | x_i | w)$$

$$= \operatorname{argmax}_w \prod_{i=1}^{n} P(y_i | x_i, w)$$

$$= \operatorname{argmax}_w \sum_{i=1}^{n} \log \left[P(y_i | x_i, w) \right]$$

$$= \operatorname{argmax}_w \sum_{i=1}^{n} \left[\log \left(\frac{1}{\sqrt{2\pi\sigma}} \right) + \log \left(e^{-\frac{(w_i^T x - y_i)^2}{2\sigma^2}} \right) \right]$$

$$= \operatorname{argmax}_w \sum_{i=1}^{n} -\frac{1}{2\sigma^2} (w^T x_i - y_i)^2$$

$$= \operatorname{argmax}_w -\frac{1}{2\sigma^2}\sum_{i=1}^{n}(w^T x_i - y_i)^2$$

$$= \operatorname{argmin}_w \sum_{i=1}^{n}(w^T x_i - y_i)^2$$

$$= \operatorname{argmin}_w \frac{1}{n}\sum_{i=1}^{n}(w^T x_i - y_i)^2$$

we divide by n so the loss is independent of number of data points.

Maximum A Posteriori

Model assumption: $P(w) = \frac{1}{\sqrt{2\pi\tau^2}} e^{\frac{-w^T w}{2\tau^2}}$ Prior

$$w = \operatorname{argmax}_w P(w|D) = \operatorname{argmax}_w \frac{P(D|w)P(w)}{P(D)}$$

$$= \operatorname{argmax}_w P(D|w)P(w)$$

$$= \operatorname{argmax}_w \left[\prod_{i=1}^{n} P(y_i|x_i,w)\right]P(w)$$

$$= \operatorname{argmax}_w \sum_{i=1}^{n} \log\left(P(y_i|x_i,w)\right) + \log\left(P(w)\right)$$

Remember this from MLE

$$= \operatorname{argmax}_w \sum_{i=1}^{n} \frac{1}{2\sigma^2}(w^T x_i - y_i)^2 + \log\left(\frac{1}{\sqrt{2\pi\tau^2}}\right) + \log\left(e^{\frac{-(w^T w)}{2\tau^2}}\right)$$

cancel

$$= \operatorname{argmax}_w \sum_{i=1}^{n} -\frac{1}{2\sigma^2}(w^T x_i - y_i)^2 - \frac{1}{2\tau^2}(w^T w)$$

$$= \cancel{\operatorname{argmax}} \operatorname{argmin}_w \frac{1}{n}\sum_{i=1}^{n}(w^T x_i - y_i)^2 + \lambda\|w\|_2^2$$

where $\lambda = \frac{\sigma^2}{n\tau^2}$

Closed form solution to minimise the loss function during Linear regression.

$$L(w) = \frac{1}{n}\sum_{i=1}^{n}(w^T x_i - y_i)^2$$

$x^T = [x_1, \ldots, x_n]$ and $y^T = [y_1, \ldots, y_n]$

$(Xw - y)^2 = (Xw-y)^T(Xw-y)$ multiply out...

$= w^T X^T X w - 2y^T Xw + y^T y$

now differentiate with respect to w

$$\frac{d}{dw}(w^T X^T X w - 2y^T y w + y^T y) = 2x^T x w - 2x^T y$$

set equal to zero and solve for w

$$2x^T x w - 2x^T y = 0 \Rightarrow 2x^T x w = 2x^T y$$

$$x^T x w = x^T y \Rightarrow \boxed{w = (x^T x)^{-1} x^T y}$$

This is the equation to find the minimum of the parabolic loss function. However, despite having the closed form solution, we don't always do this.

This is because the matrix $(x^T x)$ is gigantic and would take a very long time to compute the inverse.

Memory required to computer the inverse is the dimension squared.

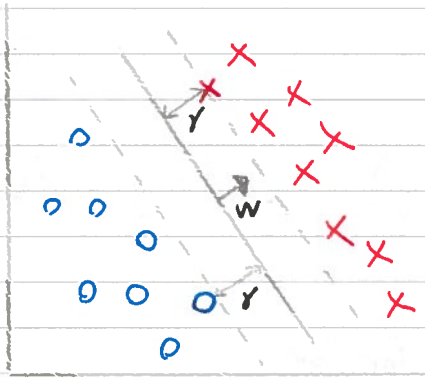
Complexity increases with dimension cubed

SUPPORT VECTOR MACHINES

Previously, classification by separation of a hyperplane was performed by the perceptron. If a hyperplane which separates the data exists, the perceptron will find it.

However, there could be infinitely many hyperplanes which seperate the data. If the perceptron finds one, it's done its job. But which one should it really be?

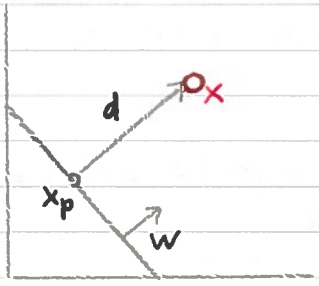
The support vector machine (SVM) is viewed as an extension of the perceptron, which will find the hyperplane with the maximum margin separating hyperplane.



In SVM, the margin γ is the distance from the hyperplane to the closest points in either class. (which touch the parallel dotted line)

Margin $H_{w,b} = \{x : w^T x + b = 0\}$

What is the distance from the hyperplane to a new point x?



Consider some point x, Let d be the vector from H to x of minimum length.

Let x_p be the projection of x onto H

$X_p = x - d$

d is parallel to w, so $d = \alpha w$ for some $\alpha \in \mathbb{R}$

also $x_p \in \mathcal{H}$ which implies $w^T x_p + b = 0$

therefore $w^T x^p + b = w^T(x - d) + b = w^T(x - \alpha w) + b = 0$

which implies $\alpha = \dfrac{w^T x + b}{w^T w}$

Now we want to find the length of d...

$$d = \alpha w = \frac{w^T x + b}{w^T w} w$$

$$\text{norm}(d) = \|d\|_2 = \sqrt{d^T d} = \sqrt{\alpha^2 w^T w} = \alpha \sqrt{w^T w}$$

$$= \frac{w^T x + b}{w^T w} \sqrt{w^T w} = \frac{w^T x + b}{\sqrt{w^T w}} = \boxed{\frac{w^T x + b}{\|w\|_2}}$$

What is the smallest distance from the hyperplane to a datapoint (γ)

$$\gamma(w, b) = \min_{x \in D} \frac{w^T x + b}{\|w\|_2}$$

compute the distance between the hyperplane and all datapoints, then return the smallest one.

Note that the margin and the hyperplane are scale invariant.

$$\gamma(\beta w, \beta b) = \gamma(w, b) \quad \forall \beta \neq 0$$

Note that if the hyperplane is such that γ is maximised, it must lie right in the middle of the two classes. In other words, γ must be the distance to the closest point within both classes.

Max Margin Classifyar

We can formulate our search for the maximum margin separating hyperplane as a constrained optimisation problem.

The objective is to maximise the margin under the constraints that all data points must lie on the correct side of the hyperplane.

$$\max_{w,b} \gamma(w,b) \quad \forall_i \; y_i(w^T x_i + b) \geq 0$$

maximise margin — separating hyperplane

If we plug-in the definition for γ we obtain

$$\max_{w,b} \frac{1}{\|w\|_2} \left(\min_{x_i \in D} |w^T x_i + b| \right) \quad \forall_i \; y_i(w^T x_i + b) \geq 0$$

Because the hyperplane is scale-invariant we can fix the scale of w and b any way we want. We choose it such that.

$$\min_{x \in D} |w^T x + b| = 1$$

We can add this rescaling as an equality constraint

$$\max_{w,b} \frac{1}{\|w\|_2} \cdot 1 = \min_{w,b} \|w\|_2 = \min_{w,b} w^T w$$

* Although this approach will not give the correct value of the margin, it will give us the correct hyperplane which is all we need right now.

$$\min_{w,b} w^T w \quad \forall_i \; y_i(w^T x_i + b) \geq 0$$

$$\min_{x \in D} |w^T x + b| = 1$$

same

we can combine these two constraints such that...:

$$\min_{w,b} w^T w \quad \forall_i \; y_i(w^T x_i + b) \geq 1$$

This new formulation is a quadratic optimization problem. The objective is quadratic and the constraints are linear.

It has a unique solution whenever a hyperplane exists.

SVM with Soft Constraints

What happens if there is no solution to the SVM and the data is not linearly separable? The returned solution will be "infeasible".

This often happens if the data is low dimensional. We can try to retrieve an answer by allowing the restraints to be slightly violated so only a few points are misclassified.

To do this we add "slack" variables.

$$\min_{w,b} w^T w + C\sum_{i=1}^{n} \xi_i \quad \forall_i \; y_i(w^T x_i + b) \geq 1 - \xi_i$$

$$\forall_i \; \xi_i \geq 0$$

The slack variable ξ_i allows the input x_i to be closer to the hyperplane (or even be on the wrong side), but there is a penalty in the objective function for such slack. If C is very large, the SVM becomes very strict and tries to get all points on the right side of the hyperplane. If C is very small, the SVM becomes very loose and may sacrify points to obtain a simpler (i.e lower $||w||_2$) solution.

Let us consider the value of ξ_i for the case of $C \neq 0$. Because the objective will always try to minimise ξ_i as much as possible, the equation must hold as an equality

$$\xi_i = \begin{cases} 1 - y_i(w^T x_i + b) & \text{if } y_i(w^T x_i + b) < 1 \\ 0 & \text{if } y_i(w^T x_i + b) \geq 1 \end{cases}$$

$$\Rightarrow \xi_i = \max\left(1 - y_i(w^T x_i + b),\ 0\right)$$

If we plug this closed form into the objective for our SVM optimisation problem, we obtain this unconstrained version

$$\min_{w,b} \underbrace{w^T w}_{\text{regularizer}} + C \sum_{i=1}^{n} \underbrace{\max\left(1 - y_i(w^T x_i + b),\ 0\right)}_{\text{hinge loss}}$$

Small C → large margin, poor classification

Large C → small margin, good classification

Empirical Risk Minimisation

The unconstrained SVM formular is

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

regularizer — Hinge-loss

The hinge-loss is the SVM's error function which penalises misclassifications, whereas the regularizer penalises complex solutions. This is an example of empirical risk minimisation. It helps answer the question "do I want good classification or do I want a simple solution?"

Here is the general form of an empirical risk minimisation problem

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} \underbrace{L(h_w(x_i), y_i)}_{\text{loss}} + \underbrace{\lambda r(w)}_{\text{regulariser}}$$

Hinge Loss

$$\max\left(1 - h_w(x_i) y_i, 0\right)^p$$

- standard if $p=1$. If $p=2$, the solution is differentiable in all cases.
- Used in SVM

Log Loss

$$\log\left(1 + e^{-h_w(x_i)y_i}\right)$$

- Used for Logistic Regression
- Outputs well calibrated probabilities

Exponential Loss

$$e^{-h_w(x_i)y_i}$$

- Used for AdaBoost
- Very aggresive. The loss of a misprediction increases exponentially with the value of $-h_w(x_i)y_i$

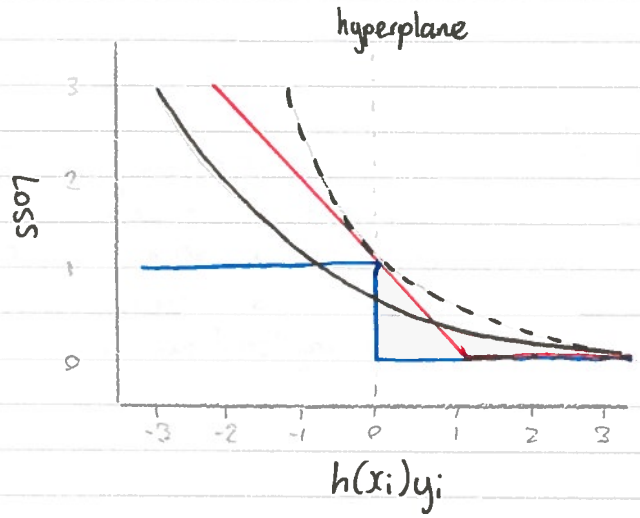
Zero-One Loss

$$\delta\left(\text{sign}\left(h_w(x_i)\right) \neq y_i\right)$$

- Actual classification loss
- Non continuous and therefore impractical to optimise.

What do these loss functions look like?

- Hinge-loss ——
- Log-loss ——
- Exponential loss - - -
- Zero-one loss ——



Which functions are strict upper bounds of the zero-one loss? This can also be phrased as "if the loss is normalised by number of points, which functions can tell us about the number of correctly classified points?".

Hinge-loss, exponential loss and zero-one loss are all strict upper-bounds of the zero-one loss. Eg.

If $\frac{1(h_w(x_i)y_i)}{N} = 0.2, \Rightarrow$ no more than 20% can be misclassified.

This is not true for the log-loss.

What can we say about the hinge-loss and the log-loss as $h_w(x_i)y_i \to -\infty$?

They both become parallel lines and are very well behaved when faced with outliers.

Regression Loss Functions

Regression $y_i \in \mathbb{R}$

Squared-loss

$(h(x_i)-y_i)^2$

- most popular regression loss function
- estimates mean label
- differentiable everywhere
- sensitive to outliers
- "ordinary least squares"

Absolute Loss

$|h(x_i)-y_i|$

- estimates median label
- less sensitive to noise
- not differentiable at 0.

Huber Loss

$$\begin{cases} \frac{1}{2}(h(x_i)-y_i)^2 & \text{if } |h(x_i)-y_i| < \delta \\ \delta(|h(x_i)-y_i| - \frac{\delta}{2}) & \text{otherwise} \end{cases}$$

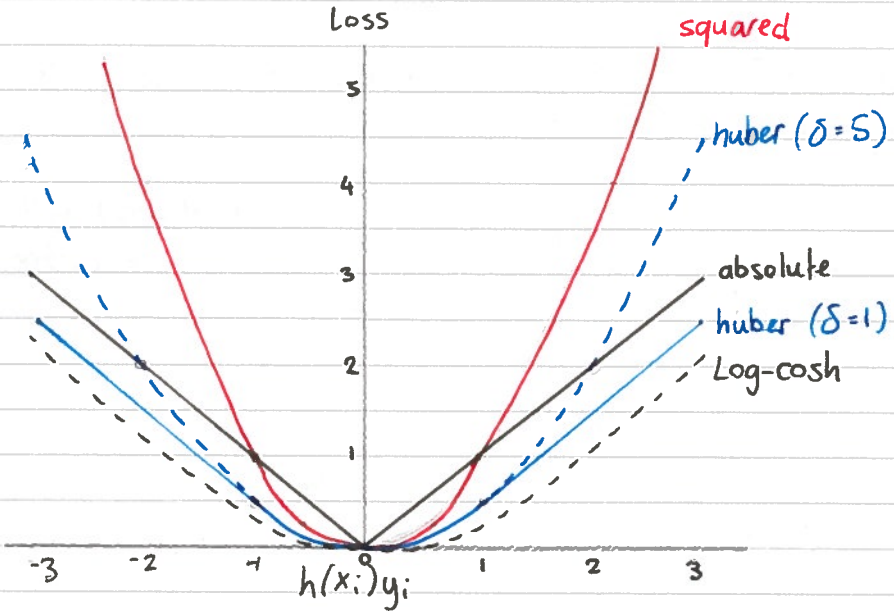
- "smooth absolute loss"
- once-differentiable
- squared-loss when loss is small
- absolute loss when loss is large
- usually the prefered choice.

Log-Cosh Loss

$$\log(\cosh(h(x_i) - y_i))$$

where $\cosh(x) = \frac{e^x + e^{-x}}{2}$

- similar to Huber loss
- differentiable everywhere (twice)



Regularisation

Regularisers are terms which favor simple solutions.

$$\min_{w,b} \sum_{i=1}^{n} l(h_w(x_i) y_i) + \lambda r(w)$$

$$\Leftrightarrow \quad \min_{w,b} \sum_{i=1}^{n} l(h_w(x_i) y_i) \quad \forall_i \; r(w) \leq B$$

For each $\lambda \geq 0$, there exists $B \geq 0$ such that two formulations are equivalent and vice-versa.

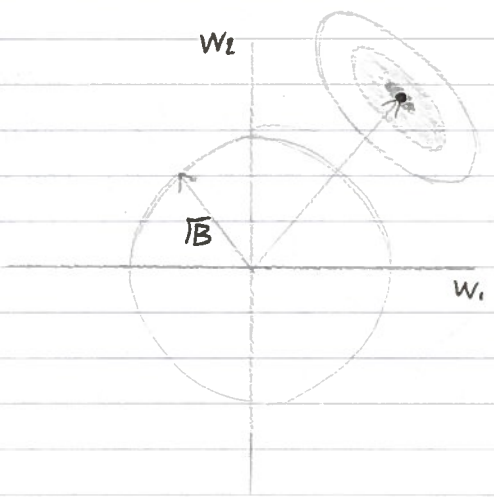
This way we can simply view the regulariser as a constraint.

$$r(w) = w^T w \quad \Rightarrow \quad w^T w \leq B$$

If w has two dimensions, we can graph its components.

$w_1^2 + w_2^2 \leq B$ is equivalent to $w^T w \leq B$

and $w_1^2 + w_2^2 \leq B$ is the equation for a circle with a radius of $\sqrt{B}$.

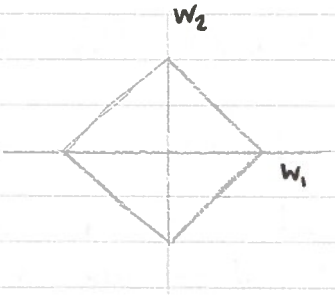


$f(w_1, w_2)$ with a minimum at the centre.

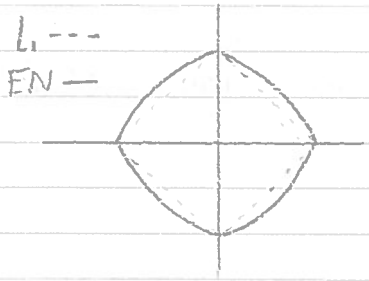
If we try to minimise some function of w ($f(w)$), with constraint of $w^T w \leq B$, the solution must lie inside the circle. If the circle has a radius of a value greater than the distance to the minimum, we will find the true minimum of $f(w)$. If the radius is smaller, the solution will lie on some point on the edge of the circle as close to the minimum as possible.

By restricting the minimisation to within the circle (or hypersphere for higher dimensions), you avoid overfitting with complex solutions.

The example of the hypersphere is obtained when the regulariser is $w^T w$ ($\|w\|_2^2$), and is known as the l_2 regulariser. This is strictly convex and differentiable, but would rely on all features to some degree.



The l_1 regulariser instead uses $\|w\|_1$. The shape here is more representative of a diamond. This can be advantageous as minimal solutions outside the area can force the weights of some features to be 0, thus eliminating their effect. This helps understand the model a lot better. Not strictly convex.



The elastic-net regulariser uses mostly the l_1 component with a small amount of l_2 $r(w) = \lambda\|w\|_1 + \mu\|w\|_2$. This is strictly convex and has a unique solution (unlike l_1) but has most properties of l_1. However, this is non-differentiable.

BIAS-VARIANCE TRADEOFF

As usual, we are given a dataset of n points

$$D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

from identically and independently from $P(X,Y) = P(Y|X)P(X)$

We assume this is a regression such that $y \in \mathbb{R}$

It may be the case that for any given input x, there may not be a unique Y. If the vector x describes and the label y is the price, you could imagine two houses with identical features selling for different prices. So for any given feature vector x, there is a distribution over possible labels.

We can therefore define the expected label as (given $x \in \mathbb{R}^d$)

$$\bar{y}(x) = E_{y|x}(Y) = \int_y y P_r(y|x) \, dy$$

Expected y given x

Integrate over all possible y's and weight them by the probability that we see this y given x.

This is basically saying given an infinite number of identical feature vectors and their labels, what label would we expect to most likely on another identical feature set x.

Next, we will call some machine learning algorithm A, and apply it to a dataset D, in order to learn some hypothesis h_D

$$h_D = A(D)$$

For a given h_D learned of dataset D using algorithm A, we can compute the generalisation error. Here we use the squared loss function. The expected test error is given as

Expected test error given h_D

$$E_{(x,y)\sim P}\underbrace{\left[(h_D(x)-y)^2\right]}_{\text{squared loss}} = \iint_{x\,y} (h_D(x)-y)^2 \Pr(x,y)\,dy\,dx$$

* We can use other loss functions. We use the squared loss because its very common, and a reasonable choice for regression. It is also an easy function to work with in later maths.

The previous statement is true given training set D. However, D itself is drawn from P^n, and is therefore a random variable. As h_D is a function of D, h_D is also a random variable. We can therefore compute the expected value of h_D.

Expected classifier (given A)

$$\bar{h} = E_{D\sim P^n}[h_D] = \underbrace{\int_D h_D \Pr(D)\,dD}$$

Integrate out all possible training datasets, and weight the classifier by the probability of extracting dataset D.

$\bar{h}$ is a weighted average over all functions. Weak law of large numbers.

We can also use the fact that h_D is a random variable to compute the expected test error given A

$$E_{\substack{(x,y)\sim P \\ D\sim P^n}}\left[(h_D(x)-y)^2\right] = \iiint_{D\,x\,y} (h_D(x)-y)^2 P(x,y)P(D)\,\partial x\,\partial y\,\partial D$$

This will evaluate the quality of the machine learning algorithm A with respect to a data distribution $P(X,Y)$. We will show that this expression decomposes into three meaningful terms.

Decomposition

Using the expected test error given A, we perform a trick whereby we add and subtract $\bar{h}(x)$ inside the square error

$$E_{\substack{(x,y)\sim P \\ D\sim P^n}}\left[(h_D(x)-y)^2\right] = E_{x,y,D}\left[\left[\underbrace{(h_D(x)-\bar{h}(x))}_{a}+\underbrace{(\bar{h}(x)-y)}_{b}\right]^2\right]$$

sums to zero so equivalent.

Now we expand the bracket ... $\quad a^2+b^2+2ab=(a+b)^2$

$$= E_{x,D}\left[\underbrace{(h_D(x)-\bar{h}(x))^2}_{a^2}\right] + 2E_{x,y,D}\left[\underbrace{(h_D(x)-\bar{h}(x))(\bar{h}(x)-y)}_{2ab}\right] + E_{x,y}\left[\underbrace{(\bar{h}(x)-y)^2}_{b^2}\right]$$

The term $2E_{x,y,D}\left[(h_D(x)-\bar{h}(x))(\bar{h}(x)-y)\right]$ is equal to zero

$$E_{x,y,D}[(h_D(x)-\bar{h}(x))(\bar{h}(x)-y)]$$

$$= E_{x,y}[E_D[h_D(x)-\bar{h}(x)](\bar{h}(x)-y)]$$

split $E_{x,y,D}$ to $E_{x,y}[E_D[\cdots]]$

$$= E_{x,y}[(\underbrace{E_D[h_D(x)]}_{=\bar{h}(x)} - \bar{h}(x))(\bar{h}(x)-y)]$$

remove $\bar{h}(x)$ from E_D as it is not dependent on D.

$$\underbrace{(\bar{h}(x)-\bar{h}(x)}_{=0}(\bar{h}(x)-y)] = E_{x,y}[0] = 0$$

Therefore, our expected test error given A becomes...

$$E_{x,y,D}[(h_D(x)-y)^2] = \underbrace{E_{x,D}[(h_D(x)-\bar{h}(x))^2]}_{\text{Variance}} + E_{x,y}[(\bar{h}(x)-y)^2]$$

The variance is one of our three terms to extract. The remaining two come from the term

$$E_{x,y}[(\bar{h}(x)-y)^2]$$

Similarly to earlier, we add and subtract the term $\bar{y}(x)$ (the expected label given $x \in \mathbb{R}^D$)

$$= E_{x,y}[(\underbrace{(\bar{h}(x)-\bar{y}(x))}_{a} + \underbrace{(\bar{y}(x)-y)}_{b})^2]$$

$= (a+b)^2$

$= a^2 + b^2 + 2ab$

$$= \underbrace{E_{x,y}[(\bar{h}(x)-\bar{y}(x))^2]}_{\text{Bias}^2} + \underbrace{E_{x,y}[(\bar{y}(x)-y)^2]}_{\text{Noise}} + 2E_{x,y}[(\bar{h}(x)-\bar{y}(x))(\bar{y}(x)-y)]$$

These are our remaining two terms.

The term $2E_{x,y}\left[(h(x)-\bar{y}(x))(\bar{y}(x)-y)\right]$ is equal to zero

$$E_{x,y}\left[(h(x)-\bar{y}(x))(\bar{y}(x)-y)\right]$$

$$= E_x\left[E_{y|x}\left[\bar{y}(x)-y\right](h(x)-\bar{y}(x))\right]$$

$$= E_x\left[(\bar{y}(x)-\underbrace{E_{y|x}[y]}_{=\bar{y}(x)})(h(x)-\bar{y}(x))\right]$$

$$= E_x\left[(\bar{y}(x)-\bar{y}(x))(h(x)-\bar{y}(x))\right] = E_x[0] = 0$$

Therefore, the test error given A can be written as

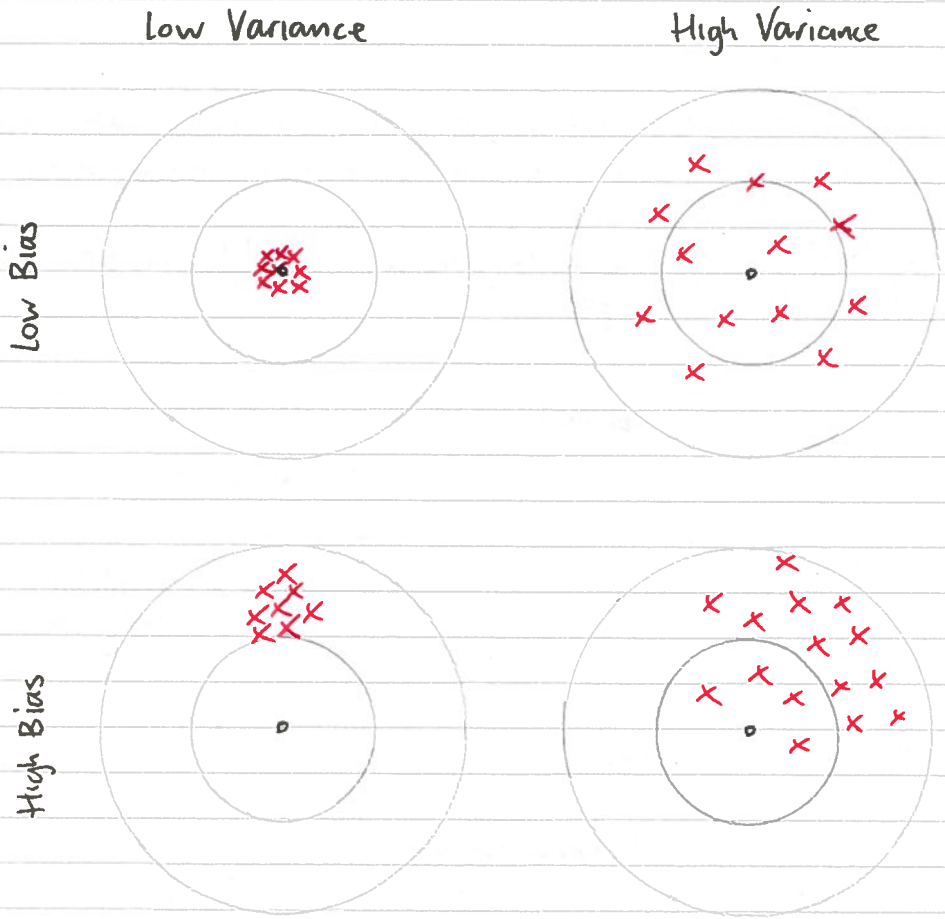
$$\underbrace{E_{x,y,D}\left[(h_D(x)-y)^2\right]}_{\text{Expected Test Error}} = \underbrace{E_{x,D}\left[(h_D(x)-h(x))^2\right]}_{\text{Variance}} + \underbrace{E_{x,y}\left[(\bar{y}(x)-y)^2\right]}_{\text{Noise}} + \underbrace{E_x\left[(h(x)-\bar{y}(x))^2\right]}_{\text{Bias}^2}$$

What does each term mean?

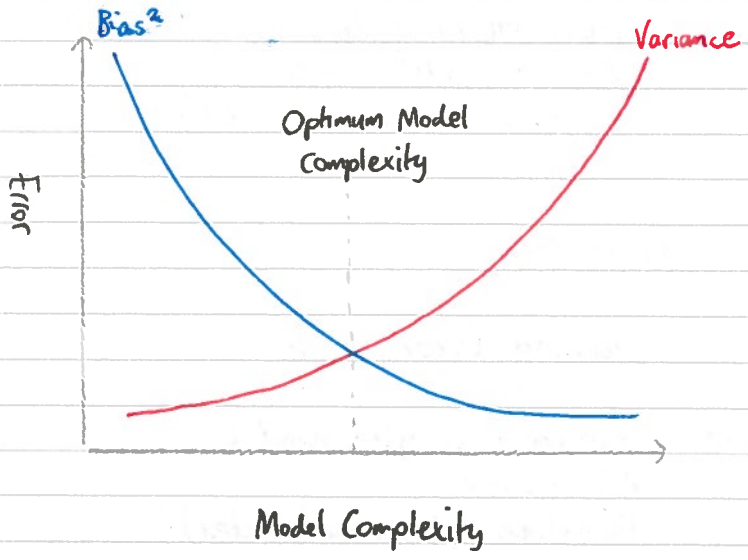
VARIANCE: Captures how much the classifier changes if you train on a different dataset. How "overspecialised" is your classifier to a particular training set (overfitting). If we have the best possible model for our training data, how far off are we from the average classifier.

BIAS: What is the inherent error you obtain from your classifier even with infinite training data? This is due to your classifier being biased to a particular kind of solution. Bias inherent to your model.

NOISE: How big is data intrinsic noise? The error measures ambiguity due to your data distribution and feature representation - You can never beat this.



- Variance is the spread of classifiers
- Bias is the systematic error of classifiers



Detecting High Variance and High Bias

If a classifier is underperforming (eg. test or training ~~data~~ error is too high), there are several ways to improve performance.

To find of which of the many techniques is right, we first need to determine the source of the problem.

Regime 1 : HIGH VARIANCE

Syptoms : Training Error $\ll$ Test Error
Training Error $< \varepsilon$
Test Error $> \varepsilon$

Treatment: More training data
Reduce model complexity
Bagging (discussed later)

Regime 2: HIGH BIAS

Syptoms: Training Error $> \varepsilon$

Treatment: Use more complex models
Add features
Boosting (discussed later)

Optimally Tuning a Classifier

The Empirical Risk Minimisation equation

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} l_s\left(h_w(x_i), y_i\right) + \lambda r(w)$$

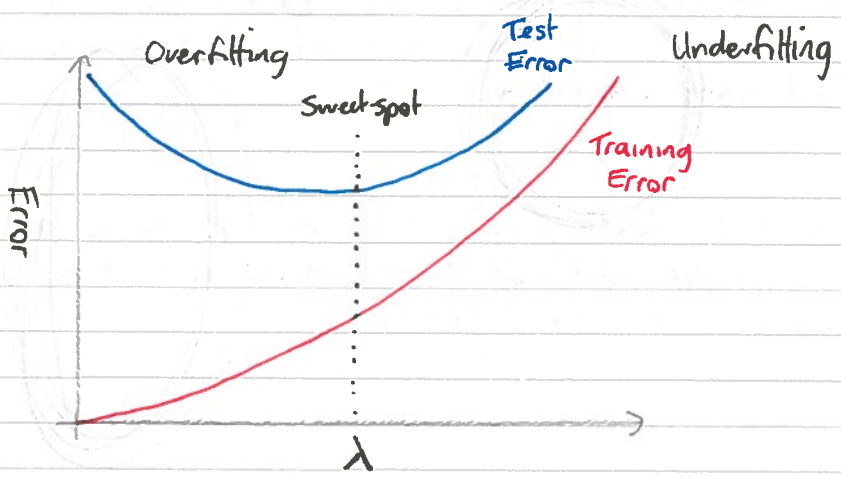
Loss — Regulariser

How do we set λ? As this regulates the bias/variance.

There are two equally problematic cases which can arise when learning a classifier on a dataset: UNDERFITTING AND OVERFITTING. Each relates to the degree to which the data in the training set is extrapolated to apply to unknown data.

UNDER-FITTING: The classifier learned on the training set is not expressive enough to even account for the data provided. In this case, both training and test errors are high as the classifier does not account for the relevant information present in the dataset.

OVER-FITTING: The classifier learned on the dataset is too specific, and cannot be used to infer anything about unseen data. Although training errors decrease over time, test error will only increase as the classifier begins to make decisions based on patterns which only exist in the training dataset and not the overall distribution.



With a small λ, all the emphasis is on classifying correctly which leads to complex models, high training classification rates and low train error. However, this model is highly specific to the training data and therefore gives poor test classification rates and high test error.

With a large λ, all the emphasis is on a simple model, which gives large test and training errors, as the model is to simple for both datasets

The sweet-spot is found using numerous methods.

Identifying the Sweet-Spot

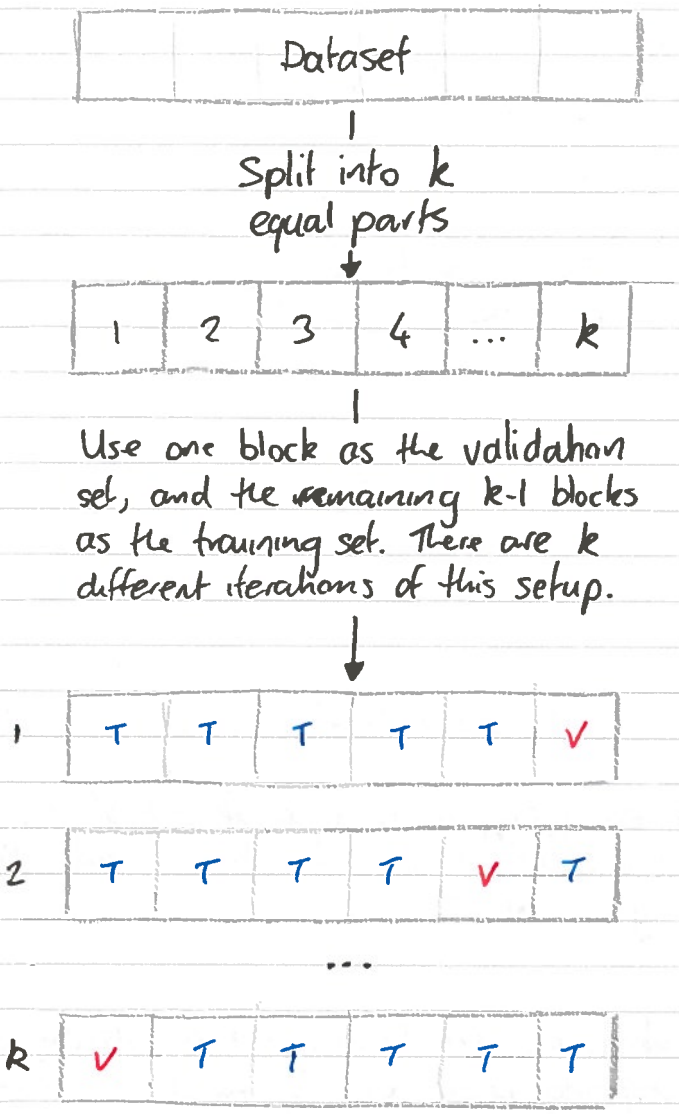
One approach is called k-fold cross validation. First, we need to understand two simple approaches.

GRID SEARCH: Split data into a training and test dataset. Test all lamda values on training and test data and pick the best λ

TELESCOPIC: Split data into train/test datasets. Look at λ values with large gaps between. Pick the best λ then carryout an isolated search around it. Pick the best λ.

The problem with this set of approaches is that trying out many different values of λ will cause inadvertant overfitting to the test dataset. Eg. The identified best value of λ may be specific to this test/train dataset. This is where we use k-fold cross validation.

We start by dividing our data into k seperate partitions. We then train our model on $k-1$ of these apartitions and leave the final one as the test dataset.



Then we try all values of λ for each of the k folds. This is good because we get a much more representative value of λ and we also get a standard deviation.

A higher value of k gives better estimates, but this is a slow process. If $k = n$ (number of samples in the dataset), this becomes "leave one out validation".

Early Stopping

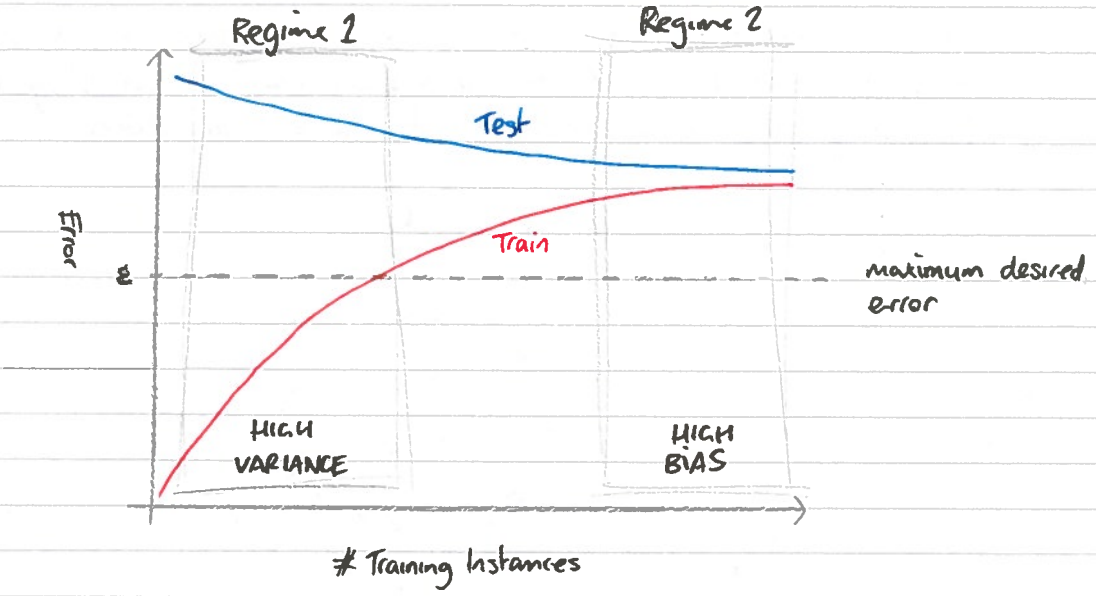
We can use gradient decent to find to optimum value of λ.

The concept of early stopping is to break out of the gradient descent algorithm early and return so sub-optimal result. Why is this connected to regularisation?

With regularisation, we are limiting the complexity of our model which means stopping before our loss function is minimised. Similarly with early stopping, we are only allowing the gradient descent algorith a limited number of steps, which means it will also not return the minimum value, and therefore limit the complexity of the model.

In practise, early stopping is used rather than trying every single λ value. When trying every single λ value, the test-train error needs to be optimised each time. In early stopping, the optimisation is only performed once, but λ values are saved throughout the algorithm.

Debugging



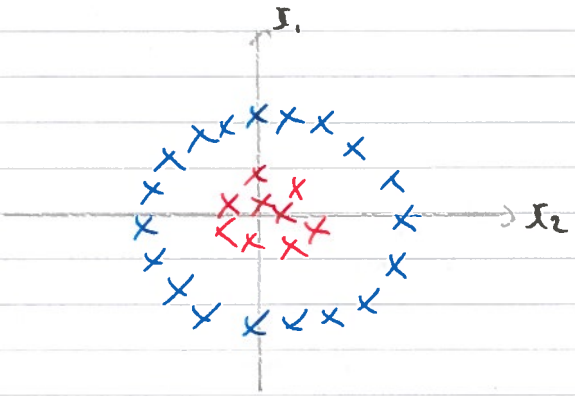
When debugging machine learning models, we are looking for the source of the error. From the bias-variance decomposition lecture, we saw that the error value breaks down into VARIANCE, BIAS and NOISE.

In regime 1, there are few training instances. When this is the case, the model is sucessful at classifying the training data because the problem is simple. With addition of more instances, the error rises because the problem is only getting harder. Training error always increases with more data. With few training samples, the test error is going to be large because the test data will not resemble the few points in the training set. However, adding more training instance will help to reduce the test error. Regime 1 is therefore an issue with HIGH VARIANCE. Improve with bagging, more data, increase regularization.

In regime 2, we have a lot of instances, but the training error is greater than ε. The test error will converge to the training error, but it will never be smaller. The training error is the lower bound of the test error. If the training error is too high, we have a bias problem. More training data won't help because training error cannot decrease with more data. We need more features, a more complex model or kernels or boosting.

KERNELS

Linear classifiers are great, but what if there exists no linear decision boundary? As it turns out, there is an elegant way to incorporate non-linearities into linear classifiers



Hand crafted feature expansion

We can make linear ~~features~~ classifiers non-linear by applying a basis function (feature transformation) on input feature vectors.

Formally, for data vector $x \in \mathbb{R}^d$, we apply the transformation $x \to \phi(x)$, where $\phi(x) \in \mathbb{R}^D$. Usually $D \gg d$ because we add dimensions that capture non-linear interactions among the original features.

Advantage: It is simple and the problem stays convex and well behaved.

Disadvantage: $\phi(x)$ might be very high dimensional

A possible example of function $\phi(x)$ is

$$\text{if } x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_d \end{bmatrix}, \quad \text{then} \quad \phi(x) = \begin{bmatrix} 1 \\ x_1 \\ \dots \\ x_d \\ x_1 x_2 \\ \dots \\ x_{d-1} x_d \\ \dots \\ x_1 x_2 \cdots x_d \end{bmatrix}$$

The dimensionality of x is d, therefore the dimensionality of $\phi(x)$ is 2^d. This is very expensive but allows for complicated non-linear decision boudaries, but dimensionality is extremely high. This makes our algorithm unbearable and prohibitively slow.

However, there is a trick to avoid this problem. It turns out we can express a linear classifier interms of inner-products.

This trick will allow is to use a linear classifier on extremely high dimensional data without ever calculating a single value of our $\phi(x)$ function, or ever computing the full vector w.

It is based on the observation that if we use gradient descent, with anyone of our standard loss functions, the gradient is a linear combination of our input samples. Use square loss for this example.

$$\ell(w) = \sum_{i=1}^{n} (w^T x - y_i)^2$$

From the previous lecture on gradient descent, we know that following an update, the estimated w becomes

$$w_{t+1} = w_t - s\left(\frac{\partial \ell}{dw}\right)$$

where $\frac{\partial L}{\partial w} = \sum_{i=1}^{n} \underbrace{2(w^T x_i - y_i)x_i}_{\gamma_i = \text{function of } x_i \text{ and } y_i} = \sum_{i=1}^{n} \gamma_i x_i = g$

We will now show that we can express w as a linear combination of all input vectors, such that...

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

Since the loss function is convex, the final solution is independent of the initialisation, and we can set w_0 to anything we want.

For convenience, we pick $w_0 = \begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \end{bmatrix}$

For this initialisation the linear combination in $w = \sum_{i=1}^{n} \alpha_i x_i$ is trivially $\alpha_1 = \dots \alpha_n = 0$.

We now show that throughout the gradient descent optimisation, coefficients of α must always exist, as we can rewrite the gradient descent updates entirely in terms of updating the α_i coefficients.

$$w_1 = w_0 - s = \sum_{i=1}^{n} \alpha_i^0 x_i - \sum_{i=1}^{n} \gamma_i^0 x_i = \sum_{i=1}^{n} \alpha_i' x_i$$

where $\alpha_i' = (\alpha_i^0 - \gamma_i^0)$

This means we can perform the entire gradient descent update rule with having to ever explicitly express w. We just need to keep track of n coefficients (1 for each data element).

Now that w can be written as a linear combination of the training set, we can also express the inner product of w with any input x_i purely in terms of inner-products between training inputs.

$$w^T x_j = \sum_{i=1}^{n} \alpha_i x_i^T x_j$$

Therefore, we can write the squared loss function in terms of inner-products of input functions

$$\ell(\alpha) = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \alpha_j x_j^T x_i - y_i \right)^2$$

The only information we ever need in order to learn a hyper-plane classifier with the squared loss function is the inner-product between all pairs of data vectors.

The final trick is a method to calculate the inner-products of the very high dimensional input vectors $\phi(x)$. The inner product of $\phi(x)^T \phi(z)$ is

$$\phi(x)^T \phi(z) = 1 \cdot 1 + x_1 z_1 + x_2 z_2 + \ldots + x_1 x_2 z_1 z_2 + \ldots + x_1 \cdots x_d z_1 \cdots z_d$$

It turns out that the equation above is equivalent to

$$= \prod_{k=1}^{d} (1 + x_k z_k)$$

which takes milliseconds compared the regular inner product equation. Therefore the computation is of the order d rather than 2^d. We call this function the kernel function. $k(x_i, x_j)$

With a finite training set of n samples, inner-products are often precomputed and stored in a kernel matrix

$$K_{ij} = \phi(x_i)^T \phi(x_j)$$

Not only have we avoided calculating $\phi(x)$ and w, but it only takes of the order n instead of 2^n.

General Kernels

LINEAR: $K(x,z) = x^T z$

POLYNOMIAL: $K(x,z) = (1 + x^T z)^d$

if $p=1$, just linear
$p>2$, polynomial

RADIAL BASIS FUNCTION (RBF): $K(x,z) = e^{\frac{-\|x-z\|^2}{\sigma^2}}$

This is the most popular kernel. It is the universally appropriate kernel. Its corresponding feature vector is infinite but it cannot be computed. It makes everything linearly separable.

Can any function $K(\cdot,\cdot) \to R$ be used as a kernel?

No. The matrix $K(x_i, x_j)$ has to correspond to real inner products after some transformation $x \to \phi(x)$. This is the case if and only if K is positive semi-definite.

Constructing Kernels

There are 8 rules to follow when constructing kernels:

1. $k(x,z) = x^T z$

   this is just the regular inner-product

2. $k(x,z) = c\,k_1(x,z)$

   if it is multiplied by any scalar, it is still a well defined kernel. $c \geq 0$

3. $k(x,z) = k_1(x,z) + k_2(x,z)$

the addtion of any two kernels is still a well defined kernel.

4. $k(x,z) = g(k_1(x,z))$

where g is a polynomial with positive coefficient.

5. $k(x,z) = k_1(x,z)\,k_2(x,z)$

the multiplication of any two kernels is still a well defined kernel.

6. $k(x,z) = f(x)\,k_1(x,z)\,f(z)$

if $f(x)$ is to the left of a kernel and the same function $f(g)$ is to the right of a kernel, the result is still a well-defined kernel.

7. $k(x,z) = e^{k_1(x,z)}$

The exponential of a kernel is still a well-defined kernel.

8. $k(x,z) = x^T A z$

where A is a positive semi-definite matrix.
$A \succeq 0$

Using these rules, we can prove that the RBF is a well defined kernel.

$$k_1(x,z) = x^T z \qquad \text{rule 1}$$

$$k_2(x,z) = \frac{2}{\sigma^2} x^T z \qquad \text{rule 2}$$

$$k_3(x,z) = e^{\frac{2x^Tz}{\sigma^2}} \qquad \text{rule 7}$$

$$k_4(x,z) = e^{\frac{-x^Tx}{\sigma^2}} e^{\frac{2x^Tz}{\sigma^2}} e^{\frac{-z^Tz}{\sigma^2}} \qquad \text{rule 6}$$

$$= e^{\frac{-x^Tx + 2x^Tz - z^Tz}{\sigma^2}}$$

$$= e^{\frac{-(x-z)^2}{\sigma^2}} = RBF$$

The following kernel is defined on any 2 sets $S_1, S_2 \subseteq \Omega$

$$k(S_1, S_2) = e^{|S_1 \cap S_2|}$$

List out all possible samples Ω and arrange them into a sorted list. We define a vector $x_S \in \{0,1\}^{|\Omega|}$, where each of it's elements indicates whether a corresponding sample is included in the set S.

$$k(S_1, S_2) = e^{|S_1 \cap S_2|} = e^{x_{S_1}^T x_{S_2}}$$

Kernel Machines

In practise, an algorithm can be kernelised in ~~four~~ three steps.

1. Prove that the solution lies in the span of the training points, i.e $w = \sum_{i=1}^{n} \alpha_i x_i$
2. Rewrite the algorithm and the classifier so that all training or test inputs x_i are only accessed in the inner-products with other inputs $x_i^T x_j$
3. Substitute in a kernel function for $x_i^T x_j$, $k(x_i, x_j)$

Kernelised Linear Regression

The vanilla least squares regression (linear regression) minimises the square loss

$$\min_w \sum_{i=1}^{n} (w^T x_i - y_i)^2$$

to find the hyper-plane w. The prediction at a test point is then $h(x) = w^T x$

If we let $X = [x_1, x_2, \ldots, x_n]$ and $y = [y_1, y_2, \ldots, y_n]^T$, the solution of of the OLS can be written in closed form as:

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

We can substitute w for $w = \sum_{i=1}^{n} \alpha_i x_i = X\alpha$

We know that vector α must always exist by observing gradient descent updates with w_0 set to a zero vector.

Similarly, during testing, a test point is only accessed through inner-products with training inputs.

$$h(z) = w^T z = \sum_{i=1}^{n} \alpha_i x_i^T z$$

We can now immediately kernelise the algorithm by substituting $k(x,z)$ for $x^T z$. It remains to show that we can also solve the values for α in closed form.

$$X\alpha = w = (XX^T)^{-1} Xy$$

multiply from the left by $(X^T X X^T)$

$$(X^T X)(X^T X)\alpha = X^T \underbrace{(XX^T (XX^T)^{-1})}_{\text{Identity}} Xy$$

substitute $K = X^T X$

$$K^2 \alpha = Ky \Rightarrow K\alpha = y \Rightarrow \alpha = K^{-1} y$$

therfore $\alpha = \underbrace{(X^T X)^{-1} y}_{\text{inner-product = KERNEL!}}$

Kernel regression can be extended to the kernelised version of ridge regression. The solution then becomes

$$\alpha = (K + \tau^2 I)^{-1} y$$

In practise, a small value of τ^2 increases stability, especially if K is not invertible. If $\tau = 0$, kernel ridge regression becomes kernelised linear regression.

Kernel SVM

An SVM with soft constraints can be written as

$$\min_{w,b} \; w^T w + C\sum_{i=1}^{n} \xi_i \qquad \forall_i \; y_i(w^T x_i + b) \geq 1 - \xi_i$$
$$\forall_i \; \xi_i \geq 0 \qquad \text{PRIMAL}$$

and has the dual form

$$\min_{\alpha_1,\ldots,\alpha_n} \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j K_{ij} - \sum_{i=1}^{n} \alpha_i \qquad \forall_i \; 0 \leq \alpha_i \leq C$$
$$\forall_i \; \sum_{i=1}^{n} \alpha_i y_i = 0$$
$$\text{DUAL}$$

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

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

One problem with the dual form of the kernel SVM is that we lose b, however we need it for classification. Fortunately, we know that the optimum solution for both equations are the same.

In the dual function, only α values of ≥ 0 are support vectors. From the constraints from primal solution

$$y_i(w^T \phi(x_i) + b) = 1$$

$$y_i\left(\sum_{j=1}^{s} y_j \alpha_j k(x_j, x_i) + b\right) = 1$$

y_i can either be 1 or -1

for $y_i \left(\sum_j y_j \alpha_j k(x_j, x_i) \right)$ to be equal to 1

either y_i = positive and $\sum_j y_j \alpha_j k(x_j, x_i)$ = positive

or y_i = negative and $\sum_j y_j \alpha_j k(x_j, x_i)$ = negative

therefore

$$\sum_j y_j \alpha_j k(x_j, x_i) + b = y_i$$

$$\Rightarrow b = y_i - \sum_j y_j \alpha_j k(x_j, x_i)$$

Smart Nearest Neighbour

For binary classifications, $y_i \in \{+1, -1\}$, we can write the decision function for test point z as

$$h(z) = \text{sign}\left(\sum_{i=1}^{\hat{n}} y_i \delta^{nn}(x_i, z) \right)$$

where $\delta^{nn}(z, x_i) \in \{0, 1\}$ with $\delta^{nn}(z, x_i) = 1$ only if x_i is one of the k-nearest neighbours of test point z, and 0 otherwise.

The SVM function

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

is very similar, but instead of limiting the decision to the k nearest neighbours, it considers all training points.

The regular SVM decision function assigns more weight to those points that are closer (large $k(z, x_i)$). In some sense, the RBF kernel can be viewed as a soft nearest neighbor assignment, as the exponential decay with distance will assign almost no weight to all those except neighbouring points of z. The kernel SVM algorithm also learns a weight $\alpha_i > 0$ for each training point and a bias b, and it essentially "removes" useless training points by setting many $\alpha_i = 0$.

Gaussian Processes

First, we need to review the definition and properties of the Gaussian distribution.

A gaussian random variable $X \sim \mathcal{N}(\mu, \Sigma)$, where μ is the mean and Σ is the co-variance matrix has the following probability density function.

$$P(x; \mu, \Sigma) = \frac{1}{(2\pi)^{d/2} |\Sigma|} e^{-\frac{1}{2}((x-\mu)^T \Sigma^{-1} (x-\mu))}$$

where $|\Sigma|$ is the determinant of Σ. The gaussian distribution occurs very often in real world data. This is because of the central limit theorem. The CLT states that the arithmetic mean of $n > 0$ samples is approximately normally distributed - independent of original sample distribution.

A gaussian distribution has the following properties:

1/ Normalisation

$$\int_y p(y; \mu, \Sigma) dy = 1$$

2/ Marginalisation

If we have two gaussian distributions and draw a value from each...

$$y = \begin{bmatrix} y_A \\ y_B \end{bmatrix}$$

where the means of the gaussians are $\mu = \begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}$

and the covariance matrix is $\Sigma = \begin{bmatrix} \Sigma_{AA}, \Sigma_{AB} \\ \Sigma_{BA}, \Sigma_{BB} \end{bmatrix}$

The marginal distributions $p(y_A) = \int_{y_B} p(y_A, y_B; \mu, \Sigma) dy_B$

and $p(y_B) = \int_{y_A} p(y_A, y_B; \mu, \Sigma) dy_A$ are gaussian:

3/ Summation

If $y \sim N(\mu, \Sigma)$ and $y' \sim N(\mu', \Sigma')$

then $y + y' \sim N(\mu + \mu', \Sigma + \Sigma')$

4/ Conditioning

The conditional probability of y_A on y_B is

$$p(y_A | y_B) = \frac{p(y_A, y_B; \mu, \Sigma)}{\int_{y_A} p(y_A, y_B; \mu, \Sigma) dy_A}$$

is also gaussian

$$y_A \mid y_B = y_B \sim \mathcal{N}\left(\underbrace{\mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}(y_B - \mu_B)}_{\text{mean}}, \underbrace{\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA}}_{\text{standard deviation}}\right)$$

Posterior Predictive Distribution

Consider a regression problem

$y = f(x) + \varepsilon$ where ε is Gaussian Noise

$y = w^T x + \varepsilon$ OLS and Ridge Regression

$y = w^T \phi(x) + \varepsilon$ Kernel Ridge Regression

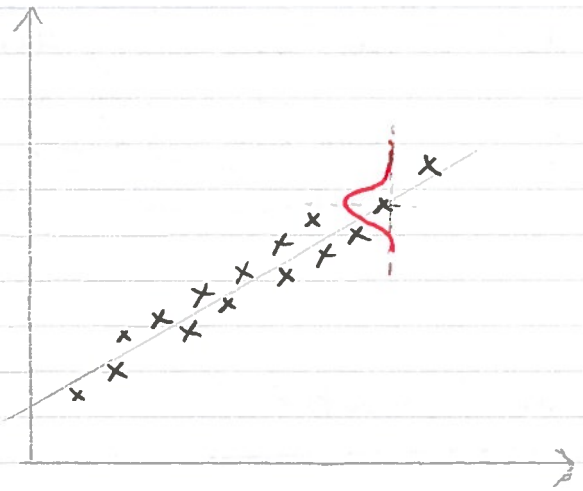
Remember the maximum likelihood estimation is trying to estimate the probability of the data given our parameters)

$$P(D|w) = \prod_{i=1}^{n} (y_i | x_i, w)$$

And the maximum a posterior approach is the opposite

$$P(w|D) \propto \frac{P(D|w)P(w)}{P(D)}$$ ($P(w)$: prior)

If the term $P(y_i | x_i, w)$ is drawn from a Gaussian distribution, such that it is equal to $\mathcal{N}(w^T x, \sigma^2 I)$



Around each point, we assume that the data was measured and taken from some small gaussian function around the point.

The mean of the Gaussian at each point is simply $w^T x$

Usually when performing regression, we want to find our line (w) and use it to make predictions. So using our data D, and finding our line (w), we can make predictions about test point x using $w^T x$.

But turns out there is a clever Bayesian method to help out. The assumption is that the value of w isn't actually of interest, we only care about the quality of the prediction.

So rather than model w so we can then model the test point, we would model the test point directly from the start. This means we want the probability of the label given our test point and our data, $P(y \mid x, D)$. This makes no assumption about the model!

To get this probability, we start by marginalising the model.

$$P(y \mid x, D) = \int_w \underbrace{P(y \mid x, w)}_{\text{Gaussian}} P(w \mid D)\, dw$$

$$= \int_w P(y \mid x, D, w)\, dw$$

} same thing.

$$P(y|x,D) = \int P(y|x,w)\,\frac{P(D|w)P(w)}{Z}\,dw$$

Gaussian! — Gaussian — Gaussian — Gaussian — some normalisation — Bayes + Gaussian

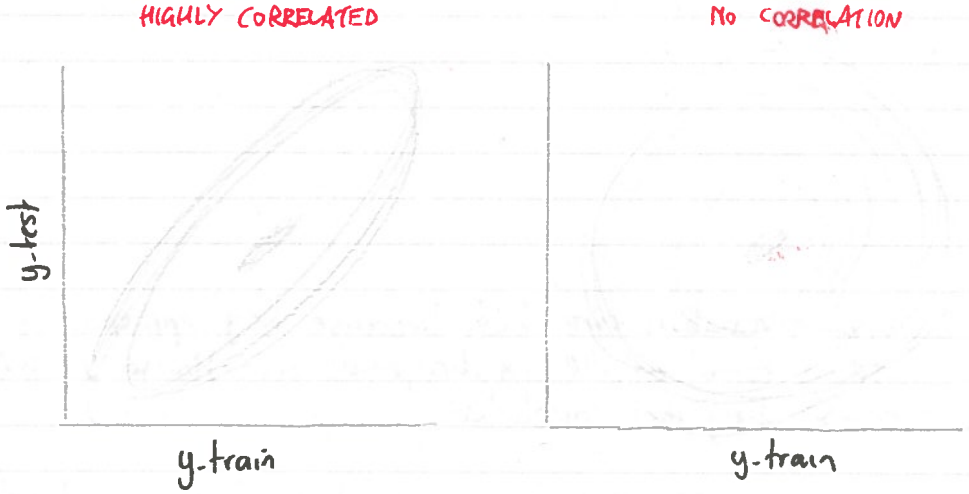
Using the properties of Gaussians, we can prove that the above equation for $P(y|x,D)$ is Gaussian. By integrating over every possible model and averaging every possible result, we still get a Gaussian.

This means we can make predictions without ever having to learn a model, because we marginalised the model out.

We know that $P(y|x,D)$ is gaussian so we can write

$$P(y|x,D) = P\left(\begin{bmatrix} y_1 \\ \dots \\ y_n \end{bmatrix} \middle| [x_1,\dots,x_n], x\right) \sim \mathcal{N}(\mu, \Sigma)$$

Therefore if we have labels (one test, one train), the value of one could tell us about the other. Imagine we have two cases: one where labels are highly correlated, and ones that are not



Suppose the labels represent house prices. In the first case, the houses are found close together and therefore have a similar price. If the price of the test point is higher, it will mean the price of the training point is higher.

In the second case, the houses are very far apart, such that the price of one doesn't effect the other.

Where do we specify how these are correlated? To do this, we need to set up our co-variance matrix Σ. In this case, it is a 2D matrix where the diagonal is the variance of the test point itself, and the off diagonals represent the correlation with other labels.

Consider the case where we have 3 houses; two are close together and one is far away and much more expensive. The co-variance matrix will look like

$$\Sigma = \begin{bmatrix} 1 & 0.9 & 0 \\ 0.9 & 1 & 0 \\ 0 & 0 & 10 \end{bmatrix}$$

House 1 / House 2 } Highly correlated

Expensive House far away

House 1 | House 2 | Expensive House far away

The blue numbers are high because they represent a high correlation. These are found on the off diagonals for House 1 and House 2 because they are correlated.

The red number shows no correlation, and are therefore 0. Because there are zeros in the rows and columns belonging to the expensive house, we know the label of the expensive house has no correlation with any other labels.

In order for the covariance matrix to be valid, it must be positive semi-definite. As this is also a condition for kernels, it turns out that this covariance matrix is also a kernel.

We want the off-diagonal in the covariance matrix to be high for highly correlated points, and low for uncorrelated points. This can be achieved using the RBF kernel.

$$k(x,z) = e^{-\frac{(x-z)^2}{\sigma^2}}$$

Gaussian Process Regression

We assume that, before we observe the training labels, that labels are drawn from a zero-mean prior gaussian

$$P(y|x) \sim \mathcal{N}(0,\Sigma) \quad \text{where } \Sigma = K \quad \text{where } K_{i,j} = k(x_i, x_j)$$

$$\sim \mathcal{N}(K_*^T K^{-1} y,\ K_{**} K^{-1} K_*)$$

and Σ is decomposed as $\Sigma = \begin{bmatrix} K & K_* \\ K_*^T & K_{**} \end{bmatrix}$

K = training kernel matrix

K_* = training-testing kernel matrix

K_{**} = testing kernel matrix

And $K_*^T K^{-1} y$ is the same as kernel regression. However, we also get the variance which was not available in kernel regression.

KD Trees

The time complexity of the k nearest neighbour algorithms is very high. If we have d dimensional data, and we add one more point, the time added is $O(d)$.

If there are n data points in the training dataset, the time complexity is $O(nd)$.

Classifying one test input is also $O(nd)$.

To achieve the best accuracy, we want $n \gg 0$, but this will soon limit our ability to perform the algorithm.

Can we make kNN faster during testing?

The general idea of KD-Trees is to partition the feature space. We want to discard lots of data points immediately because their partition is further away than our k nearest neighbours.

We achieve this by:
- Dividing our data into two halves along one feature (left and right)
- For each training input, remember which half it lies in.

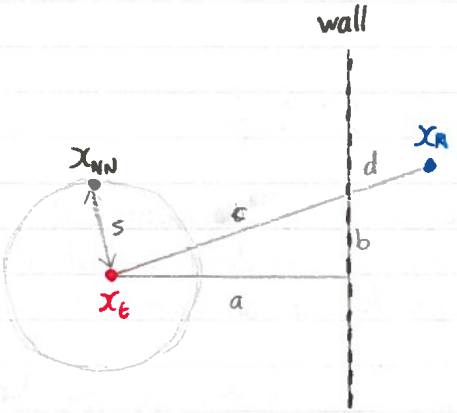
Suppose we use $k=1$, we then:
- identify which side the test point lies in
- Find the nearest neighbour x^R_{NN} of x_t in the same side
- compute the distance between x_t and the dividing wall (d_w)
- If $d_w > d(x_t, x^R_{NN})$ then we can discard the other half

$\Rightarrow$ Giving an ~~increase~~ decrease in computation time by factor 2

However, if $d_w < d(x_t, x^R_{NN})$ then all data must be used so there is no change in computation time.

If the distance to the partition is larger than the distance to the closest neighbour, we know that none of the data points inside that partition can be closer.

Can we prove that if a nearest neighbor is closes to the test point that the dividing wall, than any new point on the other side of the wall is always further away than the nearest neighbour ?



Prove that $s \leq c+d$

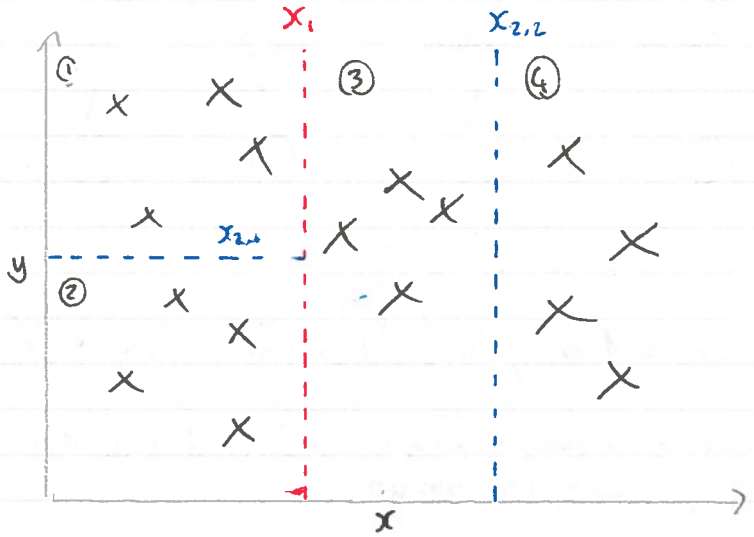
$c + d = \sqrt{a^2 + b^2} + d$

since $b^2 \geq 0$ then $c + d \geq \sqrt{a^2} + d$

since $d \geq 0$ then $c + d \geq a$

since $a \geq s$ $c + d \geq s$

KD - Tree data structure



In order to continually split the data in half into trees, we follow the following algorithm.

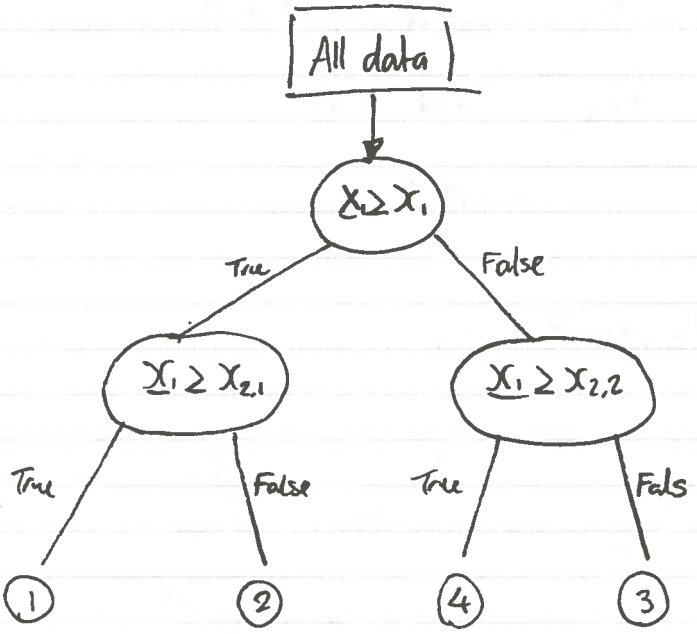
1. Identify the dimension with the highest variance
2. calculate the median in this dimension
3. split the data along the median

With n steps, there will be 2^n leafs.

In the diagram on page 86, x_1 is the initial cut. The highest variance is in the x direction so the data is cut at the median so the same number of data points are on each side.

In step 2, the data is cut at $x_{2,1}$. In this section, the highest variance is in the y-direction. The other half is also cut at $x_{2,2}$.

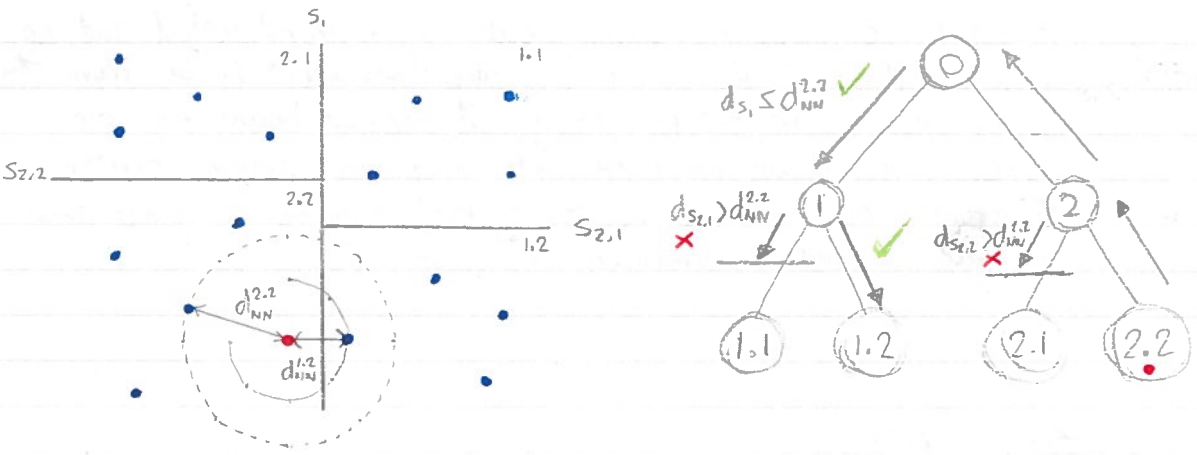
We can illustrate this as a tree.



Pros: Exact and easy to build. Time to test $O(\log_2 n)$ with n steps.

Cons: Curse of dimensionality (ineffective at high dimensions)
All splits are axis aligned.

After creating our tree, how do we find the nearest neighbour to a new test point?



Suppose our data is split into 4 sections, and our new test point (red) is in section 2.2

First, the nearest neighbour to the test point is found in section 2.2. The distance between these points is called $d_{NN}^{2.2}$.

Then we go up one level in the tree. We then check if the distance to the seperating wall $S_{2,2}$ is smaller than $d_{NN}^{2.2}$. If $d_{NN}^{2.2}$ is smaller then we can prune this leaf and move up another level in the tree.

Now check if $d_{NN}^{2.2}$ is smaller than the distance to the separating wall S_1. As the distance to S_1 is smaller than $d_{NN}^{2.2}$, we must descend into this branch.

Then we check if the distance to the partition inside this branch is smaller than $d_{NN}^{2.2}$. As the distance to $S_{2,1}$ is greater than $d_{NN}^{2.2}$, we can prune this leaf to avoid going into section 1.1.

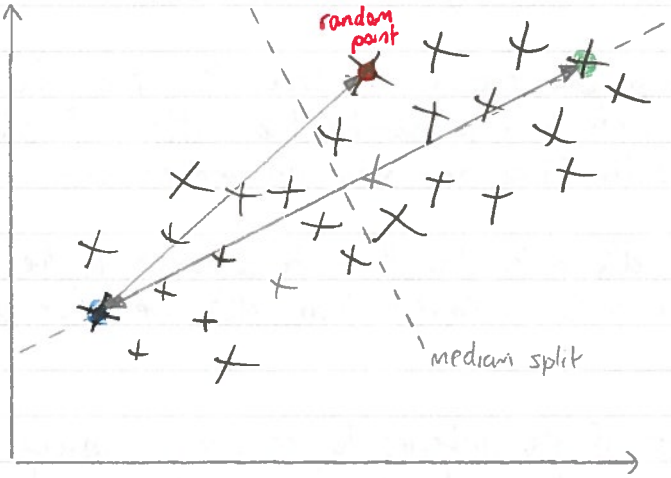
Now we check if any points inside section 1.2 are smaller than $d_{NN}^{2.2}$. In this case, there is a new point which is closer inside section 1.2, so this is our new nearest neighbour.

Generally this tree search is quicker than the standard k nearest neighbours. However, there are certain times when it is worse.

In the worse case, every leaf needs to be investigated and so is slower than k-NN as the tree algorithm needs to be ran. This occurs when points are far away and decision boundaries are closer. This occurs when the data has high dimensionality. Therefore, kD Trees are useless in high dimensions. and should probably be used in dimensions less than 10.

Ball-Trees

Ball trees extend from KD trees, but avoid the problem caused by boundary splits having to be axis aligned.



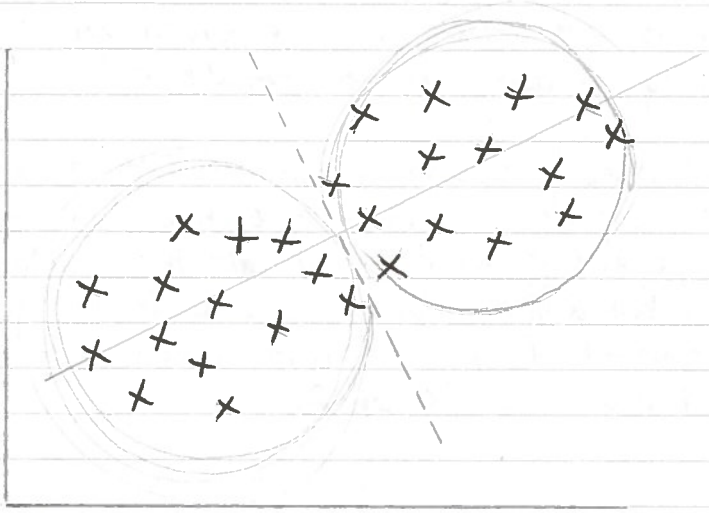
Suppose we have the data set shown above. Similarly to last time, we want to start by splitting the data in half. However there is a linear trend so splitting by a plane aligned to the axis is not desirable. Instead we want to split the data along some arbitrary axis which describes the most variance.

Usually this is done using PCA, but there is a simple estimation which produces decent estimations of this.

The first step is to choose a random point in the dataset (red) and find the most distant point (blue). Then find the most distant point from the blue (green), and use the connecting line between green and blue as the axis which describes the most variance. This is much faster than PCA and results are comparible.

A median split is then performed along this new axis so exactly half the total points are on each side.

The next step is to find the centre of mass of all points on either side of the boundary and then find the distance to the furthest point from the centre in the same half. A hypersphere is then constructed located at the centre of mass with a radius of the distance to the furthest point.



This process is performed for multiple splits. It doesn't matter that the axis change with each step because using a hypersphere makes distances to other spheres independent of direction.

Also, as the axis can change with each step, it allows the algorithm to capture the exact shape of the data.

NOTE If a test point lies in the intersection between 2 spheres, it is put into the leaf with the closest centre of mass.

Decision Trees

Decision trees are an optimised form of KD trees which makes a few assumptions and tradeoffs in order to decrease processing time.

Firstly, there is no back tracking. Once a test point is put into a leaf, it will not look at any other leafs to find a better nearest neighbour. This massively speeds up the algorithm.

Next, rather than storing data points in a leaf to then perform a nearest neighbour search, the number of labels is stored instead so a probability of the label is calculated instead. If all points were crosses, then the nearest neighbour must be a cross so we don't need to compute distances.

To avoid mixed classes in a leaf, the data could be split until only one class is present in each leaf, However, this is only possible if there are not two points with identical features but different labels. In order to avoid this, a small amount of gaussian noise is added to every feature so no two are ever the same.

However, this leads to overfitting on the training data. High depth generally means high variance, and low depth means high bias. Ideally, we want to find the smallest tree that does a decent job.

Impurity Function

Impurity functions measures the mix of labels in a leaf. To achieve a high number of similar labels in a leaf, we want to minimise the impurity function.

Data $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$

where $y_i \in \{1, 2, \ldots, c\}$ let c be number of classes.

Gini Impurity

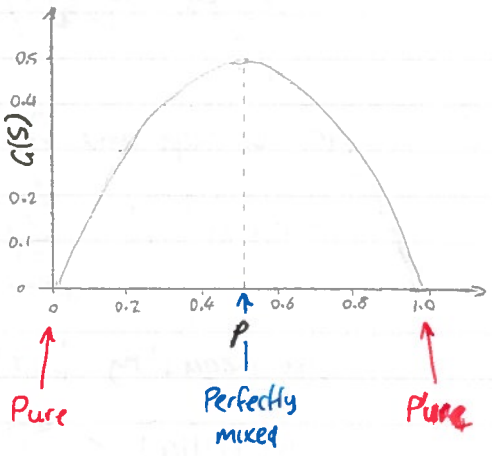
Let $S_k \subseteq S$ where $S_k = \{(x, y) \in S : y = k\}$

The probability of finding label k is:

$$p_k = \frac{|S_k|}{|S|}$$

The impurity is given by

$$G(S) = \sum_{k=1}^{c} p_k (1 - p_k)$$



Entropy

Let $p_1, \ldots, p_k$ be defined as before. We know that we don't want a uniform distribution such that $p_1 = p_2 = \ldots = p_k = 1/k$. This is the worse case as each leaf is equally likely, and prediction essentially becomes random guessing.

We could define entropy as how close we are to uniform, which can be calculated using the KL-Divergence.

Let $q_1, \ldots, q_c$ be the uniform label/distribution ie $q_k = \frac{1}{c} \; \forall k$

$$KL(p \| q) = \sum_{k=1}^{c} p_k \log\left(\frac{p_k}{q_k}\right)$$

The value of $KL(p||q)$ is ≥ 0

PROOF:

$$KL(p||q) = \sum_{k=1}^{c} p_k \log\left(\frac{p_k}{q_k}\right)$$

~~note that $\log(x) \leq x-1$~~

~~$\sum_{k=1}^{c} p_k \log\left(\frac{p_k}{q_k}\right) \leq \sum_{k=1}^{c} p_k \left(\frac{p_k}{q_k} - 1\right)$~~

~~$\sum_{k=1}^{c} p_k \log\left(\frac{p_k}{q_k}\right) \leq \sum_{k=1}^{c} p_k ($~~

reverse sign and flip log

$$-KL(p||q) = \sum_{k=1}^{c} p_k \log\left(\frac{q_k}{p_k}\right)$$

use inequality $\ln(x) \leq x-1$

$$-KL(p||q) \leq \sum_{k=1}^{c} p_k \left(\frac{q_k}{p_k} - 1\right)$$

$$-KL(p||q) \leq \sum_{k=1}^{c} \frac{p_k q_k}{p_k} - p_k$$

$$-KL(p||q) \leq \sum_{k=1}^{c} q_k - p_k$$

$$-KL(p||q) \leq \sum_{k=1}^{c} q_k - \sum_{k=1}^{c} p_k$$

$$-KL(p||q) \leq 1 - 1$$

$$-KL(p||q) \leq 0$$

$$KL(p||q) \geq 0$$

Using q as the distribution where all classes are equally likely

i.e $q_k = \frac{1}{C} \; \forall k$

We plug in $q_k = \frac{1}{C}$ into the entropy (KL-divergence) equation

$$KL(p||q) = \sum_{k=1}^{C} p_k \log\left(\frac{p_k}{q_k}\right) = \sum_{k=1}^{C} p_k \log(p_k) - p_k \log(q_k)$$

$$= \sum_{k=1}^{C} p_k \log(p_k) + p_k \log(C)$$

$$= \sum_{k=1}^{C} p_k \log(p_k) + \underbrace{\log(C)}_{\text{constant}} \overbrace{\sum_{k=1}^{C} p_k}^{=1}$$

We now want to maximise this equation. We can drop the $\log(C)$ as it is a constant and has no effect on finding the max.

$$\underset{p}{\text{argmax}} \sum_{k=1}^{C} p_k \log(p_k)$$

$$= \underset{p}{\text{argmin}} -\sum_{k=1}^{C} p_k \log(p_k) = \underset{p}{\text{argmin}} \; H(S) \quad \leftarrow \text{entropy}$$

If a tree containing data S is split into left (S_L) and right (S_R) branches, the new total entropy is

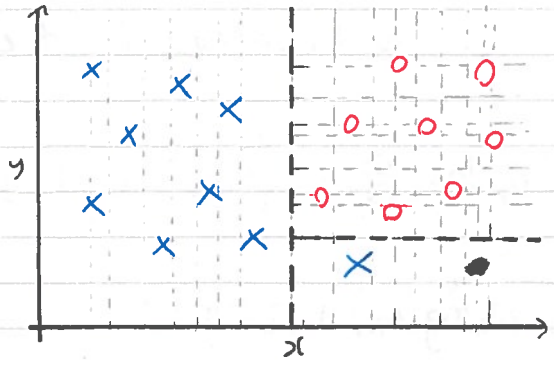
$$H(S) = p^L H(S_L) + p^R H(S_R)$$

where $p^L = \frac{|S_L|}{|S|}$ and $p^R = \frac{|S_R|}{|S|}$

This is essentially the weighted average of the entropy in all leafs.

We want to find the tree with the lowest entropy to best separate our classes. We can do this using the ID3 algorithm.

ID3 Algorithm



The ID3 algorithm is a technique to chose ophmum splits in order to minimise the entropy of classes in leaves. In order to make this achievable, the splits are axis aligned. This doesn't guarantee the best split, but does a very good job.

The ID3 algorithm first chooses an axis to begin splitting. The algorithm is greedy and calculates the entropy for splits between every possible point (dashed pencil lines on above figure). The split with the lowest entropy is chosen.

If the splits mean that each leaf contains only contains one class, the algorithm will stop. Otherwise the same split will be performed in more dimensions only in impure leaves.

The complexity of performing the ID3 algorithm can be calculated by first finding the number of total possible splits. As boundaries are drawn between all N points in D dimensions, there are $\underline{ND}$ different boundaries.

Each of these boundaries are assessed by finding the entropy they produce, where the calculation is of complexity kN as each class is assessed for all possible points. The overall complexity of the ID3 function is therefore

$$O(DkN^2) = O(ND \cdot kN)$$

boundaries — entropy calculation

Therefore the calculation time scales with size of the dataset squared, which is not ideal. However, there is a simple trick which allows us to speed up the algorithm.

To do this, we need to frame the entropy calculation differently. Instead of viewing the boundary moving between two points and having to recalculate the entropy, we can view it as simply the next point switching sides of the boundary and using our precalculated probabilities to speed up entropy calculation. The complexity of the entropy calculation then simply becomes k such that

$$O(DN \cdot k) = O(DNk)$$

(D: boundaries, k: entropy calculation)

Now the algorithm is scaling linearly with dataset size which is much more acceptible.

Regression Trees

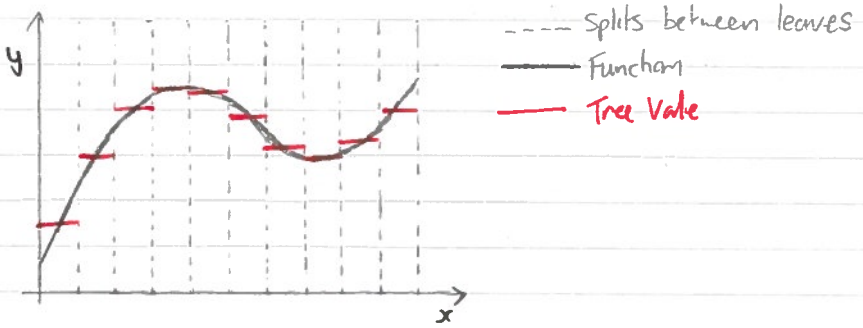
These are often called Classification and Regression Trees (CART)

Assume that labels are continuous $y_i \in \mathbb{R}$.

The impurity in CART is the squared loss

$$L(S) = \frac{1}{|S|} \sum (y - y_S)^2$$ ← average squared difference from average label.

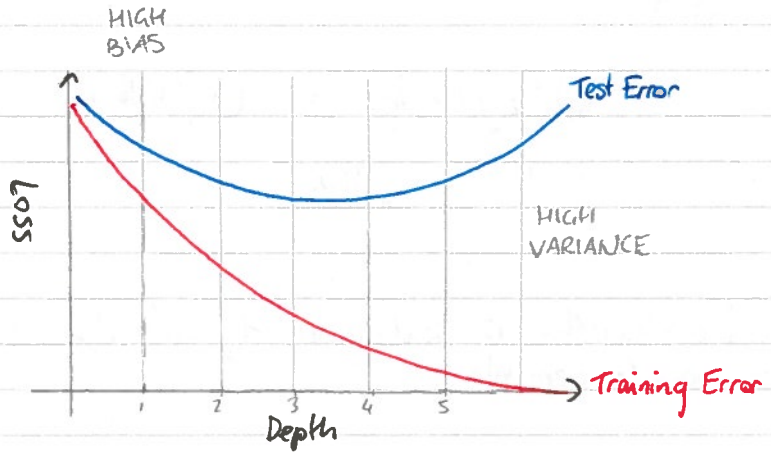
where $y_S = \frac{1}{|S|} \sum y$ ← average label



To summarise, CART are very lightweight classifiers and very fast during testing. Although they usually aren't the best classifiers unless some additional techniques are used to help (bagging & / boosting).

There is a trade-off in choosing the number of splits for CART. A large number of splits will minimise the training error to zero, but that is because the model overfits and the test results will be poor.

Typically, the training/test errors with depth follows this form.



With decision trees, the depth must go in integer steps. This means this function is difficult to minimise in the traditional way. The "sweetspot" may lie between depths but we can't possibly do this.

Rather than changing the depth to solve the bias/variance trade-off, we try alternate techniques. One is to accept we have either a high variance or high bias issue with our model and then use additional techniques to fix the problem.

In the case of high variance in CART we apply a technique known as "Bagging".

Bagging

Remember the bias/variance decomposition:

$$\underbrace{\mathbb{E}[(h_D(x)-y)^2]}_{\text{ERROR}} = \underbrace{\mathbb{E}[(h_D(x)-\bar{h}(x))^2]}_{\text{VARIANCE}} + \underbrace{\mathbb{E}[(\bar{h}(x)-\bar{y}(x))^2]}_{\text{BIAS}} + \underbrace{\mathbb{E}[(\bar{y}(x)-y(x))^2]}_{\text{NOISE}}$$

Our goal is to reduce the variance $\mathbb{E}[(h_D(x)-\bar{h}(x))^2]$

For this we want $h_D \to \bar{h}$.

Reminder: The weak law of large numbers says that for i.i.d random variables x_i with a mean $\bar{x}$, we have

$$\frac{1}{m}\sum_{i=1}^{m} x_i \to \bar{x} \quad \text{as} \quad m \to \infty$$

We can apply this to classifiers such that we have m training sets $D_1, \dots, D_2, \dots, D_n$ drawn from P^n. We can train a classifier on each one an average the result.

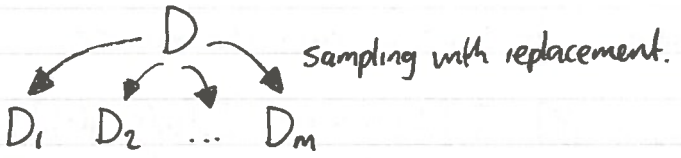
$$\hat{h} = \frac{1}{m}\sum_{i=1}^{m} h_{D_i} \to \bar{h} \quad \text{as} \quad m \to \infty$$

We refer to such an average of multiple classifiers as an ensemble of classifiers.

Therefore if $\hat{h} \to \bar{h}$ the variance component of the error disappears. However, we don't have m datasets, we only have one.

We can simulate more datasets using a technique called bootstrapping, or bagging.

Simulate drawing from P by drawing uniformly with replacement from the set D.



$Q(XY|D)$ is a probability distribution that picks a training sample (x_i, y_i) from D uniformly at random, i.e.

$$Q((x_i,y_i)|D) = \frac{1}{n} \quad \forall (x_i,y_i) \in D \quad \text{where} \quad n = |D|$$

The bagged classifier is the average classifier from all those which have been trained on $D_1, D_2, \ldots, D_m$.

ie $$\hat{h}_D = \frac{1}{m} \sum_{i=1}^{m} h_{D_i}$$

However, the bagged classifier doesn't coverge to h as the random variables are identically and independently distributed so the weak law of large numbers doesn't work. However, in practise bagging reduces variance very effectively.

Although we cannot prove that the new samples are i.i.d, we can show that they were drawn from the original distribution P. Assume P is discrete, with $P(X = x_i) = p_i$, over some set $\Omega = x_1, x_2, \ldots, x_N$

Q is our new data set ~~with~~ created by sampling with replacement from P, and can be written as

$$Q(X = x_i) = \sum_{k=1}^{n} \binom{n}{k} p_i^k (1-p_i)^{n-k} \frac{k}{n}$$

probability that there are k copies of x_i in D

Probability that one of these copies are picked.

$$Q(X=x_i) = \frac{1}{n}\sum_{k=1}^{n}\binom{n}{k}p_i^k(1-p_i)^{n-k}\,k = \frac{1}{n}\mathbb{E}(\mathbb{B}(p_i,n))$$

Expected value of binomial distribution with parameter p_i

$$Q(X=x_i) = \frac{1}{n}\mathbb{E}(\mathbb{B}(p_i,n)) = \frac{1}{n}(np_i) = p_i$$

$\therefore$ Each dataset is drawn from P, but not independently.

$$\Rightarrow Q(X=x_i) = P(X=x_i)$$

Summary

- Sample m dataset $D_1, D_2, \ldots, D_m$ from D without replacement.
- For each D_j, train a classifier h_j
- The final classifier is $h(x) = \frac{1}{m}\sum_{j=1}^{m} h_j(x)$

In practice, larger m results in a better ensemble, however at some point it will give diminishing returns. Setting m too high will increase computation time, but not give better classification.

- Easy to implement
- Reduces variance on high variance classifiers
- Able to obtain a mean and standard deviation on model parameters.
- Provides unbiased estimate of the test error.

Random Forest

One of the most useful bagged algorithms is the Random Forest. RF is essentially a bagged decision tree with slightly different splitting criteria. The algorithm follows:

1/ Sample m datasets $D_1, D_2, \ldots, D_m$ from D with replacement.

2/ For each D_j, train a full decision tree h_j (max-depth = ∞). However, before each split randomly subsample $k \leq d$ features without replacement and only consider these for the split.

3/ The final classifier is $h(x) = \frac{1}{m} \sum_{j=1}^{m} h_j(x)$

The random forest algorithm is one of the best "out of the box" algorithms. Firstly because the model has only two hyper-parameters, m and k and it is well understood how to set these. For k, it has been demonstrated that $k = \sqrt{d}$ where d is the number of features is optimal, and m just needs to be as large as you can afford.

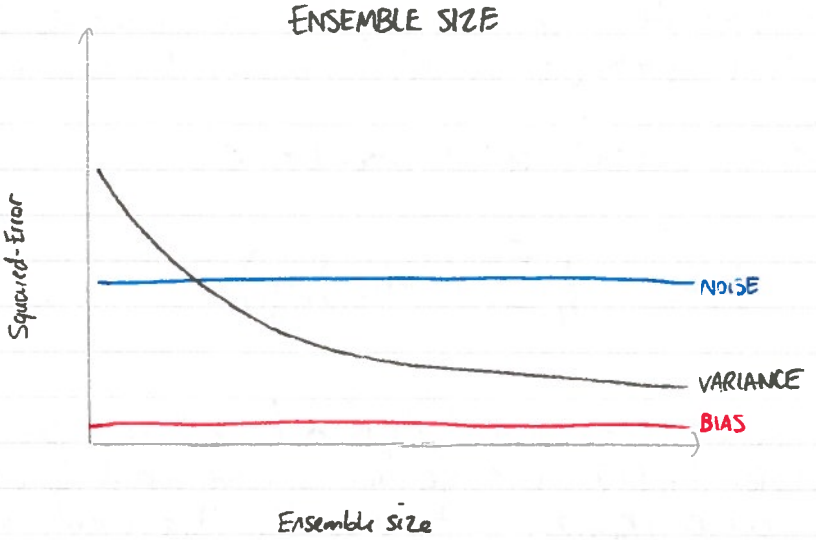
Secondly, features can be numerical, categorical, discrete, etc. and can have completely different scales, magnitudes or slopes and the algorithm will still work.

Additionally, RF is able to provide an estimate of the test error without a testing dataset. This is also known as the out of bag error. This means a training/test split is not needed.

As $D_1, D_2, \ldots, D_m$ are subsets of D sampled with replacement, some datasets will not contain a lot of points. Suppose some point (x_i, y_i) was not included in half of the subsets of D, the point (x_i, y_i) can be tested in these trees and an error can be calculated for this point. We can extend this and test all points on trees which didn't include them in training and average to find the error.

$$\mathcal{E}_{OOB} = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{z_i} \sum_{\substack{j \\ (x_i, y_i) \notin D_j}} L\left(h_j(x_i), y_i\right), \qquad z_i = \sum_{\substack{j \\ (x_i, y_i) \notin D_j}} 1$$

RANDOM FOREST ERROR WITH ENSEMBLE SIZE



As ensemble size increases, the variance of the error decreases due to the effect of bagging. The bias is not effected by averaging the classifier, as it is only a property of the average classifier itself. i.e. having more averages of more averages doesn't make any difference.

Bagging can be applied to everything, but its only useful if the model has a variance problem.

Boosting

Boosting is a method used when a problem has a particularly high bias (high training and test error).

The model used in this problem is known as a "weak learner.", as it results in a high training and testing error. The question becomes "can weak learners be combined to generate a stronger learner with low bias?"

If we create an ensemble classifyer from many weak learners.

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

where $h_t(x)$ is our weak learner and $H_T(x)$ is our strong learner.

The ensemble $H_t(x)$ is built in an iterative fashion, such that at iteration t we add $\alpha_t h_t(x)$ to the ensemble.

The loss function of the ensemble is

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

Assume we have finished iteration t and have our ensemble classifier $H_t(x)$. With iteration $t+1$ we want to add another weak learner ($h_{t+1}(x)$) to the ensemble. We want to search for the weak learner which minimises the loss the most.

$$h_{t+1} = \underset{h \in \mathbb{H}}{\operatorname{argmin}}\ l(H_t + \alpha h_t)$$

where $\mathbb{H}$ is our set of weak learners. Once h_{t+1} has been found, we add this to our ensemble.

$$H_{t+1} = H_t + \alpha h_t$$

How can we find the $h \in \mathbb{H}$? We use gradient descent in function space. We start by using the taylor approximation.

$$l(H + \alpha h) \approx l(H) + \alpha \left\langle \frac{\partial l}{\partial H}, h \right\rangle$$

(inner product; $l(H)$: constant)

$$\Rightarrow h_{t+1} = \underset{h \in \mathbb{H}}{\operatorname{argmin}} \left\langle \frac{\partial l}{\partial H}, \alpha h \right\rangle$$

$$= \underset{h \in \mathbb{H}}{\operatorname{argmin}} \sum_{i=1}^{n} \frac{\partial l}{\partial H(x_i)} h(x_i)$$

where $\frac{\partial l}{\partial H(x_i)} = H(x_i) - y_i \quad \Rightarrow l(H) = \frac{1}{2} \sum_{i=1}^{n} (H_t(x_i) - y_i)^2$

Gradient Boosted Regression Tree (GBRT)

- Classification ($y_i \in \{-1, +1\}$) or regression ($y_i \in \mathbb{R}^k$)
- Weak learners, $h \in \mathbb{H}$, are regressors (CART), $h(x) \in \mathbb{R}\ \forall x$ typically with a fixed depth (4-6 splits). $\Rightarrow$ HIGH BIAS
- step size α is fixed to a small constant. (hyper-parameter)
- Loss function: Any differentiable convex loss that decomposes over the samples. $l(H) = \sum_{i=1}^{n} l(H(x_i))$

In order to use regression trees for gradient boosting, we must find a tree, h, that minimises...

$$h = \underset{h \in \mathbb{H}}{\operatorname{argmin}} \alpha \sum_{i=1}^{n} r_i h(x_i)$$

$$\text{where } r_i = \frac{\partial l}{\partial H(x_i)}$$

We can rewrite this by first using the negative gradient $t_i = -r_i$

$$h = \underset{h \in \mathbb{H}}{\operatorname{argmin}} - \sum_{i=1}^{n} t_i h(x_i)$$

Then add a factor of 2. This doesn't effect the answer but will make the maths easier. Also can drop α as its a constant.

$$h = \underset{h \in \mathbb{H}}{\operatorname{argmin}} - 2 \sum_{i=1}^{n} t_i h(x_i)$$

Next we assume (but can prove) that $\sum_i t_i^2 =$ constant and independent of h, so we can add it to the equation without consequence.

$$h = \underset{h \in \mathbb{H}}{\operatorname{argmin}} - 2\sum_{i=1}^{n} t_i h_i(x) + t_i^2$$

Next we assume that $\sum_{i=1}^{n} h^2(x_i) = \text{constant}$. This is not strictly true but it helps the maths. We can add this to the equation

$$h = - \underset{h \in \mathbb{H}}{\operatorname{argmin}} \sum_{i=1}^{n} h^2(x_i) - 2t_i h_i(x_i) + t_i^2$$

$$\Rightarrow \quad h = - \underset{h \in \mathbb{H}}{\operatorname{argmin}} \sum_{i=1}^{n} (h(x_i) - t_i)^2$$

In otherwords, we can use regression trees and feed in the value of t_i as labels for each x_i. Each iteration will build a new tree for a different set of "labels" $t_1, t_2, \ldots, t_n$

PSEUDO CODE:

~~Input~~

$H = 0$
For $t = 1 : T$
  $t_i = y_i - H(x_i)$
  $h = \operatorname{argmin}_{h \in \mathbb{H}} (h(x_i) - t_i)^2$
  $H \leftarrow H + \alpha h$
end
return H

Ada Boost

Stands for "Adaptive Boosting."

- Setting: classification ($y_i \in \{-1, +1\}$)
- Weak Learners: $h \in \mathbb{H}$ are binary, $h(x_i) \in \{-1, +1\} \; \forall x$
- Stepsize: We perform a line search to obtain the best stepsize.
- Loss function: Exponential loss $l(H) = \sum_{i=1}^{n} e^{-y_i H(x_i)}$

The first step is to find the best weak learner. To do this, we need to define the gradient

$$r_i = \frac{\partial l}{\partial H(x_i)} = -y_i e^{-y_i H(x_i)}$$

To make the maths easier later on, we will define

$$w_i = \frac{1}{Z} e^{-y_i H(x_i)} \quad \text{where} \quad Z = \sum_{i=1}^{n} e^{-y_i H(x_i)}$$

such that $\sum_{i=1}^{n} w_i = 1$. Note Z is identical to the loss function.

Each weight w_i is therefore the relative contribution of the training points (x_i, y_i) towards the overall loss.

In order to find the best next weak learner, we need to solve the optimisation problem

$$h(x_i) = \underset{h \in \mathbb{H}}{\operatorname{argmin}} \sum_{i=1}^{n} r_i \, h(x_i)$$

substitute in $r_i = -y_i e^{-y_i H(x_i)}$

$$h(x_i) = \underset{h \in \mathbb{H}}{\operatorname{argmin}} -\sum_{i=1}^{n} y_i e^{-y_i H(x_i)} h(x_i)$$

next, we can substitute $w_i = \frac{1}{Z} e^{-y_i H(x_i)}$.

As Z is a constant, it doesn't effect the answer, so we can remove it from the substitution

$$h(x_i) = \underset{h \in \mathbb{H}}{\operatorname{argmin}} -\sum_{i=1}^{n} \underbrace{w_i y_i}_{r_i} h(x_i)$$

Note that $y_i h(x_i) = \{-1, +1\}$, so we sum over all points where $y_i h(x_i)$ is positive and then all points where $y_i h(x_i)$ is negative

$$h(x_i) = \underset{h \in \mathbb{H}}{\operatorname{argmin}} \underbrace{\sum_{i: h(x_i) \neq y_i} w_i}_{\text{weighted error } \varepsilon} - \underbrace{\sum_{i: h(x_i) = y_i} w_i}_{\text{weighted accuracy } (1-\varepsilon)}$$

$$h(x_i) = \underset{h \in \mathbb{H}}{\operatorname{argmin}} \; \varepsilon - (1 - \varepsilon)$$

$$= \underset{h \in \mathbb{H}}{\operatorname{argmin}} \; 2\varepsilon - 1$$

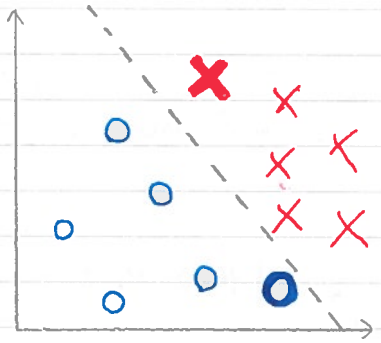
the factor of 2 and negative one are just constants so can be removed from argmin.

$$h(x_i) = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \; \varepsilon = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \sum_{i : h(x_i) \neq y_i} w_i$$

So the error is the sum of all the weights whose predicted labels are wrong. Below is an illustration of a decision boundary being decided using AdaBoost.

Initially, all points are given equal weights and a decision boundary is drawn. There are two points misclassified.

AdaBoost will assign greater weights to the misclassified points in order to get them right.



By assigning stronger weights to the misclassified points, emphasis is added to get them correct in the next iteration.

Finding Step-size α

In the previous example, GBRT, the step size alpha is set to some small value, typically around 0.1. As it turns out, in the AdaBoost setting, we can find the optimal step size (ie. the one that best minimises ℓ) in closed form every time we take a gradient step.

When we are given L, H, h we would like to solve the following optimization problem.

$$\alpha = \underset{\alpha}{\operatorname{argmin}}\ L(H + \alpha h)$$

again using the exponential loss we get

$$\alpha = \underset{\alpha}{\operatorname{argmin}} \sum_{i=1}^{n} e^{-y_i[H(x_i)+\alpha h(x_i)]}$$

differentiate with respect to α and set to zero.

$$-\sum_{i=1}^{n} y_i h(x_i) e^{-y_i[H(x_i)+\alpha h(x_i)]} = 0$$

as $y_i h(x_i) \in \{-1, +1\}$, we can split the equation into sums where prediction is correct $y_i h(x_i) = 1$, and where the prediction is incorrect $y_i h(x_i) = -1$.

$$-\sum_{i:h(x_i)y_i=1}^{n} e^{-y_i[H(x_i)+\alpha h(x_i)]} + \sum_{i:h(x_i)y_i=-1} e^{-y_i[H(x_i)+\alpha h(x_i)]} = 0$$

$$-\sum_{i:h(x_i)y_i=1}^{n} \overbrace{e^{-y_i H(x_i)}}^{w_i} e^{-\alpha} + \sum_{i:h(x_i)y_i=-1}^{n} \overbrace{e^{-y_i H(x_i)}}^{w_i} e^{\alpha} = 0$$

$$-\underbrace{\sum_{i:h(x_i)y_i=1}^{n} w_i}_{(1-\varepsilon)} e^{-\alpha} + \underbrace{\sum_{i:h(x_i)y_i=-1}^{n} w_i}_{\varepsilon} e^{\alpha} = 0$$

$$-(1-\varepsilon)e^{-\alpha} + \varepsilon e^{\alpha} = 0$$

$$(1-\varepsilon)e^{-\alpha} = \varepsilon e^{\alpha}$$

$$(1-\varepsilon) = \varepsilon e^{2\alpha}$$

$$\frac{(1-\varepsilon)}{\varepsilon} = e^{2\alpha}$$

$$\ln\left(\frac{1-\varepsilon}{\varepsilon}\right) = 2\alpha \Rightarrow \frac{1}{2}\ln\left(\frac{1-\varepsilon}{\varepsilon}\right)$$

This is the optimal step size in closed form, and this means that the AdaBoost algorithm can converge very fast.

Renormalisation

After you take a step, ie. $H_{t+1} = H_t + \alpha h$, we need to recompute all the weights and then renormalise, i.e. calculate

$$Z = \sum_{i=1}^{n} e^{-y_i H(x_i)}$$

We know that the weights become $w_i \leftarrow w_i e^{-\alpha h(x_i) y_i}$

and it can be shown that because of this, the normaliser will become

$$Z \leftarrow Z\, 2\sqrt{\varepsilon(1-\varepsilon)}$$

Combing the updated w_i and Z gives us

$$w_i \leftarrow w_i \frac{e^{-\alpha h(x_i) y_i}}{2\sqrt{\varepsilon(1-\varepsilon)}}$$

PSEUDO CODE:

$H_0 = 0$ ← zero classifier

$\forall i : \omega_i = \frac{1}{n}$ ← assign all points equal weight

for $t = 0 : (T-1)$ do

$h = A((\omega_1, x_1, y_1), \ldots, (\omega_n, x_n, y_n))$ ← get weak learner that minimizes the error using weights ω

$\varepsilon = \sum_{i : h(x_i) \neq y_i} \omega_i$ ← calculate error (weighted)

if $\varepsilon < \frac{1}{2}$ then ← if not perfectly perpendicular to $(h - y_i)$

$\alpha = \frac{1}{2} \ln\left(\frac{1-\varepsilon}{\varepsilon}\right)$ ← calculate optimum step-size

$H_{t+1} = H_t + \alpha h$ ← update ensemble

$\forall i : \omega_i \leftarrow \omega_i \frac{e^{-\alpha h(x_i) y_i}}{2\sqrt{\varepsilon(1-\varepsilon)}}$ ← recompute weights

else

return H_t ← if perfectly perpendicular to $(h - y_i)$

end

end

return H_T

As long as $\mathbb{H}$ is negation closed (this means that for every $h \in \mathbb{H}$, we must also have $-h \in \mathbb{H}$), it cannot be that the error is $\varepsilon > 1/2$. The reason is simply that if h has an error ε, it must be that $-h$ has an error of $(1 - \varepsilon)$. So we can just flip h to $-h$ for a smaller error.

The inner-loop stops if $\varepsilon = 1/2$, and it most cases this because this is an indicator it will converge to $\varepsilon = 1/2$ over time. When this happens, it is because the weak classifier is so weak its almost random.

Further Analysis

The weight update follows

$$w_i \leftarrow w_i e^{-\alpha h(x_i) y_i}$$

as $h(x_i)y_i$ is either -1 or $+1$, the weight is multiplied by either $e^{-\alpha}$ or e^{α} on each iteration.

The normalisation update follows

$$Z \leftarrow Z \times 2\sqrt{\varepsilon(1-\varepsilon)}$$

This normalisation Z is identical to the loss, we can therefore use it to bound the loss function after T iterations:

$$l(H) = Z = n \prod_{t=1}^{T} 2\sqrt{\varepsilon_t(1-\varepsilon_t)}$$

The factor n comes from the fact that the initial $Z_0 = n$ when all weights are equal to $1/n$.

If we define $c = \max_t \varepsilon_t$, we can establish that

$$l(H) \leq n\left[2\sqrt{c(1-c)}\right]^T$$

The function $c(1-c)$ is maximised at $c = 1/2$, but we know that each $\varepsilon_t < 1/2$. Therefore $c(1-c) < 1/4$.

We can rewrite this as $c(1-c) = 1/4 - \gamma^2$ for some γ

This leaves us with

$$\ell(H) \leq n \underbrace{(1-4\gamma^2)^{\frac{T}{2}}}_{<1}$$

This means the training loss decreases exponentially.

It is then possible to calculate how many steps are required to get a zero training error. The training loss is the upper-bound on the training error.

$$\text{Training Error} = \sum_{i=1}^{n} \delta_{H(x_i) \neq y_i}$$

and $\delta_{H(x_i) \neq y_i} < e^{-y_i H(x_i)}$ in all cases.

When the loss is less than 1, this implies that all points are correctly classified.

$$n(1-4\gamma^2)^{T/2} < 1$$

$$(1-4\gamma^2)^{T/2} < \frac{1}{n}$$

~~$\frac{T/2}{\log(1-4\gamma^2)} < \log\left(\frac{1}{n}\right)$~~

$$\frac{T}{2}\log(1-4\gamma^2) < \log\left(\frac{1}{n}\right)$$

$$T < \frac{2\log(1/n)}{\log(1-4\gamma^2)} \Rightarrow T > \frac{2\log(n)}{\log(1-4\gamma^2)}$$

This shows that after $O(\log(n))$ iterations, the training error must be zero. In practise it often makes sense to continue boosting even after the training error is zero.

Summary

Boosting is a great way to turn a weak classifier into a strong classifier. A few important things to know

- The step-size α is often refered to as "shrinkage"
- Sometimes gradient boosting isn't considered a boosting algorithm as there is no guarantee that the training error decreases exponentially. In this case they are classed as a stage-wise algorithm.
- AdaBoost is an extremely powerful algorithm, that turns any weak learner that can classify that can classify any weighted version of the training set with an error below 0.5, into a strong learner whoes training error decreases exponentially and only requires $O(\log(n))$ steps until training error is zero.

Neural Networks

Also known as "Deep Learning"

Originally named "Multi-Layer Perceptron" by Frank Rosenblatt in 1963 at Cornell.

Remember kernels: We made linear classifiers non-linear by mapping the data implicitly into a fixed very high dimensional feature space.

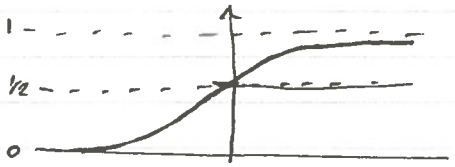
ie. $x \to \phi(x)$ $\quad h(x) = \phi(x)^T w + b$

However, in NN, the function ϕ looks like

$$\phi(x) = \sigma(Ax + b)$$

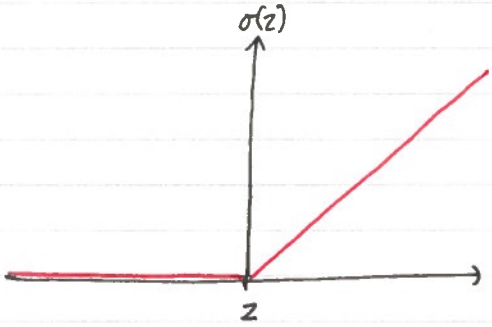
where A is some matrix and b is a constant. The term $Ax + b$ is essentially an affine transformation on the data, such as scaling, skewing, etc. And the function σ is some non-linear function which operates element wise along each dimension.

One example of σ is $\sigma(z) = \frac{1}{1 + e^{-z}}$



Without this function σ, the classifier would be linear.

More recently, this function is becoming $\sigma(z) = \max(z, 0)$



In our classifier, we map $x \Rightarrow \phi(x)$ so $h(x) = \phi(x)^T w + b$.

In kernels, we explicitly set $\phi(x)$ and learn w and b.

In NN, we are <u>also learning $\phi(x)$</u>!

Suppose we have our classifier

$$h(x) = w^T \phi(x) + b \quad \text{where} \quad \phi(x) = \sigma(Ux + c)$$

And we use the squared loss function

$$l = \sum_{i=1}^{\hat{n}} (h(x_i) - y_i)^2$$

And our σ function is $\sigma(z) = \max(0, z)$.

$$h(x) = w^T \max(0, Ux + c) + b$$

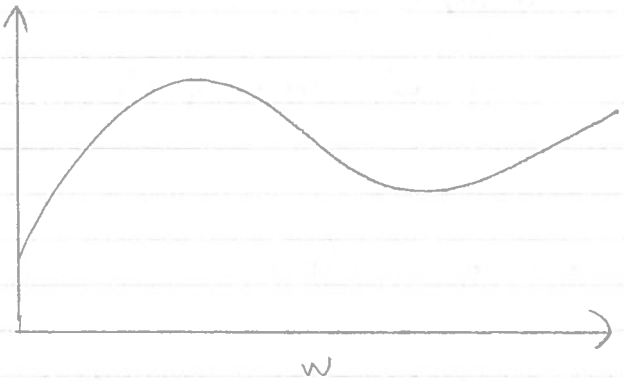
Suppose U is a matrix $\begin{bmatrix} \cdots U_1 \cdots \\ \cdots U_2 \cdots \\ \cdots U_n \cdots \end{bmatrix}$

and c is a column vector with the same number of rows as U $\begin{bmatrix} c_1 \\ c_2 \\ c_n \end{bmatrix}$

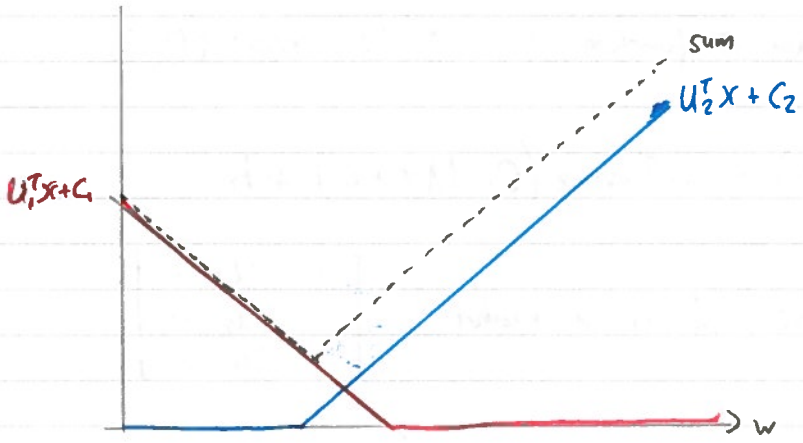
$$h(x) = \sum_j w_j \max(0, U_j^T w + c_j) + b$$ ← bias

changed matrix multiplication to series

Each term in the summation is simply a line.



Suppose we have this function illustrated above. This can be represented by the summation of many straight lines with the restriction that any value below zero is set to 0



Using weights and the restriction $y(x) = \max(0, y(x))$, any 2D function can be represented. (approximately)

APPROXIMATES FUNCTIONS BY PEICE-WISE LINEAR COMPONENTS.

Side-note about $\phi(x)$:

It has been proven analytically that $\phi(x)$ can produce an approximation of any function to arbitrary precision using a combination of linear functions.

$$\phi(x) = \sigma(Ux + c)$$

however, some groups began embedding more ϕ functions inside ϕ functions, such that $x \to \phi(x)$

$$\left.\begin{aligned} \phi(x) &= \sigma(U\phi'(x) + c') \\ \phi'(x) &= \sigma(U\phi''(x) + c'') \\ \phi''(x) &= \sigma(Ux + c'') \end{aligned}\right\} \text{known as layers}$$

This was initially seen as odd because it was proven that any function can be approximated using only one $\phi(x)$.

However, in order to approximate a function using one layer, the ϕ function had to be very wide. This is also computationally expensive.

Including additional layers is able to extract the exponential ~~component~~ complexity of the function with a narrow matrix. This makes it many times quicker to approximate the function.

More layers produces more complex solutions and is known as deep learning.

We need to learn the parameters of ϕ, and we can do this using simple gradient descent.

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

using the squared loss...

$$L(h) = \frac{1}{2} \sum_{i=1}^{n} (h(x_i) - y_i)^2$$

where $h(x) = w^T \phi(x)$.

and where $\phi(x) = \sigma(\overbrace{U\phi(x)}^{a(x)})$

... $\phi'(x) = \sigma(\overbrace{U'\phi''(x)}^{a'(x)})$

... $\phi''(x) = \underbrace{\sigma(U''x)}_{a''(x)}$

} 3 layer neural network

We will start by differentiating the square loss.

$$\frac{\partial L}{\partial w} = \sum_{i=1}^{n} (w^T \phi(x_i) - y_i) \cdot \phi(x)$$

Now differentiate with respect to U

$$\frac{\partial l}{\partial U} = \frac{\partial l}{\partial a} \frac{\partial a}{\partial U}$$

$\frac{\partial a}{\partial U} = \phi(x)$

chain rule

Then with respect to U^l

$$\frac{\partial \ell}{\partial U^l} = \frac{\partial \ell}{\partial a} \frac{\partial a}{\partial a'} \frac{\partial a'}{\partial U^l}$$

$\frac{\partial a'}{\partial U^l} = \phi'(x)$

$\frac{\partial \ell}{\partial a}$: already computed in $\frac{\partial \ell}{\partial U}$

In this case, we only need to compute $\partial a / \partial a'$. And this is also easy as it is just the derivative of the transition function.

This is known as gradient back propogation.

We can use gradient descent by calculating the gradients

$$w \leftarrow w - \alpha \frac{\partial \ell}{\partial w} \qquad c \leftarrow c - \alpha \frac{\partial \ell}{\partial c}$$

$$U \leftarrow U - \alpha \frac{\partial \ell}{\partial U} \qquad b \leftarrow b - \alpha \frac{\partial \ell}{\partial b}$$

The loss function is not convex in neural networks because of the transition function, so there are lots of minimums. Finding the global minimum is not feasible, which means that the starting conditions will matter for finding a minimum.

To help solve this, we use stochastic gradient descent.

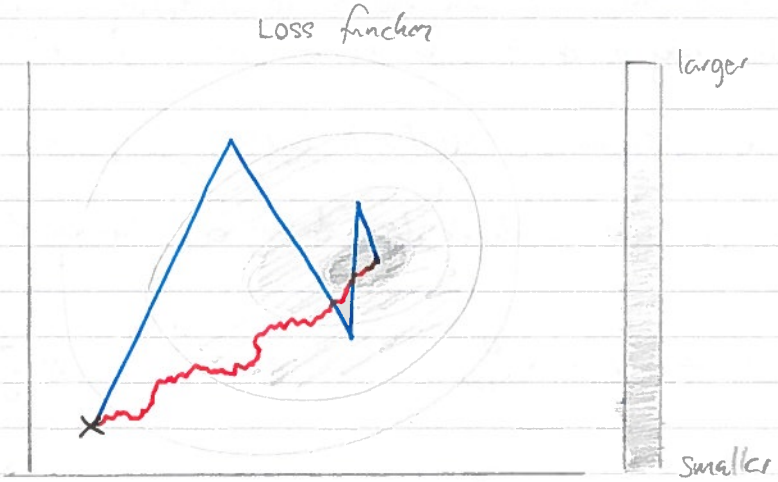
In regular gradient descent $\frac{\partial \ell}{\partial U} = \sum_{i=1}^{n} \frac{\partial \ell(h(x_i), y_i)}{\partial U}$

which is basically the average over all data points.

Stochastic gradient descent approximates that

$$\frac{\partial L}{\partial u} \approx \frac{\partial l(h(x_i), y_i)}{\partial u}$$

so only one data point is used to describe the gradient. So rather than taking a few large steps (GD), many much smaller steps are used where the gradient is changed after every step (SGD).



× starting point

— gradient descent

— stochastic gradient descent.

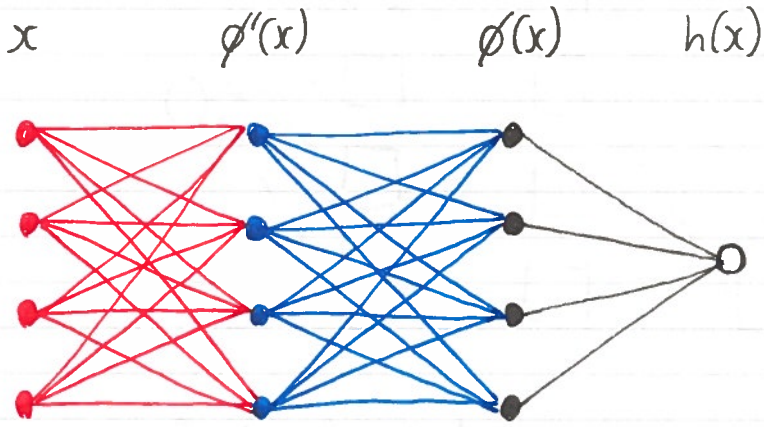
SGD is very noisey. However, this turns out to be a crucial feature to avoid local minimum and saddle points. More precise optimisation methods (especially Newton or approximated newton methods) land in local minimum. SGD is to precise to hit small, narrow (bad) minimum.

Convolutional Neural Networks

Remember how NN can be represented in layers where

$$h(x) = w^T \phi(x)$$
$$\phi(x) = \sigma(U \phi'(x))$$
$$\phi'(x) = \sigma(U'x)$$

An illustration of how x, $\phi'(x)$, $\phi(x)$ and $h(x)$ are mapped can be seen below



This is an example of a fully connected neural network as every point in each layer is connected to every point in the next layer

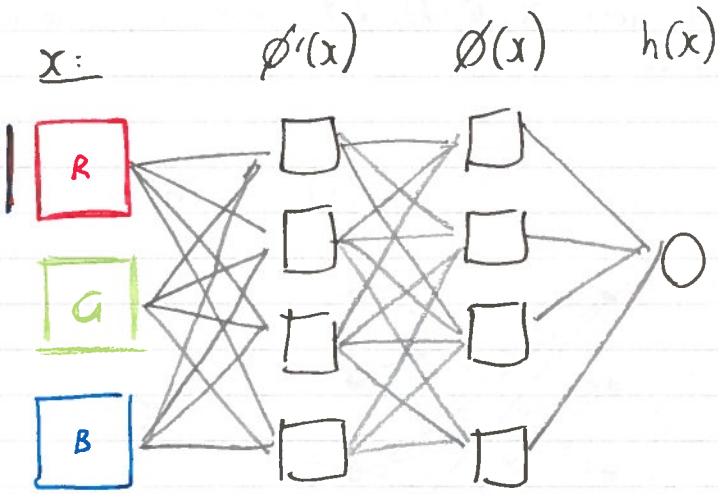
Suppose the input into our convolutional NN (CNN) is an image with heigh h, width w and RGB channels. The number of pixels is therfore $h \times w \times 3$.

Suppose we then re-arrange our image to a 1 by $hw \times 3$ vector.

However by doing this we've lost some information. We have gone from a 2D image to a 1D vector.

If some object of interest in the image is translated by some amount, the 1D vector will look very different despite containing the same information. Information contained in an image is typically translation invariant.

Convolutional neural networks will allow objects to move around without effecting the output. This is performed by respecting the fact the input data is an image, where x is no longer a vector, but 3 different images (one for each colour channel)



Each layer maintains that the format of x, $\phi(x)$, $\phi'(x)$ is an image, and only at the end does it become some value.

The convolution operation allows this to happen. This means each image within a layer is acquired using some image convolution on the previous layer.

Convolutions can be viewed as pattern detectors, that will find edges, circles, eyes, etc.