COMP9417 Notes

Alperen Onur

May 2025

1 Regression

1.1 Supervised & Unsupervised Learning

The two types of machine learning algorithms fall under supervised or unsupervised learning. Supervised learning categorises algorithms with given labels and unsupervised algorithms categorise algorithms with no given labels.

1.2 Linear Regression

Regression predicts numerical values whereas **Classification** predicts discrete values. **Linear Regression** is a particular type of regression which models the relationship between an input variable and an output variable using a straight line,

$$\hat{y} = bx + c$$

where, $\hat{y}$ are our predicted labels based on our input features x. The goal in our case is to estimate the slope b and intercept c from the data. We make the following assumptions for linear regression,

- Linearity: The relationship between x and the mean of y is linear.
- Homoscedesticity: The variance of residual is the same for any value of x.
- Independence: Observations are independent of each other.
- Normality of residuals: For any fixed value of x, y is normally distributed.

1.3 Linear Regression Formulation

In linear models, the outcome is a linear combination of attributes,

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = h(x)$$

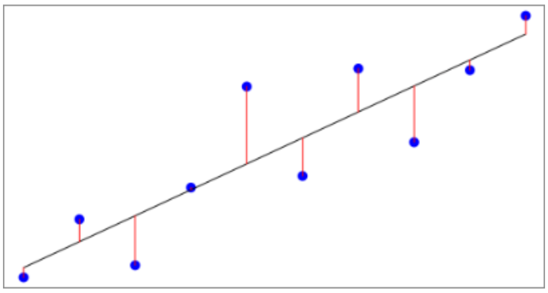
where θ_i is the weight from the observed training data for attribute i. The predicted value of the first training instance x_i is,

$$\hat{y}_1 = \theta_0 + \theta_1 x_{11} + \theta_2 x_{12} + \cdots + \theta_n x_{1n} = \textstyle\sum_{i=0}^{n} \theta_i x_{1i} = x_1^T \theta = h(x_1).$$

Typically, x_0 is set to 1 and we define $x = [x_0, x_1, ..., x_n]^T$.

1.4 Minimising Error

We can fit an infinite amount of lines to a dataset depending on what we define as the best fit criteria. The most popular estimation model is ”Least Square”, also known as ”Ordinary Least Squares” regression. This model attempts the minimise the difference between the predicted and actual values; minimising the error.



The goal is the minimise the error over all input samples. The total error is defined as the sum of squared errors and searching for $n + 1$ parameters to minimise that. The sum of squared error for m samples is denoted as,

$$J(\theta) = \textstyle\sum_{j=1}^{m} \left(y_j - \sum_{i=0}^{n} \theta_i x_{ji}\right)^2 = \sum_{j=1}^{m} (y_j - x_j^T \theta)^2 = (\mathrm{y} - X\theta)^T (\mathrm{y} - X\theta)$$

where $J(\theta)$ is the loss function and X, y are

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1n} \\ 1 & x_{21} & x_{22} & \dots & x_{2n} \\ & & \vdots & & \\ 1 & x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix} \qquad \mathrm{y} = \begin{bmatrix} y_1 \\ y_2 \\ \\ y_m \end{bmatrix}$$

The loss function can also be generalised as averaging the squared error and minimising it which results in the same θ,

$$J(\theta) = \frac{1}{m} \sum_{j=1}^{m} (y_j - x_j^T)^2.$$

This is typically referred to as the **mean squared error (MSE)**.

1.5 Least Squares Regression

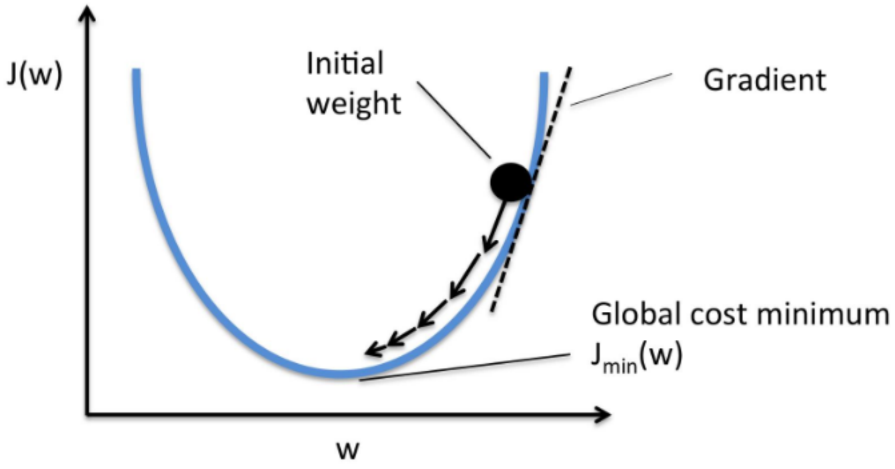
We need a method to estimate the parameters of the lost function as to minimise its overall cost. Less error means more accurate predictions. Computing $J(\theta)$ for different values of θ results in a convex function which has a single global minima. An algorithm to converge to this minima is **Gradient Descent**.

1.6 Gradient Descent

Gradient descent starts with some initial θ, and repeatedly performs an update,

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \alpha \frac{\partial}{\partial \theta_i} J(\theta_i^{(t)})$$

where α is the learning rate. In each iteration of the algorithm, it takes a "step" the size of α in the direction with the steepest increase in $J(\theta)$. To implement this algorithm, we'll need the partial derivative of the MSE.



For one sample of m samples, the cost function is,

$$J(\theta) = (y_j - h_\theta(x_j))^2 = (y_j - \sum_{i=1}^{n} x_{ji}\theta_i)^2 \tag{1}$$

$$h_\theta(x_j) = \sum_{i=1}^{n} x_{ji}\theta_i = x_j^T\theta \tag{2}$$

Now taking the derivative we get,

$$\frac{\partial}{\partial\theta_i} J(\theta) = -2(y_j - h_\theta(x_j))x_{ji}. \tag{3}$$

So for a single training sample, the update rule is,

$$\theta_i^{(t+1)} = \theta_i^{(t)} + 2\alpha(y_j - h_\theta(x_j))x_{ji}. \tag{4}$$

This update rule for squared distance is called **Least Mean Squares (LMS)**. For multiple samples, there are a couple methods used for updating the LMS rule: **Batch Gradient Descent** and **Stochastic Gradient Descent**.

1.7 Batch Gradient Descent

$$\theta_i^{(t+1)} = \theta_i^{(t)} + \alpha\tfrac{2}{m}\textstyle\sum_{j=1}^{m}(y_j - h_{\theta_i^{(t)}}(x_j))x_{ji}.$$

For every i, replace the gradient with the sum of gradient for all samples until convergence (when θ is stabilised).

1.8 Stochastic Gradient Descent

For $j = 1$ to m:
[1] $\theta_i^{(t+1)} := \theta_i^{(t)} + 2\alpha(y_j - h_\theta(x_j))x_{ji}$ (for every i)
Repeat until algorithm converges. θ gets updated at each sample separately. This is much less costly than batch gradient descent but it may also never converge to a minimum.

1.9 Minimising Square Error with Normal Equations

We can also find the minimum of $J(\theta)$ by explicitly taking its derivatives and setting them to zero. This is the closed form solution,

$$\begin{aligned}
\frac{\partial}{\partial\theta} J(\theta) &= 0 \qquad (5)\\
J(\theta) &= (\mathbb{y} - X\theta)^T(\mathbb{y} - X\theta) \qquad (6)\\
\frac{\partial}{\partial\theta} J(\theta) &= -2X^T(\mathbb{y} - X\theta) = 0 \qquad (7)\\
X^T(\mathbb{y} - X\theta) &= 0 \qquad (8)\\
\theta &= (X^TX)^{-1}X^T\mathbb{y} \qquad (9)
\end{aligned}$$

1.10 Minimising Square Error with Probabilistic Interpretation

We can write the relationship between an input variable x and output variable y as,

$$y_j = x_j^T\theta + \epsilon_j$$

where ϵ_j is an error term (random noise). If we assume all ϵ_j are independent and identically distributed according to the Gaussian distribution $\epsilon_j \sim N(0, \sigma^2)$ then,

$$p(\epsilon_j) = \tfrac{1}{\sqrt{2\pi\sigma^2}} \exp(-\tfrac{\epsilon_j^2}{2\sigma^2}).$$

This implies that, $P(\epsilon_j) = P(y_j|x_j\,;\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y_j - x_j^T\theta)^2}{2\sigma^2})$. So we want to estimate θ such that we maximise the probability of output y given input x over all m training samples. Mathematically this is,

$$\mathcal{L}(\theta) = P(\mathbb{y}|X\,;\theta)$$

where $\mathcal{L}(\theta)$ is the likelihood function. Since we assumed independence over ϵ_j, then each of our training samples are independent from each other. Then it follows that,

$$\begin{aligned}
\mathcal{L}(\theta) &= \prod_{j=1}^{m} P(y_j|x_j\,;\theta) \qquad (10)\\
&= \prod_{j=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} \exp -\frac{(y_j - x_j^T\theta)^2}{2\sigma^2}. \qquad (11)
\end{aligned}$$

This is called the **maximum likelihood**. If we want to find a θ that maximises $\mathcal{L}(\theta)$, we can also maximise any strict increasing function of $\mathcal{L}(\theta)$. If we choose to maximise the **log likelihood** $\ell(\theta)$,

$$\ell(\theta) = \log \mathcal{L}(\theta) = \prod_{j=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} \exp - \frac{(y_j - x_j^T\theta)^2}{2\sigma^2} \tag{12}$$

$$= m \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{1}{\sigma^2}\frac{1}{2}\sum_{j=1}^{m}(y_j - x_j^T\theta)^2. \tag{13}$$

So maximising $\ell(\theta)$ is equal to minimising $\sum_{j=1}^{m}(y_j - x_j^T\theta)^2$ which is the MSE.

2 Statistical Techniques for Data Analysis

2.1 Sampling

Sampling is a way to draw conclusions about a population without having to measure the entire population. For groups that are fairly homogeneous, we do not need to collect alot of data. For populations with alot of irregularities, we need to either take measurements from the entire group or find a some other way to get a good idea of the groups trends without having to do so. Sampling gives us a way to do this!

We want from our sampling method to have as little bias that we can account for and if the chance of obtaining an unrepresentative sample is high then we can choose to not draw conclusions. The chance of an unrepresentative sample decreases with the size of the sample.

2.2 Estimation

Estimation refers to the process by which one makes inferences about a population based on information obtained from a sample. A ”good” estimate means that the estimator is correct on average. If the expected value of the estimator equals the true population parameter then it is said to be an unbiased estimator. The following are a few examples of different estimators,

- Sample mean: $\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$
- Sample median: The middle value when the sample data is ordered. Less affected by outliers; biased.
- Sample Variance $s^2 = \frac{1}{N-1}\sum_i (x_i - m)^2$
- Sample Standard Deviation: $s = \sqrt{s^2}$ Indicates the average distance of each sample from the mean; biased.

- Sample Range: Provides a simple measure of data spread by taking the difference between the maximum and minimum values; biased.

2.3 Covariance

Covariance is the measure of the relationship between two random variables,

$$cov(x,y) = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{N-1} = \frac{(\sum_i x_i y_i) - N\overline{xy}}{N-1}$$

2.4 Correlation

Correlation is a measure to show how strongly a pair of random variables are. The Pearson correlation between x and y is,

$$r = \frac{cov(x,y)}{\sqrt{var(x)}\sqrt{var(y)}}.$$

This only captures linear relationships between two variables. The Pearson correlation can range between -1 and 1. A value close to 1 shows high values of x are associated with high values of y and low values of x are associated with low values of y. Generally this means that the scatter is low. A value near 0 indicated there is no association; large scatter. A value close to -1 suggests a strong inverse association between x and y. Correlation is a quick way of checking whether there is some linear association between two variables x and y where the sign tells you the direction of that association.

3 Univariate Linear Regression

In order to find the parameters we take the partial derivatives, set them to 0 and solve for θ_0. This will lead to,

$$\theta_1 = \frac{cov(h,w)}{var(h)} \tag{14}$$

$$\theta_0 = \overline{w} - \theta_1 \overline{h}. \tag{15}$$

3.1 Linear Regression Intuitions

Adding a constant to all x values affects only the intercept but not the regression coefficient. We can then zero-centre the x values by subtracting the mean of x in which case the intercept is equal to the mean of y. Similarly we can subtract the mean of y

from all y values to achieve a zero intercept, without changing the regression problem in an essential way. Another important point is that the sum of the residuals of the MSE makes linear regression susceptible to outliers far from the regression line.

3.2 Multiple Regression

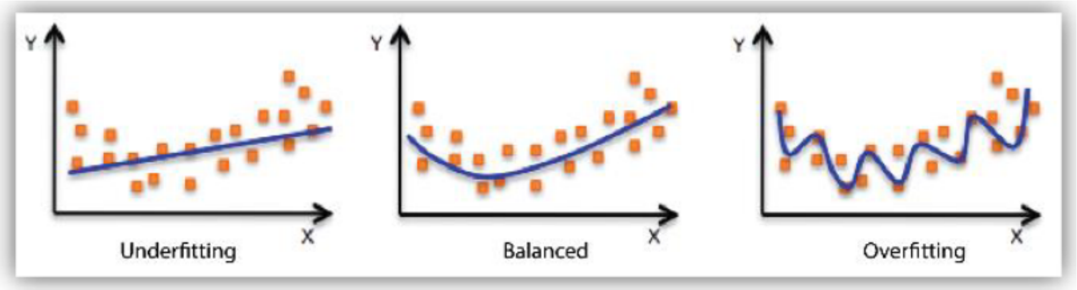
Often we need need to model the relationship of y to several other variables. Similar to univariate regression we can model this as,

$$\hat{w} = \theta_0 + \theta_1 h + \theta_2 b$$

and minimize the sum of the square residuals like so,

$$\textstyle\sum_{j=1}^{m}(w_j - (\theta_0 + \theta_1 h_j + \theta_2 b_j))^2.$$

These types of regression models tend to produce curves rather than lines. So we can predict an output by adding different amounts of sample terms with each term increasing the overall degree of the models polynomial factor.



3.3 Regularisation

To control for overfitting or underfitting we introduce the concept of **regularisation**. Regularisation applies addition constraints the the weight vectors of a model. The regularised form for linear regression could be,

$$J(\theta) = \textstyle\sum_{j=1}^{m}(y_j - h_\theta(x_j))^2 + \lambda \sum_{i=1}^{n} \theta_i^2.$$

The multiple least square regression problem is an optimisation problem and can be written as,

$$\theta^* = \arg\min \theta \ (y - X\theta)^T (y - X\theta)$$

with the regularised version of this being,

$$\theta^* = \arg\min \theta \; (y - X\theta)^T (y - X\theta) + \lambda ||\theta||^2$$

where $||\theta||^2 = \sum_i \theta_i^2$ is the square norm of the vector θ; the dot product $\theta^T \theta$.

3.4 Ridge Regression

The regularised problem has a closed-form solution,

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

where I denotes the identity matrix. This adds λ amount of regularisation to the diagonal of $X^T X$. This is known as Ride Regression.

3.5 LASSO Regression

LASSO regression replaces the ride regression term $\sum_i \theta^2$ with the sum of absolute weights $\sum_i |\theta|$. LASSO regression favours more sparse solutions.

4 Train, Validation & Test Data

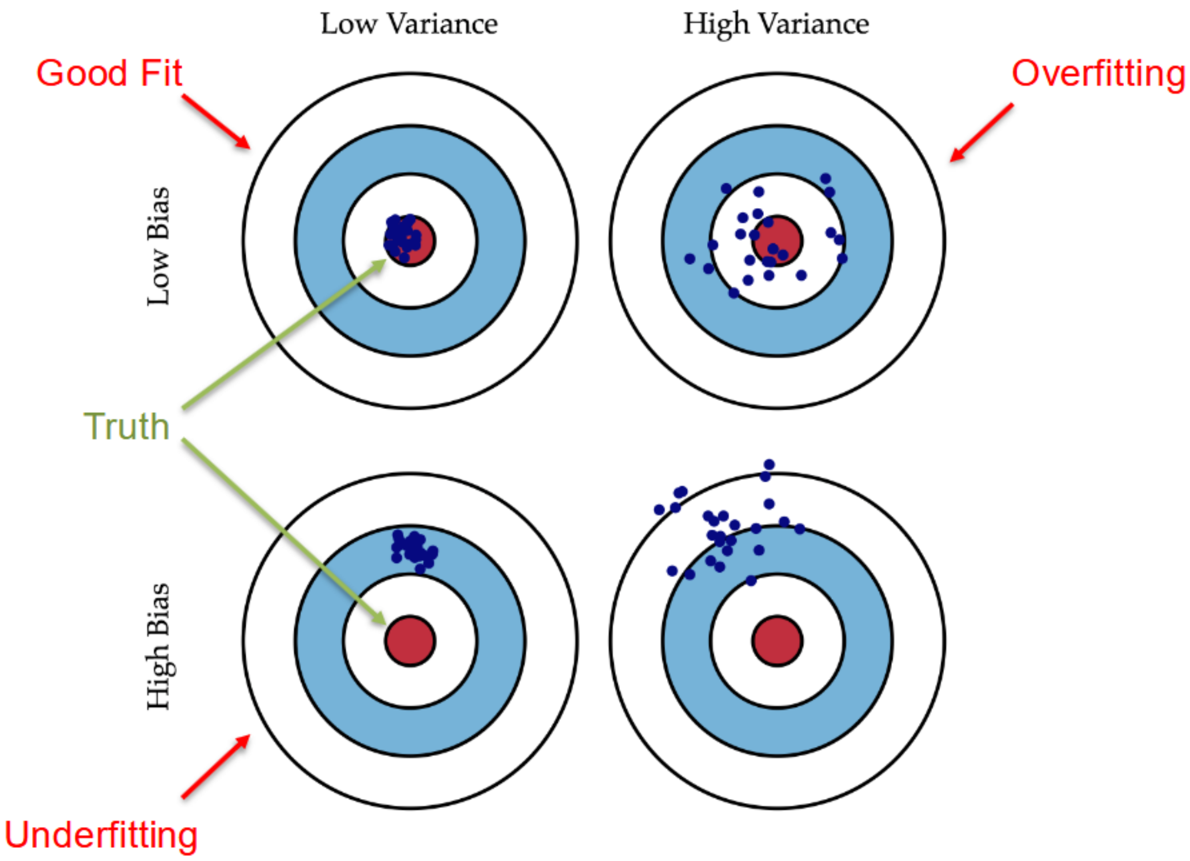
Train data is the data we use to learn our model and its parameters. Validation data is the unseen data by our model used to provide an unbiased evaluation of a model fit on the training dataset while tuning the models hyper-parameters. The test data is the data we use to test the model and shows how well our model generalises.

4.1 Model Selection

There are 3 ways to reduce complexity of a model,

- Subset-selection: search over a subset lattice such that each subset results in a new model and select one of those models.
- Shrinkage: Use regularisation to set coefficient of models to 0.
- Dimensionality reduction: Project points to a lower dimensional space.

4.2 Bias-Variance Tradeoff



Bias is the difference between the average prediction of our model and the correct value which we are trying to predict. Models with high bias pay very little attention to the training data and oversimplifies the model. **Variance** is the variability of the model for a given data point or a value which tells us the spread of data. Models with high variance pays a lot of attention to the training data but does not generalise on the data which has not been seen before. Underfitting means high bias and low variance. Overfitting means our model has captured a lot of noise along the underlying pattern in the data.

4.3 Bias Variance Decomposition

When we assume $y = f + \epsilon$ and we estimate f with $\hat{f}$, then the expectation of error is,

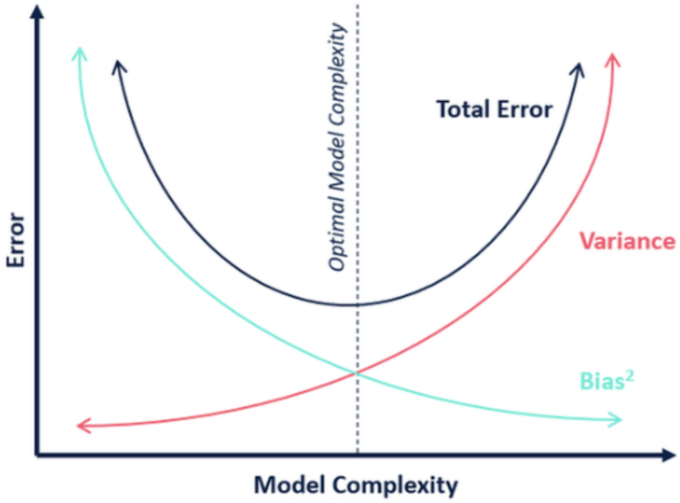
$$E[(y - f)^2] = (f - E[\hat{f}])^2 + \text{Var}(\hat{f}) + \text{Var}(\epsilon)$$

and so the MSE can be written as,

$$\text{MSE} = \text{Bias}^2 + \text{Variance} + \text{irreducible error}.$$

Irreducible error is the inherent uncertainty associated with a natural variability in a system. It can not be reduced since it is due to unknown factors. Reducible error is error that can and should be minimised further by adjustments to the model.

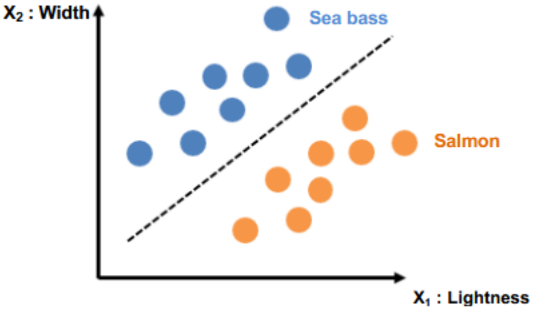
If our model is simple and has few parameters, then it may have high bias and low variance. On the other hand, if our model has a large number of parameters then it's going to have high variance and low bias. So we need to find a good balance without overfitting or underfitting to the data.



5 Classification

Classification is the concept of learning a classifier which is a function mapping from an input data point to one of a set of discrete outputs.

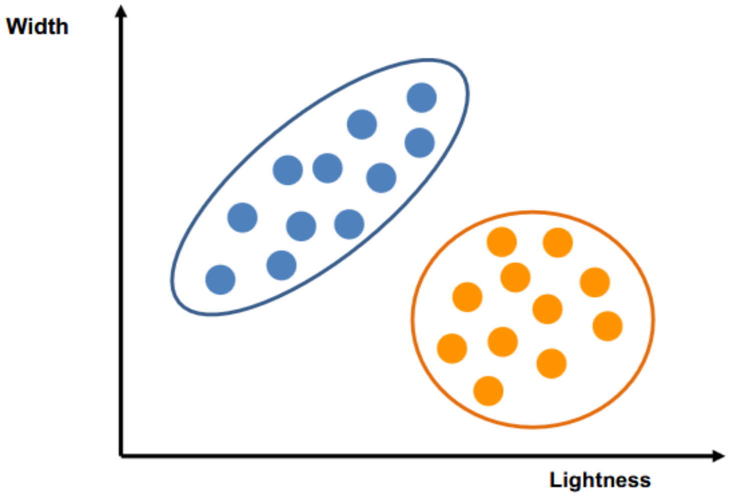
5.1 Linear Classifier



If we can find a line two separate two classes then this is called a linear classifier.

5.2 Generative algorithms

If we can instead of finding a discriminative line, build some models for each of the classes and make classifications predictions based on the test example to see which model it is most similar too. Generative algorithms learn $p(x|y)$ and $p(y)$ which is the class prior whereas discriminative algorithms only learn $p(y|x)$.



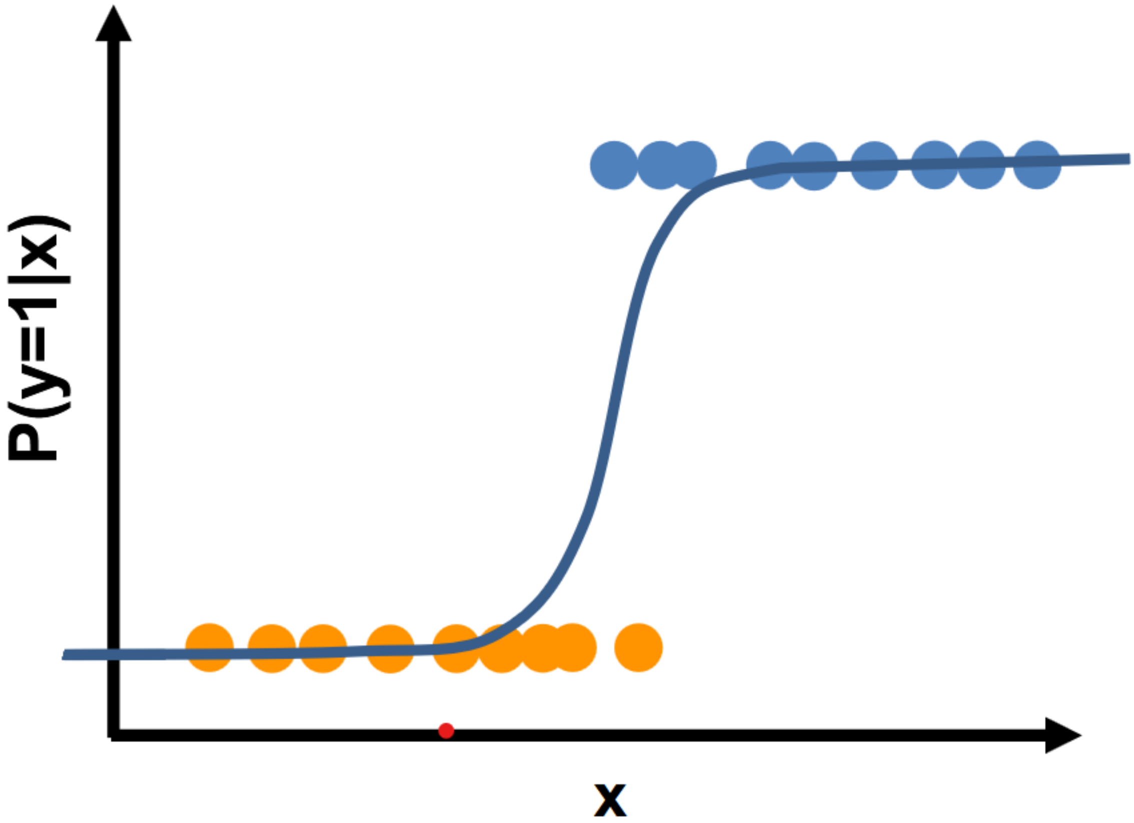
To predict the output sample x, we need to estimate $p(y|x)$,

$$p(y=0|x) = \frac{p(x|y=0)p(y=0)}{p(x)} \tag{16}$$

$$p(y=1|x) = \frac{p(x|y=1)p(y=1)}{p(x)} \tag{17}$$

5.3 Logistic Regression

In a binary classification, we can transform the y values into probability values in rage $[0, 1]$. We can model this using a sigmoid curve,



$$p(y=1|x) = \frac{1}{1+\exp(-f(x))} \tag{18}$$

$$\Rightarrow \log\left(\frac{P(y=1|x)}{1-P(y=1|x)}\right) \tag{19}$$

Now $f(x)$ can have a value between $-\infty$ and $+\infty$ so we can estimate $f(x)$ with a line,

$$\hat{f}(x) = x^T\beta \Rightarrow \log(\tfrac{P(y=1|x)}{1-P(y=1|x)}) = x^T\beta$$

We can generalise this to a linear solution with,

$$\hat{P}(y=1|x) = \tfrac{1}{1+\exp(-x^T\beta)}.$$

If $P(y=1|x) \geq 0.5$ then predict as class 1 and otherwise predict as class 0. We define the cost function as,

$$\text{cost}(h_\beta(x), y) = \begin{cases} -\log(h_\beta(x)) \text{ if } y=1 \\ -\log(1-h_\beta(x)) \text{ if } y=0 \end{cases}$$

where $\hat{P}(y=1|x) = h_\beta(x)$. This piecewise function can be combined into a single cost function as,

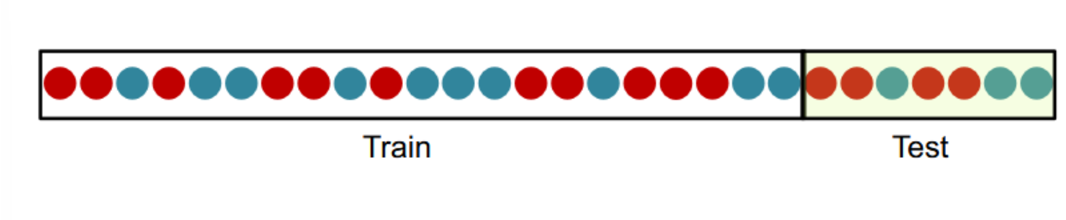
$$J(\beta) = -\tfrac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log(h_\beta(x^{(i)})) + (1+y^{(i)})\log(1-h)\beta(x^{(i)}))\right).$$

To find the parameters that minimise $J(\beta)$ the Gradient Descent algorithm can be used.

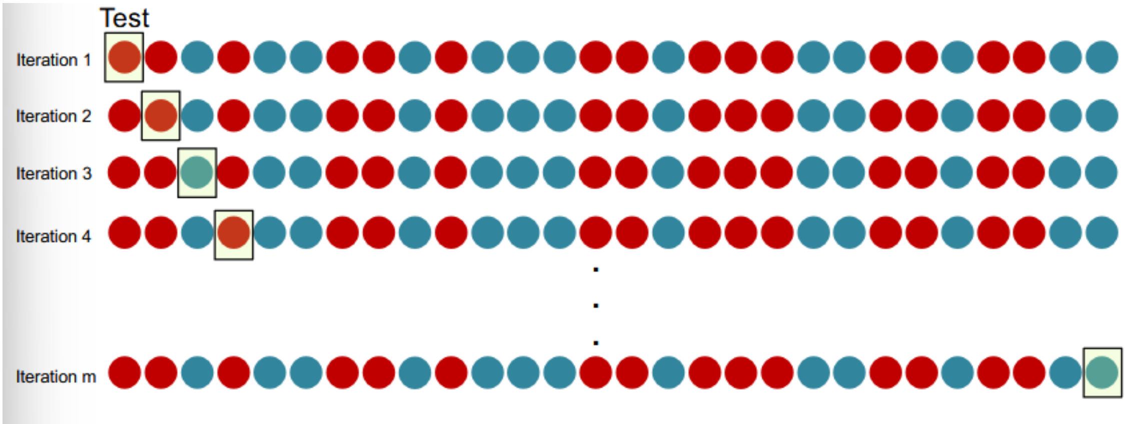
6 Generalisation

We want to generalise a sample set of data in the training set for our models. To examine how "general" our models are we can use **Cross-Validation** to asses the results of our model on an independent dataset.

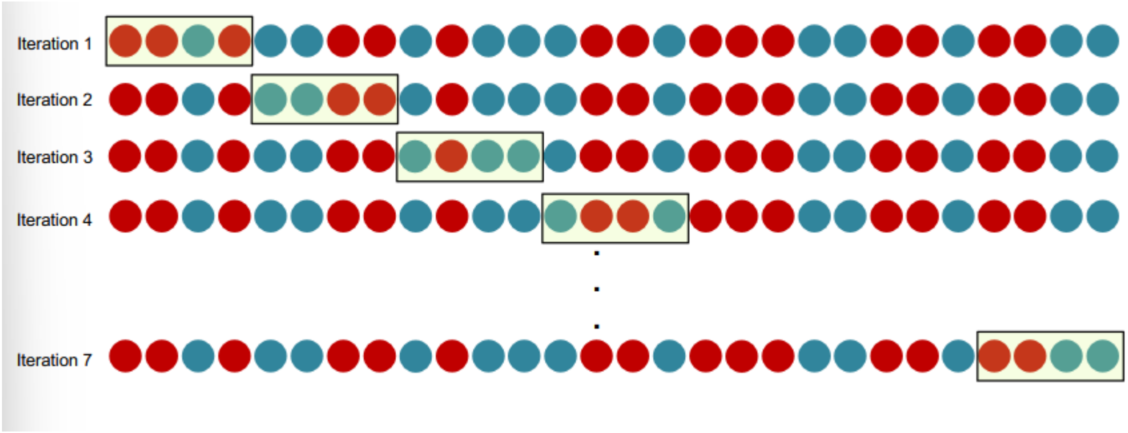
6.1 Holdout Method



6.2 Leave One Out Cross Validation

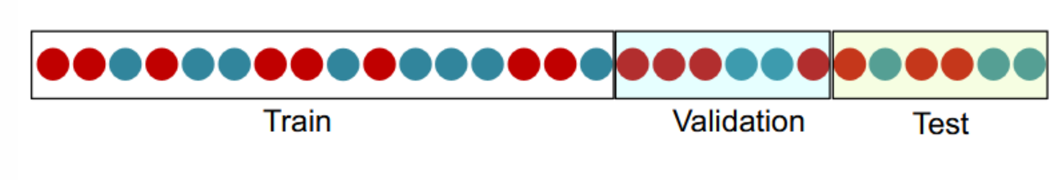


6.3 K-Fold Cross Validation



6.4 Validation set

If we want to estimate certain parameters during learning we'll instead use a validation set; separate from training and test data.



Estimating parameters is also called **hyperparameter tuning**.

6.5 Evaluation of Error

If we have a binary classification task then we have two classes, $y \in \{0, 1\}$, where we call the class $y = 1$ the positive class and $y = 0$ the negative class. Some evaluation metrics for this classification are:

- True Positive: predicted class of 1 is actually 1.
- True Negative: predicted class of 0 is actually 0.
- False Positive: predicted class of 1 is actually 0.
- False Negative: predicted class of 0 is actually 1.

6.6 Classification Accuracy

Classification accuracy on a sample of labelled pairs $(x, c(x))$ given a learned classification model that predicts, for each instance x, a class $\hat{c}(x)$,

$$\text{acc} = \tfrac{1}{|\text{Test}|} \textstyle\sum_{x\in\text{Test}} I[\hat{c}(x) = c(x)]$$

where Test is a test set and $I[]$ is the indicator function which is 1 iff its argument evaluates to true and 0 otherwise. The classification error is $1-\text{acc}$.

6.7 Precision

The number of relevant objects classified correctly divided by the total number of relevant objects classified.

$$\text{Precision} = \tfrac{TP}{TP+FP}$$

6.8 Recall

The number of relevant objects classified correctly divided by total number of relevant objects classified correctly divided by the total number of correct objects.

$$\text{Recall} = \tfrac{TP}{TP+FN}$$

6.9 F1 Score

A measure of accuracy which is the harmonic mean of precision and recall defined as,

$$F_1 = 2\tfrac{\text{precision}\times\text{recall}}{\text{precision}+\text{recall}}$$

6.10 Missing values

Sometimes values will be missing in our data, in this case we have a few techniques to "fix-up" the data:

- Deleting samples with missing values

- Replacing the missing value with some statistics from the data (mean, median, ...)
- Assigning a unique category
- Predicting the missing values
- Using algorithms which support missing values

7 Nearest Neighbour

Nearest Neighbour is a classification algorithm that predicts whatever is the output value of the nearest data point to some query. To find the nearest data point, we need to define some metric of distance. There are many distance metrics including Minkowski distance, Euclidean distance, Manhattan distance, Chebyshev distance and more.

Exemplars are centroids that refer to a point to summarise or represent a large group of data. Sometimes **exemplars** are used in distance based learning because they can be selected based on their proximity to other points with the aim of minimising the distance between the exemplar point and each point in its cluster. Exemplars can be the arithmetic mean or geometric mean of the data.

7.1 Nearest Centroid Classifier

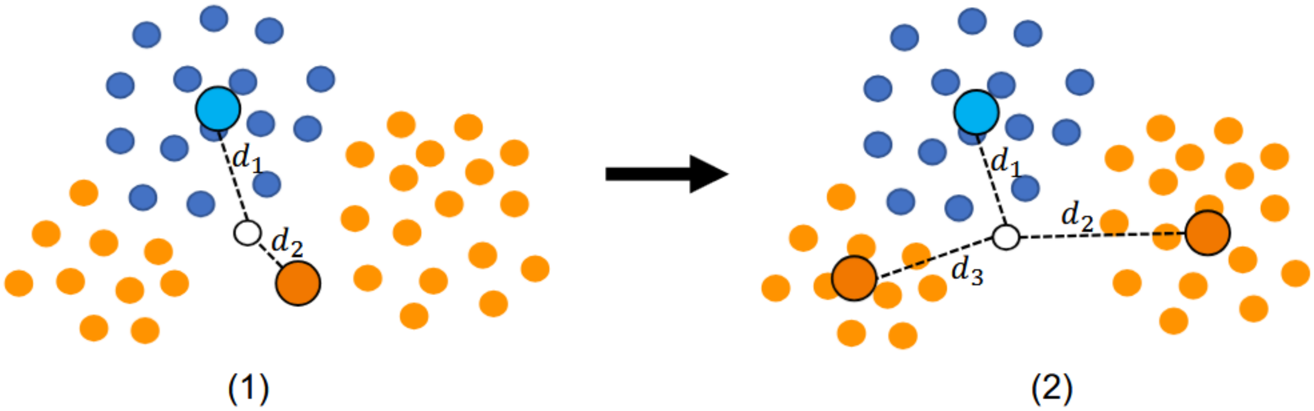
This is a classifier based on minimum distance principle where the exemplars are just the centroid means. The training sample pairs $\{(x_1, y_1), (x_2, y_2), \cdots, (x_m, y_m)\}$ where x_i is the feature vector for the sample i and y_i is the class label define class centroids as,

$$\mu_k = \tfrac{1}{|C_k|} \textstyle\sum_{j \in C_k} x_j.$$

The test set is a new unknown object with feature vector x is classified as class i if it much closer to the mean of class k than any other class mean vector.

If a class has more than one mode,

- If there is only one centroid per class, then it'll perform poorly
- If different modes can be defined, we can define one centroid per each mode which helps the classifier



7.2 Nearest Neighbour Classification

Refers to training instances that are searched for the instance that most closely resembles new or query instances. Store all training examples $< x_j, f(x_j) >$. Nearest neighbour for a given query instance x_q, first locate nearest training examples x_n, then estimate $\hat{f}(x_q) \leftarrow f(x_n)$. k**-Nearest Neighbour** instead will take a vote among its k nearest neighbours or the mean of f values of k nearest neighbours,

$$\hat{f}(x_q) \leftarrow \frac{\sum_{j=1}^{k} f(x_j)}{k}$$

7.3 K Nearest Neighbours (KNN) Algorithm

During training, for each training example $< x_j, f(x_j) >$, add the example to the list of training examples. During classification, given a query instance x_q to be classified, let $x_1, \cdots, x_k$ be the k instances from the training examples that are nearest to x_q be the distance function. We then return,

$$\hat{f}(x_q) \leftarrow \text{arg max}_{v \in V} \sum_{j=1}^{k} \delta(v, f(x_j))$$

where $\delta(a, b) = 1$ if $a = b$ and 0 otherwise. The distance function defines what is learned. Instance x_j can be described as a feature vector $x_j = (x_{j1}, \cdots, x_{jd})^T$ where x_{jr} denotes the value of the rth attribute/feature of x_j. The most commonly used distance function is Euclidean distance where the distance between x_i and x_j is defined to be,

$$\text{Dis}(x_{i,j}) = \sqrt{\sum_{r=1}^{d} (x_{ir} - x_{jr})^2}$$

7.4 Min-Max Normalisation

$$x'_{jr} = \tfrac{x_{jr} - \min(x_j r)}{\max(x_{jr}) - \min(x_{jr})}$$

where x_{jr} is the actual value of the attribute/feature r and x'_{jr} is the normalised value.

7.5 Z-score

$$x'_{jr} = \tfrac{x_{jr} - \mu_r}{\sigma_r}$$

7.6 More Nearest Neighbour

1NN perfectly separates training data so has low bias and high variance. By increasing k, we increase the bias and decrease the variance. If $k = m$ predictions will just be based on the majority class in the data set and this may overfit to the data. If we want to weigh nearer neighbours more heavily, we can use a distance function to construct a weight w_i and replace the classification algorithm by,

$$\hat{f}(x_q) \leftarrow \text{arg max}_{v \in V} \textstyle\sum_{i=1}^{k} w_i \delta(v, f(x_j))$$

where,

$$w_i = \tfrac{1}{\text{Dis}(x_q, x_i)^2}$$

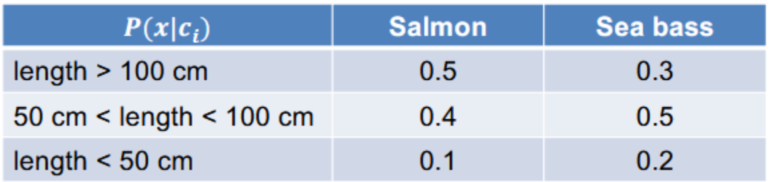
8 Classification

Inductive Bias refers to a set if assumptions that an ML model makes to generalise from limited training data. Different from bias. A strong inductive bias leads to high bias but lower variance and a weaker inductive bias reduces bias but increases variance. Frameworks in machine learning look for ways to represent the inductive bias.

8.1 Bayesian Machine Learning

Imagine a scenario where we need to classify different fish as either Salmon or Sea bass where C_i is the class. Probability based on past experience is called a **prior**. Say the prior probabilities were $P(c_{Salmon}) = 0.3$ and $P(c_{Seabass}) = 0.7$. Then only based on the prior, sea bass would always be picked.

Say we have more information about the fish now: the length. Then we can update our predictions based on the length of the fish. These are called class conditionals.



$P(x\|c_i)$	Salmon	Sea bass
length > 100 cm	0.5	0.3
50 cm < length < 100 cm	0.4	0.5
length < 50 cm	0.1	0.2

Now we have $P(c_i)$ and $P(x|c_i)$ but what we want is $P(c_i|x)$. We calculate $P(c_i|x)$ using **Bayes Theorem**.

8.2 Bayes Theorem

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

where,

- $P(h)$ is the prior probability of hypothesis h,
- $P(D)$ is the prior probability of training data D,
- $P(h|D)$ is the probability of h given D and,
- $P(D|h)$ is the probability of D given h.

8.3 Decision Rule from Posteriors

If the output belongs to a set of k classes, $y \in \{C_1, C_2, \cdots, C_k\}$ where $1 \leq i \leq k$. Then in the Bayesian framework,

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

where

- $P(y = C_i|x)$ is the **posterior probability**,
- $P(x|C_i)$ is the class conditional,

- $P(C_i)$ is the prior and,
- $P(x)$ is the marginal ($P(c) = \sum_i p(x|C_i)p(C_i)$).

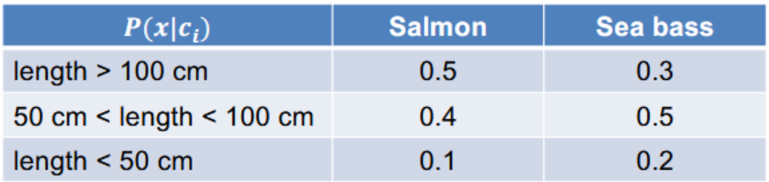
The maximum posteriori hypothesis h_{MAP} gives the most most probable hypothesis on the training data,

$$\begin{aligned} h_{MAP} &= \text{arg max}_{h \in H} P(h|D) && (20) \\ &= \text{arg max}_{h \in H} \frac{P(D|h)P(h)}{P(D)} && (21) \\ &= \text{arg max}_{h \in H} P(D|h)P(h) && (22) \end{aligned}$$

If $P(h_i) = P(h_j)$ then we can further simplify the **Maximum Likelihood Hypothesis**,

$$h_{ML} = \text{arg max}_{h_i \in H} P(D|h_i)$$

Coming back to the example about fishes and their lengths,



$P(x\|c_i)$	Salmon	Sea bass
length > 100 cm	0.5	0.3
50 cm < length < 100 cm	0.4	0.5
length < 50 cm	0.1	0.2

$$\begin{aligned} P(c = salmon|x = 70cm) &\propto P(70cm|salmon) * P(salmon) = 0.12 && (23) \\ P(c = seabass|x = 70cm) &\propto P(70cm|seabass) * P(seabass) = 0.35 && (24) \end{aligned}$$

8.4 Posterior Probability

For a hypothesis h, the posterior probability is,

$$P(h|D) = \frac{P(D|h)P(h)}{\sum_{h_i \in H} P(D|h_i)P(h_i)}$$

where the denominator ensures all posterior probabilities sum to 1.

8.5 Predictions with Posterior Probabilities

If there are two competing hypothesis h_1 and h_2 with posterior probabilities $P(h_1|D)$ and $P(h_2|D)$ respectively, then the potential decision is made based on the higher posterior probability. An alternative is the make a decision based on the loss that occurs when decision h is made using a loss function $L(h)$. If the cost of misclassification is not the same for different classes, then we need to minimise the expected loss,

$$E[L(\alpha_i)] = R(a_i|x) = \textstyle\sum_{h \in H} \lambda(a_i|h)P(h|x)$$

where, the loss associated to action a_i is $\lambda(a_i|h)$.

8.6 Learning A Real Valued Function

To compute $P(D|h)$ and $P(h)$ we can assume a parametric model and estimate the parameters using the data. Consider any real-valued function f and training samples $< x_i, y_i >$ where y_i is noisy training value. Then we have,

$$y_i = f(x_i) + \epsilon_i$$

where ϵ_i is random noise drawn independently for each x_I according to some Normal(Gaussian) distribution with variance σ^2 and mean zero. Then the maximum likelihood hypothesis, h_{ML} is the one that minimises the sum of squared errors,

$$\begin{aligned} h_{ML} = \text{arg max}_{h \in H} P(D|h) &= \text{arg max}_{h \in H} \prod_{i=1}^{m} P(y_i|h) && (25) \\ &= \text{arg max}_{h \in H} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}(\frac{y_i - h(x_i)}{\sigma})^2\right) && (26) \end{aligned}$$

taking log to give a simpler expression,

$$\begin{aligned} &= \text{arg max}_{h \in H} \prod_{i=1}^{m} \log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{1}{2}(\frac{y_i - h(x)}{\sigma})^2 && (27) \\ &= \text{arg max}_{h \in H} \prod_{i=1}^{m} -\frac{1}{2}(\frac{y_i - h(x_i)}{\sigma})^2 && (28) \\ &= \text{arg max}_{h \in H} \prod_{i=1}^{m} -(y_i - h(x_i))^2 && (29) \\ &= \text{arg min}_{h \in H} \sum_{i=1}^{m} (y_i - h(x_i))^2 && (30) \end{aligned}$$

8.7 Bayes Optimal Classifier

The Bayes optimal classifier is a probabilistic model that makes the most probable prediction for a new sample based on the training data. In MAP framework, we seek the most probable hypothesis among a space of hypothesis. In Bayesian optimal classification, the most probable classification of the new instance is obtained by combine the predictions of all hypotheses, weighted by their posterior probabilities,

$$P(v_j|D) = \textstyle\sum_{h_i \in H} P(v_j|h_i)P(h_i|D).$$

The optimal classification oof the new instance is the value v_j, for which $P(v_j|D)$ is maximum.

8.8 Bayes Error

Define the probability of error for classifying some instance x by, $P(error|x) = P(class_1|x)$ if we predict $class_2$ and $P(error|x) = P(class_2|x)$ if we predict $class_1$. This gives,

$$\sum P(error) = \sum_x P(error|x)P(x).$$

If $\hat{P}(class_1|x) > \hat{P}(class_2|x)$ predict $class_1$ and otherwise predict $class_2$.

8.9 Naive Bayes Classifier

Assume target function $f : X \rightarrow V$, where each instance x described attributes $< x_1, x_2, \cdots, x_n >$. The most probable value of $f(x)$ is,

$$\begin{aligned} v_{MAP} &= \text{arg max}_{v_j \in V} P(v_j|x_1, x_2, \cdots, x_n) && (31) \\ &= \text{arg max}_{v_j \in V} \frac{P(x_1, x_2, \cdots, x_n|v_j)P(v_j)}{P(x_1, x_2, \cdots, x_n)} && (32) \\ &= \text{arg max}_{v_j \in V} P(x_1, x_2, \cdots, x_n|v_j)P(v_j) && (33) \end{aligned}$$

In Naive Bayes we assume $P(x_1, x_2, \cdots, x_n|v_j) = \prod_i P(x_i|v_j)$ meaning attributes are statistically independent. Now we get,

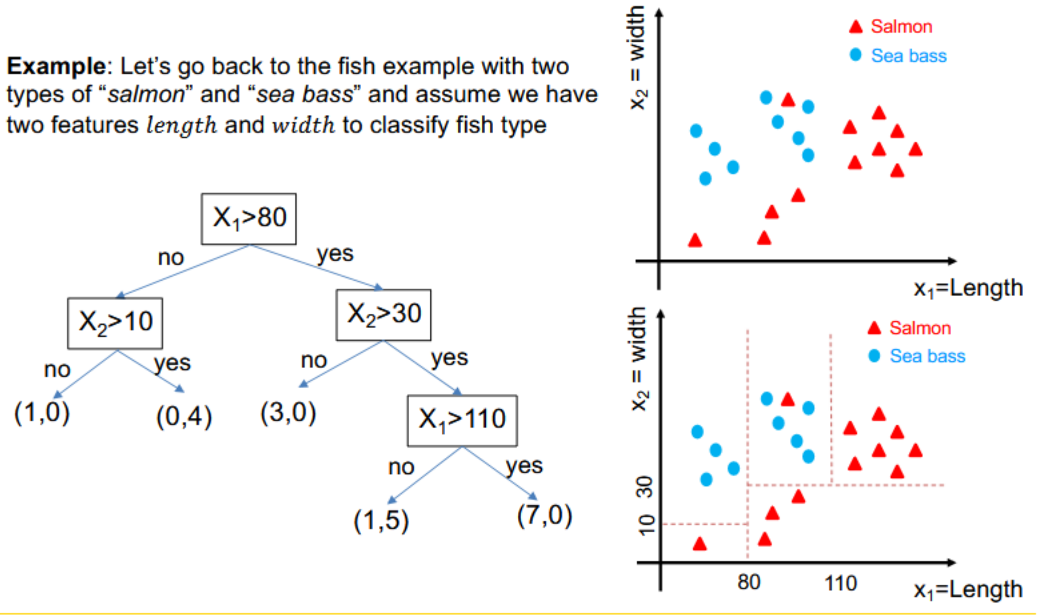
$$v_{NB} = \text{arg max}_{v_j \in V} P(v_j) \prod_i P(x_i|v_j)$$

8.10 Naive Bayes Algorithm

```
For each target value $v_j$,
[1]  $\hat{P}(v_j) \leftarrow$ estimate $P(v_j)$
[2]  For each attribute value $x_i$:
[3]      $\hat{P}(x_i|v_j) \leftarrow$ estimate $P(x_i|v_j)$ Classify the new instance,
```

$$v_{NB} = \text{arg max}_{v_j \in V} \hat{P}(v_j) \prod_i \hat{P}(x_i|v_j)$$

9 Decision Trees



Example: Let's go back to the fish example with two types of "*salmon*" and "*sea bass*" and assume we have two features *length* and *width* to classify fish type

Decision Trees work in a divide and conquer fashion where the data is split into subsets based on a criteria. If the data at the leaves are homogeneous/pure in terms of their classified value, then stop splitting. This type of splitting means that each leaf node assigns a classification. The difficult part of decision tress is at each node, which attribute/feature do we chose to split by?

9.1 Top-Down Induction of Decision Trees

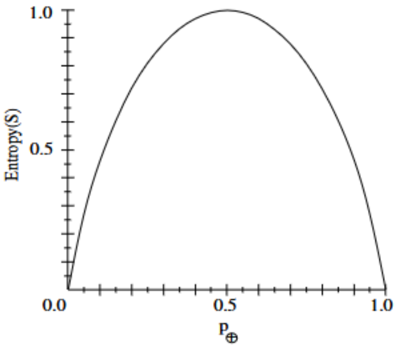
- $A \leftarrow$ the "best" decision attribute for next node to split examples
- Assign A as decision attribute for node
- For each value of A, create new child node
- Split training examples to child nodes
- if training examples perfectly classified, then stop. Else iterate over new child nodes.

9.2 Deciding on Splitting Attribute

We generally want to split by the attribute which results in the most "purity" in the child node. **Entropy** is used to measure the purity of a subset.

$$Entropy(S) = H(S) = -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

where S is the subset of training examples and $p_{\oplus}, p_{\ominus}$ are the portion of positive and negative examples in S.



9.3 Entropy

The optimal number of bits to encode a symbol with probability p is $-\log_2 p$. Suppose X can have one of $v_1, v_2, \cdots, v_k$,

$$P(X = v_1) = p_1, P(X = v_2) = p_2, \cdots, P(X = v_k) = p_k$$

Then on average, per symbol, the smallest number of bits to transmit a stream of symbols drawn from S's distribution is,

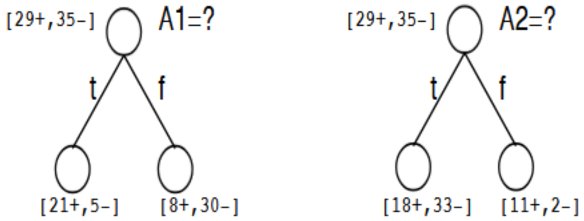
$$\begin{aligned} H(X) &= -p \log_2 p_1 - p_2 \log_2 p_2 - \cdots - p_k \log_2 p_k && (34) \\ &= -\sum_{j=1}^{k} p_j \log_2 p_j && (35) \end{aligned}$$

where $H(X)$ is the entropy of X. So the expected number of bits to encode $\oplus$ or $\ominus$ of a randomly drawn member of a set S is,

$$\begin{gathered} p_{\oplus}(-\log_2 p_{\oplus}) + p_{\ominus}(-\log_2 p_{\ominus}) \\ Entropy(S) = -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}. \end{gathered}$$

This is the Entropy that we want to minimise.

High entropy/impure set means X is very uniform and boring. Low entropy/pure set means X is not uniform and interesting.



$$H(S) = -\frac{29}{64}\log_2\frac{29}{64} - \frac{35}{64}\log_2\frac{35}{64} = 0.9936$$

9.4 Information Gain

The information gain $Gain(S, A)$ is the expected reduction in entropy due to sorting on A,

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

where v is the possible values of attribute A, S is the set of examples we want to split and S_v is the subset of examples where $X_A = v$. We want to find the attribution which maximises the gain. For the previous example, the gain from splitting on different attributes is given by,

$$\begin{aligned} Gain(S, A1) &= Entropy(S) - \left(\frac{|S_t|}{|S|} Entropy(S_t) + \frac{|S_f|}{|S|} Entropy(S_f)\right) \\ &= 0.9936 - \left(\left(\frac{26}{64}\left(-\frac{21}{26}\log_2\left(\frac{21}{26}\right) - \frac{5}{26}\log_2\left(\frac{5}{26}\right)\right)\right) + \right. \\ &\quad \left.\left(\frac{38}{64}\left(-\frac{8}{38}\log_2\left(\frac{8}{38}\right) - \frac{30}{38}\log_2\left(\frac{30}{38}\right)\right)\right)\right) \\ &= 0.9936 - (0.2869 + 0.4408) \\ &= 0.2658 \end{aligned}$$

$$\begin{aligned} Gain(S, A2) &= 0.9936 - (0.7464 + 0.0828) \\ &= 0.1643 \end{aligned}$$

So we chose $A1$ since it gives the larger expected reduction in entropy.

9.5 Gain Ratio

An issue with information gain is that it is biased towards attributes with large number of values/categories. The gain ration looks to fix this,

$$SplitEntropy(S, A) = -\sum_{v \in Values(A)} \frac{|S_v|}{|S|} \log_2 \frac{|S_v|}{|S|}$$

where,

- A: candidate attribute
- v: possible values of A
- S: Set of examples $\{X\}$ at the node
- S_v: subset where $X_A = v$

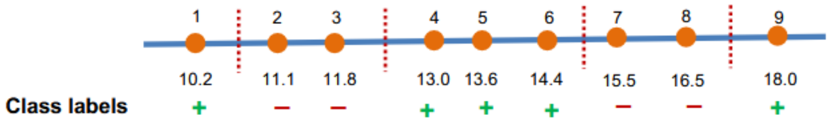
$$GainRatio(S, A) = \frac{Gain(S,A)}{SplitEntropy(S,A)}$$

9.6 Avoiding Overfitting

We can apply a method called **Pruning**. Pre-pruning is to stop growing when data split is not statistically significant. Post-pruning is to grow full tree, then remove sub trees which are overfitting (based on validation set).

9.7 Continuous Valued Attributes

Regular Decision Trees split on discrete attributes. To handle continuous attributes, we split the attribute on each boundary where the class changes. This can lead to $n-1$ splits in the worst case if we don't know the class labels.



9.8 Attributes with Cost

Sometimes we need to evaluate information gain relative to cost,

$$\frac{Gain^2(S,A)}{Cost(A)}$$

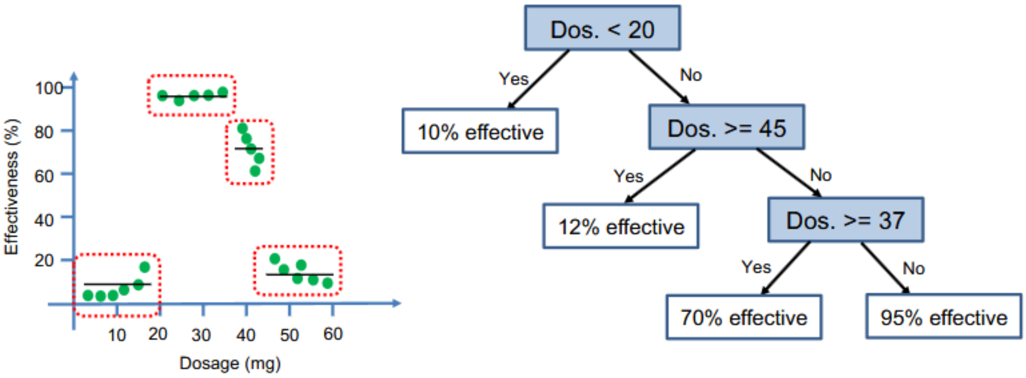
though this approach false positives a different cost to false negatives. We can define a loss function, $\lambda(\alpha_i | predicted\ class)$, and expected loss as,

$$R(\alpha_i|x) = \sum_{h \in H} \lambda(\alpha_i|predicted\ class)\ P(predicted\ class|x)$$

however if we use this loss function to split the tree then minimising R does not necessarily lead to the best long-term growth of the tree. So instead, we can grow the tree using information gain to its full depth and apply minimisation when post pruning.

10 Regression

Regression Trees are Decision Tress where each leaf performs a "mini Linear Regression".



Each leaf corresponds to average drug effectiveness in a different cluster of examples, the tree does a better job than *Linear Regression*

For splitting, the mean squared error is used for setting a threshholds,

$$MSE(Y_i) = \frac{1}{m}\sum_{j=1}^{m}(y_j - \overline{y})^2$$

where the MSE is equal to the variance of the examples in that subset. The weighted average of variance is,

$$weighted\ average\ gvariance = \sum_i^l \frac{|Y_i|}{|Y|} MSE(Y_i)$$

and we pick a threshold which minimises the weighted average of MSE/variance. That is, for each attribute, find the best split using the minimum weighted average variance and pick the minimum weighted average variance of those values.

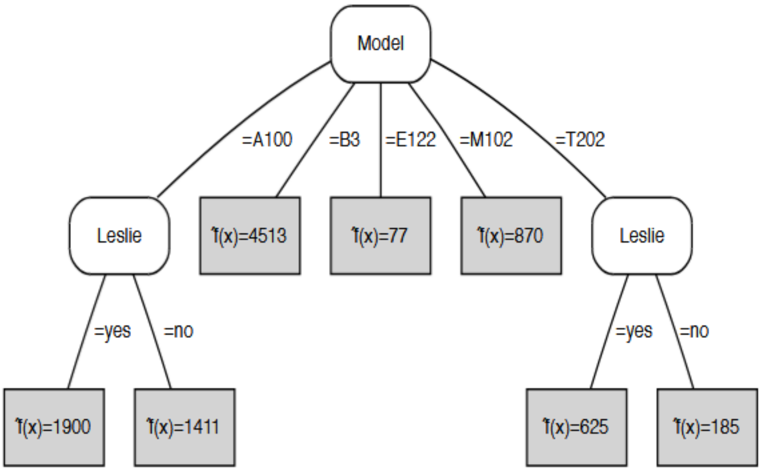
$$Model \ = \ [A100, B3, E112, M102, T202]$$
$$[1051, 1770, 1900][4513][77][870][99, 270, 625]$$

$$Condition \ = \ [excellent, good, fair]$$
$$[1770, 4513][270, 870, 1051, 1900][77, 99, 625]$$

$$Leslie = [yes, no]$$
$$[625,870,1900][77,99,270,1051,1770,4513]$$

- The means of the first split are 1574, 4513, 77, 870 and 331, and the weighted average of mean squared errors is 6.25×10^4.
- The means of the second split are 3142, 1023 and 267, with weighted average of mean squared errors 5.9×10^5;
- for the third split the means are 1132 and 1297, with weighted average of mean squared errors 1.72×10^6.

We therefore branch on $Model$ at the top level. This gives us three single-instance leaves, as well as three $A100$s and three $T202$s.

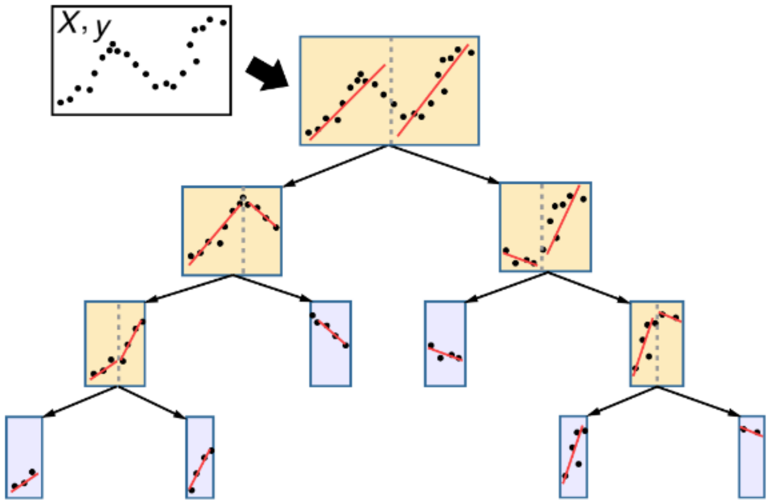


10.1 Overfitting in Regression Trees

Similar to Decision Trees, if we continue splitting all nodes that have a mean squared error bigger than zero, we will get a tree which fits the training data perfectly but will not generalise will to the test data. The simplest way to avoid this, similar to Decision Trees, is to only split examples when there is more than a minimum number, or using a maximum depth, or using a maximum number of leaves, etc.

10.2 Model Trees

Model Trees are similar to Regression Trees but with linear regression functions (or any model of your choice at each node) at each node. Linear Regression is applied to instances that reach a node after the full tree is built.



Model Trees use the splitting criteria of standard deviation reduction,

$$SDR = sd(T) - \sum_i \frac{|T_i|}{|T|} s \times sd(T_i)$$

where $T_1, T_2, \cdots$ are the sets from splits of data at node. Terminate splitting when standard deviation becomes smaller than certain fraction of sd for full training set or too few instances remain.

11 Kernel Methods

11.1 Scoring Classifier

A scoring classifier is a mapping $\hat{S} : \chi \to \mathbb{R}^k$. This indicates that a scoring classifier outputs a vector $\hat{S}(x) = (\hat{s}_1(x), \cdots, \hat{s}_k(x))$ where $\hat{s}_i(x)$ is the score assigned to class C_i for instance x. The score is how likely the class label C_i applies. If there are only two classes, then $\hat{s}(x)$ indicates the score of the positive class for instance x.

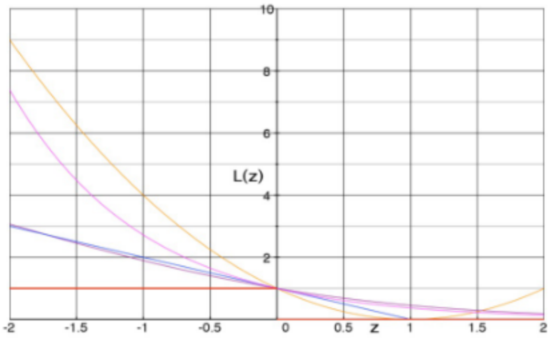
11.2 Margins and Loss Functions

If we take the true class $c(x)$ as $+1$ for positive examples and -1 for negative examples, then the quantity $z(x) = c(x) \times \hat{s}(x)$ is positive for correct predictions and negative for incorrect predictions. This quantity is called the margin assigned by the scoring classifier to the example.

We can reward large positive margins and penalise large negative values using a loss function,

$$L : R \to [0, \infty)$$

which maps each examples margin $z(x)$ to an associated loss $L(z(x))$. We will assume $L(0) = 1$ is the loss incurred by having an example on the decision boundary. We also have $L(z) \geq 1$ for $z < 0$ and $0 \leq L(z) < 1$ for $z > 0$. The average loss over a test set $Test$ is $\frac{1}{|Test|} \sum_{x \in Test} L(z(x))$.

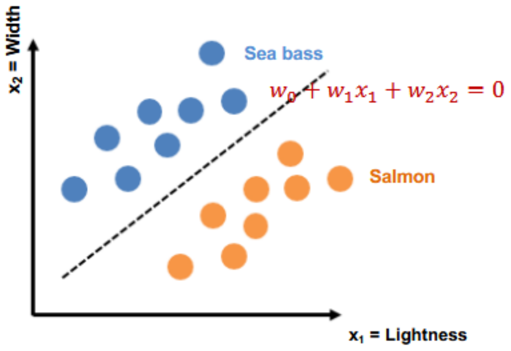


From bottom-left: (i) 0–1 loss $L_{01}(z) = 1$ if $z \leq 0$, and $L_{01}(z) = 0$ if $z > 0$; (ii) hinge loss $L_h(z) = (1 - z)$ if $z \leq 1$, and $L_h(z) = 0$ if $z > 1$; (iii) logistic loss $L_{log}(z) = log_2(1 + \exp(-z))$; (iv) exponential loss $L_{exp}(z) = \exp(-z)$; (v) squared loss $L_{sq}(z) = (1 - z)^2$ (can be set to 0 for $z > 1$, just like hinge loss).

11.3 Perceptron

Perceptron is an algorithm for binary classification that uses a linear prediction function. If we have two attributes/features of x_1 and x_2 then we can predict the target function $f(x)$ with:

$$f(x) = \begin{cases} +1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 > 0 \\ -1 \text{ otherwise} \end{cases}$$



This can be generalised to n attributes,

$$f(x) = \begin{cases} +1 \text{ if } w_0 + w_1x_1 + \cdots + w_nx_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

and if we add $x_0 = 1$ to the feature vector,

$$f(x) = \begin{cases} +1 \text{ if } \sum_{i=0}^{n} w_ix_i > 0 \\ -1 \text{ otherwise} \end{cases}$$

where $\sum_{i=0}^{n} w_ix_i$ is just the dot product $w.x$. Perceptron generalises these ideas into a simple classification method,

$$f(x) = \begin{cases} +1 \text{ if } w.x > 0 \\ -1 \text{ otherwise} \end{cases} \qquad \hat{y} = f(x) = sgn(w.x)$$

where sgn is the sign function. Now we have to find a good set of weights using our training set $x_1x_2, \cdots, x_m$ with labels $y_1, y_2, \cdots, y_m$ where x_j corresponds to observation j and is a vector of n features $x_j = [x_{j0}, x_{j1}, \cdots, x_{jn}]$. The perceptron algorithm initialises all weights w_i to zero and learns the weights using the following update rule,

$$w := w + \tfrac{1}{2}(y_j - f(x_j))x_j.$$

There are four cases:

- $y = +1, f(x) = +1 \Rightarrow (y - f(x)) = 0$
- $y = +1, f(x) = -1 \Rightarrow (y - f(x)) = +2$
- $y = -1, f(x) = +1 \Rightarrow (y - f(x)) = -2$
- $y = -1, f(x) = -1 \Rightarrow (y - f(x)) = 0$

We only want to update w when the prediction mismatches the actual class label (misclassification) and otherwise remains the same. For misclassified instances we can write:

$$w := w + y_j x_j$$

11.4 Perceptron Algorithm

1. Initialise all weights w to zero
2. Iterate through the training data. For each training sample, classify the sample if the prediction was incorrect.
3. Repeat step 2 some number of times.

11.5 Extending Linear Classifiers

Linear classifiers can not always model nonlinear class boundaries. To allow them to do so, we can map attributes into a new space consisting of combinations of attribute values. We use a trick called **duality** to achieve this. Duality is used when given an optimisation problem (primal problem), we can construct another optimisation problem (dual problem) and use this problem to solve our original problem. In convex optimisation problems, the optimal value of the primal and dual problems are equal under a constraint qualification condition.

11.6 Perceptron Classifiers in Dual Form

Every time an example x_i is misclassified, add $y_i x_i$ to the weight vector. After training, each example has been misclassified zero or more times. Denoting this number as a_i for example x_i, the weight vector for m observations can be expressed as,

$$w = \textstyle\sum_{i=1}^{m} a_i y_i x_i.$$

In the dual instance-based view of linear classification, we are learning instance weights a_i rather than feature weights w_j. An instance x is classified as,

$$\hat{y} = f(x) = sgn(w, x) \tag{36}$$

$$\hat{y} = sgn\left(\sum_{i=1}^{m} a_i y_i (x_i.x)\right) \tag{37}$$

During training, the only information needed about the training data is the pairwise dot products: the m-by-m matrix $G = XX^T$ containing these dot products called the Gram matrix.

$$G(\mathrm{x}_1, \ldots, \mathrm{x}_m) = \begin{bmatrix} \mathrm{x}_1.\mathrm{x}_1, & \mathrm{x}_1.\mathrm{x}_2, & \ldots, & \mathrm{x}_1.\mathrm{x}_m \\ \mathrm{x}_2.\mathrm{x}_1, & \mathrm{x}_2.\mathrm{x}_2, & \ldots, & \mathrm{x}_2.\mathrm{x}_m \\ & & \vdots & \\ \mathrm{x}_m.\mathrm{x}_1, & \mathrm{x}_m.\mathrm{x}_2, & \ldots, & \mathrm{x}_m.\mathrm{x}_m \end{bmatrix}$$

We can use nonlinear mapping to map attributes into a new space consisting of combinations of attribute values,

$$x \rightarrow \psi(x)$$

Then the perceptron decision will be,

$$\hat{y} = sgn\left(\textstyle\sum_{i=1}^{m} a_i y_i (\psi(x_i).\psi(x))\right)$$

So all we need is the dot product in the new feature space.

11.7 The Kernel Trick

Let $x = (x_1, x_2)$ an $x = (x_1', x_2')$ be two data points, and consider the following mapping to a three-dimensional feature space,

$$(x_{1,2}\ ) \rightarrow (x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

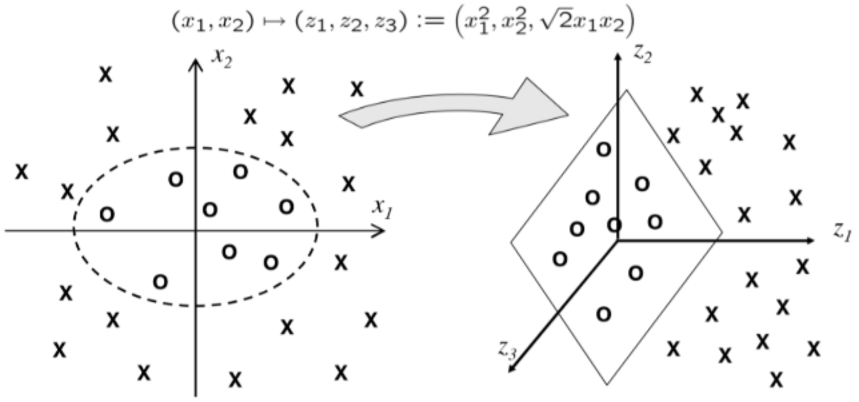
The points in the new feature space corresponding to x and x' are,

$$z = (x_1^2, x_2^2, \sqrt{2}x_1x_2) \text{ and } z' = (x_1'^2, x_2'^2, \sqrt{2}x_1'^2x_2'^2)$$

and the dot product of these features are,

$$z.z' = x_1^2x_1'^2 + x_2^2x_2'^2 + 2x_1x_1'x_2x_2' = (x_1x_1' + x_2x_2')^2 = (x.x')^2$$

This shows that by squaring the dot product in the original space we obtain the dot product in the new space without actually considering the feature vectors! A function that calculates the dot product in the new space directly from the vectors in the original space is called a **kernel**. Here the kernel is $K(x_1, x_2) = (x_1.x_2)^2$. A valid kernel function is equivalent to a dot product in some space. The kernel trick helps us go to a high-dimensional space without paying the price.



$(x_1, x_2) \mapsto (z_1, z_2, z_3) := \left(x_1^2, x_2^2, \sqrt{2}x_1x_2\right)$
x_1
x_2
z_1
z_2
z_3

For $\{x, x'\} \in \chi$, a kernel function (similarity function tat corresponds to a dot product in some expanded feature space) $L(x, x')$ can be express the dot product in another space. If $\psi(x)$ is the input x in the new feature space, then computation becomes much simpler if we have a kernel function that $K(x, x') = \psi(x).\psi(x')$.

11.8 Useful Kernels

- Polynomial Kernel: $K(x, x') = (x.x' + x)^q$
- RBF Kernel: $K(x, x') = \exp\left(-\frac{||x-x'||^2}{2\sigma^2}\right)$

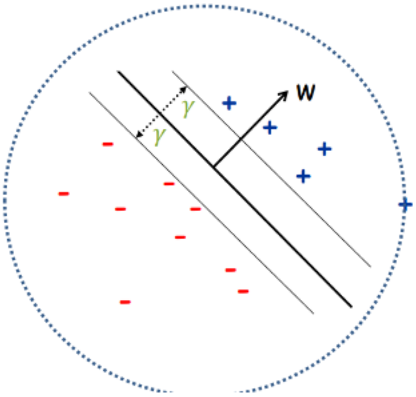
11.9 Nonlinear dual perceptron

Using the kernel trick, the nonlinear perceptron can be solved using the dual form and the kernel as follows,

$$\hat{y} = \text{sign}\left(\sum_{i=1}^{m} a_i y_i K(x_i, x)\right)$$

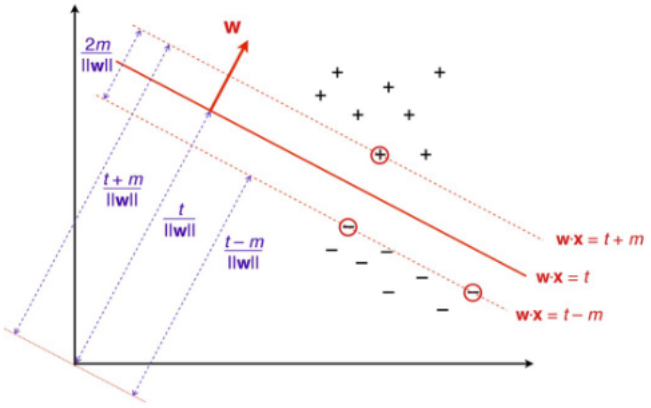
11.10 Perceptron Convergence

Perceptron Convergence Theorem: $R = max_i||x_i||_2$. The number of mistakes made by the perceptron is at most $(\frac{R}{\gamma})^2$ where R is the radius of a circle such that the furthest point lies on its boundary and γ is the length of the margin.



12 Support Vector Machines

Going back to linear classification, if there are multiple lines that can split a decision boundary which line is best. The best line is the line which maximises the margin. Support Vector Machines (SVMs) attempt to learn the decision boundary which maximises the margin.



The circled data points are **support vectors**, which are the training examples nearest to the decision boundary. The SVM finds the decision boundary that maximises the margin $\frac{m}{||w||}$.

12.1 Methodology

Let x_s be the closest point to the separating hyperplane with the following equation,

$$w.x = t.$$

Some simplifications are made for mathematical convenience first,

- Pull out $x_0 : w[w_1, \cdots, w_n]$ and $w_0 = -t$, therefore we will have $w.x - t = 0$.
- Normalise w. We know that $|w.x_s - t| > 0$ and that we can scale w and t together without having any effect on the hyperplane. We choose the scale such that $|w.x_s - t| = 1$ which means $m = 1$.

Now we have a line equation $w.x - t = 0$ and constraint $|w.x_s - t| = 1$, where x_s is the closest point to the separating hyperplane and w is perpendicular to the hyperplane (since for every two points $w.x - t = 0$ and $w.x'' - t = 0 \Rightarrow w.(x' - x'' = 0)$; dot product is zero).

We know that the distance between point x_s and hyperplane $w.x - t = 0$ is,

$$distance = \frac{|w.x_s - t|}{||w||}$$

and we have $|w.x_s - t| = 1$, so,

$$distance = \frac{1}{||w||}$$

The distance is the margin of our classifier, which we want to maximise,

$$max\frac{1}{||w||} \text{ subject to } min_{i=1,\cdots,m}|w.x_i - t| = 1$$

But this isn't very friendly since we have *min* in the constraint. Our focus is on the points that the hyperplane can predict correctly. So we have,

$$|w.x_i - t| = y_i(w.x_i - t)$$

And we can change the min in constraint as follows,

$$y_i(w.x_i - t) \geq 1 \text{ for } i = 1, \cdots, m$$

Then transform the maximisation problem into the following minimisation problem,

$$min_w\frac{1}{2}||w||^2 \text{ subject to } y_i(w._i - t) \geq 1 \text{ for } i = 1, \cdots, m,\ w \in \mathbb{R}^n \text{ and } t \in \mathbb{R}.$$

We can solve the above with Lagrangian Multipliers.

12.2 Lagrangian multipliers

Constrained optimisation problems are generally expressed as $min_{x_1,\cdots,x_n} f(x_1,\cdots,x_n)$ subject to,

$$g_1(x_1.\cdots,x_n) \leq 0, g_2(x_1,\cdots,x_n) \leq 0, \cdots, g_k(x_1,\cdots,x_n) \leq 0.$$

The Lagrange multiplier methods involve the modification of the objective function through the addition of terms that describe the constraints. The objective function $f(x)$ is augmented by the constraint equations through a set of non-negative multiplicative Lagrange multipliers, $a_j \geq 0$ and is called the dual Lagrangian:

$$\mathcal{L}(x_1,\cdots,x_n,a_1,\cdots,a_k) = f(x_1,\cdots,x_n) + \textstyle\sum_{j=1}^{k} a_j g_j(x_1,\cdots,x_n)$$

In Lagrangian form, the optimisation problem becomes:

$$max_{a_1,\cdots,a_m}\ min_{x_1,\cdots,x_n} \mathcal{L}(x_1,\cdots,x_n,a_1,\cdots,a_m) \text{ such that } a_j \geq 0\ \forall\ j.$$

minimising first with respect to $x_1,\cdots,x_n$ and then maximising with respect to $a_1,\cdots,a_m$. Adding the constraints with multipliers a_i for each training example gives the Lagrange function,

$$\mathcal{L}(w,t,a_1,\cdots,a_m) = \frac{1}{2}||w||^2 - \sum_{i=1}^{m} a_i(y_i(w.x_i - t) - 1) \tag{38}$$

$$= \frac{1}{2}||w||^2 - \sum_{i=1}^{m} a_i y_i(w.x_i) + \sum_{i=1}^{m} a_i y_i t + \sum_{i=1}^{m} a_i \tag{39}$$

$$= \frac{1}{2}w.w - w\sum_{i=1}^{m} a_i y_i(w.x_i) + t\sum_{i=1}^{m} a_i y_i + \sum_{i=1}^{m} a_i \tag{40}$$

$$\tag{41}$$

First we minimise $\mathcal{L}$ with respect to w and t. By taking the partial derivative of the Lagrange function respect to t and setting to 0 we find, $\sum_{i=1}^{m} a_i y_i = 0$. Similarly, by taking the partial derivative of the Lagrange function with respect to w and setting to 0 we obtain, $w = \sum_{i=1}^{m} a_i y_i x_i$ which is the same expression as derived for perceptron. In the SVM however,a_i are non negative reals where $a_i > 0$ means that a training sample is nearest to the decision boundary.

Eliminating w and t lead to the dual Lagrangian,

$$\mathcal{L}(a_1,\cdots,a_n) = -\frac{1}{2}\left(\sum_{i=1}^{m} a_i y_i x_i\right) \cdot \left(\sum_{i=1}^{m} a_i y_i x_i\right) + \sum_{i=1}^{m} a_i \tag{42}$$

$$= -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} a_i a_j y_i y_j x_i x_j + \sum_{i=1}^{m} a_i \tag{43}$$

The dual optimisation problem for SVMs is the maximise the dual Lagrangian under positivity constraints and one equality constraint,

$$a_1^*, \cdots, a_m^* = \text{arg max}_{a_1,\cdots,a_n} - \tfrac{1}{2}\textstyle\sum_{i=1}^m \sum_{j=1}^m a_i a_j y_i y_j x_i x_j + \sum_{i=1}^m a_i$$

subject to,

$$a_i \geq 0, 1 \leq i \leq m, \textstyle\sum_{i=1}^m a_i y_i = 0$$

Or equivalently,

$$a_1^*, \cdots, a_m^* = \text{arg min}_{a_1,\cdots,a_n} - \tfrac{1}{2}\textstyle\sum_{i=1}^m \sum_{j=1}^m a_i a_j y_i y_j x_i x_j - \sum_{i=1}^m a_i$$

subject to,

$$a_i \geq 0, 1 \leq i \leq m, \textstyle\sum_{i=1}^m a_i y_i = 0$$

When you solve this optimisation problem and get $a_1^*, \cdots, a_m^*$ you will see that it will be mostly zero and only for the points which are closest to the separating line, a_i will be non-zero and positive. This points are the support vectors.

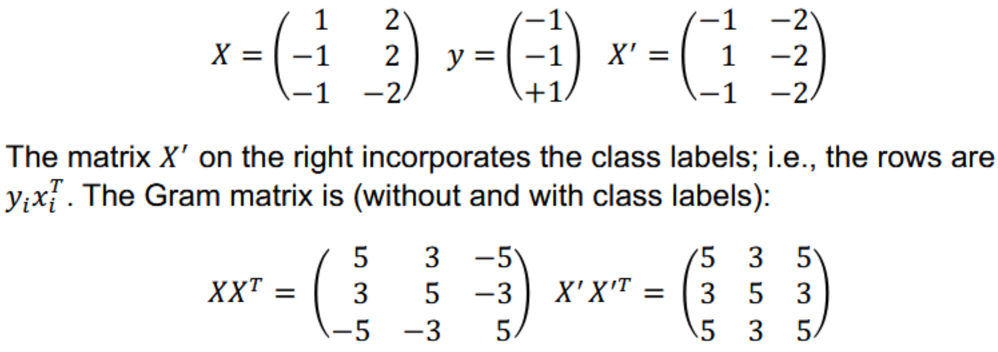
$$w = \textstyle\sum_{x_i \in \{support\ vectors\}} a_i y_i x_i$$

Solve for t using any of the support vectors,

$$y_i(w.x_i - t) = 1$$

assuming separable data. The more separated the classes, the larger the margin, the better the generalisation. Instances closest to the maximum margin hyperplane are support vectors. Support vectors define a maximum margin hyperplane, all other instances can be deleted without changing the position and orientation of the hyperplane.

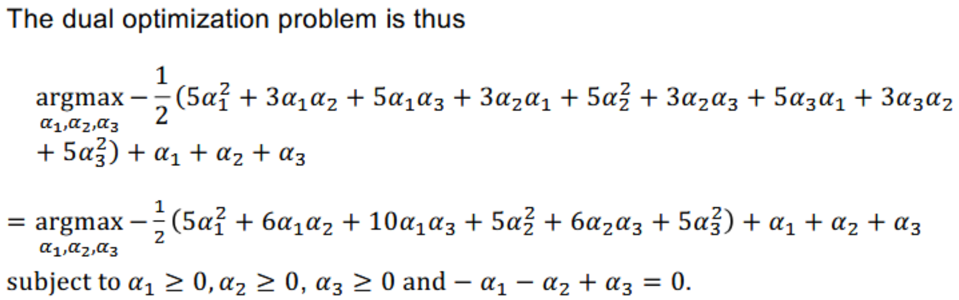
12.3 Maximum Margin Classifiers



$$X = \begin{pmatrix} 1 & 2 \\ -1 & 2 \\ -1 & -2 \end{pmatrix} \quad y = \begin{pmatrix} -1 \\ -1 \\ +1 \end{pmatrix} \quad X' = \begin{pmatrix} -1 & -2 \\ 1 & -2 \\ -1 & -2 \end{pmatrix}$$

The matrix X' on the right incorporates the class labels; i.e., the rows are $y_i x_i^T$. The Gram matrix is (without and with class labels):

$$XX^T = \begin{pmatrix} 5 & 3 & -5 \\ 3 & 5 & -3 \\ -5 & -3 & 5 \end{pmatrix} \quad X'X'^T = \begin{pmatrix} 5 & 3 & 5 \\ 3 & 5 & 3 \\ 5 & 3 & 5 \end{pmatrix}$$



The dual optimization problem is thus

$$\underset{\alpha_1,\alpha_2,\alpha_3}{\operatorname{argmax}} -\frac{1}{2}(5\alpha_1^2 + 3\alpha_1\alpha_2 + 5\alpha_1\alpha_3 + 3\alpha_2\alpha_1 + 5\alpha_2^2 + 3\alpha_2\alpha_3 + 5\alpha_3\alpha_1 + 3\alpha_3\alpha_2 + 5\alpha_3^2) + \alpha_1 + \alpha_2 + \alpha_3$$

$$= \underset{\alpha_1,\alpha_2,\alpha_3}{\operatorname{argmax}} -\frac{1}{2}(5\alpha_1^2 + 6\alpha_1\alpha_2 + 10\alpha_1\alpha_3 + 5\alpha_2^2 + 6\alpha_2\alpha_3 + 5\alpha_3^2) + \alpha_1 + \alpha_2 + \alpha_3$$

subject to $\alpha_1 \geq 0, \alpha_2 \geq 0,\ \alpha_3 \geq 0$ and $-\alpha_1 - \alpha_2 + \alpha_3 = 0$.

Using the equality constraint, we eliminate one of the variables, say a_3 and simplify the objective function to,

$$\arg\max_{a_1,a_2} -\tfrac{1}{2}(20a_1^2 + 32a_1a_2 + 16a_2^2) + 2a_1 2a_2$$

setting partial derivatives to 0 we get,

$$-20a_1 - 16a_2 + 2 = 0 \qquad (44)$$

$$-16a_1 - 16a_2 + 2 = 0 \qquad (45)$$

we therefore obtain the solution $a_1 = 0$ and $a_2 = a_3 = \frac{1}{8}$. We then have $w = \frac{1}{8}(x_3 - x_2) = (0, -\frac{1}{2})^T$ resulting in a margin of $\frac{1}{||w||} = 2$. Finally t can be obtained using any support vector, say x_2, which gives $-1.(-1 - t) = 1$ and hence $t = 0$.

12.4 Nonlinear SVMs

We can transform our input feature $x \in \chi$ into a new feature space $z \in \mathcal{Z}$. Replacing x in the Lagrangian function gives,

$$\mathcal{L}(\alpha) = -\tfrac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} a_i a_j y_i y_j z_i.z_j + \sum_{i=1}^{m} a_i$$

The support vectors in this case are maintained in the new feature $\mathcal{Z}$ feature space however they correspond to some of the points in the original space. If $z = \psi(x)$ the Lagrangian becomes,

$$\mathcal{L}(\alpha) = -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} a_i a_j y_i y_j \psi(x_i).\psi(x_j) + \sum_{i=1}^{m} a_i$$

and if we can find a kernel function such that $K(x_i, x_j) = \psi(x_i).\psi(x_j)$ then,

$$\mathcal{L}(\alpha) = -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} a_i a_j y_i y_j K(x_i, x_j) + \sum_{i=1}^{m} a_i$$

The output of your SVM for x is,

$$\hat{y} = sign(w.z - t) \tag{46}$$

$$w = \sum_{z_i \in SV} a_i y_i z_i \tag{47}$$

$$\Rightarrow \hat{y} = sign(\sum_{a_i > 0} a_i y_i K(x_i, x) - t) \tag{48}$$

where $t = \sum_{a_i>0} a_i y_i K(x_i, x_j) - y_j$ for any support vector x_j. So this is your SVM model which can change depending on the kernel you choose.

12.5 Soft Margin SVM

Misclassified examples may break the separability assumption, so we introduce slack variables ξ_i to allow for misclassification of instances. When classes were linearly separable we had,

$$y_i(w.x_i - t) \geq 1$$

but if we get some data that violates this slack value,

$$y_i(w.x_i - t) \geq 1 - \xi_i \text{ and } \xi_i \geq 0$$

so the total violation is,

$$total\ violation = \sum_{i=1}^{m} \xi_i$$

this is the measure of margin and now we optimise for,

$$min\frac{1}{2}||w||^2 + C\sum_i \xi_i$$

where the parameter C bounds influence of any one instance on the decision boundary. If C goes to infinity, you go back to the hard margin (indicating your SVM can not tolerate error). High values of user defined C means that margin errors incur high penalty and a low value for C permits more margin errors in order to achieve a large margin.

$$w^*, t^*, \xi^* = arg\ min_{w,t,\xi_i} \tfrac{1}{2}||w||^2 + C \textstyle\sum_{i=1}^{m} \xi_i$$
$$\text{subject to } y_i(w.x_i - t) \geq 1 - \xi_i \text{ and } \xi_i \geq 0, 1 \leq i \leq m$$

The Lagrange function then follows as,

$$\mathcal{L}(w, t, \xi_i, \alpha_i, \beta_i) = \frac{1}{2}||w||^2 + C\sum_{i=1}^{m} \xi_i - \sum_{i=1}^{m} \alpha_i(y_i(w.x_i - t) - (1 - \xi_i)) - \sum_{i=1}^{m} \beta_i \xi_i \quad (49)$$

$$= \mathcal{L}(w, t, a_i) + \sum_{i=1}^{m}(C - \alpha_i - \beta_i)\xi_i \quad (50)$$

And similar to before we want to minimise with respect to w, t and ξ_i,

$$\nabla_w \mathcal{L} = w - \sum_{i=1}^{m} \alpha_i y_i x_i = 0 \quad (51)$$

$$\frac{\partial \mathcal{L}}{\partial t} = \sum_{i=1}^{m} \alpha_i y_i = 0 \quad (52)$$

$$\frac{\partial \mathcal{L}}{\partial \xi_i} = C - \alpha_i - \beta_i = 0 \Rightarrow C - \alpha_i = \beta_i \geq 0 \quad (53)$$

$$0 \leq \alpha_i \leq C \quad (54)$$

The only difference here is that α_i is upper bounded by C. A solution to the soft margin optimisation problem in dual form divides the training example into three cases:

- $\alpha_i = 0$, these are outside or on the margin
- $0 < \alpha_i < C$, these are the support vectors on the margin
- $\alpha_i = C$, these are on or inside the margin (non-margin support vectors). All the points that violate the margin are non-margin support vectors and contribute to w.

13 Ensemble Methods

Ensemble methods are a form of multi-level learning: learning a number of base-level models from the data, and learning to combine these models as an ensemble. They are often used to either decrease the bias and/or variance of a learning scheme. We can use the bias-variance decomposition (Total expected error ≈ bias^2 + variance) to analyse how much restriction to a single training set affects performance. The expected error of individual ensemble members can be decomposed as follows:

- Bias = expected error of the ensemble classifier on new data
- Variance = component of the expected error due to particular training set being used to build classifier

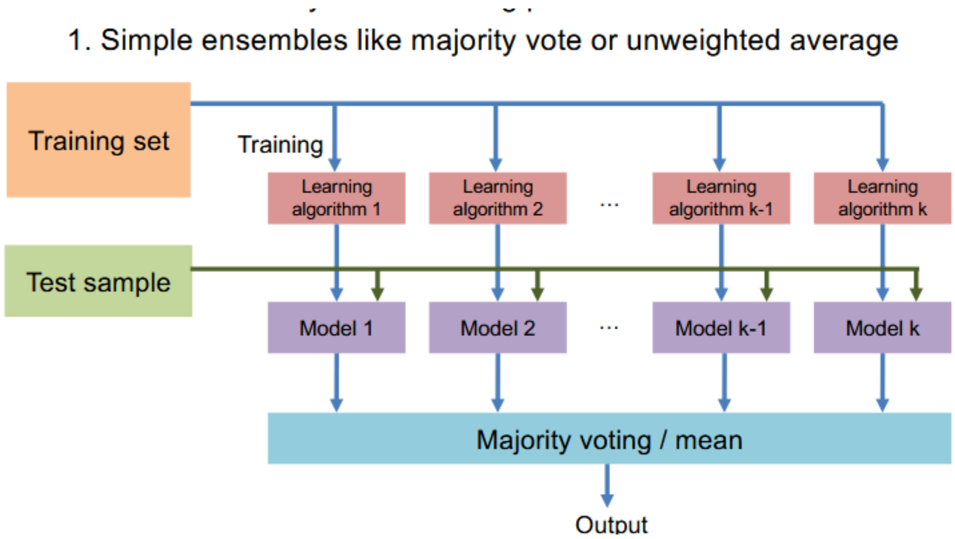
13.1 Stability

For a given data distribution $\mathcal{D}$,

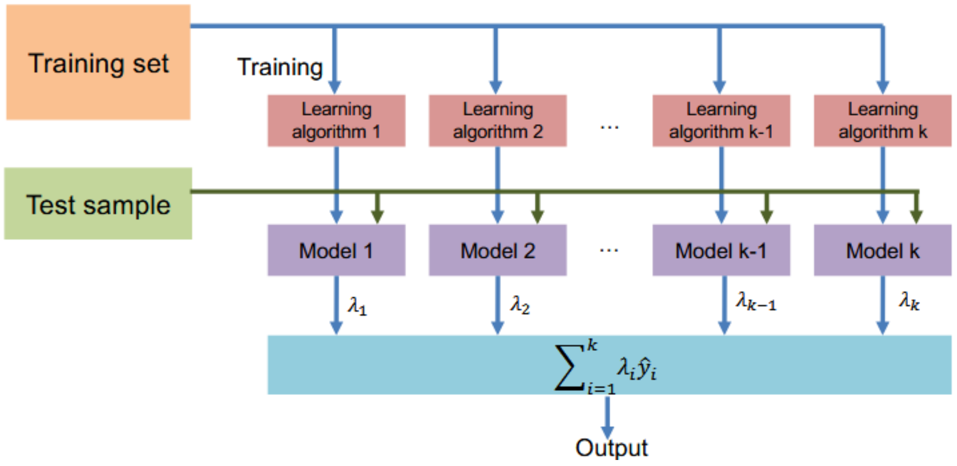
- Train algorithm L on training sets S_1, S_2 sampled from $\mathcal{D}$
- Expect that the model from L should be the same on both S_1 and S_2
- If so, we say that L is a stable learning algorithm
- otherwise, it is unstable

Common stable algorithms are Naive Bayes, Ridge Regression and kNN (for some k). Unstable algorithms are Decision Tree learning. Stable algorithms typically have high bias and unstable algorithms typically have high variance.

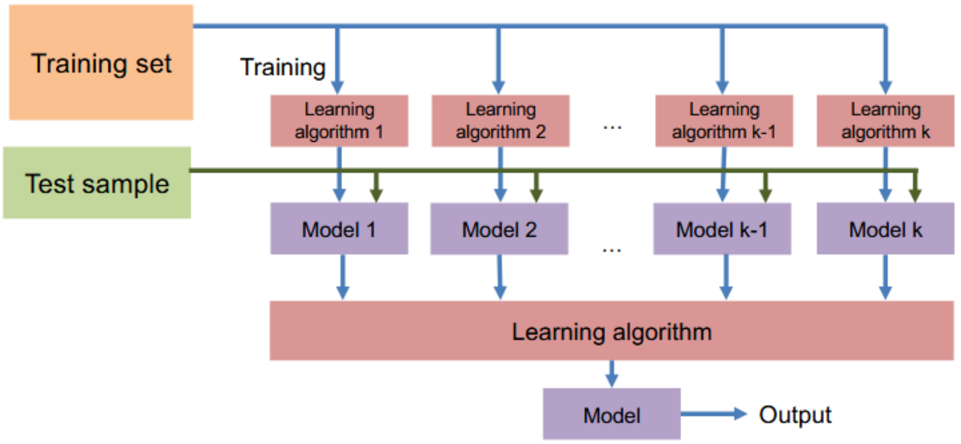
13.2 Simple Ensembles



2. weighted averages / weighted votes: Every model gets a weight (e.g. depending on its performance)



3. Treat the output of each model as a feature and train a model on that



4. Mixture of experts: Weight $\alpha_i(x)$ indicates an "expertise". Divides the feature space into homogeneous regions and can choose to use either the weighted average or pick the model with the largest expertise.

5. "Bagging" method ("**B**ootstrap **Agg**regation"): training many classifiers but each with only a portion of data. Then aggregate through model averaging/majority voting.

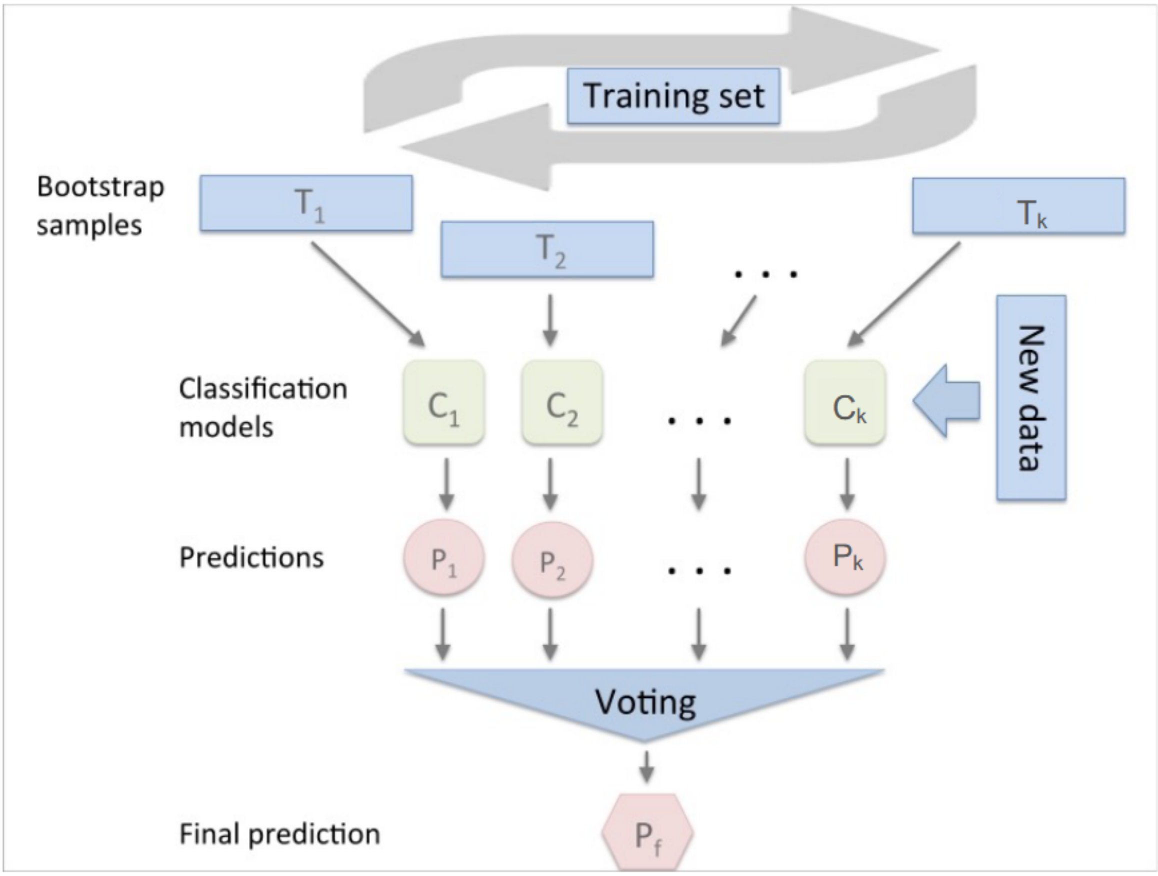
13.3 Bagging

Bootstrap:

- Create a random subset of data by sampling with replacement.
- Draw m' samples from m samples with replacement ($m' \leq m$).

Bagging:

- Repeat k times to generate k subsets. Some subsets repeated and some left out.
- Train one classifier on each subset.
- To test, aggregate the output of k classifiers that you trained in the previous step using either majority vote/unweighted average.



Bagging is applied to a collection of low-bias, high variance models and by averaging them, the bias will not be affected however the variance will be reduced. **Primarily a variance-reduction technique**.

13.4 Bagging Error

We assume that learners are independent and each learner makes an error with probability p (all learners have same error). If we use majority voting to decide the output, then the error happens if more than $\frac{k}{2}$ of learners make an error. So the error for majority voting is,

$$\sum_{k'>\frac{k}{2}} \binom{k}{k'} p^k (1-p)^{k-k'}$$

where $\binom{k}{k'}p^k(1-p)^{k-k'}$ represents the probability that k' our of k learners makes an error. If $k = 10$ and $p = 0.1$, then the error of majority voting is < 0.001.

13.5 Bagging Trees & Random Forest

Bagging Trees are classification trees that are fitted to a training sample using a some number of bootstrap samples. Generally no pruning is used and trees are quite different/high variance. This results in reducing variance by averaging and leaving bias unchanged.

For lots of data, we have extra variability in the learners and introduce more randomness to the procedure. An ensemble tree with random subset of features is called a**Random Forest**. For every model, we use only a subset of features to create diversity in the trees and taking the average/majority vote over trees(learners). The algorithm (Leo Breiman's Random Forest Algorithm) is as follows,

- Select m' number of samples from dataset with replacement (boot strap sample)
- Select a subset of features randomly (subspace sample)
- Build a full tree without pruning using selected features and samples
- Repeat previous steps k number of times

For every new sample, predict the output using all trees and then take the average/majority vote. This forces more diversity among trees in the ensemble and takes less time to train.

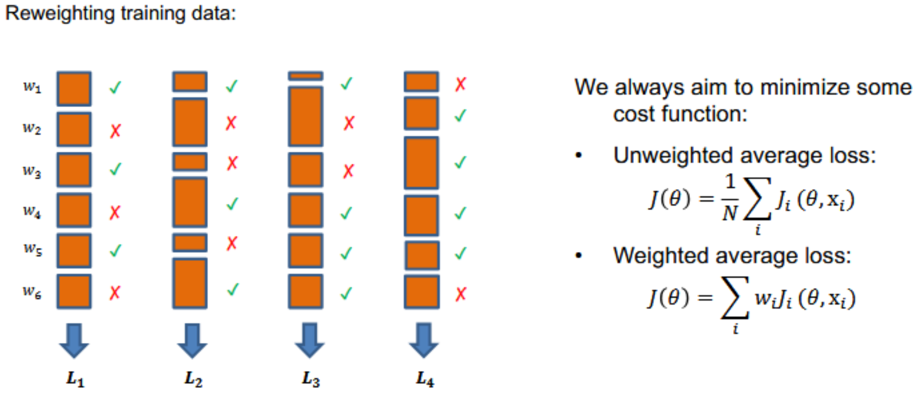
13.6 Boosting

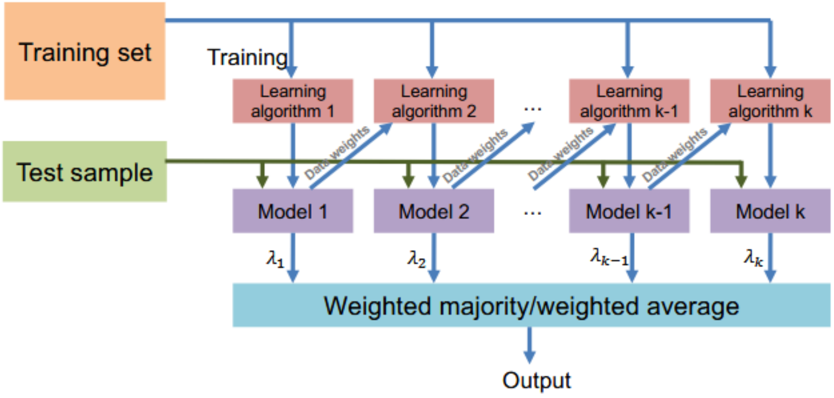
Problems arise from previous ensemble methods when all learners make mistakes in the same region. **Boosting** is a solution to this, that converts a sequence of weak learners into a very complex model to predict. **Primarily a bias reduction technique**. Uses subsequently trained, weak learners such that new learners focus on errors of earlier learners. Every subsequent learner tries to predict the tough samples correctly by operating on a weighted training set in favour of missclassified instances. This re-weighted training set operates as follows,

- Start with same weight for all instances
- Misclassifed instances gain higher weights so the next classifier is more likely to classify it correctly
- Correctly classified instances lose weight

The boosting algorithm is as follows,

- Set $w_i = \frac{1}{m}$ for $i = 1, \cdots, m$
- Train model L_j using dataset with weight w. Increase w_i for misclassified instances of L_j.
- Repeat previous step until sufficient number of hypothesis
- Ensemble hypothesis is the weighted majority/average of k learners $L_1, \cdots, L_k$ with weight $\lambda_1, \cdots, \lambda_k$ which are proportional to the accuracy of L_j


Reweighting training data:
w_1
w_2
w_3
w_4
w_5
w_6
L_1
L_2
L_3
L_4
We always aim to minimize some cost function:
Unweighted average loss:
$J(\theta) = \frac{1}{N}\sum_i J_i(\theta, x_i)$
Weighted average loss:
$J(\theta) = \sum_i w_i J_i(\theta, x_i)$



13.7 AdaBoost (Adaptive Boosting)

AdaBoost combines stump trees (trees with one node and two leaves as the base learner) into a forest to boost performance. Stumps are created sequentially and the error of each stump affects the training data weight in the next stump. Each stump gets a different weight λ_i, based on performance, for the final classification decision.

The algorithm is as follows,

- Initialise all weights: $w_i = \frac{1}{m} \forall i$.

- For $j = 1$ to k do,
  - Learn L_j (jth stump) with data weight w
  - Calculate weighed error: $\epsilon_j = \frac{\sum_{i=1}^{m} w_i'^{(j)}}{\sum_{i=1}^{m} w_j^{(j)}}$ where $w_i'^{j}$ are the weights for misclassified instances $(L_j(x_i) \neq y_i)$
  - Calculate a stump weight $\lambda_j = \frac{1}{2}\log(\frac{1-\epsilon_j}{\epsilon_j})$
  - Update w_i^{j+1} to $w_i^{(j)}\exp(\lambda_i)$ for missclassified instances or to $w_i^{(j)}\exp(-\lambda_i)$ for correct instances

Prediction is made using final model: $y(x) = sign(\sum_{j=1}^{k} \lambda_i L_j(x))$.

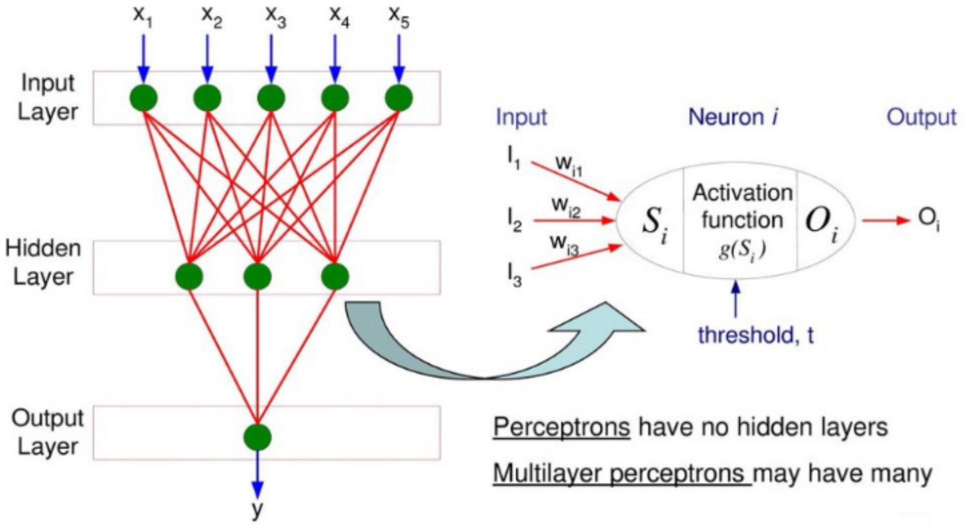
13.8 Gradient Boosting

Gradient boosting is using simple linear regression or simple regression trees as weak learner. It learns a sequence of predictors and sums the predictors to create more complex models. The models get more accurate at each step by adding more predictors. The procedure is as follows,

- Start with training a weak learner and make a set of predictions $\hat{y}_i$. The error of our prediction is $J(y_i, \hat{y}_i)$ for classification and $J = \frac{1}{2}\sum_i (y_i = \hat{y}_i)^2$ for regression.
- Improve $\hat{y}_i$ by gradually reducing the error. $\hat{y}_i = \hat{y}_i - \alpha\frac{\partial J(y,\hat{y})}{\partial \hat{y}_i}$ for classification or $\nabla J(y_i, \hat{y}_i) = y_i - \hat{y}_i$ and $\hat{y}_i = \hat{y}_i + \alpha\nabla J(y_i, \hat{y}_i)$ for regression.
- Each new learner estimates the gradient of loss $f_k(i)$.

14 Neural Network

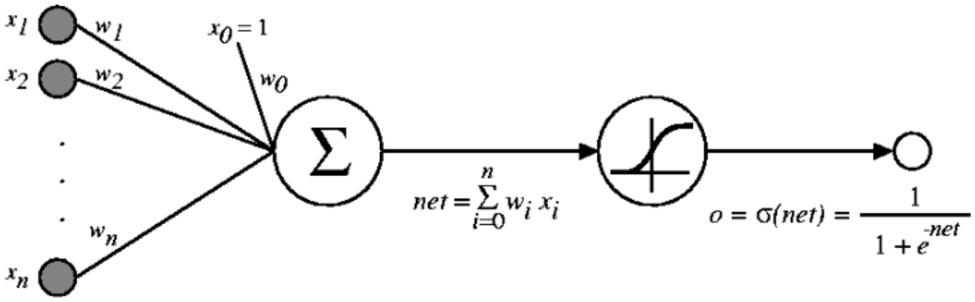
The idea of a neural network is to combine together many perceptrons to make a multiplayer perceptron; the classic neural network. This approach is particularly useful for functions that are not linearly separable.



Perceptrons have no hidden layers

Multilayer perceptrons may have many

14.1 Sigmoid Unit


x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1+e^{-net}}$

The **Sigmoid Units** works the same as a perceptron except that the step function has been replaced by a nonlinear sigmoid function. Nonlinearity makes it easy for the model to generalise or adapt to a variety of data to differentiate between the output.

14.2 Sigmoid Function

The sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ is used for its nice property $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$. This is particularly useful for **back propagation**; learning weights in a neural network via multiple applications of the chain rule.

14.3 Forward propagation

Assume we want to minimise the squared error ($\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$) over a set of training examples D. Then taking the derivative using the chain rule,

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 && (55)\\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) && (56)\\
&= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right) && (57)\\
&= -\sum_d (t_d - o_d)\frac{\partial o_d}{\partial net_d}\frac{\partial net_d}{\partial w_i} && (58)
\end{aligned}$$

In the above, t_d refers the dth neurons target/true value and o_d refers to the dth neurons output/predicted value. o is the sigmoid activation function evaluated at net ($o = \sigma(net) = \frac{1}{1+e^{-net}}$) and net_d is the sum of weights into the dth neuron $net = \sum_{i=0}^{n} w_i x_i$. We know that,

$$\begin{aligned}
\frac{\partial o_d}{\partial net_d} &= \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d) && (59)\\
\frac{\partial net_d}{\partial w_i} &= \frac{\partial(\overline{w}.\overline{x}_d)}{\partial w_i} = x_{i,d} && (60)
\end{aligned}$$

and so we derive,

$$\frac{\partial E}{\partial w_i} = -\sum_{d\in D}(t_d - o_d)o_d(1 - o_d)x_{i,d}$$

14.4 Backpropagation algorithm

When we calculated the error in the previous section, we used a batch gradient decent method as we were taking the error over all training samples d. This can be simplified if we instead use stochastic gradient decent; updating the weights for only one sample. So now,

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= -\sum_{d\in D}(t_d - o_d)o_d(1 - o_d)x_{i,d} && (61)\\
\frac{\partial E}{\partial w_i} &= -(t_d - o_d)o_d(1 - o_d)x_{i,d} \text{ for SGD} && (62)
\end{aligned}$$

We want the error for all output neurons and so we can generalise this error for k outputs as,

$$E = \sum_k E(o_k)$$

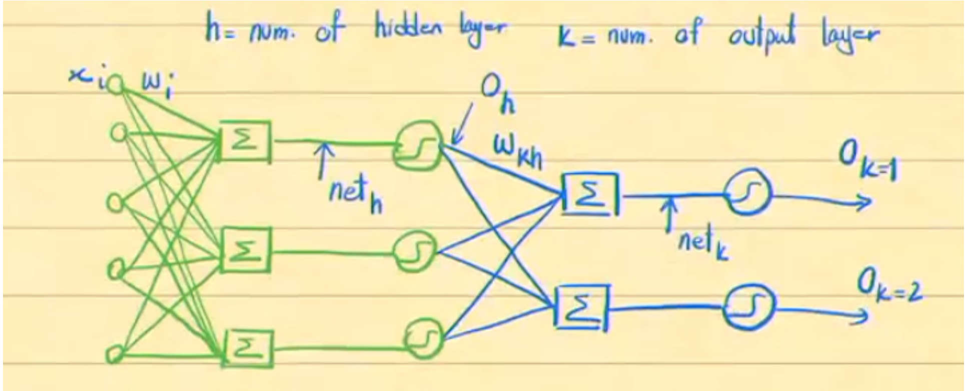
taking its derivative respect to w_i gives,

$$\tfrac{\partial E}{\partial w_i} = \textstyle\sum_k \tfrac{\partial E(o_k)}{\partial w_i}$$

If we take $\delta_k = (t_d - o_d)o_d(1 - o_d) = (t_k - o_k)\frac{\partial(o_k)}{\partial(net_k)}$ (from d to k outputs) then,

$$\frac{\partial E(o_k)}{\partial(w_i)} = \delta_k \frac{\partial(net_k)}{\partial(o_h)} \frac{\partial(o_h)}{\partial(net_h)} \frac{\partial(net_h)}{\partial(w_u)} \tag{63}$$

where o_h, net_h is the output and output and sum of weights into the hidden layer neuron h respectively.



The figure above shows how we are using the chain rule to take the derivative.

- First we take the outputs error $t_k - o_k$ and the outputs derivative with respect to net_k (accumulated as δ_k) which proceeds it,
- then take the derivative of net_k with respect to the hidden layers output o_h which proceeds net_k,
- then take the derivative of the hidden layers output o_h with respect to its net_h which proceeds it,
- Finally take the derivative of net_h with respect to the weight from the training sample w_i.

Evaluating the derivatives gives,

$$\frac{\partial(net_k)}{\partial(o_h)} = w_{kh} \tag{64}$$

$$\frac{\partial(o_h)}{\partial(net_h)} = o_h(1 - o_h) \tag{65}$$

$$\frac{\partial(net_h)}{\partial(w_i)} = x_i \tag{66}$$

Then the total error is,

$$\frac{\partial E}{\partial(w_i)} = \sum_k \delta_k.w_{kh}.o_h(1-o_h)x_i \tag{67}$$

$$= o_h(1-o_h)(\sum_k \delta_k.w_{kh})x_i \tag{68}$$

$$\tag{69}$$

Now define δ_h as

$$\delta_h = o_h(1-o_h)(\textstyle\sum_k \delta_k.w_{kh})$$

we get

$$\tfrac{\partial E}{\partial(w_i)} = \delta_h \times x_i$$

Now at this step, we can update the weight of w_{ji},

$$\triangle w_{ji} = \eta\delta_j x_{ji}$$

where η is the learning rate.

14.5 More Backpropagation

Backpropagation will converge to a local but not necessarily global, error minimum. In practice, when run enough times it works well enough. Can add a weight momentum α if needed,

$$\triangle w_{ji}(n) = \eta\delta_j x_{ji} + \alpha\triangle w_{ji}(n-1)$$

Neural networks tend to overfit so often times regularisation of the network is needed. We can add a term to the error that increases with the magnitude of the weight vector,

$$E(\overline{w}) = \tfrac{1}{2}\textstyle\sum_{d\in D}\sum_{k\in outputs}(t_{kd-o_kd})^2 + \gamma\sum_{i,j} w_{ji}^2$$

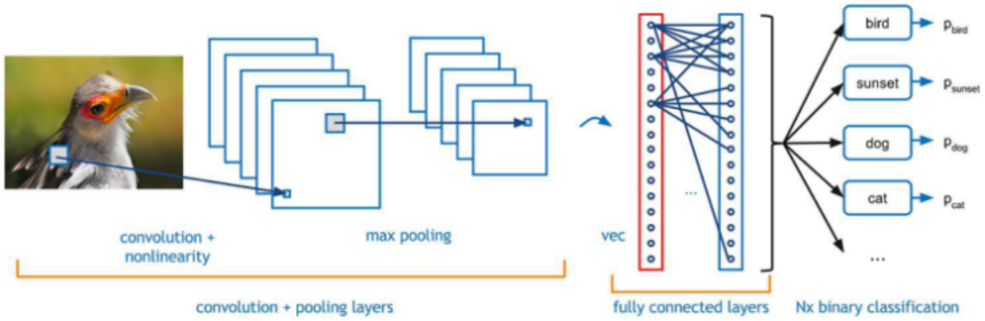
For classification tasks, MSE does not work so well. If we instead take the output $o(\overline{x})$ as the probability of class $\overline{x}$ being 1, the preferred loss function is the **cross entropy**,

$$-\textstyle\sum_{d\in D} t_d \log o_d + (1-t_d)\log(1- \ o_d)$$

where $t_d \in \{0,1\}$ is the class label for training example d and o_d is the output of the sigmoid unit. o_d is interpreted as the probability of the class of training example d being 1. To train sigmoid units for classification using this setup, one can use gradient descent and backpropagation to yield to maximum likelihood solution.

14.6 Convolutional Neural Networks (CNNs)

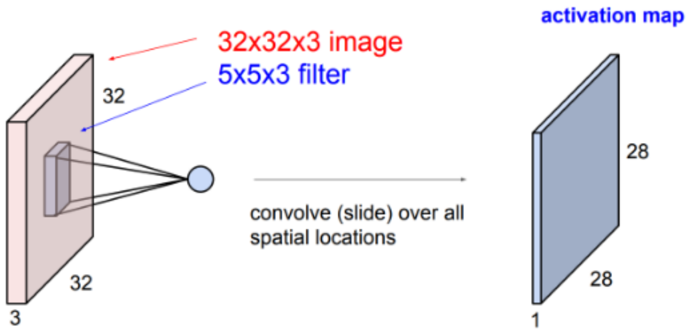
CNN architecture assumes inputs are images so that we have local features. Encode certain properties in the architecture that make the forward pass more efficient unlike neural networks which do not scale well with dimensions. CNNs arranges its neurons in three dimensions (wdith, height and depth) so that parameters are shared such that the number of parameters required by the network are minimised.



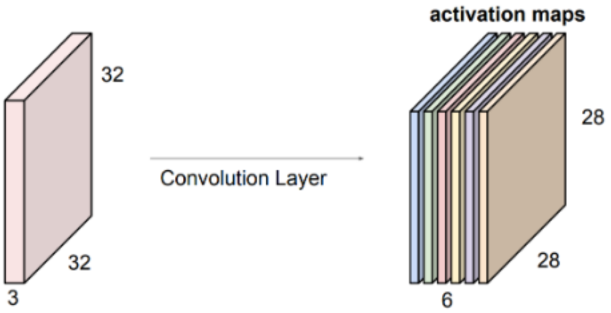
:

14.7 Convolutional Layer

The output of a convolutional layer's parameters consist of a set of learnable filters. Every filter is small spatially, but extends through the depth of the input volume. During the forward pass, each filter is slid across the width and height of the input volume, producing a 2-dimensional activation map of that filter.

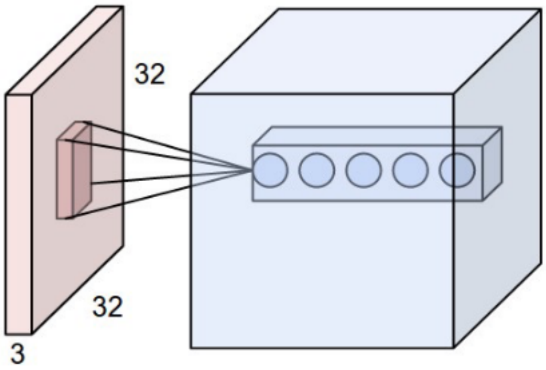


Network will learn filters (via backpropagation) that activate when they see some specific type of feature at some spatial position in the input. Stacking these activation maps for all filters along the depth dimension forms the full output volume.

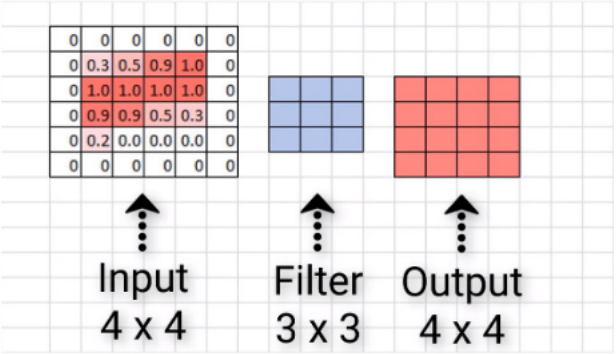


Three hyperparameters control the size of the output volume,

- **Depth:** controls the number of neurons in the convolutional layer that connect to the same region of the input volume.



- **Stride:** is the distance that the filter is moved in spatial dimensions.
- **Zero-padding:** is the padding of the input with zeros spatially on the border of the input volume.



We can compute the spatial size of the output volume as,

$$\frac{W-F+2P}{S+1}$$

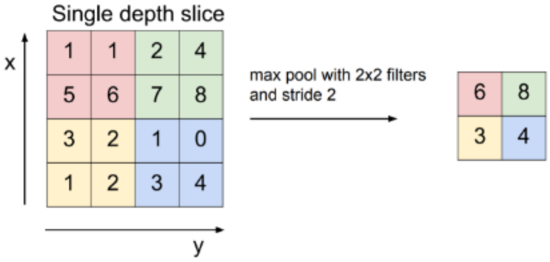
where W is the input volume size, F is the receptive field size of convolutional layer neurons, S is the stride and P is the amount of zero padding used on the border. If this number is not an integer, then the strides are set incorrectly.

The convolutional layer of a CNN also gives many useful properties,

- **Local connectivity:** Each neuron connects to a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron. The extent of the connectivity along the depth axis is always equal to the depth of the input volume.
- **Parameter Sharing:** Parameter sharing scheme used in convolutional layers to control the number of parameters. If one patch feature is useful to compute at some spatial position (x, y), then it should also be useful to compute at a different position $(x2, y2)$.

14.8 Pooling Layer

The function of a pooling layer is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network; hence controlling overfitting. Independently resize every depth slice of the input spatially using the max pooling operation. The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2, which downsamples every depth slice in the input by 2 along both width and height, discarding 75% of activations.



1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

If the previous layer is $J \times K$, and max pooling is applied with width F and stride S, the size of output will be,

$$\frac{1+(J-F)}{S} \times \frac{1+(K-F)}{S}$$

14.9 ReLU Layer

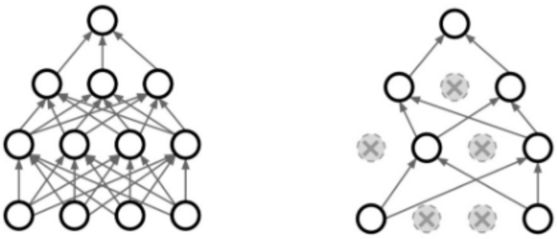
Although ReLU (**Re**ctified **L**inear **U**nit) is considered a layer, it is really an activation function: $f(x) = max(0, x)$. This is favoured over traditional activation functions as it is computationally inexpensive and accelerates convergence.

14.10 Fully Connective Layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

14.11 Dropout Layer

Dropout is a simple method to reduce overfitting. In each forward pass, randomly set some neurons to zero. The probability of dropping is set from some hyperparameter. This makes the training process noisy and forces nodes with a layer to probabilistically take on more or less responsibility for the inputs. Prevents co-adaptation of features and simulates a sparse activation



14.12 Output Layer

The output layer produces the probability of each class given the input image. This is the final layer containing the same number of neurons as the number of classes in the dataset. This layer passes through a Softmax function to normalise the outputs to a sum of one,

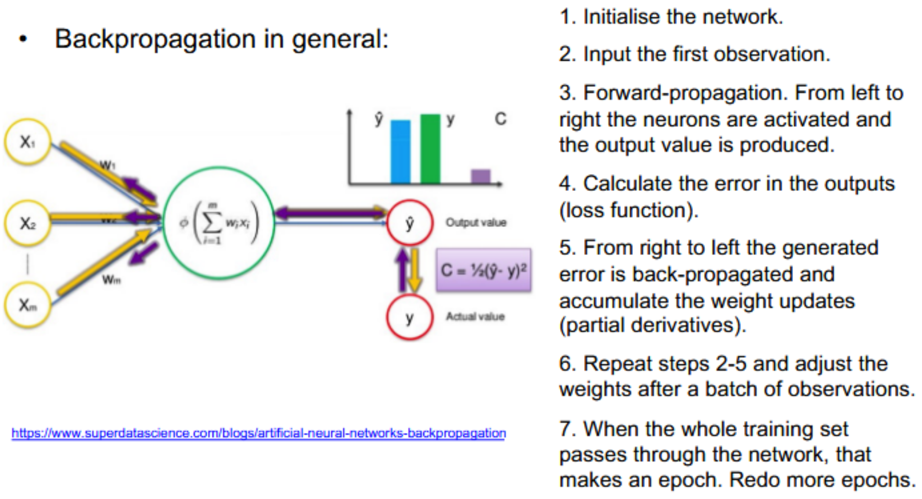
$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

14.13 Loss function

A loss function is used to compute the models prediction accuracy from the outputs. Most commonly used is the cross entropy loss function,

$$H(y, \hat{y}) = \sum_i y_i \log \frac{1}{\hat{y}_i} = -\sum_i y_i \log \hat{y}_i$$

The training objective is to minimise this loss using backpropagation to train the CNN model.



- Backpropagation in general:

https://www.superdatascience.com/blogs/artificial-neural-networks-backpropagation

1. Initialise the network.
2. Input the first observation.
3. Forward-propagation. From left to right the neurons are activated and the output value is produced.
4. Calculate the error in the outputs (loss function).
5. From right to left the generated error is back-propagated and accumulate the weight updates (partial derivatives).
6. Repeat steps 2-5 and adjust the weights after a batch of observations.
7. When the whole training set passes through the network, that makes an epoch. Redo more epochs.

Data is preprocessed and augmented to avoid overfitting and improve performance. The data is also balanced so that similar number of training images for different classes are not biased by one class. This is achieved with random sampling during each epoch of training. Assign weights in the loss function,

$$H(y, \hat{y}) = \sum_i \propto_i y_i \log \hat{y}_i \tag{70}$$

$$\propto_c = \frac{median\_freq}{freq(c)} \tag{71}$$

Finally, during forward passes of the network throughout the layers give prediction output of the input data.

15 Unsupervised Learning

Unsupervised learning is used when classes are initially unknown and need to be discovered with their definitions from data. Most real world data is unlabelled and getting humans to label data is difficult, error-prone and time-consuming.

15.1 Clustering

Finding groups of items that are similar. The class of any data instance is unknown; unsupervised learning. Partitioning methods usually require specification of the number of clusters, then try to construct the clusters and fit objects to them. Hierarchical methods involve algorithms that are either agglomerative (bottom-up) or divisive (top down).

Let $= \{x_1, \cdots, x_m\}$ be a set of instances and let $C = (C_1, \cdots, C_k)$ be a partition of m elements into k subsets. Each subset is called a cluster and each C is a clustering. The goal is to find a partition of m elements into homogeneous and well-separated clusters. Elements from the same cluster should have high similarity while elements from different clusters should have low similarity. Homogeneity and separation are usually defined by a distance metric. Bad clusterings are clusters that are close to each other but points with the clusters are far away form each other. Good clusters are clusters that are far from each other with points that are close to each other.

15.2 k-means Clustering

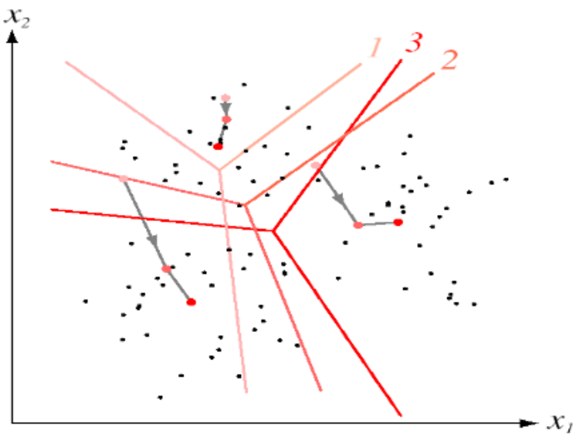
Set value for k, the number of clusters. Choose points for the centres (usually the means) of k points to form k. The procedure is as follows,

- assign each instance x to the closest of the k points to form k clusters
- re-assign the k points to be the means of each of the k clusters
- repeat previous steps until convergence to a reasonably stable clustering

The goal is to find a partition P for which the error (distance) function is minimum,

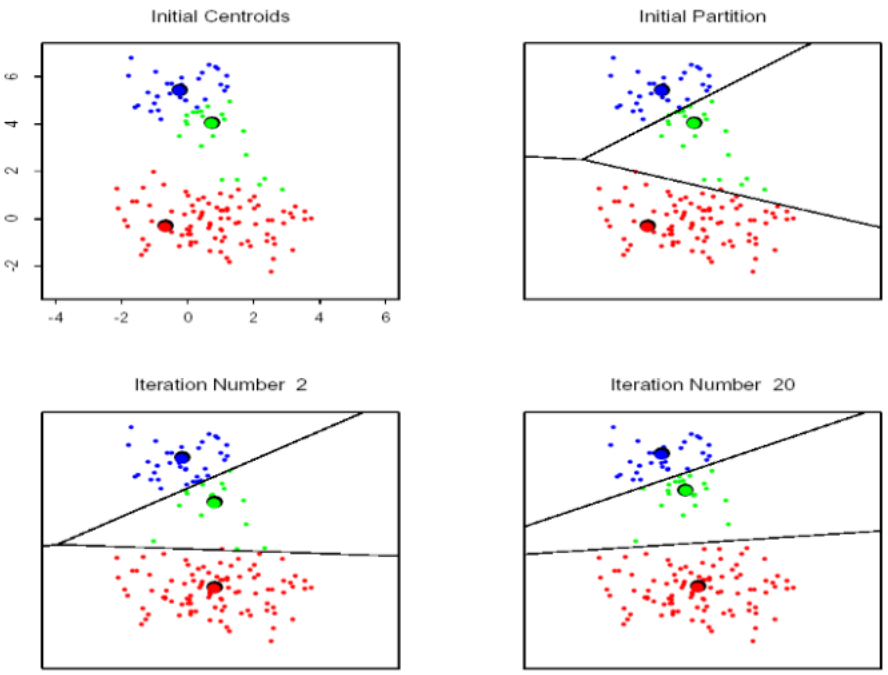
$$E_p = \sum_{i=1}^{m} d(x_i, c_{pi})$$

where P_i is the cluster assigned to element i, c_j is the centroid (mean or weighted average of cluster) of cluster j, $d(v_1, v_2)$ is the Euclidean distance between feature vectors v_1 and v_2.

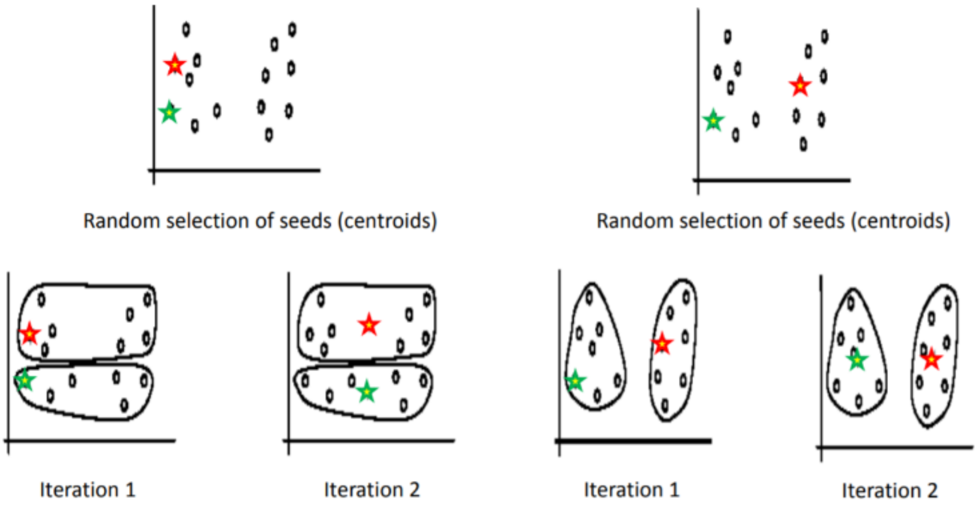


The algorithm terminates when the error is minimised; when the distance between the centroid and each point in the cluster is minimised. In essence, the k-means algorithm can be viewed as moving these centroids in the space where the clusters exists until it find a clustering deemed to minimise this error. The quality of the solution will depend on,

- the distribution of the points in the dataset
- the choice of k
- outliers
- the choice of the location to initialise the centroid



$k-$means clustering is, as one can expect, very sensitive to outliers. A way to deal with outliers is to remove some data points that are much further away from the centroids than other data points or perform random sampling (choosing small subset of data points so that the chance of selecting an outlier is low).


Random selection of seeds (centroids)
Random selection of seeds (centroids)
Iteration 1
Iteration 2
Iteration 1
Iteration 2

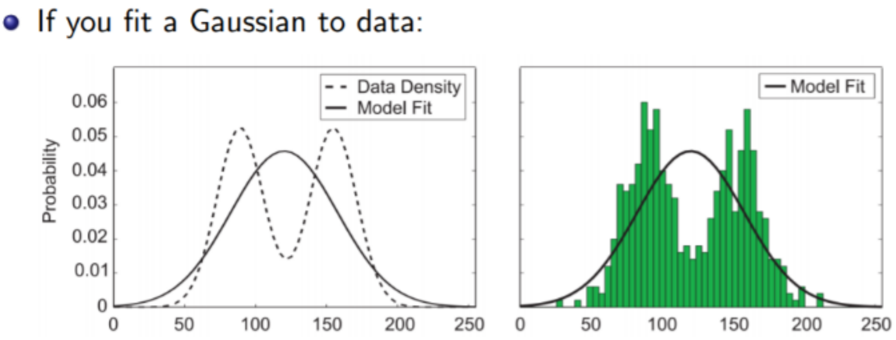
15.3 Expected Maximisation

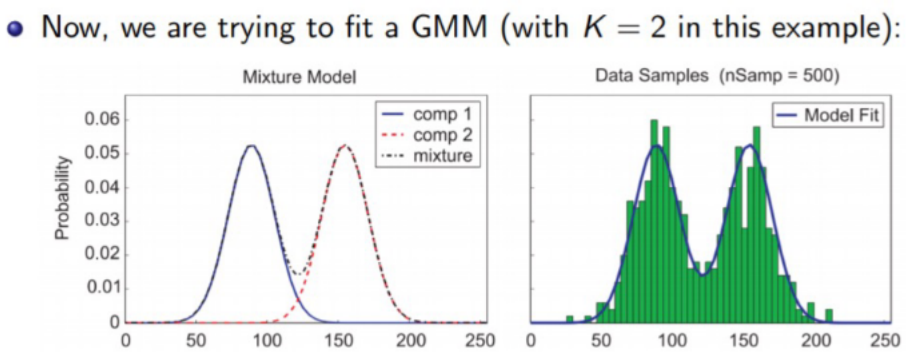
Expected Maximisation is a probabilistic technique used in conjunction with $k-$means to form a "mixture". Each instance x is generated by choosing one of the k Gaussians with uniform probability then generating an instance at random according to that Gaussian. These are called finite mixtures because there is only a finite number of generating distributions being represented.

The algorithm is given,

- Instances from X generated by mixture of k Gaussian distributions
- Unknown means of the K Gaussians
- Assuming identity of covariance matrix

We do not know which instance x_i was generated by which Gaussian. Our goal is to determine the maximum likelihood estimates for the means $\mu_i, \cdots, \mu_k$.


● If you fit a Gaussian to data:



- Now, we are trying to fit a GMM (with $K = 2$ in this example):

For each instance y_i where $k = 2$, $y_i =< x_i, z_{i1}, z_{i2}$,

- z_{ij} is 1 if generated by the jth Gaussian, otherwise zero in k means.
- x_i is observable from instance set $x_1, x_2, \cdots, x_m$
- z_{ij} is unobservable

The algorithm has two steps,

- E step: Calculate the expected value $E[z_{ij}]$ of each hidden variable z_{ij}, assuming current hypothesis $h =< \mu_1, \mu_2 >$ holds,

$$E[z_{ij}] = \frac{p(x = x_i | \mu = u_j)}{\sum_{n=1}^{2} p(x = x_i | \mu = \mu_n)} \tag{72}$$

$$= \frac{\exp(-\frac{1}{2\sigma^2}(x_i - \mu_j)^2)}{\sum_{n-1}^{2} \exp(-\frac{1}{2\sigma^2}(x_i - \mu_n)^2)} \tag{73}$$

  ⇒ Calculate the probabilities for unknown parameters for each instance.

- M step: Calculate new maximum likelihood hypothesis $h' =< \mu'_1, \mu'_2$ assuming value taken by each hidden variable z_{ij} is the expected value $E[z_{ij}]$ calculated before. Replace $h =< \mu_1, \mu_2 >$ by $h' =< \mu'_1, \mu'_2 >$,

$$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E[z_{ij}] x_i}{\sum_{i=1}^{m} E[z_{ij}]}$$

  ⇒ Estimate parameters based on the probabilities.

15.4 Hierarchical Clustering

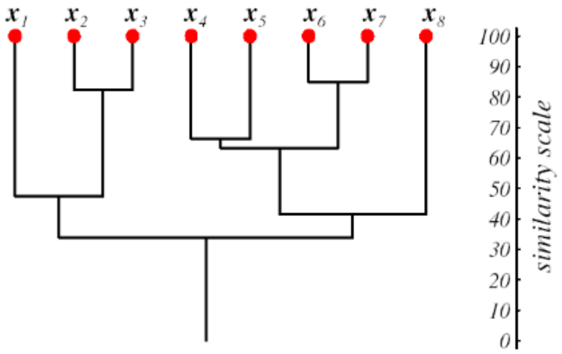
A type of clustering which produce a dendrogram (tree of clustering). Two methods for hierarchical clustering,

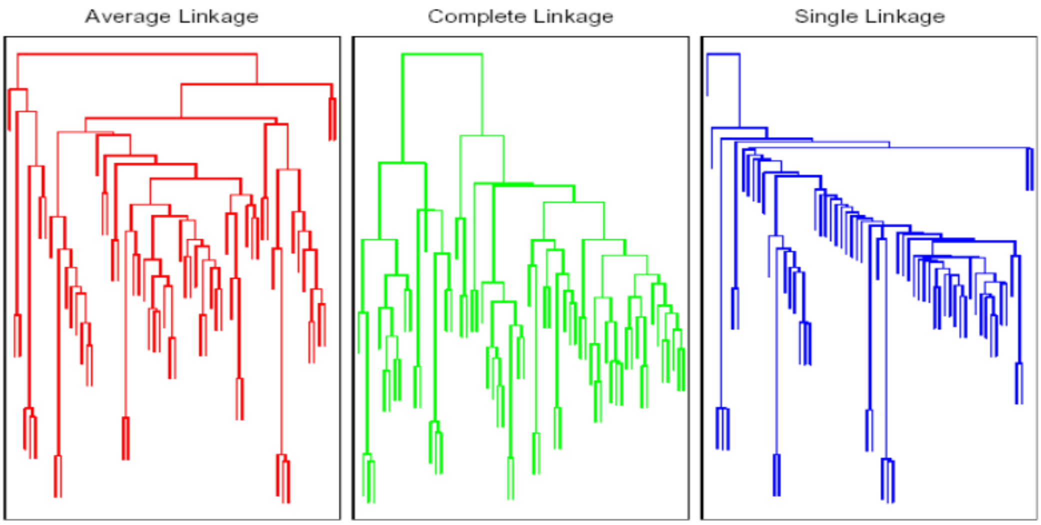
- Bottom up: at each step join the two closest clusters.

- Top down: find two clusters and then proceed recursively for the two subsets.

The algorithm relies on a general updating formula. With different operations and coefficients, many different versions of the algorithm can be used to give variant clustering. The general algorithm is as follows,

- Initialise each data point as its own cluster: clusters $\leftarrow [\{x_1\}, \{x_2\}, \cdots, \{x_n\}]$
- Compute initial pairwise distance matrix between all clusters
- While the number of clusters $> K$:
  - Find the pair of clusters (C_1, C_2) with the smallest entry according to the chosen linkage method
  - Merge C_1 and C_2 into a new cluster $C_3 \leftarrow C_1 \bigcup C_2$
  - Remove C_1 and C_2 from the cluster list
  - Add C_3 to the cluster list
  - Update the distance matrix to include distances from C_3 to all other clusters using chosen linkage method: single linkage; $min(D(x_i, x_j))$ for $x_i \in C_1, x_j \in C_2$, complete linkage; $max(D(x_i, x_j))$, average linkage or centroid linkage
- Return the final K clusters





15.5 Optimal k clusters

How do we determine the optimal amount of clusters?

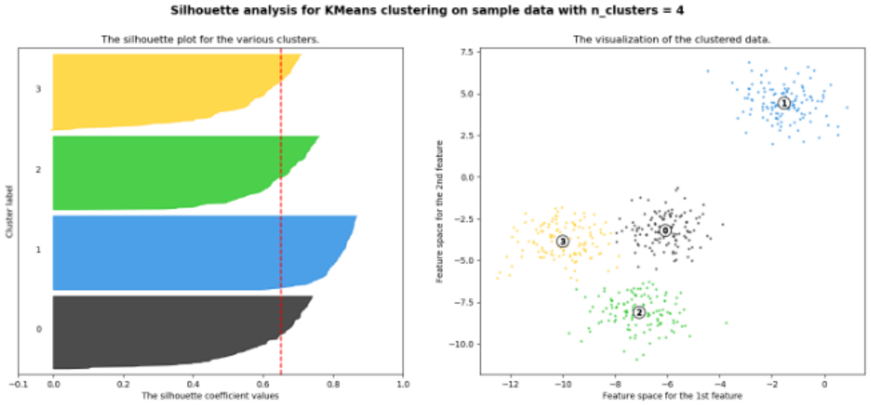
- Elbow method
  - Measure the within-cluster dispersion
  - compute for various k choices and choose the k that doesn't improve the dispersion much.
- Gap statistics
  - Cluster the observed data and compute the corresponding total within cluster variation W_k
  - Generate B reference data sets with random uniform distribution. Cluster each of these reference data sets and compute the corresponding total within-cluster variation W_{kb}
  - Compute the estimated gap statistics as the deviation of the observed W_k value from its expected W_{kb} : $Gap(k) = \frac{1}{B}\sum_b \log(W_{kb}) - \log(W_k)$. Compute also the standard deviation of statistics.
  - Choose the number of clusters as the smallest value of k such that the gap statistics is within one standard deviation of the gap at $k+1$: $Gap(k) \geq Gap(k+1) - s_{k+1}$

15.6 Cluster Quality

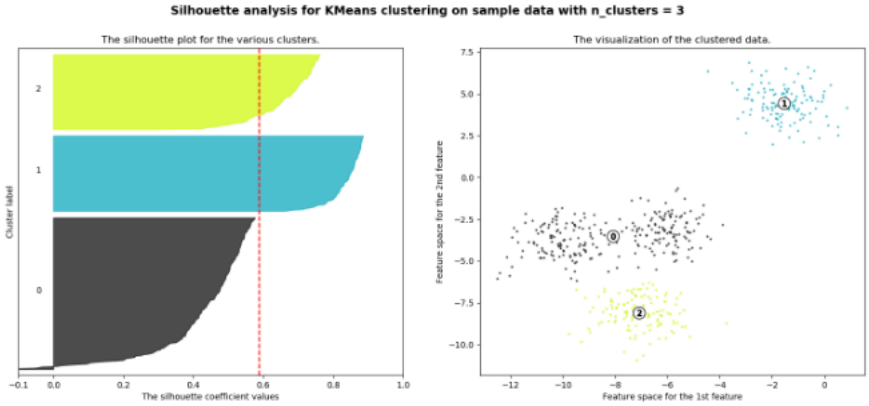
Howe do we determine cluster quality? Silhouette plots. Compare each objects separation from other clusters relative to the homogeneity of its cluster. For each object i define its silhouette width $s(i) \in [-1, 1]$ and compute,

$$s(i) - \frac{b(i)-a(i)}{max(a(i),b(i))}$$

where $a(i)$ is the average dissimilarity between x_i and elements of P_i, $d(i,C)$ is the average dissimilarity of i to elements of some other cluster C and $b(i) = min_C d(i,C)$



```
For n_clusters = 4 The average silhouette_score is : 0.6505186632729437
```



```
For n_clusters = 3 The average silhouette_score is : 0.5882004012129721
```

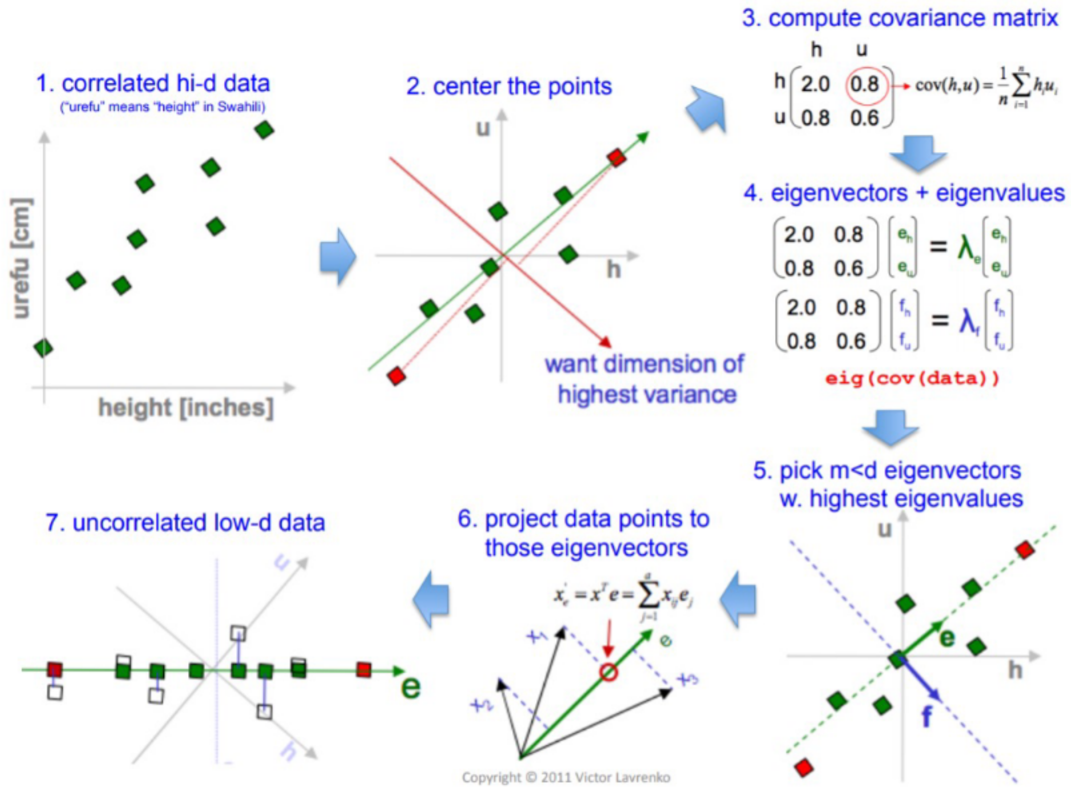
15.7 Principal Component Analysis (PCA)

If many features are related, we may not need all of them in our dataset. We can reduce the dimensionality by finding a new set of features, that preserve information from the original set, to improve the performance of our models. PCA is a method to do this. The key idea is to look for features in a transformed space so that each dimension in the new space captures the most variation in the original data when it is projected onto that dimension. New features should be highly correlated with the original features but not with any other new features.

The basic steps of the algorithm are as follows,

- Take the data as an $m \times n$ matrix X

- ”centre” the data by subtracting the mean of each column
- Construct covariance matrix C from centred matrix
- Compute eigenvector V and eigenvalue matrix S such that $V^{-1}CV = S$ where S is the diagonal $n \times n$ matrix
- Sort columns of S in decreasing order (decreasing variance) and their corresponding eigenvectors
- Remove columns of S and V that the eigenvalues are below some minimum threshold
- Transform the original data using the remaining eigenvectors


1. correlated hi-d data
("urefu" means "height" in Swahili)
urefu [cm]
height [inches]
2. center the points
u
h
want dimension of highest variance
3. compute covariance matrix
$\begin{array}{c|cc} & h & u \\ h & 2.0 & 0.8 \\ u & 0.8 & 0.6 \end{array}$
$\mathrm{cov}(h,u) = \frac{1}{n}\sum_{i=1}^{n} h_i u_i$
4. eigenvectors + eigenvalues
$\begin{pmatrix} 2.0 & 0.8 \\ 0.8 & 0.6 \end{pmatrix}\begin{pmatrix} e_h \\ e_u \end{pmatrix} = \lambda_e \begin{pmatrix} e_h \\ e_u \end{pmatrix}$
$\begin{pmatrix} 2.0 & 0.8 \\ 0.8 & 0.6 \end{pmatrix}\begin{pmatrix} f_h \\ f_u \end{pmatrix} = \lambda_f \begin{pmatrix} f_h \\ f_u \end{pmatrix}$
eig(cov(data))
5. pick m<d eigenvectors w. highest eigenvalues
u
e
h
f
6. project data points to those eigenvectors
$x_e' = x^T e = \sum_{j=1}^{d} x_j e_j$
7. uncorrelated low-d data
e

Copyright © 2011 Victor Lavrenko