Auxiliary notes on Data Science

Daniel Lin; Professor: Dr P. Thomas and Dr B. Bravi

March 10, 2023

The official notes on Data Science are self-contained, but this auxiliary note may give you a more insightful understanding of some concepts. There will also be a cheat sheet of important formulae and notations at the end.

The following table of contents is clickable.

Contents

1 Data Cleaning **2**

2 Linear Regression, ridge, Lasso **3**

3 Logistic Regression **4**

4 Naive Bayes's Method **6**

5 Decision Tree **9**
5.1 Entropy 11
5.2 Regression tree 12
5.3 Random Forest 13

6 Support Vector Machine **13**
6.1 KKT conditions 16
6.2 Soft margin 17
6.3 Kernels 18

7 Neural Networks **20**
7.1 Intro - logistic regression is a neural network 20
7.2 Back-propagation 22
7.2.1 Problems of Optimisation 24
7.3 Regularisation 25
7.4 Convolutional Neural Network 25

8 Unsupervised Learning **27**
8.1 K-means algorithm 28
8.2 Hierarchical clustering 29
8.3 Comparing Clusters 29
8.4 EM Algorithm 31
8.5 Hidden Markov Models 32
8.5.1 Forward-backward algorithm 33
8.5.2 Viterbi Algorithm 34
8.5.3 EM algorithm for HMM 35
8.6 Principal Component Analysis 36
8.7 Graph-based partitioning 37

9 Cheat Sheet **40**
9.1 Basics 40
9.2 Linear Regression 42
9.3 Optimisation 43
9.4 Classification problems 44

9.4.1 k-nearest neighbours . . . 44
9.4.2 Logistic regression . . . 44
9.4.3 Naive Bayes's method . . . 45
9.4.4 Decision Tree . . . 46
9.4.5 Support Vector Machine . . . 46
9.5 Graphs-based learning . . . 47
9.5.1 Graphs . . . 48
9.5.2 graph-based clustering . . . 49

1 Data Cleaning

In real life, most data sets are possibly contaminated by errors like the inconsistency of format, typos, missing data, duplicates etc.

Duplicates are usual when the data is taken from multiple sources. The function `dataFrame.duplicated` allows you to check duplicate rows, and you can specify some columns by `dataFrame.duplicated(['col1', 'col2'])` . For deleting duplicates, `dataFrame.drop_duplicates` () is pretty useful for cleaning duplicates in a pandas data frame. (it has optional argument `keep` . 'first' - keep the first of duplicates, 'last' - keep the last of duplicates, 'False' - delete every duplicate) It is always recommended that you keep a copy of uncleaned data, just in case you dropped important data points. For this reason, most functions in pandas create a new data frame instead of cleaning in place.

If you have categorical data, with only a few possible entries (like payment methods: cash, card, etc.) if one category only has one or two data points, they might be incorrectly recorded, e.g. a typo.

Printing `dataFrame.dtypes` helps you to see whether the formatting of each column is correct. Also, sorting data may also leave those data in abnormal format at the beginning or the end. And you may need to identify Null or NaN in a column using `dataFrame[['col1', 'col2']].isna()` which gives you all rows with NaN in col1 or col2.

There are three types of missing data, caused by different problems. You should deal with them using different techniques.

- **Completely random**: rate of missing data independent of any other factor. You may consider just deleting this whole data entry, but it is hard to guarantee the missing data is not affected by something else.
- **Random by group**: the rate of missing data may be different for each group, e.g. 0.1% in males, 1% in females. There could be more factors affecting the missing data rate.
- **Not at random**: missing data rate may be a function $f(\theta)$ where θ is some other parameter. e.g. males with lower heights are more likely to hide their heights on dating websites. But in reality, θ is difficult to find because it can be the true value of the data entry we are looking at! In the above example, θ is the true height.

Deletion of data entry, or row deletion, is a cheap choice to deal with a missing value. But it only works if the missing data rate is completely random, otherwise, a bias is introduced.
Another cheap option is to use the mean/median of the column, but this reduces the variance of data when there are too many missing data entries. To make this more precise, you can fill in the missing data using the mean/median of all other similar data. (e.g. fill in a missing data with gender = female using the mean of all females)

A more powerful tool is *Multiple imputations*. Instead of filling in the same value for all missing data entries, they can vary with other entries. An example is linear regression. But to lower the computational cost, a stochastic method is often used, e.g. if many entries of y are missing, we pick randomly 50 pairs of (x, y) out of an extremely large data set, find the expected values $\hat{y}$ for the missing data. This can be repeated (usually 5-10 times) and we can use the average of all $\hat{y}$. Of course, you need to check that $\hat{y}$ does not vary too much and that there is no abnormal value of $\hat{y}$.

Merging Data
During the training of models, you may receive a new set of data. Or while collecting the data, you get them from

multiple sources. You may not need all of the new data, so data selection may be required. Compatibility of the data sources is very important. You need to check the number of columns, units, and format. And sometimes, fields with the same name may have different interpretations. e.g. one data source may categorise people with age > 65 as old, but the other source may use the criterion age > 63.

2 Linear Regression, ridge, Lasso

The ordinary least square (OLS) we learnt many times before may have an extremely large variance when two covariates are highly correlated the same, e.g. length of the left leg and right leg. Let us assume that $x_1 \approx kx_2$ for some constant k, then $\beta_1 x_1 + \beta_2 x_2 \approx (k\beta_1 + \beta_2)x_1$. If the true value of the coefficient of x_1 is 10, then there are infinitely many choices s.t. $k\beta_1 + \beta_2 = 10$, and possible β_1, β_2 trace out a line in $\mathbb{R}^2$. Then, basically, the variance of estimators of β will be infinite. One simple solution to keep the variance small is to control $\|\beta\|_2$. For example, in figure 1 where the straight line represents possible values of β_1, β_2, we restrict the choice of β in a small circle around the origin. The smaller the circle, the smaller the variance.

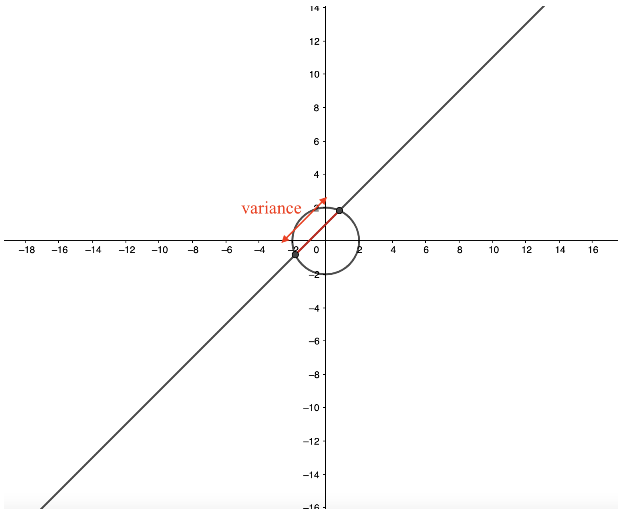


Figure 1: Restricting $\|\beta\|_2$ to control variance

This is achieved by changing the loss function from $\|y - X\boldsymbol{\beta}\|^2$ to

$$L_{\text{Ridge}}(\boldsymbol{\beta}) := \|y - X\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|_2^2$$

(the square is added to ensure convexity and differentiability) This modified version of linear regression is called *ridge regression*. Note here $\lambda > 0$ does not correspond to the radius of the circle in figure 1 because we will minimise this loss function. So actually larger λ will shrink the size of the circle, further press the variance down. Therefore, λ is sometimes called the *penalty parameter*.

You can use similar techniques to least square (differentiate L_{Ridge}, find the minimum point) to derive $\boldsymbol{\beta}$ that minimises L_{Ridge}, this is also derived in the official notes

$$\boldsymbol{\beta}^*_{\text{ridge}} = (X^T X + \lambda I)^{-1} X^T y$$

We have $X^T X + \lambda I \succ 0$ (positive definite). And for reasonably large λ, the condition number of $X^T X + \lambda I$ is well controlled (recall condition number of a matrix A is $\|A\|\|A^{-1}\|$ where the matrix norm is operator 2-norm), so the matrix inversion will not have a large numerical error. But for extremely large λ, $X^T X$ becomes ignorable, and then $\boldsymbol{\beta}^*_{\text{ridge}} \approx (\lambda I)^{-1} X^T y = \frac{1}{\lambda} A^T y \approx 0$. But the true value of $\boldsymbol{\beta}$ is usually not 0, so λ cannot be too large.

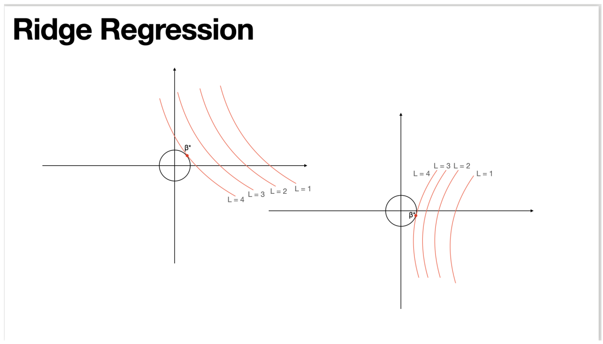
In practice, many values of λ will be tested and the resulting residuals will be compared.

LASSO
LASSO regression uses a loss function

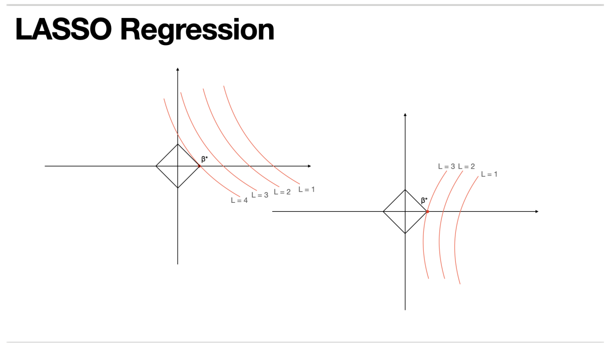
$$L_{\text{LASSO}}(\boldsymbol{\beta}) := \|y - X\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|_1^2$$

the only difference with the ridge regression is that the 1-norm penalty on $\boldsymbol{\beta}$ is used instead of the 2-norm. 1-norm is not differentiable, so no analytical solution of LASSO is given, but this optimisation problem is still convex.

Imagine increasing level curves of loss function on $\mathbb{R}^2$ approaches the circle, representing ridge regression in figure 2a. When the level curve hits the circle, we get the optimal solution. Depending on the shape of level curves, the location of $\boldsymbol{\beta}^*$ is relatively random.



(a) Ridge regression



(b) LASSO regression

Figure 2: Geometrical understanding of two regressions for two different loss functions

In comparison, the result of LASSO regression almost always lands on the four edges of the square. (see figure 2b) In higher dimensions, this means sparsity, i.e. many 0 appearing in the $\boldsymbol{\beta}$. This eliminates unnecessary coefficients.

As mentioned in lecture notes, you may use other norms, i.e.

$$L_{\text{LASSO}}(\boldsymbol{\beta}) := \|y - X\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|_q^2$$

but if $0 \leq q < 1$, the optimisation is no longer convex.

3 Logistic Regression

The linear regression ideas in the previous section only work for continuous data type y, but what if y looks like categories, for example, $y \in \{0, 1\}$?

Such y could be winning/losing of teams, bearing a child or not. One option to deal with these discrete data is kNN as mentioned in lecture notes, but we can still use regression. The key idea is that $P(Y = 1|x)$ is a continuous variable. So we can perform regression on it.

The usual linear regression $P(y = 1) = \beta_0 + \beta_1 x$ does not work, LHS is a probability, a quantity in $[0, 1]$, but RHS takes any value in $\mathbb{R}$. $\log(P(y = 1))$ is a good choice, which falls in $(-\infty, 0]$. To allow positive numbers, clearly, we need a function that takes $[0, 1]$ to $[0, +\infty)$, then log can take $[0, +\infty)$ to $\mathbb{R}$.

One common choice is $x \mapsto x/(1 - x)$. Indeed when $x \to 0$, $x/(1 - x) \to 0$ and when $x \to 1$, $x/(1 - x) \to +\infty$. Composition of $x \mapsto x/(1 - x)$ and log takes $[0, 1]$ to $\mathbb{R}$. See the plot of the functions in figure 3.

The inverse of $x \mapsto \log(x/(1 - x))$ is called the logistic function, with expression

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

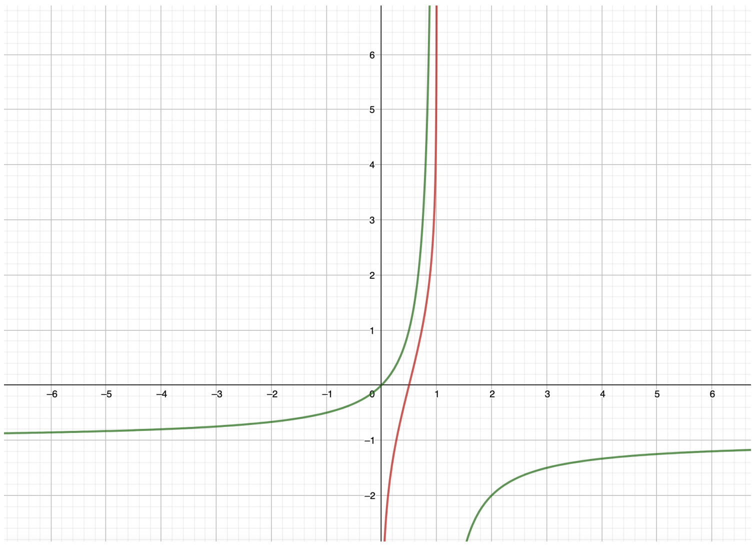


Figure 3: $x \mapsto x/(1-x)$ in green, and composition with $\log$ in red

and it is plotted in figure 4.

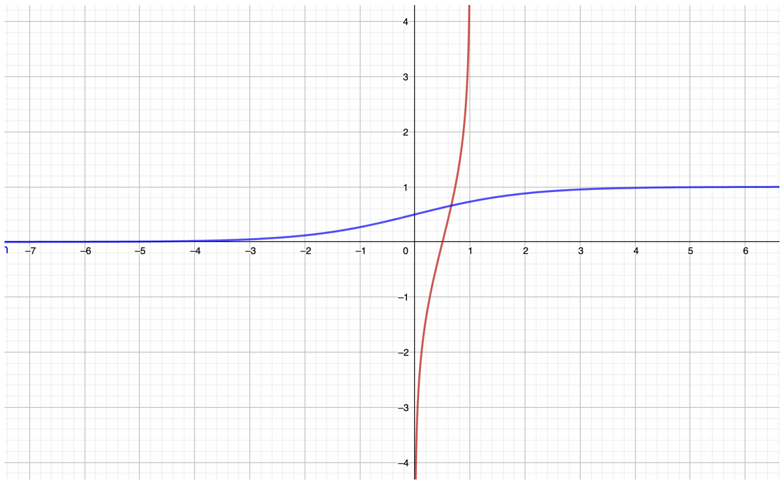


Figure 4: A plot of the logistic function(in blue) and its inverse (in red)

Now, the model is

$$\log\left(\frac{P(y=1)}{1-P(y=1)}\right) = \beta_0 + \beta_1 x$$

or equivalently,

$$P(y=1) = h(\beta_0 + \beta_1 x)$$

If there are more covariates x_i, the general form is

$$P(y=1) = h(\boldsymbol{x}^T\beta) := h_\beta(\boldsymbol{x})$$

After finding an estimate of $P(y=1)$, we can decide a threshold τ s.t. if $P(y=1) > \tau$, we let $\hat{y} = 1$, and otherwise, $\hat{y} = 0$.

Suppose observations $\boldsymbol{x}, y$ are already obtained, we need to find an estimate of $P(y=1)$. We find coefficients β that maximises the likelihood $l(\boldsymbol{x}|\boldsymbol{\beta}, y) := P(y|\boldsymbol{x}, \boldsymbol{\beta})$ Unfortunately, no analytical solution exists for this maximisation problem, but there is a normal equation

$$X^T(\boldsymbol{y} - h(X^T\boldsymbol{\beta})) = 0$$

which is pretty similar to that of linear regression: $X^T(\boldsymbol{y} - X\boldsymbol{\beta}) = 0$. This equation means that $h(X^T\boldsymbol{\beta})$ is an orthogonal projection from $\boldsymbol{y}$ to space spanned by columns of X. Then there are a bunch of optimisation algorithms to numerically find the MLE estimator $\boldsymbol{\beta}_{\text{log}}$.

For models predicting y values that are discrete, the performance is measured by contingency tables. The table for the case $y \in \{0, 1\}$ is given below

	$y = 1$	$y = 0$
$\hat{y} = 1$	TP(true positive)	FP(false positive)
$\hat{y} = 0$	FN(false negative)	TN(true negative)

The table is larger if there are more classes. Such tables are also used for the chi-squared test.

Some commonly used measures using entries of the contingency table are
Precision:

$$\frac{\text{TP}}{\text{TP} + \text{FP}}$$

it measures how confident we are when the model predicts positive (i.e. $\hat{y} = 1$).
Recall

$$\frac{\text{TP}}{\text{TP} + \text{FN}}$$

it measures the proportion of positive values identified by the model.
Accuracy

$$\frac{\text{TP} + \text{TN}}{\text{N}_{\text{validation}}}$$

where $\text{N}_{\text{validation}} = \text{TP} + \text{TN} + \text{FP} + \text{FN}$. It measures the proportion of correct outcomes of the model.
F-score

$$2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

it contains two pieces of information: how close precision and recall are to 1, and how close precision and recall are. See the figure 5 for a 3-D plot of the F-score function.

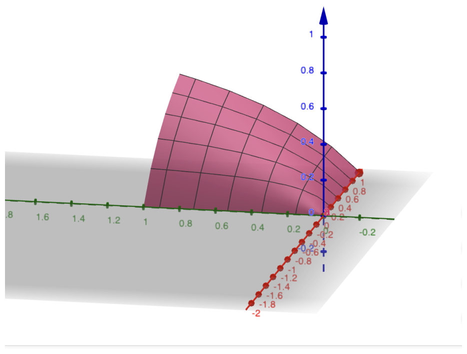


Figure 5: Plot of F-score function, with the F-score on z-axis and precision, recall on x, y axes

4 Naive Bayes's Method

Multinomial logistic regression is a useful generalisation of logistic regression, but there is a Bayesian method for classification problems where there are more classes.

First, recall the Bayesian theorem. Suppose we want to estimate a conditional probability $P(y|x)$, when the problem is easy, like $y \in \{0,1\}$ indicates whether the person got a disease or not, $x \in \{0,1\}$ is whether the person is coughing or not. Simply find a group of samples and use the frequentist's view, e.g.

$$P(y=1|x=1) \approx \frac{\text{num}(y^{(i)}=1,\, x^{(i)}=1)}{\text{num}(x^{(i)}=1)}$$

where $(x^{(i)}, y^{(i)})$ is the ith collected data, and num() is a counting function that counts the occurrence of events among all data points (i.e. sum along index i). But what happens if more symptoms are added to the data x, saying x now includes coughing, fever, diarrhoea, fatigue, etc.. You have to find a group of people for each combination of these symptoms, for example, people who have diarrhoea, are coughing, but do not have a fever and do not feel fatigued. A more cumbersome case is when x is the body temperature, it is a continuous variable! Frequentist's approach does not work. Here comes Bayes's theorem:

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

$P(x|y)$ is easier to estimate since y belongs to a finite set. (set of classes) In the example above, you just have to estimate $P(x|0)$ (symptoms of people not having the disease) and estimate $P(x|1)$. (symptoms of people having the disease) By the law of total probability, $P(x) = \sum_{y' \in \mathcal{C}_q} P(x|y')P(y')$, so the formulae becomes

$$P(y|x) = \frac{P(x|y)P(y)}{\sum_{y' \in \mathcal{C}_q} P(x|y')P(y')}$$

The denominator is like a normaliser, it normalises $P(x|y)P(y)$ among all classes. All probabilities on RHS can be estimated by the frequentist's method. e.g. for class c_k

$$P(y = c_k) = \frac{\text{num}(y^{(i)} = c_k)}{N}$$

where N is the total number of data points collected.

When data $\boldsymbol{x}$ is multi-dimensional (say dimension p), possible combinations of $\boldsymbol{x}$ become huge. Even if each x_i is binary, we still have 2^p combinations, these are difficult to enumerate and count. Therefore, the independence assumption is used. This may not always be true in reality, so the method is called *naive* Bayes's method. We assume x_i are independent of each other conditioned on y, so that

$$P(\boldsymbol{x}|y) = P(x_1, \cdots, x_p|y) = \prod_{j=1}^{p} P(x_j \,|\, y)$$

now for each class $y = c_k$, each term in the above formulae can be estimated using Frequentist's method

$$P(x_j \,|\, c_k) \approx \frac{\text{num}(x_j^{(i)} = x_j,\, y^{(i)} = c_k)}{\text{num}(y^{(i)} = c_k)}$$

where $(\boldsymbol{x}^{(i)}, y^{(i)})$ is the ith collected data.

Another possible problem is that if $P(x_j \,|\, c_k)$ is tiny, like 0.01, then it is highly likely that $N(X_j = x_j) = 0$(i.e. this event may not occur), especially in small samples. But the above formulae estimate it as 0, which means the event is impossible. This is NOT a plausible estimation. So we use the Laplace smoothing: we increase the count of every x_j by 1, so now

$$P(X_j = x_j \,|\, c_k) \approx \frac{\text{num}(x_j^{(i)} = x_j,\, y^{(i)} = c_k) + 1}{\text{num}(y^{(i)} = c_k) + A}$$

where A is the number of possible values of X_j. The general Laplace smoothing is

$$P(X_j = x_j \,|\, c_k) \approx \frac{\text{num}(x_j^{(i)} = x_j,\, y^{(i)} = c_k) + \lambda}{\text{num}(y^{(i)} = c_k) + A\lambda}$$

Laplace smoothing can also be used on $P(y = c_k)$ is too small for some classes:

$$P(y = c_k) \approx \frac{\text{num}(y^{(i)} = c_k) + \lambda}{N + q\lambda}$$

where q is the total number of classes and λ is the smoothing constant, usually chosen to be 1.

When the variables X_j are continuous, they are usually assumed to have Gaussian distribution, i.e. so $X_j|y = c_k \sim N(\mu_{j,k}, \sigma^2_{j,k})$ where

$$\mu_{j,k} = \frac{1}{N_k} \sum_{y^{(i)}=c_k} x_j^{(i)} \quad N_k \text{ is total number of samples in class } c_k$$

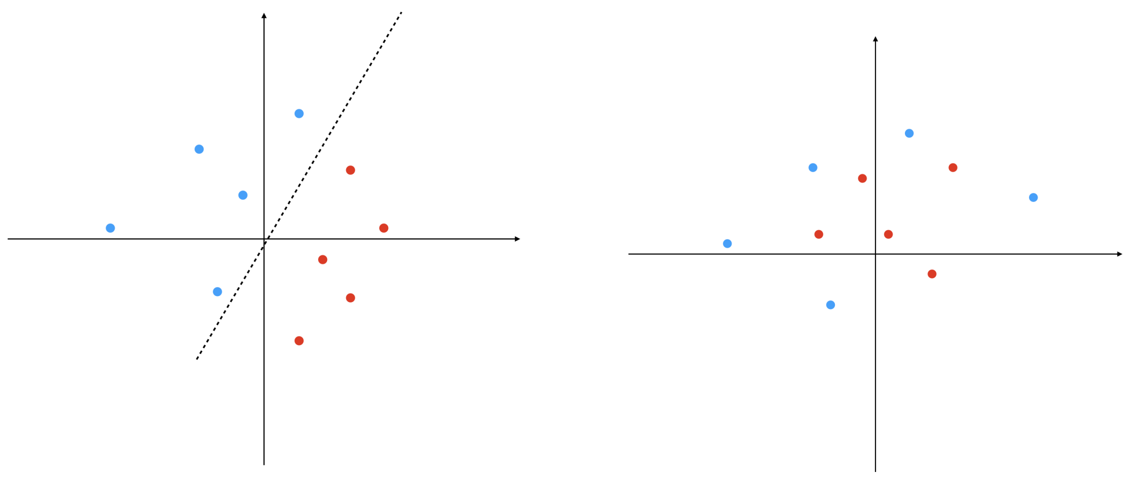
and $\sigma^2_{j,k}$ is the unbiased estimator of variance,

$$\sigma^2_{j,k} = \frac{1}{N_k - 1} \sum_{y^{(i)}=c_k} x_j^{(i)}(x_j^{(i)} - \mu_{j,k})^2$$

So after all the probabilities $P(Y = c_k), P(X_j = x_j \,|\, Y = c_k)$ are estimated using the training set, given a new data point, we can use Bayes's formulae to estimate q probabilities $\boldsymbol{\pi}^{(NB)} = (\pi_1^{(NB)}, \pi_2^{(NB)}, \cdots, \pi_q^{(NB)})$ representing the probability of the data belonging to each class. Then $\hat{y}^{(NB)} := \text{argmax}_k\{\pi_k^{(NB)}\}$, i.e. choose the class with the highest probability.

5 Decision Tree

For some well-organised data, like the one shown in figure 6a, classification is easier. We can easily find a straight line that separates the two data sets.

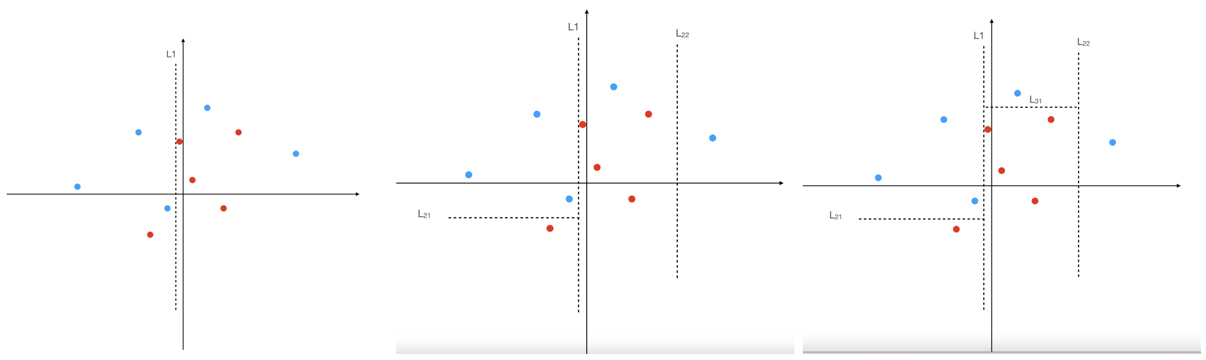


(a) Linear separation applicable **(b) Linear separation not applicable**

Figure 6: Linear Separation

But for data like figure 6b, there is no simple curve that separates the two classes.

Instead of finding a curvy, weird line to separate the classes, we try to separate the space several times using straight lines. (See figure 7)



(a) First layer **(b) Second layer** **(c) Third layer**

Figure 7: Decision tree demonstration on a data set

First, line L_1 is drawn to roughly separate the two groups. There is only one red point to the left, and it can be separated from the blue points with one line L_{21}. To the right of L_1, there is no one-step solution to separate the points, so we need another step. First, draw L_{22} to isolate the blue point at the far right, and then the other blue point left can be separated using L_{31} (in the third layer).

This process can be demonstrated using a tree, see figure 8. Each point represents a region and is called *node*. The number of layers is the *depth* of the tree. Each node has either no child or two children (left child and right child), where child means the nodes in the next layer connected to this node. For the children, the node in the above layer connecting them is the *parent node*. A node without any child is called a *leaf node*.

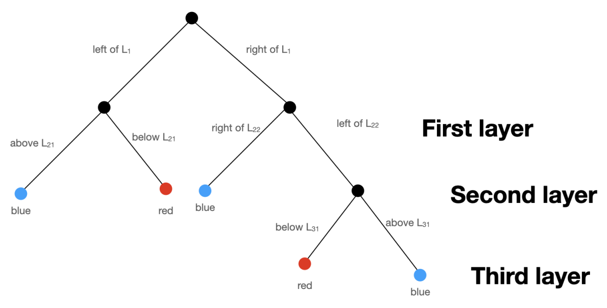


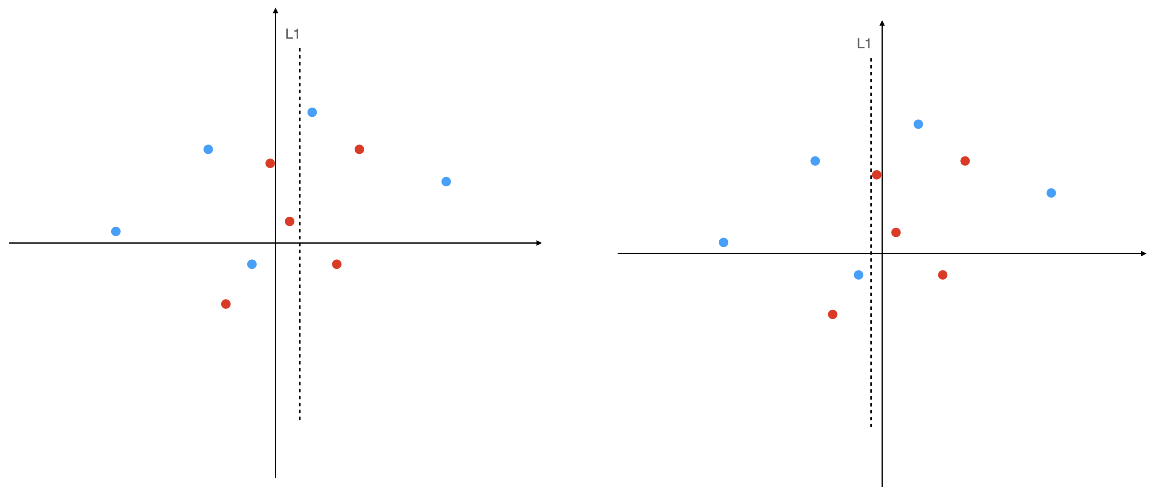
Figure 8: The tree, Decision Tree

There are many other choices to separate these points, in fact infinitely many! L_1 can be any of the following

$$\{(x_1, x_2) : x_1 = s\}, \quad \{(x_1, x_2) : x_2 = s\}$$

where s is any real number.

But clearly, some choices are better than others, let us consider two choices of L_1 (see figure 9)



(a) Choice 1 **(b) Choice 2**

Figure 9: Different choices of the initial separation

For choice 1, there are 3 blue, 3 red on the left side and 2 blue, and 2 red on the right side. If we stop here and estimate using this separation, there is only 50% accuracy, which is not different from that before separation. (In information theory, we say the entropy did not change, or information gain is 0) But for choice 2, at least points to the left are more likely to be blue, and points to the right are more likely to be red. Suppose we stop here, and if a new point is on the left, we say it is blue, if a new point is on the right, we estimate it to be red. (this is majority rule) Let B represent blue and R represent red, the accuracy of estimation is approximately

$$P(\text{left})P(B \,|\, \text{left}) + P(\text{right})P(R \,|\, \text{right}) = \frac{1}{2}\frac{3}{4} + \frac{1}{2}\frac{4}{6} = \frac{17}{24} > \frac{1}{2}$$

so choice 2 is better than choice 1 in terms of accuracy.

Note: the L_{21} separation in figure 7b is perfect, all points above the line are blue, and all points below are red. Such regions are called *pure* regions. And a good separation should create more pure regions, or close to being a pure region (e.g. 6 blue vs 1 red).

Mathematically, the impurity of region R_α, $\boldsymbol{\pi}(R_\alpha)$, is defined by

$$\boldsymbol{\pi}(R_\alpha) := \begin{pmatrix} \pi_1(R_\alpha) \\ \pi_2(R_\alpha) \\ \vdots \\ \pi_Q(R_\alpha) \end{pmatrix} \quad \text{where } \pi_q(R_\alpha) := \frac{\sum_{i=1}^N I(\boldsymbol{x}^{(i)} \in R_\alpha,\, y^{(i)} = c_q)}{\sum_{i=1}^N I(\boldsymbol{x}^{(i)} \in R_\alpha)}$$

each $\pi_q(R_\alpha)$ represents estimated proportion of class c_q in this region R_α. For pure region, $\boldsymbol{\pi}(R_\alpha)$ only has one 1 and 0 in all other entries.

We will decide on better quantitative measures of the quality of separation later. These measures will be optimised to find the first separation, and then if any region is not pure, a new line is searched by optimisation again. This is repeated until a reasonable purity is achieved in all regions. Such an algorithm is called *Greedy algorithm*.

5.1 Entropy

A common technique to measure the quality of separation for a decision tree is entropy. This quantifies how much information would be given on average by an event. For events with lower probabilities, like the moon has been bombarded, you will be surprised and gain a lot of information from this event. In contrast, if you see the sun rising from the east, there is no extra information for you because this happens every day, it has a high probability. In general, we wish the entropy of events with probability 1 to be 0, and the entropy of events with low probabilities to be high.

There are many decreasing functions, but we wish our information function $h : \mathcal{F} \to \mathbb{R}^+$ (negative information does not make sense) to satisfy $h(E \cap F) = h(E) + h(F)$ where E, F are two independent events. i.e. we wish the information of E, F happening together is exactly the sum of information given by E and information given by F. Note $P(E \cap F) = P(E)P(F)$, so clearly, h should involve probability and logarithm. Note information decreases with probability, so let's try $h(E) := \log_2(1/P(E)) = -\log_2(P(E))$ (other base also works, 2 is the conventional choice of information theory) Note $1/P(E) \in [1, \infty)$, so $h(E) \geq 0$ and $h(E) = 0$ iff $P(E) = 1$.

The entropy of a random variable is defined as the expected value of information h, i.e. for discrete variables

$$\text{entropy}(X) = E_X(h(X)) = -\sum_x P(x) \log_2(P(x))$$

if the number of possible values of X is fixed, this quantity is maximised when all probabilities are equal. For example, if $X \in \{0, 1\}$, the entropy is maximised when $P(X = 0) = P(X = 1) = 0.5$. Figure 10 is a plot of entropy against $P(X = 0), P(X = 1)$.

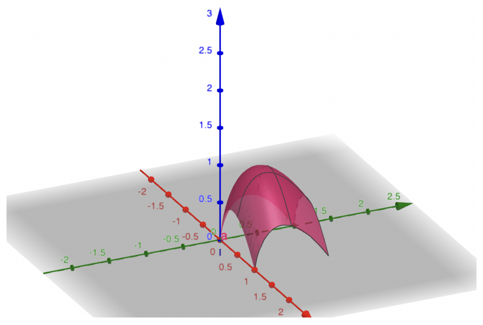


Figure 10: Demonstration of entropy on two values

In the case of the decision tree, the impurity distribution $\boldsymbol{\pi}(R_\alpha)$ (recall that is the proportion of each class in the region R_α, which are basically estimated probabilities of each class) is used,

$$\mathrm{CE}[\boldsymbol{\pi}(R_\alpha), \boldsymbol{\pi}(R_\alpha)] := -\sum_{q=1}^{Q} \pi_q(R_\alpha) \log\left(\pi_q(R_\alpha)\right)$$

What if we already obtained a model distribution $\hat{\boldsymbol{\pi}}(R_\alpha)$? It may not be accurate, so if entropy is calculated by

$$\mathrm{CE}[\hat{\boldsymbol{\pi}}(R_\alpha), \hat{\boldsymbol{\pi}}(R_\alpha)] := -\sum_{q=1}^{Q} \hat{\pi}_q(R_\alpha) \log\left(\hat{\pi}_q(R_\alpha)\right)$$

the error (in estimating entropy) can be quite high. Instead, we calculate the so-called cross-entropy between collected data and model distribution:

$$\mathrm{CE}[\boldsymbol{\pi}(R_\alpha), \hat{\boldsymbol{\pi}}(R_\alpha)] := -\sum_{q=1}^{Q} \pi_q(R_\alpha) \log\left(\hat{\pi}_q(R_\alpha)\right)$$

this can be effectively used to measure the model's quality. Warning: this cross-entropy function $\mathrm{CE}[\cdot,\cdot]$ is not commutative, so $\mathrm{CE}[P,Q] \neq \mathrm{CE}[Q,P]$.

So in each step of the greedy algorithm for constructing the decision tree, the entropy/Gini index is minimised. Note splitting threshold s can be any real number, there are infinitely many choices. Optimisation techniques can be applied, but there is a simpler solution. For each x_i, simply let s loop through all values of x_i in the data set and find the s giving the smallest entropy/Gini index. For example, to split $x_1^{(1)} = 1$, $x_1^{(2)} = 2$, there is no need to try thresholds s between $1, 2$. Because they all have the same splitting effect as setting $s := 1$ and define the left child as $\{x_1^{(i)} \leq s\}$ and right child as $\{x_1^{(i)} > s\}$.

One drawback of using entropy as a loss function is that log is computationally expensive. There is an alternative: Gini's index. We simply remove log in $-\sum_x P(X)\log(P(X))$, obtaining $-\sum P(X)^2$. But now if the probability is concentrated, i.e. $P(x) = 1$ for only one value of x, then $-\sum P(x)^2 = -1$. Negative information does not make sense, so we add 1 in front. This is Gini's index

$$GI(X) = 1 - \sum_x P(X = x)$$

If you plot Gini's index and entropy, they are actually very similar and are both maximised when X follows a uniform distribution.

Once you have decided all layers of the decision tree, you may estimate the label for a new data $\boldsymbol{x}^{\text{in}}$. First find region R_α (i.e. the leaf node in the tree) that contains the point $\boldsymbol{x}^{\text{in}}$, then calculate $\boldsymbol{\pi}(R_\alpha)$ and let $\hat{y} := \mathrm{argmax}_q \pi_q(R_\alpha)$. (this is the majority rule)

5.2 Regression tree

If the output variable y is continuous instead of classification, we can still apply the idea of the decision tree. i.e. we split the space of X into binary parts at each layer, and as long as the purity of a region is low enough (when y is continuous, the impurity can be measured by the squared error), we can define it as a leaf node, stop splitting and estimated the y value simply by the mean of values $y^{(i)}$ with $\boldsymbol{x}^{(i)}$ inside this region.

The key problem is again, finding the best split. This is about minimising the squared error (i.e. maximise variance/entropy reduction) the loss function at each step is defined as

$$\sum_{\boldsymbol{x}^{(i)} \in R_1} (y^{(i)} - \overline{y}_{R_1})^2 + \sum_{\boldsymbol{x}^{(i)} \in R_2} (y^{(i)} - \overline{y}_{R_2})^2$$

where R_1, R_2 are the split regions and $\overline{y}_{R_1}, \overline{y}_{R_2}$ are the mean value of y for points in R_1, R_2 respectively.

The greedy algorithm is used again, at each step best split is found by the above method and the impurity is calculated and assessed. For a new point $\boldsymbol{x}^{\text{in}}$, simply find region R_α containing $\boldsymbol{x}^{\text{in}}$ and use the mean value $\overline{y}_{R_\alpha}$ as $\hat{y}$ (prediction)

Remark: Although this is called regression, no linearity is present. The resulting prediction will be many discrete values like stairs. There is an alternative of using linear tree, i.e. linear regression is done in each region R_α.

5.3 Random Forest

There are several reasons that decision trees are widely used:

- They work for classifications of even messy data. You just need more layers of trees
- Robust: if x_i is not relevant to y, it will not be chosen as splitting criteria in any step. You don't have to identify these x_i and remove them beforehand.
- interpretable: it is not like a black box, each layer of the tree has a clear meaning.

But a drawback is that the decision tree has the potential to over-fit if it is not properly trained. e.g. if class 0 has points $x = 3, 5$ and class 1 has points $4, 6$. It is possible that the tree splits out 4 regions and each region only has 1 point. This problem can be relieved a bit by setting minimal leaf node side. (i.e. stop splitting further if a node contains less than m points where m is a chosen threshold) This makes the error of the decision tree pretty high for samples that are pretty different from the training set.

The idea of random forest is to build B decision trees, where each tree is trained using a bootstrap sample (sampling from the sample X with replacement) and randomly chosen features, i.e. randomly choose $\tilde{p}$ of all p features x_i. Then for a new data point $\boldsymbol{x}^{\text{in}}$, a prediction $\hat{f}_b^{\text{DT}}$ is obtained from each tree and the result is aggregated: For regression, simply take the mean of all results. For classification, use the majority rule. This bootstrapping and aggregating technique is called *bagging*.

Each tree may not be perfect, some may over-fit in one way and some in another way. But the basic idea is that for larger B, these flaws are averaged out by aggregation. Only choosing some features x_i to fit is because fitting all features may cause a high correlation. Some stronger features (strong features for our training sample may not be strong for new data sets) may persist in all decision trees, causing inaccuracy in estimation based on other features.

The number of trees, minimum leaf size, and the number of features bagged are all hyper-parameters that should be decided before training.

The drawbacks of random forests are:

- Computationally heave, especially for large B, p (more trees and more features)
- Interpretability: instead of a single tree, we have many decision trees and it is not clear what each tree represents and how they work together. But yet there is a way to extract the important features: first train in the usual way and estimate the accuracy on the test set. Then permute the feature you want to study in the training set, say x_i, but keep other features unchanged. This means messing up with the ith feature. If it is important for prediction then the error of this perturbed model on the same testing set will be much higher.

6 Support Vector Machine

We mentioned in chapter 5 that some problems are linearly separable (see figure 6a). We can find a hyper-plane $\mathcal{H} := \{\boldsymbol{x} : \boldsymbol{x} \cdot \boldsymbol{w} + b = 0\}$ ($b \in \mathbb{R}$ and the normal direction $\boldsymbol{w} \in \mathbb{R}^n$ are parameters of the plane) s.t. for all points $\boldsymbol{x}$ in one class, $\boldsymbol{x} \cdot \boldsymbol{w} + b > 0$ and for all points $\boldsymbol{x}$ in another class, $\boldsymbol{x} \cdot \boldsymbol{w} + b < 0$. Support vector machines help to find such a hyperplane. Note we are only considering binary separation in this chapter, and for convenience, let the classes be $y \in \{\pm 1\}$ (somehow this indicates whether $\boldsymbol{x} \cdot \boldsymbol{w} + b$ is positive or negative). The separation criteria can now be written compactly as

$$y(\boldsymbol{x} \cdot \boldsymbol{w} + b) > 0$$

Given a training set $\{\boldsymbol{x}^{(i)}\}$ and labels $\{y^{(i)}\}$, there are infinitely many such choices of the hyperplane, and SVM aims to find the most accurate one. The line in figure 11 is not a good choice, it correctly separates the training set(figure on the left) but if a new data x^{new} comes in(figure on the left), visually it seems to belong the blue class, but separation plane H classifies x^{new} as red.

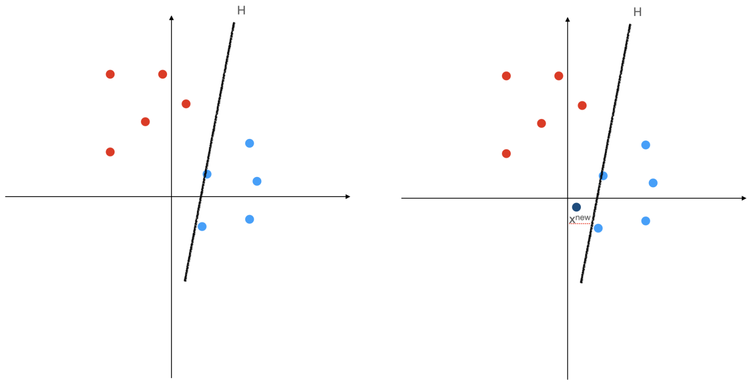


Figure 11: A bad separation line

An accurate separation line should be far enough from both classes to leave some room for new data like x^{new} in figure 11. One convenient measure is to pick two points x_+, x_- in classes ± 1 s.t. its distance to the hyperplane is shortest in its class. The mathematical definition is given below:

$$\boldsymbol{x}_+ := \text{Argmin}_{\boldsymbol{x}^{(i)} \text{ s.t. } y^{(i)}=1} \{d(\boldsymbol{x}, \mathcal{H})\}$$

$$\boldsymbol{x}_- := \text{Argmin}_{\boldsymbol{x}^{(i)} \text{ s.t. } y^{(i)}=-1} \{d(\boldsymbol{x}, \mathcal{H})\}$$

where $d(\boldsymbol{x}, \mathcal{H})$ is the distance between point $\boldsymbol{x}$ and hyperplane $\mathcal{H}$.

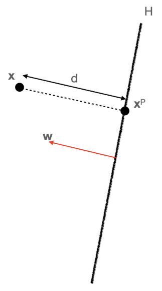


Figure 12: Distance from a point to hyper-plane

One method to find the distance is shown in figure 12. Given $\boldsymbol{x}$ and its perpendicular projection $\boldsymbol{x}^P$ onto hyperplane H, the distance $d(\boldsymbol{x}, H) = \|\boldsymbol{d}\|$ where $\boldsymbol{d} := \boldsymbol{x} - \boldsymbol{x}^P$ is a vector parallel to $\boldsymbol{w}$ (the normal vector of plane H) i.e. $\boldsymbol{d} = \alpha \boldsymbol{w}$ for some $\alpha \in \mathbb{R}$. Note $\boldsymbol{x}^P = \boldsymbol{x} - \boldsymbol{d}$ is on the hyperplane H, so $0 = \boldsymbol{x}^P \cdot \boldsymbol{w} + b = (\boldsymbol{x} - \boldsymbol{d}) \cdot \boldsymbol{w} + b = -\alpha(\boldsymbol{w} \cdot \boldsymbol{w}) + \boldsymbol{x} \cdot \boldsymbol{w} + b$. That means

$$\alpha = \frac{\boldsymbol{x} \cdot \boldsymbol{w} + b}{\boldsymbol{w} \cdot \boldsymbol{w}} = \frac{\boldsymbol{x} \cdot \boldsymbol{w} + b}{\|\boldsymbol{w}\|^2}$$

so

$$d(\boldsymbol{x}, H) = \|\boldsymbol{d}\| = \|\alpha \boldsymbol{w}\| = \left|\frac{\boldsymbol{x}\cdot\boldsymbol{w}+b}{\|\boldsymbol{w}\|^2}\right| \|\boldsymbol{w}\| = \frac{|\boldsymbol{x}\cdot\boldsymbol{w}+b|}{\|\boldsymbol{w}\|}$$

We can draw two hyperplanes parallel to H passing x_+ and x_- respectively. They are given by

$$\{\boldsymbol{x} : \boldsymbol{x}\cdot\boldsymbol{w}+b = c_1\} \quad \text{where } c_1 := \boldsymbol{x}_+\cdot\boldsymbol{w}+b$$

$$\{\boldsymbol{x} : \boldsymbol{x}\cdot\boldsymbol{w}+b = -c_2\} \quad \text{where } -c_2 := \boldsymbol{x}_-\cdot\boldsymbol{w}+b$$

note here $c_1, c_2 \geq 0$. See an example in figure 13.

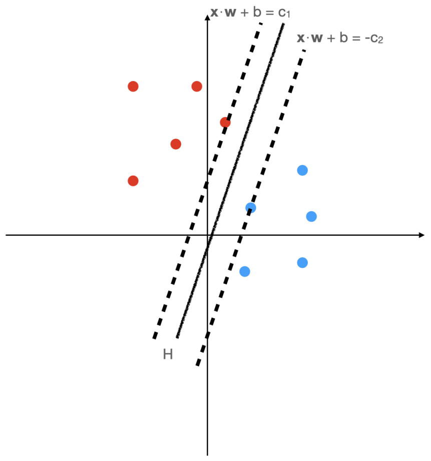


Figure 13: Separation hyperplane and its margins

The width of the road between two dotted margins measures the quality of separation plane H. And intuitively, we should place H in the middle of the road so that H is not too close to any of the classes. In this case, $c_1 = c_2$. And the width of the road is given by the projection of $\boldsymbol{x}_+ - \boldsymbol{x}_-$ onto the normal direction of H, i.e. distance $= (\boldsymbol{x}_+ - \boldsymbol{x}_-)\cdot\boldsymbol{w}/\|\boldsymbol{w}\|$. (see figure 14) When $c_1 = c_2$, road width is

$$(\boldsymbol{x}_+\cdot\boldsymbol{w} - \boldsymbol{x}_-\cdot\boldsymbol{w})/\|\boldsymbol{w}\| = \frac{c_1+c_2}{\|\boldsymbol{w}\|} = \frac{2c_1}{\|\boldsymbol{w}\|}$$

Note the two dotted planes (I will call them margins) in figure 13 and the road width are sensitive to changes in $\boldsymbol{x}_+, \boldsymbol{x}_-$, but not sensitive to changes in other points (unless a new point appears between two margins). i.e. SVM is sensitive to points in the training set closer to the margin.

Hyperplane parameters $\boldsymbol{w}, b$ are scale-invariant. i.e. if $\mathcal{H} = \{\boldsymbol{x} : \boldsymbol{x}\cdot\boldsymbol{w}+b = 0\}$, then the scaled version $\mathcal{H}' = \{\boldsymbol{x} : \boldsymbol{x}\cdot(k\boldsymbol{w})+kb = 0\}$ $(k \in \mathbb{R})$ is the same hyperplane. Therefore, we can scale $\boldsymbol{w}, b$ s.t. $c_1 := \boldsymbol{x}_+\cdot\boldsymbol{w}+b = 1$. In this case, the road width becomes simply $2/\|\boldsymbol{w}\|$. Now imagine you are rotating the plane H, i.e. rotating normal vector $\boldsymbol{w}$ (two margins should rotate together), the road width may change. We aim to find the maximal road distance, so we should solve the optimisation problem

$$\max_{\boldsymbol{w}} \frac{2}{\|\boldsymbol{w}\|}$$
$$\text{s.t. } \forall i,\ y^{(i)}(\boldsymbol{x}^{(i)}\cdot\boldsymbol{w}+b) \geq 1$$

the constraint means all points are lying away from the road formed by two margins. Maximising $2/\|\boldsymbol{w}\|$ is equivalent to minimising $\|\boldsymbol{w}\|/2$, which is equivalent to minimising $\|\boldsymbol{w}\|^2/2$ (norm is squared so that it can be differentiated, but the optimisation problem is still the same) So the SVM optimisation problem is

$$\min_{\boldsymbol{w}} \frac{\|\boldsymbol{w}\|^2}{2}$$
$$\text{s.t. } \forall i,\ y^{(i)}(\boldsymbol{x}^{(i)}\cdot\boldsymbol{w}+b) \geq 1$$

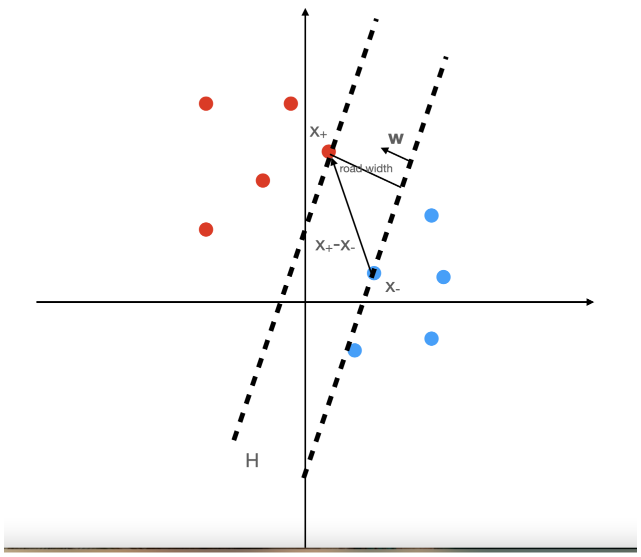


Figure 14: The road width of SVM separation

All $\boldsymbol{x}^{(i)}$ s.t. $y^{(i)}(\boldsymbol{x}^{(i)} \cdot \boldsymbol{w} + b = 1$ are called support vectors, so $\boldsymbol{x}_+, \boldsymbol{x}_-$ are both support vectors. Support vectors ”support” the two dotted margins.

This is a quadratically constrained quadratic programming (QCQP), you may search for many existing solvers that find the optimal $\boldsymbol{w}^*$. I will briefly introduce the KKT conditions for solving this QCQP.

6.1 KKT conditions

For the optimisation problem

$$\min_{\boldsymbol{x}} f(\boldsymbol{x})$$
$$\text{s.t. } \forall i,\ g_i(\boldsymbol{x}) \le 0$$

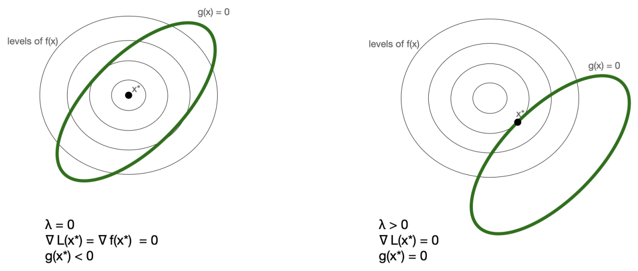


Figure 15: KKT graphical explanation

There are two cases for the optimal point $\boldsymbol{x}^*$ (see figure 15):

Case 1. $\boldsymbol{x}^*$ is in the interior of the feasible set, i.e. $\boldsymbol{x}^* \in \{\boldsymbol{x} : g_i(\boldsymbol{x}) < 0$ for at least one $i\}$.
In this case, the optimisation constraint is inactive (if you remove it, the optimal point is still $\boldsymbol{x}^*$). So the target function is $f(x)$ and Lagrange multipliers $\alpha_i = 0$ in the Lagrangian $\mathcal{L} = f(\boldsymbol{x}) + \sum_i \alpha_i g_i(\boldsymbol{x})$. So we have

$\nabla \mathcal{L}(\boldsymbol{x}^*) = \nabla f(\boldsymbol{x}^*) = 0$.

Case 2. $\boldsymbol{x}^*$ is on the boundary of the feasible set, i.e. $g_i(\boldsymbol{x}^*) = 0$ for all i.
This happens when the unconstrained optimal point is not in the feasible set. In this case, $\nabla f(\boldsymbol{x}^*), \nabla g_i(\boldsymbol{x}^*)$ are parallel (because the green circle cuts the level curve of f on the right of figure 15), and this is encoded in $\nabla \mathcal{L}(\boldsymbol{x}^*) = 0$. So α_i are the coefficients relating $\nabla f(\boldsymbol{x}^*), \nabla g_i(\boldsymbol{x}^*)$ and $\alpha_i \neq 0$ for at least one i.

Note in either case, $\alpha_i g_i(\boldsymbol{x}^*) = 0$, this is called the complementary slackness condition. This condition compactly includes both cases

Summarising all conditions above, we have the KKT conditions on $\boldsymbol{x}$,

$$\begin{aligned}
&\nabla \mathcal{L}(\boldsymbol{x}) = 0\\
&\forall i,\ g_i(\boldsymbol{x}) \leq 0\\
&\forall i,\ \alpha_i \geq 0\\
&\forall i,\ \alpha_i g_i(\boldsymbol{x}^*) = 0
\end{aligned}$$

When the objective function f is convex and the inequality constraints g_i are linear, KKT conditions are equivalent to $\boldsymbol{x}$ being the optimal value. This is the case for SVM.

6.2 Soft margin

The classification we mentioned above is a strict, hard classification. We rigorously require every point in the training set to fall on the correct side of the hyperplane. But in reality, there may be errors and abnormal points. See an example in 16.

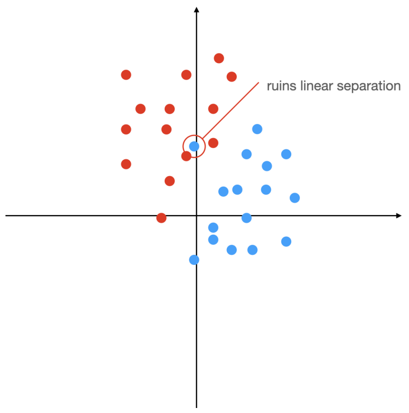


Figure 16: linear separation problem with abnormality

The unusual blue point makes the problem not linearly separable, and SVM (with a hard margin) mentioned above does not work for this data set. But by intuition, there is still a clear separation between the two classes if we ignore the blue point. Therefore, slackness $\zeta^{(i)} \geq 0$ are added to the constraint $y^{(i)}(\boldsymbol{x}^{(i)} \cdot \boldsymbol{w} + b) \geq 1$, now we only require

$$y^{(i)}(\boldsymbol{x}^{(i)} \cdot \boldsymbol{w} + b) \geq 1 - \zeta^{(i)}$$

i.e. some points i can fall on the road between two margins, or even fall on the wrong side (when $y^{(i)}(\boldsymbol{x}^{(i)} \cdot \boldsymbol{w} + b) < 0$) but we can't just allow any slackness, all $\zeta^{(i)}$ are summed and added to the objective function as a penalty. Now the optimisation problem becomes

$$\begin{aligned}
&\min_{\boldsymbol{w}} \left(\frac{\|\boldsymbol{w}\|^2}{2} + \lambda \sum_{i=1}^{N} \zeta^{(i)} \right)\\
&\text{s.t. } \forall i,\ \zeta^{(i)} \geq 1 - y^{(i)}(\boldsymbol{x}^{(i)} \cdot \boldsymbol{w} + b)\\
&\text{and } \zeta^{(i)} \geq 0
\end{aligned}$$

this is called SVM with *soft margin*. Since the objective function always minimises $\zeta^{(i)}$, if $1-y^{(i)}(\boldsymbol{x}^{(i)}\cdot\boldsymbol{w}+b) < 0$, $\zeta^{(i)}$ will simply be set to 0. However, if $1-y^{(i)}(\boldsymbol{x}^{(i)}\cdot\boldsymbol{w}+b) \geq 0$, $\zeta^{(i)}$ will be set to that value to satisfy the constraint. Therefore,

$$\zeta^{(i)} = \max\{0, 1 - y^{(i)}(\boldsymbol{x}^{(i)} \cdot \boldsymbol{w} + b)\}$$

6.3 Kernels

Suppose we have a problem that is clearly not linearly separable, for example, the one in figure 17, then neither hard-margin nor soft-margin SVM works. Note for simplicity, a 1-dimensional example is used ($p = 1$)

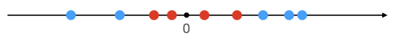


Figure 17: Not linearly-separable problem

But what if we add a new dimension, by creating $(x^{(i)}, (x^{(i)})^2)$? This is shown in figure 18. A new dimension allows more flexibility, as the separation hyperplanes in the 1D case are just points. (1 degree of freedom)

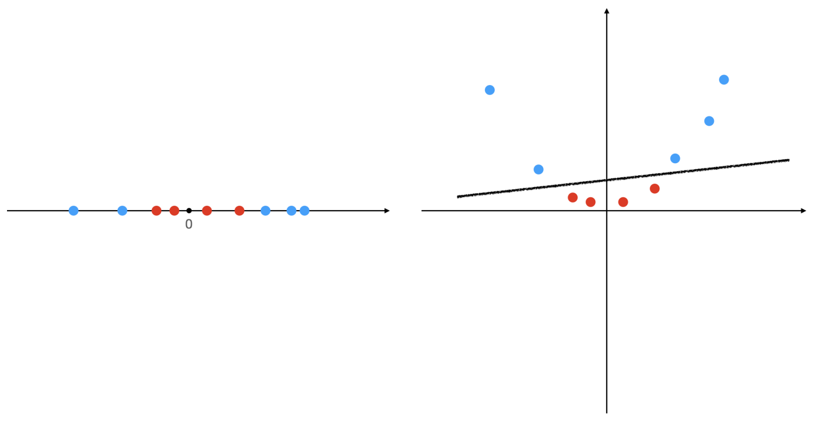


Figure 18: transforming non-linearly-separable problem to a higher dimension

When we have two variables: $(x_1^{(i)}, x_2^{(i)})$, there are 6 (two-term) combinations possible:

$$1,\ x_1^{(i)},\ x_2^{(i)},\ x_1^{(i)}x_2^{(i)},\ (x_1^{(i)})^2,\ (x_2^{(i)})^2$$

You can see that the dimension D of new space will quickly blow up even if we just consider the two-term combinations. Therefore, using SVM in the new space will be computationally expensive.

Before introducing kernels, let us first argue why the SVM algorithm does not need full data, but only the inner products:

In lecture notes, using the dual formulation of the optimisation problem, the objective Lagrangian can be written as

$$\mathcal{L} = \sum \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y^{(i)} y^{(j)} \boldsymbol{x}^{(i)} \cdot \boldsymbol{x}^{(j)} \qquad (*)$$

this may seem complicated, but by complementary slackness condition, Lagrange multiplier $\alpha_i \neq 0$ only when $g_i(\boldsymbol{x}) = 1 - y^{(i)}(\boldsymbol{x}^{(i)} \cdot \boldsymbol{w} + b) = 0$, i.e. $\boldsymbol{x}$ is a support vector. Usually, there are only a few support vectors, so the summation above is actually quite simple. e.g. when the only support vectors are $\boldsymbol{x}_+$, $\boldsymbol{x}_-$, if the corresponding Lagrange multipliers are α_+, α_-,

$$\mathcal{L} = \alpha_+ + \alpha_- - \frac{1}{2}(\alpha_+^2 \boldsymbol{x}_+ + \alpha_-^2 \boldsymbol{x}_- - 2\alpha_+\alpha_-(\boldsymbol{x}_+ \cdot \boldsymbol{x}_-))$$

Further, from (*) we can see that $\boldsymbol{w}$ is not present, and only the inner products of $\boldsymbol{x}$ are required. So this is helpful when the number of parameters p is much larger than N (e.g. in image processing, p is the number of pixels of each image and N is the number of images, $p \gg N$) the full data is $N \times p$, but there are only $N \times N$ inner products. $N \times N \ll N \times p$ when $N \ll p$.

Now back to the transformation problem. Kernels compute transformed inner product $T(\boldsymbol{x}^{(i)}) \cdot T(\boldsymbol{x}^{(j)})$ without actually finding all the $T(\boldsymbol{x}^{(i)})$. For example, if the transformation is $x^{(i)} \mapsto (x^{(i)}, (x^{(i)})^2)$, the new inner product is

$$(x^{(i)}, (x^{(i)})^2) \cdot (x^{(j)}, (x^{(j)})^2) = x^{(i)}x^{(j)} + (x^{(i)}x^{(j)})^2$$

you only need the inner products $x^{(i)}x^{(j)}$ in $\mathbb{R}^1$. If you think carefully about this example, actually the new dimension is redundant! The information contained in $\{(x^{(i)}, (x^{(i)})^2)\}_{i=1,\cdots,N}$ is the same as $\{x^{(i)}\}_{i=1,\cdots,N}$.

Kernels are defined as functions on $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$ s.t. $k(\boldsymbol{x}, \boldsymbol{y}) = T(\boldsymbol{x}) \cdot T(\boldsymbol{y})$ where $T(\boldsymbol{x}), T(\boldsymbol{y}) \in \mathbb{R}^D$ (where $D > p$). The aim is to write the inner products in the transformed space $\mathbb{R}^D$ using inner products in the original space $\mathbb{R}^p$. Such kernels solve linear separations in higher dimensions without actually computing anything in $\mathbb{R}^D$. So this is a powerful tool.

One commonly used kernel is the *polynomial kernel*, $k(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x} \cdot \boldsymbol{y} + r)^n$. For example, if $x, y \in \mathbb{R}^1$, and $r = 1/2, n = 2$:

$$k(x,y) = (xy + \frac{1}{2})^2 = x^2y^2 + xy + \frac{1}{4} = xy + (xy)^2 + \frac{1}{4}$$

this corresponds to the inner product of (x, x^2) and (y, y^2), which is the one we studied above. (1/4 is a constant so we can ignore it)
The one used in lectures is

$$k(x,y) = (xy+1)^2 = 2xy + (xy)^2 + 1$$

which corresponds to inner product $(\sqrt{2}x, x^2) \cdot (\sqrt{2}y, y^2)$. But there is no harm of this scaling $\sqrt{2}$ on the first axis, the classification problem will not change.

The degree n of the polynomial kernel decides the maximum interaction you are considering. e.g. if $n = 3$, there are terms like $x_1x_2x_3$, $x_5^2x_4$ (3-term interactions) in the kernel. The *radial basis kernel* considers all possible interactions, so it raises the problem to a ∞-dimensional SVM problem which is impossible to solve directly. The kernel will be given below and I will expand it so that you understand why this kernel considers all the interactions

$$k_R(\boldsymbol{x}, \boldsymbol{y}) := e^{\frac{-\|\boldsymbol{x}-\boldsymbol{y}\|^2}{\sigma}}$$

for simplicity we assume $\sigma = 2$ here

$$
\begin{aligned}
e^{\frac{-\|\boldsymbol{x}-\boldsymbol{y}\|^2}{2}} &= \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{y})\cdot(\boldsymbol{x}-\boldsymbol{y})\right) \\
&= \exp\left(-\frac{1}{2}(\boldsymbol{x}\cdot\boldsymbol{x}+\boldsymbol{y}\cdot\boldsymbol{y})\right)\exp(\boldsymbol{x}\cdot\boldsymbol{y}) \\
&= c\exp(\boldsymbol{x}\cdot\boldsymbol{y}+1) \quad \text{where } c := \exp\left(-\frac{1}{2}(\boldsymbol{x}\cdot\boldsymbol{x}+\boldsymbol{y}\cdot\boldsymbol{y})-1\right) \\
&= c\sum_{n=1}^{\infty}\frac{(\boldsymbol{x}\cdot\boldsymbol{y}+1)^n}{n!} \quad \text{by Taylor expansion}
\end{aligned}
$$

You can see from the last line that the Radial basis kernel encodes polynomial kernels $(\boldsymbol{x}\cdot\boldsymbol{y}+1)^n$ for all $n \in \mathbb{N}$. Despite how much information the radial basis kernel contains, it is computationally cheap.

The sigmoid kernel given in lecture notes will only make sense after studying neural networks so I will postpone the explanation to the next chapter

7 Neural Networks

7.1 Intro - logistic regression is a neural network

We learnt the logistic regression in a previous chapter, where we predict an outcome $y \in \{0, 1\}$ via estimating the probability $p := P(y = 1|\boldsymbol{x})$. It is estimated via a linear model

$$
\sigma^{-1}(p) \approx \boldsymbol{x}^T\boldsymbol{\beta}, \qquad \text{where sigmoid function } \sigma(z) := \frac{1}{1+e^{-z}}
$$

the vector $\boldsymbol{\beta}$ contains the coefficients on the predictors x_i. Then we determine a threshold $\tau \in [0, 1]$ and find estimate for y by:

$$
y = \begin{cases} 1, & \text{if } p > \tau \\ 0, & \text{if } p \leq \tau \end{cases}
$$

This is like passing p to a step function $s_\tau(p)$ defined as in the above equation.

After obtaining the optimal $\boldsymbol{\beta}$, given new entry $\boldsymbol{x}$, one estimate y by finding $p = \sigma(\boldsymbol{x}^T\boldsymbol{\beta})$, then find $y = s_\tau(p)$. This procedure above can be drawn into a network (see figure 19).

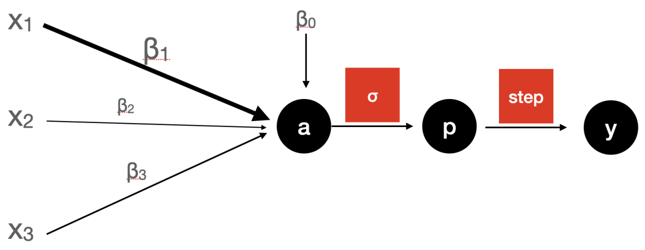


Figure 19: Logistic regression represented in a network

For simplicity, only three x_i are used, but in reality, you can add as many predictors as you like. The variable $a := \beta_0 + \sum \beta_i x_i = \beta_0 + \boldsymbol{x}^T\boldsymbol{\beta}$, and $p := \sigma(a)$. Note the thickness of lines β_i represents the magnitude of β_i, i.e. the effect of x_i on the value of a. σ (the function in the red block) is usually called an activation function.

A more general (single-layer) neural network is defined by

$$a := \boldsymbol{x}^T \boldsymbol{w} + b \quad \text{(first layer)}$$

$$y := \sigma(a) \quad \text{(hidden(activation) layer)}$$

$\boldsymbol{w}$ are the weights on $\boldsymbol{x}$, $b \in \mathbb{R}$ is the intercept. There are many choices for the activation function σ. A simple choice of σ is a step function (if step function is used instead of σ in logistic regression, we have a perceptron), but that brings the problem of non-differentiability, which makes gradient descent difficult. Therefore, many differentiable versions are found (see page 73 of lecture notes)

A one-layer neural network is enough for a simple binary classification problem, but in reality, there are more complicated outcomes. In 3blue1brown's videos on neural networks, he described an example of hand-written number recognition. The input $\boldsymbol{x} \in \mathbb{R}^2$ is $n \times n$ matrix of pixels of a picture of a hand-written number, and the output $y \in \{1, \cdots, 9\}$ is supposed to be the number written in the picture. We need to evaluate ten probabilities $(P(Y = 1), P(Y = 2), \cdots, P(Y = 9))$ instead of just one, therefore we need to construct a second layer of variables

$$p_j := \sigma(a_j) = \sigma(\boldsymbol{w}_j^T \boldsymbol{x} + b_j) \quad \text{for } j = 1, 2, \cdots, n_h$$

where the weights $\boldsymbol{w}_j := (w_{j1}, \cdots, w_{jn_h})^T$. In the case of hand-written number recognition, $n_h = 9$. This can be compactly written in matrix form (so that in python we can optimise computational cost)

$$\boldsymbol{p} := \sigma(\boldsymbol{a}) = \sigma(W\boldsymbol{x} + \boldsymbol{b}), \quad \text{where } \boldsymbol{p} = \begin{pmatrix} p_1 \\ \vdots \\ p_{n_h} \end{pmatrix}, \ \boldsymbol{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_{n_h} \end{pmatrix}, \ W = \begin{pmatrix} \boldsymbol{w}_1^T \\ \vdots \\ \boldsymbol{w}_{n_h}^T \end{pmatrix}$$

Then to find estimated outcome $\hat{y}$, you aggregate the p_i by

$$\hat{y} := \text{argmax}_{j \in \{1, 2, \cdots, n_h\}} p_j$$

but wait, how do you guarantee that the values $\boldsymbol{p}$ trained from the neural network are valid probabilities, i.e. p_i sum up to 1 and $p_i \in [0, 1]$. This is quite difficult to achieve within the layers, so we rather add another output layer called the soft-max. From now on I will write p_i as scores a_i instead, because $\sigma(W\boldsymbol{x} + \boldsymbol{b})$ may not be proper probabilities. One first idea to convert a_i to probabilities is defining

$$p_j := \frac{a_j}{\sum_i a_i}$$

but if suppose $a_j = 0$, then we would have $p_j = 0$ (i.e. jth class is impossible) this is too arbitrary. And suppose instead of having nice scores like $\boldsymbol{a} = (1, 2, 3)$, we have $\boldsymbol{a} = (101, 102, 103)$, then $\boldsymbol{p}$ calculated from the above formulae are (1/6, 1/3, 1/2) and (0.330, 0.333, 0.337) respectively. Clearly, this formula is sensitive to translations of scores.

A better score is given by treating a_j as exponents, i.e.

$$p_j := \frac{e^{a_j}}{\sum_i e^{a_i}}$$

Now if $a_j = 0$, p_j is a small value instead of 0 (it is kind of like the Laplace smoothing used in the naive Bayes method) And now if we plug in two $\boldsymbol{a}$ given above, p_j are both (0.09, 0.245, 0.665), surprisingly, this formula is invariant under any translation of scores. (Try to prove it!)

We do not have to restrict on two layers, the generalisation is given below: start with $\boldsymbol{h}^{(0)} := \boldsymbol{x}$,

$$\boldsymbol{h}^{(k)} := \sigma\left(W^{(k-1)}\boldsymbol{h}^{(k-1)} + \boldsymbol{b}^{(k-1)}\right) \qquad \text{for } k = 1, 2, \cdots, L$$

we are just adding (k) on the top which represents these are the predictors of kth layer. If there are L layers in total, the prediction

$$\hat{y} = \sigma_{\text{out}}\left(\boldsymbol{w}^{(L)}\boldsymbol{h}^{(L)} + b^{(L)}\right)$$

Sometimes, to make this neural network more interpretable, σ_{out} is set to identity. So the neural network extracts features $\boldsymbol{h}^{(L)}$ assigned with weights $\boldsymbol{w}^{(L)}$ from the data $\boldsymbol{x}$, because y is a linear combination of $h_i^{(L)}$.

Note I used vector $\boldsymbol{w}^{(L)}$ instead of a matrix in the expression of $\hat{y}$ to produce a single output. But if the matrix is used instead, we have multiple outputs, i.e. $\boldsymbol{y}$ is a vector. And if necessary, different activation functions σ can be applied at each layer.

Warning: the activation function σ cannot be chosen as identity, as otherwise,

$$\begin{aligned}
\boldsymbol{h}^{(2)} &= \sigma\left(W^{(1)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(1)}\right) \\
&= W^{(1)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(1)} \quad \text{identity } \sigma \\
&= W^{(1)}(W^{(0)}\boldsymbol{h}^{(0)} + \boldsymbol{b}^{(0)}) + \boldsymbol{b}^{(1)} \\
&= W^{(1)}W^{(0)}\boldsymbol{h}^{(0)} + (W^{(1)}\boldsymbol{b}^{(0)} + \boldsymbol{b}^{(1)})
\end{aligned}$$

which means $\boldsymbol{h}^{(2)}$ is essentially another linear combination of $\boldsymbol{h}^{(0)}$.

7.2 Back-propagation

In the introduction, we have only introduced the structure of a neural network that is already trained, i.e. the $W^{(k)}$ and $\boldsymbol{b}^{(k)}$ are determined. But how to choose these coefficients and intercepts so that the neural network gives reliable prediction? We should adjust $\theta^{(k)} := (W^{(k)}, \boldsymbol{b}^{(k)})$ so that the cost function (average squared error of all N data points)

$$L(\theta) := \frac{1}{2N}\sum_i (y_i - f_\theta(\boldsymbol{x}_i))^2$$

is minimised. f_θ is the prediction function corresponding to the neural network. $2N$ is divided instead of N just to make the derivative more decent.

One of the ways to achieve this minimisation is gradient descent. For those of you who have not learnt gradient descent before, let us take a simple single-layer neural network representing linear regression (i.e. activation σ_{out} is identity) with three inputs x_i and one output a (figure 20)

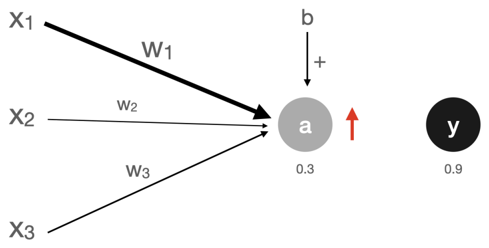


Figure 20: Gradient descent on a simple neural network

Note the output of neural network $a = 0.3$, which is much smaller than the true y value 0.9. So we should somehow try to increase a. We cannot control the input x_i, but we can adjust w_i and b. Since you are not sure how exactly each of the variables affects loss L, you can try to tune each w_i. For example, if an increase of w_1 results in an increase of L, then you should go the opposite direction and decrease w_1. After trying various combinations of changes, you will come up with a change $\Delta\theta := (\Delta w_1, \Delta w_2, \Delta w_3, \Delta b)$ that gives a reasonable decrement in loss L. $\Delta\theta$ is called a descent direction. The gradient $\nabla_\theta L$ is the direction of the steepest descent, and its meaning is essentially the same as the approach we described above. But once we derived a formula for gradient, we do not have to repeat this search of optimal descent direction for every new data set. Now we update θ by defining new $\theta' := \theta - \lambda \nabla_\theta L$. Note we defined a coefficient λ here, this is the *learning rate* and it decides how long should we walk along the descent direction. λ has to be carefully controlled because recall that gradient describes local behaviour, the steepest descent direction does not guarantee descent forever in this direction. The above process can be repeated until a satisfactory low loss L is reached, and this method is called *gradient descent*.

But now if we have m inputs and n output instead, where m, n could be thousands or millions. There are mn weights to tune, i.e. mn partial derivatives to calculate in the gradient. This is computationally expensive. What if instead of trying to tune all weights at each iteration, we randomly pick a small subset of the weights, and find a good descent using this subset? Though we do not have optimal descent of L in each step, the amount of descent in L brought by tuning a few weights is enough to lead us to its minima. This method is called *stochastic gradient descent*.

To perform gradient descent, one must find the gradient of the loss function first. For neural networks, finding the gradient seems to be cumbersome. There is no direct formula relating $\hat{y}$ and $\boldsymbol{x}$. (and the gradient formulae given in lecture notes look scary) But I will start with a simple example and gradually lead you to the final formula.

For the one-layer neural network given in figure 21, with relations

$$\hat{y} = \sigma(a), \quad a := \boldsymbol{w}^T \boldsymbol{x} + b$$

and loss function $L = \frac{1}{2}(\hat{y} - y)^2$

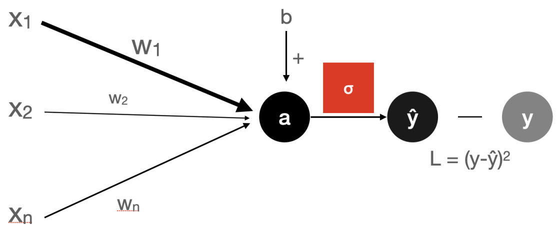


Figure 21: Neural network with a single layer

To find $\frac{\partial L}{\partial w_i}$, we need to trace how w_i affects L step by step. Small change in w_i first changes a (with scale $\frac{\partial a}{\partial w_i}$), then a influences $\hat{y}$ (with scale $\frac{\partial \hat{y}}{\partial a}$). Finally, $\hat{y}$ influences L with scale $\frac{\partial L}{\partial \hat{y}}$. Summing up (actually we multiply them together) all information above, we have

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a} \frac{\partial a}{\partial w_i}$$

this is essentially the chain rule, and it describes how the influence of w_i passes through the network. Since this is a simple case, you may work out each derivative and give an explicit formula of $\frac{\partial L}{\partial w_i}$. We have not found $\frac{\partial L}{\partial b}$ yet, but the idea is essentially the same. If you have multiple outputs y_i instead of just one, then the only difference is that loss L becomes loss L_i (one loss for each y_i). Or you can aggregate them by finding the mean loss across all output variables.

For multi-layer networks, we should travel backwards along the network. (because some weights in the first few layers are deeper, i.e. further from the output) For simplicity, we discuss a neural network with only one unit in each layer. (see figure 22)

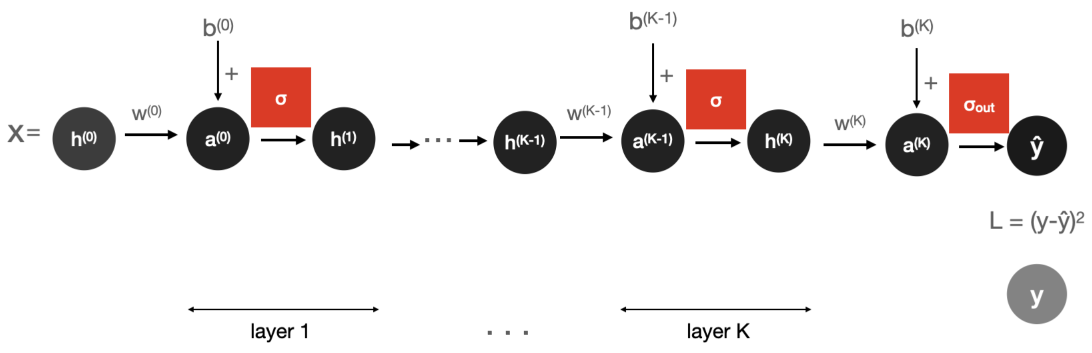


Figure 22: A K-layer neural with a single unit in each layer

in the figure, I did not create a layer for the activation function σ, but they should be imposed on each layer. Starting from the last layer K, we can consider three gradients: $\frac{\partial L}{\partial w^{(K)}}, \frac{\partial L}{\partial b^{(K)}}, \frac{\partial L}{\partial a^{(K)}}$. (here $a^{(K)} := w^{(K-1)}h^{(K-1)} + b^{(K-1)}$, a linear combination of h from the previous layer) Note we cannot directly change $a^{(k)}$, but we can record this gradient, and use it for the next layer. i.e. we define the error of layer k

$$\delta^{(k)} := \frac{\partial L}{\partial a^{(k)}}$$

for $k = 1, \cdots, K$.
In this example,

$$\delta^{(K-1)} = \frac{\partial L}{\partial a^{(K-1)}} = \frac{\partial h^{(K)}}{\partial a^{(K-1)}} \frac{\partial a^{(K)}}{\partial h^{(K)}} \frac{\partial L}{\partial a^{(K)}} = \sigma'(a^{(K)}) w^{(K)} \delta^{(K)}$$

i.e. gradient of layer $K-1$ depends on gradient of layer K and the weights at layer K. This is the core of back-propagation.

As you can see, finding these derivatives boils down to tracing along the network from the weight to the output. As an exercise, try to write expressions of $\frac{\partial L}{\partial w^{(K-1)}}, \frac{\partial L}{\partial b^{(K-1)}}$ using $\delta^{(K)}$.

Note if there are multiple units in each layer (i.e. $h^{(k)}$ becomes vector now, then we need two indices for w to indicate which two nodes it is connecting, for example, $w_{pq}^{(k-1)}$ connects $a_p^{(k)}$ with $a_q^{(k-1)}$. Then you obtain the formulae 8.16 in lecture notes. And the back-propagation rule becomes

$$\delta^{(K-1)} = \sigma'(\boldsymbol{a}^{(K)})(W^{(K)})^T \delta^{(K)}$$

where we just changed $a^{(K)}$ to vector $\boldsymbol{a}^{(K)}$ and changed w to matrix.

7.2.1 Problems of Optimisation

For all optimisation problems encountered in this course, including the one for back-propagation, one needs to think about which optimisation method converges to a global minimum instead of a local one. And if there are multiple local minima (on the training set), which one will give a lower loss on the test set?

There are many optimisers to choose from, but studies have indicated that possibly optimisers with smaller initialisation, smaller batch size (for stochastic optimiser) and smaller momentum will give the global minima with better ability to generalise to new data.

7.3 Regularisation

Neural networks have a large number of parameters, including $\{w_{pq}^{(k)}\}$ and $\{b_j^{(k)}\}$ for each layer k. This put us at risk of over-fitting. e.g. in regression, if the true model is a linear model, but you used a polynomial model with 10 parameters, that gives a perfect performance on the training set, but it cannot generalise. Again, controlling the number of parameters amounts to the variance-bias trade-off. If the number of parameters is too small to capture the structure (under-fitting), the bias is large, but the variance (variation in the models when different training sets are used) is small. In the case of over-fitting, the bias is small, but the variance is large.

To control model complexity for neural networks, we need to add a term to the loss function that measures complexity. However, the number of parameters is not differentiable. So a function (e.g. norm) of all parameters is used. i.e. If θ represents a vector of all the parameters, the regularised loss function is

$$L(\theta, \alpha) = L_0(\theta) + \lambda R(\theta)$$

L_0 is the original loss function, R can be any non-negative function measuring the complexity of the model, and $\lambda \in \mathbb{R}$ is the regularisation strength.

In gradient descent, if the 2-norm regularisation is used, the method is called *weight decay*, because the update $\theta \mapsto \theta - \eta \nabla L(\theta, \alpha)$ (where η is learning rate) now becomes

$$\theta \mapsto \theta - \eta\lambda\theta + \nabla L_0(\theta) = (1 - \eta\lambda)\theta + \nabla L_0(\theta)$$

i.e. we add weight to parameters before doing gradient descent. Other regularisation includes:

- $R(\theta) = \|\theta\|_0 :=$ number of non-zeros in θ. This is useful for models where many parameters could be 0.
- $R(\theta) = \|\theta\|_1$ relaxation of the 0-norm (because it is continuous), also useful when we need parameters to be sparse.
- drop-out: randomly ignore units (by setting all weights associated with that unit to 0) in each iteration of the training stage. Note this is not deleting the units, the units will be restored when the neural network is used for the prediction of new data.
- Operator/spectral norm of matrices $W^{(k)}$.
- You can also use a combination of different regularisation, but you need one regularisation strength parameter for each of them.

For any regularisation you use, you must carefully control regularisation strength so that regularisation is working, but not causing a significant raise in the bias.

7.4 Convolutional Neural Network

The combination rule for the neural network we described above is the linear combination or inner product. i.e. $\boldsymbol{a} = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ But there are other combination rules possible. And if you use the convolution $\boldsymbol{a} = \boldsymbol{w} * \boldsymbol{x} + b$, this is the *convolution neural network*. The idea of convolution comes from a simple probability problem, that is finding the distribution of $X + Y$ where X, Y are discrete/continuous random variables. Suppose for simplicity $X, Y \in \{1, 2, \cdots, 10\}$, then $X + Y \in \{1, 2, \cdots, 20\}$ and

$$P(X + Y = n) = \sum_{i+j=n} P(X = i)P(Y = j) = \sum_{i=1}^{9} P(X = i)P(Y = n - i)$$

But we do not have to restrict summations from $i = 1$ to $i = 9$, we can sum from $-\infty$ to ∞ while keeping the value of the sum unchanged, because at least one of $P(X = i)$, $P(Y = n - i)$ is 0 for $i \notin \{1, 2, \cdots, 9\}$. This formula generalises to any pair of discrete random variables, and it is called the convolution between X and Y

$$X * Y := \sum_{i=-\infty}^{\infty} P(X = i)P(Y = n - i)$$

And if X, Y are lists of numbers, the convolution can also be formed by

$$X * Y := \sum_{i=-\infty}^{\infty} X_i Y_{n-i}$$

where we would define $X_i = 0$ if i is not an index in list X. Intuitively, the convolution of two lists is like taking a scanner Y and scanning through the list X, see figure 23. This convolution is like taking a moving 3-average. Note for the first term, I implicitly added two 0s in front of X so that the first sum makes sense. This is called *padding* and you can adjust the padding to change the output size. For example, if we do not extend the list (no padding), and start with the sum $1 \times 1/3 + 2 \times 1/3 + 3 \times 1/3$ directly, then the output length is only 4.

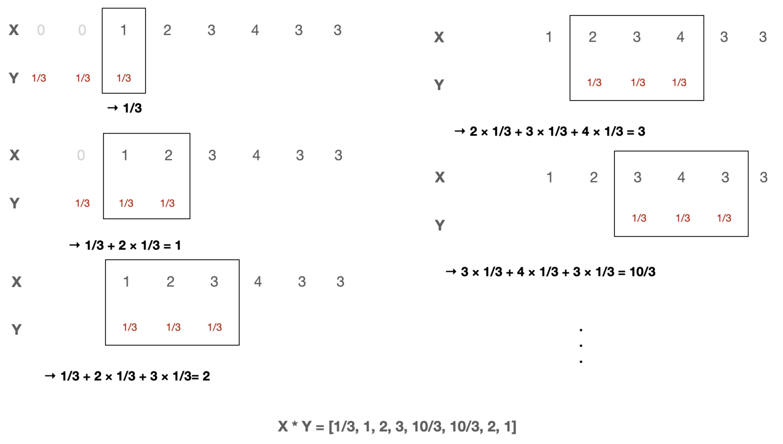


Figure 23: Convolution of 1-dimensional lists

Convolution can also be taken between two functions h, k

$$(h * k)(t) := \sum_{\tau=-\infty}^{\infty} h(\tau)k(t-\tau)$$

strictly speaking, the integral is used in the proper definition of convolution between two functions, but for computers, sums are easier than integration. The k here is usually called kernel, and you can choose it to satisfy various purposes. For example, if k is the PDF of the standard normal distribution, then it makes h smoother. (Think of why using the idea in figure 23)

If we imagine a square scanner scanning through a 2-D grid (after scanning), this is a 2-D convolution. This is very useful in feature extraction in image processing. For example, figure 24 shows a hand-written number 1 on a 6×6 grid (h is defined as the pixel values) and the scanner (i.e. kernel) k is a 3 by 3 array.

if we define

$$k = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

and scan it through the array h, the value will be maximised when there is a straight line in the middle of the scanner. i.e. we extracted the feature ”vertical line” in the image. Of course, you can change k to capture other features. The convolution (with no padding) is given below and you may verify it on your own

$$h * k = \begin{pmatrix} 1 & 3 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 3 & 0 & 0 \end{pmatrix}$$

Since the second column of the convolution matrix describes the same feature, we can use the pooling technique. That is merging the 2×2 sub-blocks by taking the maximum value

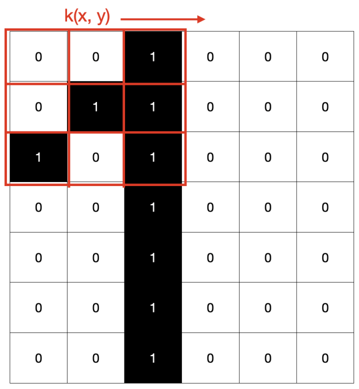


Figure 24: Convolution of 1-dimensional lists

$$\text{pool}(h * k) = \begin{pmatrix} 3 & 0 \\ 3 & 0 \end{pmatrix}$$

Then we can put these four values into the neural network and train the model. This is a great save in computational cost compared to using the 36 pixel values as input $\boldsymbol{x}$.

Suppose the image size is larger (e.g. 15×15 in figure 25) but you still wish to use a 3×3 kernel k. Scanning through the whole image like before may be redundant, you can add a stride (i.e. a spacing between each point of the scanner k) to scan faster. In real life the size of images may be 1800×1200 or even higher, in those cases, you may consider using larger strides and larger kernels.

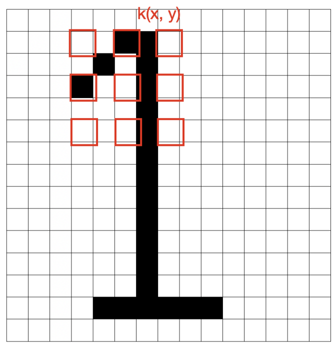


Figure 25: Convolution with stride $s = 2$

This idea of convolution can be generalised to any dimension (e.g. to 3D for video processing)

8 Unsupervised Learning

In all chapters before, we are given data sets with predictors $x^{(i)}$ and target variable (or variable of interest) $y^{(i)}$. In some cases, the target variable comes from observations or measurements. But in other cases like training a model for image recognition ($x^{(i)}$ are the pixel values and $y^{(i)}$ are labels of the image), we need humans to label the data. (i.e. $y^{(i)}$ are artificially defined) This can be time-consuming. Unsupervised learning aims to extract features of the data $x^{(i)}$ without any label $y^{(i)}$. For example, the K-mean algorithm covered in this course is one of the clustering methods, which tries to classify $x^{(i)}$ into different groups. It can be separating customers into groups according to their expense, age, and number of visits, or it can be virus classification using shape, substances contained in the virus etc. But the classical K-means and hierarchical clustering are rather restricted. There are many pieces of research on boosting these algorithms and other popular clustering methods like DBSCAN(density-based spatial clustering of applications with noise), and distribution-based clustering.

8.1 K-means algorithm

As the name of this algorithm suggests, it tries to find the k centroids (i.e. mean of all points in that cluster) of K clusters. Suppose the clusters are sets c_l for $l = 1, 2, \cdots, k$, the centroid of cluster l is defined as

$$\boldsymbol{m}_l := \frac{1}{|c_l|} \sum_{i \in c_l} \boldsymbol{x}^{(i)}$$

The algorithm can start by randomly assigning data points to clusters and then finding the centroids, or randomly picking k centroids in the first place. Then we update by assigning points to the closest centroid, this is shown in figure 26 using a simple data set where you can clearly see there are two clusters.

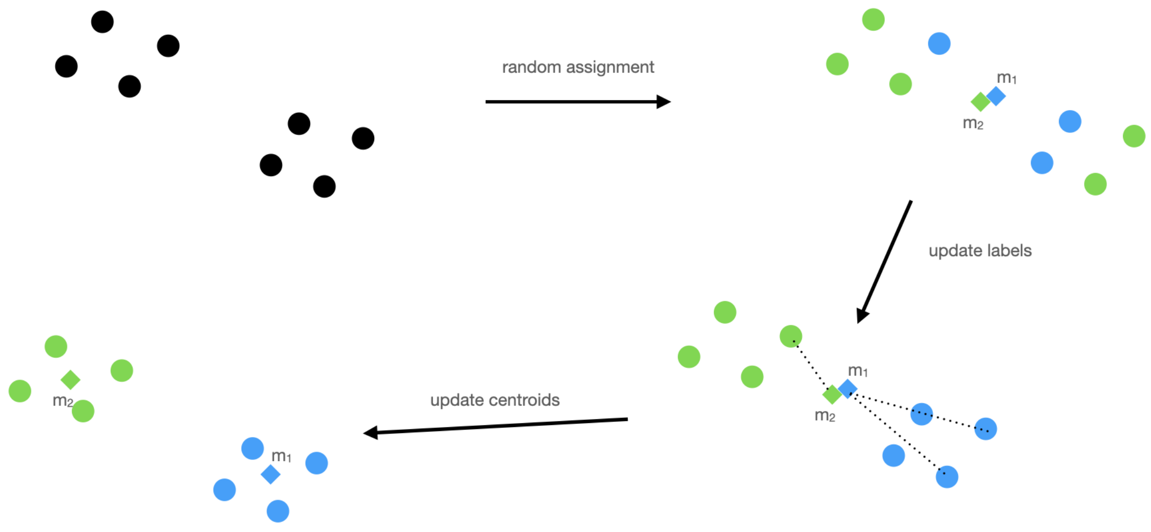


Figure 26: A simple example of K-means algorithm with $k = 2$

First, we randomly assign points to one group. (one of green and blue) You can see that the points are quite messy, but this will be fixed after the update. We find the centroids $\boldsymbol{m}_1, \boldsymbol{m}_2$ for two clusters. Then, we update the assignments of points by choosing the cluster l where its centroid m_l is closer to the point (the distances are shown as dotted lines) Intuitively, the points are already assigned to the correct clusters. We update the centroids again. Note if we try to perform another iteration, there will be no change to the assignment of data, so we claim that the algorithm has converged. Other convergence criteria may be used.

There are some questions not yet answered: how to choose the hyper-parameter k? is this algorithm guaranteed to converge? If two clusters are arranged in weird shapes like half-moons, K-means actually have bad performance. As for the choice of k, gap statistics are used. (You may search online for this)

For mathematical details of K-means clustering, please check the official lecture notes.

8.2 Hierarchical clustering

Unlike K-means, hierarchical clustering does not aim to explicitly separate the data into k groups. It rather groups some similar data at each stage, then merge the groups, and then repeats. Just like an evolution hierarchy. An example is given in figure 27.

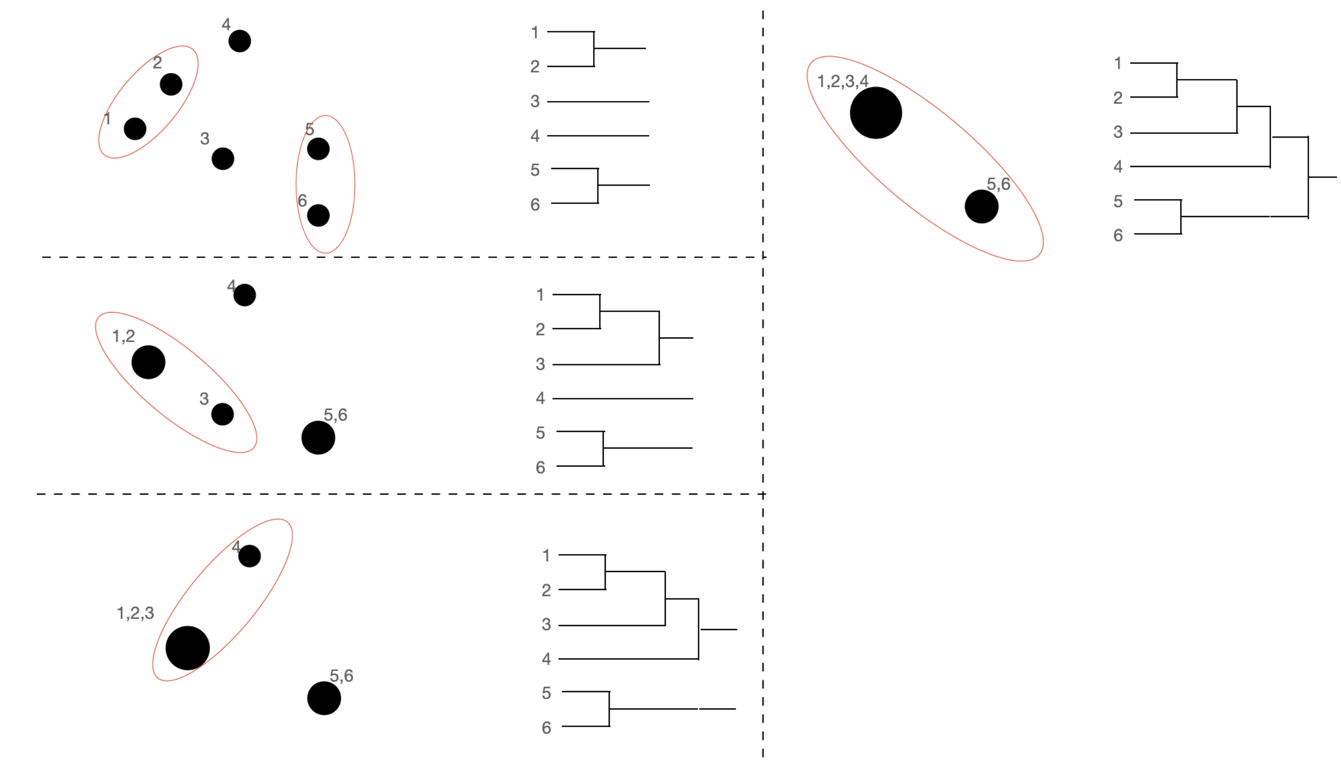


Figure 27: A demonstration of hierarchical clustering

In the first round, data 1, 2 and 5, 6 are similar enough (in practice, you can decide the threshold of "similarity"), so they are grouped together. 3, and 4 are left alone (or you can treat them as groups with a single element). Then we treat $\{1, 2\}$ as a group and $\{5, 6\}$ as a group, and use their centroid as the representation of the group. The data set is reduced to 4 points and we can run another round of clustering. The points $\{1, 2\}$ and 3 are close, so they are grouped together. We repeat this until all the points are grouped together. So each round, the group at least once to reduce the number.

There are dendrograms drawn to the right of each stage of grouping, you can see how data are related. Dendrograms can get quite complicated (especially for larger data-set), but when implementing the algorithm we do not need to draw them, we are just giving data points labels.

To perform hierarchical clustering, one needs to define the distance measure. At each iteration, points with a small enough distance are grouped together. Distance measures like simple linkage, complete linkage and group average are discussed in section 5.1 of the official lecture notes.

8.3 Comparing Clusters

As described in lecture notes, the quality of a single clustering result is measured by how compact each cluster is compared to its distance to other clusters. (measures like silhouettes use this idea) What if now we have two clustering results: $X_1, \cdots, X_r$ (sets of clusters obtained from model X) and $Y_1, \cdots, X_s$? (sets of clusters obtained from model Y) Note here r may not be the same as s because you do not have control of the number of clusters in clustering methods like DBSCAN (finding the number of clusters is part of DBSCAN). I will introduce the adjusted rand index(ARI) measure and the intuitions behind it.

First, recall the accuracy measure mentioned before in supervised learning:

$$\text{ACC} = \frac{\text{TP} + \text{TN}}{\text{N}}$$

where N is the total number of test samples. In unsupervised learning, there is no ground truth labels $y^{(i)}$, so instead we consider the agreements between two clustering results. i.e. we replace TP with the number of pairs of data there are classified into the same cluster for both clustering results. We defined this as A_s (A stands for agreement, s stands for same cluster)

$$A_s := \left|\left\{(i,j) \,:\, x^{(i)}, x^{(j)} \in X_k, \text{and } x^{(i)}, x^{(j)} \in Y_l \quad \text{for some } l, k\right\}\right|$$

Similarly, the TN can be replaced by the number of pairs of data that are both NOT classified into the same cluster (define this as A_d, d means different clusters). There are $N := \binom{n}{2}$ pairs where n is the total number of data points. So the alternative for ACC (called Rand index) in unsupervised learning is defined as

$$\text{index} := \frac{A_s + A_d}{N} = \frac{A_s + A_d}{\binom{n}{2}}$$

This index can be compared with the index for a pair of random clustering results (with the same number of clusters as model X and Y), and we call this $E(\text{index})$, the expected index. So if index $- E(\text{index})$ is positive, the similarity between two clustering results is better than random classifications. But how to interpret the scale of this quantity? We use the max-scaling technique:

$$\frac{\text{index} - E(\text{index})}{\max \text{index} - E(\text{index})}$$

where max index is the maximum index (i.e. maximum agreement) for two clustering results (with the same number of clusters as model X and Y). The scaled version is exactly the measure ARI, and it is 1 when the agreement between two clustering results is maximum. Note ARI can be negative if the index is even worse than random clustering.

The formulae given in official lecture notes is the contingency table version of ARI. The quantities n_{ij} (number of common data points in cluster X_i and Y_j) are used. Firstly, the agreement (index) is given by

$$\sum_{ij} \binom{n_{ij}}{2}$$

this is simple to interpret, if there are n_{ij} common data points in cluster X_i and Y_j, then there are $\binom{n_{ij}}{2}$ pairs of data points classified into the same cluster in both model X and Y. The expected value of $\binom{n_{ij}}{2}$ is given by

$$\binom{\sum_j n_{ij}}{2} \binom{\sum_i n_{ij}}{2} \Big/ \binom{n}{2} = \frac{\binom{a_i}{2}\binom{b_j}{2}}{\binom{n}{2}}$$

where we defined $a_i := \sum_j n_{ij}$ (size of cluster X_i) and $b_j := \sum_i n_{ij}$ (size of cluster Y_j). The formula means

$$\frac{\text{total number of possible distinct pairs in } X_i \times \text{total number of possible distinct pairs in } Y_j}{\text{total number of pairs}}$$

Therefore,

$$\begin{aligned} E(\text{index}) &= E\left(\sum_{ij} \binom{n_{ij}}{2}\right) = \sum_{ij} E\left[\binom{n_{ij}}{2}\right] \\ &= \sum_{ij} \frac{\binom{a_i}{2}\binom{b_j}{2}}{\binom{n}{2}} = \frac{\sum_{ij} \binom{a_i}{2}\binom{b_j}{2}}{\binom{n}{2}} \\ &= \frac{\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}}{\binom{n}{2}} \end{aligned}$$

Finally, the maximum index possible is given by summing the total number of possible distinct pairs in all X_i and Y_j, which is

$$\frac{1}{2}\left[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}\right]$$

2 is divided because there is symmetry when comparing two clustering results. Plugging all formulae derived above into the ARI formulae, we have

$$ARI = \frac{\sum_{ij}\binom{n_{ij}}{2} - \sum_i \binom{a_i}{2}\sum_j \binom{b_j}{2}/\binom{n}{2}}{\frac{1}{2}\left[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}\right] - \sum_i \binom{a_i}{2}\sum_j \binom{b_j}{2}/\binom{n}{2}}$$

which is the formulae given in the lecture notes.

For more details on ARI, see (Rand, 1971), (Hubert, 1977), (Hubert and Arabie, 1985), (Samuh, Leisch and Finos, 2014).

The Adjusted mutual information (AMI) given in section 7.2 of the lecture notes seems quite scary, but you can see now that the mutual information (MI) assembles the cross-entropy given in supervised learning, and AMI used the same max-scaling as ARI. Though finding the expected value of MI is more cumbersome.

8.4 EM Algorithm

EM algorithm implements simple ideas to perform a difficult task: finding the distribution of data when the distribution is complicated. I will provide a brief introduction here.

Suppose we suspect the data $x^{(i)}$ follows some distribution consisting of different types, e.g. a mixture of Gaussian and exponential distribution

$$p(x) = \pi \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} + (1-\pi)\lambda e^{-\lambda x}$$

where π is the ratio between two distributions. We perform the famous expectation-maximisation algorithm to find the parameters of the distributions.

The general pdf for a mixture of distributions is

$$p_\theta(x) = \sum_{k=1}^{m} \pi_k p_k(x|\theta)$$

where θ contains the parameters the distribution components p_k and π_k are the mixture proportions. $m \in \mathbb{Z}$ is the total number of distribution components in this mixture model. The *Gaussian mixture models* are models where each p_k is a normal distribution.

Suppose now we collect a set of data $\{x^{(i)}\}_{i=1,\cdots,N}$. The *latent variables* (hidden variables) $z^{(i)}$ taking values in $\{1, \cdots, m\}$ can be added to indicate which distribution component (or cluster) the ith data $x^{(i)}$ belongs to.

$$p(x^{(i)}, z^{(i)}) = p(x^{(i)}|z^{(i)})p(z^{(i)})$$

$z^{(i)} \in \mathbb{R}$ follows multi-nomial distribution taking values in $\{1, \cdots, k\}$ (multi-nomial distribution is an extension of the binomial distribution, but instead of two outcomes we can have an arbitrary number of outcomes) Note, $z^{(i)}$ are not given with the data set, we only have access to $x^{(i)}$.

If we have access to latent variable $z^{(i)}$, the $p(x^{(i)}, z^{(i)})$ only depends on the parameters of each distribution in the mixture and the mixture proportions π_k. Therefore, we can estimate the parameters using statistical estimators like

MLE(maximum likelihood estimator) and MME(method of moment estimator). The estimator for π_k can simply be taken as the number of data points in each distribution component k, i.e.

$$\hat{\pi}_k = \frac{1}{N}\sum_{i=1}^{N} \delta_{z^{(i)},k}$$

If you have the parameters $\boldsymbol{\theta}$ of the mixture distribution and π_k, $z^{(i)}$ cannot be estimated directly, but you can estimate the probability of ith data belonging to cluster k: $r_{ik}(\boldsymbol{\theta}, \boldsymbol{\pi}) := P(z^{(i)} = k|x^{(i)}, \boldsymbol{\theta}, \boldsymbol{\pi})$ using Bayes's theorem. (the idea is similar to Naive Bayes's method) This is a *deadlock*. $z^{(i)}$ can be estimated using θ, π_k (called the E-step), and θ, π_k can be found using $z^{(i)}$ (called the M-step) but we know neither of them.

EM(expectation-maximisation) algorithm solves this deadlock by randomly guessing the latent variables $z^{(i)}$ or parameters θ, π_k, and then estimating the other using scheme proposed above. We repeat the E-step and M-step, iteratively updating all the parameters until the distribution found fits the data. This is like a neural network with two layers (one layer contains the latent variables and another contains parameters, there is no output or input for this neural network), and we repeatedly forwards-propagate and backwards-propagate.

This is only a brief introduction, and there are many details yet to explain. Please refer to the lecture notes.

8.5 Hidden Markov Models

In natural language processing, given a sentence, it is often necessary to determine the part of speech (POS) of each word. e.g. "I am fine" has POS labels [pronoun, verb, adjective]. Note in other sentences, "fine" could be a noun, verb or adverb and the meaning will change. The process of finding POS labels is called *POS tagging*.

Clearly, the order of the words matters. And some flow of words is more likely than others. For example, it is not likely to have two adjectives connecting together in English, but the combination [verb, adjective] is quite common. For this reason, we could model the POS as a *discrete Markov chain*. If the POS of tth word is Z_t, imagine Z jumping between the states (all types of POS) with certain transition probabilities. (see figure 28 where we only consider nouns, verbs and adjectives) There should be 9 transition probabilities for a 3-state Markov chain and they can be recorded in a 3 by 3 matrix A, but those with probability 0 are not drawn on the diagram. The transition matrix A for Markov chain in figure 28 is

$$A = \begin{pmatrix} 0.6 & 0.4 & 0 \\ 0.4 & 0 & 0.6 \\ 0.8 & 0 & 0.2 \end{pmatrix}$$

where each row contains transition probabilities starting with nouns, verbs and adjectives respectively. For example, the first entry in the first row means that given the current POS is a noun, the probability that the next word has POS noun is 0.6. (Warning: the values of transition probabilities in figure 28 are just arbitrary choices)

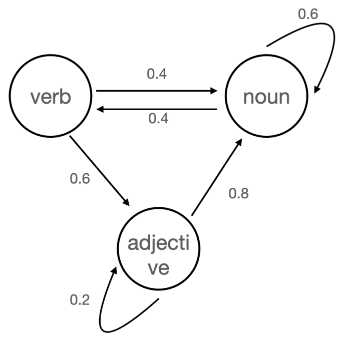


Figure 28: A Markov chain of part of speech

Each POS Z_t emits a word X_t, the probability that the word is x is $P(X_t = x \mid Z_t = k)$ and this is called the *emission probability.* Similar to transition probabilities, we can build a matrix B of emission probabilities. If there are W possible words and P possible POS, the matrix has shape $W \times P$. The model with variables (X_t, Z_t) and probabilities A, B is called *Hidden Markov Model*(HMM), an example of HMM for POS tagging is given in figure 29. HMM can be applied to many other fields like genetics and finance. In practice, the latent(state) variables Z_t are not observable, only the emissions X_t are observed. For POS tagging, the part of speech is the hidden variable that can not be observed.

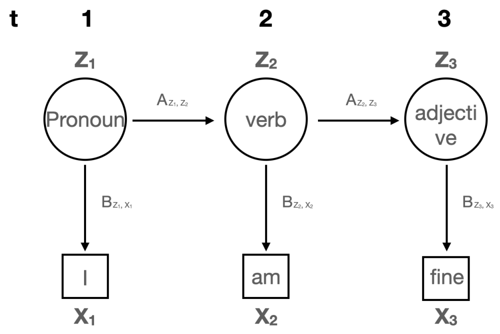


Figure 29: An example of hidden Markov Model

8.5.1 Forward-backward algorithm

Suppose the model has already been trained and transition matrix A and emission matrix B are known(we say the model is known in this case). There are several ways to estimate Z_t depending on the condition.
You can estimate as data X_t streams in (imagine acoustic information coming while the speech recognition algorithm is running), probability distribution $P(Z_t \mid X_1 = x_1, \cdots, X_t = x_t)$ is found at time t. This is called *filtering* (or online estimation), and it gradually reduces noise compared to constant noise for $P(Z_t \mid X_t)$.
Another option called *smoothing* (or offline estimation) waits until all the data $\{X_t\}_{t=1,\cdots,T}$ are generated and estimates each Z_t using all the data, i.e. $P(Z_t \mid X_1 = x_1, \cdots, X_T = x_T)$.
A delay can be added to filtering (called *lagged filtering*) to improve accuracy, namely estimating $P(Z_{t-l} \mid X_1 = x_1, \cdots, X_t = x_t)$ at time t where hyper-parameter l is the lag. There is a trade-off between lag and accuracy.
The *prediction* is $P(Z_{t+u} \mid X_1 = x_1, \cdots, X_t = x_t)$ where u is usually 1 or 2, this is used for word auto-filling like when you search using google.

The filtering (the algorithm finding belief state $\alpha_t(k) := P(Z_t = k \mid X_{1:t} = x_{1:t})$ at time t, where $X_{1:t} = x_{1:t}$ is shorthand of $X_1 = x_1, \cdots, X_t = x_t$ for convenience) is accomplished by the *forward algorithm*, which utilises forward probabilities $P(Z_t = k \mid X_{1:t-1} = x_{1:t-1})$. Note by Bayes theorem with conditioning:

$$\boldsymbol{\alpha}_t(k) = P(Z_t = k \mid X_{1:t-1} = x_{1:t-1}, X_t = x_t) \propto P(X_t = x_t \mid Z_t = k, X_{1:t-1} = x_{1:t-1}) P(Z_t = k \mid X_{1:t-1} = x_{1:t-1})$$

the first term $P(X_t = x_t \mid Z_t = k, X_{1:t-1} = x_{1:t}) = P(X_t = x_t \mid Z_t = k) = B_{k,x_t}$ which is the local emission probability at time t. and note by law of total probability,

$$P(Z_t = k \mid X_{1:t-1} = x_{1:t-1}) = \sum_i P(Z_t = k \mid Z_{t-1} = i) P(Z_{t-1} = i \mid X_{1:t-1} = x_{1:t-1}) = \sum_i A_{i,k} \alpha_{t-1}(i)$$

which we can compactly write in vector form as $A_k^T \boldsymbol{\alpha}_{t-1}$ where A_k is the kth column of matrix A. Therefore,

$$\boldsymbol{\alpha}_t \propto B_{x_t} \odot (A^T \boldsymbol{\alpha}_{t-1})$$

where B_{x_t} is the x_tth column of B and $\odot$ is element-wise multiplication.

The full process of the forward algorithm is given below:

- find initial belief state $\boldsymbol{\alpha}_1$ by

$$\boldsymbol{\alpha}_1 =\propto B_{x_1} \odot \boldsymbol{\pi}$$

  where $\boldsymbol{\pi}$ is the initial distribution. Note to deal with proportionality in Bayes theorem, α_t should be normalised at each stage. The normalising constant should be stored if the true values of α_t are required

- At time t, update α by

$$\boldsymbol{\alpha}_t =\propto B_{x_t} \odot (A^T \boldsymbol{\alpha}_{t-1})$$

- Upon reaching time T, return $\boldsymbol{\alpha}_1, \cdots, \boldsymbol{\alpha}_T$.

A similar analysis can be carried out to update the backward probability

$$\beta_t(k) := P(X_{t+1:T} = x_{t+1:T}, Z_t = k)$$

the formulae is given by

$$\boldsymbol{\beta}_{t-1} = A(B_{x_t} \odot \boldsymbol{\beta}_t)$$

It is not hard to see that the smoothing probabilities $P(Z_t = k|X_{1:T} = x_{1:T})$ can be calculated using forward and backward probabilities. Therefore, these forward and backward updates can be used to compute smoothing probabilities. This is called the *Forward-backward algorithm.*

8.5.2 Viterbi Algorithm

Suppose the probabilities A, B are already known, and a sequence of observations $\{X_1, \cdots, X_T\}$, how to find the most possible sequence of latent variables $\{Z_1, \cdots, Z_T\}$. The brute force approach is to find probabilities for all possible sequences of latent variables by multiplying transition and transmission probabilities along the way. For example, the probability of the sequence in figure 29 is

$$A_{Z_1,Z_2} A_{Z_2,Z_3} B_{Z_1,X_1}, B_{Z_2,X_2}, B_{Z_3,X_3}$$

the general formula is

$$P(X_1 = x_1, \cdots, X_T = x_T, Z_1 = z_1, \cdots, Z_T = z_T) = \pi_{z_1} \prod_{t=2}^{T} A_{z_{t-1},z_t} \prod_{t=1}^{T} B_{z_t,x_t}$$

the initial distribution $\pi \in \mathbb{R}^P$ is added to indicate the probabilities of the sequence starting at each value of Z_1. The problem of brute force is that suppose there are P possible POS and m words in the sentence, there are P^m probabilities to check. But since the target is to find the sequence of Z_t with the highest probability, some routes are useless.

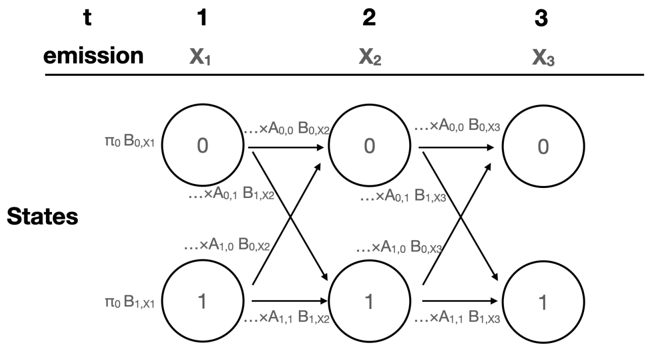


Figure 30: Illustration of a 2-state HMM

Figure 30 shows an HMM with only two states. The probability starts with the initial distribution π_i and initial emission probability B_{i,X_1} for $i = 0, 1$. As you go along each path, you should multiply a transition probability and an emission probability at each step. Note at $t = 2$, there are two paths going to state 0, namely [0, 0], [1, 0]. We can pick the one with a higher probability and cut off the other one. Say we picked [0, 0], no matter what the future states are, the path starting with [0, 0] always has a higher probability than the path starting with [1, 0] as future states are not dependent on Z_1. But we cannot choose between paths ending at different states. For example, we cannot delete one of [0, 1] and [1, 0] with a lower probability. Suppose [0, 1] has a probability 0.5, [1, 0] has a probability 0.1, $A_{1,0} = 0.1, A_{0,0} = 0.9$, then at $t = 3$, [0, 1, 0] has probability $0.5 \times 0.1 = 0.05$ whereas [1, 0, 0] has probability $0.1 \times 0.9 = 0.09$. If you cut off [1, 0] at $t = 2$, you lose the path [1, 0, 0] which has a higher probability. Therefore, at each stage t, we keep one possible path for each state, there are only P^2 paths to check for each t so the complexity is TP^2 compared to P^T for brute force.

The above approach is called the *Viterbi algorithm*. The algorithm is implemented by back-tracking the optimal paths at stage t: keeping a record of the previous state $a_{t,j}$ along the best path at stage t ending with state j (this saves memory compared to keeping track of the whole path), and also calculate the probability $\delta_{t,j}$ indicating the (cumulative) probability of that optimal path. Full process:

- initialise by taking
$$\delta_{1,j} := \pi_j B_{j,X_1}$$
- propagates forward by
$$\delta_{t+1,j} := \max_i \; \delta_{t,i} A_{i,j} B_{j,X_{t+1}}$$
$$a_{t+1,j} = \text{argmax}_i \; \delta_{t,i} A_{i,j} B_{j,X_{t+1}}$$
- takes the endpoint of the path with the highest probability by
$$z_T^* := \text{argmax}_i \; \delta_{T,i}$$
- extract the optimal path $\{z_1^*, \cdots, z_T^*\}$ using the stored $a_{t,j}$:
$$z_{t-1}^* := a_{t,z_t^*}$$

8.5.3 EM algorithm for HMM

Now we know various techniques for inference of HMM (i.e. estimate probabilities for latent variable Z_i from the HMM), we need to tune the parameters (transition and transmission probabilities) to maximise the probability of an observed sequence of emissions. i.e. We need to train the HMM model.

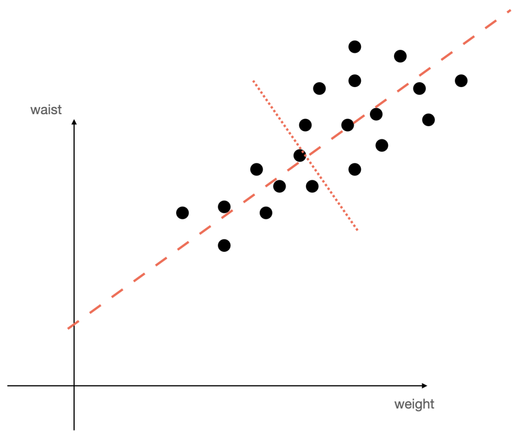
One common way to do that is through the EM algorithm. As you can see from our set of variables, HMM falls into the situation of the EM algorithm. If the latent variables Z_t are given, probabilities A_{ij}, B_{kl} can be estimated simply by counting. For example, estimate of emission probability $\hat{B}_{kl} := N_{kl}^X / N_k$ where N_k is the total number of latent variables Z equalling k and N_{kl}^X counts observations with state k and emission l. If instead of point-wise estimation, you wish to let the emission from state k follow Gaussian mixture models (e.g. when each data point is multi-dimensional), the mean and variance can also be estimated in a similar way to the EM algorithm.

If latent variables are not known, their probabilities based on the given data and parameters can be estimated by
$$\gamma_{t,\boldsymbol{\theta}}(k) := P(Z_t = k \mid X_{1:T} = x_{1:T}^{(i)}, \boldsymbol{\theta})$$
where $x_{1:T}^{(i)}$ is the ith data sequence, then you normalise γ over $t = 1 : T$ and replace Z_t in the estimations described above when Z_t are known with $\gamma_{t,\boldsymbol{\theta}}$ instead. This process is iterated until convergence. Detailed parameter update formulae are given in lecture notes.

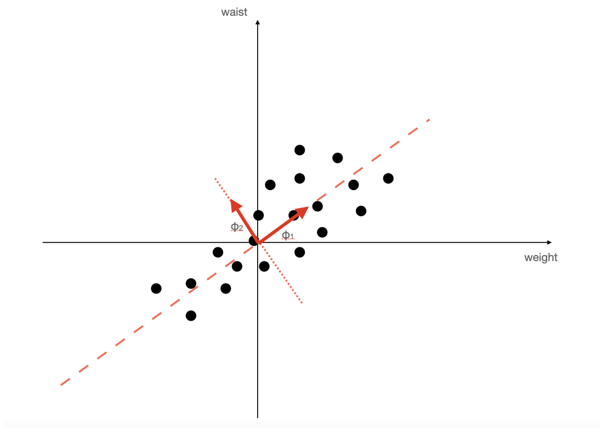
To prevent the EM algorithm from converging to a local minimum, setting multiple random initial guesses for $\gamma_{1,\boldsymbol{\theta}}$ can be necessary. Another choice is to get a small data set with labels first and use that to initialise γ.

8.6 Principal Component Analysis

Suppose you are using waist (circumstance of the waist) and weight of patients as predictors, then somehow one of them is redundant. Because patients with wider waists tend to have a higher weight. In figure 31a, there are arbitrary data points representing patients. You can clearly see that data varies the most along the dashed line, and varies the least along the dotted line. For convenience, we centralise the data and draw the two lines mentioned above across the origin. (see figure 31b) Now, two vectors ϕ_1, ϕ_2 (usually assumed to be unit vectors) are enough to describe the two lines. ϕ_1, ϕ_2 are called the *principal components*, and if we project all data points to the dashed line (the line along ϕ_1), a new data set with dimension reduced by 1 is obtained.



(a) **Original version**



(b) **Original version**

Figure 31: Weights and waist of patients

Now we investigate how the principal components can be found mathematically. Suppose data $\{x^{(i)} \in \mathbb{R}^M : i = 1, \cdots, N\}$ are projected onto the line spanned by given unit vector ϕ (say this line is axis l), the projected points have value $(x^{(i)})^T\phi$ on axis l (i.e. the distance to the origin is $|(x^{(i)})^T\phi|$) Since the data is already centralised, the variance of projected data is simply given by

$$\frac{1}{n}\sum_{i=1}^{n}((x^{(i)})^T\phi)^2 = \frac{1}{n}\sum \phi^T \sum x^{(i)}(x^{(i)})^T\phi = \phi^T\left(\frac{1}{n}x^{(i)}(x^{(i)})^T\right)\phi$$

maximisation of this variance ought to find the Eigenvectors of the matrix $\frac{1}{n}x^{(i)}(x^{(i)})^T$. (this is also derived in lecture notes using the Lagrange multiplier) Suppose we have found the (orthonormal) Eigenvectors $\phi_1, \cdots, \phi_M$ of $\frac{1}{n}x^{(i)}(x^{(i)})^T$ with Eigenvalues $\lambda_1, \cdots, \lambda_M$. Larger λ_i corresponds to higher variance, because

$$\phi_i^T\left(\frac{1}{n}x^{(i)}(x^{(i)})^T\right)\phi_i = \phi_i^T(\lambda_i\phi_i) = \lambda_i$$

. Now, one has to decide how many dimensions of $x^{(i)}$ to keep. Because if we collect all projections to these M Eigenvectors, the dimension is still M. I will borrow figure 10.1 from the lecture notes to explain the choice of m (number of Eigenvectors to keep), see figure 32. The Eigenvalues are sorted and plotted, smaller Eigenvalues correspond to smaller variance, so we discard the smallest ones. In 32 (a) this is easy because of the jump in Eigenvalue. But in reality, Eigenvalues may not have such a jump (32 (b)), and the choice of m may affect the result.

Suppose $m < M$ is already chosen, then we can construct new data $\hat{x}$ using projections

$$\hat{x}^{(i)} := \begin{pmatrix} (x^{(i)})^T\phi_1 \\ (x^{(i)})^T\phi_2 \\ \vdots \\ (x^{(i)})^T\phi_m \end{pmatrix}$$

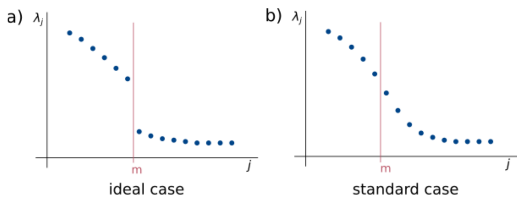


Figure 32: Eigenvalues found from PCA, x-axis is simply index, y-axis represents the Eigenvalues

The lecture notes minimise the distances from data points (the length of dotted lines in figure 33) to projected points instead of maximising variance along the line, and singular value decomposition is used. But both optimisation problems yield the same solution, they are just two different perspectives to view PCA. For more mathematical details, see the lecture notes.

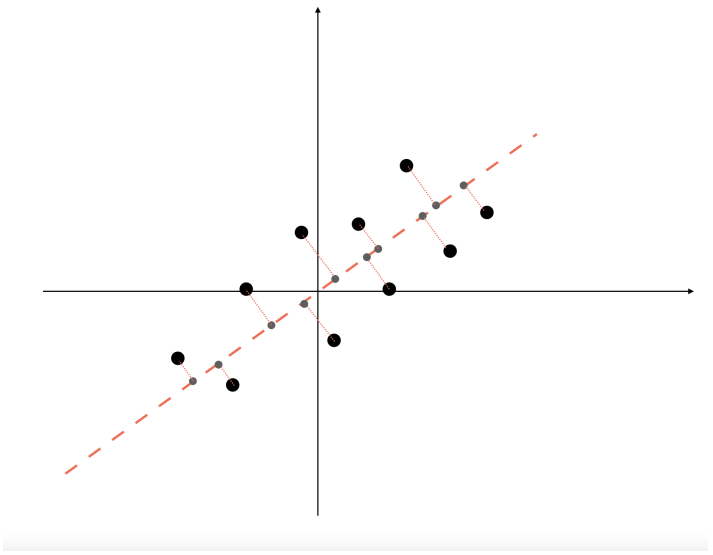


Figure 33: Error minimisation in PCA

8.7 Graph-based partitioning

For basic knowledge of graphs, please refer to the official lecture notes.

Setup: data set $\{x^{(i)}\}_{i=1,\cdots,N}$ where N is number of data points.

If we assume nodes $\{i\}_{i=1,\cdots,N}$ corresponds to data points $x^{(i)}$ and edges are weighted, undirected with weight of edge (i, j) being S_{ij}. (the similarity between $x^{(i)}$, $x^{(j)}$) Then data partitioning and be viewed as cutting some edges to make K disconnected components. (see figure 34) For simplicity, only bi-partitions ($K = 2$) of graphs are considered in this section, and define membership vector $\boldsymbol{s} = (s_i)$ where $s_i \in \{\pm 1\}$ indicates which cluster the data $x^{(i)}$ belongs to.

As mentioned in the clustering section, the quality of clustering depends on similarities within the cluster and dissimilarities between clusters. Spectral partitioning takes the total similarity between clusters (i.e. the total weights of edges that have to be cut to make this clustering) as the loss function, whereas the modularity method concerns both similarities within clusters and dissimilarities between clusters.

So the loss of spectral clustering is given by

$$C = \frac{1}{2} \sum_{i,j \text{ in different clusters}} A_{ij} = \frac{1}{4} \boldsymbol{s}^T L \boldsymbol{s}$$

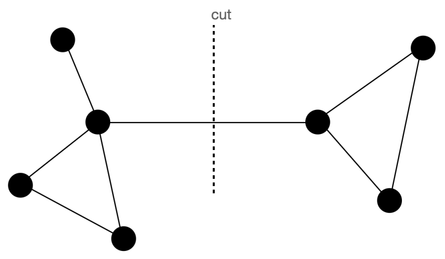


Figure 34: Cutting an edge of graph to partition the graph

(see the derivation of second equality in lecture notes) where A_{ij} is the weight of edge (i, j). But without constraint, the best way is then making no cut, i.e. choose $\boldsymbol{s} = \mathbf{1}$ or $-\mathbf{1}$. Therefore, we need to control $\sum_i s_i$, this indicates the balance of sizes between two clusters. Smaller $\sum_i s_i$ (ideally 0) indicates that the sizes of clusters are roughly equal. Another way to reach balanced clustering is to use the normalised cutting cost

$$\frac{\sum_{i\in S_1, j\in S_2} A_{ij}}{\|S_1\|} + \frac{\sum_{i\in S_2, j\in S_1} A_{ij}}{\|S_2\|}$$

where $\{S_1, S_2\}$ is a partition of the graph.

The best partition $\boldsymbol{s}^* \in \{-1, 1\}^N$ is found by minimising cost

$$\boldsymbol{s}^* = \text{Argmin}_{\boldsymbol{s}\in\{-1,1\}^N} \boldsymbol{s}^T L \boldsymbol{s}$$

with some constraint on $\sum_i s_i$. e.g. if N is even, one can add constraint $\sum_i s_i = 0$ to make size of two clusters exactly the same. But this minimisation problem is NP-hard. We need to change this discrete problem to a continuous problem to use common optimisation techniques, i.e. now we work with $\boldsymbol{s} \in \mathbb{R}^N$ instead of just $\{0, 1\}^N$. But a new constraint should be added: $\boldsymbol{s}^T\boldsymbol{s} = N$, because this is always true for $\boldsymbol{s} \in \{-1, 1\}^N$. i.e. the optimisation problem is

$$\min_{\boldsymbol{s}\in\mathbb{R}^N} \boldsymbol{s}^T L \boldsymbol{s} \quad \text{s.t.} \sum_i s_i = 0, \ \sum_i s_i^2 = N$$

you may change constraint $\sum_i s_i = 0$ to $\sum_i s_i = a$ if you have some preference on the cluster size. This is called the *relaxed problem.*

It is shown in the lecture notes with the Lagrange multiplier that the solution can be found using the spectral connectivity (the second smallest Eigenvalue of the Laplacian L) and Fiedler Eigenvector. A common technique to change solution $\boldsymbol{s}^*$ to the relaxed problem to a vector in $\{-1, 1\}^N$ is to take the sign of each component.

Modularity concerns intra-cluster similarity and inter-cluster dissimilarities. First, define membership matrix $H \in \mathbb{R}^{N\times K}$ where $H_{ik} = 1$ if $x^{(i)}$ is in cluster k and zero otherwise. The expression

$$\frac{1}{2}(H^T A H) \in \mathbb{R}^{k\times k}$$

counts the total weight of edges within the cluster(on the diagonal) and between clusters (off-diagonal). The i, j entry of $\frac{1}{2}H^T AH$ counts weights of edges with one end in cluster i and another end in cluster j. 2 is divided to erase symmetry for counting edges within each cluster, i.e. symmetry of the diagonal entries of the $\frac{1}{2}H^T AH$. So we want values of $\frac{1}{2}H^T AH$ to be concentrated on the diagonal. (i.e. a large trace)

Then we penalise the adjacency matrix A by the matrix R where R_{ij} is probabilities of nodes i, j being connected if the graph edges are randomised (but keeping the degree of each node is preserved). R_{ij} is given by $d_i d_j / 2E$

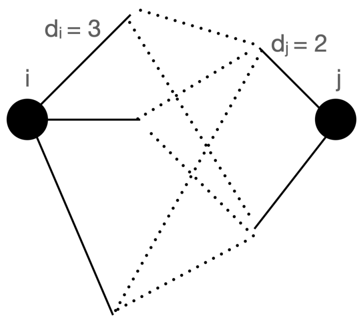


Figure 35: Finding fraction of edges between i, j in randomised graph

where E is the number of edges. $d_i d_j$ is total number of ways that i, j may be reconnected, see figure 35.

So modularity is defined as

$$Q := \frac{1}{2E}\text{tr}\left[H^T \underbrace{\left(A - \frac{1}{2E}\boldsymbol{d}\boldsymbol{d}^T\right)}_{=:Z} H\right]$$

where Z is called the modularity matrix.
Modularity method: solve the optimisation problem $\max_H Q$. Advantage of modularity: no need to determine the number of clusters K, already encoded in matrix H.

The optimisation problem $\max_H Q$ is solved by *Louvain algorithm*. It is hierarchical clustering that gradually merges nodes until all nodes are merged into one community, or Q is not improving. Each iteration of the Louvain algorithm first finds, for each node, the best new cluster assignment that maximises the modularity gain ΔQ. It has been shown that the order of changing assigned clusters for nodes does not matter. This is repeated until $\Delta Q \leq 0$. Then each cluster is merged into a single node, and the weights of new edges are given by the sum of edges within each cluster (weights for self-connecting edges) and the sum of edges between clusters i, j. The two steps given above are then repeated on the new graph.

We discussed two popular bi-partitions of graphs above, but we can partition a graph into more pieces. After bi-partition, you can further bi-partition one or both of the clusters to get more clusters. This is called sequential bi-partitions. (see figure 36)

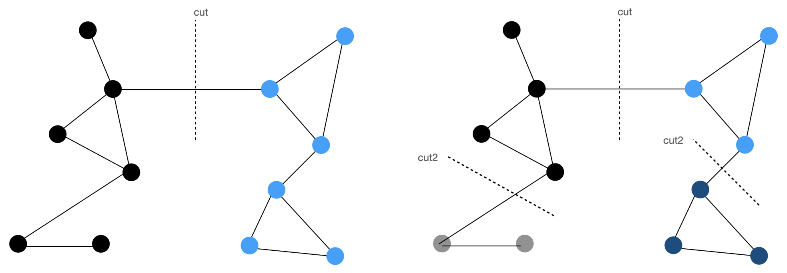


Figure 36: sequential partitioning

9 Cheat Sheet

9.1 Basics

Basic procedure of Supervised Learning

Symbol	**Name**	**Side note/Definition**
Name	Symbol	Definition
sample size	N	size of the training set
training set	$\{x^(i), y^{(i)}\}_{i=1}^N$	set of observed data used for training model
relation function	f	proposed model of function relating x, y, i.e. $y = f(x)$
Estimated model	$\hat{f}$	Estimation of function f based on the training set
Loss function	$L(y, \hat{f}(x))$	error between the estimated model and the ground truth
Mean sample loss	$E(L(y, f(x)))$	$\frac{1}{N}\sum_{i=1}^N L(y^{(i)}, f(x^{(i)}))$
Test data	$x^{\text{in}}, y^{\text{in}}$	new pair of data, to test the model
Expected test loss		$E(L(y^{\text{in}}, f(x^{\text{in}})))$
Mean squared error	$L_{\text{MSE}}(y, f(x))$	$\frac{1}{N}(y^{(i)} - f(x^{(i)}))^2$
generalisability		generalisable model has small mean sample loss and small expected test loss as well
hyper-parameters		specify the overall structure of the model, will be fixed during training

An estimator with small variance: reliable
An estimator with small bias: accurate
Variance-bias trade-off: in general, if the variance is reduced, the bias becomes larger. And if bias is kept small (even 0), then the variance tends to be larger.

local model

$$\hat{y} = \hat{f}_{\mathcal{N}(\boldsymbol{x}^{\text{in}})}(\boldsymbol{x}^{\text{in}})$$

the model depends on the input $\boldsymbol{x}^{\text{in}}$, where $\mathcal{N}(\boldsymbol{x}^{\text{in}})$ means the k nearest neighbours in kNN algorithm.
global model

$$\hat{y} = \hat{f}(\boldsymbol{x}^{\text{in}})$$

$\hat{f}$ does not depend on $\boldsymbol{x}^{\text{in}}$.

Vector differentiation rules
Assume $\boldsymbol{\beta}$ is the independent variable, A is a matrix

- $\nabla_\beta(\alpha^T\beta) = \nabla_\beta(\beta^T\alpha) = \alpha$
- $\nabla_\beta(\beta^T A\beta) = (A + A^T)\beta$
- If $\vec{f}(\boldsymbol{\beta}) = \begin{pmatrix} f_1(\boldsymbol{\beta}) & \cdots & f_m(\boldsymbol{\beta}) \end{pmatrix}^T$, then

$$\nabla_\beta(\vec{f}) = \begin{pmatrix} \nabla_\beta(f_1) \\ \vdots \\ \nabla_\beta(f_m) \end{pmatrix}$$

- $\nabla_\beta(A\beta) = A^T$

Other differentiation rules

- $\frac{d}{dt}\langle v(t), w(t)\rangle = \langle v'(t), w(t)\rangle + \langle v(t), w'(t)\rangle$
- $\frac{d}{dt}\left(v(t) \times w(t)\right) = v'(t) \times w(t) + v(t) \times w'(t)$.

Statistics
MSE and variance, bias: for parameter θ and its estimator θ^*,

$$E[(\theta - \theta^*)^2] = \mathrm{Var}\theta^* + (\theta - E(\theta^*))^2$$

T-fold cross validation

Data	$\mathcal{S}$	$\{\boldsymbol{x}^{(i)}, y^{(i)}\}$
Number of folds	T	a hyper parameter
Validation subset	$\mathcal{S}_t$	$\lvert\mathcal{S}_t\rvert = \lvert S\rvert/T$, $\mathcal{S} = \bigcup_{t=1}^T \mathcal{S}_t$, $\mathcal{S}_t$ disjoint
training subset	$\overline{\mathcal{S}_t}$	$\overline{\mathcal{S}_t} = \mathcal{S} - \mathcal{S}_t$
Validation model	$\hat{f}_{\overline{\mathcal{S}_t}}$	model trained using $\overline{\mathcal{S}_t}$
error(of one fold)	MSE_t	$\sum_{i\in\mathcal{S}_t}\left[\hat{f}_{\overline{\mathcal{S}_t}}(\boldsymbol{x}^{(i)}) - y^{(i)}\right]^2$
average MSE	$\langle\mathrm{MSE}\rangle$	$\frac{1}{T}\sum_{t=1}^T \mathrm{MSE}_t$

Quality measure for classifiers
Case $\mathcal{C}_q = \{0, 1\}$, use confusion matrix:

	$y = 1$	$y = 0$
$\hat{y} = 1$	TP(true positive)	FP(false positive)
$\hat{y} = 0$	FN(false negative)	TN(true negative)
column sum	P	N

True positive rate(TPR)/sensitivity/recall

$$\frac{\mathrm{TP}}{\mathrm{P}} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

True negative rate(TNR)/specificity

$$\frac{\mathrm{TN}}{\mathrm{N}} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}}$$

Accuracy

$$\frac{\mathrm{TP} + \mathrm{TN}}{N_{\mathrm{validation}}}$$

$N_{\mathrm{validation}} := \mathrm{TP} + \mathrm{FN} + \mathrm{TN} + \mathrm{FP}$.
Precision

$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

F-score

$$F := 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}$$

two important curves:
ROC(receiver operating characteristic) plot of TPR against FPR := 1− TNR for different values of threshold.
Precision against the recall for different values of threshold.

9.2 Linear Regression

Ordinary Least Square(OLS)
Table of basic symbols

Symbol	**Name**	**Side note/Definition**
Name	Symbol	Definition
	p	number of parameters/coefficients
parameters	$\boldsymbol{\beta}$	vector in $\mathbb{R}^{p+1}$, unknown, to be found
	$\{y^{(i)}\}_{i=1}^{N}$	observed variables, assumed to be univariate, i.e. $y^{(i)} \in \mathbb{R}$, all $y^{(i)}$ often compactly written as a vector $\boldsymbol{y}$
covariates	$\{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$	assumed to be independent variables, $x^{(i)} \in \mathbb{R}^{p+1}$
prediction	$\{\hat{y}^{(i)}\}_{i=1}^{N}$	$\hat{y}^{(i)} = f_{\mathrm{LR}}(\boldsymbol{x}^{(i)}, \boldsymbol{\beta}) := \boldsymbol{x}^{(i)T}\boldsymbol{\beta}$
covariate matrix	X	$= (x_j^{(i)})$, a matrix in $\mathbb{R}^{N\times(p+1)}$
error	$\boldsymbol{e}$	$:= \boldsymbol{y} - X\boldsymbol{\beta}$
Mean squared error	$L(f_{\mathrm{LR}}(\boldsymbol{x}), y)$	$:= \frac{1}{N}\boldsymbol{e}^T\boldsymbol{e}$
loss function (for LR)	$L(\boldsymbol{\beta})$	$:= \frac{1}{N}(\boldsymbol{y} - X\boldsymbol{\beta})^T(\boldsymbol{y} - X\boldsymbol{\beta})$
Moore-Penrose pseudo-inverse	X^+	$:= (X^TX)^{-1}X^T$

$$\text{(model)} \quad y = X\beta$$

The statistical assumption:

$$\text{(model with randomness)} \quad y = X\beta + \epsilon, \quad \epsilon \sim N(0, \sigma^2 I_n)$$

Least square solution: (this is also the MLE of $\boldsymbol{\beta}$)

$$\boldsymbol{\beta}^* = (X^TX)^{-1}X^T\boldsymbol{y} = X^+\boldsymbol{y}, \quad \text{hessian of loss function } \; H(L) = \frac{2}{N}(X^TX)$$

$$\mathrm{Bias}(\beta^*) = 0, \quad \mathrm{Var}(\beta^*) = E((\boldsymbol{\beta} - \boldsymbol{\beta}^*)(\boldsymbol{\beta} - \boldsymbol{\beta}^*)) = (X^TX)^{-1}\sigma^2$$

Estimated output

$$\hat{\boldsymbol{y}} := X\hat{\boldsymbol{\beta}} = X(X^TX)^{-1}X^T\boldsymbol{y}$$

normal equation condition

$$X^T(\boldsymbol{y} - X\boldsymbol{\beta}) = 0$$

Unbiased estimator for σ^2

$$\hat{\sigma}^2 := \frac{1}{N-(p+1)}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2$$

Numerical methods:
Gradient Descent to minimise $L(\boldsymbol{\beta})$: set initial $\boldsymbol{\beta}_0$, step size η_k for $k \in \mathbb{N}$, then follow iterations

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \eta_k \nabla_\beta(L(\boldsymbol{\beta}_k))$$

Newton's method to minimise $L(\boldsymbol{\beta})$: set initial $\boldsymbol{\beta}_0$, then follow iterations

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - H(\boldsymbol{\beta}_k)^{-1}\nabla L(\boldsymbol{\beta}_k)$$

where $H(\beta)$ is the Hessian matrix of $L(\boldsymbol{\beta})$.

Variance Reduction Methods:
Brute Force complete enumeration

- take all subsets of covariates with size k, find the best one
- Modification Leaps and bounds possible
- Drawback: k hard to pick; computationally expensive; combinatorial expansion

Greedy sequential selection:

- **Forward**: start from the constant model $y = \beta_0$, add covariant $\boldsymbol{x}_i$ one at a time by choosing β_i s.t. error reduction is maximised.
- **Backward**: starts with the full model with p parameters, discarding one by one s.t. the error increment is minimal.
- Both are implemented using the QR algorithm.
- stopping criteria chosen based on heuristic(experience)
- Drawback: hard to choose the stopping criteria, combinatorial expansion

Shrinkage method: Ridge Regression
Penalty added to $\boldsymbol{\beta}$:

$$L_{\text{Ridge}}(\boldsymbol{\beta}) = \|\boldsymbol{y} - X\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|_2^2$$

minimisation solution:

$$\boldsymbol{\beta}^*_{\text{Ridge}} = (X^TX + \lambda I)^{-1}X^T\boldsymbol{y}$$

$$\text{bias}(\beta^*_{\text{Ridge}}) = -\lambda(X^TX + \lambda I)^{-1}\boldsymbol{\beta}, \quad \text{Var}(\beta^*_{\text{Ridge}}) = \sigma^2 V\mathcal{P}V^T$$

$$\text{the diagonal matrix } \mathcal{P} := (D + \lambda I)^{-1}D(D + \lambda I)^{-1} \text{ with } \mathcal{P}_{ii} = \frac{d_i}{(d_i + \lambda)^2}$$

D, V comes from the Eigendecomposition, $X^TX = VDV^T$.
Asymptotic behaviour: $\lambda \to \infty$: $\text{Var}(\beta^*_{\text{Ridge}}) \to 0$, $\text{bias}(\beta^*_{\text{Ridge}}) \to -\boldsymbol{\beta}$

LASSO regression:

$$L_{\text{LASSO}}(\boldsymbol{\beta}) = \|\boldsymbol{y} - X\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|_1^2$$

no analytical solution, numerical methods are required.

Nolinear regression

$$L_{\text{nonlin}}(\boldsymbol{\beta}) = (h_m(\boldsymbol{x}))^T\boldsymbol{\beta}$$

where h_m is a function that may not be linear.

9.3 Optimisation

Equality constraint

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}), \quad \text{s.t. } g_i(\boldsymbol{x}) = 0 \text{ for } i = 1, \cdots, m$$

use the Lagrange multiplier:

$$\mathcal{L} := f(\boldsymbol{x}) + \sum_{i=1}^{m} \alpha_i g_i(\boldsymbol{x})$$

Minimiser: $\boldsymbol{x}, \boldsymbol{\alpha}$ s.t. $\nabla_{\boldsymbol{x}} \mathcal{L} = 0,\ \nabla_{\boldsymbol{\alpha}} \mathcal{L} = 0$.

Linear inequality constraint KKT conditions used

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}), \quad \text{s.t. } g_i(\boldsymbol{x}) \leq 0 \text{ for } i = 1, \cdots, m$$

the optimum point $\boldsymbol{x}^*$ satisfies the KKT conditions:

$$\mathcal{L} = f(\boldsymbol{x}^*) + \sum_{i=1}^{m} \alpha_i g_i(\boldsymbol{x}^*) = 0$$

$$\forall i,\ g_i(\boldsymbol{x}^*) \leq 0$$

$$\forall i,\ \alpha_i \geq 0$$

$$\forall i,\ \alpha_i g_i(\boldsymbol{x}^*) = 0$$

9.4 Classification problems

When the outcomes y are in $\mathcal{C}_q := \{c_1, \cdots, c_q\}$ where c_i can be viewed as a class, then we are considering a classification problem with q classes.

9.4.1 k-nearest neighbours

kNN with equal weights, continuous case

$$\hat{f}_{\mathcal{N}_k(\boldsymbol{x}^{\text{in}})}(\boldsymbol{x}^{\text{in}}) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\boldsymbol{x}^{\text{in}})} y^{(i)}$$

where $\mathcal{N}_k(\boldsymbol{x}^{\text{in}})$ is the set of k nearest neighbours to $\boldsymbol{x}^{\text{in}}$.
kNN, discrete case
Use Hamming distance to find neighbours, and pick $\hat{f}_{\mathcal{N}_k(\boldsymbol{x}^{\text{in}})}(\boldsymbol{x}^{\text{in}})$ using the majority rule.

The distance used in kNN:

$$\text{(Euclidean)} \quad \|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|_2$$

when $\boldsymbol{x}^{(i)}$ are discrete,

$$\text{(Hamming)} \quad d_H(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \sum_{m=1}^{p} I(x_m^{(i)} x_m^{(j)} \neq x_m^{(j)})$$

9.4.2 Logistic regression

In this case, $q = 2$, $C_q = \{0, 1\}$.

log-odds: if $y \in \{0, 1\}$,

$$\text{log odds } = \log \frac{P(y=1)}{P(y=0)}$$

logistic function,

$$h(x) = \frac{1}{1 + e^{-x}}, \quad h^{-1}(y) = \log\left(\frac{y}{1-y}\right)$$

logistic linear model

$$\boldsymbol{x}^T \boldsymbol{\beta} = \log\left(\frac{P(y=1)}{1 - P(y=1)}\right), \quad \text{i.e.} P(y=1) = h(\boldsymbol{x}^T \boldsymbol{\beta}) =: h_{\boldsymbol{\beta}}(\boldsymbol{x})$$

the full model:

$$P(Y = y \,|\, \boldsymbol{x}, \boldsymbol{\beta}) = (h_{\boldsymbol{\beta}}(\boldsymbol{x}))^y (1 - h_{\boldsymbol{\beta}}(\boldsymbol{x}))^{1-y} \quad y \in \{0, 1\}$$

Loss function (log-likelihood)

$$\mathcal{L} = \sum_{i=1}^{N} y^{(i)} \log h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)}) + (1 - y^{(i)}) \log \Big(1 - h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)})\Big)$$

Normal equation:

$$\nabla_{\boldsymbol{\beta}}\, \mathcal{L}|_{\boldsymbol{\beta}^*_{\log}} = \sum_{i=1}^{N} \Big(y^{(i)} - h_{\boldsymbol{\beta}^*_{\log}}(\boldsymbol{x}^{(i)})\Big)\, \boldsymbol{x}^{(i)} = \mathbf{0}$$

solution $\boldsymbol{\beta}^*_{\log}$ to this equation will be the logistic estimator.

Classification threshold τ (a hyper-parameter):

$$P(y = 1 \,|\, \boldsymbol{x}^{\text{in}}) > \tau \Rightarrow \boldsymbol{x}^{\text{in}} \in \{1\}$$

9.4.3 Naive Bayes's method

Suppose possible values of y(classes) are $\mathcal{C} := \{c_1, \cdots, c_Q\}$ The probability vector is

$$\boldsymbol{\pi}^{(NB)} := \begin{pmatrix} \pi_1^{(NB)} \\ \vdots \\ \pi_Q^{(NB)} \end{pmatrix}$$

$$\text{where } \pi_q^{(NB)} := P(Y = c_q \,|\, X = \boldsymbol{x}^{\text{in}}) = \frac{P(Y = c_q)P(X = \boldsymbol{x}^{\text{in}}|Y = c_q)}{\sum_{k=1}^{Q} P(Y = c_k)P(X = \boldsymbol{x}^{\text{in}}|Y = c_k)}$$

Estimation of the probabilities in the RHS
$P(Y = c_q)$:

$$P(Y = c_q) = \frac{\sum_{i=1}^{N} I(y^{(i)} = y_q)}{N}$$

Or with Laplace smoothing:

$$P(Y = c_q) = \frac{\sum_{i=1}^{N} I(y^{(i)} = y_q) + \lambda}{N + Q\lambda}$$

$P(X = \boldsymbol{x}|Y = c_k)$:
Discrete case

$$P(X_j = x_j \,|\, Y = c_q) = \frac{\sum_{i=1}^{N} I(y^{(i)} = c_q, x_j^{(i)} = x_j)}{\sum_{i=1}^{N} I(y^{(i)} = c_q)}$$

where I is the indicator function. Or with Laplace smoothing:

$$P(X_j = x_j \,|\, Y = c_q) = \frac{\sum_{i=1}^{N} I(y^{(i)} = c_q, x_j^{(i)} = x_j) + 1}{\sum_{i=1}^{N} I(y^{(i)} = c_q) + A_j}$$

where A_j is the number of possible values of X_j.
Continuous case

$$P(X_j = x_j \,|\, Y = c_q) = \frac{1}{\sqrt{2\pi\sigma_{j,q}^2}} e^{\frac{-(x_j - \mu_{j,q})^2}{2\sigma_{j,q}}}$$

where

$$\mu_{j,k} = \frac{1}{N_k} \sum_{y^{(i)} = c_k} x_j^{(i)} \quad N_k \text{ is total number of samples in class } c_k$$

$$\sigma_{j,k}^2 = \frac{1}{N_k - 1} \sum_{y^{(i)} = c_k} x_j^{(i)}(x_j^{(i)} - \mu_{j,k})^2$$

9.4.4 Decision Tree

Input $\boldsymbol{x}^{(i)} = (x_1^{(i)}, \cdots, x_p^{(i)})$
Output $y^{(i)} \in \{c_1, \cdots, c_Q\}$ (classification)
Separation region R_α: a region at some depth of the decision tree

Separation Quality Measures
Impurity of node α:

$$\boldsymbol{\pi}(R_\alpha) = \begin{pmatrix} \pi_1(R_\alpha) \\ \vdots \\ \pi_Q(R_\alpha) \end{pmatrix}, \qquad \text{where } \pi_q(R_\alpha) = \frac{\sum_{i=1}^N I(\boldsymbol{x}^{(i)} \in R_\alpha,\ y^{(i)} = c_q}{)} \sum_{i=1}^N I(\boldsymbol{x}^{(i)} \in R_\alpha)$$

Gini index:

$$\text{GI}[\boldsymbol{\pi}(R_\alpha)] = \sum_{q=1}^Q \pi_q(R_\alpha)(1 - \pi_q(R_\alpha))$$

Cross Entropy

$$\text{CE}[\boldsymbol{\pi}(R_\alpha),\ \boldsymbol{\pi}(R_\alpha)] = -\sum_{q=1}^Q \pi_q(R_\alpha) \log(\pi_q(R_\alpha))$$

The algorithm works by finding a separation threshold s and separation axis j (i.e. divide data into two groups: $\boldsymbol{x}_j^{(i)} > s$, $\boldsymbol{x}_j^{(i)} \leq s$) that minimises Gini index or cross-entropy, then the data is separated. For each sub-group, repeat this process. Stop when the impurity of all regions is low enough.

Prediction:
for classification, $\hat{y} = \text{argmax}_q \pi_q(R_\alpha(\boldsymbol{x}^{in}))$ where $R_\alpha(\boldsymbol{x}^{in})$ is the region $\boldsymbol{x}^{in}$ belongs to in the decision tree.
for regression, $\hat{y} = \overline{y}_{R_\alpha(\boldsymbol{x}^{in})}$.

Random forest
Build B trees, each trained using a bootstrap sample from the training set, and $\tilde{p}$ (Rule of thumb: $\tilde{p} \approx p/3$ for regression, $\tilde{p} \approx \sqrt{p}$ for classification) random features are chosen out of p features. For new data, predict using all trees and aggregate:
Classification: average the probability vector $\boldsymbol{\pi}$, and use the majority rule.
Regression: Average of predicted values from all trees.

9.4.5 Support Vector Machine

Input $\boldsymbol{x}^{(i)} = (x_1^{(i)}, \cdots, x_p^{(i)})$
Output $y^{(i)} \in \{\pm 1\}$ (binary classification)
Separation hyper-plane $\boldsymbol{x} \cdot \boldsymbol{w} + b = 0$ where $\boldsymbol{w}, b$ are parameters to be decided
Classification width:

$$(\boldsymbol{x}_+ - \boldsymbol{x}_-) \cdot \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}$$

where support vectors $\boldsymbol{x}_+, \boldsymbol{x}_-$ are the closest data point of class $+1, -1$ from the separation hyper-plane respectively.

Hard Classification Optimisation problem

$$\min_{\boldsymbol{w}} \frac{1}{2}\|\boldsymbol{w}\|^2, \qquad \text{s.t. } y^{(i)}(\boldsymbol{x}^{(i)} \cdot \boldsymbol{w} + b) \geq 1 \ \forall i$$

Solutions
The Lagrangian at the optimal point:

$$\mathcal{L}(\alpha) = \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,\,j=1}^N \alpha_i \alpha_j y^{(i)} y^{(j)} (\boldsymbol{x}^{(i)} \cdot \boldsymbol{x}^{(j)})$$

Lagrangian is independent of $\boldsymbol{w}^*$.
The conditions obtained from the dual problem:

$$\sum_{i=1}^{N} y^{(i)}\alpha_i = 0$$

First, find $\alpha_i \geq 0$ by maximising the Lagrangian subjected to the condition above and then

$$\boldsymbol{w}^* = \sum_{i=1}^{N} \alpha_i y^{(i)} \boldsymbol{x}^{(i)} = \alpha_+ \boldsymbol{x}_+ - \alpha_- \boldsymbol{x}_-$$

$$b = 1 - \boldsymbol{x}_+ \cdot \boldsymbol{w}^* = 1 - \alpha_+ \boldsymbol{x}_+ + \alpha_- \boldsymbol{x}_-$$

Soft Classification Optimisation Problem

$$\min_{\boldsymbol{w}} \frac{1}{2}\|\boldsymbol{w}\|^2 + \lambda \sum_{i=1}^{N} \zeta^{(i)}, \quad \text{s.t. } y^{(i)}(\boldsymbol{x}^{(i)} \cdot \boldsymbol{w} + b) \geq 1 - \zeta^{(i)} \; \forall i$$

where slackness(hinge loss function) $\zeta^{(i)} := \max\left(0, 1 - (\boldsymbol{x}^i \cdot \boldsymbol{w} + b) y^{(i)}\right)$.

Kernel
Meaning of kernel: if $\phi : \mathbb{R}^d \to \mathbb{R}^D$ is a transformation (usually assume $d < D$),

$$k(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{y})$$

but kernel function only uses $\boldsymbol{x} \cdot \boldsymbol{y}$, it will not compute $\phi(\boldsymbol{x}), \phi(\boldsymbol{y})$.

The optimisation problem for kernel SVM: define $\boldsymbol{z} := \phi(\boldsymbol{x})$

$$\min_{\boldsymbol{w} \in \mathbb{R}^D} \frac{1}{2}\|\boldsymbol{w}\|^2, \quad \text{s.t. } y^{(i)}(\boldsymbol{z}^{(i)} \cdot \boldsymbol{w} + b) \geq 1 \; \forall i$$

plugging the kernel

$$\min_{\boldsymbol{w} \in \mathbb{R}^D} \frac{1}{2} k(\boldsymbol{w}, \boldsymbol{w}), \quad \text{s.t. } y^{(i)}(k(\boldsymbol{z}^{(i)}, \boldsymbol{w}) + b) \geq 1 \; \forall i$$

the Lagrangian with kernel

$$\mathcal{L}(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y^{(i)} y^{(j)} k(\boldsymbol{z}^{(i)}, \boldsymbol{z}^{(j)})$$

Commonly used kernels

$$k(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x} \cdot \boldsymbol{y} + 1)^n \quad \text{(polynomial)}$$

$$k(\boldsymbol{x}, \boldsymbol{y}) = e^{-\|\boldsymbol{x} - \boldsymbol{y}\|^2/\sigma} \quad \text{(Gaussian radial basis)}$$

$$k(\boldsymbol{x}, \boldsymbol{y}) = \tanh(\beta(\boldsymbol{x} \cdot \boldsymbol{y}) + c) \quad \text{(sigmoid)}$$

9.5 Graphs-based learning

Data matrix: $X \in \mathbb{R}^{N \times p}$. p: number of features, N: number of data.
$\boldsymbol{x}^{(i)}$: ith data point (ith row of X)

Similarities:

$$\text{(cosine similarity)} S_{ij} = \frac{\boldsymbol{x}^{(i)} \cdot \boldsymbol{x}^{(j)}}{\|\boldsymbol{x}^{(i)}\| \|\boldsymbol{x}^{(j)}\|}$$

$$\text{(statistical similarity)} S_{ij} = \frac{\text{cov}(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})}{\sqrt{\text{Var}(\boldsymbol{x}^{(i)})}\sqrt{\text{Var}(\boldsymbol{x}^{(j)})}}$$

Dissimilarities:

$$\text{(Euclidean distance)} D_{ij} = \|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|$$

$$\text{(kernel distance)} D_{ij} = \exp\left(-\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|^2 / c\right)$$

9.5.1 Graphs

$G(V, E)$ where V is a set of vertices, and E is a set of edges (each edge defined as a pair (i, j)). E can also refer to the number of edges.
Undirected graph: edges (i, j) are unordered pairs. Directed graph: edges (i, j) are ordered.
Weighted graphs: weights w_{ij} given to each edge (i, j). If the graph is unweighted, let $w_{ij} \equiv 1$.
Connected graph: there is a possible path between every pair of nodes. Disconnected graphs can have many connected components (clusters).
Strongly connected graph: there is a possible DIRECTED path between every pair of nodes.
Complete graph: every pair of nodes is connected by an edge.

Matrices of graphs
Adjacency matrix $A \in N \times N$: $A_{ij} = w_{ij}\delta_{(i,j)\in E}$.
In-degree vector $\boldsymbol{d}_{\text{in}} = A\mathbf{1}$ (number of edges sourced from each node) where $\mathbf{1}$ is vector of all ones.
Out-degree vector $\boldsymbol{d}_{\text{out}} = A^T\mathbf{1}$ (number of edges ending at each node)
Degree matrix $D := \text{diag}(\boldsymbol{d}) = \text{diag}(A\mathbf{1})$.
Laplacian $L := -A + D$. This is the discrete version of the operator ∇^2.
Incidence matrix $B_{inc} \in \mathbb{R}^{E\times N}$: each row contains two 1s that indicate two ends of an edge. Or a pair of -1, 1 for directed graph.
$B^T B = L$ where B is the incidence matrix.
Importance of Laplacian: it describes the graph fully,

- There is always an Eigenvalue 0 with Eigenvector $\mathbf{1}$. For connected graph, $\mathbf{1}$ is the only eigenvector of 0
- If graph has k disconnected components, L can be rearranged into block-diagonal form (with k blocks)
- L is positive semi-definite, so Eigenvalues are non-negative.
- L is symmetric for undirected graph
- If $\boldsymbol{v}_i$ are Eigenvectors of L with Eigenvalues λ_i (s.t. $0 \le \lambda_1 \le \lambda_2 \le \cdots$), then solution to discrete heat equation $\frac{d\boldsymbol{u}}{dt} = -L\boldsymbol{u}$ is
$$\boldsymbol{u}(t) = e^{-tL}\boldsymbol{u}(0) = \sum_{i=1}^{N} [\boldsymbol{v}_i \cdot \boldsymbol{u}(0)] e^{-\lambda_i t}\boldsymbol{v}_i$$
limiting solution: $\lim_{t\to\infty} \boldsymbol{u}(t) = N\langle \boldsymbol{u}(0)\rangle \mathbf{1}$ where $\langle\cdot\rangle$ is the mean of vector. convergence rate to limiting solution $\propto e^{-\lambda_2 t}\boldsymbol{v}_2$.

Spectral connectivity: λ_2, *Fiedler Eigenvector*: $\boldsymbol{v}_2$. Determines the behaviour of the solution to the discrete heat equation.
Transition matrix $M := AD^{-1}$, describes evolution of random walk
Random walk Laplacian: $L_{\text{RW}} = I - AD^{-1} = LD^{-1}$, describes
normalised Laplacian $\mathcal{L} := D^{-1/2}LD^{-1/2} = D^{-1/2}L_{\text{RW}}D^{1/2}$. It is symmetric, and $\mathcal{L}$ is isospectral (has the same set of Eigenvalues counting multiplicities) with L_{RW}.

Graph distances
geodesic distance: length of the shortest path connecting i, j.
Average distance: average length of all paths connecting i, j.
the "length" here is defined as the sum of the weights of edges on that path.

Markov processes on graphs
Discrete-time random walk: if $\boldsymbol{p}_t \in \mathbb{R}^N$ is the vector containing the probabilities of the random walk being at each node at time t
$$\boldsymbol{p}_{t+1} = (AD^{-1})\boldsymbol{p}_t =: M\boldsymbol{p}_t$$
solution: $\boldsymbol{p}_t = M^t\boldsymbol{p}_0$.

Continuous-time random walk:
$$\frac{\boldsymbol{p}(t)}{dt} = -L_{\text{RW}}\boldsymbol{p}_t$$

9.5.2 graph-based clustering

Convert data X to graph: $V := \{i\}_{i=1}^N$, use similarity S or dissimilarity D as adjacency matrix. (this graph will be weighted and complete)

Sparsification:

- Thresholding: remove all edges with weights below a chosen threshold τ.
- ϵ-ball: only keep edges connecting two points within distance ϵ from each other.
- kNN: only keep edges for k nearest neighbours for each node.
- use MST: minimum spanning tree(MST) is a tree with a subset of edges that connects all N nodes with a minimum total weight of edges. Effective edges are added according to MST.

Spectral decomposition:
Bi-partition: assume there are only two clusters S_1, S_2
Indicator vector $\boldsymbol{s} = (s_i)_{i=1,\cdots,N}$ and difference t_{ij}:

$$s_i = \begin{cases} 1, & \text{if } i \in S_1 \\ -1, & \text{if } i \in S_2 \end{cases}, \quad t_{ij} := \frac{1}{2}(1 - s_i s_j) = \begin{cases} 1, & \text{if } i, j \text{ are in different clusters} \\ 0, & \text{if } i, j \text{ are in the same cluster} \end{cases}$$

Cost function (total weights of edges that have to be cut to split the graph)

$$C = \frac{1}{2} \sum_{i \in S_1,\, j \in S_2} A_{ij} = \frac{1}{4} \sum_{i,j} t_{ij} A_{ij} = \frac{1}{4} \boldsymbol{s}^T L \boldsymbol{s}$$

The optimisation problem (relaxed from discrete to continuous)

$$\min_{\boldsymbol{s} \in \mathbb{R}^N} \boldsymbol{s}^T L \boldsymbol{s} \quad \text{s.t. } \boldsymbol{s}^T \boldsymbol{s} = N, \boldsymbol{s}^T \mathbf{1} = n_1 - n_2$$

the first constraint corresponds to the fact that there are N points in total, and the second corresponds to the difference between the two clusters.

The solution is given by $\boldsymbol{s}^*_{relax} = \boldsymbol{v}_2 + \frac{n_1 - n_2}{N} \mathbf{1}$ where $\boldsymbol{v}_2$ is the Fiedler eigenvector. Optimal cost:

$$C^* = \lambda_2 \frac{n_1 n_2}{N}$$

where λ_2 is the spectral/algebraic connectivity (the second smallest Eigenvalue of L). Smaller λ_2: good bi-partition quality. Closest vector $\boldsymbol{s}^* \in \{-1, +1\}^N$ to $\boldsymbol{s}^*_{relax}$:

$$\max_{\boldsymbol{s} \in \{-1,+1\}^N} \boldsymbol{s}^T \boldsymbol{s}^*_{relax} = \max_{\boldsymbol{s} \in \{-1,+1\}^N} \boldsymbol{s}^T \boldsymbol{v}_2 + \frac{(n_1 - n_2)^2}{N}$$

Solution:

- If n_1, n_2 are known: align the largest n_1 elements of $\boldsymbol{v}_2$ with cluster 1, and the rest to cluster -1.
- If n_1, n_2 are unknown: assign all elements with $[\boldsymbol{v}_2]_i > 0$ to cluster 1 and assign elements with $[\boldsymbol{v}_2]_i < 0$ to cluster -1.

Modularity clustering Membership matrix $H \in \mathbb{R}^{N \times K}$ (K is number of clusters) where $H_{ik} = 1$ if $x^{(i)}$ is in cluster k and 0 otherwise.
Edge count matrix: $\frac{1}{2} H^T A H$.
Optimisation problem

$$\max_H \operatorname{tr}[H^T A H]$$

Modularity loss:

$$Q = \frac{1}{2E} \operatorname{tr}\left[H^T \left(A - \frac{1}{2E} \boldsymbol{d}\boldsymbol{d}^T \right) H \right]$$