Introduction to machine learning — BA3
Detailed summary

Joachim Favre
Course by Prof. Mathieu Salzmann

Autumn semester 2022

Contents

Version 2025–02–07

1 Prerequisites

Supervised and unsupervised learning

In **supervised learning**, we are given data and its groundtruth labels. The goal is, given new data, we want to predict new labels, by doing regression or classification.

In **unsupervised learning**, we are only given data (without any label), and we want to output some information about it, by doing dimensionality reduction or clustering.

Regression and classification

The goal of **regression** is to predict a continuous value for a given sample. The goal of **classification** is to output a discrete label (typically encoded in one-hot encoding with 0s and 1s or -1s and 1s).

The main difference is that there is the notion of closeness in regression (when predicting a date, outputting 1970 when it should be 1980 is better than outputting 2100), which is not in classification (when predicting what is on a picture, outputting a car when it should be a cat is not better or worse than outputting an elephant).

Dimensionality reduction

Dimensionality reduction has two main advantages.

The first one is that it allows to decrease the dimension of our data, which typically yield a tremendous speed-up while preserving a lot of the precision.

The second one is that, depending on the model, we can also map data back from the lower dimensional space to the higher one. This can be very interesting since it allows to create new samples. For instance, applying dimensionality reduction on a set of human faces, we could create new points $\vec{\boldsymbol{y}}$ randomly thanks to the distribution of our data, and map it back to high dimension. That way, we created a new random face. Another use for this is to denoise the data: mapping a sample to a lower dimensional space and back to high dimensions may result to a lot less noise.

Notations

We consider the following notation throughout this course, with some slight exceptions when specified otherwise. N is the number of samples we have, D is the dimensionality (the number of components) of any input, and C is the dimensionality of any output.

Without specified otherwise, the input $\vec{\boldsymbol{x}}_i \in \mathbb{R}^{D+1}$ begins with a constant 1 to account for a bias, followed by the input data. We let $\vec{\boldsymbol{y}}_i \in \mathbb{R}^C$ to be the i^{th} output. $x_i^{(k)}$ is the k^{th} component of the i^{th} input, and similarly for $y_i^{(k)}$. To sum up, we have:

$$\vec{\boldsymbol{x}}_i = \begin{pmatrix} 1 \\ x_i^{(1)} \\ \vdots \\ x_i^{(D)} \end{pmatrix} \in \mathbb{R}^{D+1}, \quad \vec{\boldsymbol{y}}_i = \begin{pmatrix} y_i^{(1)} \\ \vdots \\ y_i^{(C)} \end{pmatrix} \in \mathbb{R}^C$$

Any value output by our model will be represented by a $\hat{\vec{\boldsymbol{y}}} \in \mathbb{R}^C$, to make a difference with the groundtruth $\vec{\boldsymbol{y}} \in \mathbb{R}^C$.

We will also need weights, which represent the parameters of our model. $\vec{\boldsymbol{w}}_{(i)}$ represents the weight to convert any $\vec{\boldsymbol{x}}$ to the i^{th} component of the output y_i. We need their size to compute dot products with $\vec{\boldsymbol{x}}$, i.e. $\vec{\boldsymbol{x}}_i^T \vec{\boldsymbol{w}}_{(i)}$ must make sense. Thus, without specified otherwise, we use $\vec{\boldsymbol{w}}_{(i)} \in \mathbb{R}^{(D+1)}$.

To simplify the notation and the computations, we will stack our values in matrices. Thus, we let:

$$X = \begin{pmatrix} \vec{\boldsymbol{x}}_1^T \\ \vdots \\ \vec{\boldsymbol{x}}_N^T \end{pmatrix} = \begin{pmatrix} 1 & \cdots & x_1^{(D)} \\ \vdots & \ddots & \vdots \\ 1 & \cdots & x_N^{(D)} \end{pmatrix} \in \mathbb{R}^{N \times (D+1)}$$

$$Y = \begin{pmatrix} \vec{\boldsymbol{y}}_1^T \\ \vdots \\ \vec{\boldsymbol{y}}_N^T \end{pmatrix} = \begin{pmatrix} y_1^{(1)} & \cdots & y_1^{(C)} \\ \vdots & \ddots & \vdots \\ y_N^{(1)} & \cdots & y_n^{(C)} \end{pmatrix} \in \mathbb{R}^{N \times C}$$

$$W = \begin{pmatrix} \vec{\boldsymbol{w}}_{(1)} & \cdots & \vec{\boldsymbol{w}}_{(C)} \end{pmatrix} \in \mathbb{R}^{(D+1) \times C}$$

Feature expansion

Increasing the amount of dimensions from D to F of our input data may help our models (using non-linear functions, since they would be of no help). Thus, we may let the following function:

$$\varphi(\vec{\boldsymbol{x}}) = \begin{pmatrix} 1 & x^{(1)} & \dots & x^{(D)} & \left(x^{(1)}\right)^2 & \dots & \left(x^{(D)}\right)^2 & \dots \end{pmatrix}^T \in \mathbb{R}^F$$

We also put it in a matrix so simplify notation:

$$\Phi = \begin{pmatrix} \varphi(\vec{\boldsymbol{x}}_1)^T \\ \vdots \\ \varphi(\vec{\boldsymbol{x}}_N)^T \end{pmatrix} \in \mathbb{R}^{N \times F}$$

We can replace $\vec{\boldsymbol{x}}$ by $\varphi(\vec{\boldsymbol{x}})$ and X by Φ in mostly every model, especially the ones which will be kernalised. Note that, in this case, we need $\vec{\boldsymbol{w}} \in \mathbb{R}^F$, and thus $W = \mathbb{R}^{F \times C}$.

Remark — The 1 we added to the input data to account for the bias is some kind of feature expansion.

Cover's Theorem — Cover's theorem states (more or less) that doing non-linear feature expansion, then it is more likely for our data to be linearly separable.

Kernel

We can notice that defining our φ functions for feature expansion can be really tedious. However, since most of our methods depend on a dot product of $\varphi(\vec{\boldsymbol{x}}_i)^T \varphi(\vec{\boldsymbol{x}}_j)$, which gives some kind of measure of similarity between $\vec{\boldsymbol{x}}_i$ and $\vec{\boldsymbol{x}}_j$ (since it is proportional to the cosine of their angle), we can define a similarity function named a **kernel**, such that:

$$k(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j) = \varphi(\vec{\boldsymbol{x}}_i)^T \varphi(\vec{\boldsymbol{x}}_j)$$

As usual, we put everything in vectors and matrices to simplify our notation. We first have a way to measure the similarity between a sample $\vec{\boldsymbol{x}}_i$ and all the other samples:

$$k(X, \vec{\boldsymbol{x}}_i) = \begin{pmatrix} k(\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_i) \\ \vdots \\ k(\vec{\boldsymbol{x}}_n, \vec{\boldsymbol{x}}_i) \end{pmatrix} \in \mathbb{R}^N$$

And we can then stack all of them in a matrix to define similarity between all pairs of samples:

$$K = \begin{pmatrix} k(\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_1) & k(\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2) & \cdots & k(\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_N) \\ k(\vec{\boldsymbol{x}}_2, \vec{\boldsymbol{x}}_1) & k(\vec{\boldsymbol{x}}_2, \vec{\boldsymbol{x}}_2) & \cdots & k(\vec{\boldsymbol{x}}_2, \vec{\boldsymbol{x}}_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(\vec{\boldsymbol{x}}_N, \vec{\boldsymbol{x}}_1) & k(\vec{\boldsymbol{x}}_N, \vec{\boldsymbol{x}}_2) & \cdots & k(\vec{\boldsymbol{x}}_N, \vec{\boldsymbol{x}}_N) \end{pmatrix} \in \mathbb{R}^{N \times N}$$

Note that, since $k(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j) = k(\vec{\boldsymbol{x}}_j, \vec{\boldsymbol{x}}_i)$ by the commutativity of the dot product, K is symmetric ($K^T = K$).

Remark — The main advantage of a kernel is that we don't need to know what function φ is linked to it.

Examples — We can for instance use the **polynomial kernel**:

$$k(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j) = \left(\vec{\boldsymbol{x}}_i^T \vec{\boldsymbol{x}}_j + c\right)^d$$

c is often set to 1 and d to 2. For this kernel, the corresponding mapping φ is known. This is, except for multiplicative constants, all the possible monomials of degree less than or equal to d composed of the components of $\vec{\boldsymbol{x}}_i$ and $\vec{\boldsymbol{x}}_j$.

We can also use the **Gaussian kernel** (or radial basis function (RBF)):

$$k(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j) = \exp\left(-\frac{\|\vec{\boldsymbol{x}}_i - \vec{\boldsymbol{x}}_j\|^2}{2\sigma^2}\right)$$

σ is typically chosen relatively to the data.

Representer theorem

The minimizer of a regularized empirical risk function can be represented as a linear combination of expanded features. In other words, we can write:

$$\vec{\boldsymbol{w}}^* = \sum_{i=1}^{N} \alpha_i^* \varphi(\vec{\boldsymbol{x}}_i) = \Phi^T \vec{\boldsymbol{\alpha}}^*$$

where $\vec{\boldsymbol{\alpha}} \in \mathbb{R}^N$.

Remark

This theorem is really important to kernalise algorithms. When using it, the goal is to get rid of the Φ since we do not know the mapping φ. Switching our view onto variables $\vec{\boldsymbol{\alpha}}$ instead of variables $\vec{\boldsymbol{w}}$ is typically a way to do so.

Loss function

The **loss function** $\ell\left(\hat{\vec{\boldsymbol{y}}}_i, \vec{\boldsymbol{y}}_i\right)$ computes an error value between the prediction and the true value.

This is a measure of the error for any given prediction.

Empirical risk

Given N training samples $\left\{\left(\hat{\vec{\boldsymbol{x}}}_i, \hat{\vec{\boldsymbol{y}}}_i\right)\right\}$, the **empirical risk** is defined as:

$$R\left(\left\{\hat{\vec{\boldsymbol{x}}}_i\right\}, \left\{\hat{\vec{\boldsymbol{y}}}_i\right\}, W\right) = \frac{1}{N}\sum_{i=1}^{N} \ell\left(\hat{\vec{\boldsymbol{y}}}_i, \vec{\boldsymbol{y}}_i\right)$$

This represents the global error on the training samples, this is what we try to minimise:

$$W^* = \underset{W}{\operatorname{argmin}}\, R\left(\left\{\hat{\vec{\boldsymbol{x}}}_i\right\}, \left\{\hat{\vec{\boldsymbol{y}}}_i\right\}, W\right)$$

Regularised

Sometimes, we want to regularise our objective function, so that we prevent weights to become to large and make a lot of overfitting. We then instead seek to minimise.

$$E(W) = R(W) + \lambda E_W(W)$$

where λ is an hyperparameter and $E_W(W)$ is the regulariser.

A regulariser that can be used is Tikhonov regularisation, and it will be used in ridge regressionb:

$$E_W(W) = \|W\|_F^2$$

where $\|W\|_F^2$ is the squared Frobenius norm of W, meaning the sum of its values squared.

Gradient descent

The goal of **gradient descent** is to minimise a function (an empirical risk $R(W)$ in this context). The idea is to begin with an estimate W_0 (typically completely random), and then to update it iteratively, by following the direction of greatest decrease (the opposite of the gradient):

$$W_k = W_{k-1} - \eta \nabla_W R(W_{k-1})$$

where $\eta > 0$ is the learning rate.

This algorithm can then be stopped after the change in the function reaches a threshold $|R(W_k) - R(W_{k-1})| < \delta_R$, the change in parameter value is less than a threshold $|W_k - W_{k-1}| < \delta_w$, or if a maximum number of iterations (also known as epochs) is reached (even though this gives no guarantee on a potential convergence).

Remark

This algorithm does not always converge and, when it does, not necessarily to a minimum nor to the global minimum.

Stochastic gradient descent

When we have a lot of input and a non-convex function (meaning that it probably does not have a single minimum, and thus that gradient descent can loose itself in some of them), we can use **stochastic gradient descent**. The idea is, instead of using all our samples, we only use a random subset (named **mini-batch**) of B samples. The update rule becomes:

$$W_k = W_{k-1} - \eta \sum_{b=1}^{B} \nabla \ell_{i(b,k)}(W_{k-1})$$

This converges faster for the same number of computations, and it may get out of bad local minima.

Evaluation metrics

Once a supervised machine learning model is trained, we want to be able to understand how well it performs on unseen test data (which must absolutely be separated from the train data).

We could use the loss function, but we may also use a different one.

For regression, we typically use **Mean Squared Error** (MSE):

$$\text{MSE} = \frac{1}{N_t} \sum_{i=1}^{N_t} \left\| \hat{\vec{\boldsymbol{y}}}_i - \vec{\boldsymbol{y}}_i \right\|^2$$

where N_t is the number of test sample.

For classification, this is typically more complicated. Defining TP to be the number true positive predictions (samples correctly predicted positive), FN to be the number of false negative predictions (samples incorrectly predicted negative), and similarly for TN and FP, we can define the **accuracy** (the percentage of correctly classified samples), the **precision** (the percentage of samples classified positives, which are truly positives) and the **recall** (the percentage of positive samples that are correctly classified as positives):

$$\text{accuracy} = \frac{TP + TN}{N_t}, \quad \text{precision} = \frac{TP}{TP + FP}, \quad \text{recall} = \frac{TP}{P}$$

We can then combine the two last to make a **F1 score**:

$$\text{F1} = 2 \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

We can typically then use accuracy and the F1 score to see how well our classification model did.

Remark

There are many more metrics for regression and classification. For the former, we could quote RMSE (root mean squared error), MAE (mean absolute error) or the MAPE (mean absolute percentage error). For the latter, making a confusion matrix or computing the AUC (area under the ROC curve) can give good insights too.

Decision boundary

A classifier leads to a decision boundary. This is an object of dimension $D-1$ (a line if our data lies on a plane for instance), which splits the space into two regions: one where samples are considered positive (the predicted value is closer to the value of positive samples), and one where they are considered to be negative (the predicted value is closer to the value of negative samples).

Since it is the set of points which are equivalently distant to both labels, the boundary is parametrised by the following equation:

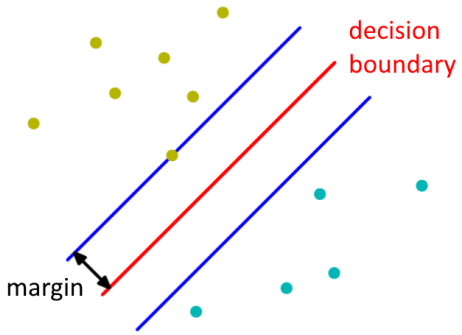
$$\hat{\vec{\boldsymbol{y}}}(\vec{\boldsymbol{x}}) = \frac{y_{pos} + y_{neg}}{2}$$

where y_{pos} is the value for positive samples, and y_{neg} is the value for negative samples.

Remark A classifier is said to be linear if its decision boundary is an hyperplane (a straight line if the data lies on a plane for instance).

Margin

If $C = 1$, the orthogonal distance between the decision boundary and the nearest sample is called the margin.



Overfitting

When we increase the complexity of the model (by changing the hyperparameters) we get better and better result for both training and test data. However, there is a point at which increasing the complexity keeps decreasing error on training data but increases the error on test data. This is a very general phenomenon, named **overfitting**.

Cross-validation

Cross-validation is a way to find good hyperparameters that prevent overfitting. We test different models (in a predefined set), assign to each of them an error value, and pick the one yielding the smallest error.

The idea of ***k*-fold cross-validation** is, to evaluate the error of a model, to first randomly split the dataset into k partitions (where k is predefined). Then, we do k iterations: at iteration i we drop the i^{th} partition, train the model on the $(k-1)$ other folds merged, and use the i^{th} partition to compute the error. At the end of all the iterations, we average all the errors.

Note that we never test the model on data we used to train it, allowing to avoid overfitting. Also, we use all of our training data to get a complete insight over it (doing only one iteration, would give less information about the model). It is important to notice that the larger the k, the less data we waste but the more expensive this method becomes.

Remark Note that leaving $k = N$ (where N is the number of training samples) is also sometimes named **leave-one-out cross-validation**. This is really expensive but wastes (almost) no data.

Another way to do cross-validation, which is much cheaper, is to split our training data into training and validation data, using **validation-set cross-validation**. This is like like doing only one iteration of k-fold cross-validation, meaning that it is very cheap but wastes a lot of data.

Data representation

All the models we will see only work for fixed size data. If we want to handle data of varying size (such as text or pictures), a good way is to consider **bag of words**: consider the number of times each word from a dictionary appears in the text and put this as a big vector.

Note that we don't need to consider the whole English dictionary, only picking the set of words appearing in the training data is enough. Also, it is often interesting to make a histogram out of our vector: divide it by the number of words in the sample, so that we instead have a repartition and give less importance to the length of the text.

Also, we can apply this to images. To do so, we need to extract "words", meaning fixed-size picture elements.

Pre-processing

The training data might have problems: it might have noise (because of measurement errors), incorrect values, and so on. To fix those, a good idea is to do pre-processing.

Normalisation

To begin with, a good idea is to scale each individual feature dimension to fall within a specified range (so that we don't give more impact to a dimension ranging from 1000 to 10000 than to another dimension ranging from 0 to 1). This can typically be done by using **normalisation**, such as z-score normalisation:

$$\widetilde{x}_i^{(d)} = \frac{x_i^{(d)} - \mu^{(d)}}{\sigma^{(d)}}$$

where $\mu^{(d)}$ is the mean of the d^{th} dimension, and $\sigma^{(d)}$ is its standard deviation.

Note that there are many other ways to do normalisation, such as min-max normalisation (computing $\widetilde{x}_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$), max-normalisation (computing $\widetilde{x}_i = \frac{x_i}{x_{max}}$ where x_{max} is the maximum in absolute value) or decimal-scaling (compute $\widetilde{x}_i = \frac{x_i}{10^k}$ where k is the smallest integer k such that $|\widetilde{x}_i| \leq 1$ for the largest $\widetilde{x}_i$).

Imbalanced data

Another important thing to consider is **imbalanced data**. There might be 10 times as much data in one class as in another (between-class imbalance), or data inside a class might have a lot of representative at some point in space and much less at other points (within-class imbalance).

To fix those problems, we can either work on the data, or on the cost function.

To work on the data, the first method is to decrease the data set by **undersampling**: we remove samples randomly, or more intelligently by removing samples considered redondant. The second method is to increase the data set by **oversampling**: we replicate exactly some of the samples (which might lead to overfitting), or do interpolation (which might create a lot of noise).

To work on the cost function, we can give more impact to some samples:

$$R(\{\vec{\boldsymbol{x}}_i\}, \{\vec{\boldsymbol{y}}_i\}, W) = \frac{1}{N} \sum_{i=1}^{N} \beta_i \ell\left(\hat{\vec{\boldsymbol{y}}}_i, \vec{\boldsymbol{y}}_i\right)$$

We can for instance use weights inversely proportional to the class frequency, such as $\beta_i = 1 - \frac{N_k}{N}$.

2 Regression and classification

Ridge regression

The output of **ridge regression** is computed by a simple dot product:

$$\hat{\vec{\boldsymbol{y}}}_i = W^T \varphi(\vec{\boldsymbol{x}}_i)$$

The training objective function we want to minimise is a squared Euclidean distance regularised by the sum of squares of the weights:

$$E(W) = R(W) + \lambda E_W(W) = \sum_{i=1}^{N} \left\| \hat{\vec{\boldsymbol{y}}}_i - \vec{\boldsymbol{y}}_i \right\|^2 + \lambda \|W\|_F^2$$

where $\|W\|_F^2$ is the Frobenius norm of W, meaning the sum of the square of all its values, and $\lambda \geq 0$ is a hyperparameter.

This can be solved explicitly:

$$W^* = \left(\Phi^T \Phi + \lambda I_F\right)^{-1} \Phi^T Y$$

where I_F is the $F \times F$ identity matrix.

Linear regression

Leaving $\lambda = 0$, we get the special case of **linear regression**. Then, the closed-form formula can be rephrased as:

$$W^* = \left(\Phi^T \Phi\right)^{-1} \Phi^T \vec{\boldsymbol{y}} = \Phi^\dagger Y$$

where $\Phi^\dagger$ is the Moore-Penrose pseudo-inverse.

Kernelisation

Using the representer theorem, we can find that:

$$A^* = (K + \lambda I_N)^{-1} Y$$

This is not of much use on its own, but we can use this result to find how we predict a value $\hat{\vec{\boldsymbol{y}}}$ for a new $\vec{\boldsymbol{x}}$:

$$\hat{\vec{\boldsymbol{y}}} = Y^T (K + \lambda I_N)^{-1} k(X, \vec{\boldsymbol{x}})$$

Note that the value $Y^T (K + \lambda I_N)^{-1}$ can be computed once during training, and then be reused at every evaluation of the model.

Classification

Ridge regression can be used for classification tasks, by inputting the result into a softmax function (defined right after), but this does not work very well because we are not encoding this in the objective function.

This is named a **least-square classifier**.

Logistic regression

In **logistic regression** (which is a linear classification algorithm), we consider negative samples to be $y_i = 0$.

The output of logistic regression is computed by using the **softmax function**:

$$\hat{y}^{(k)} = \frac{\exp\left(\vec{\boldsymbol{w}}_{(k)}^T \vec{\boldsymbol{x}}\right)}{\sum_{j=1}^{C} \exp\left(\vec{\boldsymbol{w}}_{(j)}^T \vec{\boldsymbol{x}}\right)} \in [0, 1]$$

where $\hat{y}^{(k)}$ represents the probability that the sample $\vec{\boldsymbol{x}}$ is in class k.

The empirical risk we try to minimise is the **cross entropy**:

$$R(W) = -\sum_{i=1}^{N} \sum_{k=1}^{C} y_i^{(k)} \ln\left(\hat{y}_i^{(k)}\right)$$

Note that $y_i^{(k)}$ is non-zero only for a single sample i.

There is no closed-form formula, so we need the gradient of the empirical risk in order to do gradient descent (the solution is unique since the function is convex):

$$\nabla_W R(W) = \sum_{i=1}^{N} \vec{\boldsymbol{x}}_i \left(\hat{\vec{\boldsymbol{y}}}_i - \vec{\boldsymbol{y}}_i\right)^T$$

One dimension

Let's consider $C = 1$. This special case of the softmax function is named the **logistic function**:

$$\hat{y}(\vec{\boldsymbol{x}}) = \sigma\left(\vec{\boldsymbol{w}}^T \vec{\boldsymbol{x}}\right) = \frac{1}{1 + \exp(-\vec{\boldsymbol{w}}^T \vec{\boldsymbol{x}})}$$

One-dimensional cross-entropy can be rewritten as:

$$R(\vec{\boldsymbol{w}}) = -\sum_{i \in P} \ln(\hat{y}_i(\vec{\boldsymbol{w}})) - \sum_{i \in N} \ln(1 - \hat{y}_i(\vec{\boldsymbol{w}}))$$

where P is the set of positive samples and N the set of negative samples.

The gradient of one-dimensional cross-entropy is:

$$\nabla_{\vec{\boldsymbol{w}}} R(\vec{\boldsymbol{w}}) = \sum_{i=1}^{N} (\hat{y}_i - y_i) \vec{\boldsymbol{x}}_i$$

Kernelisation

This algorithm can be kernalised even though this is not very common.

Support vector machine

In **support vector machine** (SVM) classification (which is also a linear classifier), we consider negative samples to be $y_i = -1$. Also, we leave $\widetilde{\vec{\boldsymbol{w}}}$ to be the vector of parameters without $w^{(0)}$, and $\vec{\boldsymbol{x}} \in \mathbb{R}^D$ to not have an added 1. Note that we only consider $C = 1$ for now.

The idea of SVM is to maximise the size of the margin. A prediction is given by whether $w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_i$ is closest to -1 or 1 (as always for linear classifiers), and the decision boundary is given by $w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}} = \frac{-1+1}{2} = 0$.

We show in one of the following paragraphs that the problem can be formulated as:

$$\underset{\vec{\boldsymbol{w}}, \{\xi_i\}}{\operatorname{argmin}} \frac{1}{2} \left\| \widetilde{\vec{\boldsymbol{w}}} \right\|^2 + C \sum_{i=1}^{N} \xi_i,$$

subject to $y_i \left(w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_i \right) \geq 1 - \xi_i$ and $\xi_i \geq 0$, for all i. Note that C is an hyperparameter, and the ξ_i are **slack variables** we added that we also minimise. Those slack variables allow for data which is not linearly separable data to also be used.

When $\xi_i = 0$, the point is on the correct side of the margin, this is how everything should work. If $0 < \xi_i < 1$, then the point i are on the wrong side of the margin, but correctly classified. If $\xi_i = 1$, the point is on the decision boundary (and thus misclassified). If $\xi_i > 1$, then the point is on the wrong side of the decision boundary, and thus misclassified.

This representation of the problem is known as the **primal problem**.

Support vectors

We notice that, for the margin to be maximised, there must be at least a point from each class lying on it. Such points are named **support vectors**.

Hinge loss

By rewriting the constraints, we get:

$$\xi_i \geq 1 - y_i \left(w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_i \right)$$

For samples i that satisfy the support vector machine problem (they are on not in the margin nor misclassified), we have $\xi_i = 0$ (since they are forced to be non-negative). For samples i which don't, this inequality is held. This allows us to rewrite our SVM primal

problem as:

$$\underset{\vec{\boldsymbol{w}},\{\xi_i\}}{\operatorname{argmin}} \frac{1}{2C}\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|^2 + \sum_{i=1}^{N} \max\left(0, 1 - y_i\left(w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_i\right)\right),$$

subject to the same conditions
This new term is called the **hinge loss**:

$$\max\left(0, 1 - y_i\left(w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_i\right)\right)$$

Dual problem

We can reformulate our problem by letting one variable per training sample (meaning that we have N variables instead of $(D+1)$):

$$\underset{\{\alpha_i\}}{\operatorname{argmax}} \left(\sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \vec{\boldsymbol{x}}_i^T \vec{\boldsymbol{x}}_j\right),$$

subject to $\sum_{i=1}^{N} \alpha_i y_i = 0$ and $0 \leq \alpha_i \leq C$ for all i.
The solution is equivalent to the primal problem:

$$\widetilde{\vec{\boldsymbol{w}}}^* = \sum_{i=1}^{N} \alpha_i^* y_i \vec{\boldsymbol{x}}_i \implies \hat{y}(\vec{\boldsymbol{x}}) = w^{(0)*} + \sum_{i=1}^{N} \alpha_i y_i \vec{\boldsymbol{x}}_i^T \vec{\boldsymbol{x}}$$

Note that $w^{(0)}$ can also be found thanks to those α_i^*.
We can show that, at the solution, we have:

$$\alpha_i^*\left(y_i\left(w^{(0)*} + \widetilde{\vec{\boldsymbol{w}}}^{*T} \vec{\boldsymbol{x}}_i\right) - 1 + \xi_i^*\right) = 0$$

In other words, for every sample, either of those terms is equal to 0. The samples for which $\alpha_i^* \neq 0$ are the support vectors. However, most samples are not support vectors, so we can decrease the computations by leaving $\mathcal{S}$ to be the set of support vectors:

$$\hat{y}(\vec{\boldsymbol{x}}) = w^{(0)*} + \sum_{i \in \mathcal{S}} \alpha_i^* y_i \vec{\boldsymbol{x}}_i^T \vec{\boldsymbol{x}}_i$$

Kernelisation

We need the dual problem to kernelise the SVM:

$$\underset{\{\alpha_i\}}{\operatorname{argmax}} \left(\sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j k(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j)\right),$$

subject to $\sum_{i=1}^{N} \alpha_i y_i = 0$ and $0 \leq \alpha_i \leq C$ for all i.
The prediction is also similar:

$$\hat{y}(\vec{\boldsymbol{x}}) = w^{(0)*} + \sum_{i \in \mathcal{S}} \alpha_i^* y_i k(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}})$$

Note that we still have $\alpha_i^* = 0$ for all samples that are not support vectors.

Multi-class SVM

To generalise our algorithm to multiple class, we can use multiple ways. The idea is always to use several binary classifiers.

One way is to use **one-vs-rest**: we train classifiers stating if the component is in class i or not. Another way is to use **one-vs-one**: we train classifiers to know if the component is closer to class i or j, and then pick the best one. However, in both cases, there are some

samples which will give ambiguous answers (belonging to multiple classes or to none).

Primal derivation

Let's consider how we got the formula for the primal model, since it may typically help to remember and understand it.

First, we know that any two points on the decision boundary have the same prediction (which is 0), which yields that:

$$0 = \hat{y}_1 - \hat{y}_2 = \left(w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_1\right) - \left(w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_2\right) = \widetilde{\vec{\boldsymbol{w}}}^T (\vec{\boldsymbol{x}}_1 - \vec{\boldsymbol{x}}_2)$$

This is a dot product, and it thus means that $\widetilde{\vec{\boldsymbol{w}}}$ is orthogonal to the decision boundary.

Second, we use this information to split any point into a component colinear to $\widetilde{\vec{\boldsymbol{w}}}$, and one orthogonal to it (meaning colinear to the decision boundary):

$$\vec{\boldsymbol{x}} = \vec{\boldsymbol{x}}_\perp + r\frac{\widetilde{\vec{\boldsymbol{w}}}}{\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|}$$

where r is the signed orthogonal distance of any point to the decision boundary.

Now, looking at the prediction yielded by this point, we get:

$$\hat{y}(\vec{\boldsymbol{x}}) = w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}} = w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_\perp + r\frac{\widetilde{\vec{\boldsymbol{w}}}^T \widetilde{\vec{\boldsymbol{w}}}}{\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|} = y(\vec{\boldsymbol{x}}_\perp) + r\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|$$

However, we know that $y(\vec{\boldsymbol{x}}_\perp) = 0$ since it is on the decision boundary, meaning that:

$$\hat{y}(\vec{\boldsymbol{x}}) = r\left\|\widetilde{\vec{\boldsymbol{w}}}\right\| \iff r = \frac{\hat{y}(\vec{\boldsymbol{x}})}{\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|}$$

which is, to recall, the signed orthogonal distance of $\vec{\boldsymbol{x}}$ to the decision boundary.

Note that we can then make use of the ground truth label being -1 or 1 to make an unsigned distance:

$$\widetilde{r}_i = y_i r_i = \frac{y_i\left(w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_i\right)}{\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|}$$

This is what we would like to maximise, but it is hard. We thus need to turn it to an equivalent problem. To do so, we notice that there is an infinite number of solutions that matter, since we only want the direction of $\vec{\boldsymbol{w}}$ and that its magnitude does not matters. This can be proven mathematically by multiplying our weights by a λ, and seeing that we get the same $\widetilde{r}_i$ above (the λ cancel out in the fraction).

So, we may as well require that the margin has size $\frac{1}{\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|}$, meaning that:

$$r_i \geq \frac{1}{\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|} \iff r_i\left\|\widetilde{\vec{\boldsymbol{w}}}\right\| \geq 1 \iff y_i\left(w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_i\right) \geq 1$$

Now, maximising the margin means maximising $\frac{1}{\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|}$ which can be shown to be equivalent to minimising $\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|^2$. Our problem has thus

become:

$$\underset{\vec{\boldsymbol{w}}}{\operatorname{argmin}} \frac{1}{2}\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|^2, \quad \text{subject to } y_i\left(w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_i\right) \geq 1, \ \forall i$$

where the factor $\frac{1}{2}$ was added for convenience.

However, if the data is not linearly separable, we need a way to let some samples violate this rule. This is done by adding **slack variables** $\xi_i \geq 0$ for each sample:

$$y_i\left(w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}\vec{\boldsymbol{x}}_i\right) \geq 1 - \xi_i$$

In other words, if $0 < \xi_i \leq 1$, the sample i lies inside in the margin but is still correctly classified. If $\xi_i \geq 1$, then the sample i is misclassified.

We minimise those variables jointly with our original problem, giving us our final formulation:

$$\underset{\vec{\boldsymbol{w}},\{\xi_i\}}{\operatorname{argmin}} \frac{1}{2}\left\|\widetilde{\vec{\boldsymbol{w}}}\right\|^2 + C\sum_{i=1}^{N}\xi_i,$$

subject to $y_i\left(w^{(0)} + \widetilde{\vec{\boldsymbol{w}}}^T \vec{\boldsymbol{x}}_i\right) \geq 1 - \xi_i$ and $\xi_i \geq 0$, for all i.

K-nearest neighbours

The idea of ***k*-nearest neighbours** (kNN) is to compute the distance between the test sample $\vec{\boldsymbol{x}}$ and all training samples $\{\vec{\boldsymbol{x}}_i\}$ and find the k samples with minimum distances. Then, we can do classification by finding the most common label amongst these k nearest neighbours, or do regression by computing $\hat{\boldsymbol{y}}$ as a function of those neighbours and their distance to $\vec{\boldsymbol{x}}$.

To compute close points efficiently we can use datastructures such as k-d trees.

Remark

The result of this model depends on the choice of the distance function. One can take the **Euclidean distance**:

$$d(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}) = \sqrt{\sum_{d=1}^{D}\left(x_i^{(d)} - x^{(d)}\right)^2}$$

However, for some other structures such as histograms (each data point only has component between 0 and 1, and the sum of all of the components of a data point is equal to 1; like for a probability distribution), then we instead tend to use a Chi-square distance:

$$d(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}) = \sqrt{\chi^2(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}})} = \sqrt{\sum_{d=1}^{D}\frac{\left(x_i^{(d)} - x^{(d)}\right)^2}{x_i^{(d)} + x^{(d)}}}$$

The important thing to remember from this paragraph is that the choice of the distance function is important.

Curse of dimensionality

Because of a principle named the **curse of dimensionality**, we need exponentially more points to cover a space as the number of dimensions increases. Using dimensionality reduction is a good idea with this algorithm.

Complexity

Unlike most of the other models, increasing the hyperparameter of this model (the k) leads to decreased complexity: the higher the k, the less complex the decision boundary is and thus the less overfit we have.

Neural networks

Neural networks can do both classification and regression (depending on the output representation and the empirical risk used, typically square loss for regression and cross-entropy for classification), and their main advantage is that they learn a good model during the training.

This method is named **neural network**, **multi-layer perceptron** (MLP), or **deep learning** (as long as there are at least two hidden layers).

The idea is to have layers composed of neurons. Every neuron of a layer is connected to every neurons of the previous layer, meaning that its value is computed by a weighted sum over them, plus a bias. More than that, the important thing for our model not to be just a big linear regression, is that each neuron is passed through a non-linear activation function. Mathematically speaking, this is given by:

$$\vec{z}_{(l)} = f_{(l)}\left(W_{(l)}^T \vec{z}_{(l-1)}\right)$$

where $\vec{z}_{(0)} = \vec{x}$ is the input layer, $\vec{z}_{(L+1)} = \hat{\vec{y}}$ is the output layer, L is the number of hidden layers (layers which are neither input nor output), and $f_{(l)}(x)$ is applied to every component of the vector. Note that each $\vec{z}_\ell$ has a "bias term" just like the input data (meaning a 1 appended at the beginning).

This is trained to optimality using stochastic gradient descent, by focusing on a single loss term $\ell\left(\hat{\vec{y}}_i, \vec{y}_i\right)$ at a time. To do so, we need to compute the gradients $\frac{\partial \ell_i}{\partial W_l^{(k,j)}}$ (where we are considering the loss of the i^{th} sample, and the weight at position (k, j) of the weight matrix from layer l). This is done by an algorithm named **backpropagation**.

We can notice that, by the chain rule (and abusing slightly of the notation of the derivative):

$$\frac{\partial \ell_i}{\partial W_l} = \frac{\partial z_{(l)}}{\partial W_l} \frac{\partial z_{(l+1)}}{\partial z_l} \cdots \frac{\partial z_{(L)}}{\partial z_{(L-1)}} \frac{\partial \ell_i}{\partial z_{(L)}}$$

We can compute $\frac{\partial z_{(l)}}{\partial z_{(l-1)}}$ and $\frac{\partial z_{(l)}}{\partial W_l}$ rather easily if we store the values of each layer during the forward pass. We can then propagate backwards those value, by computing the gradients from the end and updating the weights.

Activation functions

There are multiple choice for activation functions. The important thing is that they are non-linear.

We can for instance take the ReLU (Rectified Linear Unit) activation function:

$$f(a) = \begin{cases} a, & \text{if } a > 0 \\ 0, & \text{otherwise} \end{cases}$$

Another choice could be the sigmoid:

$$f(a) = \frac{1}{1 + \exp(-a)}$$

Convolutional network

When working with pictures, just vectorising them may give a lot of elements while removing the fact that the picture is inherently two-dimensional. A way to circumvent this problem is using convolutions.

To make a convolution, we need a small matrix of elements (plus a bias). We center this matrix at an element, compute the weighted sum resulting from it, add the bias, and use this as our result. We can then shift it to center it on each element, yielding a new picture. This allows to extract some features of our original picture, such as the edges.

We can also use multiple filters to create multiple channels, increasing the amount of data. If at some point we have 3 channels (for instance) and want to do a convolution with a 5×5 filter, then we

will use matrices of size $5 \times 5 \times 3$ (going three-dimensional over our channels). Note that we can also use some other parameters, such as strides (skip one pixels over two for instance) or padding (add some pixels on the edges).

We can also use pooling layers, splitting the pixels into $k \times k$ squares and extracting only one value per partition (by taking the maximum value or the average for instance). This allows to decrease the size of our pictures.

The main interest then comes from stacking those operations. For instance, beginning with a 28×28 input, we may apply convolutions to get a $3 \times 24 \times 24$ layer (we loose some pixels because we may require the filter to ignore pixels where it has to be partly outside of the picture) and then a pooling layer to get a $3 \times 12 \times 12$ layer.

The idea of a **convolutional neural network** (CNN) is to make some convolutions (typically before the input layers, but also in-between some layers). The main idea is that we also optimise the values inside the filters of our convolutions. To compute the gradient for them, we also use back-propagation.

3 Dimensionality reduction and clustering

Principle component analysis

Sometimes, we realise that data is given in many dimensions, but actually lies in many less dimensions. The idea of **principle component analysis** (PCA) is to project the data on some orthogonal axis (of lower dimension), in a way to maximise the kept variance.

Leaving $\overline{\vec{\boldsymbol{x}}} = \frac{1}{N}\sum_{i=1}^{N} \vec{\boldsymbol{x}}_i$ to be the mean, we have:

$$\vec{\boldsymbol{y}}_i = W^T\left(\vec{\boldsymbol{x}}_i - \overline{\vec{\boldsymbol{x}}}\right)$$

To find this matrix $W \in \mathbb{R}^{D\times d}$ (where d is the number of dimensions after the projection), we first need to consider the data covariance matrix:

$$C = \frac{1}{N}\sum_{i=1}^{N}\left(\vec{\boldsymbol{x}}_i - \overline{\vec{\boldsymbol{x}}}\right)\left(\vec{\boldsymbol{x}}_i - \overline{\vec{\boldsymbol{x}}}\right)^T$$

Then, picking the d eigenvectors which highest eigenvalues of C, we get our matrix:

$$W = \begin{pmatrix} \vec{\boldsymbol{w}}_{(1)} & \cdots & \vec{\boldsymbol{w}}_{(d)} \end{pmatrix} \in \mathbb{R}^{D\times d}$$

The explained variance yielded by our projection can be found by computing:

$$\text{exval} = \frac{\sum_{j=1}^{d}\lambda_j}{\sum_{j=1}^{D}\lambda_j}$$

Using PCA can make the computations of the models much faster without losing much precision, or getting some insight of the data.

Remark

Since the axis on which we project the data are orthogonal, we have:

$$W^TW = I_d$$

To make sure of this, we need to take the eigenvector so that they are orthogonal. This can always be done because C is symmetric, thanks to the spectral theorem (this theorem also allows to know that we can compare eigenvalues, since they are real).

Mapping

From our computation, we can notice that, for any point $\vec{\boldsymbol{y}} \in \mathbb{R}^d$, we can move it to the high-dimensional space:

$$\hat{\vec{\boldsymbol{x}}} = \overline{\vec{\boldsymbol{x}}} + W\vec{\boldsymbol{y}}$$

This yields all the advantages presented in the first section of this document.

Kerenelisation

PCA can be kernalised in a non-trivial fashion.

First, we need to account for the fact that our data may not be centered in input-space (this was done above by considering the mean of our input values), letting:

$$\widetilde{K} = K - 1_N K - K 1_N + 1_N K 1_N$$

where 1_N is an $N \times N$ matrix, with every element equal to $\frac{1}{N}$.

Leaving $\vec{\boldsymbol{a}}$ to be the vector of unknowns given by the representer theorem, we can find that it follows the following equation:

$$\widetilde{K}\vec{\boldsymbol{a}} = \lambda N \vec{\boldsymbol{a}}$$

This is an eigenvalue problem, which could be solved to find a solution $\vec{\boldsymbol{a}}$. From there, we can project our data:

$$y_i = \sum_{j=1}^{N} a_j k(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j)$$

supposing that we want $d = 1$.

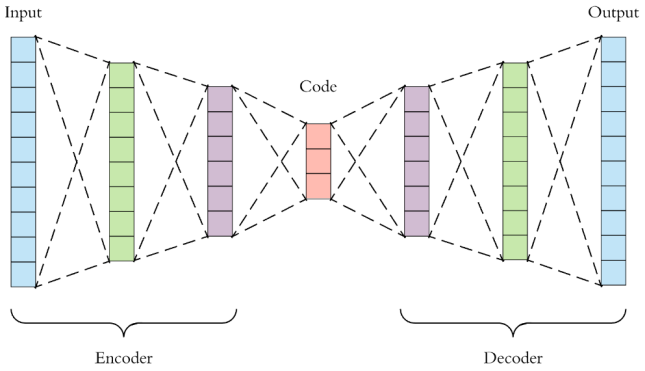
If we want $d > 1$, we can again take the d eigenvectors with greatest eigenvalues, and compute each component of $\vec{\boldsymbol{y}}_i$ thanks to a different eigenvector.

Note that we can no longer map data from d dimensions back to D dimensions with the kernalised method (since it would require us to know the feature expansion mapping $\varphi(\vec{\boldsymbol{x}})$).

Autoencoder

Another way to do dimensionality reduction is through a neural network.

The idea is to have a double funnel shaped neural network: an encoder decreasing the dimension, a layer with d neurons, and a decoder increasing back the number of dimensions.



We can train it to output the same data as what we input it, using a least square empirical risk. That way, it must learn an intelligent code.

Remark This can also be used to both do dimensionality reduction and mapping data back from the low-dimensional space.

Convolutional autoencoder We can use convolutional neural networks for autoencoders. To do so, we use the inverse functions of those convolutions.

Fisher linear discriminant analysis

Even though **Fisher linear discriminant analysis** (LDA) is a dimensionality reduction algorithm, it is a supervised learning one. Its goal is to project data on lower space, while keeping classes (hence the supervision) clustered. It considers that clustering should be done relatively to compactness, meaning distance between points within a cluster should be small, whereas distances across clusters should be large.

The goal is to minimise the distances within a class, meaning the distance of the elements of a class to their class center $E_W(\vec{\boldsymbol{w}})$, while maximising the distance of cluster centers (weighted by the number of elements in the class) $E_B(\vec{\boldsymbol{w}})$. This is better expressed thanks to the within-class scatter matrix S_W and the between-class scatter matrix S_B:

$$S_W = \sum_{c=1}^{C} \sum_{i \in c} (\vec{\boldsymbol{x}}_i - \vec{\boldsymbol{\mu}}_c)(\vec{\boldsymbol{x}}_i - \vec{\boldsymbol{\mu}}_c)^T, \quad S_B = \sum_{c=1}^{C} N_c \left(\vec{\boldsymbol{\mu}}_c - \overline{\vec{\boldsymbol{x}}}\right)\left(\vec{\boldsymbol{\mu}}_c - \overline{\vec{\boldsymbol{x}}}\right)^T$$

where $\overline{\vec{\boldsymbol{x}}}$ si the mean of all the samples, and $\vec{\boldsymbol{\mu}}_c$ the mean of the values in class c.

Our problem can be specified as the following generalised eigenvector problem:

$$S_B \vec{\boldsymbol{w}}_{(1)} = \lambda_1 S_w \vec{\boldsymbol{w}}_{(1)}$$

As for PCA we are in fact looking for the d eigenvectors with largest eigenvalues.

K-means clustering

The idea of ***k*-means clustering** is to consider that clustering should also be done relatively to compactness.

To do so, we consider K (an hyperparameter) cluster centers $\{\vec{\boldsymbol{\mu}}_1, \ldots, \vec{\boldsymbol{\mu}}_K\}$. While we are not converged, we assign each data point $\vec{\boldsymbol{x}}_i$ to the nearest center $\vec{\boldsymbol{\mu}}_k$, and then move $\vec{\boldsymbol{\mu}}_k$ to the mean of the points that were assigned to it.

This algorithm is guaranteed to converge, even though it may not be to the expected solution: some clusters may end up completely empty. A good way to fix this problem is to run it multiple times with different initialisation, and pick the solution minimising the sum of the distance between each point and their assigned cluster center.

Hyperparameter To choose the K, a good way is to draw a graph of the average within-cluster sum of distances with respect to the number of cluster, and pick a point at its "elbow" (where the drop in the y-axis becomes less significant).

Spectral clustering

The idea of **spectral clustering** is to consider that clustering should be done relatively to connectivity instead: we group the points based on edges in a graph, and remove some of the edges with longest length. Let us first consider the case where we only want to make 2 clusters, meaning that we only want one cut.

To create the graph, we need a way to give weights to edges in order to represent their affinity, a way to do so is:

$$w(i,j) = \exp\left(\frac{-\|\vec{\boldsymbol{x}}_i - \vec{\boldsymbol{x}}_j\|^2}{\sigma^2}\right)$$

where σ is an hyper-parameter. Note that this weight decreases as the distance between $\vec{\boldsymbol{x}}_i$ and $\vec{\boldsymbol{x}}_j$ increases. Also, considering this full graph may be expensive, so we can also restrict to the k nearest neighbours of each points.

The goal is now to find a partition $\{A, B\}$, minimising the sum of weights of the edges we have to remove. We thus let this value to be named the cut:

$$\text{cut}(A, B) = \sum_{i \in A} \sum_{j \in B} w(i, j)$$

Just minimising the cut might favour imbalanced partitions, so we also define the degree of a node in the graph d_i and the volume of a partition to be given by:

$$d_i = \sum_j w(i, j), \quad \text{vol}(A) = \sum_{i \in A} d_i$$

Our goal is now to minimise a normalised cut:

$$\text{ncut}(A, B) = \frac{\text{cut}(A, B)}{\text{vol}(A)} + \frac{\text{cut}(A, B)}{\text{vol}(B)}$$

This problem is NP-hard, but it can be relaxed as the following eigenvector problem:

$$(D - W)\vec{\boldsymbol{y}} = \lambda D\vec{\boldsymbol{y}}$$

where D is the diagonal degree matrix ($D_{i,i} = d_i$) and W is the affinity matrix of the graph ($W_{i,j} = w(i, j)$).
The eigenvector with smallest eigenvalue can be shown to be a vector of all ones with eigenvalue 0. We are thus looking for the eigenvector with second smallest eigenvalue. A positive value in this vector indicates that the corresponding point belongs to one partition, and a negative value to the other.

Remark — Since we had to relax the problem, the solution is not always optimal.

K-way partitions — To obtain more than two clusters, we have two choices.
The first one is to recursively apply the two-way partitioning algorithm. This is inefficient and unstable.
The second one is to find K eigenvectors. This leads to each point being represented by a K-dimensional vector. We can then apply K-means clustering to those resulting vectors.