Summary - Machine Learning

David Hägele, August 5, 2018

Contents

1 Machine Learning Briefly

The goal of machine learning is to estimate or "learn" a mapping f from one domain $\mathbb{X}$ to another $\mathbb{Y}$ by using a set of data points.

$f : \mathbb{X} \to \mathbb{Y}$

In the field of supervised learning a data set D usually contains observed data points of domain $\mathbb{X}$ and corresponding points of domain $\mathbb{Y}$. The objective is to find a mapping f_{optimal} that minimizes an error err between observed and estimated y.

$$D = \{x_i, y_i\} \quad | \quad x_i \in \mathbb{X},\ y_i \in \mathbb{Y},\ i \in \mathbb{N}$$
$$err : \mathbb{Y} \times \mathbb{Y} \to \mathbb{R}$$
$$f_{\text{optimal}} = \underset{f}{\operatorname{argmin}}\ err(y, f(y))$$

In the field of unsupervised learning there are no corresponding observed points of domain $\mathbb{Y}$. Optimality is not measured using an error to the observation, instead other objectives are to be fulfilled.

2 Linear Regression

One of the most basic algorithms for estimating a real valued function is linear regression. Given a set of multivariate inputs $x_i \in \mathbb{R}^m$ and corresponding real valued outputs $y_i \in \mathbb{R}$ the following "loss" L is to be minimized by a linear function $f_\beta : \mathbb{R}^m \to \mathbb{R}$.

$$L(\beta) = \sum_{i=1}^{n} (\ y_i - f_\beta(x_i)\ )^2 \tag{1}$$
$$f_\beta(x) = x^\top \beta \quad \text{with} \quad \beta \in \mathbb{R}^m \tag{2}$$

This is the mean squared error and we want to minimize it using a linear function. Lets call the matrix of transposed x_i's X and the vector of y_i's y. This allows for rewriting the loss as follows.

$$L(\beta) = (y - X\beta)^\top (y - X\beta)$$
$$X = \begin{pmatrix} x_1^\top \\ \vdots \\ x_n^\top \end{pmatrix} \quad y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

When minimizing we seek the roots of the derivative of the loss, by doing so we can derive the optimal solution for beta.

$$\begin{aligned} \frac{\partial L(\beta)}{\partial \beta} = 0 \ &\Leftrightarrow -(y - X\beta)^\top X - (y - X\beta)^\top X = -2 \cdot (y - X\beta)^\top X = 0 \\ &\Leftrightarrow (y - X\beta)^\top X = X^\top y - X^\top X\beta = 0 \\ &\Leftrightarrow X^\top y = X^\top X\beta \\ &\Leftrightarrow (X^\top X)^{-1} X^\top y = \beta \end{aligned}$$

2.1 Non-Linear Features

Sadly the very simple linear regression algorithm only allows for linear functions to be estimated. Fortunately we can make things non-linear by defining a feature function which maps an input x to feature space like so:

$\phi(x) \in \mathbb{R}^l \quad \text{with} \quad l \in \mathbb{N}$

For real valued $x \in \mathbb{R}$ we can create the quadratic feature vector using the following.

$\phi(x) = \begin{pmatrix} 1 & x & x^2 \end{pmatrix}^\top$

When using linear regression on these features we end up with a function that is quadratic in x but linear in features $\phi(x)$.

$$X = \begin{pmatrix} \phi(x_1)^\top \\ \vdots \\ \phi(x_n)^\top \end{pmatrix}$$

$$f_\beta(x) = \phi(x)^\top \beta = \phi(x)_1 \cdot \beta_1 + \phi(x)_2 \cdot \beta_2 + \phi(x)_3 \cdot \beta_3 = \beta_1 + x\beta_2 + x^2\beta_3$$

Even more sophisticated features could be using gaussian functions at the datapoints which then amounts to a radial basis function.

$\phi(x) = \begin{pmatrix} 1 & \phi_1(x) & \cdots & \phi_N(x) \end{pmatrix}^\top \quad \text{with} \quad \phi_i(x) = e^{-\frac{1}{2}||x_i - x||_2^2}$

2.2 Regularization

Maybe you may have noticed that the linear regression algorithm may fail in case that $(X^\top X)$ is not invertible. To ensure invertibility a tiny quantity can be added to the diagonal of the matrix like so: $(X^\top X + \lambda I)$. Interestingly, the optimal solution to the following loss has this exact form.

$$\begin{aligned} L(\beta) &= (y - X\beta)^\top (y - X\beta) + \lambda \cdot \beta^\top \beta \\ \frac{\partial L(\beta)}{\partial \beta} = 0 &\Leftrightarrow -2(y - X\beta)^\top X + 2\lambda \cdot \beta^\top = 0 \\ &\Leftrightarrow \lambda \cdot \beta^\top - (y - X\beta)^\top X = 0 \\ &\Leftrightarrow \lambda \cdot \beta - X^\top y + X^\top X\beta = 0 \\ &\Leftrightarrow X^\top y = X^\top X\beta + \lambda \cdot \beta = (X^\top X + \lambda I)\beta \\ &\Leftrightarrow (X^\top X + \lambda I)^{-1} X^\top y = \beta \end{aligned}$$

The loss function can be viewed as a combination of penalties for β. The first penalty is the mean squared error from before which is large when β cannot reproduce the samples y well. The second penalty is for β of long lenghts, which means that short β with small coefficients are preferred. These two penalties probably are contradictive and the optimal beta will be

somewhere in between the individual minima for the penalties. This kind of regularization using $\lambda\beta^\top\beta$ is called L^2 regularization and is a special case of the general Tikhonov regularization formulating constraints on β as $\beta^\top\Gamma^\top\Gamma\beta$ with Tikhonov matrix Γ. $\Gamma\beta$ is a linear mapping which can be chosen to also formulate special constraints on beta like differences between coefficients, or projections to different vector spaces.

When we use only half of the data set for "training" our model i.e. solving for β we can check how well it reproduces the other half of the data set, which indicates how well the model generalizes. While the model may be very accurate for the data used for training, it may be inaccurate for the data used for "validation" (the other half), which means that the model is "over fitted" to the training data and does not approximate the actually underlying function well. In our case we can use regularization to allow the model to have a less perfect fit to the data when it instead fulfills the other objective of being small. The degree to which the model can violate the mean squared error depends on λ.

2.3 Cross Validation

In order to find the perfect regularization parameter λ for our model to generalize well, we need to try some λs out. This can be done using cross validation. In cross validation, we divide our data set into K euqually sized parts (randomly), then we train our model using K-1 parts of the data and validate the model by computing the mean suared error using the missing part. This is done k times so that each part has been used for validation once and the errors are averaged. The following pseudo code should make the algorithm clear.

1: Split data D into K equally sized parts $\{D_1, \ldots, D_K\}$ randomly
2: **for** $k = 1 \ldots K$ **do**
3: train on data $D_{\setminus D_k}$ yielding β_i
4: calculate MSE using D_k $\quad err_k = L^{MSE}(\beta_k, D_k)$
5: **end for**
6: calculate total MSE as $err = \frac{1}{K}\sum_{k=1}^{K}\frac{err_k}{|D_k|}$

The total error will be a measure for generalization of the model, using this measure an optimal λ can be found by trying different λ and adaptively tightening the search space. For the start it may be useful to try lambdas of different orders of magnitude first.

3 Support Vector Machine

Imagine a binary classification problem where the data set consists of multivariate inputs $x_i \in \mathbb{R}^M$ and discrete outputs $y_i \in \{-1, 1\}$ $(i = \{1, \ldots, N\})$ which means that a point x may either belong to class 1 or -1. For this kind of problem we want to find a function that can separate the points of one class from another. In the 2 dimensional case, a simple separator would be a straight line and a point can either be on one or the other side of it. For higher dimensional x the separator would be a hyperplane.

To get used to the plane stuff again, lets review the point-normal form of a plane. In the simplest case where the plane intersects the origin, we can check if a point is in the plane by projecting its position vector x onto the plane normal n. When x is orthogonal to n the projection $x^\top n = 0$ and x is in the nullspace of n that is the plane. When the plane does not intersect the origin we first need to translate the coordinatesystem so that it does. We can do this by subtarcting a support vector of the plane p_0 (a position vector of a point that is part of the plane) from x, resulting in the check $(x - p_0)^\top n = 0$ which can be rewritten as follows.

$$(x - p_0)^\top n = 0 \Leftrightarrow x^\top n + d = 0 \quad \text{with constant} \quad d = -p_0^\top n$$

If n is of unit length, then the distance of x to the plane is given by the projection onto the normal $dist = x^\top n + d$. When not normalized then the distance is given by the following.

$$dist = (x - p_0)^\top \frac{n}{|n|} = \frac{x^\top n}{|n|} - \frac{p_0^\top n}{|n|} = \frac{x^\top n + d}{|n|}$$

This means that the projection on the normal is the distance to the plane scaled by the normals length $x^\top n + d = dist \cdot |n|$. For a fixed projection value, this means, the smaller the normal the greater the distance. In fact for a projection of value 1 the distance is $\frac{1}{|n|}$. We want the margin, that is the "width" of the separator, to be maximal. If we require projections to be at least 1 (or -1 depending on the side) the margin is $\frac{2}{|n|}$.

Lets formulate the separation objective as a constrained optimization problem with an inequality constraint, from which we can set up a Lagrangian.

$$\hat{n} = \underset{n}{\operatorname{argmin}} \frac{1}{2}|n|^2 \quad \text{subject to} \quad y_i(x_i^\top n + d) \geq 1$$

$$\mathcal{L}(n, d, \lambda) = \frac{1}{2}\sum_{j=1}^{M}(n_j)^2 - \sum_{i=1}^{N}\lambda_i \cdot (y_i(x_i^\top n + d) - 1)$$

When the data does not allow for a linear separation of the datapoints, then this problem cannot be solved because there is no feasible region. But we can modify the problem in order to allow points to be on the wrong side of the plane at a cost. For this a new variable $\xi_i \geq 0$ is introduced which measures by how much the inequality constrained has been violated and sanitizes the constraint so that it becomes feasible. Of course, we want the amount of violations to be minimal, so we introduce a penalty on large ξ which we control by a weighting parameter C.

$$\hat{n} = \underset{n}{\operatorname{argmin}} \frac{1}{2}|n|^2 + C \cdot \sum_{i=1}^{N}\xi_i \quad \text{subject to} \quad y_i(x_i^\top n + d) \geq 1 - \xi_i \text{ and } \xi_i \geq 0$$

$$\mathcal{L}(n, d, \lambda, \mu) = \frac{1}{2}\sum_{j=1}^{M}(n_j)^2 + C \cdot \sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\lambda_i \cdot (y_i(x_i^\top n + d) - 1 + \xi_i) - \sum_{i=1}^{n}\mu_i \cdot \xi_i$$

When solving the dual problem (which is always convex), we get the optimal normal and

classifier as

$$\hat{n} = \sum_{i=1}^{N} \hat{\lambda}_i y_i x_i$$

$$f(x) = \text{sign}(x^\top \hat{n} + d)$$

$$\hat{\lambda} = \underset{\lambda}{\text{argmax}} \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{k=1}^{N} \lambda_i \lambda_k y_i y_k x_i^\top x_k \quad \text{subject to} \quad \sum_{i=1}^{N} \lambda_i y_i = 0 \text{ and } C \geq \lambda_i \geq 0$$

Since λ_i are the lagrangian multipliers of the inequality constraint they are either inactive ($= 0$) if the corresponding point is correctly classified and outside the margin, or active ($\neq 0$) if the corresponding point violates the constraint. So all active x_i contribute to the solution and are called the support vectors.

4 Logistic Regression

We already discussed one classifier, the support vector machine, that is a binary classifier. In this section we will take a look at a different approach to classification. As usual we have multivariate $x_i \in \mathbb{R}^M$ as inputs which are labeled with a discrete variable $y_i \in \mathbb{Y} = \{Y_1, \ldots, Y_l\}$ $(i = 1, \ldots, N)$. This time we want to know the probability of a class y given that we observed some x which is denoted by $p(y|x)$.

To estimate the probabilities we make use of a discriminative function $f(x, y) : \mathbb{R}^M \times \mathbb{Y} \to \mathbb{R}$ that we have to learn. The discriminative function has to have a high value when x is likely to be of class y, and a low value when it's not so that $y(x) = \text{argmax}_y f(x, y)$ where $y(x)$ is the correct class for x.

Similar to the Boltzmann distribution $p(x) = e^{-E(x)}$ where $E(x)$ is an energy (or error), the probabilities are calculated as:

$$p(y|x) = \frac{e^{f(x,y)}}{Z} \quad \text{with normalization} \quad Z = \sum_{y' \in \mathbb{Y}} e^{f(x,y')}$$

Now that we set up the basic framework, we need to define the discriminative function. Lets formulate it as a linear function in features (similar to linear regression).

$$f(x, y) = \phi(x, y)^\top \beta \quad \text{with} \quad \phi(x, y) = \begin{pmatrix} \phi(x)[y = Y_1] \\ \phi(x)[y = Y_2] \\ \vdots \\ \phi(x)[y = Y_l] \end{pmatrix}$$

The $[cond]$ operator is 1 if the condition is true, otherwise 0.

The objective of our model is to maximize the likelihood $p(y_i|x_i)$ for each (x_i, y_i), but instead it could also maximize the log likelihood $\ln(p(y_i|x_i))$ since the logarithm is a monotonic increasing

function. Also, in machine learning we rather like to minimize stuff, so lets minimize the neg-log-likelihood by minimizing the following loss.

$$L(\beta) = -\sum_{i=1}^{N} \ln(\ p(y_i|x_i)\ ) = -\sum_{i=1}^{N} \Big(\ln\big(e^{f(x_i,y_i)}\big) - \ln\big(\sum_{y'\in\mathbb{Y}} e^{f(x_i,y')}\big)\Big)$$
$$= -\sum_{i=1}^{N} \Big(\ln\big(e^{f(x_i,y_i)}\big)\Big) + \sum_{i=1}^{N} \Big(\ln\big(\sum_{y'\in\mathbb{Y}} e^{f(x_i,y')}\big)\Big) = -A(\beta) + B(\beta)$$

The derivative of the loss is then calculated from the two separated sums $A(\beta)$ and $B(\beta)$:

$$\frac{\partial L(\beta)}{\partial \beta} = -\frac{\partial A(\beta)}{\partial \beta} + \frac{\partial B(\beta)}{\partial \beta}$$
$$\frac{\partial A(\beta)}{\partial \beta} = \sum_{i=1}^{N} \frac{\partial f(x_i, y_i)}{\partial \beta} = \sum_{i=1}^{N} \phi(x_i, y_i)^\top$$
$$\frac{\partial B(\beta)}{\partial \beta} = \sum_{i=1}^{N} \frac{\partial}{\partial \beta} \ln\big(\sum_{y'\in\mathbb{Y}} e^{f(x_i,y')}\big)$$
$$= \sum_{i=1}^{N} \frac{1}{\sum_{y'\in\mathbb{Y}} e^{f(x_i,y')}} \cdot \frac{\partial}{\partial \beta} \sum_{y'\in\mathbb{Y}} e^{f(x_i,y')}$$
$$= \sum_{i=1}^{N} \frac{1}{Z} \cdot \sum_{y'\in\mathbb{Y}} e^{f(x_i,y')} \cdot \phi(x_i, y')^\top$$
$$= \sum_{i=1}^{N} \sum_{y'\in\mathbb{Y}} \frac{e^{f(x_i,y')}}{Z} \cdot \phi(x_i, y')^\top$$
$$= \sum_{i=1}^{N} \sum_{y'\in\mathbb{Y}} p(y'|x_i) \cdot \phi(x_i, y')^\top$$
$$\Big(\frac{\partial L(\beta)}{\partial \beta}\Big)^\top = \nabla_\beta L(\beta) = \sum_{i=1}^{N} \Big(\sum_{y'\in\mathbb{Y}} p(y'|x_i) \cdot \phi(x_i, y')\Big) - \phi(x_i, y_i)$$

As can be seen, the gradient of the loss is not linear in β because β sits inside of the non-linear calculation of $p(y'|x_i)$. So unfortunately we cannot analytically derive the optimal β, but instead we can use the Newton algorithm to iteratively get there.

4.1 Newton Algorithm

The newton algortihm can be derived from a taylor expansion of a function. Lets review the taylor expanson first, by writing down a univariate $(n-1)$ order taylor of f around x with step size h.

$$f(x+h) \approx f(x) + \frac{f'(x)\cdot h}{1!} + \frac{f''(x)\cdot h^2}{2!} + \frac{f^3(x)\cdot h^3}{3!} + \cdots + \mathcal{O}(h^n)$$

From this, the forward difference can be derived (which is not what we want, but shows how neat Taylor is).

$$f(x+h) \approx f(x) + f'(x) \cdot h + \mathcal{O}(h^2) \;\Leftrightarrow\; \frac{f(x+h) - f(x)}{h} \approx f'(x) + \mathcal{O}(h^2)$$

Lets do something else, lets set a first order Taylor to zero (root finding):

$$\begin{aligned} f(x_{n+1}) = f(x_n + h) &\approx f(x_n) + f'(x_n) \cdot h + \mathcal{O}(h^2) \\ 0 &\approx f(x_n) + f'(x_n) \cdot h \\ 0 &\approx \frac{f(x_n)}{f'(x_n)} + h \\ h &\approx -\frac{f(x_n)}{f'(x_n)} \end{aligned}$$

Here we derived the stepsize h of a newton step, which we can use to update the position x_n in order to get $x_{n+1} = x_n + h$. When iterating this newton step, we approach a root of $f(x)$.

In our case, we want to find a root of the gradient ∇L, lets again formulate a Taylor expansion, this time multivariate second order with step δ and Hessian $\mathcal{H}_L$.

$$L(\beta_{n+1}) = L(\beta_n + \delta) \approx L(\beta_n) + \nabla L(\beta_n)^\top \delta + \frac{1}{2}\delta^\top \mathcal{H}_L(\beta_n)\delta + \mathcal{O}(|\delta|^3)$$

The derivative of this taylor with respect to the step δ yields an approximation of the derivative of L around β_n. To find a root of it, we set it to zero.

$$\begin{aligned} 0 = \frac{\partial L(\beta_n + \delta)}{\partial \delta} &\approx \nabla L(\beta_n)^\top + \delta^\top \mathcal{H}_L(\beta_n) \\ 0 &\approx \nabla L(\beta_n)^\top [\mathcal{H}_L(\beta_n)]^{-1} + \delta^\top \mathcal{H}_L(\beta_n)[\mathcal{H}_L(\beta_n)]^{-1} = \nabla L(\beta_n)^\top [\mathcal{H}_L(\beta_n)]^{-1} + \delta^\top \\ \delta &\approx -[\mathcal{H}_L(\beta_n)]^{-1} \nabla L(\beta_n) \end{aligned}$$

This is the newton step we can use to iteratively work our way to the optimal β using $\beta_{n+1} = \beta_n + \delta$. What is missing in order to optimize β for logistic regression is the Hessian of the loss $\mathcal{H}_L(\beta) = \frac{\partial}{\partial \beta} \nabla L(\beta)$, which is left to the reader to derive.

5 Neural Networks

Neural networks currently are the very hype in many fields of inteligent systems, probably because they are so flexible. But lets not get carried away and start by defining a regression problem. Given data $D = \{x_i, y_i\}_{i=1}^N$ with multivariate $x_i \in \mathbb{R}^M$ and scalar $y \in \mathbb{R}$ we want to estimate again as in linear regression $y = f(x)$. For this, we model the function f as follows:

$$f(x) = z_{L+1} = W_L \cdot x_L \quad \text{with} \quad x_l = \sigma(z_l) \;\; l \in \{2, \dots, L\} \quad , \quad \sigma(z) = \frac{1}{1+e^{-z}} \quad \text{and} \quad x_1 = x$$

In this equation we have l matrices $W_1 \dots W_l$ with dimensions $W_1^{d(2)\times M}\ W_2^{d(3)\times d(2)} \dots W_l^{1\times d(l)}$. The sigmoid function σ takes the role of the "activation function" which is an analogy to neurons in a brain which can propagate an action potential based on the magnitude of the signal.

Lets get more intuition on this from a simple example. Suppose $x \in \mathbb{R}^3$ and $y \in \mathbb{R}^1$ and we have three matrices $W_1\ W_2\ W_3$. To be able to input our x into the neural net we have to choose the first dimension of W_1 to be 3 and the last dimension of W_3 to be 1 in order to map to $\mathbb{R}^1$. The other dimensions are left to be choosen freely. Lets choose $W_2^{10\times 10}$ which implies $W_1^{10\times 3}$ and $W_3^{1\times 10}$. Our function is then:

$$f(x) = W_3\ \sigma\Big(W_2\ \sigma(W_1 x)\Big)$$

This algorithm is called the forward propagation of x through the neural network. The unknowns in this equation are the entries of the matrices, the so called weights. In order to optimize the weights we need to define a cost function $c(f, y)$ (a loss) for the prediction f. In the literature, we can find many different cost functions that can be used, we'll stick to the well known mean squared error.

$$c(f, y) = (f - y)^2 \quad \text{so that} \quad C_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{N} c(f(x_i), y_i)$$

$$\frac{\partial\ c(f, y)}{\partial f} = 2(f - y)^\top \quad \text{if } f \text{ and } y \text{ were vectors, in our case we can ommit the transpose}$$

Lets call $\frac{\partial\ c(f,y)}{\partial f} = \delta_{L+1}$ as it is the derivative of the loss with repect to z_{L+1} (in our case δ_4 the derivative of $f(x) = z_4 = W_3\ x_3$). Due to the chain rule the derivatives for other layers are recursively calculated like this:

$$\delta_l = \frac{\partial c}{\partial z_l} = \frac{\partial c}{\partial z_{l+1}} \cdot \frac{\partial z_{l+1}}{\partial x_l} \cdot \frac{\partial x_l}{\partial z_l}$$

The derivative of the of the loss with respect to a specific weight of a matrix $W_{l,ij}$ is given by:

$$\frac{\partial c}{\partial W_{l,ij}} = \frac{\partial c}{\partial z_{l+1,i}} \cdot \frac{\partial z_{l+1,i}}{\partial W_{l,ij}}$$

To optimize the weights of the matrices, we feed the neural net all of our data samples x_i and collect the losses $c(f(x_i), y_i)$. We then backpropagate the losses through the net, which means we collect the derivatives of the losss for all weights and sum corresponding derivatives up forming a total gradient for a specific weight. The weights are then updated using a gradient descend step which reads $W_{l,ij} = W_{l,ij} - \alpha \frac{\partial c}{\partial W_{l,ij}}$ with learning rate α (probably $\alpha = 0.1$). And then we iterate forward and backward propagation.

6 PCA

Principle Component Analysis (PCA) is a method for dimensionality reduction. This may be useful when you have very high dimensional data, but want to reduce the amount of data you

want to process in order to learn a function. In this case you may want to get a few of the "most important" parts of your highdimensional data point, or lets say a low dimensional datapoint which encodes the most important aspects of your high dimensional point.

A way to do that, is to find a projection of our data $x_i \in \mathbb{R}^M$ onto $z_i \in \mathbb{R}^P$ where $P \leq M$. Lets look at the projection from the other side, lets say we transform z to x using the following, where V_P is the projection matrix and μ is a vector accounting for a translation in $\mathbb{R}^M$.

$$x \approx V_P z + \mu$$
$$err = \sum_{i=1}^{N} ||V_P z_i + \mu - x_i||^2$$

The error err measures the mean squared error of the transformation of z to x. Using this, we can derive the optimal transformation of x to z when minimizing with respect to z.

$$\hat{z}_{1:N} = \underset{z_{1:N}}{\operatorname{argmin}} \sum_{i=1}^{N} ||V_P z_i + \mu - x_i||^2$$
$$\begin{aligned} 0 &= \frac{\partial}{\partial z_i} \sum_{i=1}^{N} ||V_P z_i + \mu - x_i||^2 = \frac{\partial}{\partial z_i} (V_P z_i + \mu - x_i)^\top (V_P z_i + \mu - x_i) \\ &= 2 \cdot (V_P z_i + \mu - x_i)^\top V_P \\ \Leftrightarrow 0 &= V_P^\top V_P z_i + V_P^\top (\mu - x_i) \end{aligned}$$

If we constrain V_p to be orthonormal (composed of orthonormal vectors), then $V_P^\top V_P = I$ and this simplifies to the followning:

$$z_i = V_P^\top (x_i - \mu)$$

Doing the same for μ yields the following:

$$\begin{aligned} \hat{\mu} &= \underset{\mu}{\operatorname{argmin}} \sum_{i=1}^{N} ||V_P z_i + \mu - x_i||^2 \\ 0 &= \frac{\partial}{\partial \mu} \sum_{i=1}^{N} (V_P z_i + \mu - x_i)^\top (V_P z_i + \mu - x_i) = 2 \cdot \sum_{i=1}^{N} (V_P z_i + \mu - x_i)^\top \\ \Leftrightarrow 0 &= N \cdot \mu \cdot \sum_{i=1}^{N} (V_P z_i - x_i) \\ \Leftrightarrow \mu &= \frac{1}{N} \sum_{i=1}^{N} V_P z_i - \frac{1}{N} \sum_{i=1}^{N} x_i \end{aligned}$$

So μ is the difference of the expeted value of the projected z_i and the expected value x_i. Lets assume that $E[z] = 0$ because that would be nice, and we don't care about the expected value of z as it is unrelated to what we try to achieve. Due to the linearity of the expected value the projection of z also has 0 as expected value.

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i = E[x]$$

So the projection $z_i = V_P^\top (x_i - \mu)$ is projecting the "centered" x_i. For the derivation of V_P lets assume that x_i are already centered ($E(x) = 0$). What we want to minimize is the projection error, which we can estimate by back projecting the projection of x_i and comparing it with the initial x_i.

$$\hat{V}_P = \underset{V_P}{\operatorname{argmin}} \sum_{i=1}^{N} ||x_i - V_P V_P^\top x_i||^2$$

This time we will derive the optimum by observing the properties of the equation instead of root finding in the derivative. Lets reformulate the loss a bit.

$$\begin{aligned}
L(V_P) &= \sum_{i=1}^{N} (x_i - V_P V_P^\top x_i)^\top (x_i - V_P V_P^\top x_i) \\
&= \sum_{i=1}^{N} x_i^\top x_i - 2 x_i^\top V_P V_P^\top x_i + x_i^\top V_P V_P^\top V_P V_P^\top x_i \\
&= \sum_{i=1}^{N} x_i^\top x_i - 2 x_i^\top V_P V_P^\top x_i + x_i^\top V_P V_P^\top x_i \\
&= \sum_{i=1}^{N} x_i^\top x_i - x_i^\top V_P V_P^\top x_i \\
&= C - \sum_{i=1}^{N} x_i^\top V_P V_P^\top x_i \quad \text{with} \quad C = \sum_{i=1}^{N} x_i^\top x_i \\
&= C - \sum_{i=1}^{N} ||V_P^\top x_i||^2 \quad (V_P^\top \text{ composed of row vectors } v_j^\top) \\
&= C - \sum_{i=1}^{N} \sum_{j=1}^{P} (v_j^\top x_i) \cdot (v_j^\top x_i) \;=\; C - \sum_{j=1}^{P} \sum_{i=1}^{N} (x_i^\top v_j) \cdot (x_i^\top v_j) \\
&= C - \sum_{j=1}^{P} ||X v_j||^2 \quad (X \text{ composed of row vectors } x_i^\top) \\
&= C - \sum_{j=1}^{P} v_j^\top X^\top X v_j
\end{aligned}$$

So what we can see from this, is that the loss will be minimal if we maximize $\sum_{j=1}^{P} v_j^\top X^\top X v_j$. We could again find the root of its derivative but, it will be more fun with some clever factorization of $X^\top X$, which is the Eigendecomposition. Since $X^\top X$ is a symmetric matrix, its Eigendecomposition is

$$X^\top X = Q \Lambda Q^\top \quad \text{with orthonormal} \quad Q \quad \text{and diagonal} \quad \Lambda$$

$$\underset{v_{1:P}}{\operatorname{argmax}} \sum_{j=1}^{P} v_j^\top Q \Lambda Q^\top v_j$$

Suppose the eigenvalues $\lambda_1, \ldots, \lambda_M$ are sorted in value descending order on the diagonal of Λ. To maximize the above expression, we need to choose $v_{1:P}$ to be the first P eigenvectors q_i corresponding to the P largest eigenvalues. If we could, we would choose all v_j to be the first eigenvector, as this would truly yield the maximum, but due to our constraint of v_js being orthonormal to each other we can't do that.

One last thing to be mentioned before moving on is, that instead of using the Eigendecomposition of $X^\top X$ to get the optimal V_P we can as well use the singular value decomposition (SVD) of X to get there:

$X = USW^\top \quad$ with orthonormal $\quad U,\ W \quad$ and diagonal $\quad S$
$X^\top X = (USW^\top)^\top (USW^\top) = WS^\top U^\top USW^\top = WS^2W^\top = Q\Lambda Q^\top \implies W = Q \quad S^2 = \Lambda$

7 The Kernel Trick

Remember when we introduced features to make things non-linear in linear regression. We use features to calculate new quantities from a given datapoint such as
$\phi(x) = \begin{pmatrix} 1 & x & x^2 & e^x & e^{-x} & \sin(x) & \cos(x) \end{pmatrix}^\top$. This is a very powerful way to allow for non-linearity but on the downside of it is the question: How to choose features so that some property of the original function or distribution becomes linear? This is not easy and sometimes coming up with the right features is impossible.

The kernel trick is a way to get rid of explicitly defining features. Before diggin in, lets define what a kernel in this sense even is. A kernel is the dot product of the feature vectors of two inputs x and x'.

$k(x, x') = \phi(x)^\top \phi(x')$

This implies that a kernel is symmtric with respect to argument order $k(x, x') = k(x', x)$. Of course when we difine features, we imply a kernel function. If on the other hand we define a kernel function, we imply features. Due to the kernel being a dot product, it is a measure of similarity of the feature vectors and thus also a measure of similarity of x and x'. If we keep this in mind, we can define kernels with respect to the distance of two points, for example a gaussian kernel:

$$k_{\mathcal{N}}(x, x') = \mathcal{N}(||x - x'||, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(||x-x'||-\mu)^2}{2\sigma^2}}$$

The "trick" in the kernel trick is now to get rid of the features $\phi(x)$ by reformulating the problem to only include dot products $\phi(x)^\top \phi(x')$, which are then replaced by kernels $k(x, x')$. Lets illustrate this process by deriving kernel PCA in the following.

7.1 Kernel PCA

In PCA we have following eigenvalue problem.

$X^\top X v_j = \lambda v_j$ where v_j is one of the eigenvectors used for the projection

$\lambda v_j = X^\top X v_j = \sum_{i=1}^{N} \phi(x_i)\phi(x_i)^\top v_j$ (sum of outer products of $\phi(x_i)$)

From this we can extract a calculation for v_j by replacing the dot product of v_j and $\phi(x_i)$ by a new variable b_{ij}:

$$\sum_{i=1}^{N} \phi(x_i)\phi(x_i)^\top v_j = \sum_{i=1}^{N} \phi(x_i) b_{ij} = \lambda v_j$$

$$\sum_{i=1}^{N} \phi(x_i)\alpha_{ij} = v_j \quad \text{with} \quad \alpha_{ij} = \frac{b_{ij}}{\lambda}$$

$$\sum_{i=1}^{N} \phi(x_i)\alpha_{ij} = X^\top \alpha_j = v_j$$

When we substitue this into the initial eigenvalue problem we get the following we get to the following.

$$\begin{aligned} X^\top X v_j = \lambda v_j \quad &= \quad X^\top X X^\top \alpha_j = \lambda X^\top \alpha_j \\ &\Leftrightarrow \quad X X^\top X X^\top \alpha_j = \lambda X X^\top \alpha_j \end{aligned}$$

This is pretty much what we have been waiting for. We generated an expression using matrices of dot products of features, which we can replace by a kernel matrix
$XX^\top_{ik} = \phi(x_i)^\top \phi(x_k) = k(x_i, x_k) = K_{ik}$. This brings us to a new eigenvalue problem which only contains kernel expressions.

$$K K \alpha_j = \lambda K \alpha_j \quad \Leftrightarrow \quad K \alpha_j = \lambda \alpha_j$$

We probably can get to this a little bit more straight forward since we know we want $XX^\top$ in our equation:

$$\begin{aligned} X^\top X v_j = \lambda v_j \quad &\Leftrightarrow \quad X X^\top X v_j = \lambda X v_j \\ &\Leftrightarrow \quad X X^\top X X^\top \alpha_j = \lambda X X^\top \alpha_j \quad \text{with} \quad v_j = X^\top \alpha_j \\ &\Leftrightarrow \quad K K \alpha_j = \lambda K \alpha_j \\ &\Leftrightarrow \quad K \alpha_k = \lambda \alpha_j \end{aligned}$$

Alright we can solve for the eigenvectors in kernel form, but how do we project x to z?
In feature representation we previously had

$z = V_P^\top \phi(x)$ with V_P composed of eigenvectors v_j

We now have A_P composed of eigenvectors α_j where $v_j = X^\top \alpha_j$, which means that $V_P = X^\top A_P$. When substituting this we get:

$$z = (X^\top A_P)^\top \phi(x) = A_P^\top \, X \, \phi(x) = A_P^\top \, \kappa(x) \quad \text{with} \quad \kappa(x) = X\phi(x) = \begin{pmatrix} k(x, x_1) \\ \vdots \\ k(x, x_N) \end{pmatrix}$$

Mission completed, we got rid of all occurences of features ϕ and instead use kernel values $k(x, x')$.

8 Clustering

Clustering is used to guess classes for data points (unlabeled so there is no ground truth $\neq$ classification). So the goal is to find K classes and assign the data points to them in a fashion that similar data is in the same cluster.

Two well known clustering algorithm are K-Means and the Gaussian Mixture Model. Lets briefly discuss K-Means before diving into GMM.

8.1 K-Means

In K-Means, we define clusters by centroids and cluster membership by minimal distant centroid. So a point belongs to the cluster which has the closest centroid to it. The loss that is minimized is:

$$L(\mu) = \sum_{i=1}^{N} \sum_{k=1}^{K} C_{ik} \cdot ||x_i - \mu_k||^2$$

With data points $x_i \in \mathbb{R}^M$ centroids $\mu_k \in domR^M$ and membership matrix C (one entry of a row is 1 others are 0). The algorithm works like this:

1. initialize centeroids μ randomly
2. calculate C (set matrix entries C_{ik} to one where corresponding μ_k is clostest to x_i).
3. calculate new centroids as $\mu_k = \frac{1}{\sum_{i=1}^{N} C_{ik}} \sum_{i=1}^{N} C_{ik} x_i$
4. repeat from (2) until centroids are no longer changing.

This algorithm will only find local minima, which is why we usually do several runs to hopefully find the global minimum or at least a quite good local minimum. The main problem tough is to find a reasonable K. Of course, the loss decreases with higher K, when $K = N$ then every point is its own cluster and the loss is 0. Usually we examine the change of loss for increasing K and stop increasing when the change is not substantial anymore.

8.2 Gaussian Mixture Model

In GMM instead of K centroids we have K gaussians from which we want to describe the whole distribution of points. Each gaussian has a weight called "mixing value" π_k, its mean μ_k and covariance matix Σ_k. The whole distribution of all points is given by:

$$p(x) = \sum_{k=1}^{K} \pi_k \cdot \mathcal{N}(x|\mu_k, \Sigma_k) \quad \text{with} \quad 0 \leq \pi_k \leq 1 \quad \text{and} \quad \sum_{k=1}^{K} \pi_k = 1$$

The loss that is minimized is the neg-log-likelihood of the data (equivalent to maximizing likelihood):

$$L(\mu, \Sigma, \pi) = -\sum_{i=1}^{N} \ln \Big(\sum_{k=1}^{K} \cdot \pi_k \cdot \mathcal{N}(x_i|\mu_k, \Sigma_k) \Big)$$

The algorithm works similar to K-Means but more parameters need to be updated:

1. Initialize mixing values uniformly $\pi_{1:K} = 1/K$, covariances to identity $\Sigma_{1:K} = I$ and means μ_{1_K} randomly.
2. Calculate the posterior probabilities that point x_i belongs to cluster k, $P_{ik} = \mathcal{N}(x_i|\mu_k, \Sigma_k) \cdot \pi_k / Z_i$ with normalization $Z_i = \sum_{k'=1}^{N} \mathcal{N}(x_i|\mu_{k'}, \Sigma_{k'}) \cdot \pi_{k'}$.
3. Sum up the posteriors for each cluster $N_k = \sum_{1=1}^{N} P_{ik}$.
4. Update the mixing values $\pi_k = N_k/N$.
5. Update the means $\mu_k = (1/N_k) \cdot \sum_{i=1}^{N} P_{ik} x_i$.
6. Update the covariances $\Sigma_k = (1/N_k) \sum_{i=1}^{N} P_{ik} \cdot (x_i - \mu_k)(x_i - \mu_k)^\top$.
7. Repeat from (2) until convergence.

The main difference to K-Means is, that we do not have exclusive memberhips C_{ik} but membership probabilities P_{ik} which is like a soft assignment of points to clusters.