The Document

João Morais

November 27, 2019

Abstract

This will be the document where I write everything I know about what I learn during my studies. I hope it serves me in the future as a "go to" document when I need to remind something.

Introduction

This document will focus on definitions, understanding concepts and transmitting intuitions/perspectives.

Each chapter will have the same structure. First the definitions and logic connections between everything with references to derivations and demonstrations that will be in attach at the end of the document. Second an overview of the derivations and how formula connect together to make a quick & easy way of finding the connections for hurry times. And third, simply one or two pages with the most important formulas from that chapter.

The first chapter will be about antennas. The second about microwaves. And the ones after that will consist of applications such as Microwave Links/ Hertzian Beams, Satellites and Radar. Maybe I'll join in a chapter about Probabilistic/Statistic Detection and Estimation of signals and one about some fundamentals of communications.

Please be aware that this is a document always at work. I'll write right here when I think a certain chapter is closed, otherwise it might be target of modifications.

Finally, there's a resource folder that I'll be refferencing frequently. It has many of my notes hand written, books, slides. Things I thought to be interesting or important. Probably all books mentioned in any section will be there.

This folder can be accessed through my Shared Drive Link

Enjoy

Contents

8 Machine Learning - Supervised Learning

Supervised learning concerns the problems where the objective is to predict something based on previous data. The counterpart Unsupervised Learning tries to find patterns in unlabelled data.

More generally, the dataset for supervised learning problems consists on a feature vector **x** and a output vector **y** as opposed to unsupervised learning where everything is features / data.

There are two main types of problems, regression and classification. The only difference between them is the expected output: regression aims to predict continuous outcomes while classification regards separating inputs in classes, thus a discrete output.

Some tools can be used to solve both problems, like Neural Networks. We'll have a look which tools are best for which problems.

8.1 Regression Problems - Least Squares

These are the two most commons types of problems. Probably every supervised learning problem can be *classified* as one of these.

Regression is when the output should be continuous, classification when the output should be in discrete classes.

About the first, a measure to minimize is the difference between our prediction to the value we want to achieve. The Sum of the Square Errors (SSE) is very standard cost function to minimize.

The function that is required to minimize is loss/cost/risk function. Nomenclature wise is a problem... Therefore, the following letters/terms can be will be used interchangibly: L (Loss Function) or J (more used when the weights or coefficients are θ) or R (Risk Function):

$$R = \frac{1}{2m} \sum_{i=1}^{m} \left(y^{(i)} - \hat{y}^{(i)} \right)^2$$

Where $\hat{y}$ is our prediction or estimation of the true value of y and **m** is the number of training samples we have. Therefore $y^(i)$ constitutes the outcome of sample i.

One prediction can be made with:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$$

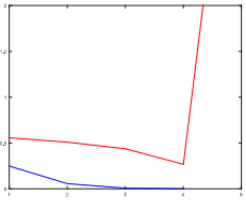
Where **n** is the number of features we are using. Note that features and data points are different things. We can have data regarding only one measure but take the square and the cube of the measure multiplied to different coefficients in order to try to better estimate the function.

β's are usually used in regressions while in neural networks letters like θ or w are more common.

One intuition that is important to have is that **the more we increase the features, the more likely it is that we end up overfitting our training set and loosing generalization capabilities for the actual test data** . This is why it is important to divide all the available data in sets, the model will be trained to guess the training samples, not samples that it never saw before.

- One set for training and one set for testing the prediction capabilities;
- One training set, one validation set and one test set.

The last option is meant to validate the model. The model is trained with the training set, then some parameters are tuned with the validation set, namely the number of polynomial features, regularization term and step size related parameters (like momentum or adaptive step size), and the actual performance is testing in the test set. This way, we avoid optimistic measures of performance by not testing in data used for training.



Average loss in the training set (blue) and in an independent set (red), as a function of polynomial degree.

The result wouldn't be very different if done with the number of iterations or number of features. In particular, it is called doing an "early stop" when the iteration that minimises the loss in the test set is early in the minimization process. It is useful when the model starts overfitting the data.

8.1.1 How to calculate the coefficientes

After having the coefficients, given any other set of data points we can already give predictions on the output.

Note that we are trying to minimize the cost/risk/loss function. The actual cost function would have to be some sort of prediction because it's impossible to know exactly how much the actual outcome will be, despite knowing exactly what the outcome of the model will be to a certain feature vector. As opposed to the empirical risk function where the training outcome and the training predicted result are used, therefore being able to calculate the difference between each estimation and the supposed outcome. To compute the real risk, the expected value of the SSE is necessary. Also, there are continuous results so an integral is required:

$$R = E[y - \hat{y}] = \int_{-\infty}^{+\infty} L(y, x)\phi(y)dxdy$$

Because a potential function to give us a measure on how frequent certain values are is not known, the only way is to approximate the actual error empirically, using the model with some test data. This will degenerate in the actual cost function presented before. L here is meant to denote the loss of one sample which is nothing more than the squared error.

One thing that won't happen in all problems is having an analytical and optimum solution for them. Actually, minimizing the SSE is a kind of problem is called the **Least Squares** . This kind of problem is very usually used in optimisation and often a closed solution is possible.

In this case, since we are searching for the function's minimum, the functions partial derivatives need to be zero in order to have a critical point (maximum, minimum or saddle point).

In this case,

$$\frac{\delta R}{\delta \beta_0} = -2\sum_{i=1}^{n}(y^{(i)} - \beta_0 - \beta_1 x^{(i)}) = 0$$
$$\frac{\delta R}{\delta \beta_1} = -2\sum_{i=1}^{n}(y^{(i)} - \beta_0 - \beta_1 x^{(i)})x^{(i)} = 0$$

Is possible to simplify further these equations, putting the β's in evidence and separating sums, arriving at:

$$\begin{bmatrix} \sum_{i=1}^{n} 1 & \sum_{i=1}^{n} x^{(i)} \\ \sum_{i=1}^{n} x^{(i)} & \sum_{i=1}^{n} x^{(i)2} \end{bmatrix} \begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y^{(i)} \\ \sum_{i=1}^{n} y^{(i)} x^{(i)} \end{bmatrix}$$

Is possible to invert these equations and get the expressions for the coefficients. However there's one important factor to have in mind. Are we aiming at a minimum, maximum or something different like a saddle point? The Hessian Matrix will tell us.

8.1.2 Extrema Conditions and Hessian Matrix

Videos: 87 - Warm up to the second partial derivative test to... 89 - Second partial derivative test intuition

$$H = \begin{bmatrix} \frac{\partial^2 SSE}{\partial \beta_0^2} & \frac{\partial^2 SSE}{\partial \beta_0 \partial \beta_1} \\ \frac{\partial^2 SSE}{\partial \beta_1 \partial \beta_0} & \frac{\partial^2 SSE}{\partial \beta_1^2} \end{bmatrix} = 2 \begin{bmatrix} \sum_{i=1}^{n} 1 & \sum_{i=1}^{n} x^{(i)} \\ \sum_{i=1}^{n} x^{(i)} & \sum_{i=1}^{n} x^{(i)2} \end{bmatrix}$$

In one dimension, to find an extrema is necessary to equalise the first derivative to zero, and the second derivative must be positive - in case of a minimum - or negative - in case of a maximum. If the second derivative is zero at the critical point, then there's an inflexion point. A similar analysis must be done in $(n+1)$-D where n is the number of features used in the regression.

Guaranteeing that the first derivative is zero, which in N dimensions is correspondent to guaranteeing that the gradient is zero in that point, is the first step. Setting $\nabla f = 0$ means that, in that point, the function is not increasing or decreasing in any of the N directions. So the first derivative makes sense.

However, when we go to the second derivative, the meanings get a bit more complicated.

In 2D, if the second derivative was 0, it was certainly a saddle point, if it was > 0 or < 0 it was, respectively, a local minimum or maximum.

The conditions we should impose in 3D is to have a positive (for finding a minimum) or negative (for finding a maximum) definite Hessian Matrix. While it is possible to attribute a meaning to second derivatives in order to just one variable, being nothing more than the concavity in those 2 directions, why do the cross derivatives play a role as well? And, why do they mean really?

Well, first the explanation on why it is needed: there are functions that across multiple dimensions still show that it is an extrema but then there's an inflexion along directions that are not along the axis. So, checking the axis is not enough. Why checking the cross partial derivatives makes it enough?

The Second Derivative Test

$$f_{xx}(x_o, y_o) f_{yy}(x_o, y_o) - f_{xy}(x_o, y_o)^2 \gtrless 0$$

If it is greater than 0, we have a maximum or a minimum and have to check the value of f_{xx} or f_{yy} to be sure. If it is less than 0, we have a saddle point. If it equals 0, then we don't know if it is a saddle point, but it is not a min or max therefore, at least for now, we certainly don't care.

Cross or Mixed partial derivatives can be switched? Yes if the function is C^2. (Boring to prove theorem called: Schwarz' Theorem)

Therefore, we just need to compute one of the cross derivatives.

Also this works because the second derivative test is nothing more than the determinant of the Hessian matrix. The determinant is the product of every eigenvalue of that matrix, therefore it can only be positive if they are both positive or both negative, in which cases there is, respectively, a minimum or a maximum.

But why do eigenvalues tell us this? Because they tell us how the eigenvectors are scaled! And the eigenvectors of such matrix will be the greatest and the least curvatures. Therefore, they either have the same signal / are scaled the same amount, or

Some other links that helped with this:

- Differential Geometry
- Criterior for critical points - Maximum, Minimum or Saddle?
- David Butler - Facts about Eigenvalues

The two main properties of eigenvalues that allow us to quickly calculate them from the Hessian matrix (specially if it is 2x2) are:

$$\mathrm{tr}(A) = \sum_{i=1}^{n} \lambda_i$$

$$\det(A) = \prod_{i=1}^{n} \lambda_i$$

Because there are only 2 variables, there can't be very big changes across more than 2 main directions, so it is possible to quantify the main directions which will be the eigenvectors directions.

The eigenvalues of the Hessian Matrix are also called principal curvatures and the eigenvectors the principal directions.

8.1.3 Analytical Expression for the Coefficientes

From the equation presented in the end of 8.1.1, we can re-write the SSE and the normal equations in the following way.

$$\text{cost function:} \quad SSE(\beta) = \|y - X\beta\|^2$$

$$\text{normal equations:} \quad (X^T X)\hat{\beta} = X^T y$$

And arrive at the analytical expression through the simple inversion of the normal equations. Another way of reaching the analytical expression is deriving the cost function.

$$\text{parameter estimates:} \quad \hat{\beta} = (X^T X)^{-1} X^T y$$

Cost function

$$\begin{aligned} SSE &= \|y - X\beta\|^2 = (y - X\beta)^T (y - X\beta) \\ &= y^T y - 2y^T X\beta + \beta^T X^T X\beta. \end{aligned}$$

Computing the gradient and making it equal to zero

$$\nabla_\beta SSE = -2X^T y + 2X^T X\beta = 0,$$

Note however that:

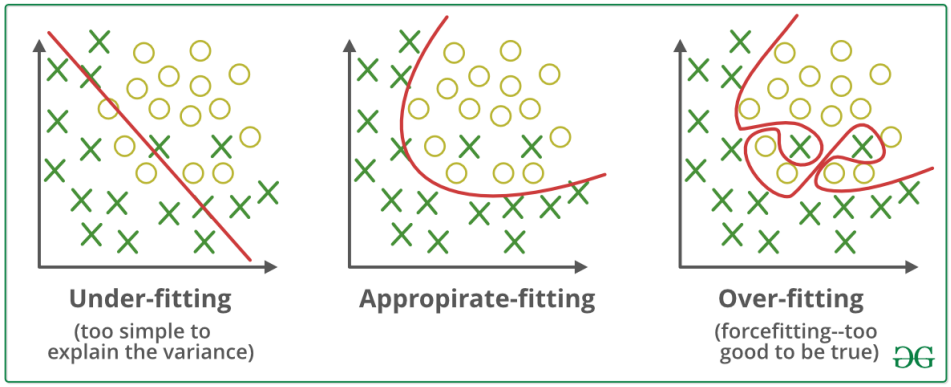
The inverse of matrix $X^T X$ may not exist due to two main reasons:

- small amount of data *e.g.*, number of data points smaller than the number of features.
- redundant features (linearly dependent) *e.g.*, duplicated features.

A final remark on multiple outputs: in case our feature vector serves to estimate more than one output, we can simply use it separately for each output!

8.1.4 Regularization

This is the method of taking importance away from the minimization of the errors between the training set supposed outcomes and the actual model outcomes for those samples. If we don't take importance away, the model may become too good at predicting training examples and may forget that it should predict a tendency and generalize well for the test data.



Performing a regularization consists on nothing more than adding a new parameter to the cost function, in order to shift away the focus of minimizing the SSE.

There are generally two terms that can be added. One with the **norm of the coefficients squared** and the other is the with the module of the coefficients squared.

For the norm squared, if the regularization is applied to a regression - **which is not a necessity since regularization can even be applied to Neural Networks** - it's called Ridge Regression:

$$\hat{\beta}_{\text{ridge}} = \arg\min_{\beta} \|y - X\beta\|^2 + \lambda\|\beta\|^2$$

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

Two key things to note:

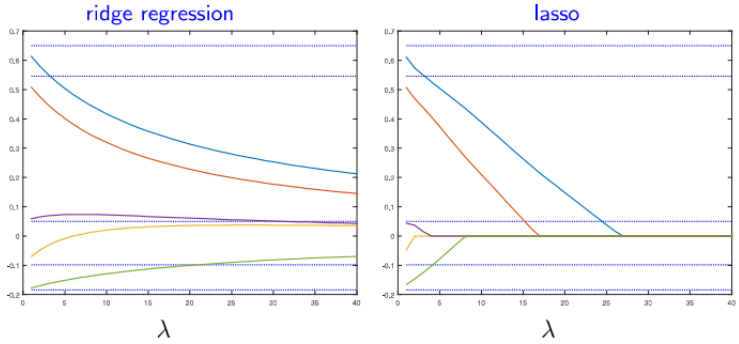
- Note that β_0 is usually not included as the data should be normalised already for much better results. Normalised data means data that has zero mean in every feature and outcome.
- If $(X^T X)$ is singular, the least squares estimate is not unique. Regularization will help finding an estimate even then because $(X^T X + \lambda I)$ is always non-singular.

For the simple norm of the coefficients, when applied to a regression problem it is called the Lasso Regression:

$$\beta_{\text{lasso}} = \arg\min_{\beta} \|y - X\beta\|_2^2 + \lambda\|\beta\|_1$$

$$||\beta||_1 = \sum_{j=1}^{n} |\beta_j|$$

The key difference between the two is that Ridge aims to minimize the norm of all of them while Lasso aims to minimize the sum of module of each of them. Therefore, Ridge it is much likely to pull closer to zero the biggest ones as those are the ones that matter the most for the Euclidean norm, while Lasso will try to pull each of them to 0, there's a direct dependency between a coefficients and the cost function. This is also why the Lasso Regression is called to do feature selection: because if the SSE doesn't depend on the coefficients, the regularization term with the sum of the norms will put that coefficients to zero very quickly.



Again, recall that the data should be centered - have zero mean - and that after calculating the model we need to de-centre it to obtain the real predictions!

How should we proceed if the training data $\mathcal{T} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$ are not centered?

1. pre-processing: $x'^{(i)} = x^{(i)} - \bar{x}$, $y'^{(i)} = y^{(i)} - \bar{y}$ ($\bar{x}, \bar{y}$ average values computed in the training set);
2. estimate linear model without intercept: estimate model $y' = x'^T \beta'$, with $\beta' \in \mathbb{R}^p$, using the pre-processed data $\mathcal{T} = \{(x'^{(1)}, y'^{(1)}), \ldots, (x'^{(n)}, y'^{(n)})\}$ and regularization;
3. invert pre-processing: $\hat{\beta} = [\hat{\beta}_0 \ \hat{\beta}'^T]^T$ where $\hat{\beta}_0 = \bar{y} - \bar{x}^T \hat{\beta}'$;

8.1.5 Optimization problems - Gradient Descent and Newton's Method

The gradient descent is probably the most know method to approximate a functions minimum. By changing the direction of the step we have the gradient ascent.

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)})$$

The above expression works because the gradient points the direction of the maximum growth of the function. Therefore, taking a step in the opposite direction will lead to the minimum.

Momentum and Adaptive StepSize

Momentum $0 \leq \alpha \leq 1$:

$$\Delta \mathbf{x}^{(n+1)} = \alpha \Delta \mathbf{x}^{(n)} - \eta \nabla f[\mathbf{x}^{(n)}]$$
$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \Delta \mathbf{x}^{(n+1)}$$

or, alternatively, by

$$\Delta \mathbf{x}^{(n+1)} = \alpha \Delta \mathbf{x}^{(n)} - (1-\alpha)\eta \nabla f[\mathbf{x}^{(n)}]$$
$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \Delta \mathbf{x}^{(n+1)}$$

We'll use this second version.

Note that we want to pick the $\mathbf{x}$ that brings the function to a minimum. Therefore, the "exponent" will simply refer to the iteration number, not the sample like in the previous section. If α is closer to 1 the memory of the previous increment is more taken into account, meaning that the increment will change only slightly. The closer the parameter gets to 0, the closer we get to the normal situation. This is called the momentum term because it gives the convergence some inercia, the behaviour of momentum. By changing the increment slowly, it may converge faster and have less abrupt changes.

Adaptive Step size, with typical values: $u = 1.2, d = 0.8$:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \eta_i^{(t)} \frac{\partial J}{\partial \theta_i}\left(\theta^{(t)}\right).$$

Step size update

$$\eta_i^{(t)} = \begin{cases} u\eta_i^{(t-1)} & \text{if } \frac{\partial J}{\partial \theta_i}\left(\theta^{(t)}\right) \cdot \frac{\partial J}{\partial \theta_i}\left(\theta^{(t-1)}\right) > 0 \\ d\eta_i^{(t-1)} & \text{otherwise} \end{cases}.$$

Let f be the function where the objective minimum lies, the Divergence criterium is given by:

$$f(x^{(n+1)}) > f(x^{(n)})$$

Where the threshold is found in the equality.

Newton's method

Given by:

$$\theta^{(t+1)} = \theta^{(t)} - \left[H(\theta^{(t)})\right]^{-1} \nabla J_\theta(\theta^{(t)})$$

Where the gradient and the Hessian Matrix are given by:

$$\nabla_\theta J = \begin{bmatrix} \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_d} \end{bmatrix} \qquad H = \begin{bmatrix} \frac{\partial^2 J}{\partial \theta_1^2} & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_d} \\ \frac{\partial^2 J}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 J}{\partial \theta_2^2} & \cdots & \frac{\partial^2 J}{\partial \theta_2 \partial \theta_d} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial^2 J}{\partial \theta_d \partial \theta_1} & \cdots & \frac{\partial^2 J}{\partial \theta_2 \partial \theta_2} & \frac{\partial^2 J}{\partial \theta_d^2} \end{bmatrix}$$

The Newton's method converges insanely fast! But requires the inversion of the Hessian matrix which can be a serious problem...

Newton's Method - Intuition and Demonstration

This is a very very good proof!

8.1.6 How to optimise hyperparameters

There will always be parameters to optimise in order to obtain the best model possible. It was said before that 3 sets should be selected: training, validation and testing. And that it was in the validation set that all hyperparameter tunning should be done. The simplest way is calculate the model with all combinations of the parameters possible and see which one performs better in the validation set.

There can be one other problem: little data. If the data is too little, dividing it into sets can start to give biased results to the accuracy.

One way of calculating the accuracy with more ... accuracy ... is splitting the data in k folds and rotate:

Data: k folds $\mathcal{T}_k$.
for k=1, ..., K **do**
  $f = \text{train}(\mathcal{T} \setminus \mathcal{T}_k)$;
  $P_k = \text{perform}(f, \mathcal{T}_k)$;
end
$P = \bar{P}_k$

Finally, if both things need to be done at the same time, then: 1- k folds need to be made. 2- for each fold, all values of the hyperparameters need to be used for training and tested in the test set. Note: it will be used for training T except (T_i and T_j) that are, respectively, the test set and the validation set. So the hyperparameters testing will be done with T_i. When the all combinations are done, the hyperparameters are selected for the bet one and the actual model is trained with T except T_i depending on the fold considered. 3- use the performances of each fold to get the best average of performance.

(Note that this is not very used...)

Data: k folds $\mathcal{T}_k$.
for $i = 1, \ldots, K$ **do**
  for *all values of* ξ **do**
    for $j \neq i$ **do**
      $f = \text{train}(\mathcal{T} \setminus (\mathcal{T}_i \cup \mathcal{T}_j), \xi)$;
      $P(\xi)_j = \text{perform}(f, \mathcal{T}_j)$;
    end
    $P(\xi) = P(\bar{\xi})_j$
  end
  $\hat{\xi}_i = \arg\min_\xi P(\xi)$;
  $f = \text{train}(\mathcal{T} \setminus \mathcal{T}_i, \hat{\xi}_i)$;
  $P_i = \text{perform}(f, \mathcal{T}_i)$;
end
$P = \bar{P}_i$

8.2 Neural Networks

8.2.1 Formalisation

On the surface, a NN is nothing more than a set of weights connecting a set of neurons. This is represented in Figure 4.

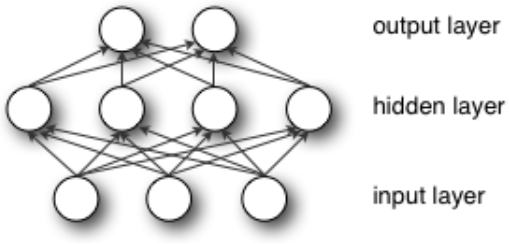


Figure 4: Overview of a Neural Network

This particular architecture is called a Multilayer Perceptron (MLP), the standard NN. In other architectures not all layers are required to be fully connected e.i every neuron from the previous layer is connected to all neurons from the next layer, however, for simplicity, let's restrict this formalisation to MLPs.

Let $w_{ij}^{(l)}$ be the weight that connects the output of the j-th neuron of layer $l-1$ to the i-th neuron of layer l. If we call the output of a neuron j-th of layer l, $z_j^{(l)}$, and the input of the i-th neuron of layer l $s_i^{(l)}$, one may write Equation (3).

$$s_i^{(l)} = w_{i0}^{(l)} + \sum_{j=1}^{N^{(l)}} w_{ij}^{(l)} z_j^{(l-1)} \tag{3}$$

With $N^{(l)}$ being the number of neurons of layer l. Also, the input of the first layer is the input of the network, i.e $z^{(0)} = \mathbf{x}$.

Note further that is possible to relate the input of a neuron to its output through that neuron's activation function $g(x)$. Equation (4) shows this relation, with L the number of layers in the MLP.

$$z_i^{(l)} = g\left(s_i^{(l)}\right) \quad , i = 1, ..., N^{(l)} \quad , l = 1, ..., L \tag{4}$$

Activation functions of neurons may vary across layers and they simply relate the input with the output.

There are many kinds of activation functions, each with its advantages and disadvantages - refer to actFunctions for a more in-depth analysis. In this work only two are used, Rectified Linear Unit (ReLu) and Softmax, which are represented in the set of Equations (5).

$$\begin{cases} \text{ReLu}\left(s_i^{(l)}\right) = \max\left(0, s_i^{(l)}\right) \\ \text{Softmax}\left(s_i^{(l)}\right) = \frac{\exp\left(s_i^{(l)}\right)}{\sum_{j=1}^{N^{(l)}} \exp(s_j)} \end{cases} \tag{5}$$

So far we've see how to get the input to the output - forward propagation is the technical term - but wasn't explained yet how to adjust the weights such that the networks starts behaving like expected. It is done with backpropagation.

Backpropagation is an algorithm that consists of calculating the effect that each weight has on the output and adjust that weight accordingly to that relation and accordingly to how wrong the output is. Backpropagation can be done with Gradient Descent methods and all their associated optimization techniques. The simple version of backpropagation with the classic gradient descent is presented in Equation (6) where η denotes the step size.

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \eta \bigtriangledown_{ij}^{(l)} \quad , \text{with} \quad \bigtriangledown_{ij}^{(l)} = \frac{\delta J}{\delta w_{ij}^{(l)}} \tag{6}$$

The complete expression for the partial derivatives of every weight is in Equation (7).

$$\bigtriangledown_{ij}^{(l)} = \delta_i^{(l)} z_j^{(l-1)} \quad , \tag{7}$$

$$\text{with} \begin{cases} \delta_i^{(l)} = g\left(s_i^{(l)}\right) \sum_{k=1}^{N^{(l+1)}} \delta_k^{(l+1)} w_{ki}^{(l+1)} & , \text{for } l = 1, ..., L-1 \\ \delta_i^{(L)} = g\left(s_i^{(L)}\right) \frac{\delta J}{\delta z_i^{(L)}} & , \text{ otherwise, i.e for } l = L) \end{cases}$$

Note that $z_i^{(L)} = \hat{y}_i$. Additionally, J is the cost/loss function, the function that tells us how far from the correct result the output is. For classification problems, a good cost function usually is Cross Entropy, Equation (8). However, bear in mind that modifying the cost function to one that is more frequently used in Regression problems one can easily use the NN in regression problems.

$$J(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} \mathbb{1}_{y_i = y_k} \log\left(\hat{y}_i\right) \tag{8}$$

Where N is the number of samples and $\mathbb{1}_{y_i=y_k}$ denotes the indicator function that only is 1 when the supposed output is y_k. In other words, the inner sum should always have just one term corresponding to the logarithm of exit of the neural network that should be 1 for that class. This is because due to the common use of Softmax activation function in the last layer, the outputs will be percentages of certainty. And the cost function should be the logarithm of that the certainty of that class only.

More specifically, if we want to categorise images of digits, our neural networks will have 10 outputs, one for each class/digit. When a sample that has the number 3 written on it is propagated until the end, the loss of that computation should be the logarithm of the probability in the 3^{rd} exit, because a perfect NN would output a 1 in the 3^{rd} exit and 0 in the others. In fact, $\log(1)$ is 0 loss and $-\log(0)$ is infinite (positive) loss.

Further optimizations

In order to achieve an efficient implementation, the previous equations can be written in a vectorized way and the forward and backward propagation will be reduced to matrix multiplications. Additionally, for a low level analysis it becomes relevant to keep track of all matrices dimensions, thus they are the following:

- $z^{(l)}$ is $\left(1 + N^{(l)}\right) \times 1$ and $z^{(l)} =$
- $s^{(l)}$ is $N^{(l)} \times 1$ and $s^{(l)} = W^{(l)} z^{(l-1)}$;
- $W^{(l)}$ is the weights matrix and is $N^{(l)} \times \left(1 + N^{(l-1)}\right)$ which should make sense when looking for the above formula and that along its rows are the weights multiplied to the previous layer plus one for the bias unit;
- $\bigtriangledown^{(l)} = \delta^{(l)} \ldots$

Note that feed forward of all samples at once is possible through the correct definition of X matrix and the correct changes.

Some history

Data: k folds $\mathcal{T}_k$.
for k=1, ..., K **do**
  $f = \text{train}(\mathcal{T} \setminus \mathcal{T}_k)$;
  $P_k = \text{perform}(f, \mathcal{T}_k)$;
end
$P = \bar{P}_k$

Pros
It can be proved that the Rosenblatt algorithm solves any binary problem in a finite number of iterations, provided the training data can be separated by a hyperplane in feature space.

However, there's a big problem with only being able to distinguish data that can be separate with an hyperplane: very often the data doesn't behave that way.

How should we choose the number of layers?
Cybenko (1989) proved that a multilayer perceptron with 1 hidden layer is an universal approximator of any continuous function defined on a compact subset of $\mathbb{R}^p$. This is a useful theorem but it does not explain how many units are needed nor how should the weights be chosen.

- Common practice shows that it is often better to use more layers since the network can synthesize a wider variety of nonlinear functions with less units.
- It also shows that deeper networks (with more layers) are more difficult to train.

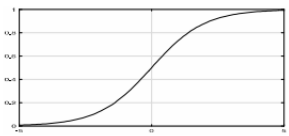
One can make a more complex analysis of the situation...

Activation Functions and Architecture

Some activation functions:

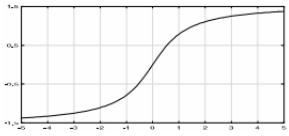
sigmoid: logistic function

$$g(s) = \frac{1}{1 + e^{-s}}$$



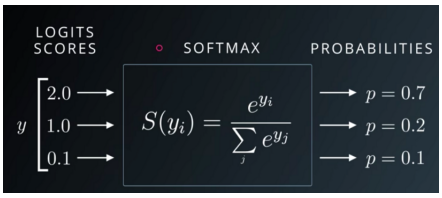
sigmoid: arctangent function

$$g(s) = \arctan s$$



Two other are Rectifier Linear Unit and the softmax:

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



In the all layers with the exception of the last one, ReLu or the logistic function work very well. The last layer must have a function that returns results between 0 and 1, therefore only SoftMax and sigmoid functions like the logistic function would work properly.

Training methods:

The gradient vector includes the contribution of all the training patterns. The weight update using all the training patterns in each iteration is called the batch mode.

$$\Delta w_{ij} = -\eta \frac{\partial \mathcal{R}}{\partial w_{ij}} = -\frac{\eta}{n} \sum_k \frac{\partial L(y^{(k)}, \hat{y}^{(k)})}{\partial w_{ij}} = -\frac{\eta}{n} \sum_k \frac{\partial L^k}{\partial w_{ij}} .$$

Another alternative consists of using one training pattern k, only, and updating the weights with that information. This is called the on-line mode

$$\Delta w_{ij} = -\eta \frac{\partial L^k}{\partial w_{ij}} .$$

A third hypothesis consists of updating the NN weights using a small subset of training patterns. This is known as mini-batch mode.

One very interesting fact is that the image features are not selected by anyone. The network itself crafts its features throught backpropagating the changes required to get the images right. This is true for all Neural Networks.

Also, in image recognition for instance, but in deep neural networks in general, the last layers usually are fully connected.

8.2.2 Neural Networks - BackPropagation

Some of the most useful websites to check while trying to demonstrate the backpropagation algorithm:

- all backpropagation derivatives
- Error (deltas) derivation for backpropagation in neural networks
- 3Blue1Brown Neural Networks - Specially the last one, on backpropagation.

The fastest way is the following:

Hand written demonstration on Drive, link in Intro - the first page.

Note that multiple training modes are possible. The normal one is on-line, where the increment to the weight is nothing more than the step multiplied by the respective partial derivative - Equation (9). Then one can do the batch-mode where all the samples are considered for the weights update - Equation (10). Finally, there's the mini-batch mode that doesn't use all the training samples.

$$w_{ij}^{(l)} = w_{ij}^{(l)} + \triangle w_{ij}^{(l)}$$

$$\triangle w_{ij}^{(l)} = -\eta \frac{\delta C^k}{\delta w_{ij}^{(l)}} \tag{9}$$

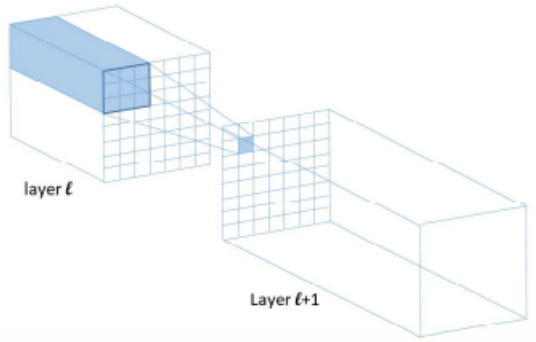
$$\triangle w_{ij}^{(l)} = -\frac{\eta}{m} \sum_{k=1}^{m} \frac{\delta C^k}{\delta w_{ij}^{(l)}} \tag{10}$$

8.2.3 Neural Networks - Convolutional

Convolutional Neural Networks are extremely useful for applications like image recognition. They take a width x height x image depth 3D array and convolve it with a kernel, generating an 2D array.

Because many elements in an image are translation invariant, considering patches of the image is one of the best ways of acquiring features.

Convolutional neural networks allow us to use less inputs to our neural networks. We map a region of a picture to just one pixel/input. We tend to use many different kernels - a set of weights to multiply to each of pixel in the set we chose - to perform this step.



At the end of the convolution, an activation function (like ReLu) is applied.

Note that this convolutional layer can be applied to a 3D array to reduce it to 2D. Moreover, many different kernels can be convolved with the section of the 3D array creating several layers. If there are 20 kernels, we'll have 20 2D arrays, therefore a new 3D array that is all the pixels in the several vicinities, weighted with different kernels.

3D input: $z_{ijk}^{\ell-1}$ $\ell - 1$ - number of input layer

3D kernel: h_{ijk}^{ℓ}

2D output:

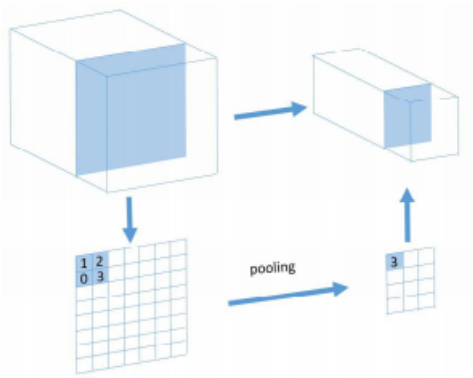
$$s_{ij}^{\ell} = \sum_{p} \sum_{q} \sum_{r} h_{pqr}^{\ell} z_{i+p,j+q,0+r}^{\ell-1}$$

$$z_{ij}^{\ell} = g(s_{ij}^{\ell})$$

Each filter produces a 2D output known as a feature map. Stacking the feature maps produced by multiple filters leads to a 3D array.

Pooling

2D to 2D but with smaller dimensions. Downsize the feature array by choosing what we consider to be the most important values. For instance, for each 4x4 square of pixels, we choose the maximum of them (the highest value in grey scale, for instance). Therefore, the end result will have 16x less pixels/inputs.



Overall, the tendency is the kernel to be smaller, but the network to have many many layers.

8.2.4 Kernels

Why kernels are important? They facilitate a lot the mapping of features to higher dimensions!

Why kernels?

They will become specially important in support vector machines when the data is not separable in certain dimensions but if we increase the dimensions is possible to separate the data by an hyperplane.

8.2.5 Classification Problems

When the required output is a discrete class. Instead of regressing something, the objective is to separate data into classes. Through the learning of what features each class has, be able to classify new data accordingly.

A Bayes Estimator is a Maximum Likelihood estimator, a perfect estimator. However, it requires information that we usually don't have. Estimators like the Naive Bayes are useful because they enable simplifications if one is willing to accept the assumptions they entail.

We'll start with the maximum likelihood estimator and then particularize to Naive Bayes.

Let's assume K classes and y_k being each one. We want to choose y_k such that $k = \underset{k}{\text{argmax}} p(y_k|\mathbf{x})$. This is a maximum likelihood estimator because we want to choose the class that is more likely given the data we received.

From the Bayes Theorem, one can rewrite that probability as $p(y_k|\mathbf{x}) = \frac{p(\mathbf{x}|y_k)p(y_k)}{p(\mathbf{x})}$. Given the denominator as a scaling factor and, normally, an equal probability of each class, one can rewrite the maximum likelihood estimator as:

$$y_k : k = \underset{k}{\text{argmax}}\ p(\mathbf{x}|y_k)$$

Note that the rigorous shape of the above probability is $p(x_1, x_2, ..., x_n|y_k)$, where n is the number of features in our feature vector $\mathbf{x}$.

So far, some considerations have been made but no approximations. The Naive Bayes estimator simplifies the estimation at a cost: assuming the features are independent. If the features are independent of each other, the joint probability distribution degenerates in the multiplication of the marginal PDFs. Likewise, the joint conditional probability density function degenerates in the multiplication of the marginal probability density functions. Mathematically:

$$p(x_1, x_2, ..., x_n|y_k) = \prod_{i=1}^{n} p(x_i|y_k)$$

Further, one should note that due to the logarithmic function being monotone, maximizing a function or the its logarithmic has the same maximizing argument. This is a way of transforming the above multiplication into $\sum_{i=1}^{n} \log(p(x_i|y_k))$.

One particular case where the Naive Bayes Estimator performs decently is language recognition. Not because the letters in the *ngrams* considered are independent - that is certainly not the case, e.g 'ã' is much more likely to be followed by a 'o' in portuguese - but because the assumption of independence throughout the languages has somewhat of the same effect, not influencing the estimation too much. Therefore, assuming independence between characters in a case where there are so many characters and so many *ngrams* combinations to derive our estimator from, doesn't return that bad results and is quite a good application of Naive Bayes, a very simple estimator.

8.3 Support Vector Machines

8.3.1 Linear Classifiers

We call linear methods the classifiers whose decision boundaries are linear, hyperplanes.

A way of classifying multiple classes is to assign to class i, a discriminant $f_i(x) = \left[1x^{\mathrm{T}}\right]\beta_i$.

The purpose of this function is to be 1 when the class of the input is class i, and be 0 when the class of the input is not y_i.

Therefore, decisions are made with $\hat{y}_i : i = \underset{i}{\operatorname{argmax}} f_i(x)$.

Logistic Regression is a Generalized Linear Model (GLM) which can perform prediction and inference while linear Perceptrons can only achieve predictions, and in this case will perform similarly as to logistic regression. Statistical Modelling versus Machine Learning in practice.

In logistic regression, since each output represents a probability, the maximisation of the correct probability is the aim. Therefore gradient ascent (the same apart from a signal) is used.

8.3.2 SVM's

This is a great tutorial on the vector math part of SVM.

Vectors of SVMs

This is a very very good and complete tutorial about SVM: Fletcher: SVM explained

SVMs can only separate between two classes, then is necessary to do strategies like 1 vs All to separate more classes. This is our n sample dataset with p features per sample.

$$\mathcal{D} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in \mathbb{R}^p,\ y_i \in \{-1, 1\}\}_{i=1}^{n}$$

From Fletcher's:

In order to use an SVM to solve a linearly separable, binary classification problem we need to:

- Create $\mathbf{H}$, where $H_{ij} = y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$.
- Find $\boldsymbol{\alpha}$ so that

$$\sum_{i=1}^{L} \alpha_i - \frac{1}{2} \boldsymbol{\alpha}^T \mathbf{H} \boldsymbol{\alpha}$$

  is maximized, subject to the constraints

$$\alpha_i \geq 0 \quad \forall_i \text{ and } \sum_{i=1}^{L} \alpha_i y_i = 0.$$

  This is done using a QP solver.

- Calculate $\mathbf{w} = \sum_{i=1}^{L} \alpha_i y_i \mathbf{x}_i$.
- Determine the set of Support Vectors S by finding the indices such that $\alpha_i > 0$.
- Calculate $b = \frac{1}{N_s} \sum_{s \in S} (y_s - \sum_{m \in S} \alpha_m y_m \mathbf{x}_m \cdot \mathbf{x}_s)$.
- Each new point $\mathbf{x}'$ is classified by evaluating $y' = sgn(\mathbf{w} \cdot \mathbf{x}' + b)$.

We'll now have a look to some of the whys. Note that this is for a Linearly Separable Binary Classification.

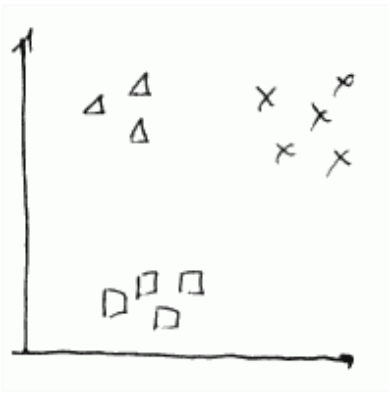
- How about non-Binary? One vs All.
- What if it is not Linearly Separable? Kernels.
- What else, Linear Regression? Yes.
- But the SVM always separates with an hyperplane right, always linearly? Unless a Non-linear SVM is created.

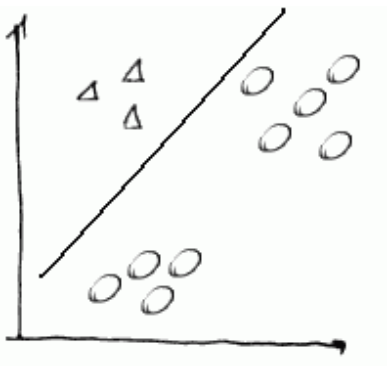
8.3.3 One vs All approach

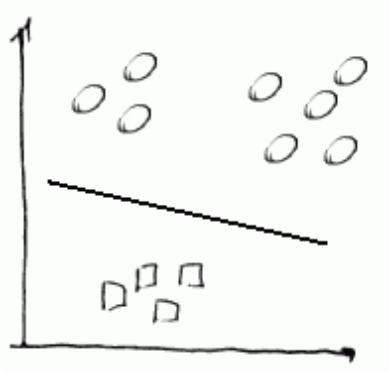
Applies to SMVs and all classification problems that can only distinguish between 2 classes.

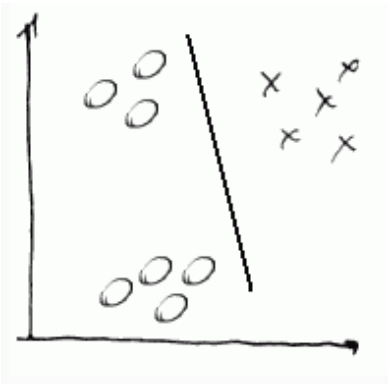
It consists of training K classifiers that will make decisions between the K classes of option. Each classifier will consider the labels of that class against the "other" label that will be all the other samples.

The class with higher certainty is the correct one.









8.3.4 Formulation of SVMs

All the edge cases explained above will be constructed above this, usually being a simple modification.

We want to define a plane in an n-dimension space that divides our data in its classes.

Also, the best plane will be the one that has the highest margin of error, thus choosing correctly with higher probability.

One can define a plane in $\mathbb{R}^n$ like:

$$\vec{n} \cdot (\vec{r} - \vec{r_o}) = 0$$

$\vec{n}$ is the vector normal to the plane, $\vec{r}$ is a random point and $\vec{r_o}$ is the vector that has the point $x_o = (x_1, x_2, \ldots, x_n)$.

For instances, in 3D, calling now the normal to the plane $\vec{w} = (a, b, c)$ and $\vec{x} = (x, y, z)$ and representing vectors in bold:

$$\begin{aligned} & \vec{n} \cdot (\vec{r} - \vec{r_o}) = 0 \\ \Leftrightarrow \; & a(x - x_o) + b(y - y_o) + c(z - z_o) = 0 \\ \Leftrightarrow \; & ax + by + cz + d = 0 \\ \Leftrightarrow \; & \mathbf{w} \cdot \mathbf{x} + b = 0 \end{aligned}$$

To compute the distance of a point to the hyperplane, we simply have to calculate the norm of the projection to the unitary normal vector.

$$dist = \mathbf{x} \cdot \frac{\mathbf{w}}{||\mathbf{w}||}$$

Note now that if we want the distance of the plane to the origin, calling $\mathbf{v}$ to our vector such that $\mathbf{v} = (x_1 - x_o, y_1 - y_o, z_1 - z_o)$, where P_o belongs to the plane and P_1 is the point in question, and $\mathbf{n}$ to our unitary normal just so:

$$\begin{aligned} dist &= \left| v \cdot \frac{\mathbf{w}}{||\mathbf{w}||} \right| \\ dist &= \left| (x_1 - x_o, y_1 - y_o, z_1 - z_o) \cdot \frac{\mathbf{w}}{||\mathbf{w}||} \right| \\ dist &= \frac{|A(x_1 - x_o) + B(y_1 - y_o) + C(z_1 - z_o)|}{\sqrt{A^2 + B^2 + C^2}} \end{aligned}$$

Note now that we need the point in the plane that is closer to the origin to calculate this distance. In fact is possible to find that point by intersecting with the plane a line directed with the normal to the place from the origin. This point however is not required for this analysis. Also, note that if P_1 is the origin, and P_o is

the nearest point to the origin belonging to the plane, because it belongs to the plane $\mathbf{w} \cdot \mathbf{x_o} + b = 0$, so the expression degenerates in:

$$dist = \frac{b}{||\mathbf{w}||}$$

More generally:

1. Is possible to calculate the distance of any point to the plane if we choose any point in the plane and get a vector from that point to the point the distance is required. This is because the plane is not defined uniquely by the orthogonal vector.
2. A sign can be given to the distance $\frac{b}{||\mathbf{w}||}$. If it is below the plane, the distance is negative. If above it is positive. Before only the absolute value was considered, however if we consider w to always be the normal pointing up, we can choose this convention;

This is where everything gets interesting:

This is called Support Vector Machines because they use Support vectors that are the points that are closer together. The amount of support vectors required will depend on the dimensions, in 2 dimensions, 2 points are enough to define a plane of equal distance to both. In 3 dimensions 2 points are not enough because there's a line of points of equal distance and infinite planes can pass through that line. Therefore, 3 points are required. N dimensions, n points to define the problem. Actually, these points will be used to define $\mathbf{w}$. The value of b will be set with the margin between the hyperplanes!

Two hyperplanes can be defined as:

$$\mathbf{w} \cdot \mathbf{x} + b = \pm\delta$$

The actual value of δ won't matter because it won't be more than a factor of scale.

The mid way between the two hyperplanes will be the boundary. And the decision will be based on what side of the boundary the point lies in.

$$x \cdot w + b > 0 \Rightarrow \hat{y} = +1$$
$$x \cdot w + b < 0 \Rightarrow \hat{y} = -1$$

$$\hat{y} = \text{sign}(x \cdot w + b)$$

We want to maximise the distance between the two planes in order to have the boundary as far from the support vector as possible.

Also, may be it is a good idea to take more than the n closest points because those might generate a plane that doesn't divide the data properly. But this can be taken into account later. First, what is the margin between the two described planes and how to maximise it?

Note that since the hyperplanes are parallel, the constant will be the only thing moving them along the normal direction and the distance between them can certainly be deduced directly from that constant.

The constants of both planes are $b \pm \delta$. To calculate the distance between them, is doing $\frac{|D_1 - D_2|}{||w||}$. This formula can be explained by the calculation of the difference of both planes' distance to the origin. Note now that the b will be $b - \delta$.

Therefore, we get:

$$d = \frac{2\delta}{||w||}$$

And to maximize this distance is necessary to minimize $||w||$, with the a constraint per training sample:

Constrains: training data must obey

$$\left.\begin{array}{ll} x^{(i)} \cdot w + b \geq +1 & \text{for } y^{(i)} = +1 \\ x^{(i)} \cdot w + b \leq -1 & \text{for } y^{(i)} = -1 \end{array}\right\} \Rightarrow y^{(i)}(x^{(i)} \cdot w + b) - 1 \geq 0, \ \forall i$$

Constraint optimization is a problem for the Lagrange multipliers. See Section 10.4.2 to see how to apply the method carefully.

The first application gets us to the primary Lagrangean function:

$$L_P = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \alpha_i [y^{(i)}(x^{(i)} \cdot w + b) - 1] \,,$$

$$L_P = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \alpha_i y^{(i)}(x^{(i)} \cdot w + b) + \sum_{i=1}^{n} \alpha_i \,,$$

$\alpha_i \geq 0$ are Lagrange multipliers.

Then, by doing the gradient and replacing it in the above expression:

Optimization

$$\frac{\partial L_P}{\partial w} = 0 \Rightarrow w = \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)} \,, \qquad \frac{\partial L_P}{\partial b} = 0 \Rightarrow \sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \,.$$

Replacing these variables, we obtain the dual formulation,

$$L_D = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i y^{(i)} (x^{(i)} \cdot x^{(j)}) y^{(j)} \alpha_j \,, \quad s.t. \quad \sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \,,$$

Note that this is a problem dual from the first one. We are still solving the same optimization problem but with this substitution, the final expression to optimize depends only on the dot inner product of $x^{(i)}$ and $x^{(j)}$ which will be very important for the **kernel trick** . Also, with the formulation below (after some simplifications) it's possible to use a Converx Quadratic Programming (QP) Solver . See the end of Section 10.4.2 for more info on this Duality.

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \alpha^T H \alpha \,, \quad s.t. \quad \alpha_i \geq 0 \;\; \forall \, i \,, \quad \sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \,,$$

where $\alpha = [\alpha_1 \; \ldots \; \alpha_n]^T$ and $H_{ij} = y^{(i)}(x^{(i)} \cdot x^{(j)}) y^{(j)}$.

This is a convex quadratic programming (QP) problem that can be solved by standard QP algorithms and provides all α_i.

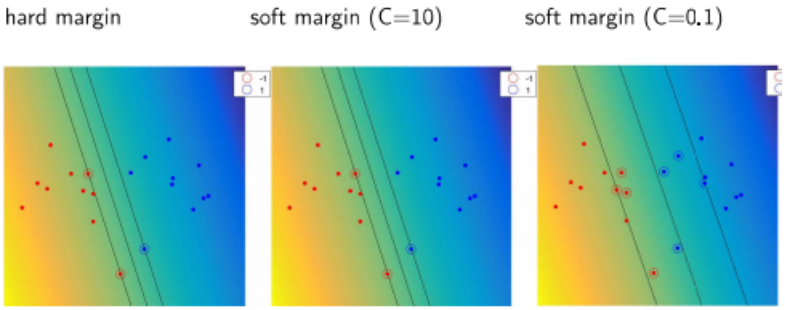
From α's we may obtain:

1. support vectors (S): all $x^{(s)}$ such that $\alpha_s > 0$,
2. normal vector: $w = \sum_{s \in S} \alpha_s y^{(s)} x^{(s)}$
3. offset: $b = \frac{1}{N_s} \sum_{s \in S} \left[y^{(s)} - \sum_{m \in S} \alpha_m y^{(m)} (x^{(m)} \cdot x^{(s)}) \right]$
4. Classification of data: $f(x) = sign(x \cdot w + b)$

In conclusion, with $f(x) = xcdotw + b$, the SVM provides more than a decision, a score, a certainty associated with that decision.

8.3.5 Soft Margin - Slack Variables

This was performed for an hard margin in linear separable data. If the data is not linear separable, either some **Kernel Trick** is performed or some slack is added to account for variables in the other side of the boundary. Slack variables are the penalties that will be added to points in the wrong side of the boundary and thieir sum should be minimised, i.e $C \sum_{i=1}^{n} \epsilon_i$, where C is the scale of the penalty.



8.3.6 Non-Linear SVM & The Kernel Trick

In higher dimensions the data may be separable, but in the current one it may not be.

There are mappings/transformations to increase the feature space dimensionality. With higher dimensions and more complex features that are the weird combinations of the current ones, is possible that there's a separation.

$$\tilde{\mathbf{x}} = \phi(\mathbf{x}) = \begin{bmatrix} \phi_1(\mathbf{x}) \\ \phi_2(\mathbf{x}) \\ \vdots \\ \phi_p(\mathbf{x}) \end{bmatrix}$$

The kernel function $\phi(\mathbf{x})$ will map n features to p, where (usually at least) $p \geq n$.

The biggest problem is that this high dimensionality may drive the problem unfeasible since that each sample will have much more information. And the process of mapping all samples, apply the SVM and them map the boundary back to the first feature space is quite time consuming.

Instead of that, since the dual formulation just requires the inner product of features, is possible to define a kernel function that is the product of the two mappings and compute the inner products in the low dimension input space instead of really having to climb the dimensionality ladder.

$$k(x^{(i)}, x^{(j)}) = \phi(x^{(i)}) \cdot \phi(x^{(j)})$$

The typical kernels:

linear: $k(x^{(i)}, x^{(j)}) = x^{(i)T} x^{(j)}$

rbf: $k(x^{(i)}, x^{(j)}) = e^{-\frac{1}{2\sigma^2}\|x^{(i)} - x^{(j)}\|^2}$

polynomial: $k(x^{(i)}, x^{(j)}) = (x^{(i)T} x^{(j)} + a)^b$

The linear kernel is the one adopted in linear SVM.

We note that some kernels depend on hyperparameters that have to be specified or learned during the training phase *e.g.*, typically by *ad hoc* procedures or by cross validation.

9 Machine Learning - Unsupervised Learning

9.1 Reinforcement Learning and Decision Making

An agent to make the wisest decision in its situation has to have knowledge on what state he is in, how the environment will evolve, how it will be like if he performs a certain action (taking into account stochastic environments) and a utility function to know how much that state contributes to its happiness.