Using Machine Learning For DOTA2 Match Prediction

Andrius Buinovskij

April 24, 2017

1 Dictionary

There are some more obscure terms littered throughout the text. Here is a short explanation for each one.

- A **feature** is some aspect of something. For instance, a coin may have a mass, an area, an average colour and so on. Each of those are a feature.
- **Classification** is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known.[1]

[1]Wikipedia.

2 Introduction

This text describes four contemporary machine learning techniques, and then shows their application in match outcome prediction in a game of Dota 2, namely, which team will win, based on players' hero selection. In other words, it is classifying team composition as preferring a radiant win, or preferring a dire win.

The game in question, Defence of the Ancients (DOTA) 2, is very nuanced and has a plethora of mechanics. For the purposes of this paper however, it will suffice to say that in a game of DOTA, two teams, "Dire" and "Radiant"[2], five players per team, try to destroy the other team's base. At the beginning of a game, each player chooses a "hero" to play, a character within the game. Different heroes have different abilities, and some heroes synergize, whilst others cancel each other out. As of writing of this text, there are 113 different heroes, resulting in a bewildering number of possible team compositions.

[2]Dire being evil and red, Radiant being green good.

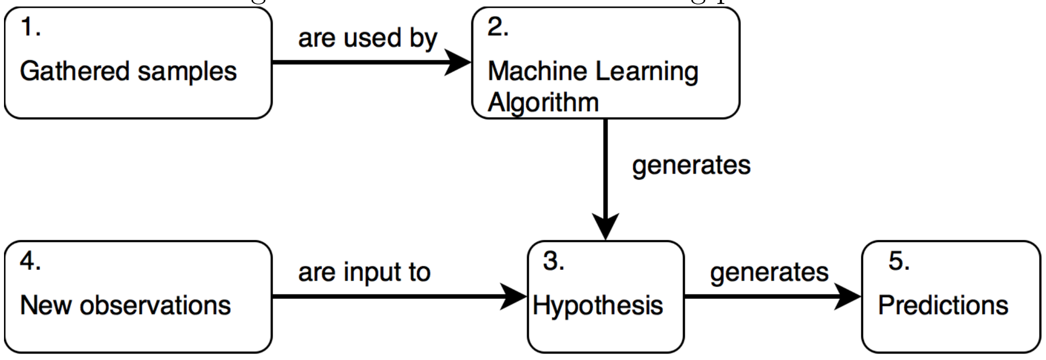
3 Background

The machine learning techniques used are: Nearest Neighbours, Neural Networks, Support Vector Machines and Boosting.

3.1 Machine Learning Generalities

In general, the (supervised, since this is what this particular project is about) machine learning pipeline looks something like this:

Figure 1: General machine learning process.



We begin with some dataset of examples. This could be a list of coins, their value and their mass, or it could be a list of houses and their square footage, number of bedrooms, the price they sold for and so forth. The dataset will be split into the data, and ”the label”, which will be the feature the algorithm will try to learn about.

Then we pass this dataset to the machine learning algorithm. The algorithm will ”train” a model, called a hypothesis for historical reasons, basically adjusting parameters of the model such that the model will perform as well as possible on the dataset, which will hopefully generalize to out-of-sample performance.

What is meant by performance is that, in classification, each sample will belong to some category. The more of the training sample the algorithm classifies correctly, the better is it's in-sample performance. This could be, given the square footage and number of bedrooms of a house, predict whether it is worth more than 500,000 Euro. The algorithm is trained such that it will classify correctly as many training samples as possible.

This trained model or hypothesis will then be able to take some new observations, perhaps a new coin or a house description, and predict the label.

The hypothesis is formed using the training set, but is validated using a testing set. This means that the hypothesis is tested on a set of samples that it was not trained on to see how well the hypothesis will perform in the "real world", or how well it will "generalize" to out-of-training set samples.

For a more in-depth explanation, please see[1].

3.2 Nearest Neighbours

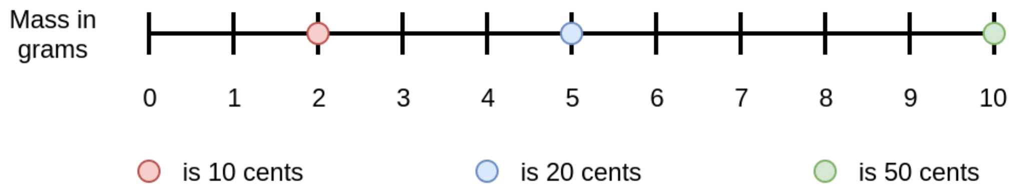
3.2.1 Nearest Neighbours Intuition

The simple logic behind the nearest neighbours approach to machine learning is: when asked to classify a new observation, one should see if the scenario in question has occurred before within the training set, and see what was done then. If the precise scenario has not occurred, then one should find the most similar scenario on record for guidance.

Let us say that we are trying to build an algorithm which will be used as a part of vending machines, namely, deciding on the value of coins that are inserted into the machine. Let us also say that the value of the coin will be decided based solely on it's mass (for simplicity), and that we have a training set of coins already weighed and labelled.

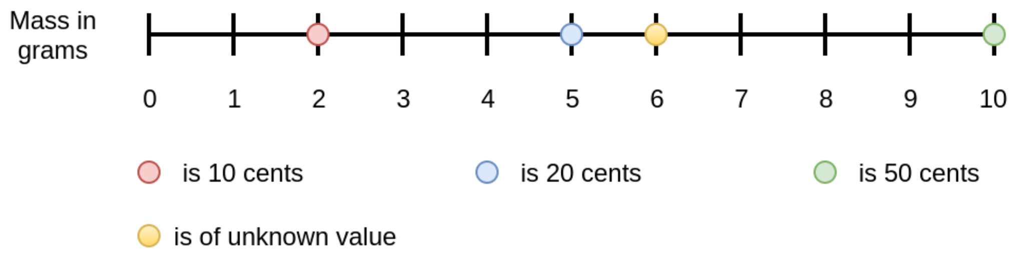
A coin is inserted into the vending machine, how can we tell it's precise value? All we have to go by is the coin's mass. Currently, our training set is illustrated below.

Figure 2: A visualization of coins placed on mass axis.



So we have three coins of mass of 2, 5 and 10 grams. Let us say a coin is inserted into the machine, of mass of 6 grams. Here it is on our graph.

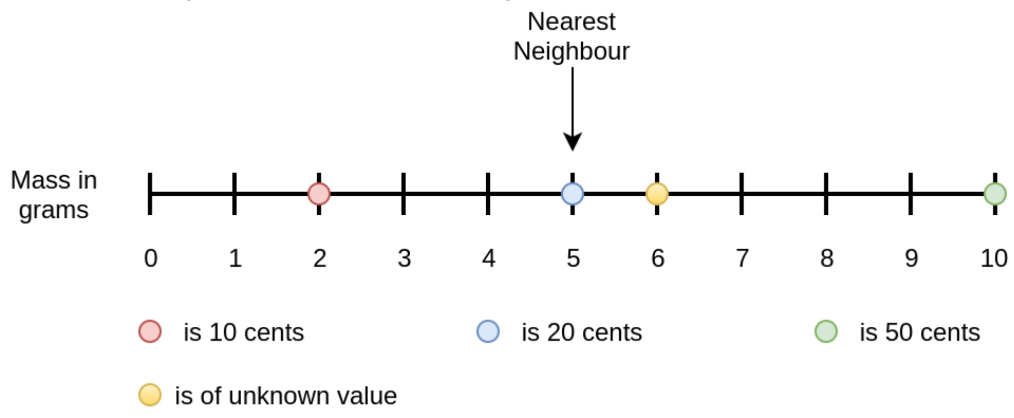
Figure 3: Coin graph including mismatching coin.



We would like to decipher the value of the newly inserted coin. An immediately obvious approach is to query our training set of already known mass-value pairs of coins to find a coin that most closely resembles the new coin, and say that the two are the same. Within the graph, this is simply looking for the nearest neighbouring coin[3]. Here it is again on our graph.

[3]Hence the name of the method, Nearest Neighbours.

Figure 4: The nearest neighbour to the unknown coin



Having found the nearest neighbour, the 20 cent coin, we conclude that the new coin must also be a 20 cent coin, perhaps with some imperfections.

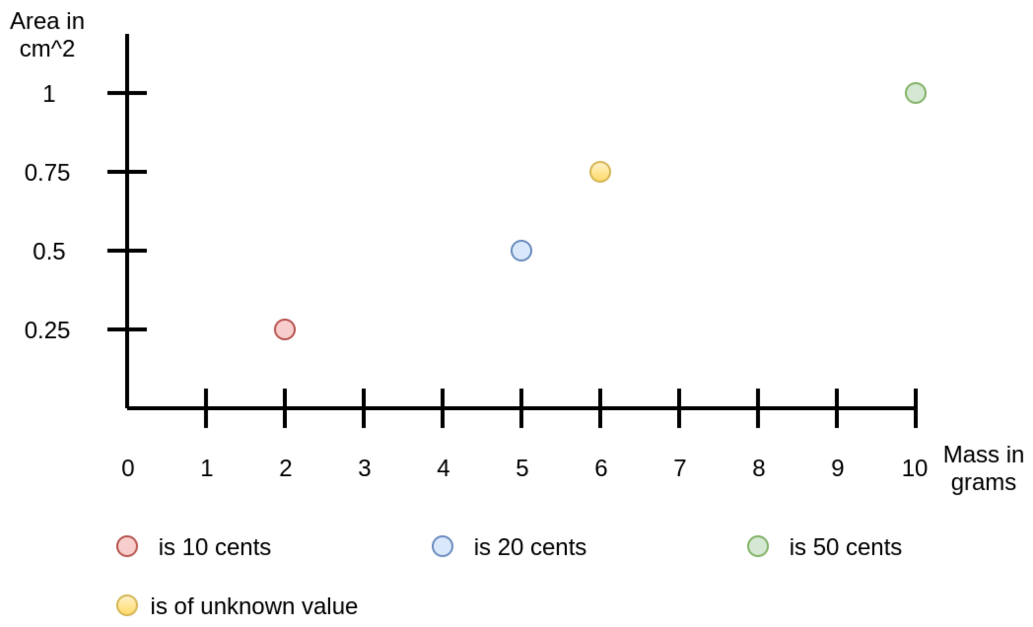
This short example illustrates the benefits and shortcomings of Nearest Neighbours. The method is simple and logical, and directly uses experience of the past. However, it is not extracting underlying patterns out from underneath the data, but is instead simply[4] looking to the past for similar cases.

[4]Not for long.

3.2.2 Adding complexity and Trees

Our coin example was quite simple, using a single feature, and just three neighbour entries. A single feature gives rise to a single axis, however, we could have used more. For instance, we could have used the coin's area, which would have looked like the following:

Figure 5: A visualization of coins places on area and mass axes



The search for a nearest neighbour is now a two-dimensional one, which we can still handle intuitively. However, as the number of features, and consequently axes, grows, the question of how does one find nearest neighbours arises.

So, how does one find the nearest neighbours, and do so efficiently? Presented are three algorithms.

3.2.3 Brute force

Let the set of exemplar coins (the 10, 20 and 50 cent coins) be **S** and the newly observed coin be **T**.

The brute force approach to finding k number of nearest neighbours is to simply calculate the distance[5] between **T** and every member of **S**, and keep the k nearest neighbours.

Unfortunately, this method has a running time of *O(ckn)*[6], where c is the cost of calculating the distance function, k is the number of nearest neighbours desired and n is the number of exemplars in the set **S**.

3.2.4 Kd-trees

The idea behind kd-trees is to build a tree structure to segment the set **S** such that to find the nearest neighbours, one only has to examine one of those segments[7]. This is similar to how a human would look for a nearest neighbour on a graph. Instead of checking every point, no matter how far away on the graph, we would focus our attention on the nearby points, since the far away points are irrelevant.

[5]Preferably a non-negative, symmetric and satisfying the triangle inequality function.
[6]or, less a less naive cn
[7]Not quite, more on that later

Here is a new set of exemplars **S**, to better demonstrate the algorithm.

Id	Feature A	Feature B	Feature C
0	2	8	4
1	1	4	3
2	3	5	7
3	10	2	3
4	2	12	1
5	7	4	8
6	5	4	9
7	2	7	1

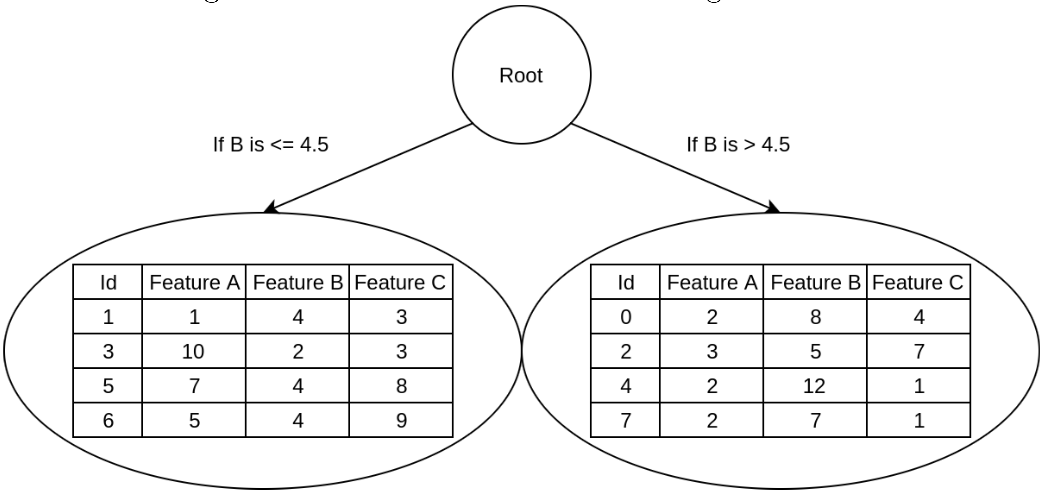
Each row corresponds to a single exemplar, where the exemplars have three features each, namely A, B, and C.

Now we are going to split **S** in two. A question arises of how to best do this. The answer is to pick the feature with the largest variance, and then choose the median value from the current set of exemplars being split.[4][8]

[8]The reasoning is explained under "The Optimized k-d Tree", page 7.

On the first pass through, the feature with the greatest variance is B. Now we choose the median value, which in this case is 4.5 (4+5/2, even numbers). So now all exemplars which have a value of B which is less than or equal to this median go to the left branch, whilst all greater than 4.5 go to the right branch, as follows:

Figure 6: A visualization of the first segmentation.

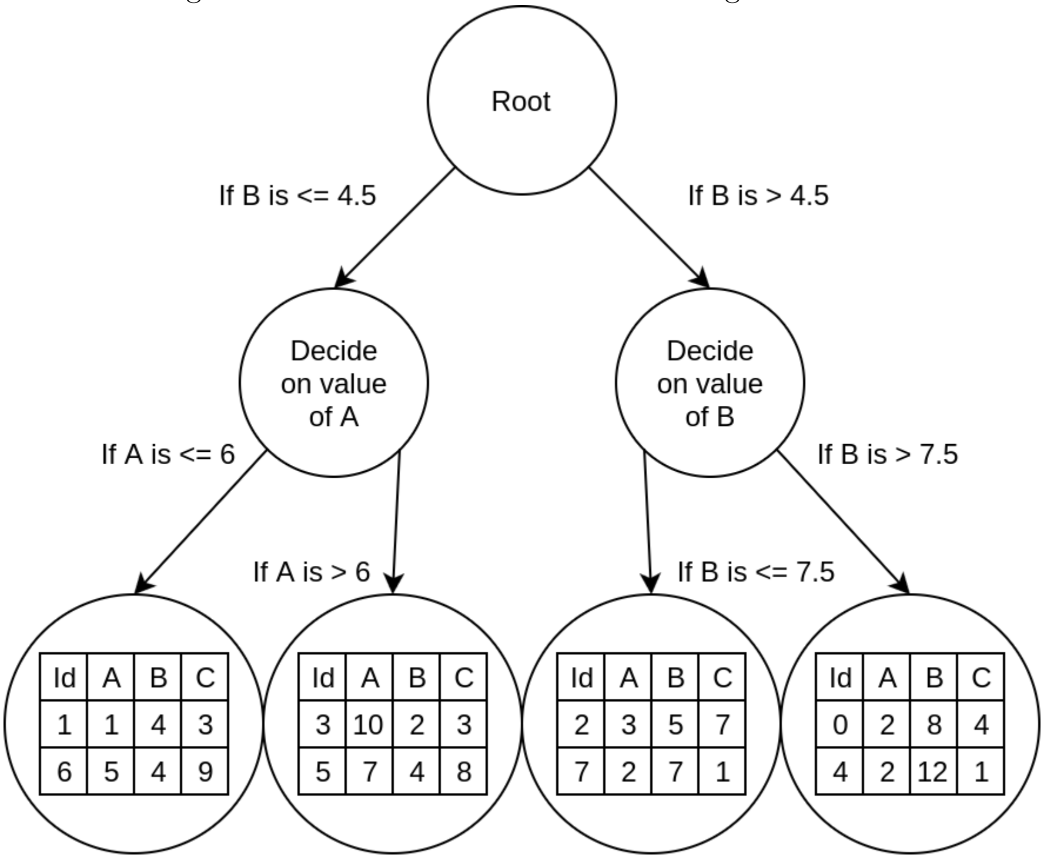


Id	Feature A	Feature B	Feature C
1	1	4	3
3	10	2	3
5	7	4	8
6	5	4	9

Id	Feature A	Feature B	Feature C
0	2	8	4
2	3	5	7
4	2	12	1
7	2	7	1

Now we do this again on both children nodes of the root. On the left-hand node, the feature with the greatest variance is A, and the median is 6. On the right-hand node, the feature with the greatest variance is B, and the median is 7.5. This is the result:

Figure 7: A visualization of the second segmentation.



Id	A	B	C
1	1	4	3
6	5	4	9

Id	A	B	C
3	10	2	3
5	7	4	8

Id	A	B	C
2	3	5	7
7	2	7	1

Id	A	B	C
0	2	8	4
4	2	12	1

Now the tree is 2 levels deep. The number of levels, and consequently the number of sections the exemplar set **S** is subdivided into is up to the person running the algorithm.

That concludes the construction of the tree. Say now we have observed a new exemplar, with values A: 3, B:7, C:5. To find the nearest neighbour, we simply traverse down the tree structure. At the root, our B is 7, which is more than 4.5, and so we take the right branch. Now again we decide on B, but B is less than or equal to 7.5, so we take the left branch. From there we manually calculate distance to each of the exemplars (2 and 7) and take the closer one. We only needed to compute the distance 2 times, instead of 7 for all of the nodes.

Note however that this means of finding the nearest neighbour is approximate, and is not guaranteed to find the true nearest neighbour. For that, please see the "bounds-overlap-ball" and "ball-within-bounds" functions in [4].

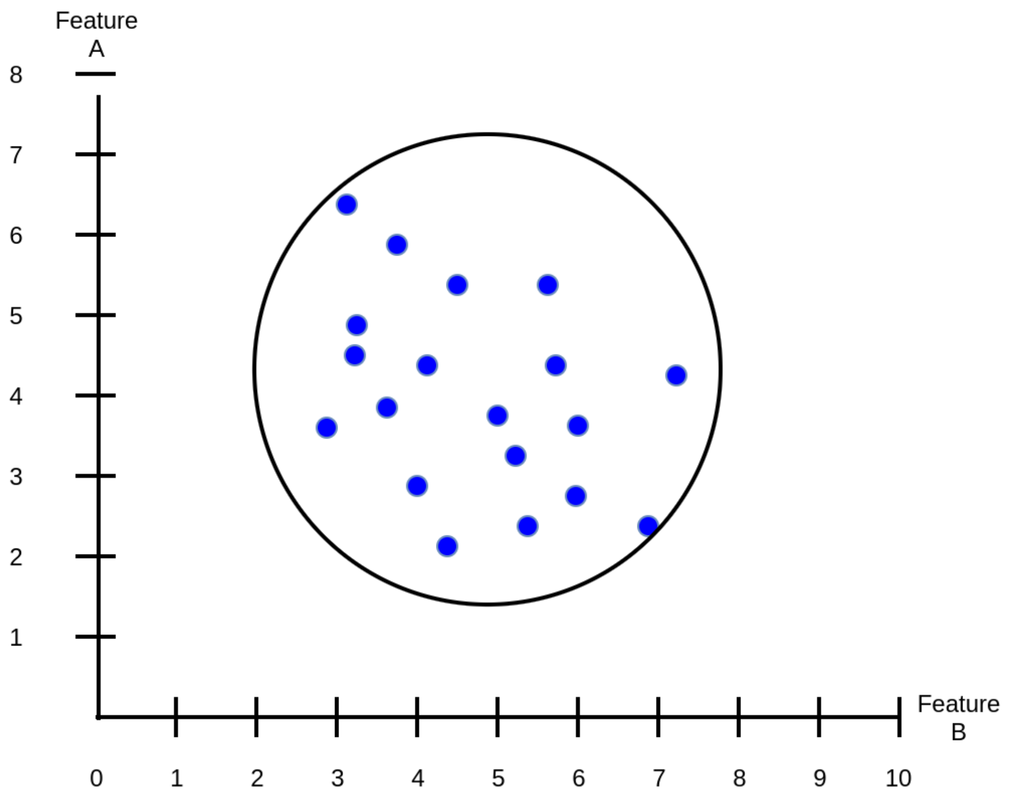
3.2.5 Balltree

There are many versions of the balltree algorithms[5], however, here we will describe the most basic version for the sake of illuminating the concept.

The idea behind balltrees is quite similar to kd-trees, in that we are still essentially building a tree structure to segment the exemplars such that not all of them have to be examined when looking for a nearest neighbour. However, the method in which we segment the exemplars is centred around subdividing them with hyperspheres.

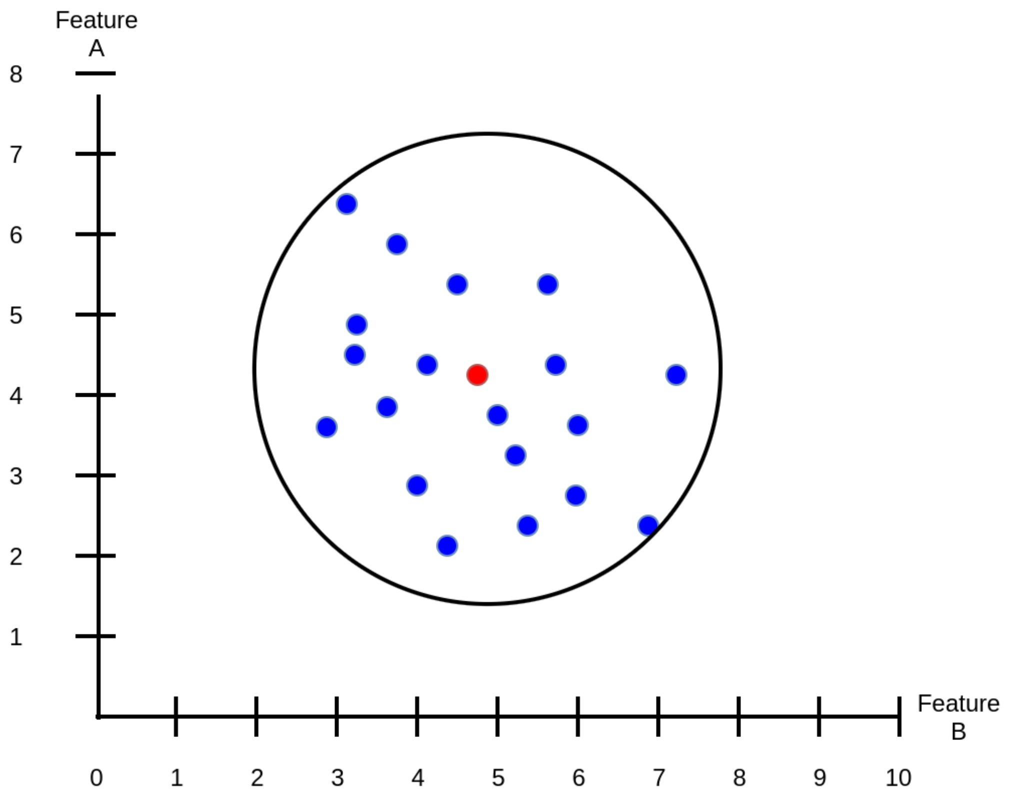
Suppose a balltree is to be built from the following exemplars:

Figure 8: The root of the balltree.



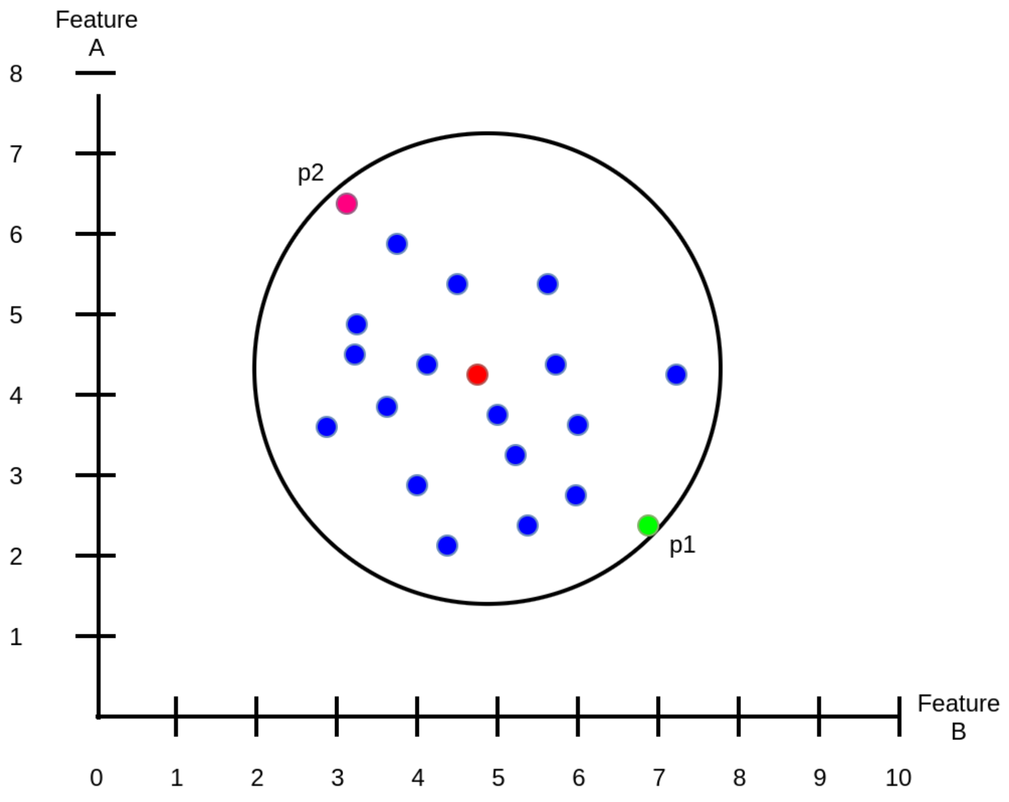
Currently we have one node, the entire exemplar set. To divide it, we calculate the centroid of the current node, which looks like:

Figure 9: The root of the balltree, with centroid in red.



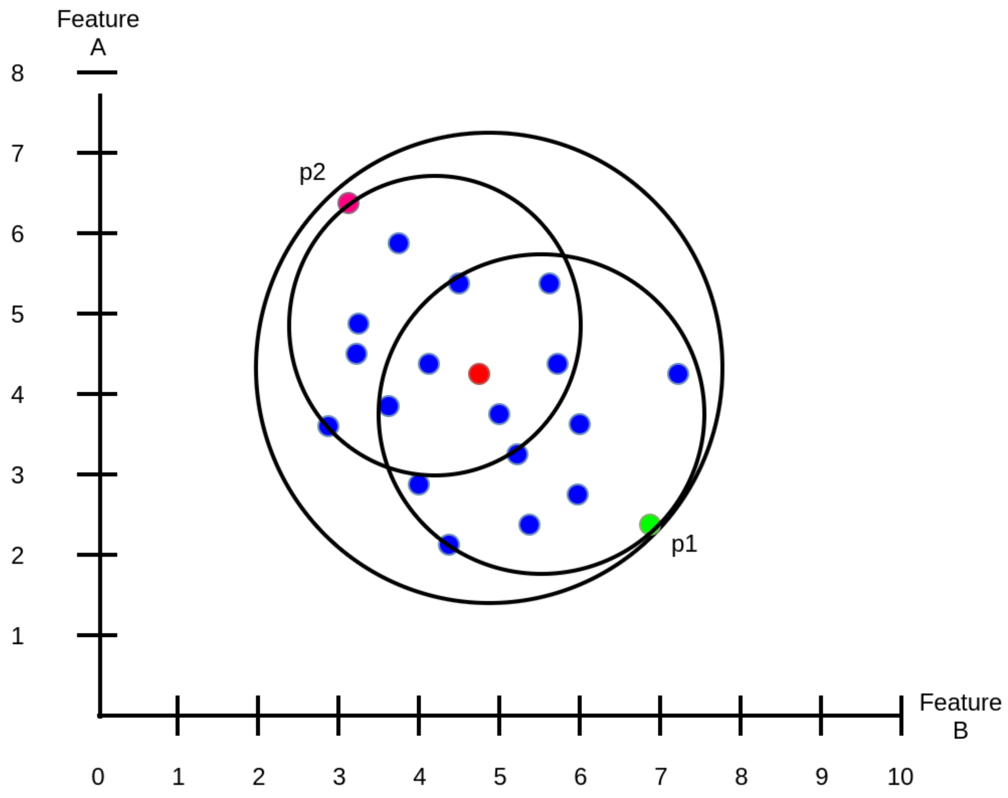
Now we pick the point the furthest away from the centroid, call this *p1*. Then pick a point farthest away from *p1*, and call it *p2*.

Figure 10: The root of the balltree, with centroid in red, p1 in green and p2 in purple.



Now we calculate the distance of each point to p1 and p2. Points closer to p1 become one subset, and points closer to p2 become the other subset:

Figure 11: Two new segments of the set.



Note that the circles can intersect, but each of the exemplars belongs only to one of the two new segmentations. This is done again and again until the desired granularity is achieved.

So how does one query a balltree structure? There are multiple ways, some more efficient than others[5], but at the core we are still simply segmenting the tree and then only looking at the relevant, close-by segments, or in this case, relevant hyperspheres. The hyperspheres and thus all the points assigned to them are decided to be relevant via some distance guarantees[9], the derivation for which can be found at [5].

[9]which are only possible if a proper metric is used.

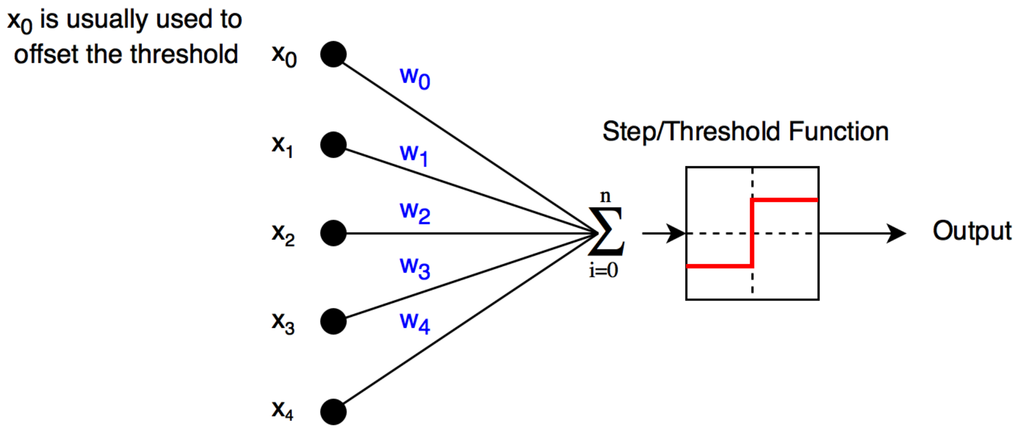
3.3 Neural Network

There are two parts to neural networks, namely the individual neurons, and how they are connected to form a network.

3.3.1 The Neuron

The basic idea behind a neuron is: a neuron receives a set of input signals, and if the sum of those input signals is greater than the threshold, then the neuron "fires" and outputs True, "1", or some indication of excitation. Visually it looks like:

Figure 12: A simple neuron.



A neuron has n inputs, in this case, 5. Each of the inputs x_i has with it associated a weight, w_i, by which the input is multiplied. This is how different factors are emphasized/diminished, and what is adjusted during "training" of the neuron.

The inputs, multiplied by their respective weights, are then summed (so x_0w_0 + x_1w_1 + ... x_4w_4 in this particular example), and passed through a threshold function. In the end it looks like:

$$\text{output} = \textit{threshold}(\sum_{i=0}^{n} x_{\text{i}} w_{\text{i}})$$

A threshold function simply fires if the summation is more than some value. So, for example, if the threshold is 0 and if the inputs x_0w_0 and so on sum to 0 or more, the output will be 1, and 0 otherwise.

The input x_0 is defined as a constant 1 and the weight w_0 is used to adjust the threshold. If, for instance, w_0 is 0.5, then the entire summation of inputs is essentially shifted to the right by 0.5. If the threshold is set at 0, but it would be preferable to have it at -0.5, we can set w_0 to 0.5. Now, if the inputs, aside from x_0w_0 sum to -0.5, 0.5 is added (since 1 * 0.5 = 0.5), the entire summation becomes 0, and the threshold function outputs 1.

That's most of what there is to it. Let us take an example of a neuron which is trying to predict whether a home costs more than 500,000 Euro. The features x_0, x_1 and so on can be square footage, number of bedrooms and so forth. Different features may be more or less suggestive of the cost of a home, for instance a single square meter increase in area may scarcely affect the price of a home, but an additional bedroom could have a significant impact. Consequently, it would make sense to attach a greater weight w to the number of bedrooms than to the square footage of the property.

But how would one automatically adjust those weights? The answer, in this particular case, is gradient descent.

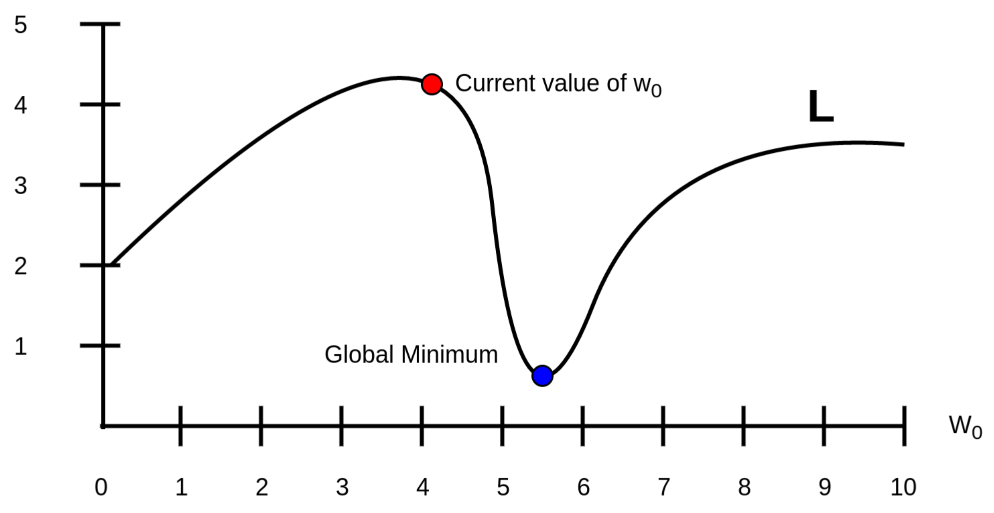
3.3.2 Gradient Descent

Gradient descent (and it's sibling, gradient ascent) is a simple learning algorithm concerned with utilizing the slope of some abstract space.

The intuition is as follows: suppose we formulate a "loss" function **L** which captures the performance of our neuron, and a set **S** of houses and their details, as well as whether or not they sold for more than half a million Euro. What **L** captures is how many of our training samples does our neuron classify correctly.

The performance of the loss function depends on the weights we have assigned to each of the inputs, so the function is actually L(W), where W is the set of weights. Suppose there is only a single weight (in order to have a 2-dimensional graph) that we initialize randomly, then the loss function may look like:

Figure 13: Example loss function.

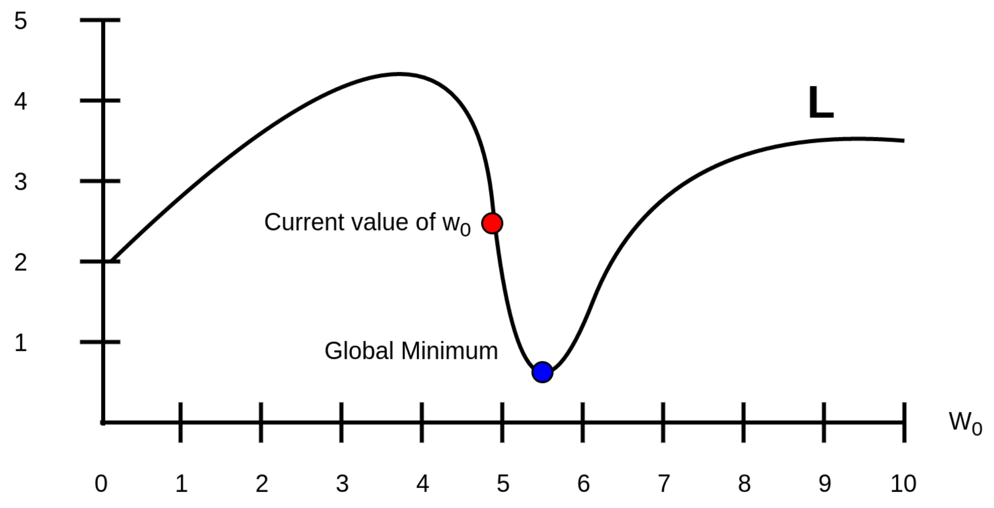


Now, as we adjust the weight (current red point), the magnitude of our loss function will alter. We can traverse that loss function by changing the weight w. If we change the weight w such that the value loss function decreases, this corresponds to having greater accuracy within our sample, since that is what the loss function captures.

What gradient descent does is alter the weight in correspondence with the slope of the line. Take our current point, which is fortunately on the cusp of a precipice towards the global minimum of the loss function. The slope at this point is negative, meaning as the value of the variable on the x-axis increases, the value of the variable on the y-axis decreases. We are trying to descend towards the lowest point[10], so a negative slope tells us we should increase the value of w to reduce the value of **L**.

When we do so, we might get here (depending on some parameters):

Figure 14: Adjusted w.



The iteration then continues until the slope is fount to be 0 at the global minimum.

This is a greedy algorithm, and one can see that should w_0 have been initialized to the value of 2 or so, the gradient descent algorithm would have

[10]Hence the term gradient descent.

taken it towards the left, rather than the right. Thus, gradient descent is susceptible to "local minima", or little kinks in the graph it may get stuck on. For more details, please see [2][11].

Gradient descent is defined as follows:

$$\mathrm{W_i} = \mathrm{W_i} \text{ - } \alpha \frac{\partial L(W)}{\partial w_{\mathrm{i}}}$$

To explain what is going on here. The $\mathrm{w_i}$ is the weight we are currently adjusting, let us say the the weight that will be multiplier of the number of bedrooms in our neuron. The reason for the subscript i is because we may have a whole lot of these, one for each feature, and each of them will be updated differently, since the magnitude of the weights of each of the features will (hopefully) be different. So as we alter the weights, the weight for the number of bedrooms should hopefully increase in larger increments than the weight for the square footage.

The α there is the "learning step size". The larger the α, the greater the size of the updates to the weights will be. One may use a constant alpha, or one may reduce it as the number of updates thus far increases and so forth.

Now that partial derivative, $\frac{\partial L(W)}{\partial w_{\mathrm{i}}}$. As You can see, the derivative is of our function L, which should capture the performance of the neuron. The W is the set of all the weights (in our example neuron, this set is of size 5). The L is the function that we are trying to descend or ascend, as the case may be (descend if it is minus α, ascend if plus). Finally, the partial derivative is with respect to the current $\mathrm{w_i}$, since that is the particular weight that we are currently concerned with. This will all make much more sense with an

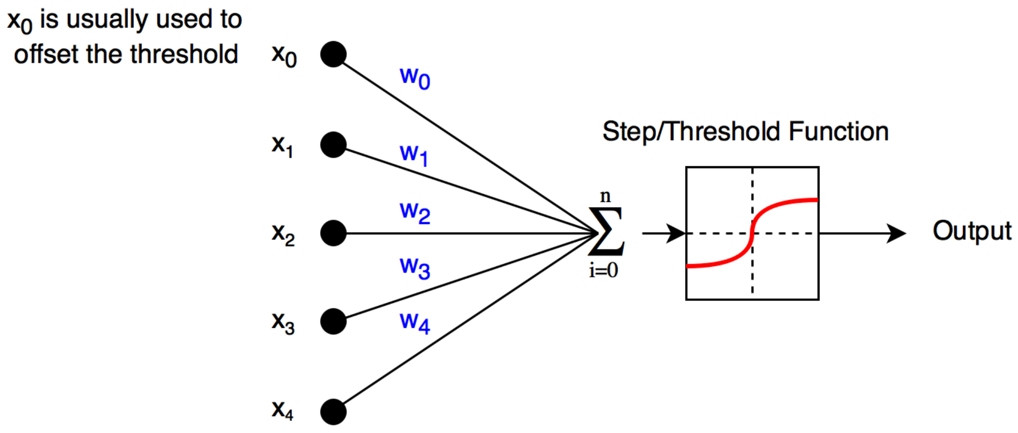
[11]Relevant part starts at 53:15 minute mark.

example.

3.3.3 Training the Neuron

All that is left is to apply gradient descent to the neuron and the weights. Unfortunately however, our threshold function is non-continuous and consequently non-differentiable. This is remedied by using a ”soft” threshold[8][12], which makes our neuron diagram look like:

Figure 15: A simple neuron, with a soft threshold.



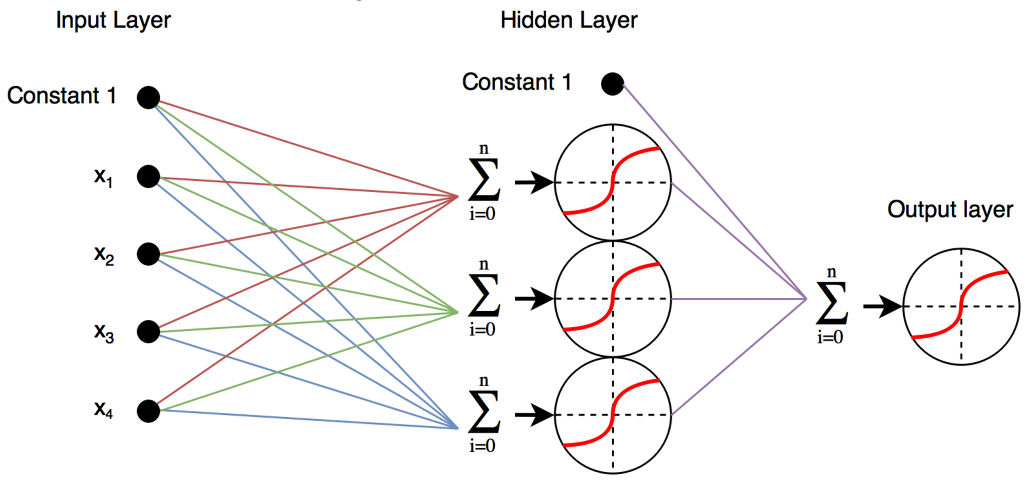
The details of the differentiation can be found at [2][13], but the intuition should be sufficient for a general understanding.

[12]Explained at 25:00 minute mark.
[13]Explained at the 25:00 minute mark, not a duplicate of above footnote.

3.3.4 A Network of Neurons, or a Neural Network

Here is what a neural network looks like visually:

Figure 16: A neural network.



As the name would suggest, it is simply a network of neurons.

Note the "hidden" layer, so termed because it is not strictly visible to anyone outside the algorithm. Each of the outputs of the hidden layer have a weight attached to them, just as did the inputs. The network can consist of more than one hidden layer, and the number of neurons in each hidden layer may differ (giving rise to a technique known as autoencoding[7]).

A neural network is trained with gradient descent (either stochastic or batch [3]). However, a problem arises in the differentiation of the neural network, namely, how does one efficiently find the derivative of the entire neural net with respect to each of the weights w_i? The change in some weight w_i affects all downstream neurons, making the derivations very involved[14].

The answer is an algorithm named "backpropagation". A very broad

[14]Explained in much more detail in [8], 43:00 minute mark.

overview is that by deriving from the output layer backwards (hence back propagation) we can re-use partial derivatives making the differentiation feasible. The algorithm is explained in great detail in [3], at the 33:00 minute mark.

Otherwise, the process by which a neural network is trained is extremely similar to how a single neuron is trained: We are still just optimizing parameters, namely the weights, in order to minimize the value of the loss function, which minimization is done by using gradient descent. The core difference between training a neuron and a neural network is the specialized algorithm for generating partial derivatives with respect to each parameter, backpropagation.

3.4 Support Vector Machine

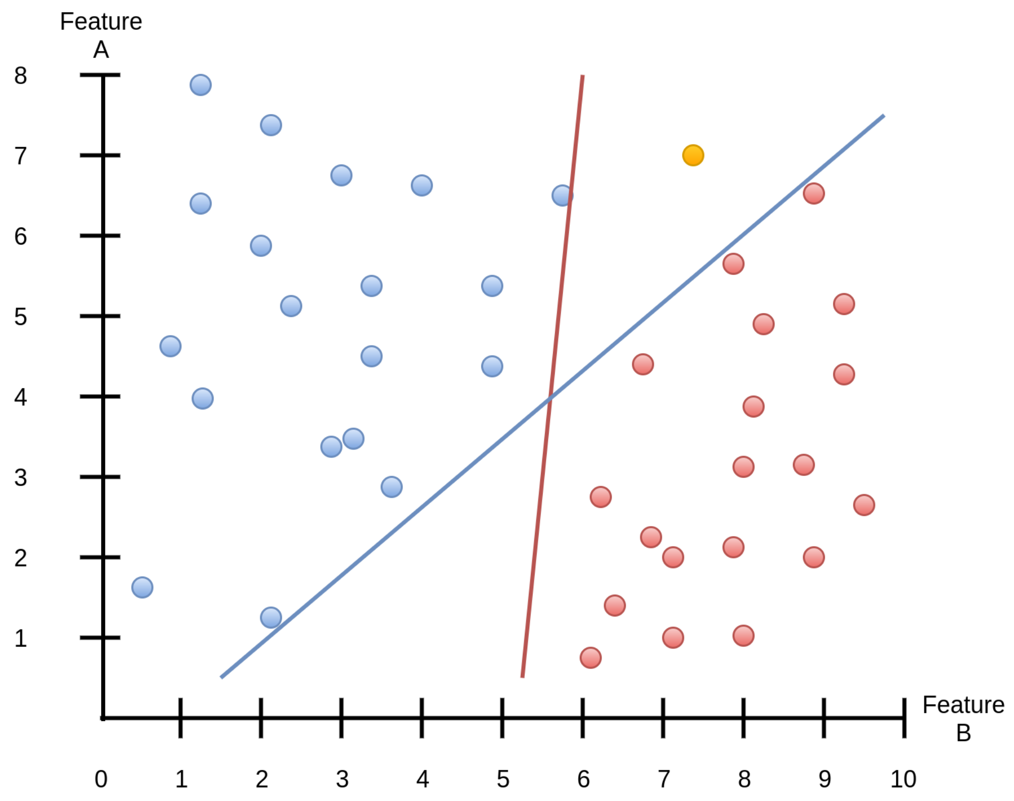
3.4.1 Intuition

There are two core ideas behind support vector machines: optimal linear separators and kernel functions.

3.4.2 Optimal Linear Separator

Picture the following two-dimensional training set:

Figure 17: A linearly separable set.



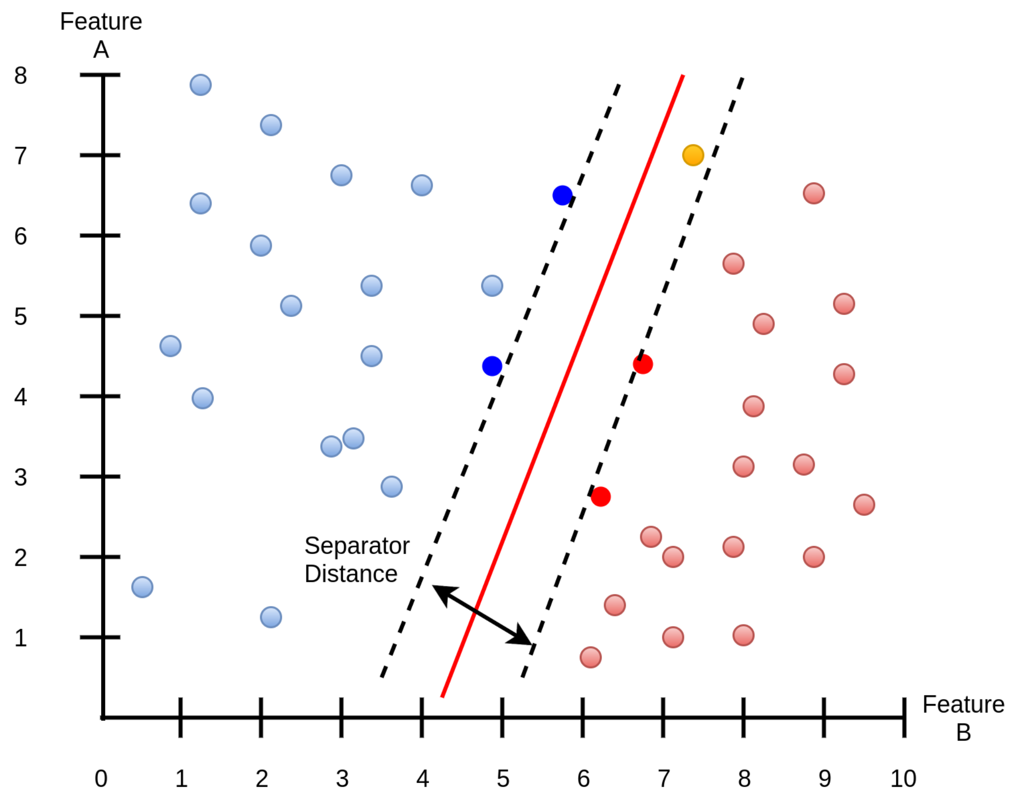
The above is a ”linearly separable” set, which simply means that the blue points and the red points can be separated by a straight line[15].

[15]Which, straight lines are linear functions, hence the term linearly separable.

In the figure, there are two linear separators, the blue and the red. Each gives a different classification to the newly observed sample, which is in gold. Which is the better linear separator? Furthermore, an infinite number of possible separators exist, rather than just the two pictured. How would one go about picking the "best one", and what metric would one use to decide?

One idea is to try to make a separator that puts as much space between the two clusters (red and blue) as possible, in hopes that that will most faithfully capture the underlying relationship. That separator looks something like:

Figure 18: A better classifier, allegedly.



To explain some of what is going on there. Picture that the punctuated lines are the edges of a road, and that the red separator in between them marks the center of the road.[16]

References

[1] Yaser Abu-Mostafa. *Lecture 01 - The Learning Problem.* 2012. URL: `https://www.youtube.com/watch?v=mbyG85GZ0PI`.

[2] Yaser Abu-Mostafa. *Lecture 09 - The Linear Model II.* 2012. URL: `https://www.youtube.com/watch?v=qSTHZvN8hzs`.

[3] Yaser Aby-Mostafa. *Lecture 10 - Neural Networks.* 2012. URL: `https://www.youtube.com/watch?v=Ih5Mr93E-2c`.

[4] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. "An Algorithm for Finding Best Matches in Logarithmic Expected Time". In: *ACM Trans. Math. Softw.* 3.3 (Sept. 1977), pp. 209–226. ISSN: 0098-3500. DOI: `10.1145/355744.355745`. URL: `http://doi.acm.org/10.1145/355744.355745`.

[5] Ting Liu, Andrew W Moore, and Alexander Gray. "New algorithms for efficient high-dimensional nonparametric classification". In: *Journal of Machine Learning Research* 7.Jun (2006), pp. 1135–1158.

[6] Patrick Wilson. *16. Learning: Support Vector Machines.* 2014. URL: `https://www.youtube.com/watch?v=_PwhiWxHK8o`.

[7] Patrick Wilson. *Lecture 12b: Deep Neural Nets.* 2016. URL: `https://www.youtube.com/watch?v=VrMHA3yX_QI`.

[16] This pedagogical device is from [6]

[8] Patrick Wilson. *Lecture 12b: Neural Nets*. 2016. URL: `https://www.youtube.com/watch?v=VrMHA3yX_QI`.