

Using Machine Learning For DOTA2 Match Prediction

Andrius Buinovskij

May 1, 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

¹Wikipedia.

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.

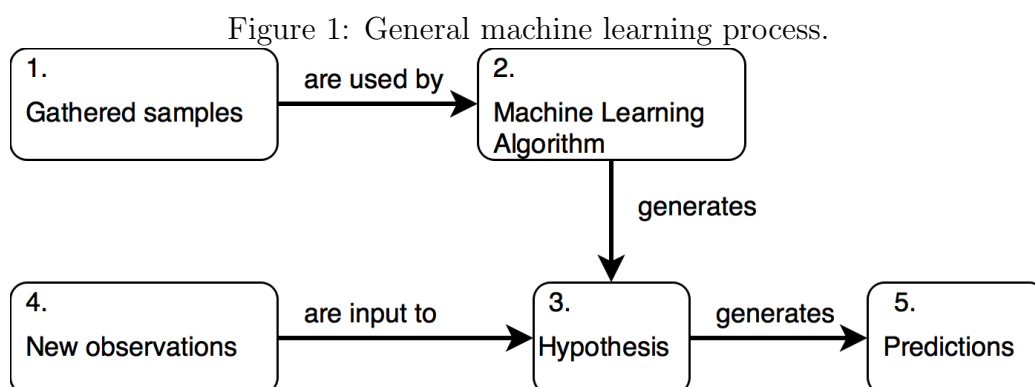
²Dire being evil and red, Radiant being green and good.

3 Background

The machine learning techniques used are: Nearest Neighbours, Neural Networks, Support Vector Machines and Boosting. The intuition and broad strokes of each are explained below, with pointers to more information should it be desired.

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:



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", the label being 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, 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 its 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.³

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

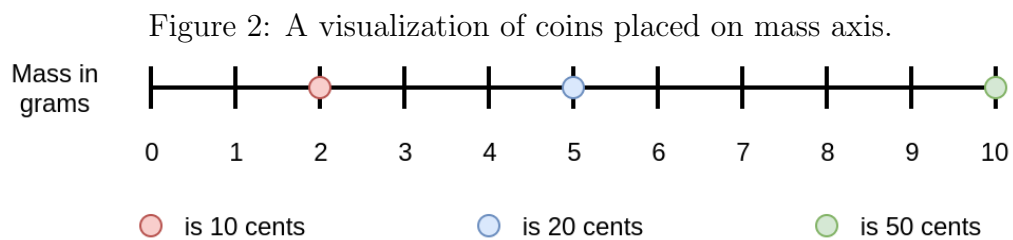
The simple logic behind the nearest neighbours approach to machine learning is: when asked to classify a new observation, one should see if the scenario in question has occurred before within the training set, and see what was done then. If the precise scenario has not occurred, then one should find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

³Something called regularization aside. Regularization hinders in-sample performance in order to avoid over-fitting. All the details can be found at [4] and [5].

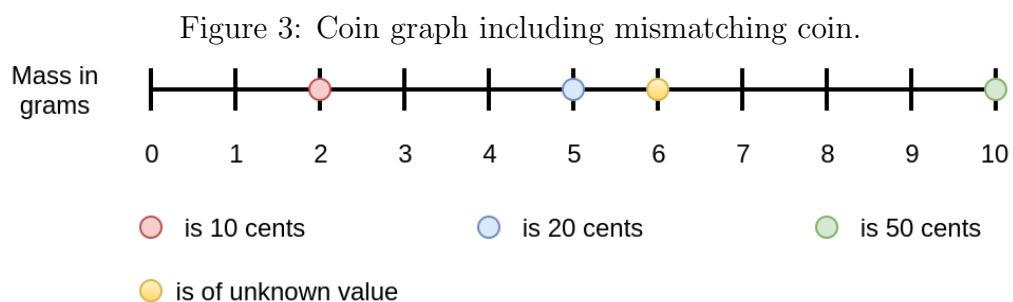
similar scenario on record for guidance.

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.



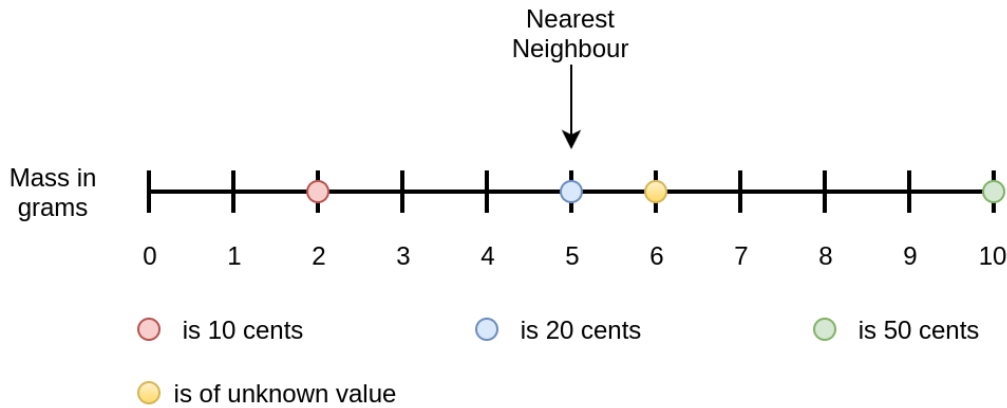
So we have three coins of mass of 2, 5 and 10 grams. Let us say a coin is inserted into the machine, of mass of 6 grams. Here it is on our graph.



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 coin⁴. Here it is again on our graph.

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 simply⁵ looking to the past for similar cases.

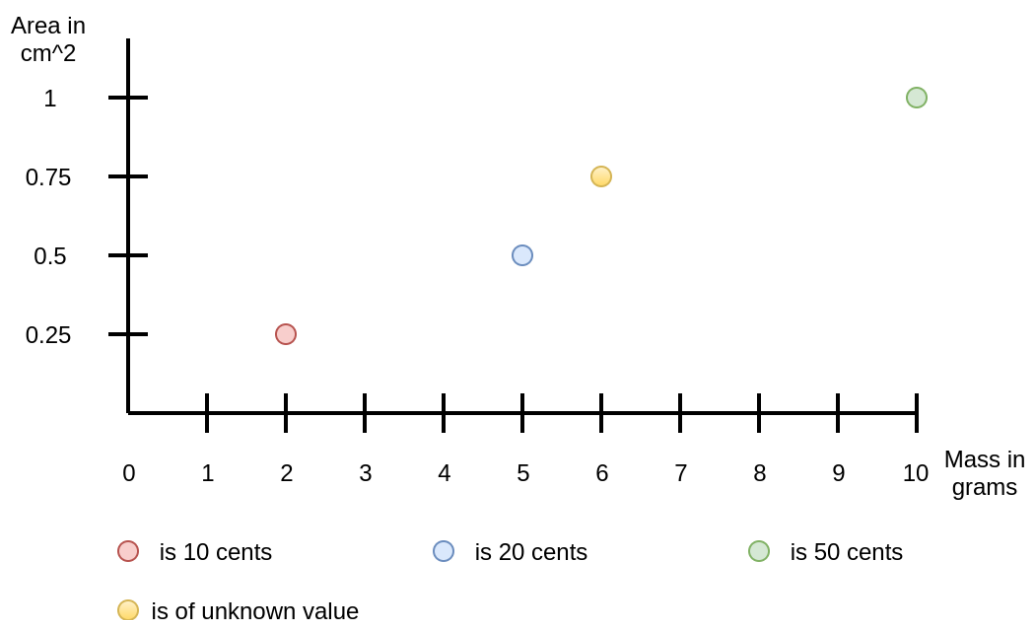
⁴Hence the name of the method, Nearest Neighbours.

⁵Not 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 \mathbf{S} and the newly observed coin be \mathbf{T} .

The brute force approach to finding k number of nearest neighbours is to simply calculate the distance⁶ between \mathbf{T} and every member of \mathbf{S} , and keep the k nearest neighbours.

Unfortunately, this method has a running time of $O(ckn)$ ⁷, 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 \mathbf{S} .

3.2.4 Kd-trees

The idea behind kd-trees is to build a tree structure to segment the set \mathbf{S} such that to find the nearest neighbours, one only has to examine one of those segments⁸. 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.

⁶The function to calculate distance preferably being a non-negative, symmetric and satisfying the triangle inequality function.

⁷or, less a less naive cn

⁸Not quite, more on that later

Here is a new set of exemplars \mathbf{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

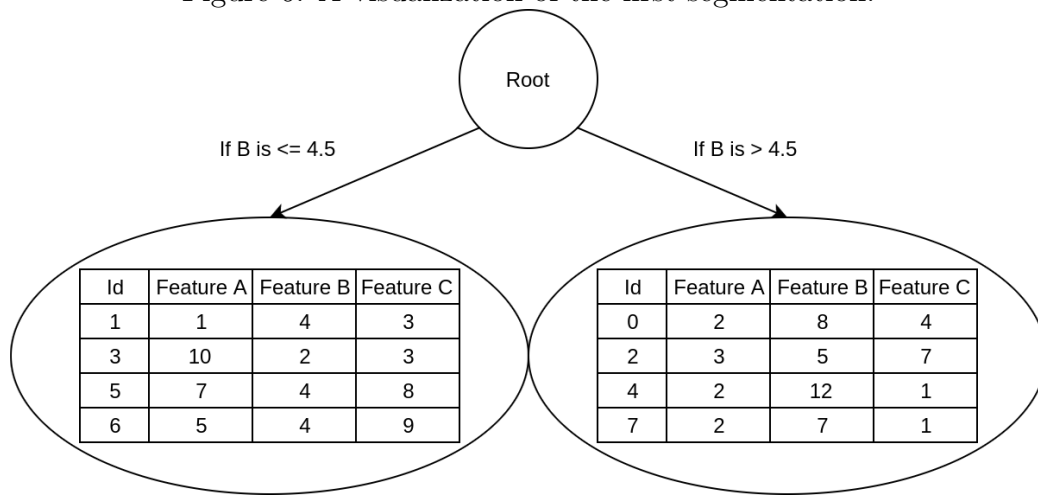
Each row corresponds to a single exemplar, where the exemplars have three features each, namely A, B, and C.

Now we are going to split \mathbf{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.[9]⁹

⁹The reasoning is explained under "The Optimized k-d Tree", page 7.

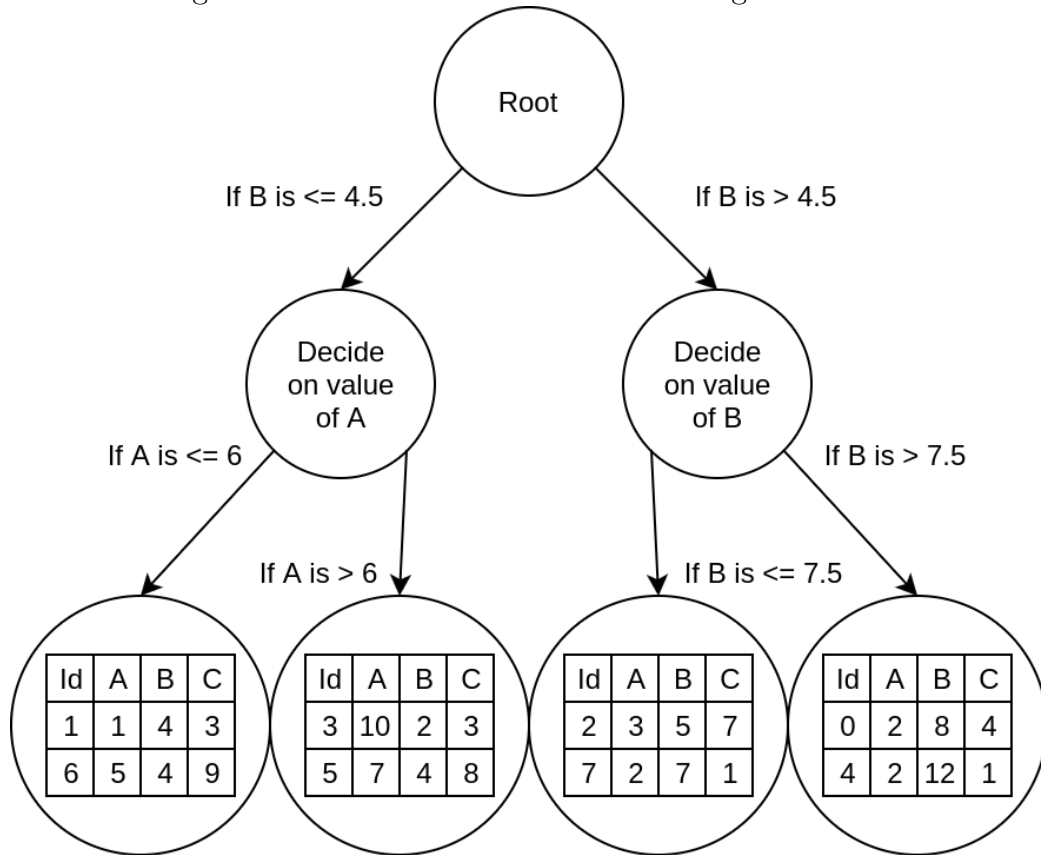
On the first pass through, the feature with the greatest variance is B. Now we choose the median value, which in this case is 4.5 ($(4+5)/2$, even numbers). So now all exemplars which have a value of B which is less than or equal to this median go to the left branch, whilst all greater than 4.5 go to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

Figure 6: A visualization of the first segmentation.



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.



Now the tree is 2 levels deep. The number of levels, and consequently the number of sections the exemplar set \mathbf{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 [9].

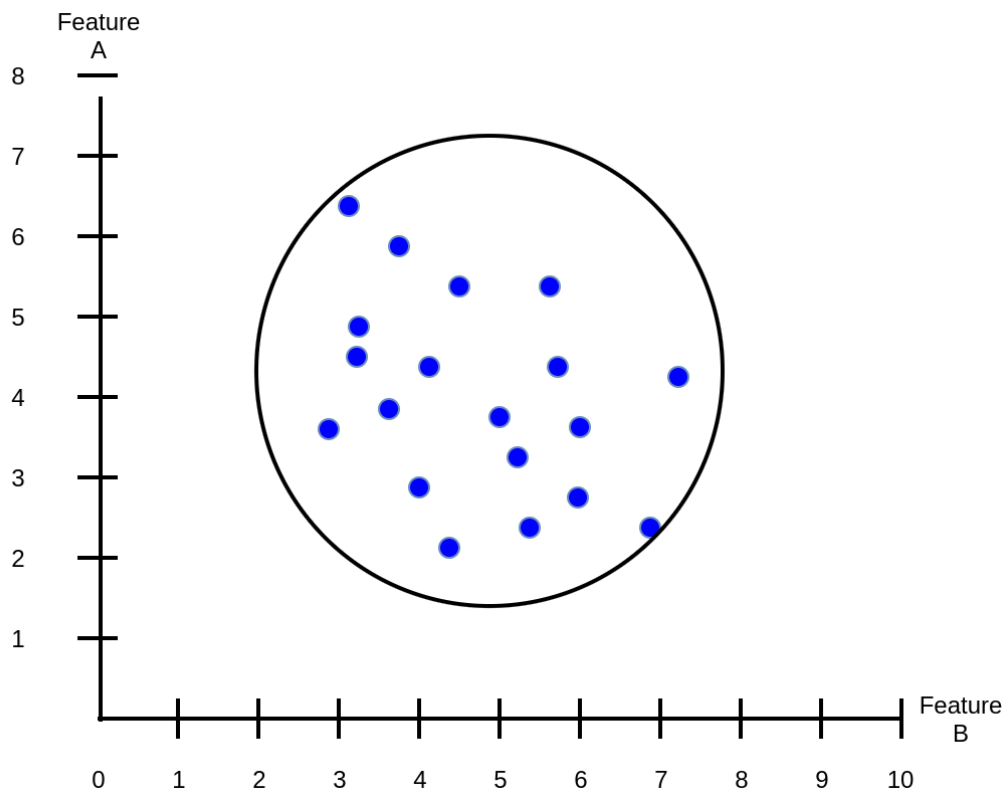
3.2.5 Balltree

There are many versions of the balltree algorithms[10], 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.

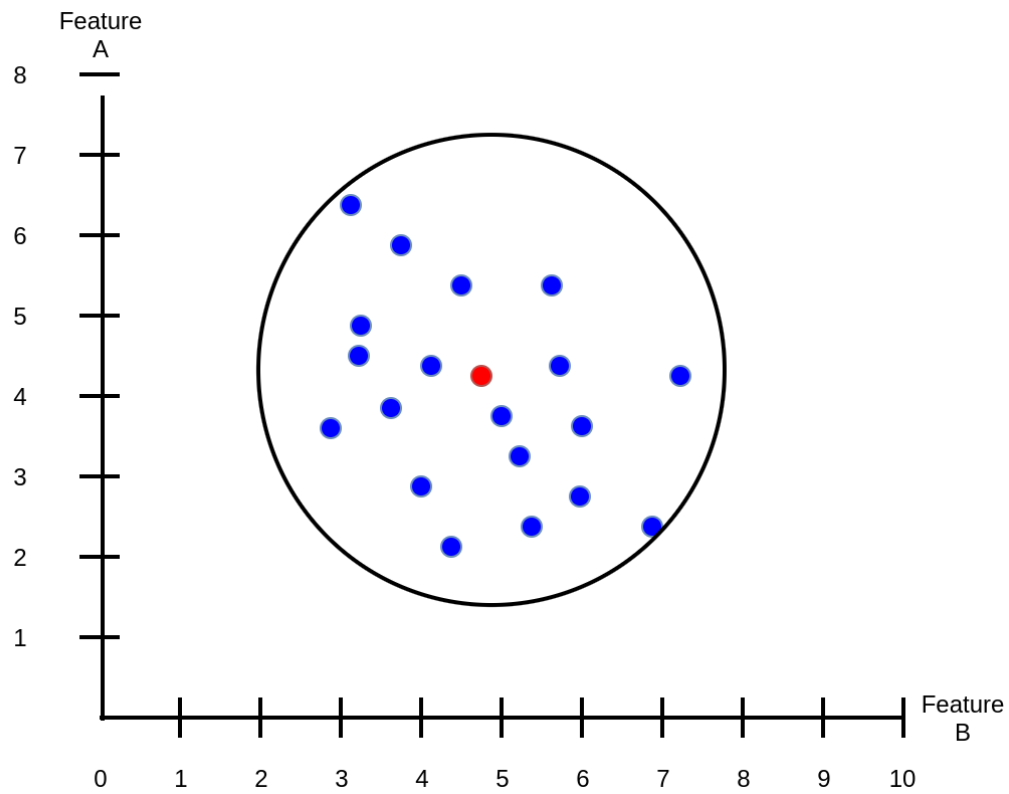
Suppose a balltree is to be built from the following exemplars:

Figure 8: The root of the balltree.



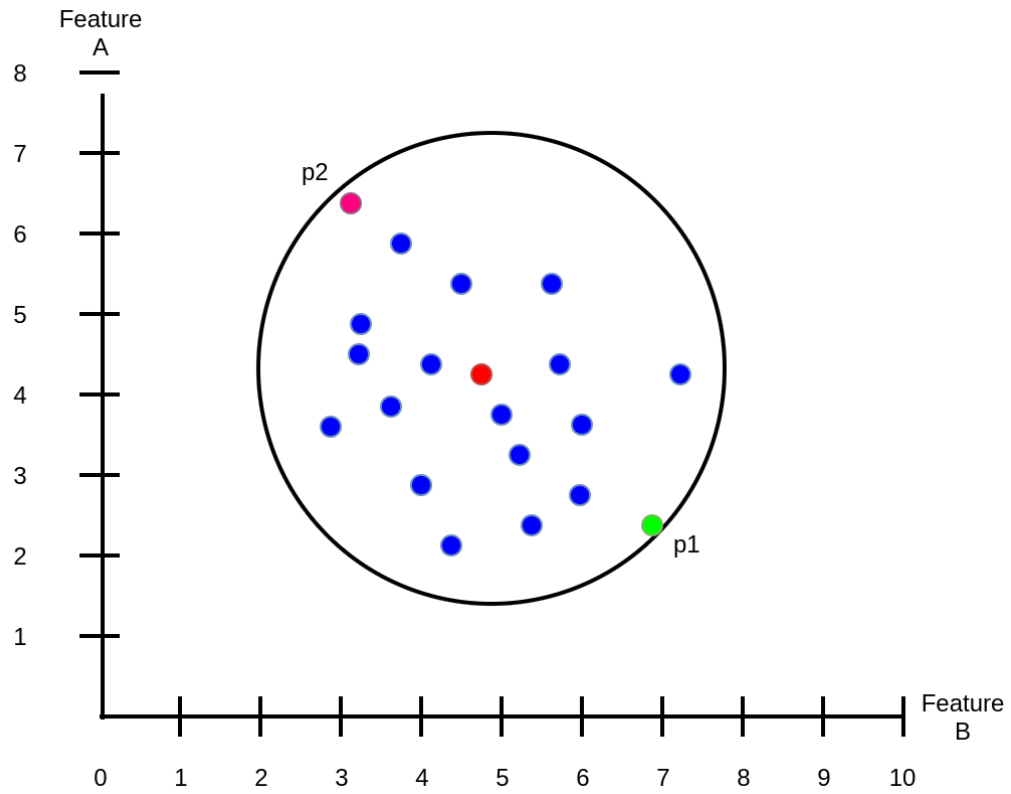
Currently we have one node, the entire exemplar set. To divide it, we calculate the centroid of the current node, which looks like:

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



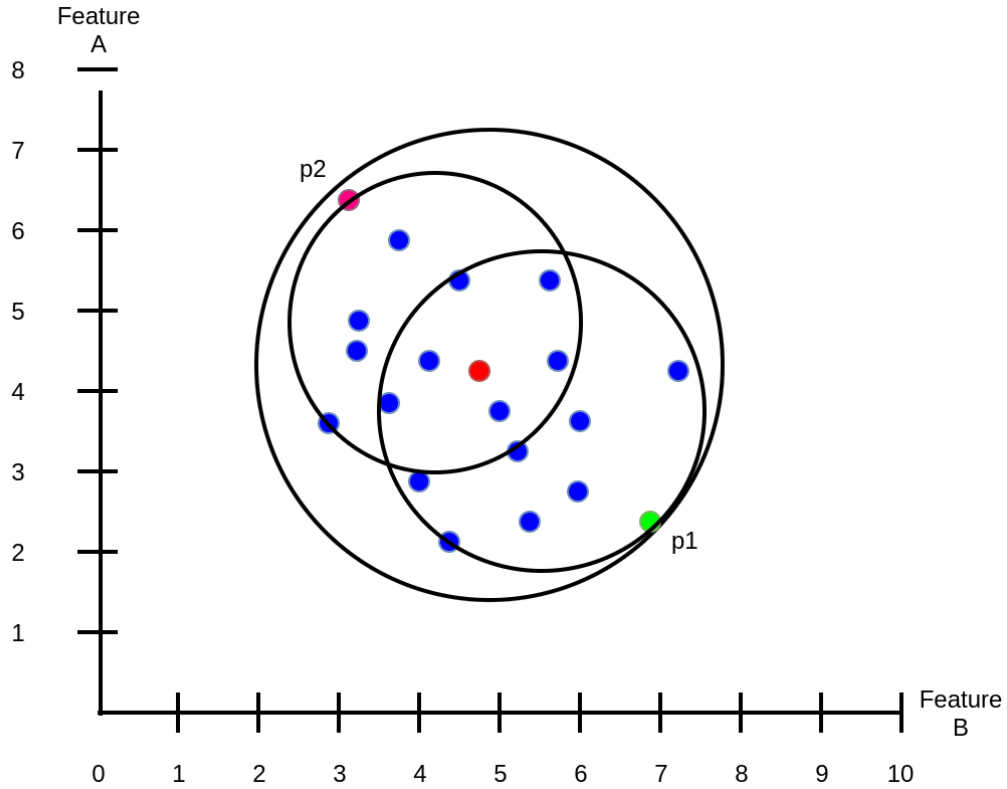
Now we pick the point the furthest away from the centroid, call this $p1$.
Then pick a point farthest away from $p1$, and call it $p2$.

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 p_1 and p_2 . Points closer to p_1 become one subset, and points closer to p_2 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[10], 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 guarantees¹⁰,
the derivation for which can be found at [10].

¹⁰which are only possible if a proper metric is used.

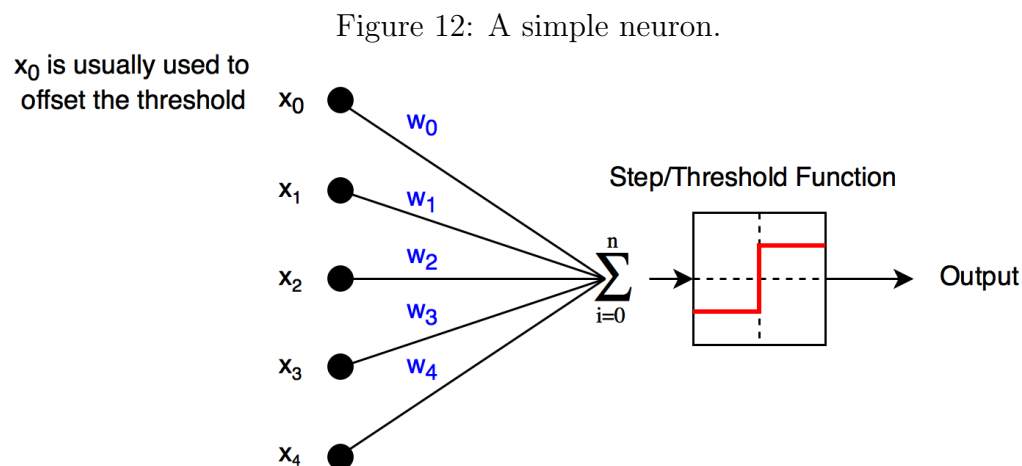
3.3 Neural Network

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:



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 $x_0w_0 + x_1w_1 + \dots + x_4w_4$ in this particular example), and passed through a threshold function. In the end it looks like:

$$\text{output} = \text{threshold}(\sum_{i=0}^n x_i w_i)$$

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 $x_0 w_0$ 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 $x_0 w_0$ 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_1 , x_2 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.

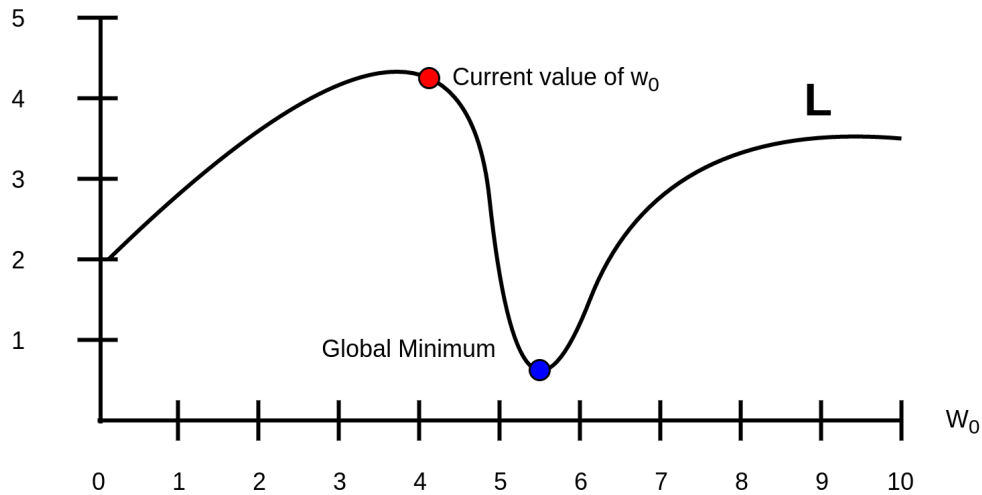
3.3.2 Gradient Descent

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 \mathbf{L} which captures the performance of our neuron, and a set \mathbf{S} of houses and their details, as well as whether or not they sold for more than half a million Euro. What \mathbf{L} captures is how many of our training samples does our neuron classify correctly. The fewer errors our neuron makes, the lower the value of \mathbf{L} .

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.



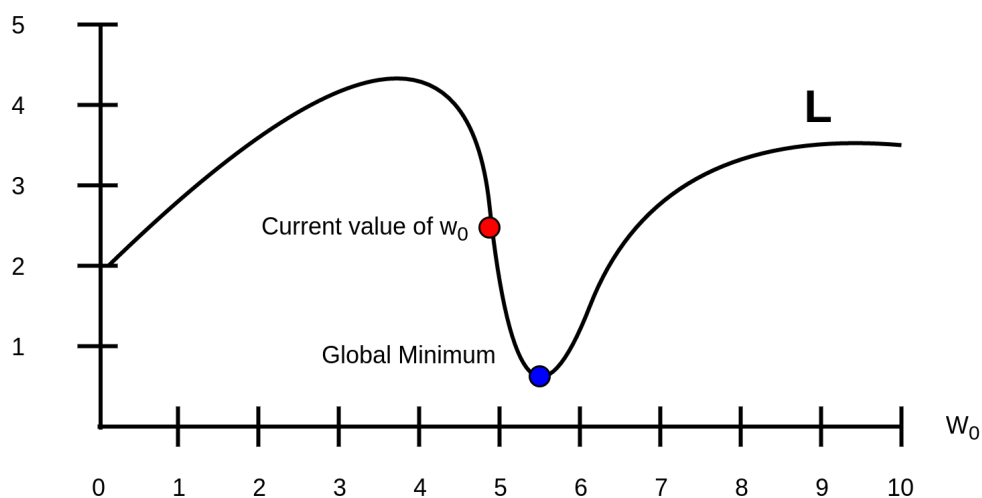
Now, as we adjust the weight (current red point), the magnitude of our loss function will alter. We can traverse that loss function by changing the weight w . If we change the weight w such that the value loss function de-

creases, 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 point¹¹, so a negative slope tells us we should increase the value of w to reduce the value of \mathbf{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 found to be 0 at the global minimum.

This is a greedy algorithm, and one can see that should w_0 have been

¹¹Hence the term gradient descent.

initialized to the value of 2 or so, the gradient descent algorithm would have 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]¹².

Gradient descent is defined as follows:

$$W_i = W_i - \alpha \frac{\partial L(W)}{\partial w_i}$$

To explain what is going on here. The 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.

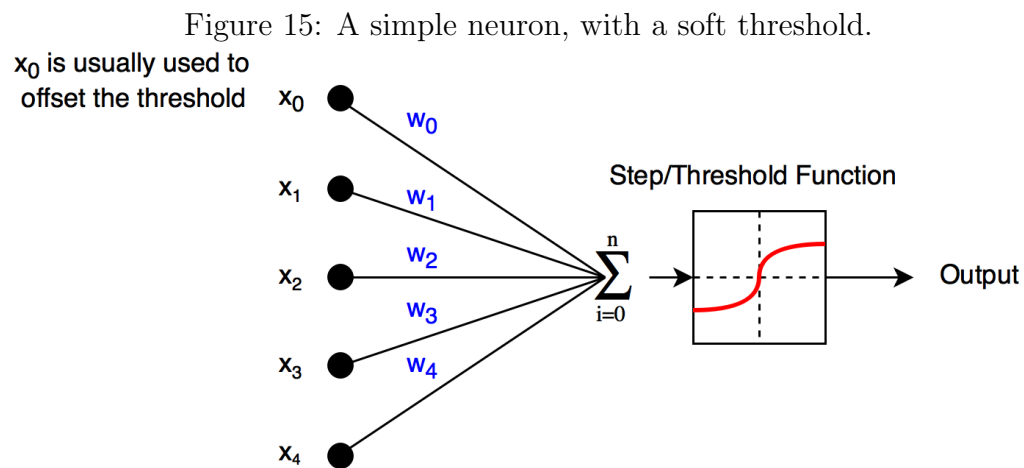
Now that partial derivative, $\frac{\partial L(W)}{\partial w_i}$. As You can see, the derivative is of our function L , which should capture the performance of the neuron. The W is the set of all the weights (in our example neuron, this set is of size 5). The L is the function that we are trying to 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 w_i , since we want to see how does altering w_i

¹²Relevant part starts at 53:15 minute mark.

affect the value of the loss function.

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^{[16]¹³}, which makes our neuron diagram look like:



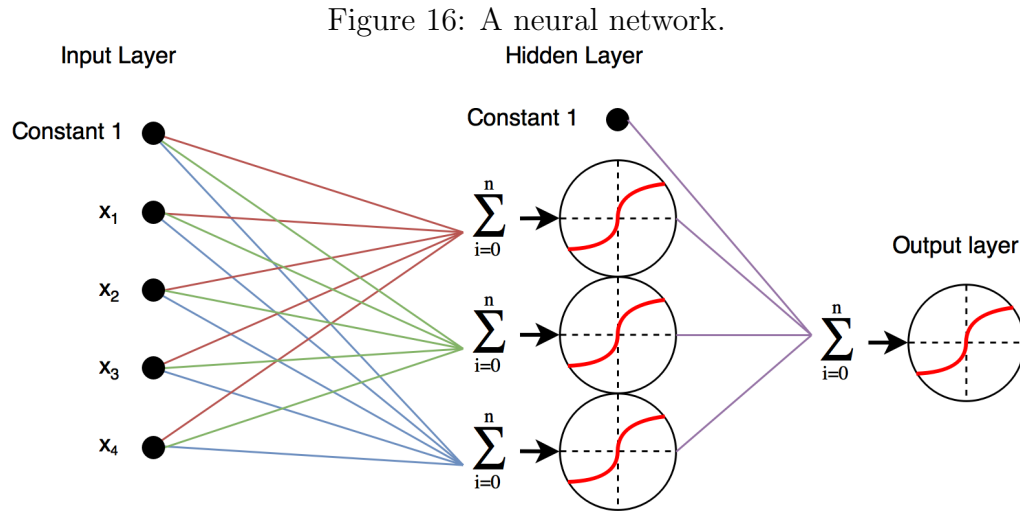
The details of the differentiation can be found at [2]¹⁴, but the intuition should be sufficient for a general understanding.

¹³Explained at 25:00 minute mark.

¹⁴Explained 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:



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[15]).

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 involved¹⁵.

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

¹⁵Explained in much more detail in [16], 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 derivatives with respect to each parameter, backpropagation.

3.4 Support Vector Machine

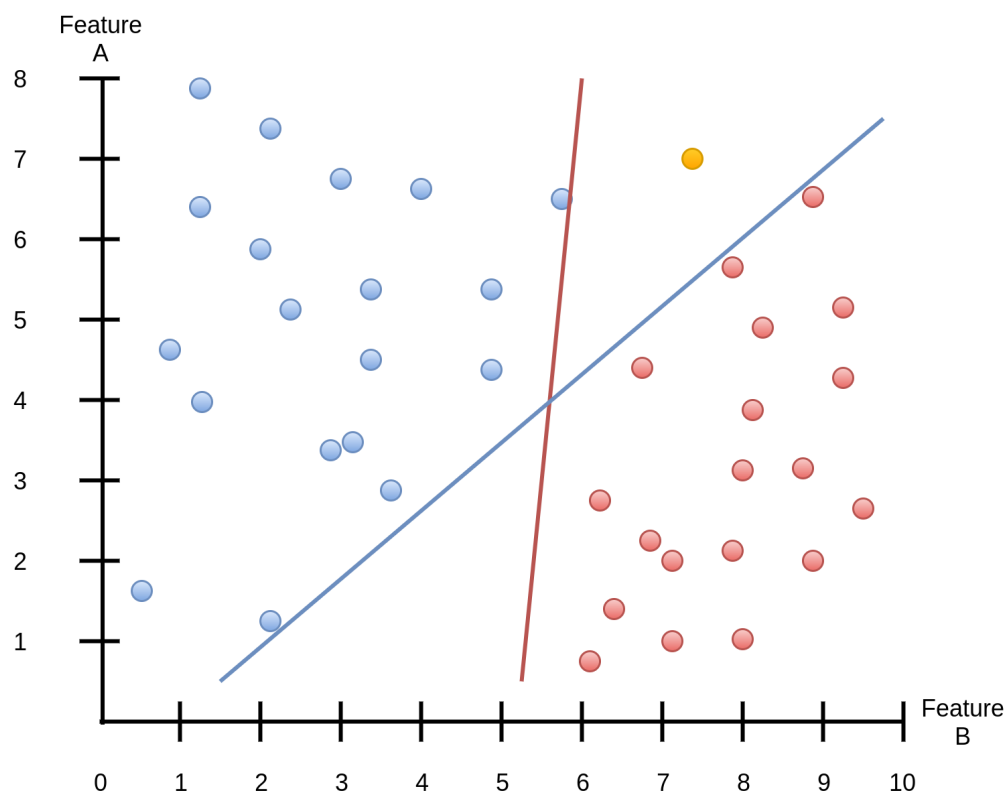
3.4.1 Intuition

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.



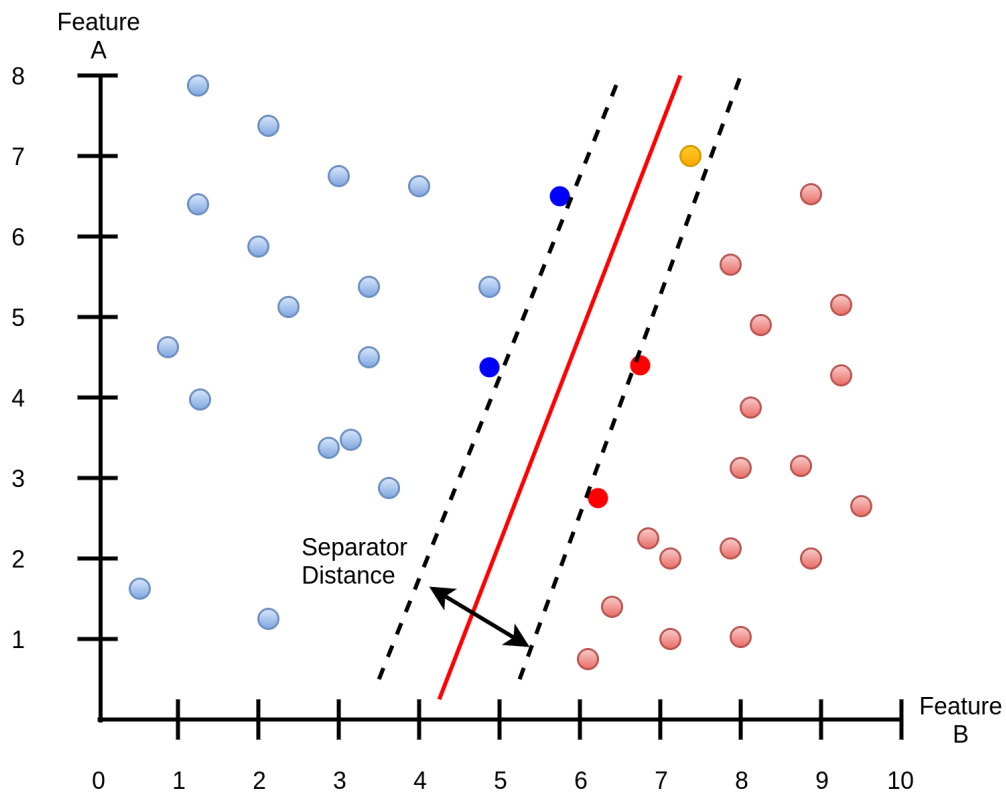
The above is a "linearly separable" set, which simply means that the blue points and the red points can be separated by a straight line¹⁶.

¹⁶Which, 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.¹⁷ The wider we can make this road, the greater the separation between the two clusters, the more effective our linear separator should be on newly observed data.

There are two points in saturated blue, and two points in saturated red. Those are the "support vectors"¹⁸. They define how our linear separator in red was drawn, since they are the ones forming the boundaries of our street.

That's one of the two core ideas: draw the "best" line You can to separate the sample. But what if the sample is not linearly separable, namely, no straight line can separate the red and blue points? That's where kernel functions step in.¹⁹

A from the ground derivation for support vector machines can be found at [13] and [6].

¹⁷This pedagogical device is from [13]

¹⁸Hence the name, Support Vector Machine.

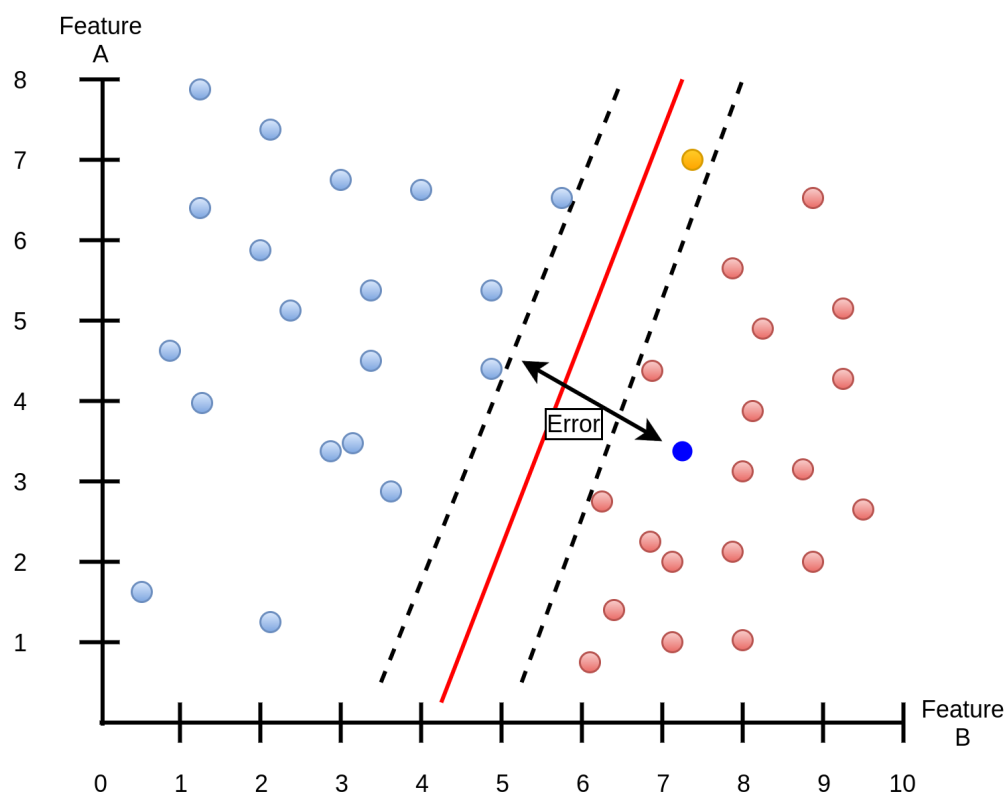
¹⁹Not quite. More on that in the next section.

3.4.3 Soft-Margin SVMs

In the previous graphs, we had a neatly linearly separable training set. Such sets, however, are seldom seen in the real world. Broadly speaking, there are two main cases of non-separability: mild and severe.

Below is an example of *very* mild inseparability:

Figure 19: A *very* mildly inseparable set.



The offending point is in bold blue. Now, it is quite apparent that the bold blue sample is over-all just an outlier, and that even though it might be reasonable to shift our separator a little, completely altering it would be ill-advised.

The solution is as follows: in the original, linearly separable case, we were

trying to optimize for the widest possible division of the sample clusters in question. We should keep that, since that is the core of SVMs, but let's add in an "error" for every point. If the point is on the correct side of the linear separator, as well as outside the "road"²⁰, then it's error is 0. Otherwise, the error is the distance between the correct side of the "road" and the sample point. Take for example the point in bold blue. The error is from the left dotted line to the point, since the bold blue point *should* be able to the left of the "road".

To get the total error, simply sum the error of all points in the training set relative to the "road".

Now, instead of only optimizing for the width of the separation between our clusters, we simultaneously optimize for a trade-off between the greatest width and least error. This, however, introduces another parameter, namely, how important do we consider errors to be. This is a parameter to be optimized on a problem-per-problem basis. This notion is explained in detail in [7].²¹.

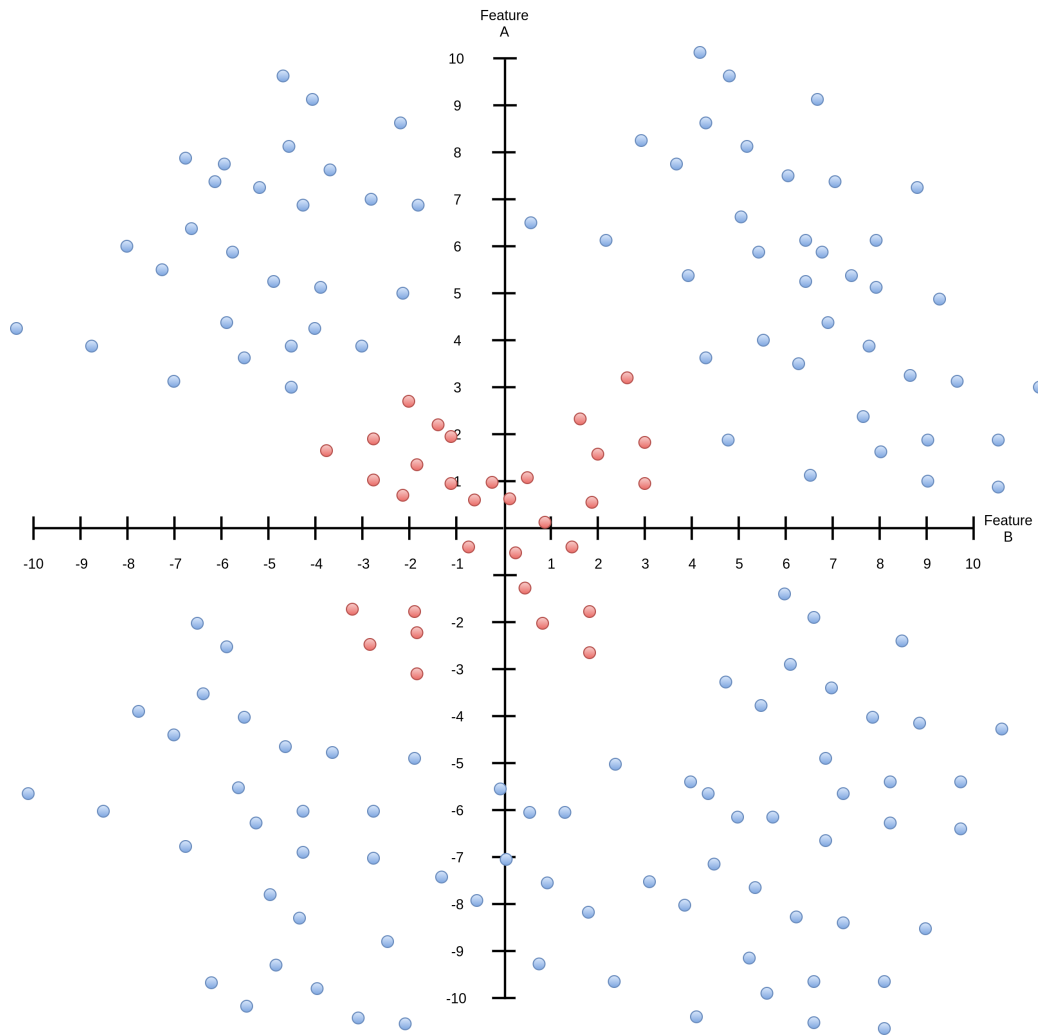
²⁰The road is delineated by the dotted lines, and represents the width of the separation of the clusters.

²¹46:00 minute mark.

3.4.4 Non-Linear Transformations and Kernel Functions

We've dealt with linearly separable and mildly linearly inseparable training sets. What about severely inseparable training sets?

Figure 20: A severely non-linearly separable training set.

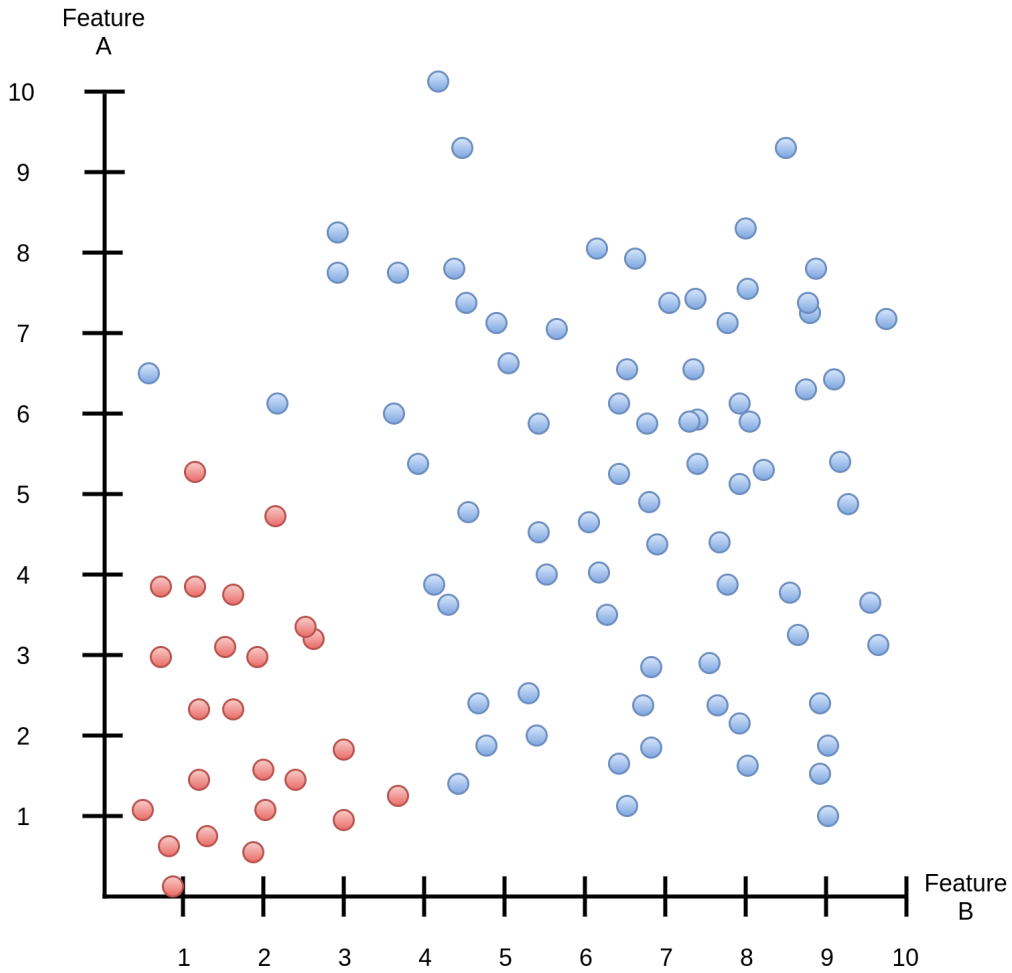


No straight line can separate the above set in a satisfactory fashion, and all is lost for a support vector machine which tries to work in the above space.

A solution is to take the above the above set and apply a transformation

to each point. In this two dimensional case, each point has an x and a y co-ordinate. A possible transformation then is, for each point, to square both of the co-ordinates, namely $(x, y) \mapsto (x^2, y^2)$. This would transform the above set to:

Figure 21: Non-linearly transformed space.



The points from top left, bottom left, and bottom right quadrants have been translated to the top right quadrant as a consequence of the transformation²², whilst their distance from the origin was preserved, resulting in a

²²Squaring values ensures that the result is positive, hence the top right quadrant

mildly inseparable case which soft margin SVMs would be able to handle.

Kernel functions are these non-linear transformations, mathematically adjusted to exploit some naturally occurring mathematical conveniences to make them more powerful in SVM context.²³ The particulars can be found at [7]²⁴.

²³Namely, SVM optimization is dependent solely on the dot product between SVM parameters and the training samples, which allows for some trickery.

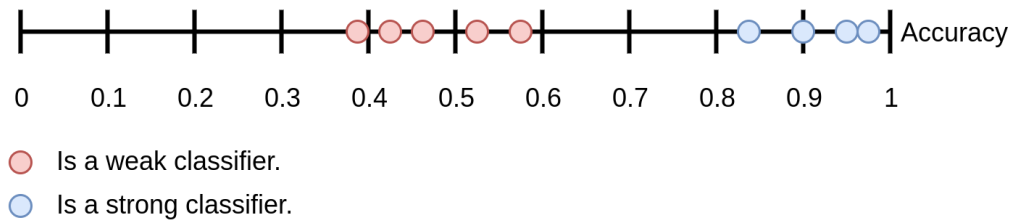
²⁴Beginning at the 06:00 minute mark.

3.5 Boosting

The core idea behind boosting is the combination of a number of *weak classifiers* to form a *strong classifier*.

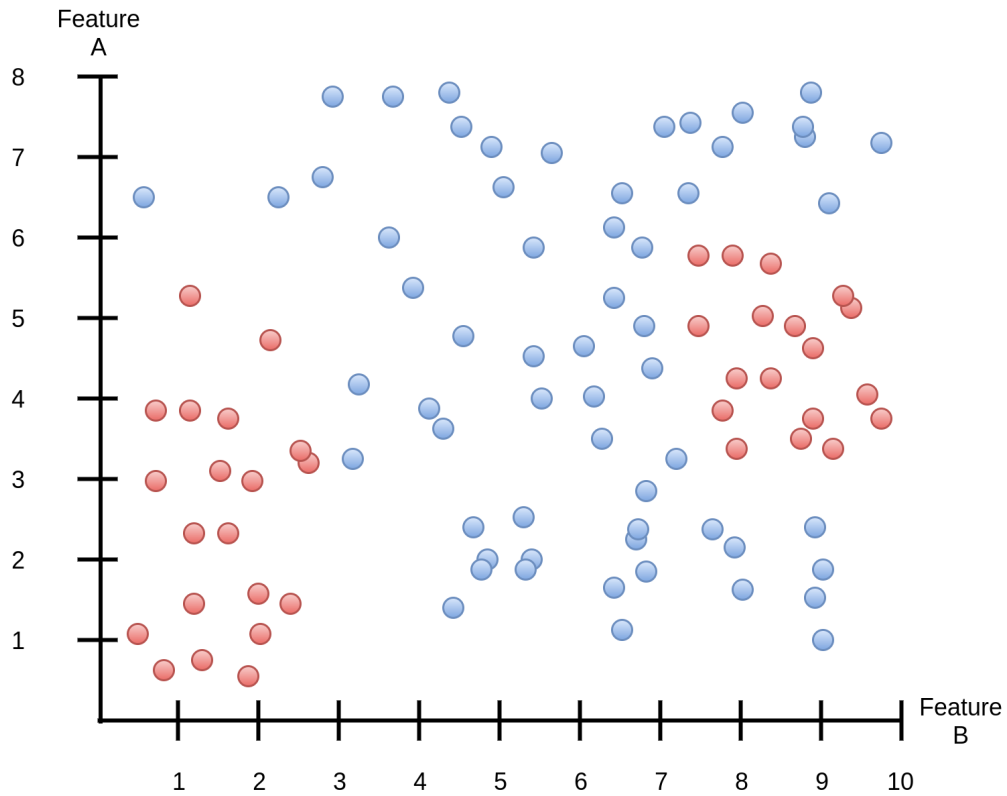
A weak classifier is a classifier which, according to some performance measure, classifies samples correctly about 50% of the time. Similarly, a strong classifier is a classifier which correctly classifies samples almost 100% of the time.

Figure 22: Weak and strong classifiers.



A question arises: Can a series of weak classifiers be combined to form a strong classifier? Let the notion be illustrated via an example. Picture the following training set:

Figure 23: Training set.



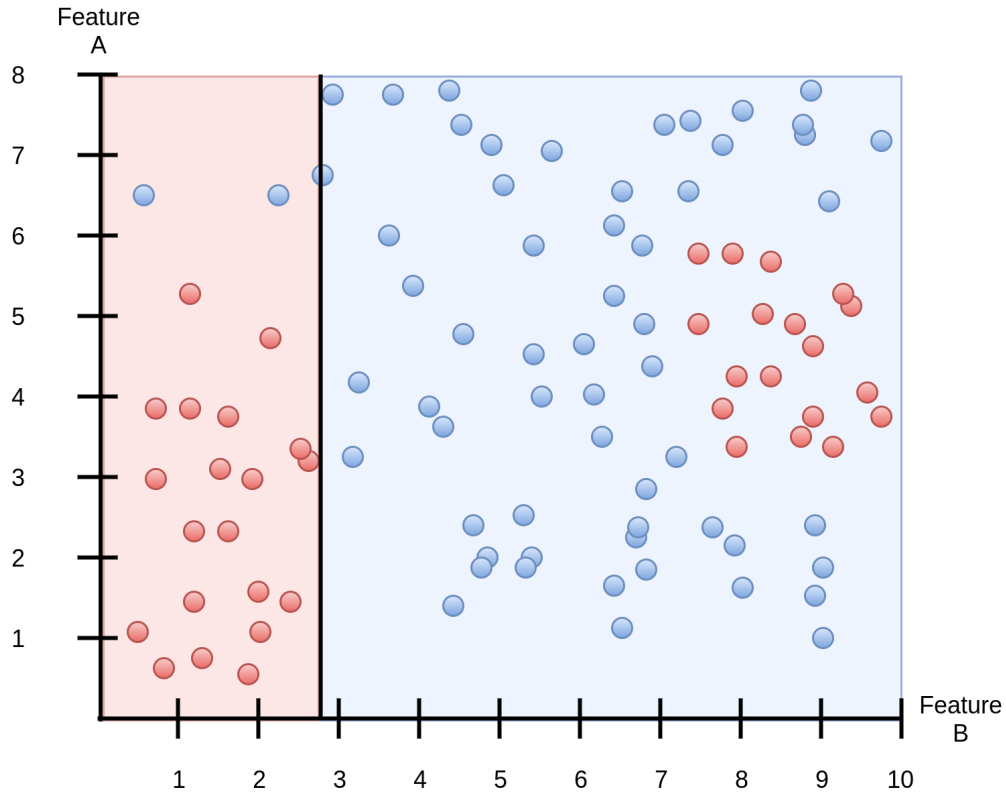
Let us associate a weight w_i with each of the samples, with all of the weights being equal at the beginning. From now and throughout every step, the weights will be assigned such that the summation of all of the weights sums to 1, namely: $\sum_i w_i = 1$ ²⁵

In the search for a weak classifier, let us limit ourselves to horizontal and

²⁵This is to keep the weights as representative of a distribution, more details at [14], 20:00 minute mark.

vertical linear separators.²⁶

Figure 24: The first weak classifier.



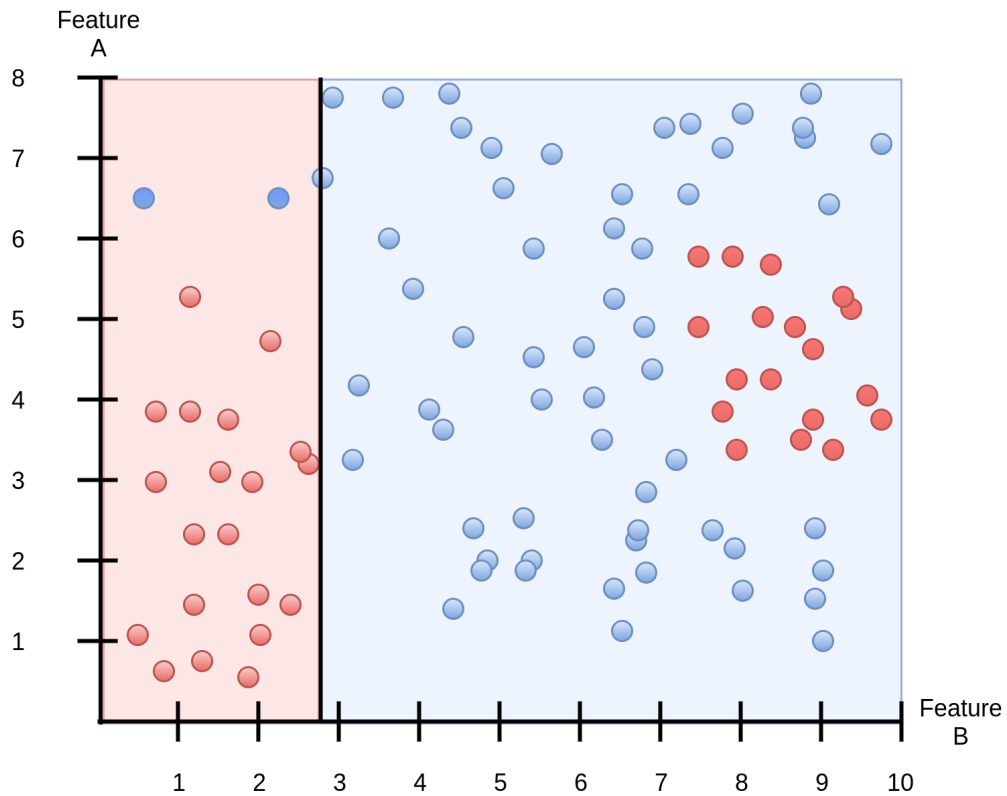
Above, we created a horizontal linear separator at 2.8 mark. The classifier claims that everything to it's left is a red point, and that everything to the right is blue. We are going to assign another weight, not to be confused with the weights we've already assigned to each of the samples (which are all currently equal), to our newly created classifier. The greater the accuracy of our weak classifier, the greater it's weight, since it makes sense to pay more attention to what this classifier has to say.

Whilst most of the points are correctly classified, some of those blue points

²⁶Namely, decision tree stumps. If You'd like to know all there is to know about decision trees, see [12]

to the left, and all of those red points to the right are mis-classified. Now we are going to alter the weights of the samples, and increase the weights of the samples which were misclassified. To reflect this, their colouring is increased in intensity²⁷:

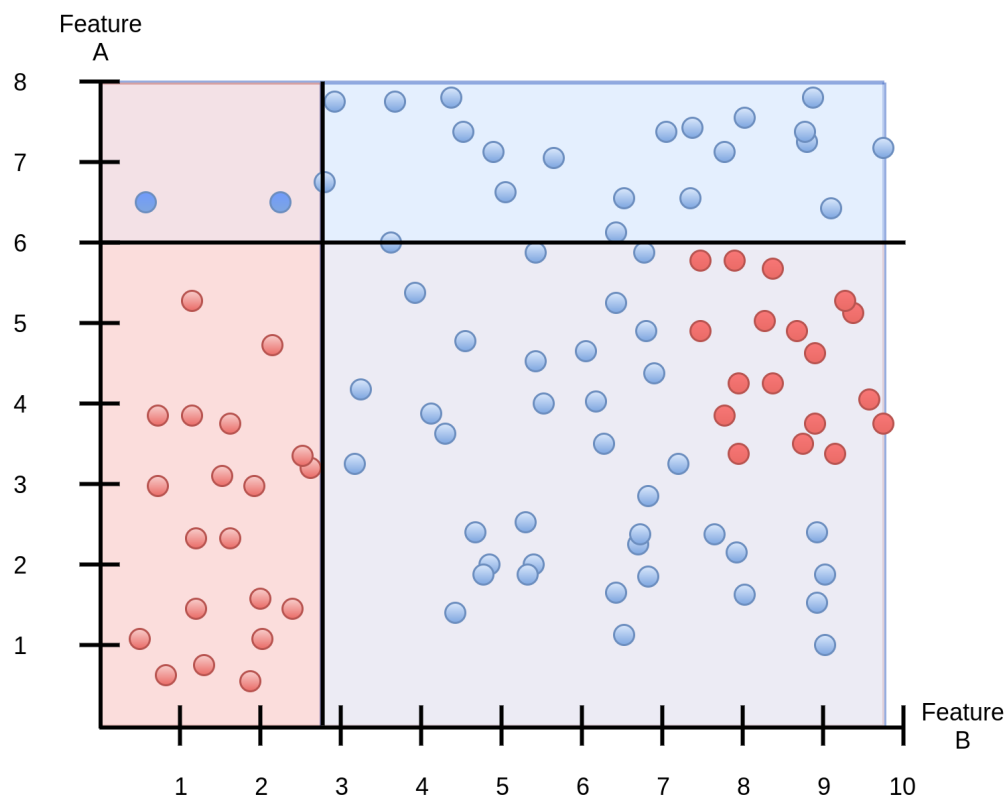
Figure 25: Altered sample weights.



²⁷Since the sum of all weights is to be kept a constant 1, an increase in weight of incorrectly classified points is equivalent to a decrease in the weight of the correctly classified ones.

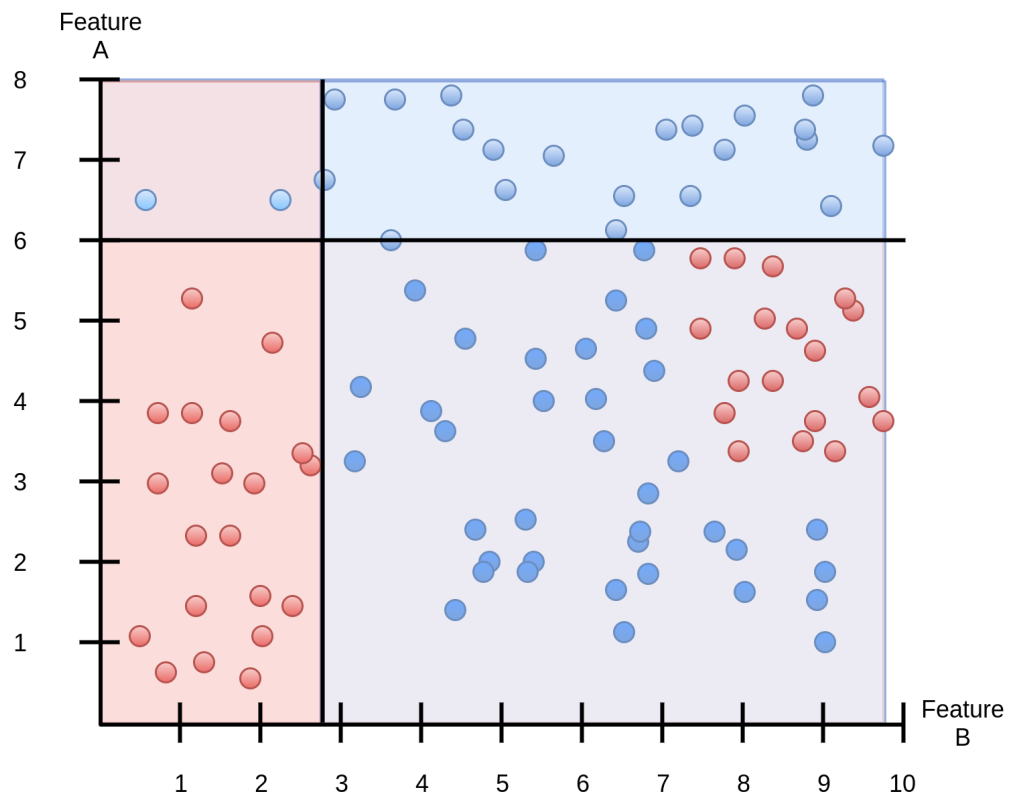
Now we introduce another weak classifier, for instance:

Figure 26: The second weak classifier.



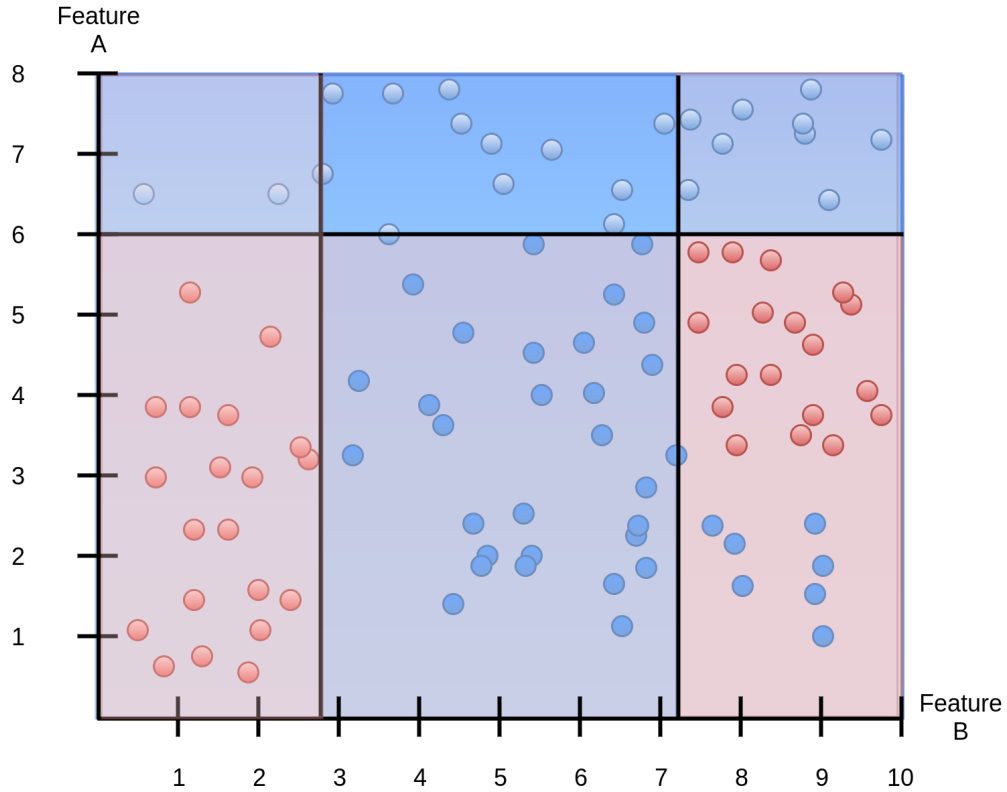
The new weak classifier suggests that everything over 6 on the y-axis should be classified blue, and everything less than 6 should be red. Again, we assign a weight to the second classifier in accordance to how well it has performed. The importance is now shifted the freshly misclassified blue points as follows:

Figure 27: Again altered weights.



Now for the third weak classifier:

Figure 28: The third weak classifier.



The picture is starting to form. Just with these three weak classifiers, the in-sample error is beginning to swiftly diminish. All the blue points above 6 on the y-axis are correctly classified as blue. The red clusters on either side, as well as the center blue cluster have also found their way to correct classification. In fact, only misclassified points remaining are the blue samples in the lower right. With additional weak classifiers, they too will be captured, as their weight will now be increased.

Once the algorithm has completed training by either achieving a certain in-sample accuracy or exhausting the number of weak classifiers allowed, a

new observation is predicted to be either blue or red depending on whether the area it maps onto is blue or red²⁸.

²⁸Which corresponds to checking each of the weak classifiers for their answer, adding up their weighted answers and taking the sign.

4 Methodology

This section walks through the process by which this project was completed, detailing the work done and the challenges encountered along the way.

4.1 Research

As of the writing of this text, TCD has no machine learning options within it's four year curriculum. Consequently, other sources had been sought.

In preparation for the project, I read "Artificial Intelligence: A Modern Approach"[11]. Although it is a great introduction to the field, it proved mostly irrelevant to this project²⁹.

It was found that the Massachusetts Institute of Technology make some of their courses available to the public, in particular, their course on artificial intelligence, 6.043[8]. However, even though over-all superb, I found that both neural networks and kernel functions needed more attention and explanation. The Caltech course

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>.

²⁹Gradient descent and a tiny section on neural networks aside.

- [3] Yaser Abu-Mostafa. *Lecture 10 - Neural Networks*. 2012. URL: <https://www.youtube.com/watch?v=Ih5Mr93E-2c>.
- [4] Yaser Abu-Mostafa. *Lecture 11 - Overfitting*. 2012. URL: <https://www.youtube.com/watch?v=EQWr3GGCdzw>.
- [5] Yaser Abu-Mostafa. *Lecture 12 - Regularization*. 2012. URL: <https://www.youtube.com/watch?v=I-VfYXzC5ro>.
- [6] Yaser Abu-Mostafa. *Lecture 14 - Support Vector Machine*. 2012. URL: <https://www.youtube.com/watch?v=eHsEr1PJWJU>.
- [7] Yaser Abu-Mostafa. *Lecture 15 - Kernel Methods*. 2012. URL: <https://www.youtube.com/watch?v=XUj5JbQih1U>.
- [8] *Artificial Intelligence - MIT 6.043*. Accessed: 2016/4/30. URL: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/>.
- [9] 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>.
- [10] 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.
- [11] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2nd ed. Pearson Education, 2003. ISBN: 0137903952.

- [12] Patrick Wilson. *11. Learning: Identification Trees, Disorder*. 2014. URL: https://www.youtube.com/watch?v=SXBG3RGr_Rc.
- [13] Patrick Wilson. *16. Learning: Support Vector Machines*. 2014. URL: https://www.youtube.com/watch?v=_PwhiWxHK8o.
- [14] Patrick Wilson. *17. Learning: Boosting*. 2014. URL: <https://www.youtube.com/watch?v=UHBmv7qCey4>.
- [15] Patrick Wilson. *Lecture 12b: Deep Neural Nets*. 2016. URL: https://www.youtube.com/watch?v=VrMHA3yX_QI.
- [16] Patrick Wilson. *Lecture 12b: Neural Nets*. 2016. URL: https://www.youtube.com/watch?v=VrMHA3yX_QI.