Decision Trees — Introduction (ID3)

You meet different types of persons throughout your life, after some experience, you get the idea that what kind of person you like, right? I mean after several experiences with many humans, when you meet a new human, most of the time you get the idea if you like them or not. How do you do that? With **‘Experience’**! right? But you don’t keep all years of experience at the top of your brain always, rather than that it feels some simple and quick decision mechanism working inside your brain.

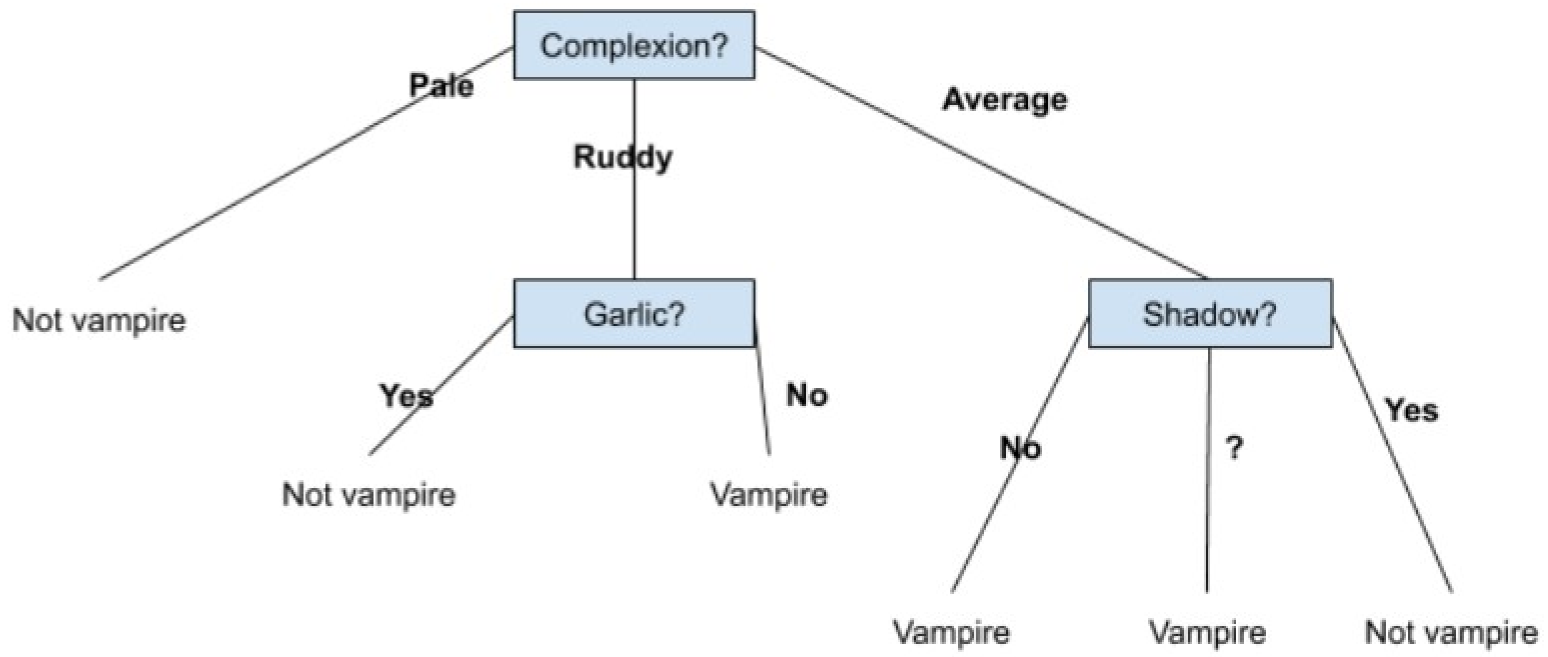
So, rather than going deeper into the biology of the brain, let’s try to build a similar mechanism at a simpler level.

Let's say after your encounter with several people, you don't want vampires to be your friend in future :P

So you made a list of several people you met, their characteristics and if they turned out to be a vampire or not. ("?" in shadow attribute is because you met those people only in dark conditions so you couldn't verify if they cast a shadow or not)

#	Casts shadow?	Eats garlic?	Skin complexion?	accent?	Vampire?
1	? (don't know)	Yes	Pale	None	No
2	Yes	Yes	Ruddy	None	No
3	? (don't know)	No	Ruddy	None	Yes
4	No	No	Average	Heavy	Yes
5	? (don't know)	No	Average	Odd	Yes
6	Yes	No	Pale	Heavy	No
7	Yes	No	Average	Heavy	No
8	? (don't know)	Yes	Ruddy	Odd	No

After observing **this data**, we may come up with a naive model as this tree,



Since with the help of that tree we can make a decision, we call it "Decision Tree". This tree must satisfy all data in the given dataset, and we hope that it will also satisfy future inputs.

But how could we come up with such a tree? The tree given above is made just by some random observation on data…

Following observations…

- All people with pale complexion **are not vampires**.
- All people who have a ruddy complexion and eats garlic **are not vampires** and if they don't eat garlic then they **are a vampire**.
- All people who have an average complexion, and they don't cast a shadow or we don't know if they cast a shadow or not, then they **are a vampire**, or else if they cast a shadow then they **are not a vampire**.

But is that the right way to build a decision tree? Is that tree is the simplest tree we can get from the given dataset?

Such random analysis on a large dataset will not be feasible. We need some systematic approach to attack this problem.

Let’s try to attack this with a greedy approach…

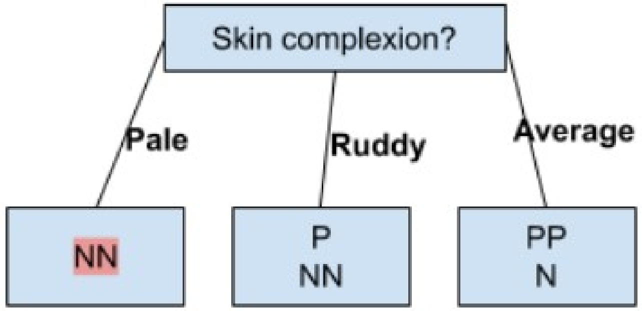
So first, we look at the dataset and decide which attribute should we pick for the root node of the tree…

This is a Boolean classification, so at the end of the decision tree we would have 2 possible results (either they are a vampire or not), so each example input will classify as true (a positive example) and false (a negative example).

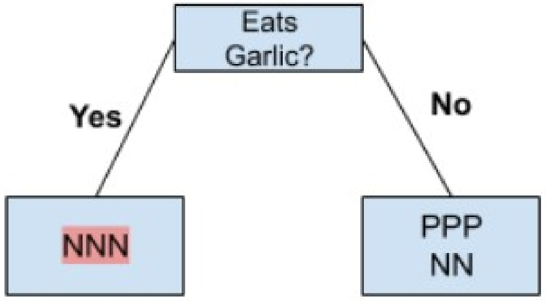
Here ‘**P**’ refers to positive, which means a person **is a vampire,** and ’N’ refers to negative, which means the person **is not a vampire.**

We want attribute which divides more data into homogenous sets, which means in such sets where only P or only N exists because if we have that, we can definitely answer about a vampire or not, thus those will be leaf nodes of the tree.

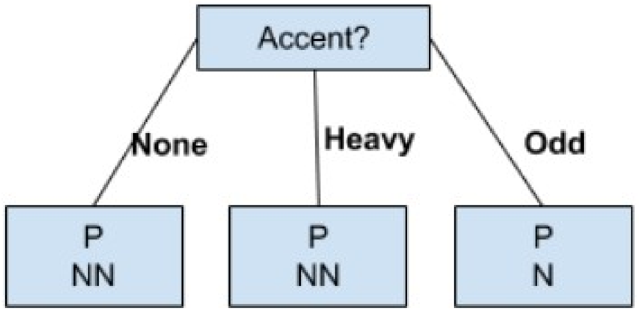
A **greedy algorithm** is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage[1] with the intent of finding a global optimum. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.



Total elements in homogenous sets : 2

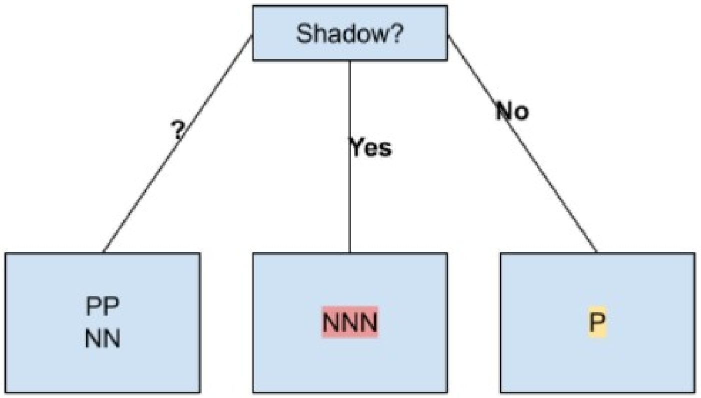


Total elements in homogenous sets : 3



Total elements in homogenous sets : 0

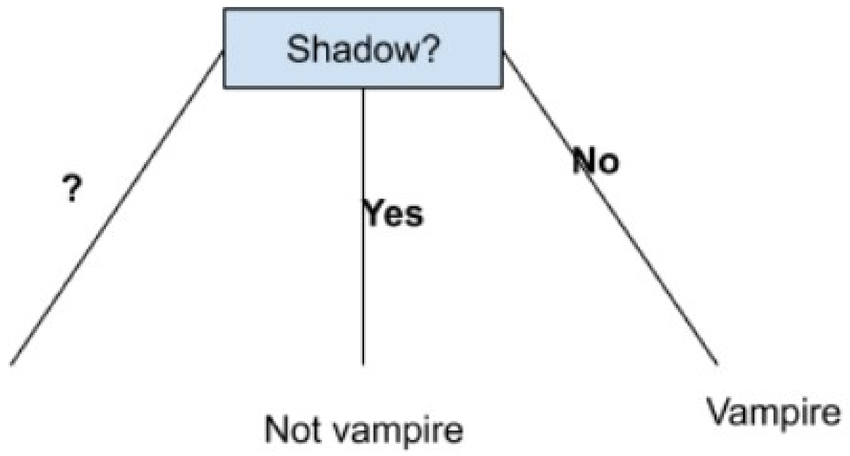
Shadow?	Vampire?
?	No
Yes	No
?	Yes
No	Yes
?	Yes
Yes	No
Yes	No
?	No



Total elements in homogenous sets : 4

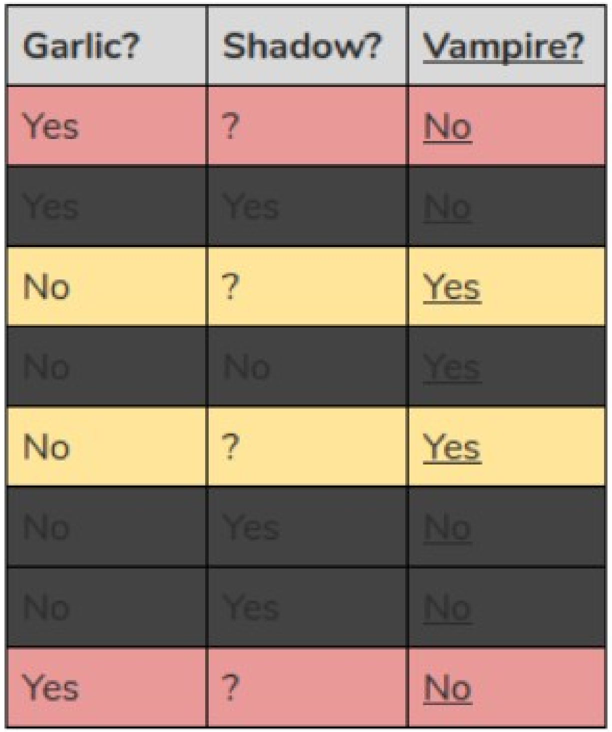
Check for each attribute, and see which one has the highest number of elements in the homogenous set. Here we find that the **'Shadow'** attribute has the highest count for elements in a homogenous set, so we choose this attribute.

So till now, we got this much of the tree…

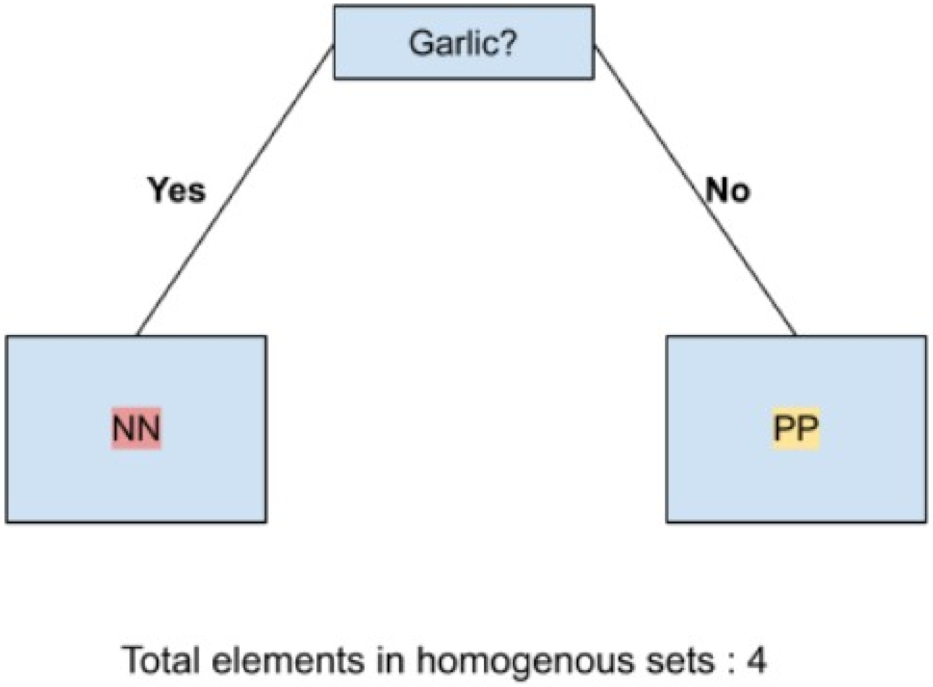


For the shadow attribute "**yes**" and "**no**", we can decide if a person is a vampire or not, but in case of "**?**" we don't know, we need to decide which attribute divides data well when **shadow = '?'**

So, let’s analyze another attribute while the shadow is unknown…

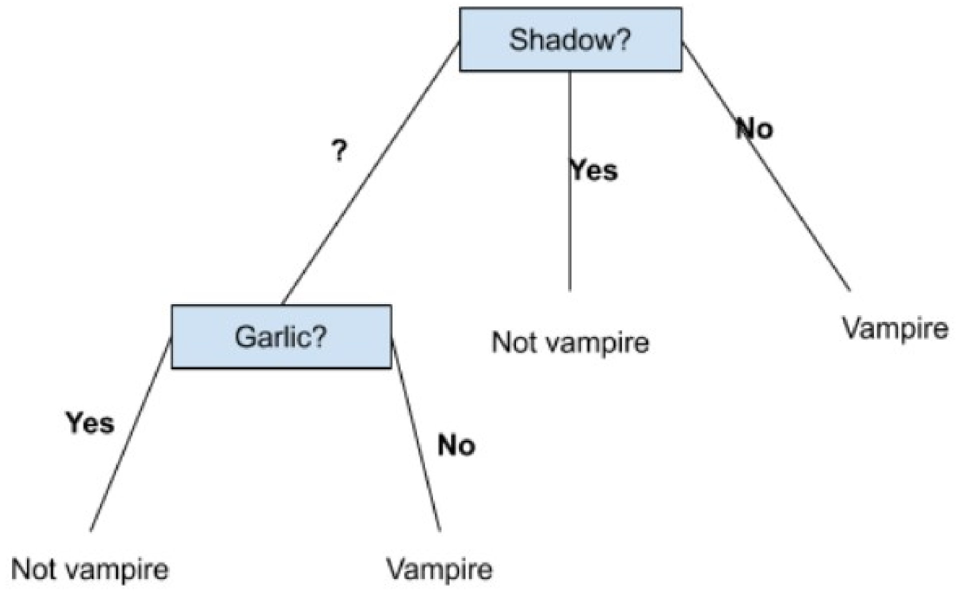


Garlic?	Shadow?	Vampire?
Yes	?	No
Yes	Yes	No
No	?	Yes
No	No	Yes
No	?	Yes
No	Yes	No
No	Yes	No
Yes	?	No



Here we find that “Garlic?” attribute divides maximum elements, in fact, all elements in homogenous sets.

So, our tree now looks like this,



This tree looks simpler than the one we created by picking random attributes, so we observe that the greedy approach is helping us to get better results.

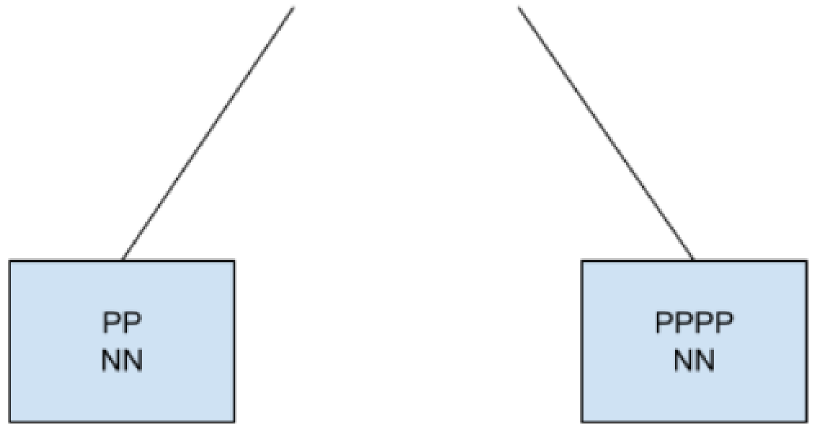
But, is that the right way to do so?

No, because if the dataset is large, we need not end up with attributes dividing into the homogenous set, we may find for all attributes elements in the homogeneous set are 0.

How should we proceed then?

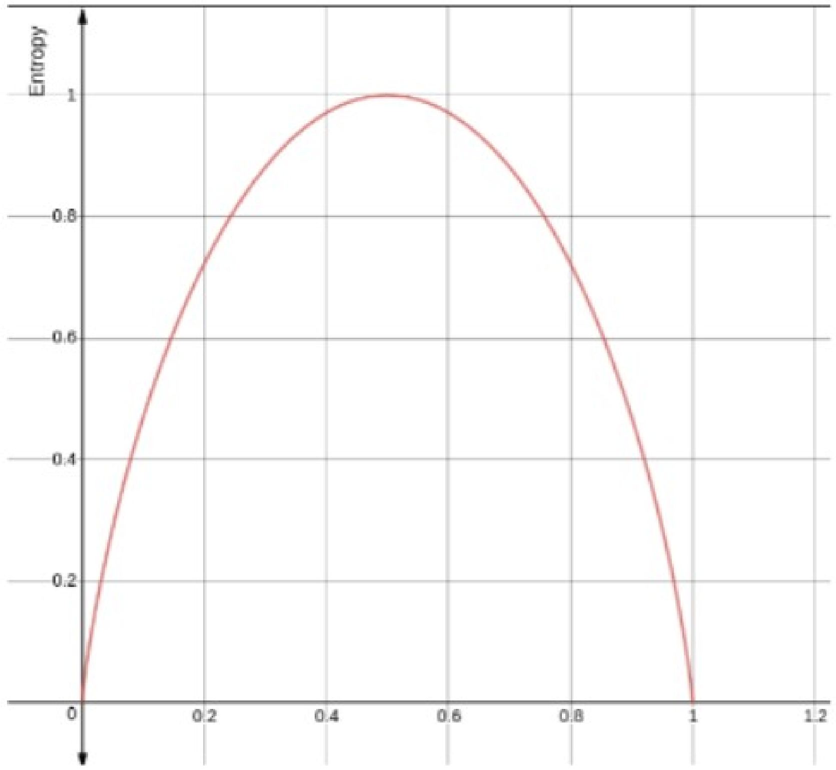
So now let's dive into the **ID3 algorithm** for generating decision trees, which uses the notion of **information gain**, which is defined in terms of **entropy**, the fundamental quantity in information theory.

Imagine these 2 divisions of some an attribute…



We observe that that one on left has the equal number of **P**s and **N**s, so that doesn't give us any hint about the decision, but one on right has more **P**s than **N**s, so it may direct us somewhat towards **P**, so in these 2 we might consider right one.

So, now instead of scoring them 0 right away, let's go with another way. Let's say, one where **P**s and **N**s are equal numbers has the highest entropy (1), and one where there are only **P**s or **N**s has the lowest entropy (0). We can have something like this, **P/(P+N) vs Entropy** graph.



So, when **P=N**, thus **P/(P+N) = 0.5** then **Entropy = 1**,
if **P=k**(some integer) & **N=0** then **Entropy = 0**.

That feels like a pretty much appropriate graph to achieve what we want, so is there some mathematical way to to get this graph…

Luckily for us, this curve can be achieved by the following equation

$$y = -x\, log_2(x) - (1-x) log_2(1-x)$$

Which can be written in **P/(P+N)** and **Entropy** form,
by replacing **x= P/(P+N)** and **y = Entropy**,

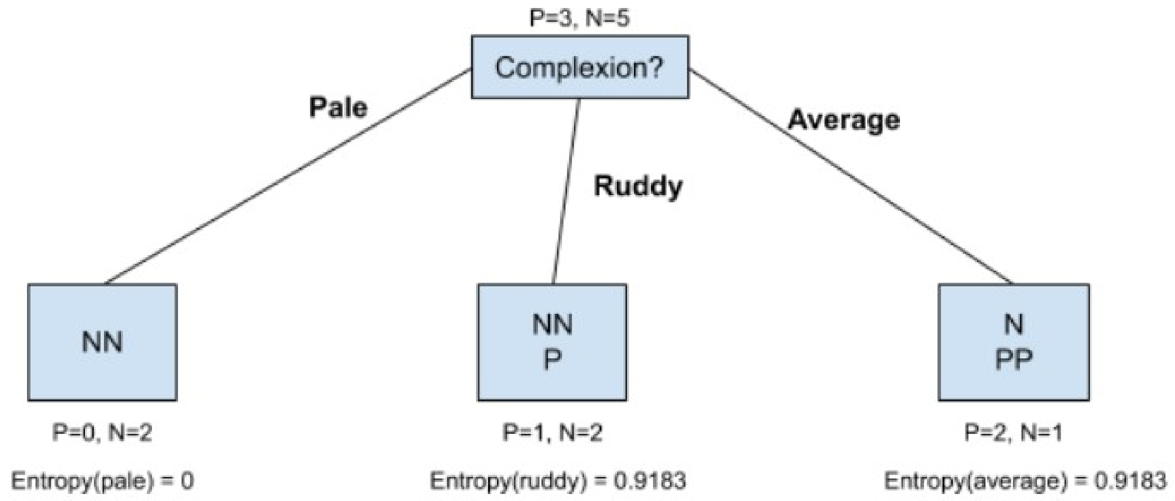
$$\begin{aligned} Entropy &= -\tfrac{P}{P+N}\, log_2(\tfrac{P}{P+N}) - (1-\tfrac{P}{P+N}) log_2(1-\tfrac{P}{P+N}) \\ &= -\tfrac{P}{P+N}\, log_2(\tfrac{P}{P+N}) - (\tfrac{N}{P+N}) log_2(\tfrac{N}{P+N}) \end{aligned}$$

$$Entropy = -\frac{P}{P+N}\, log_2(\frac{P}{P+N}) - (\frac{N}{P+N}) log_2(\frac{N}{P+N})$$

Where P and N is the count of **P**s and **N**s of an attribute for which we are finding the attribute, We want to find information gain from the attribute, which is defined as, (IG — Information gain from some attribute **A** is the expected reduction in entropy)

IG(Attribute) = Entropy of attribute — Weighted average of Entropy of each child set

For example,



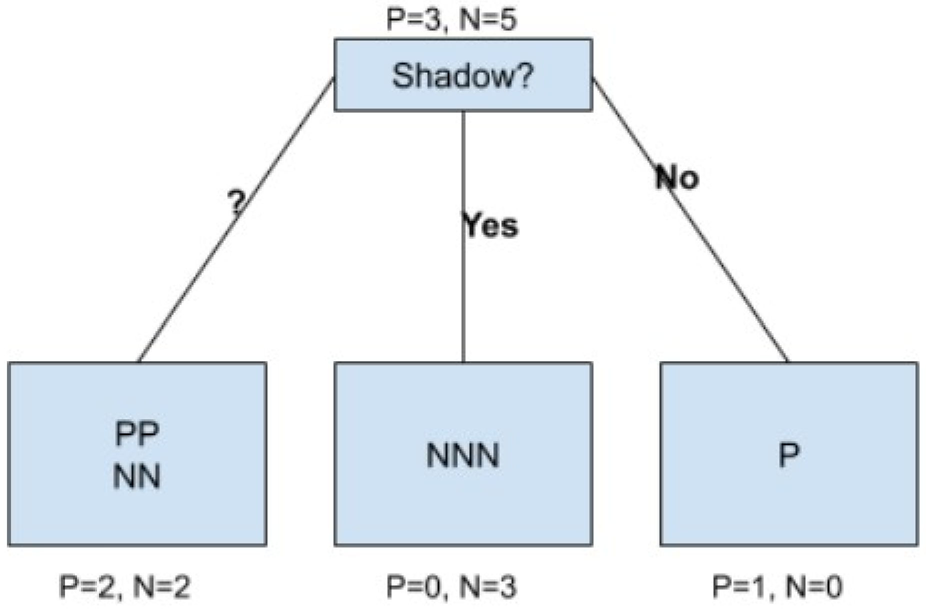
Entropy(Complexion) = 0.9544

Weighted average of child sets = (2*Entropy(pale) + 3*Entropy(ruddy) + 3*Entropy(average)) / 8
= 0.6887

Information Gain (Complexion) = 0.9544 - 0.6887
= 0.2656

Since now you got the idea about **Entropy** and **Information Gain**, let's build our decision tree again from scratch with this new approach!

Shadow?	Vampire?
?	No
Yes	No
?	Yes
No	Yes
?	Yes
Yes	No
Yes	No
?	No

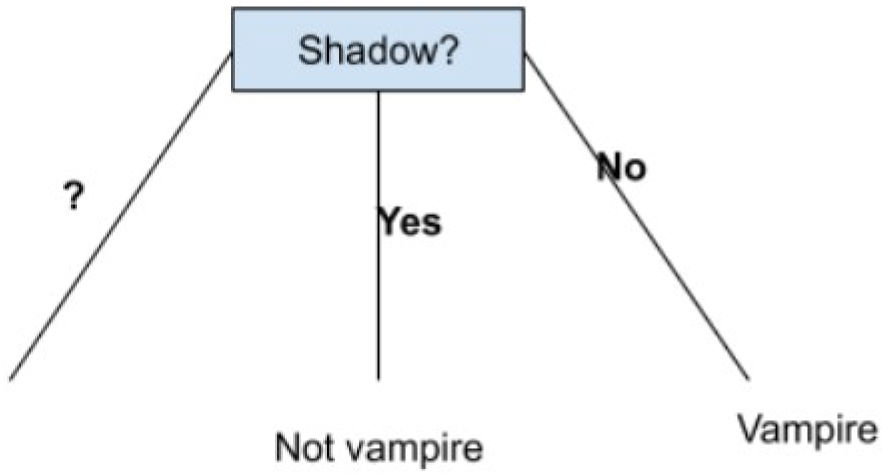


Entropy(Shadow) = 0.9544

Weighted average = 0.5

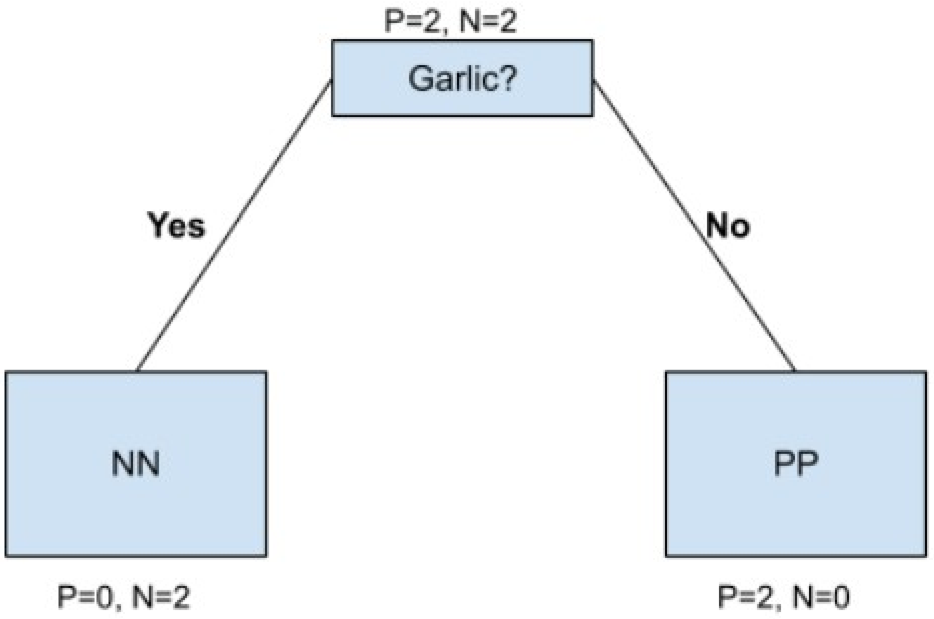
Information Gain = 0.9544 - 0.5 = 0.4544

We observe here that we get maximum Information Gain from shadow attribute, Choosing this as our root node,



We need to decide another attribute for **Shadow = '?'**

Garlic?	Shadow?	Vampire?
Yes	?	No
Yes	Yes	No
No	?	Yes
No	No	Yes
No	?	Yes
No	Yes	No
No	Yes	No
Yes	?	No

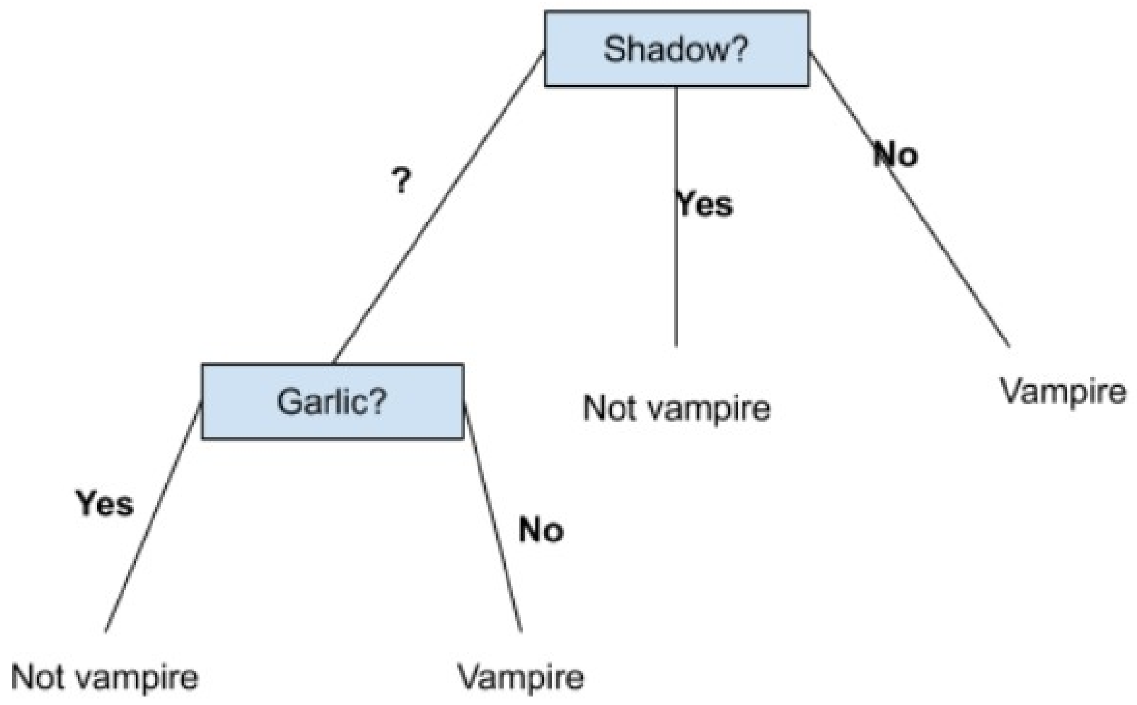


Entropy(Garlic when shadow = “?”) = 1

Weighted average = 0

Information Gain = 1 - 0 = 1

We get maximum Information Gain from Garlic,
So our tree will look like this,



This is exactly the same as the previous approach, because luckily at each step we were able to find some attributes dividing into a homogenous set, but the approach with Information Gain is more robust, which can be applied to make a decision tree from a large dataset.

3.3 Decision Tree Learning

A decision tree is an acyclic graph that can be used to make decisions. In each branching node of the graph, a specific feature j of the feature vector is examined. If the value of the feature is below a specific threshold, then the left branch is followed; otherwise, the right branch is followed. As the leaf node is reached, the decision is made about the class to which the example belongs.

As the title of the section suggests, a decision tree can be learned from data.

3.3.1 Problem Statement

Like previously, we have a collection of labeled examples; labels belong to the set $\{0, 1\}$. We want to build a decision tree that would allow us to predict the class of an example given a feature vector.

3.3.2 Solution

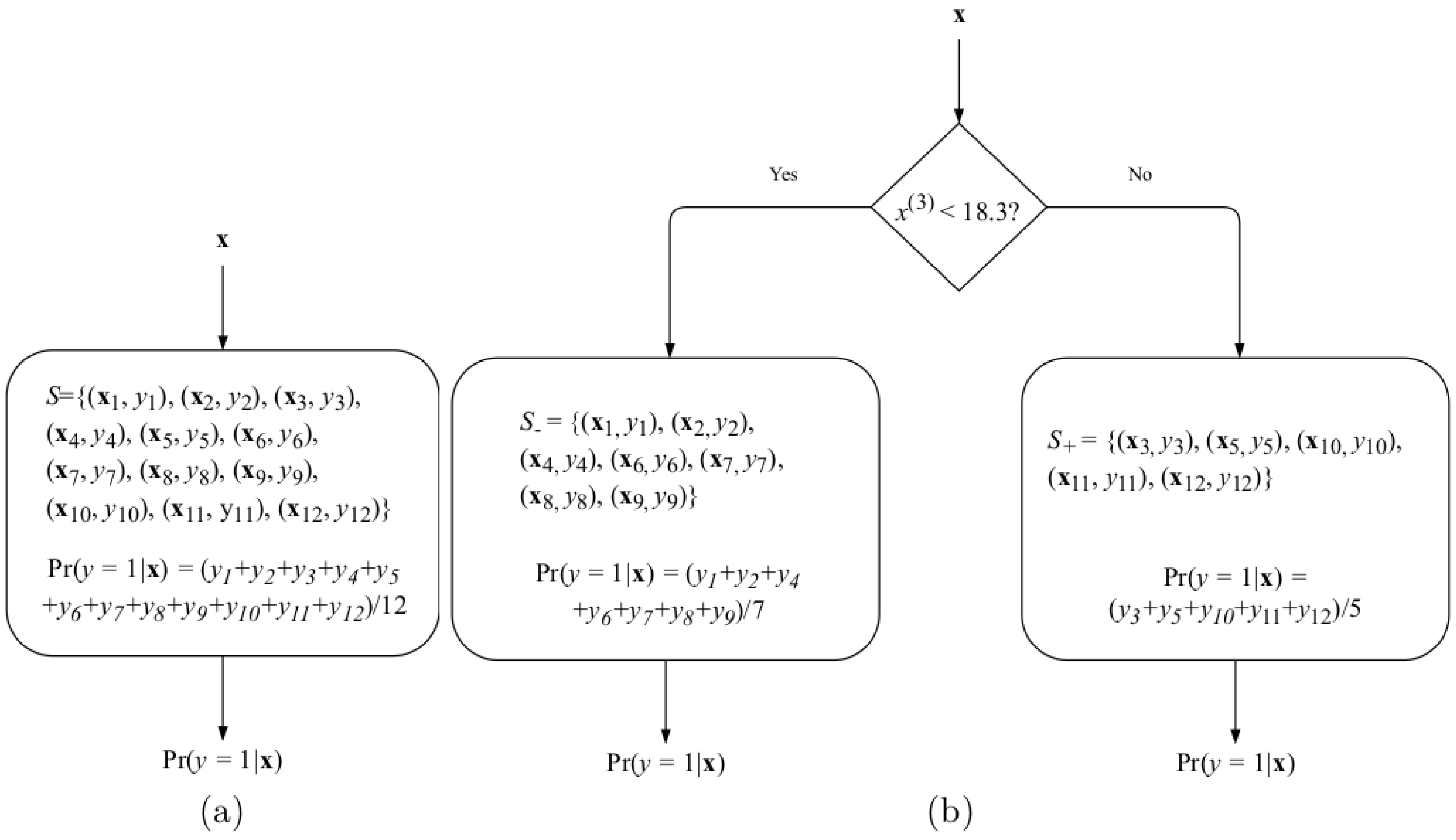
There are various formulations of the decision tree learning algorithm. we consider just one, called **ID3**.

The optimization criterion, in this case, is the average log-likelihood:

$$\frac{1}{N}\sum_{i=1}^{N} y_i \ln f_{ID3}(\mathbf{x}_i) + (1 - y_i) \ln (1 - f_{ID3}(\mathbf{x}_i)), \tag{5}$$

where f_{ID3} is a decision tree.

By now, it looks very similar to logistic regression. However, contrary to the logistic regression learning algorithm which builds a **parametric model** $f_{\mathbf{w}^*,b^*}$ by finding an *optimal solution* to the optimization criterion, the ID3 algorithm optimizes it *approximately* by constructing a **non-parametric model** $f_{ID3}(\mathbf{x}) \stackrel{\text{def}}{=} \Pr(y = 1|\mathbf{x})$.



x
S={(x1, y1), (x2, y2), (x3, y3), (x4, y4), (x5, y5), (x6, y6), (x7, y7), (x8, y8), (x9, y9), (x10, y10), (x11, y11), (x12, y12)}
Pr(y = 1|x) = (y1+y2+y3+y4+y5 +y6+y7+y8+y9+y10+y11+y12)/12
Pr(y = 1|x)
(a)
x
x(3) < 18.3?
Yes
No
S- = {(x1, y1), (x2, y2), (x4, y4), (x6, y6), (x7, y7), (x8, y8), (x9, y9)}
Pr(y = 1|x) = (y1+y2+y4 +y6+y7+y8+y9)/7
Pr(y = 1|x)
S+ = {(x3, y3), (x5, y5), (x10, y10), (x11, y11), (x12, y12)}
Pr(y = 1|x) = (y3+y5+y10+y11+y12)/5
Pr(y = 1|x)
(b)

Figure 4: An illustration of a decision tree building algorithm. The set $\mathcal{S}$ contains 12 labeled examples. (a) In the beginning, the decision tree only contains the start node; it makes the same prediction for any input. (b) The decision tree after the first split; it tests whether feature 3 is less than 18.3 and, depending on the result, the prediction is made in one of the two leaf nodes.

The ID3 learning algorithm works as follows. Let $\mathcal{S}$ denote a set of labeled examples. In the beginning, the decision tree only has a start node that contains all examples: $\mathcal{S} \overset{\text{def}}{=} \{(\mathbf{x}_i, y_i)\}_{i=1}^N$. Start with a constant model f_{ID3}^{S}:

$$f_{ID3}^{\mathcal{S}} = \frac{1}{|\mathcal{S}|} \sum_{(\mathbf{x},y)\in\mathcal{S}} y. \tag{6}$$

The prediction given by the above model, $f_{ID3}^{S}(\mathbf{x})$, would be the same for any input $\mathbf{x}$. The corresponding decision tree is shown in fig 4a.

Then we search through all features $j = 1, \ldots, D$ and all thresholds t, and split the set S into two subsets: $\mathcal{S}_- \overset{\text{def}}{=} \{(\mathbf{x}, y) \,|\, (\mathbf{x}, y) \in \mathcal{S}, x^{(j)} < t\}$ and $\mathcal{S}_+ = \{(\mathbf{x}, y) \,|\, (\mathbf{x}, y) \in S, x^{(j)} \geq t\}$. The two new subsets would go to two new leaf nodes, and we evaluate, for all possible pairs (j, t) how good the split with pieces $\mathcal{S}_-$ and $\mathcal{S}_+$ is. Finally, we pick the best such values (j, t), split $\mathcal{S}$ into $\mathcal{S}_+$ and $\mathcal{S}_-$, form two new leaf nodes, and continue recursively on $\mathcal{S}_+$ and $\mathcal{S}_-$ (or quit if no split produces a model that's sufficiently better than the current one). A decision tree after one split is illustrated in fig 4b.

Now you should wonder what do the words "evaluate how good the split is" mean. In ID3, the goodness of a split is estimated by using the criterion called *entropy*. Entropy is a measure of uncertainty about a random variable. It reaches its maximum when all values of the random variables are equiprobable. Entropy reaches its minimum when the random variable can have only one value. The entropy of a set of examples $\mathcal{S}$ is given by:

$$H(\mathcal{S}) = -f_{ID3}^{\mathcal{S}} \ln f_{ID3}^{\mathcal{S}} - (1 - f_{ID3}^{\mathcal{S}}) \ln(1 - f_{ID3}^{\mathcal{S}}).$$

When we split a set of examples by a certain feature j and a threshold t, the entropy of a split, $H(\mathcal{S}_-, \mathcal{S}_+)$, is simply a weighted sum of two entropies:

$$H(\mathcal{S}_-, \mathcal{S}_+) = \frac{|\mathcal{S}_-|}{|\mathcal{S}|} H(\mathcal{S}_-) + \frac{|\mathcal{S}_+|}{|\mathcal{S}|} H(\mathcal{S}_+). \tag{7}$$

So, in ID3, at each step, at each leaf node, we find a split that minimizes the entropy given by eq. 7 or we stop at this leaf node.

The algorithm stops at a leaf node in any of the below situations:

- All examples in the leaf node are classified correctly by the one-piece model (eq. 6).
- We cannot find an attribute to split upon.
- The split reduces the entropy less than some ϵ (the value for which has to be found experimentally[3]).
- The tree reaches some maximum depth d (also has to be found experimentally).

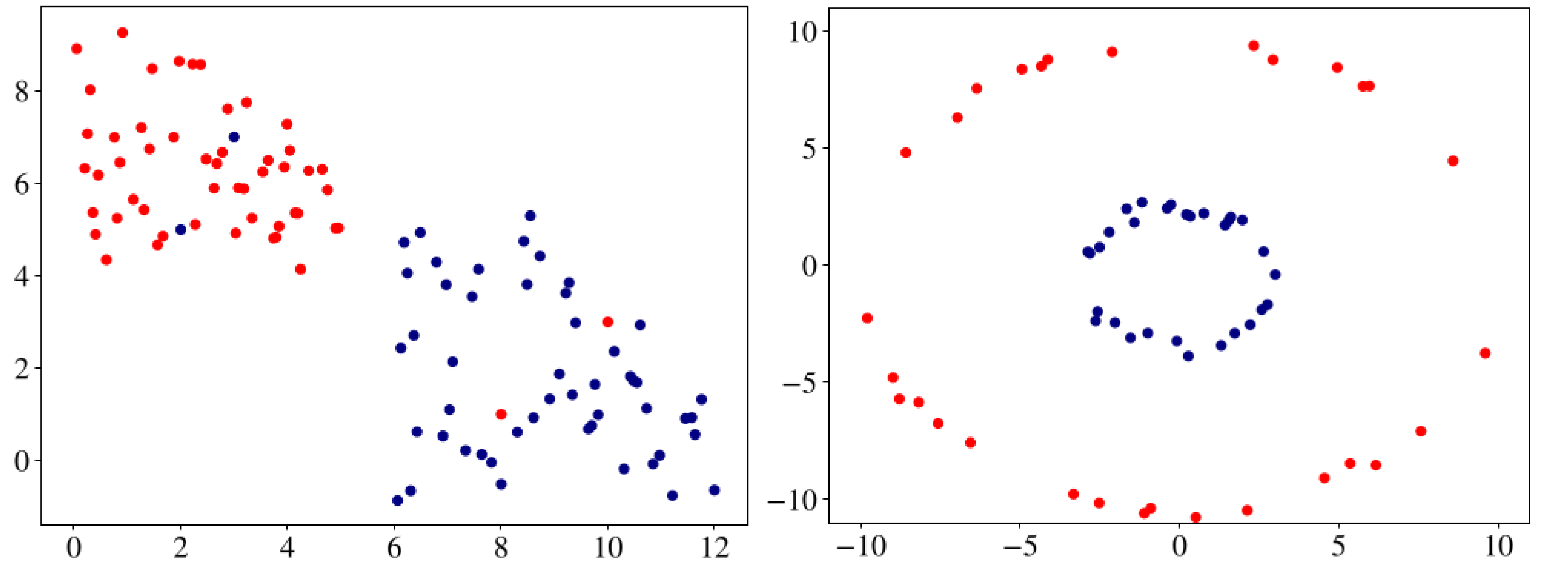
Because in ID3, the decision to split the dataset on each iteration is local (doesn't depend on future splits), the algorithm doesn't guarantee an optimal solution. The model can be improved by using techniques like *backtracking* during the search for the optimal decision tree at the cost of possibly taking longer to build a model.

The entropy-based split criterion intuitively makes sense: entropy reaches its minimum of 0 when all examples in $\mathcal{S}$ have the same label; on the other hand, the entropy is at its maximum of 1 when exactly one-half of examples in $\mathcal{S}$ is labeled with 1, making such a leaf useless for classification. The only remaining question is how this algorithm approximately maximizes the average log-likelihood criterion.

3.4 Support Vector Machine

We already considered SVM in the introduction, so this section only fills a couple of blanks. Two critical questions need to be answered:

1. What if there's noise in the data and no hyperplane can perfectly separate positive examples from negative ones?
2. What if the data cannot be separated using a plane, but could be separated by a higher-order polynomial?



You can see both situations depicted in fig 5. In the left case, the data could be separated by a straight line if not for the noise (outliers or examples with wrong labels). In the right case, the decision boundary is a circle and not a straight line.

Remember that in SVM, we want to satisfy the following constraints:

a) $\mathbf{w}\mathbf{x}_i - b \geq 1$ if $y_i = +1$, and
b) $\mathbf{w}\mathbf{x}_i - b \leq -1$ if $y_i = -1$

We also want to minimize $\|\mathbf{w}\|$ so that the hyperplane was equally distant from the closest examples of each class. Minimizing $\|\mathbf{w}\|$ is equivalent to minimizing $\frac{1}{2}||\mathbf{w}||^2$, and the use of this term makes it possible to perform quadratic programming optimization later on. The optimization problem for SVM, therefore, looks like this:

$$\min \frac{1}{2}||\mathbf{w}||^2, \text{ such that } y_i(\mathbf{x}_i\mathbf{w} - b) - 1 \geq 0, i = 1, \ldots, N. \tag{8}$$

3.4.1 Dealing with Noise

To extend SVM to cases in which the data is not linearly separable, we introduce the **hinge loss** function: $\max(0, 1 - y_i(\mathbf{w}\mathbf{x}_i - b))$.

The hinge loss function is zero if the constraints a) and b) are satisfied, in other words, if $\mathbf{w}\mathbf{x}_i$ lies on the correct side of the decision boundary. For data on the wrong side of the decision boundary, the function's value is proportional to the distance from the decision boundary.

We then wish to minimize the following cost function,

$$C\|\mathbf{w}\|^2 + \frac{1}{N}\sum_{i=1}^{N} \max(0, 1 - y_i(\mathbf{w}\mathbf{x}_i - b)),$$

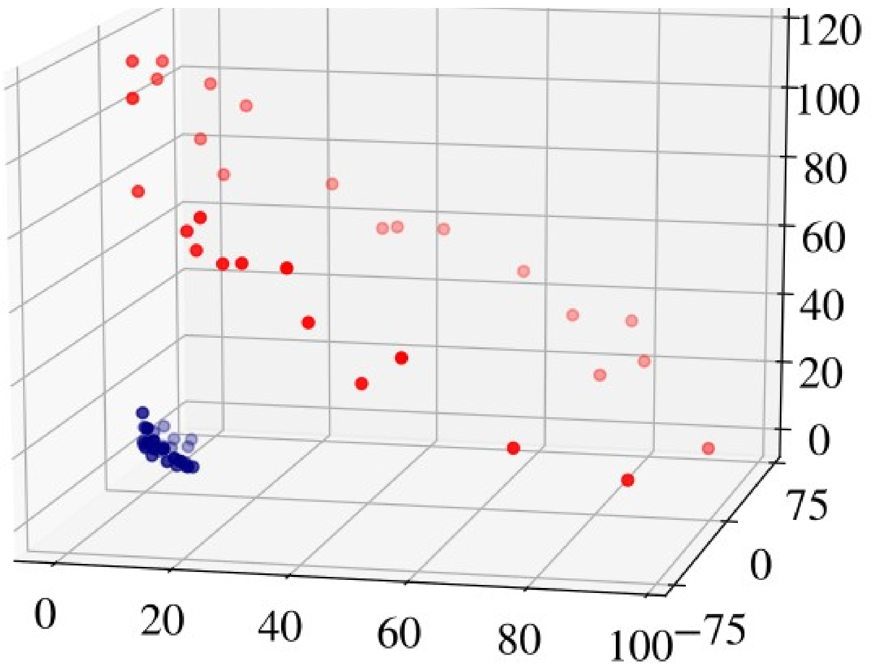
where the hyperparameter C determines the tradeoff between increasing the size of the decision boundary and ensuring that each $\mathbf{x}_i$ lies on the correct side of the decision boundary. The value of C is usually chosen experimentally, just like ID3's hyperparameters ϵ and d. SVMs that optimize hinge loss are called *soft-margin* SVMs, while the original formulation is referred to as a *hard-margin* SVM.

As you can see, for sufficiently high values of C, the second term in the cost function will become negligible, so the SVM algorithm will try to find the highest margin by completely ignoring misclassification. As we decrease the value of C, making classification errors is becoming more costly, so the SVM algorithm will try to make fewer mistakes by sacrificing the margin size. As we have already discussed, a larger margin is better for generalization. Therefore, C regulates the tradeoff between classifying the training data well (minimizing empirical risk) and classifying future examples well (generalization).

3.4.2 Dealing with Inherent Non-Linearity

SVM can be adapted to work with datasets that cannot be separated by a hyperplane in its original space. However, if we manage to transform the original space into a space of higher dimensionality, we could hope that the examples will become linearly separable in this transformed space. In SVMs, using a function to *implicitly* transform the original space into a higher dimensional space during the cost function optimization is called the **kernel trick**.

The effect of applying the kernel trick is illustrated in fig. 6. As you can see, it's possible to transform a two-dimensional non-linearly-separable data into a linearly-separable three-dimensional data using a specific mapping $\phi : \mathbf{x} \mapsto \phi(\mathbf{x})$, where $\phi(\mathbf{x})$ is a vector of higher dimensionality than $\mathbf{x}$. For the example of 2D data in fig. 5 (right), the mapping ϕ for example $\mathbf{x} = [q, p]$ that projects this example into a 3D space (fig. 6) would look like this $\phi([q, p]) \overset{\text{def}}{=} (q^2, \sqrt{2}qp, p^2)$, where q^2 means q squared. You see now that the data becomes linearly separable in the transformed space.



However, we don't know a priori which mapping ϕ would work for our data. If we first transform all our input examples using some mapping into very high dimensional vectors and then apply SVM to this data, and we try all possible mapping functions, the computation could become very inefficient, and we would never solve our classification problem.

Fortunately, scientists figured out how to use **kernel functions** (or, simply, **kernels**) to efficiently work in higher-dimensional spaces *without doing this transformation explicitly*. To understand how kernels work, we have to see first how the optimization algorithm for SVM finds the optimal values for $\mathbf{w}$ and b.

The method traditionally used to solve the optimization problem in eq. 8 is the *method of Lagrange multipliers*. Instead of solving the original problem from eq. 8, it is convenient to solve an equivalent problem formulated like this:

$$\max_{\alpha_1 \dots \alpha_N} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{k=1}^{N} y_i \alpha_i (\mathbf{x}_i \mathbf{x}_k) y_k \alpha_k \text{ subject to } \sum_{i=1}^{N} \alpha_i y_i = 0 \text{ and } \alpha_i \geq 0, i = 1, \dots, N,$$

where α_i are called Lagrange multipliers. When formulated like this, the optimization problem becomes a convex quadratic optimization problem, efficiently solvable by quadratic programming algorithms.

Now, you could have noticed that in the above formulation, there is a term $\mathbf{x}_i\mathbf{x}_k$, and this is the only place where the feature vectors are used. If we want to transform our vector space into higher dimensional space, we need to transform $\mathbf{x}_i$ into $\phi(\mathbf{x}_i)$ and $\mathbf{x}_j$ into $\phi(\mathbf{x}_j)$ and then multiply $\phi(\mathbf{x}_i)$ and $\phi(\mathbf{x}_j)$. It would be very costly to do so.

On the other hand, we are only interested in the result of the dot-product $\mathbf{x}_i\mathbf{x}_k$, which, as we know, is a real number. We don't care how this number was obtained as long as it's correct. By using the kernel trick, we can get rid of a costly transformation of original feature vectors into higher-dimensional vectors and avoid computing their dot-product. We replace that by a simple operation on the original feature vectors that gives the same result. For example, instead of transforming (q_1, p_1) into $(q_1^2, \sqrt{2}q_1p_1, p_1^2)$ and (q_2, p_2) into $(q_2^2, \sqrt{2}q_2p_2, p_2^2)$ and then computing the dot-product of $(q_1^2, \sqrt{2}q_1p_1, p_1^2)$ and $(q_2^2, \sqrt{2}q_2p_2, p_2^2)$ to obtain $(q_1^2q_2^2 + 2q_1q_2p_1p_2 + p_1^2p_2^2)$ we could find the dot-product between (q_1, p_1) and (q_2, p_2) to get $(q_1q_2 + p_1p_2)$ and then square it to get exactly the same result $(q_1^2q_2^2 + 2q_1q_2p_1p_2 + p_1^2p_2^2)$.

That was an example of the kernel trick, and we used the quadratic kernel $k(\mathbf{x}_i, \mathbf{x}_k) \stackrel{\text{def}}{=} (\mathbf{x}_i\mathbf{x}_k)^2$. Multiple kernel functions exist, the most widely used of which is the **RBF kernel**:

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right),$$

where $\|\mathbf{x} - \mathbf{x}'\|^2$ is the squared **Euclidean distance** between two feature vectors. The Euclidean distance is given by the following equation:

$$d(\mathbf{x}_i, \mathbf{x}_k) \stackrel{\text{def}}{=} \sqrt{\left(x_i^{(1)} - x_k^{(1)}\right)^2 + \left(x_i^{(2)} - x_k^{(2)}\right)^2 + \cdots + \left(x_i^{(N)} - x_k^{(N)}\right)^2} = \sqrt{\sum_{j=1}^{D}\left(x_i^{(j)} - x_k^{(j)}\right)^2}.$$

It can be shown that the feature space of the RBF (for "radial basis function") kernel has an infinite number of dimensions. By varying the hyperparameter σ, the data analyst can choose between getting a smooth or curvy decision boundary in the original space.

3.5 k-Nearest Neighbors

k-Nearest Neighbors (kNN) is a non-parametric learning algorithm. Contrary to other learning algorithms that allow discarding the training data after the model is built, kNN keeps all training examples in memory. Once a new, previously unseen example **x** comes in, the kNN algorithm finds k training examples closest to **x** and returns the majority label (in case of classification) or the average label (in case of regression).

The closeness of two points is given by a distance function. For example, Euclidean distance seen above is frequently used in practice. Another popular choice of the distance function is the negative **cosine similarity**. Cosine similarity defined as,

$$s(\mathbf{x}_i, \mathbf{x}_k) \stackrel{\text{def}}{=} \cos(\angle(\mathbf{x}_i, \mathbf{x}_k)) = \frac{\sum_{j=1}^{D} x_i^{(j)} x_k^{(j)}}{\sqrt{\sum_{j=1}^{D} \left(x_i^{(j)}\right)^2} \sqrt{\sum_{j=1}^{D} \left(x_k^{(j)}\right)^2}},$$

is a measure of similarity of the directions of two vectors. If the angle between two vectors is 0 degrees, then two vectors point to the same direction, and cosine similarity is equal to 1. If the vectors are orthogonal, the cosine similarity is 0. For vectors pointing in opposite directions, the cosine similarity is -1. If we want to use cosine similarity as a distance metric, we need to multiply it by -1. Other popular distance metrics include Chebychev distance, Mahalanobis distance, and Hamming distance. The choice of the distance metric, as well as the value for k, are the choices the analyst makes before running the algorithm. So these are hyperparameters. The distance metric could also be learned from data (as opposed to guessing it). We talk about that in Chapter 10.

Now you know how the model building algorithm works and how the prediction is made. A reasonable question is what is the cost function here? Surprisingly, this question has not been well studied in the literature, despite the algorithm's popularity since the earlier 1960s. The only attempt to analyze the cost function of kNN I'm aware of was undertaken by Li and Yang in 2003[4]. Below, I outline their considerations.

For simplicity, let's make our derivation under the assumptions of binary classification ($y \in \{0,1\}$) with cosine similarity and **normalized** feature vectors[5]. Under these assumptions, kNN does a locally linear classification with the vector of coefficients,

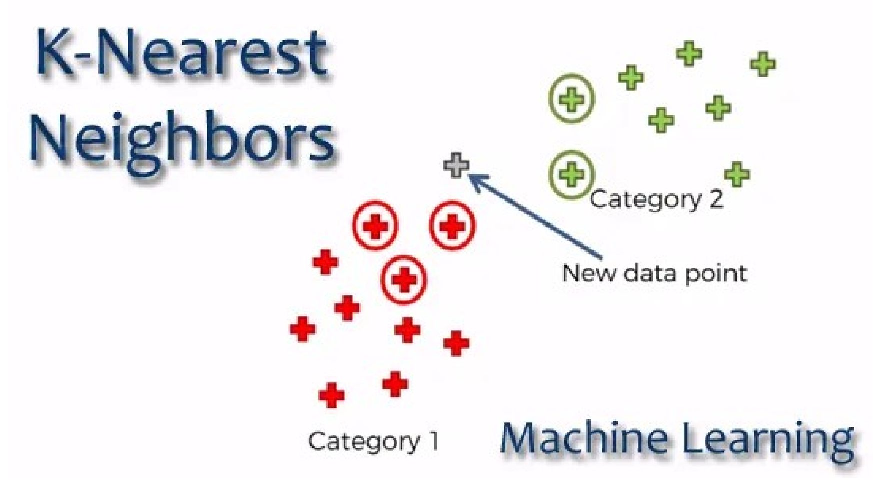
$$\mathbf{w_x} = \sum_{(\mathbf{x}',y') \in \mathcal{R}_k(\mathbf{x})} y'\mathbf{x}', \tag{9}$$

where $\mathcal{R}_k(\mathbf{x})$ is the set of k nearest neighbors to the input example $\mathbf{x}$. The above equation says that we take the sum of all nearest neighbor feature vectors to some input vector $\mathbf{x}$ by ignoring those that have label 0. The classification decision is obtained by defining a threshold on the dot-product $\mathbf{w_x x}$ which, in the case of normalized feature vectors, is equal to the cosine similarity between $\mathbf{w_x}$ and $\mathbf{x}$.

Now, defining the cost function like this:

$$L = -\sum_{(\mathbf{x}',y') \in R_k(\mathbf{x})} y'\mathbf{x}'\mathbf{w_x} + \frac{1}{2}||\mathbf{w}||^2$$

and setting the first order derivative of the right-hand side to zero yields the formula for the coefficient vector in eq. 9.


K-Nearest Neighbors
Category 2
New data point
Category 1
Machine Learning