AIML430/COMP309: ML Tools and Techniques
Lecture 3: ML Models & Algorithms—Classification

Ali Knott
School of Engineering and Computer Science
Victoria University of Wellington

My job...

...is to introduce a whole lot of ML models for you.

- The focus is on practical tools that you can use...
- So I tell you just enough to let you start using those tools.

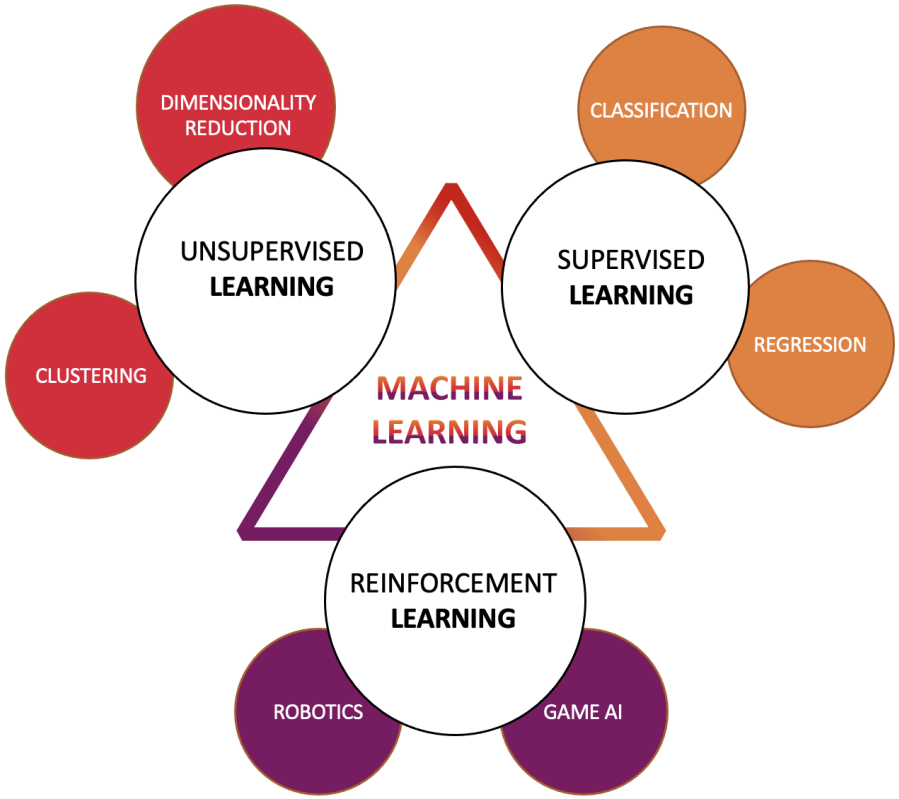
This week I'll introduce the most commonly used classifiers.

- We'll focus on supervised classification methods.
  (Clustering is an unsupervised way of putting things into classes.)

But first, some terminology!

Machine learning ‘tasks’

ML can be used to achieve a few different tasks.



We’ll get into most of these. This week is classification!

Definitions: Models and algorithms

We'll be discussing ML 'models', and ML 'algorithms'.

- A ML algorithm executes a *training process*, that runs on a set of *training data* ('training set').
- The *result* of this training process is a ML model.

Another term in ML literature is 'probability model'.

- A probability model is often used to capture how a task should *ideally* be defined.
- A ML algorithm often learns an *approximate model* of this ideal. I'll give some examples below.

Definitions: Inputs and outputs

A classifier maps some set of input features $X_1 \ldots X_n$ onto some discrete output Y, which is a class in the set $\{C_1 \ldots C_m\}$.

- E.g. input features are image pixels; output is an object category.
- E.g. input features are monitored user behaviour on a social media site; output is 'click likelihood' for a new ad.

Aside: a regression model maps a set of input features onto a *continuous output* Y.

- We'll talk about regression next week!

Two possible purposes for a classifier

One purpose is simply that we want to be able to *predict* the output from the input.

- Often, we know the input features, but not the output.
  - It might be in the future. . . or it might just be unknown. . .
- If we can build a machine that accurately does the prediction, this can be very useful.

Another purpose is to *better understand* how inputs map to outputs. (To get *insight* into the mapping.)

- A classifier can be used to analyse which input features are *most important*—or to understand *relationships* between input features.

Definitions—and visualisations

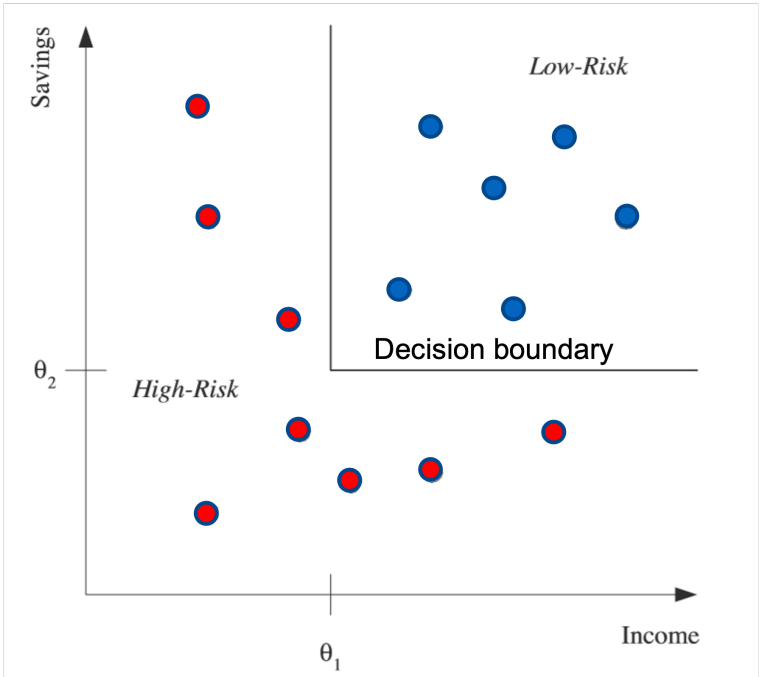
The input space is the space of all possible input feature combinations.

- If there are n input features, the space has n dimensions.

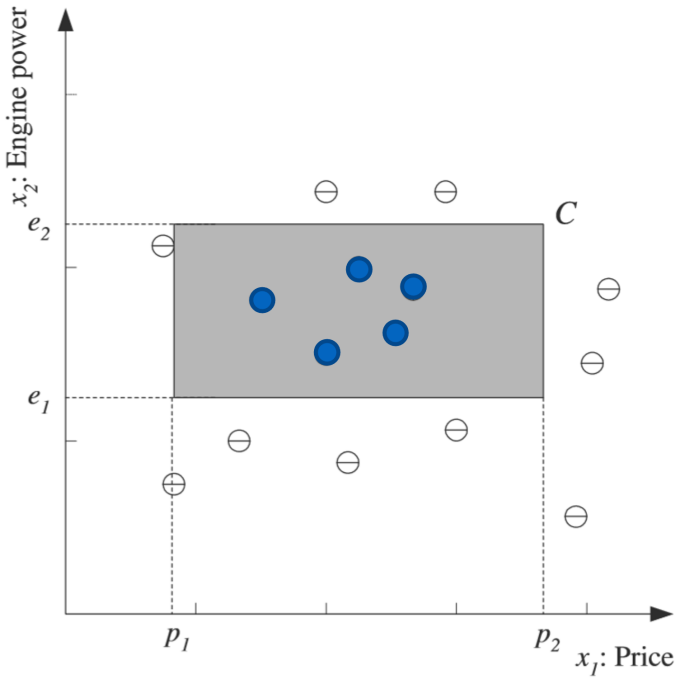
Items in the training set can be represented as points in the space.

- With different colours for different categories.

Here's an example from insurance:

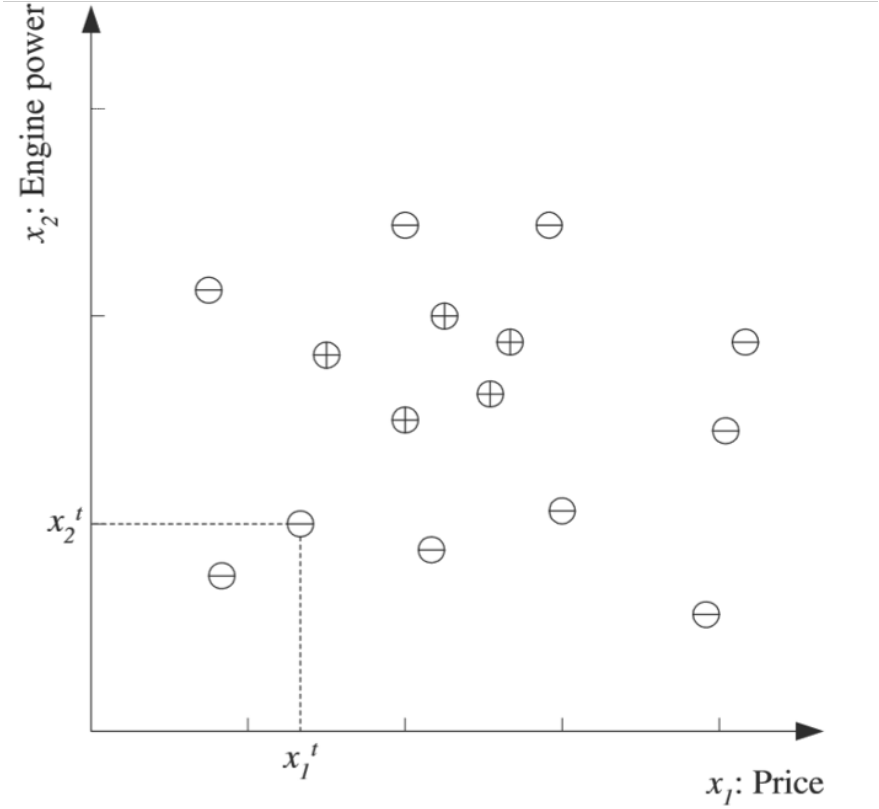


Here's one from car sales:



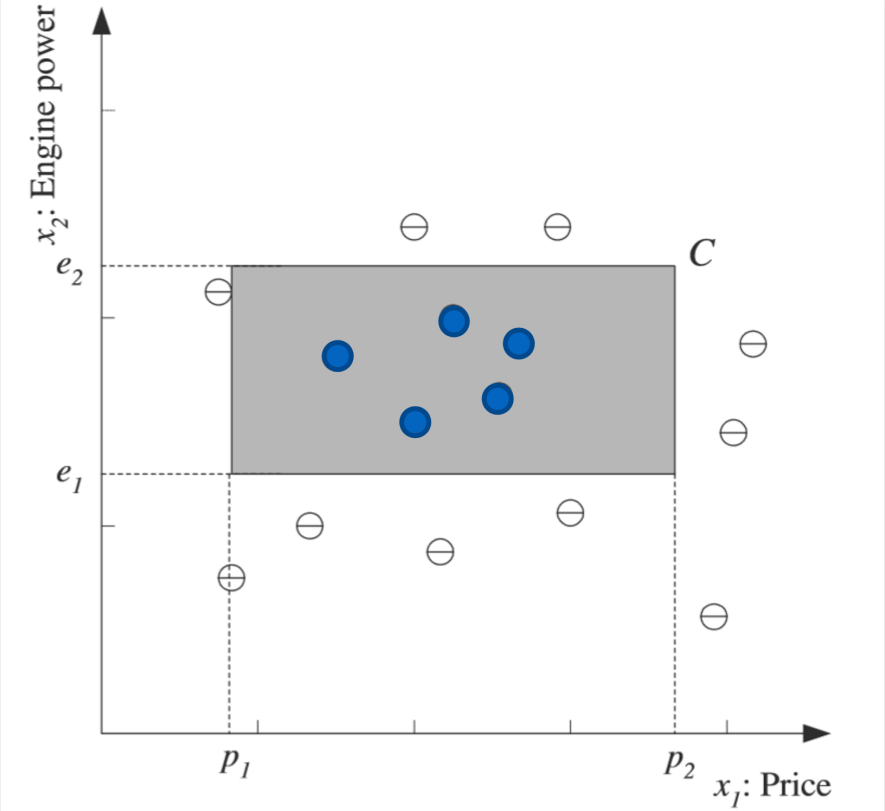
Some useful concepts defined visually

Each instance in the input space has a defined value for each feature.



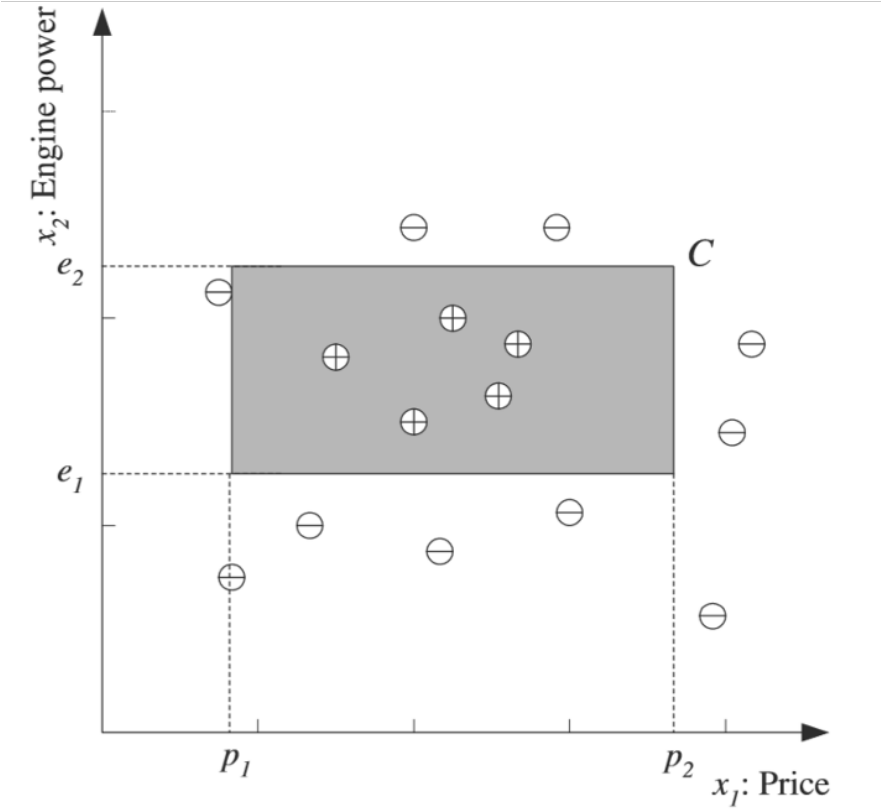
Some useful concepts defined visually

Each class can be defined as a region (or volume) of the input space.



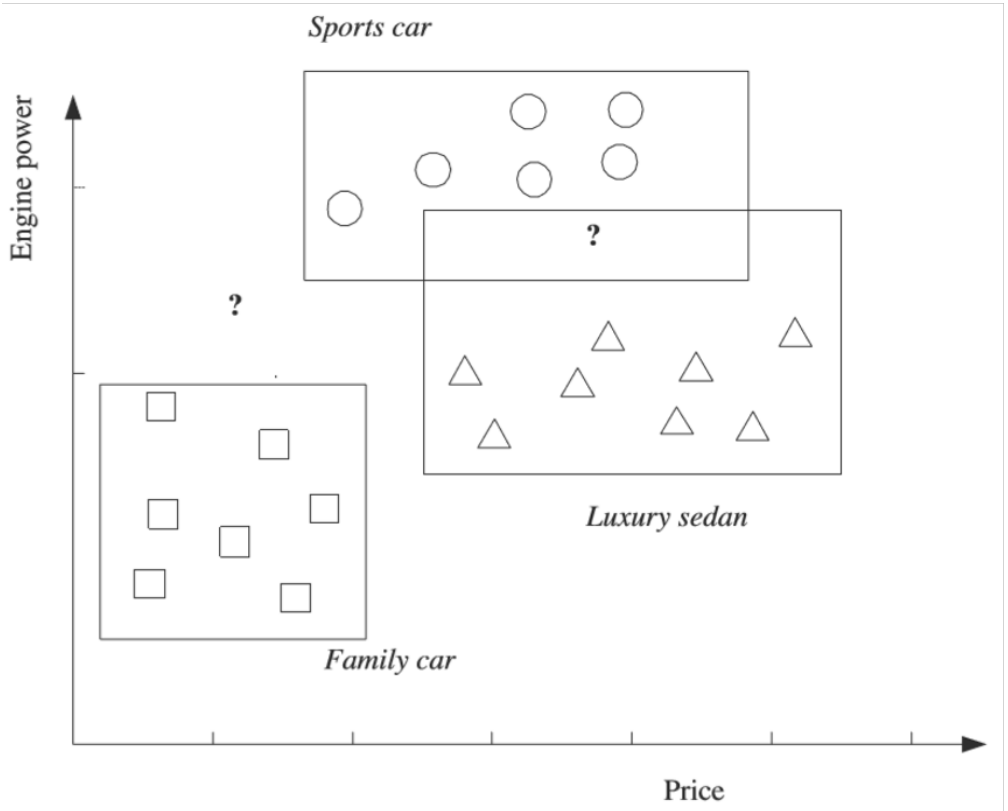
Some useful concepts defined visually

Classification tasks can be binary (with just two classes)



Some useful concepts defined visually

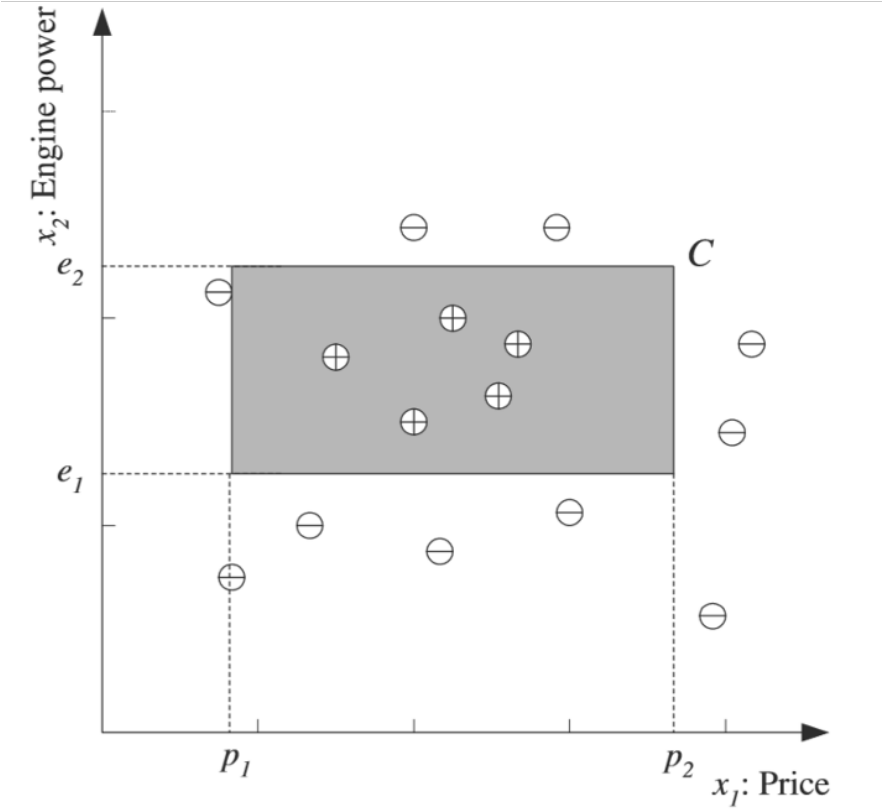
Classification tasks can be binary (with just two classes)
Or multi-class (with more than two classes).



Some useful concepts defined visually

Defining regions for classes may get a few things wrong. . .

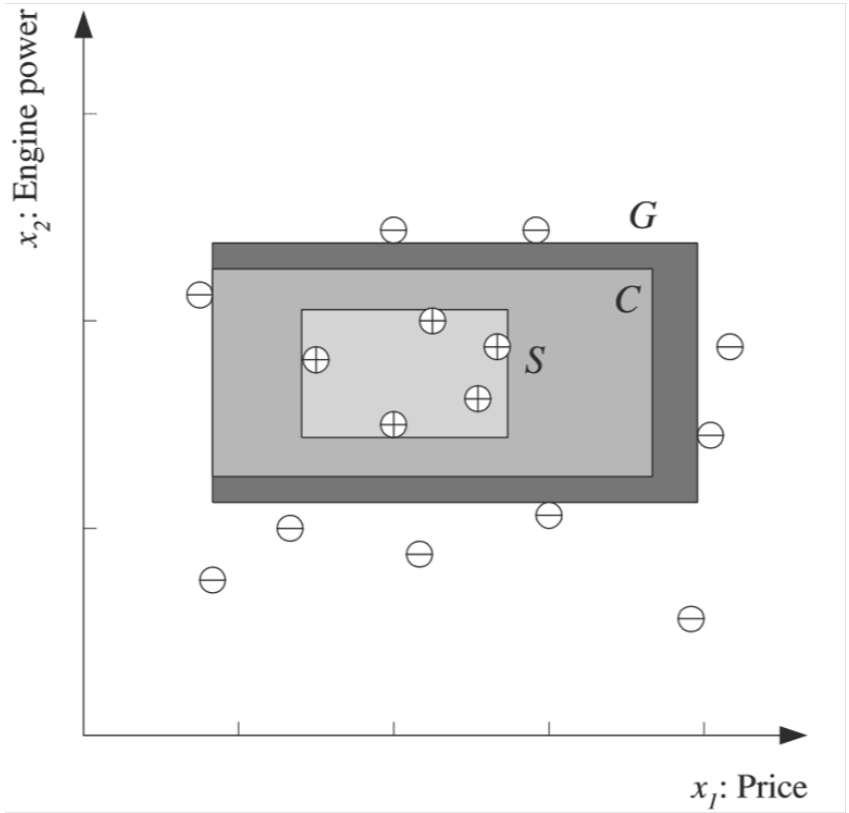
- For binary tasks with Boolean outputs (‘positive’, ‘negative’), we can speak of false positives, and false negatives.



Some useful concepts defined visually

We can define our class-region ‘tightly’ or ‘loosely’ around the training instances.

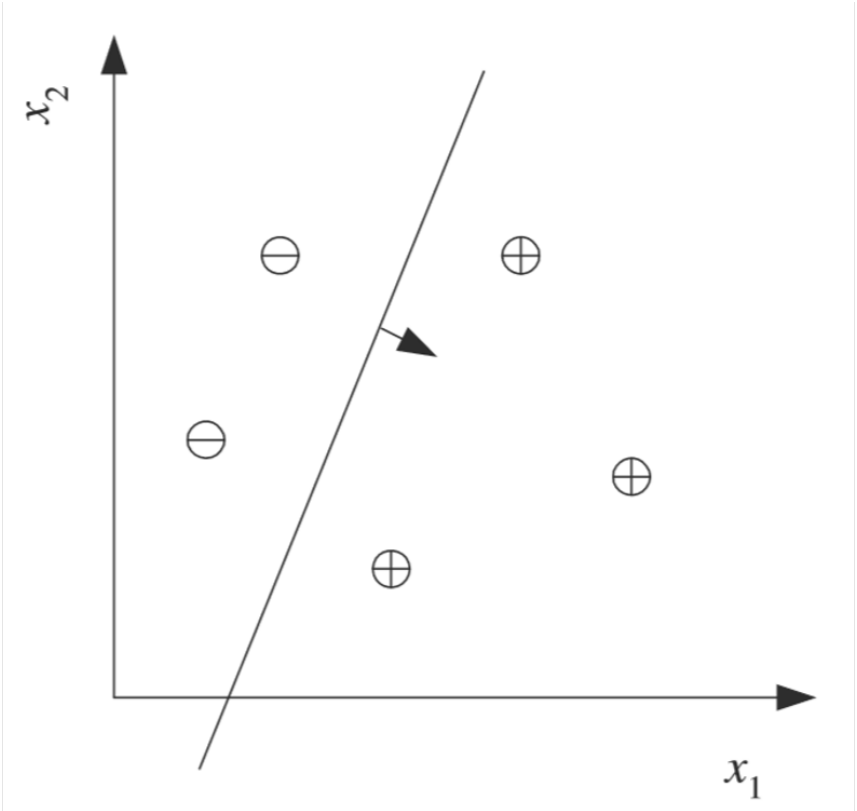
- Tight gives a ‘specific’ (S) definition; loose gives a ‘general’ one.



Some useful concepts defined visually

The ‘boundaries’ between class-regions can be linear

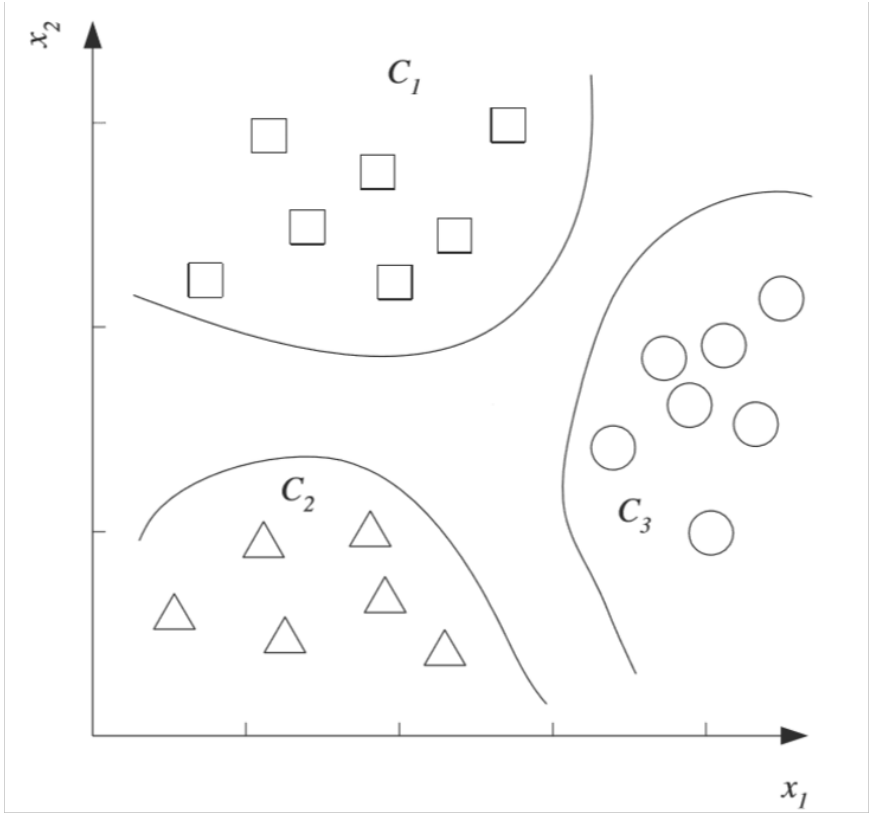
- Non-linear ones can be curvy to different degrees.



Some useful concepts defined visually

The ‘boundaries’ between class-regions can be linear or non-linear.

- Non-linear ones can be curvy to different degrees.



The classifier methods we'll look at

We'll cover five popular classifiers.

- K nearest neighbours (KNN) classifiers
- Perceptrons
- Naive Bayes classifiers
- Decision Trees
- Support vector machines (SVMs)

We'll also cover two ensemble classifiers, that *combine* classifiers.

- Random Forests
- Boosting

Classifier 1: K nearest neighbours

A K-nearest neighbours (KNN) classifier takes a new (unseen) instance, and works out its *distance* (in input space) to all the training instances.

- It picks the K *nearest* training instances, and outputs the *majority class* within those nearest instances.
- K is a hyperparameter, which must be separately defined.

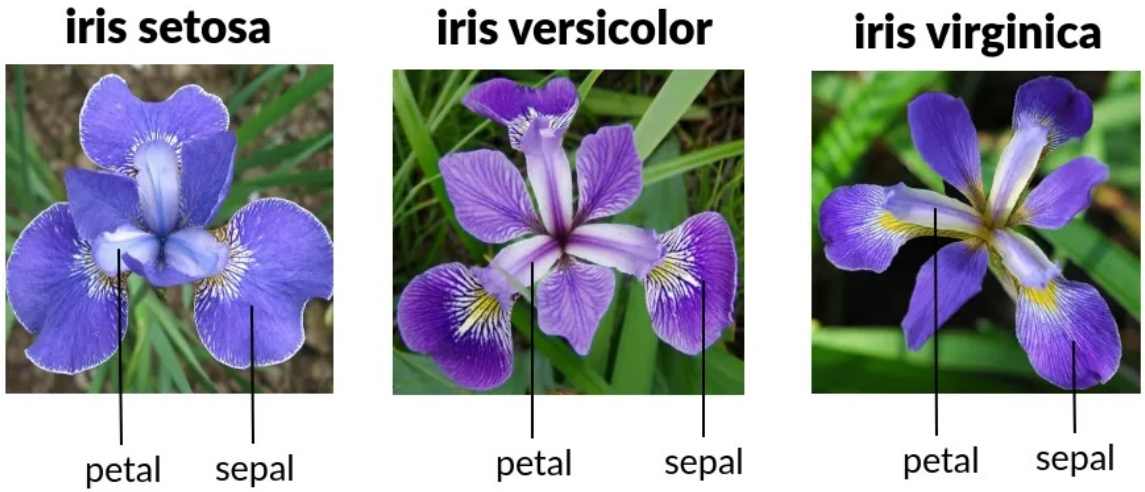
KNN doesn't explicitly learn a model. It just *remembers* all the training instances.

- So it's really fast to train—actually, there's no training process!
- But it can be super slow to *use*, because you have to compute lots of distances each time.

In simple cases, you can compute the category for *each point* in the input space. This is called a 'Voronoi tessellation'—useful to illustrate.

Aside: the Iris dataset

This is a classic dataset for illustrating ML concepts. It's for classifying three types of Iris flower:

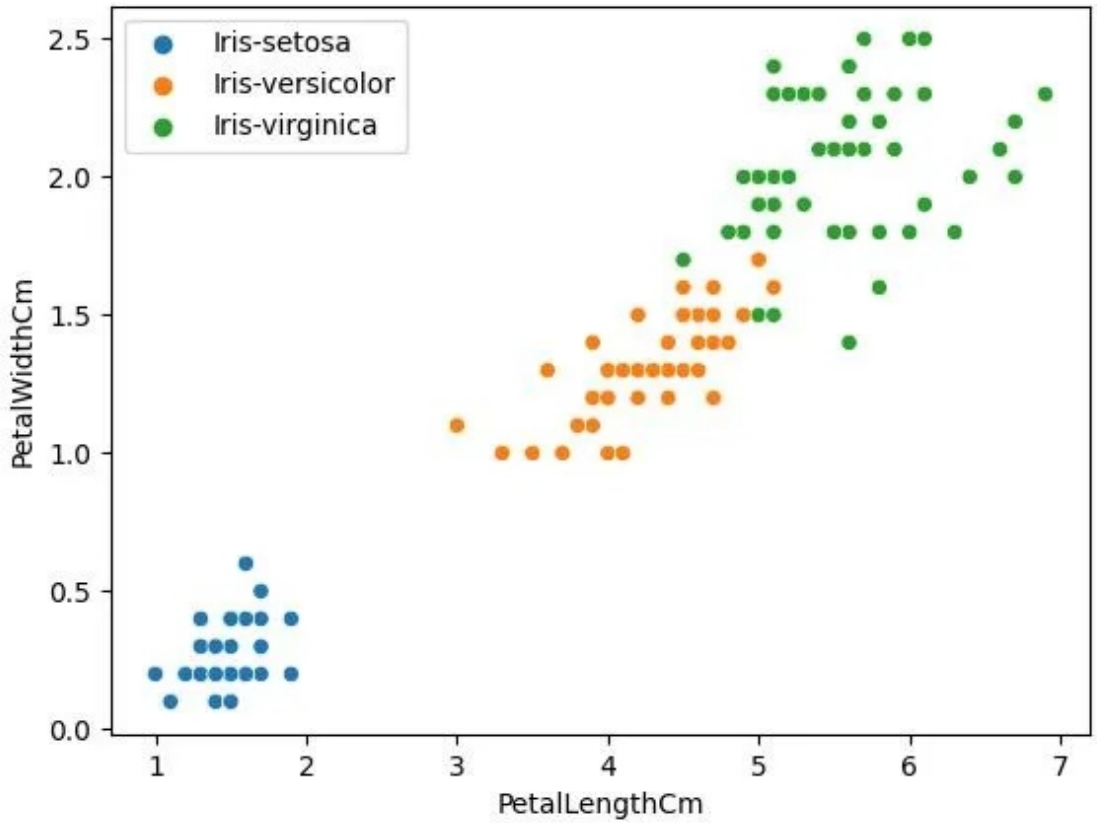


There are four input features for each flower:

- Petal length, petal width
- Sepal length, sepal width

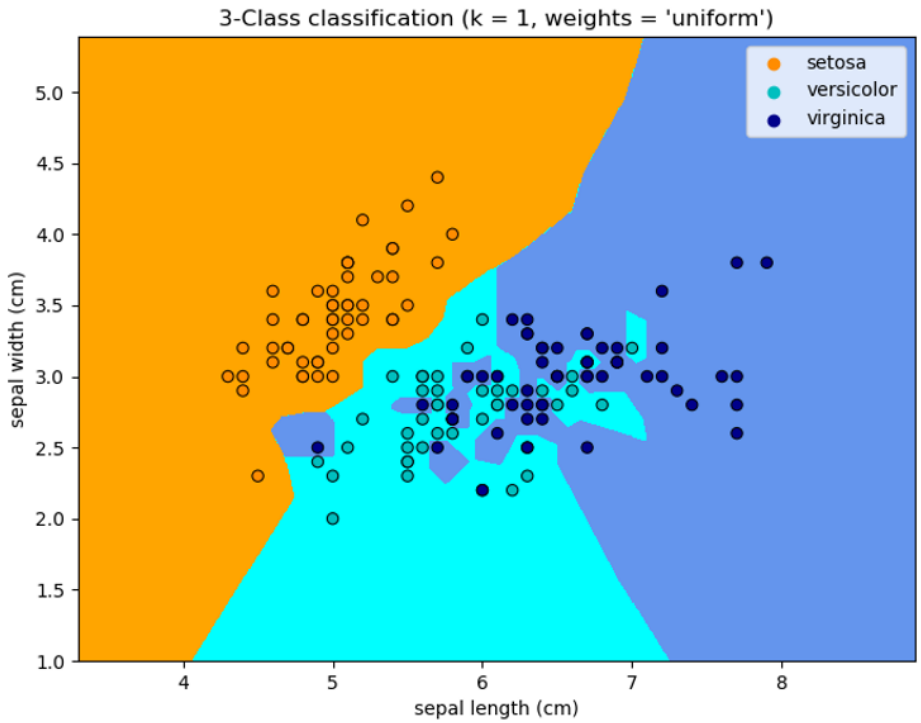
Aside: the Iris dataset

The classes are broadly distinguished in feature space, but overlap a little too.



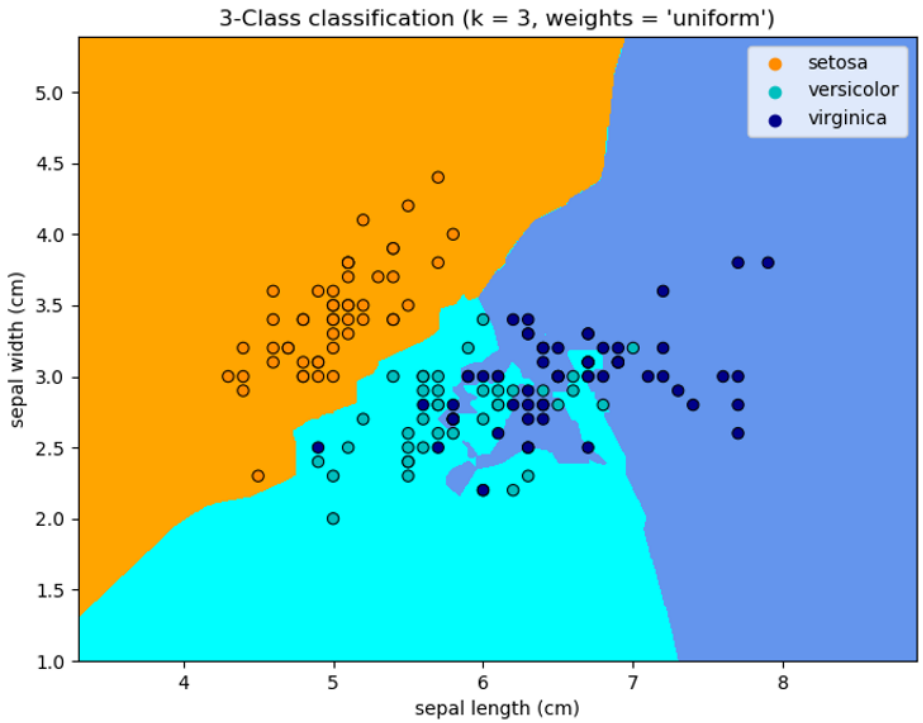
Voronoi diagrams for KNN on the Iris dataset

KNN with K=1:



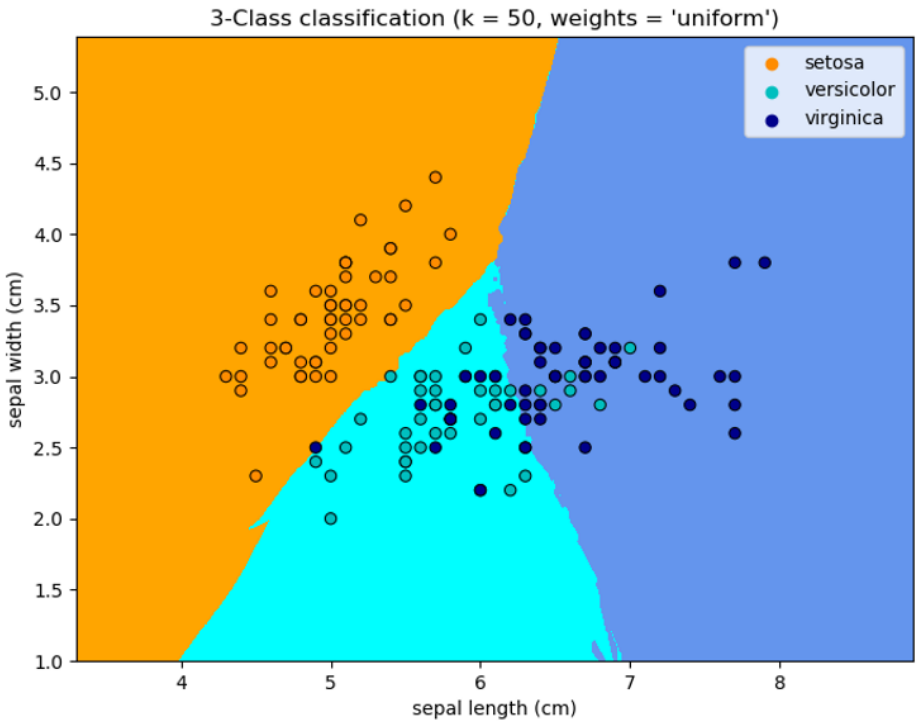
Voronoi diagrams for KNN on the Iris dataset

KNN with K=3:



Voronoi diagrams for KNN on the Iris dataset

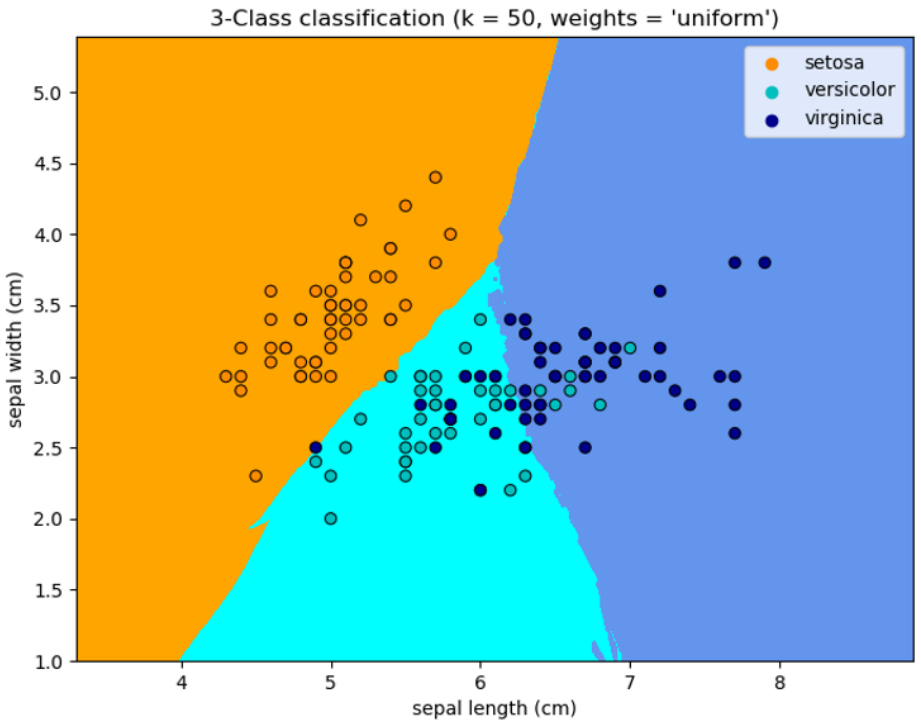
KNN with K=50:



Voronoi diagrams for KNN on the Iris dataset

What happens to decision boundaries as K increases?

- They get smoother…
- There are different pros and cons of high and low values of K.



Evaluating classifiers, and picking hyperparameters

To evaluate a classifier, we have to measure its performance on some *unseen instances*, that weren't in the training set.

- The set of unseen instances 'held out from training' is called the test set.

If your classifier learns decision boundaries that are too jagged, it won't generalise well to unseen instances.

- That's called overfitting. We want to avoid that!

If your classifier learns decision boundaries that are too smooth, it won't capture *actual patterns* in the input space.

- That's called underfitting. We want to avoid that too!

We can explore the tradeoff between over/underfitting by trying *different hyperparameters*.

Choosing values of K

A low value of K is likely to *overfit*...
A high value of K is likely to *underfit*.
How can we find a good value of K, that minimises these problems?

- The answer is to use a validation set.

Evaluating classifiers, and picking hyperparameters

We're not allowed to consult the test set, to find a good value of K.

- The test set must remain unseen, or we're cheating!

Instead, we can hold out a second set from training, called the validation set.

- We can use the validation set to find the right value of K.
- And to tune our other hyperparameters (if there are others).

Classifier 1: *K* nearest neighbours

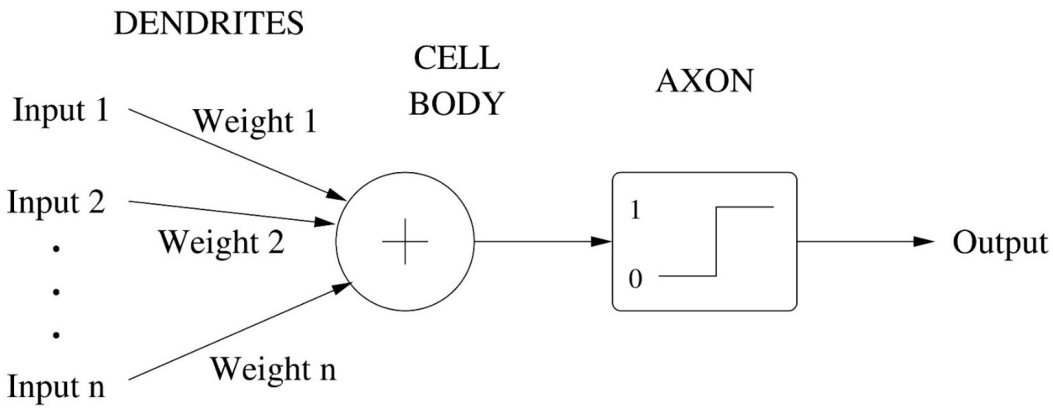
A code snippet:

```
from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(ta_x, ta_y)
pre_y = neigh.predict(te_x[0:1])
```

Classifier 2: the Perceptron

A perceptron is an early kind of neural network.

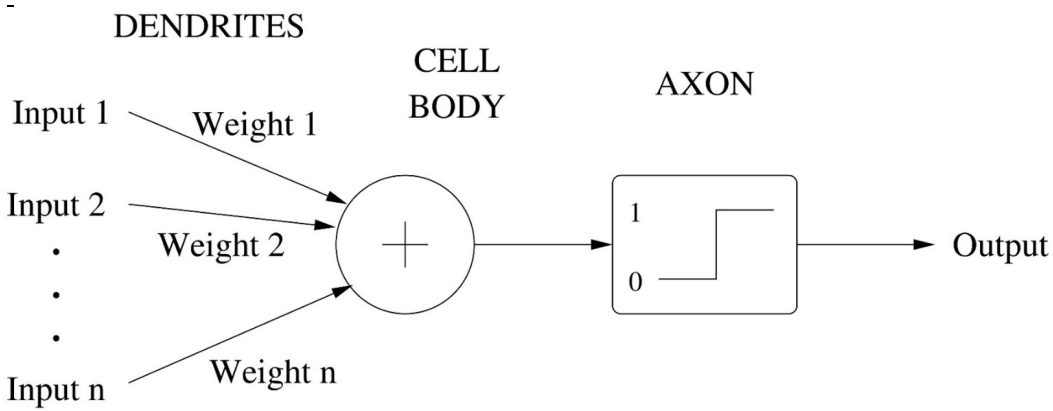
- The simplest kind is just a binary classifier, with two output classes. This is like a really simple model of 'one neuron'.
- It takes a set of input features, represented as real numbers.
- It computes a weighted sum of these inputs.
- If this sum is greater than 1, the output class is 0; otherwise it's 1.



Classifier 2: the Perceptron

The perceptron learning algorithm progressively *changes the weights* in the weighted sum. At each iteration,

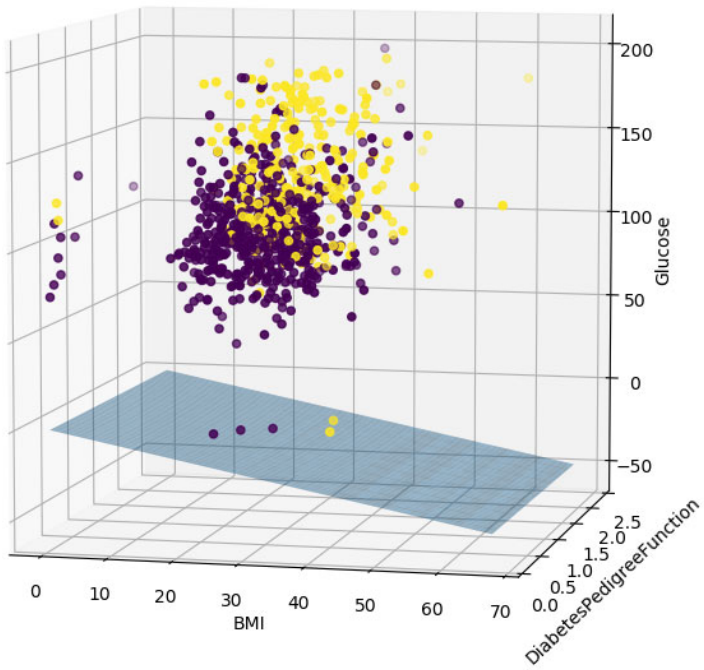
- We calculate the output for the training items...
- And look at the *errors* we make...
- Then we incrementally change the weights, to reduce the number of errors.
- We can make these changes for each item, or for the whole training set.

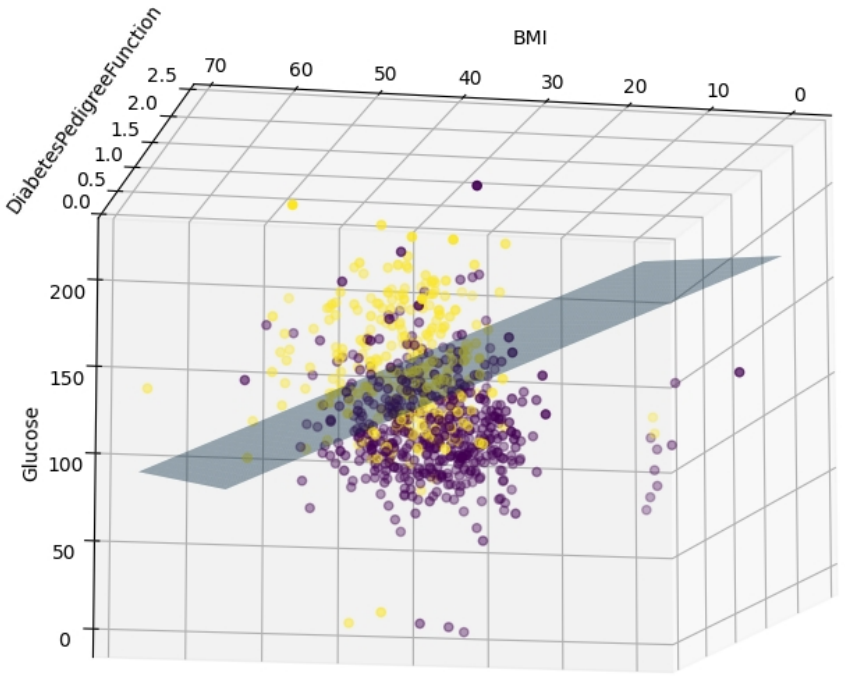


Visualising the training process

Each possible combination of weights defines a ‘hyperplane’ in input space, separating the two output classes.

- Training moves the hyperplane towards an optimal position.

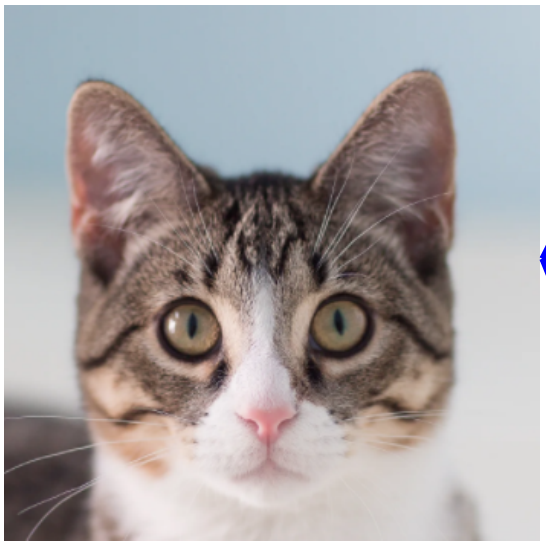


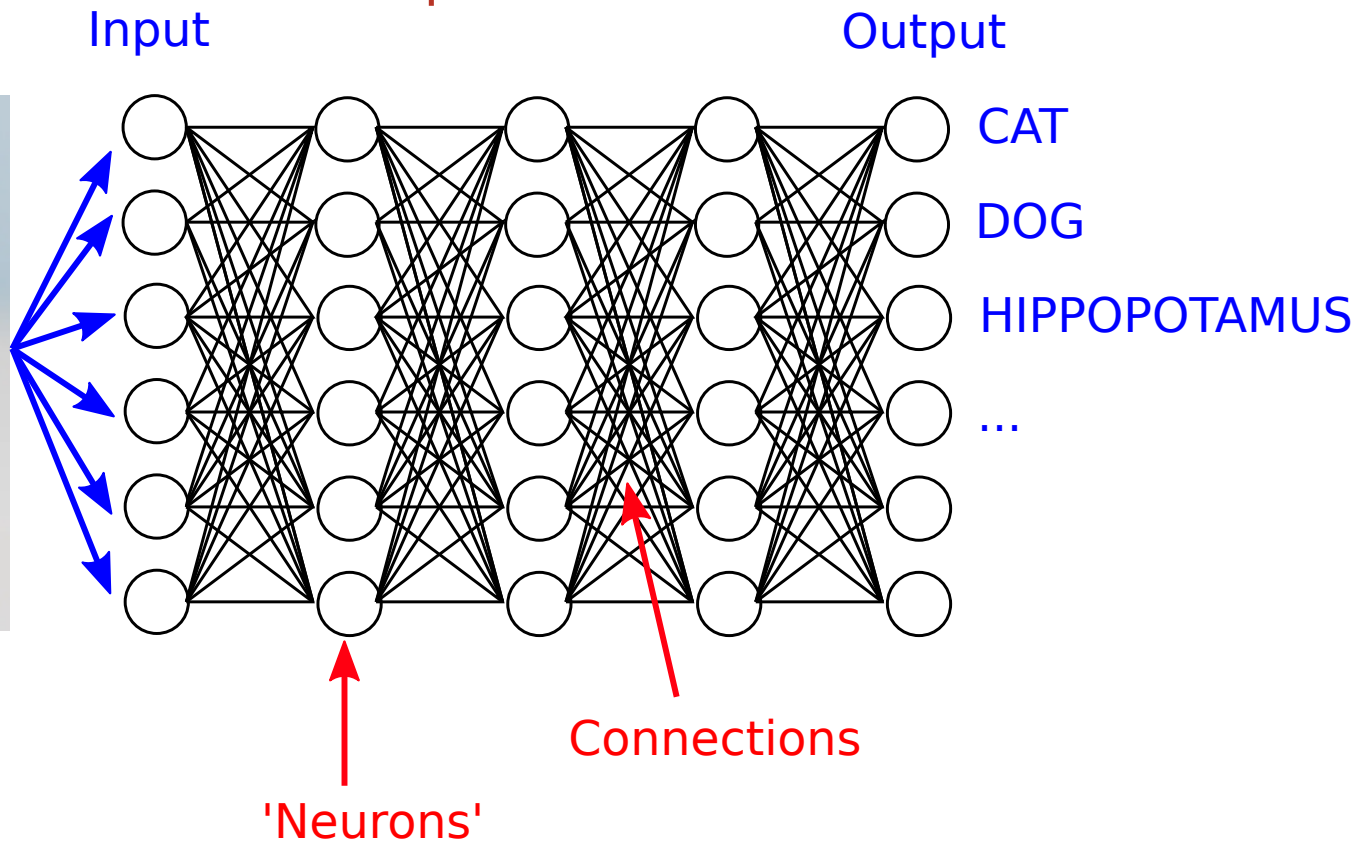


Weights on a bias input let the hyperplane move away from the origin.

The training process in a bit more detail

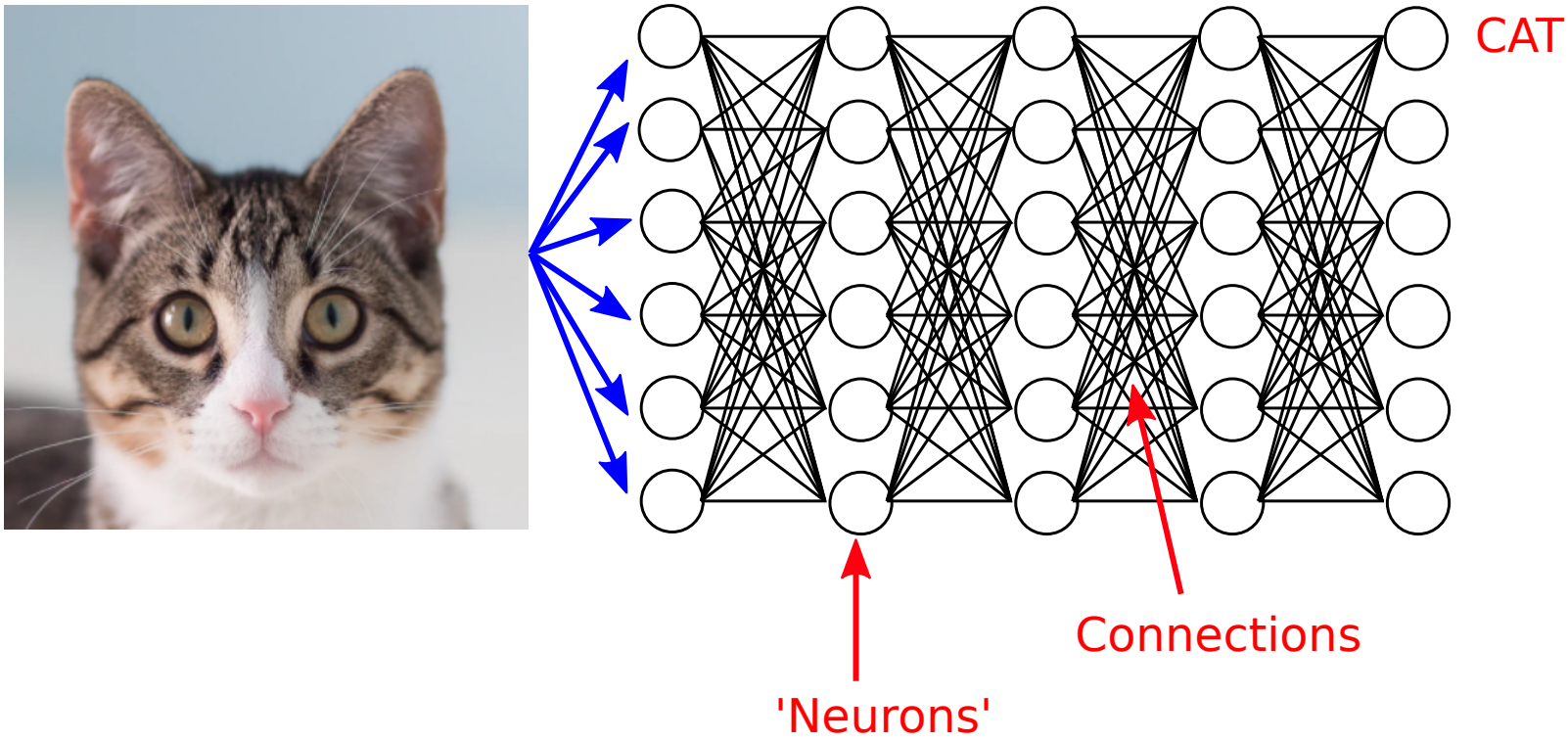
Modern perceptrons are organised into *layers* of neuron-like units. Neurons are linked together by connections. One layer holds the input to the network; another holds its output.





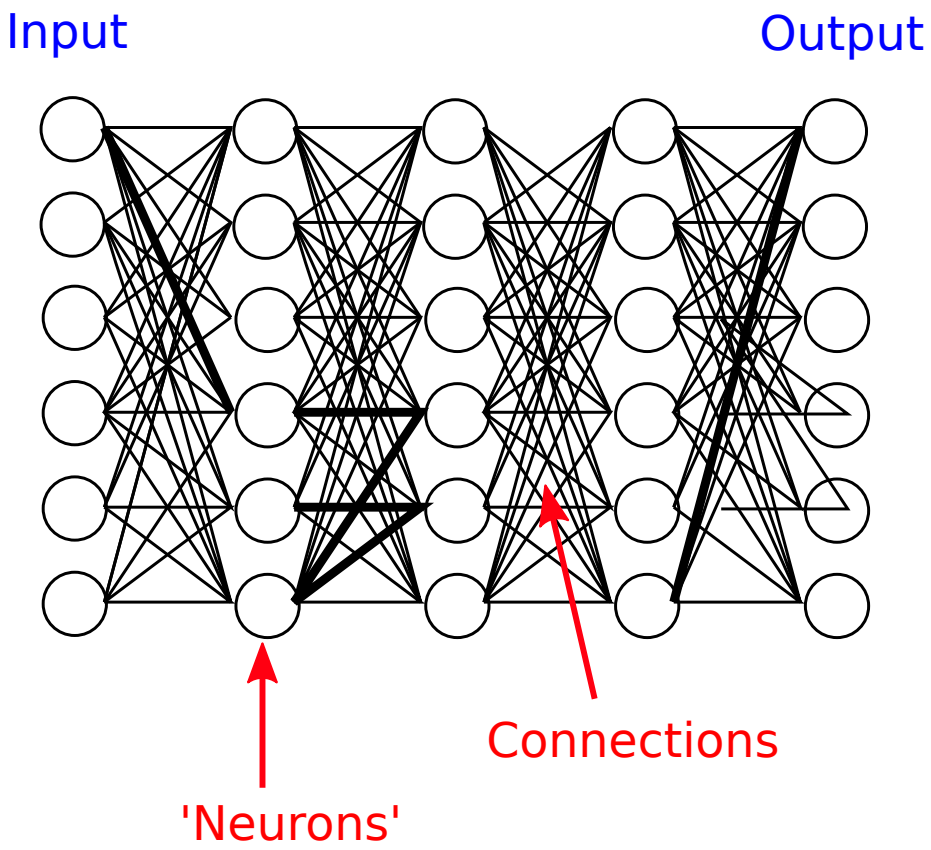
The training process in a bit more detail

To build a classifier, we train it on many training examples.
(Say we have 1000.)



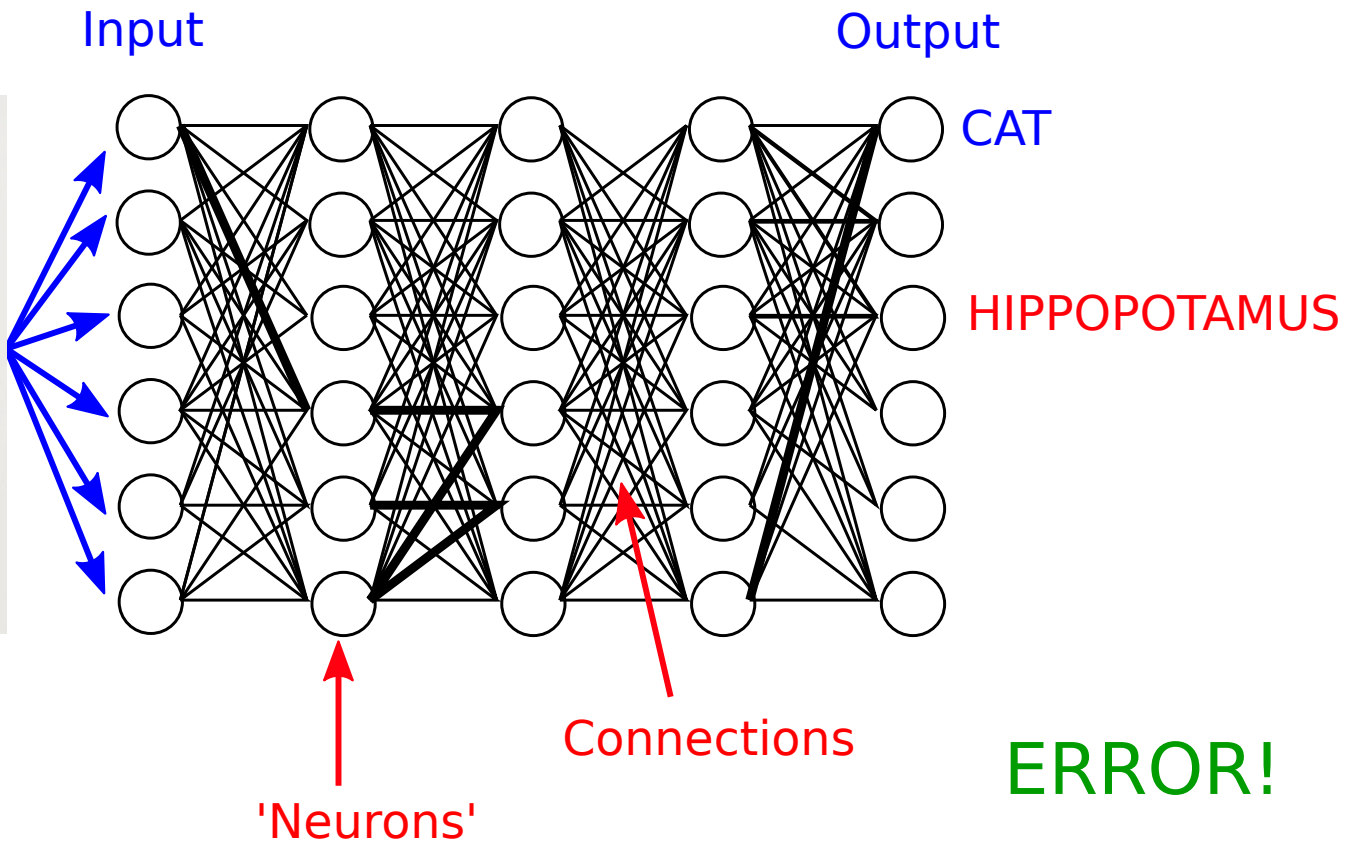
The training process in a bit more detail

At the start of training, we set all the weights to *random values*.



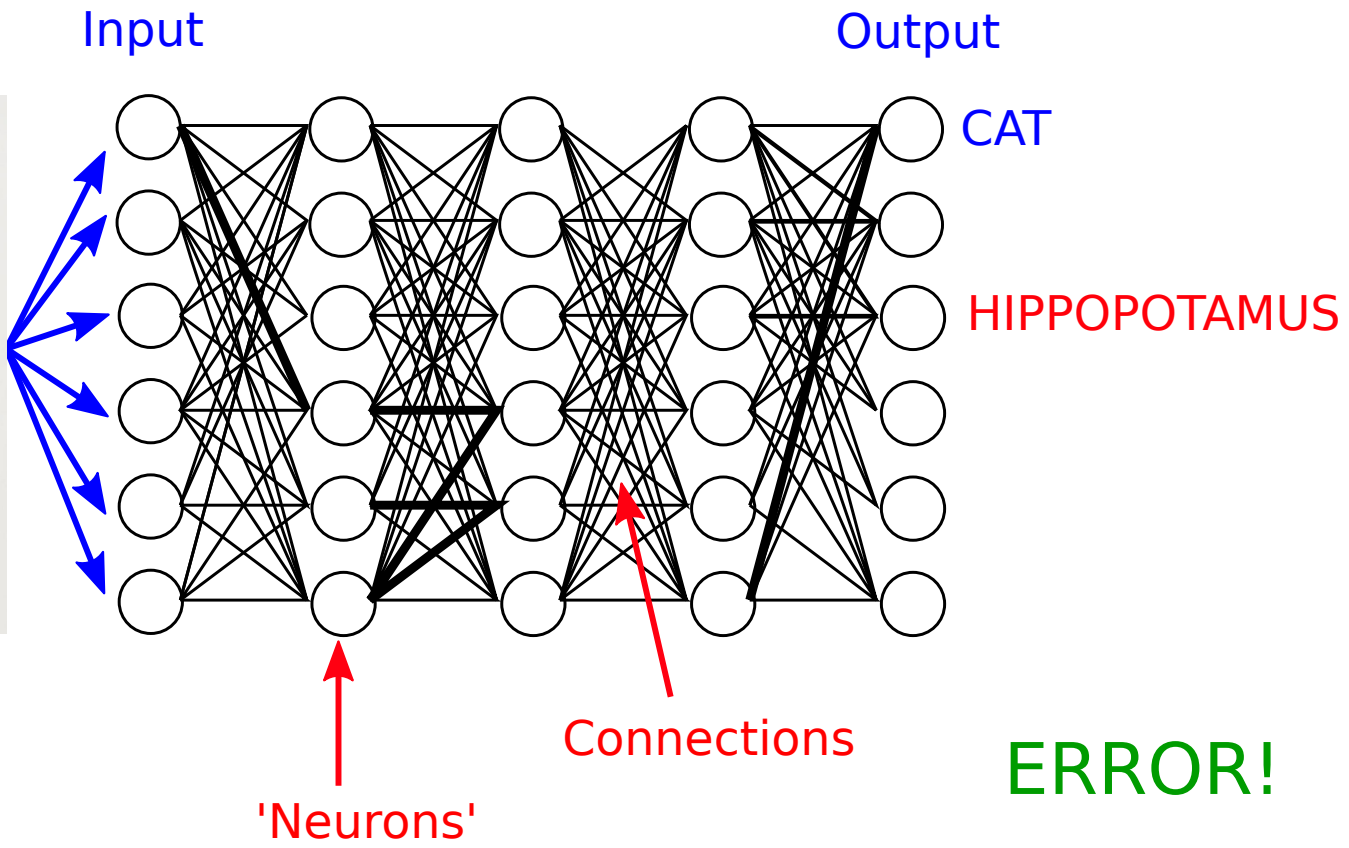
The training process in a bit more detail

To begin with, the network will output nonsense. But we can *score* how well it does, because we know what the answers should be.



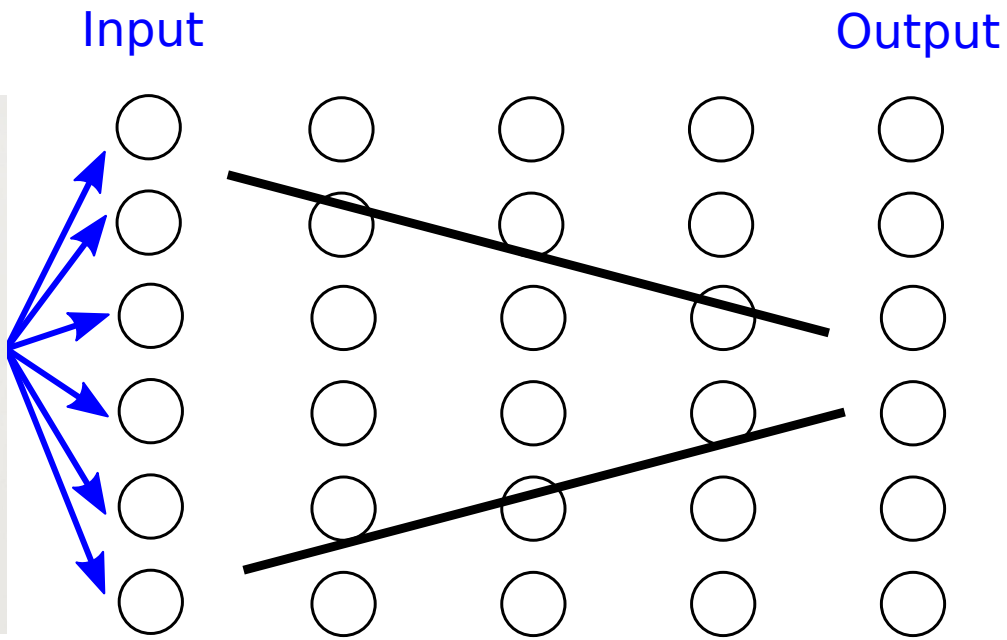
The training process in a bit more detail

Training involves *changing the weights to improve the score*.



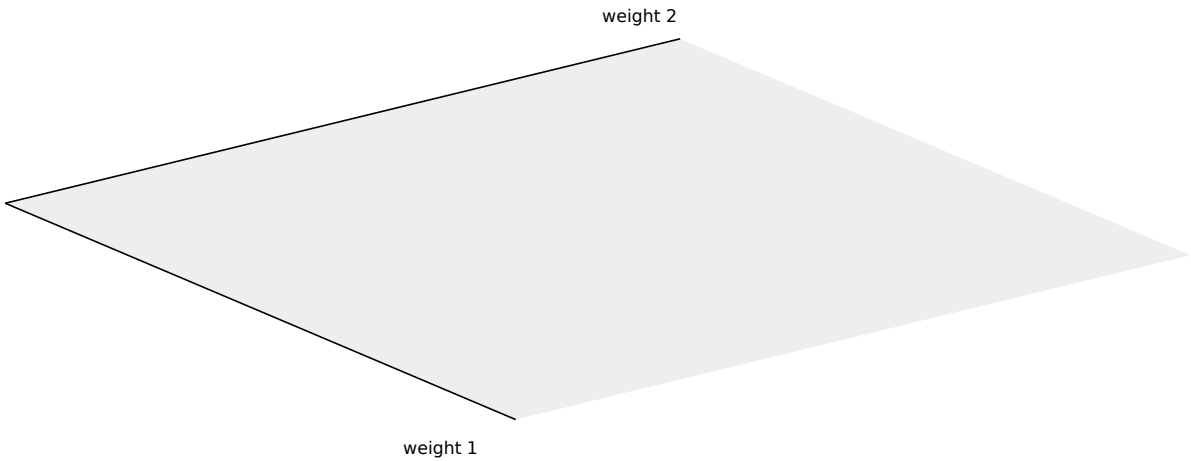
The training process in a bit more detail

To visualise training, assume the network has just *two weights*.



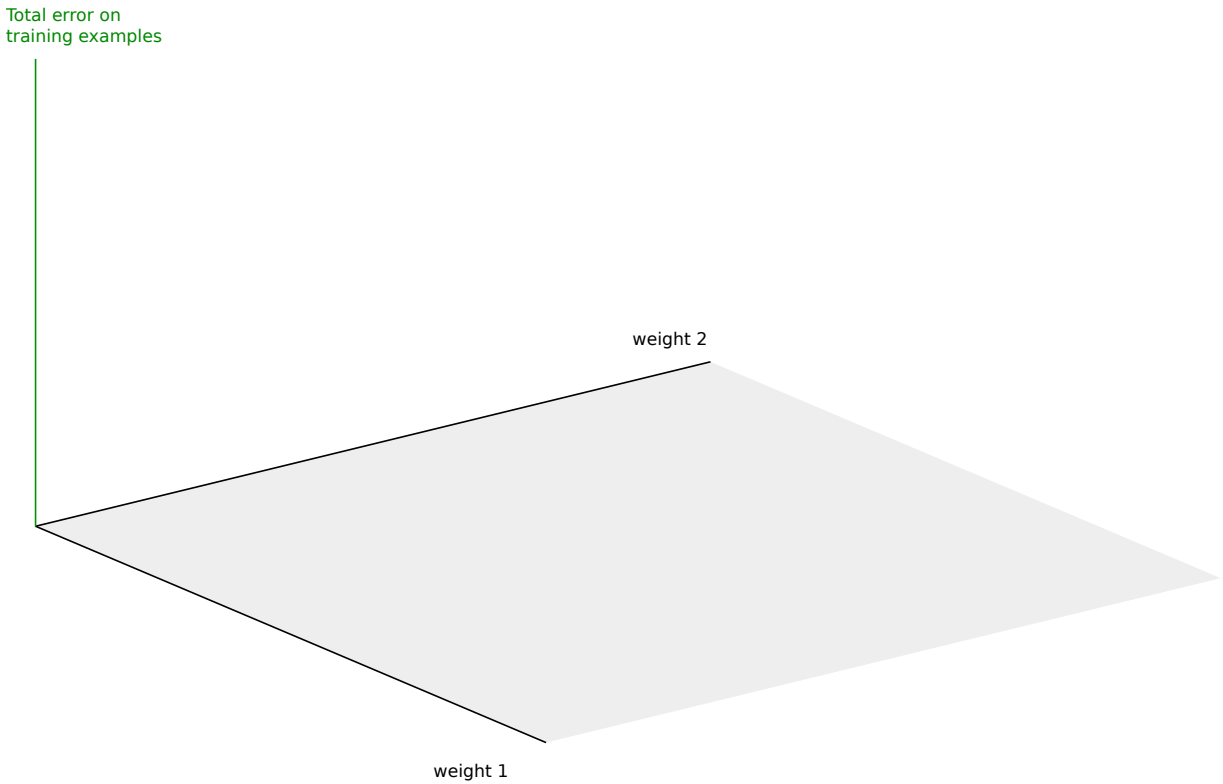
The training process in a bit more detail

Now imagine a two-dimensional graph, where we can plot *all possible values* of those two weights.



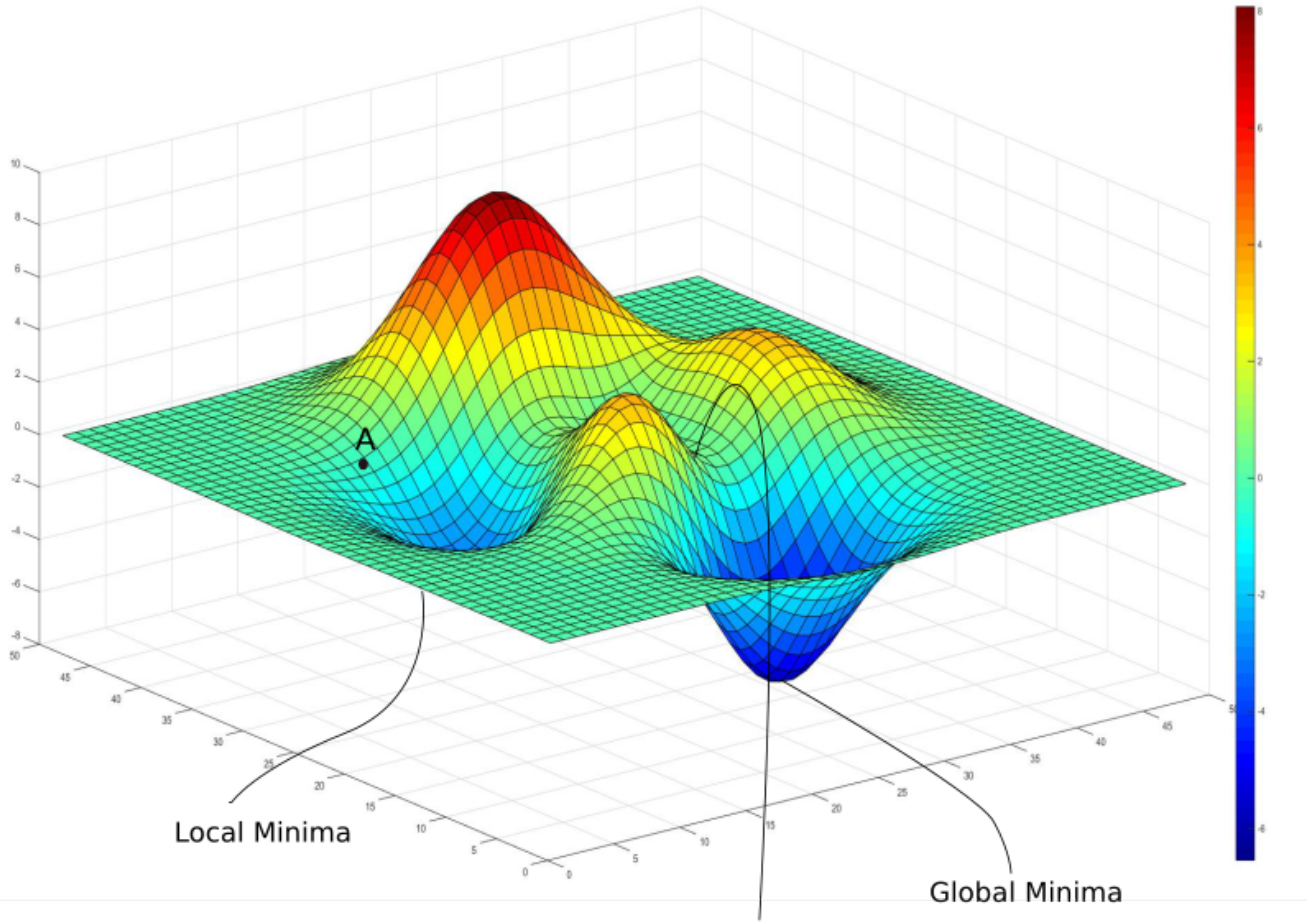
The training process in a bit more detail

Now visualise a third dimension, representing the number of errors for each weight combination.



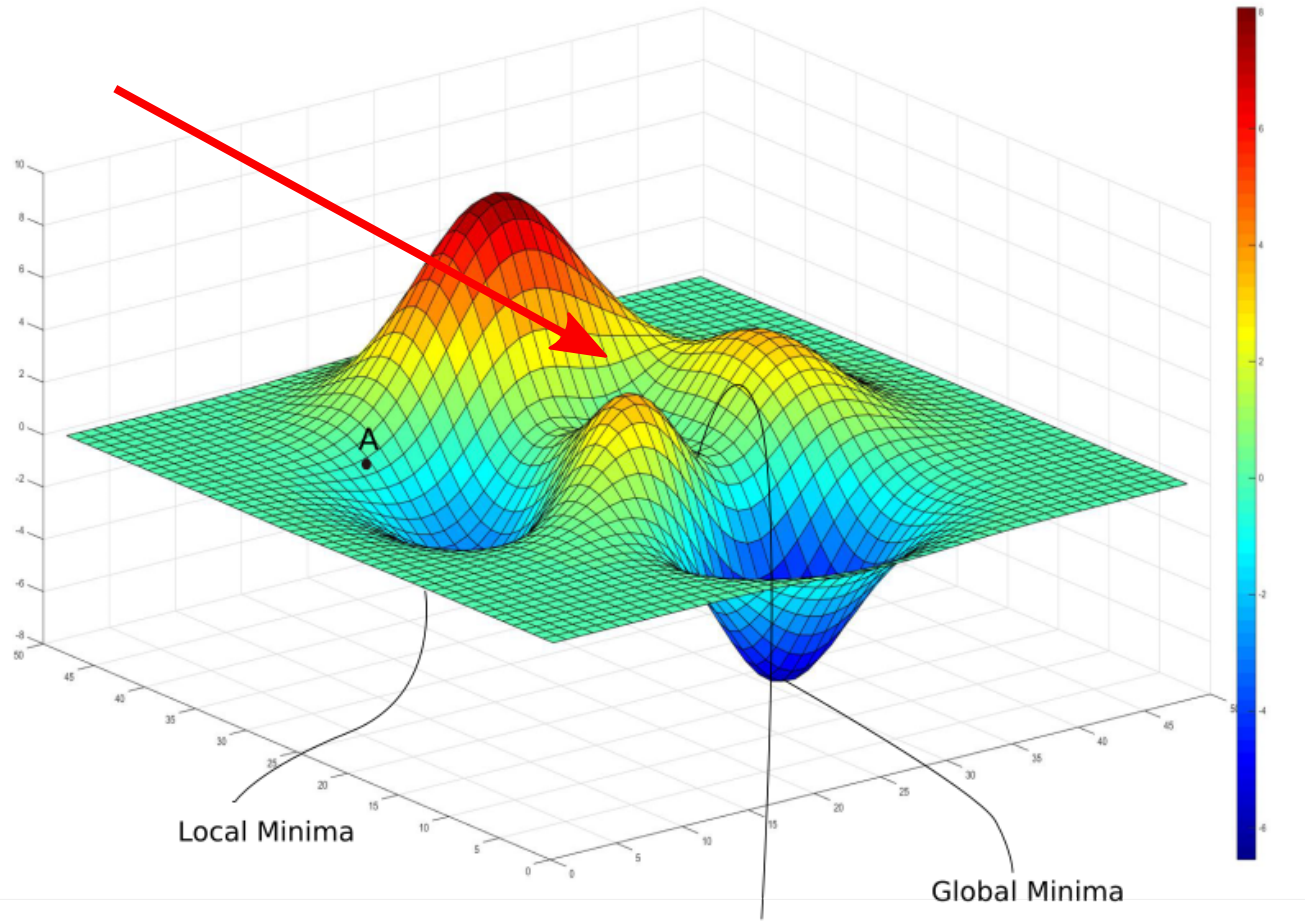
The training process in a bit more detail

In this graph, we can visualise an error surface, which shows the error for *each* set of weights.



The training process in a bit more detail

From here, training just involves *moving downhill* in the error surface. This training algorithm is called gradient descent.



Regularisation for perceptrons

Many machine learning models learn a set of 'weights' of some kind.

- If weights are allowed to grow *arbitrarily large*, this tends to encourage overfitting.
- Regularisation is used to constrain weights, in various ways. Also called Complexity control.

Regularisation makes training mininise the network's *weights*, as well as its errors.

- In scikit_learn, the parameter `C` defines how much attention to pay to weights. (Small values mean more regularisation.)
- There are several regularisation methods: L1 (Lasso), L2 (Ridge), Elastic Net.
- L1 regularisation also performs 'feature selection', which we'll get into later.

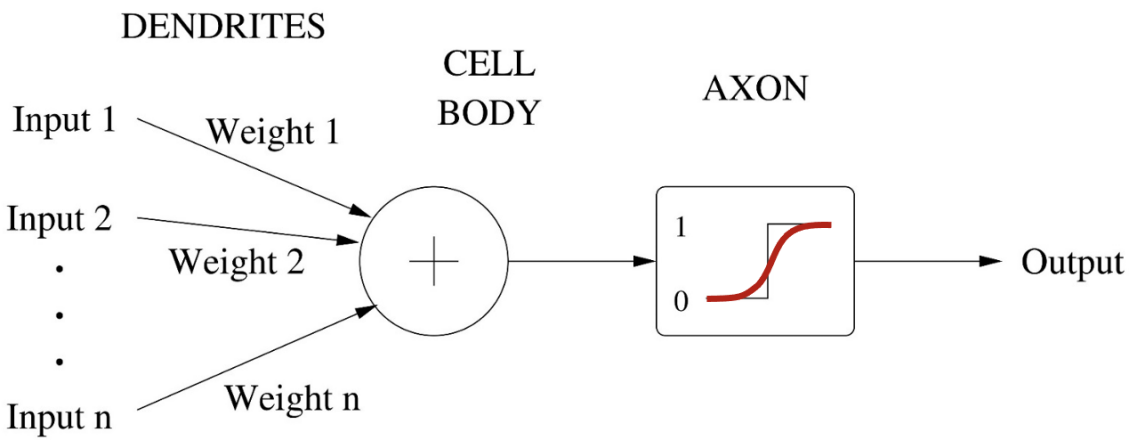
Perceptron code snippet

```
from sklearn.linear_model import Perceptron
clf = Perceptron(tol=1e-3, random_state=0)
#Training will stop when loss > previous_loss - tol.
clf.fit(ta_x, ta_y)
clf_pre_y = clf.predict(te_x[0:5])
print("Predictions:"+str(clf_pre_y))
print("Labels     :"+str(te_y[0:5]))
```

The Perceptron as a 'logistic regression' model

If you change the perceptron's threshold function to a smooth function (the logistic function), the perceptron learning rule approximates a probability model of the output class.

- The trained model estimates the conditional probability of output class 1, *given* a point in the input space.



A tad confusing: this is a *regression* model (with a continuous output), but it's still being used for discrete classification.

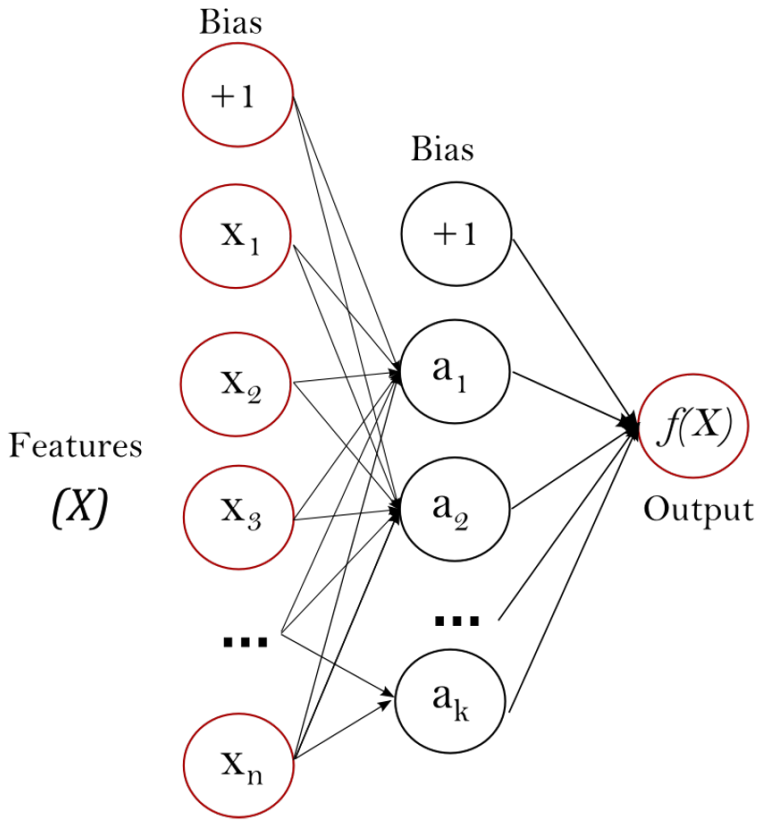
Logistic regression code snippet

```
from sklearn.linear_model import LogisticRegression

# Create instance of Logistic Regression Classifier
logreg = LogisticRegression(C=0.5)

# Fit some data
logreg.fit(ta_x, ta_y)
```

Classifer 3: The Multilayer perceptron

A Multilayer perceptron (MLP) is a neural network with many perceptrons, arranged in layers.

- Here's a simple MLP, with just one 'hidden layer' of perceptron units.



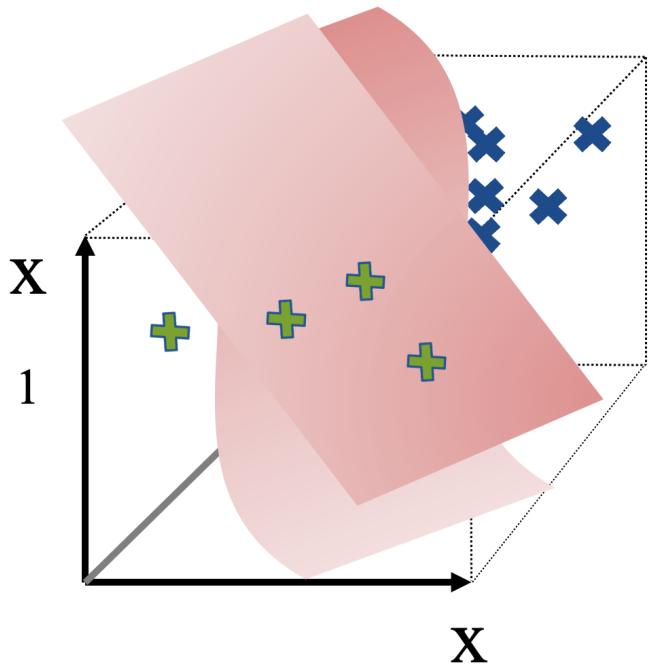
Classifer 3: The Multilayer perceptron

A Multilayer perceptron (MLP) is a neural network with many perceptrons, arranged in layers.

- Here's a simple MLP, with just one 'hidden layer' of perceptron units.

MLPs can learn more complex functions than perceptrons.

- Perceptrons can only learn flat decision boundaries.
- MLPs can learn curved ones.



MLP code snippet

```
from sklearn.neural_network import MLPClassifier
mlp_clf = MLPClassifier(random_state=0, max_iter=100,
alpha=0.1, activation="logistic")

mlp_clf.fit(ta_x, ta_y)
mlp_clf.predict(te_x[0:5])
```

Classifer 4: The Naive Bayes classifier

Bayesian models are *probability models*.

- They estimate conditional probabilities, of the form $P(Y|X)$. What's the probability of output Y, given inputs X?

A basic principle is to estimate conditional probabilities by taking each possible input in the training set, and *counting* the frequency of each possible output.

- E.g. estimate *P(restaurant_full | sunday)*.

But if there are many input features, there are too many 'possible inputs'.

- In a spam detector, we're estimating $P(spam|word_1 \ldots word\_n) \ldots$

Classifer 4: The Naive Bayes classifier

Bayesian models let us re-express $P(Y|X)$ as $P(X|Y)$.

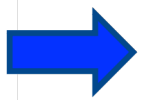
- So now we have $P(word_1 \ldots word\_n|spam)$.
- We can easily divide our training set into *spam* and *not_spam*.
- But we still have a counting problem: each possible word combination is ridiculously rare.

The 'naive' solution is to assume that words are *independent* of one another (given a spam category).

- Now we have $P(word_1|spam) \times \ldots \times P(word_n|spam)$.

More generally:

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots, x_n \mid y)}{P(x_1, \ldots, x_n)}$$



$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \ldots, x_n)}$$

Classifer 4: The Naive Bayes classifier

What we're doing is estimating the likelihood of the output, given some inputs, using information about which inputs are *typically associated* with each possible output.

- The classifier is 'naive', because it considers each input feature *individually*, ignoring *relationships between* input features.
- In practice, this 'naive' solution often works pretty well.

There are two varieties of Naive Bayes:

- Multinomial Naive Bayes is used if X_i are discrete
- Gaussian Naive Bayes is used if X_i are real (floats).

Multinomial Naive Bayes

In the spam case, we might often find one word $word_i$ that *doesn't occur* in *any* of the the 'spam' training items.

- If we're counting, we end up with $P(word_i|spam) = 0$.
- Zeros are deadly, becasue we multiply our probabilities together!

Multinomial Naive Bayes has to smooth counts, to avoid zeros.

- The simplest way is just to add some constant to all counts, to give us pseudo-counts.
- This is called additive smoothing, or Laplace smoothing.

Gaussian Naive Bayes

Gaussian Naive Bayes works with continuous inputs. (E.g. petal length.)

- In this model, we use input data to estimate a *Gaussian curve* for each input feature. (For each class.)
- $P(X_i|Y)$ is a Gaussian distribution, showing the probability of different values of X_i.

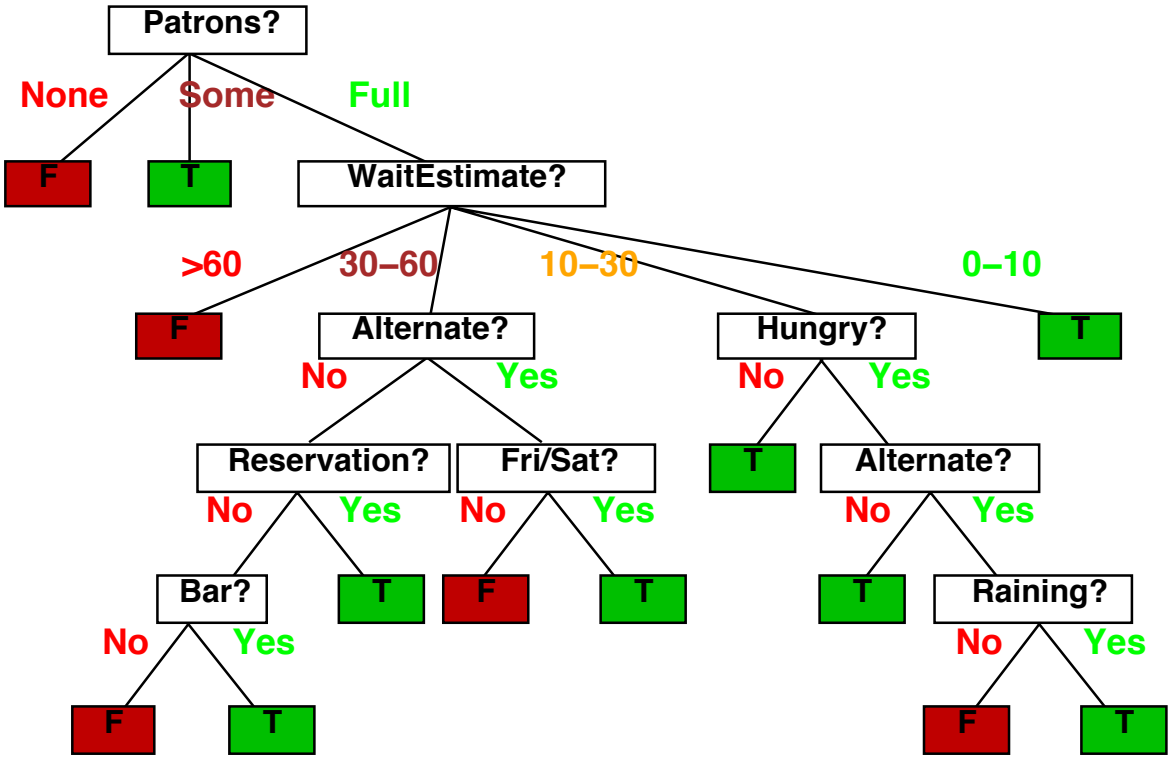
We need some smoothing for low counts here too, not for zero counts, but for complexity control.

- Low counts for X_i lead to badly estimated Gaussians, and *overfitting*.
- The scikit-learn parameter for this is called `var_smoothing`.

Classifier 5: Decision Trees

A decision tree decides on an output class by considering input features *one by one*.

- Each input feature is a node in the tree, with different branches for different output classes. (Terminal nodes are output classes.)
- Here's how Stuart Russell decides if he will wait for a restaurant.



A worked example for the restaurant scenario

Say we're observing Russell's behaviour, and compiling a dataset.

Item	Attributes										Class
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
X_5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*F*	*T*	*French*	*>60*	*F*
X_6	*F*	*T*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Italian*	*0–10*	*T*
X_7	*F*	*T*	*F*	*F*	*None*	*$*	*T*	*F*	*Burger*	*0–10*	*F*
X_8	*F*	*F*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Thai*	*0–10*	*T*
X_9	*F*	*T*	*T*	*F*	*Full*	*$*	*T*	*F*	*Burger*	*>60*	*F*
X_{10}	*T*	*T*	*T*	*T*	*Full*	*$$$*	*F*	*T*	*Italian*	*10–30*	*F*
X_{11}	*F*	*F*	*F*	*F*	*None*	*$*	*F*	*F*	*Thai*	*0–10*	*F*
X_{12}	*T*	*T*	*T*	*T*	*Full*	*$*	*F*	*F*	*Burger*	*30–60*	*T*

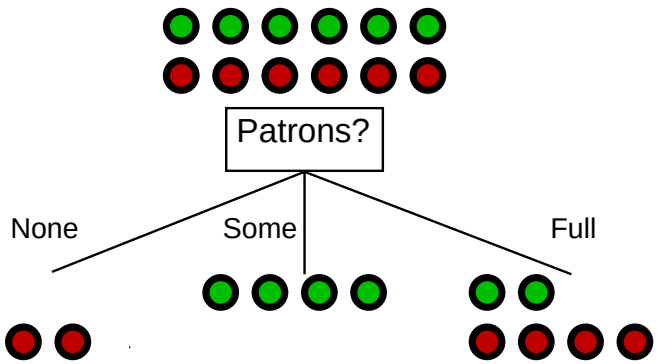
The decision tree algorithm iteratively picks the feature which carries *most information* about the output category.

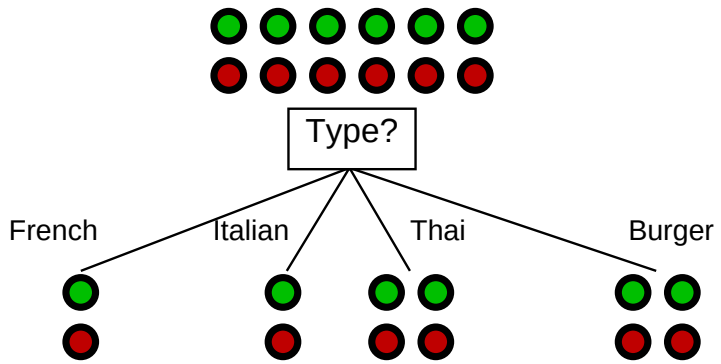
Choosing features for the decision tree

Which feature should we choose at each point?

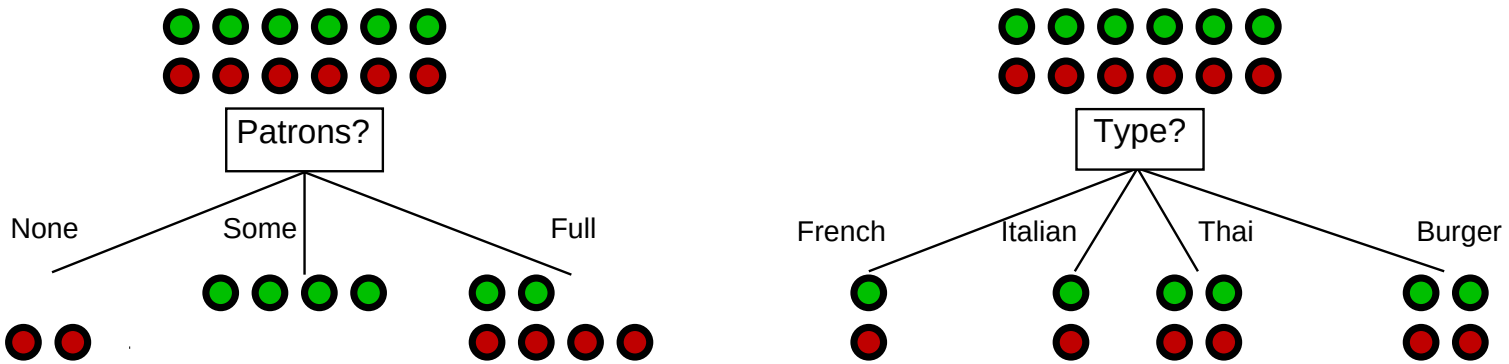
- Ideally, we want one which allows us to predict the output variable with 100% accuracy.
- But even if there isn't one, some features are still better than others.

Say we're starting off with all 12 training examples, and trying to decide between *Patrons* and *Type*. Which feature provides more information about *WillWait*?





Choosing features for the decision tree

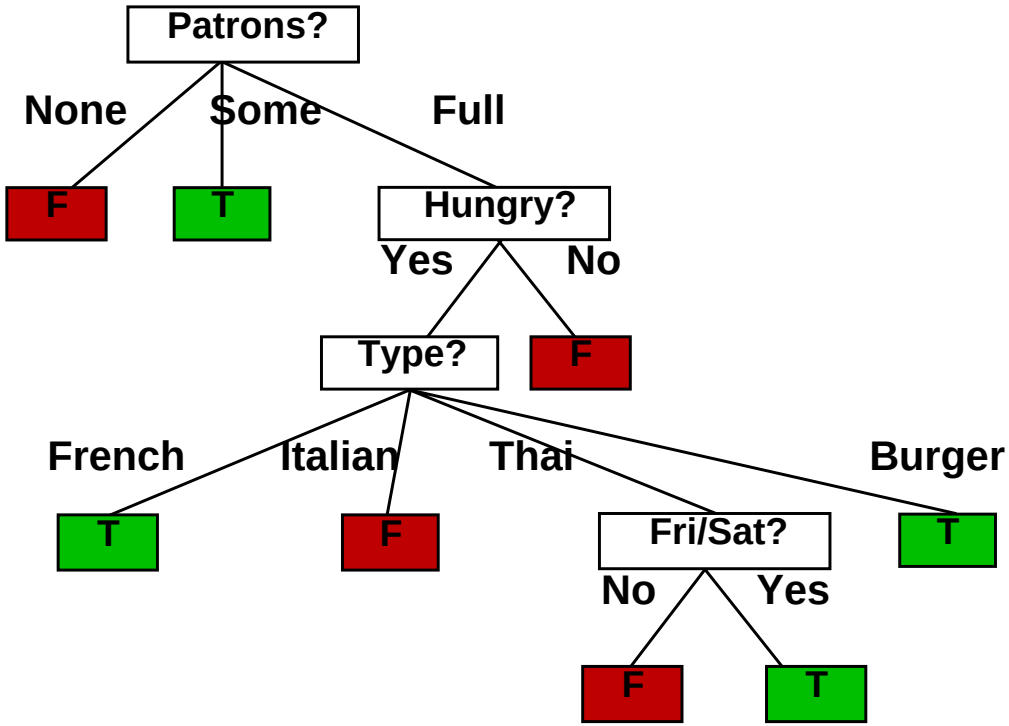


Here's what we do:

- Calculate how much 'information' we gain about *WillWait* if we learn the value of *Patrons*.
  This depends on the probabilities of *None*, *Some* and *Full*...
  But also on the probabilities of *WillWait* given each of these options.
- Then do the same for *Type*.
- And pick the feature which gives the highest information gain.

A worked example for the restaurant scenario

Here's the decision tree that algorithm creates, from that toy dataset.



The decision tree algorithm iteratively picks the feature which carries *most information* about the output category.

Varieties of decision tree in scikit-learn

If input features are continuous:

- The algorithm must identify a *range* of values for each branch.

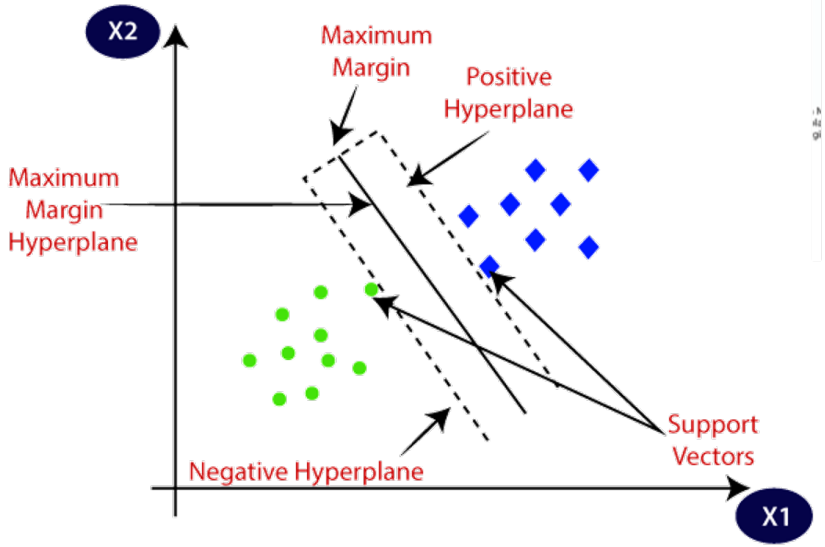
Complexity control:

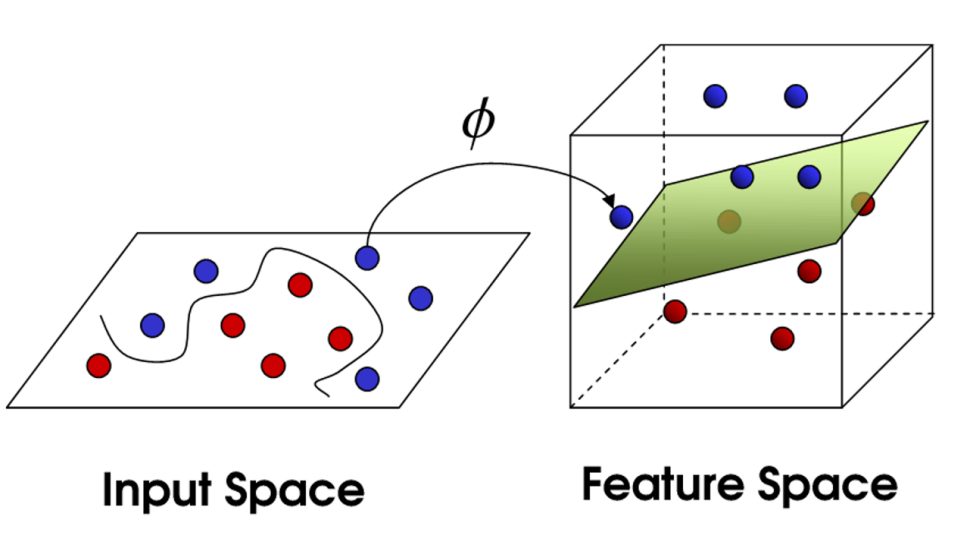
- Scikit-learn suggests the `max_depth` of the tree that is created.

Classifier 6: Support Vector Machines

A Support Vector Machine (SVM) finds the *linear boundary* (hyperplane) that keeps classes most cleanly separated.

- Often, there is *no* good linear boundary in input space.
- The SVM trick is to project input space into a *higher-dimensional space*. (With a kernel function.)
- Data that's not 'linearly separable' in input space, is often linearly separable in the higher-dimensional space.

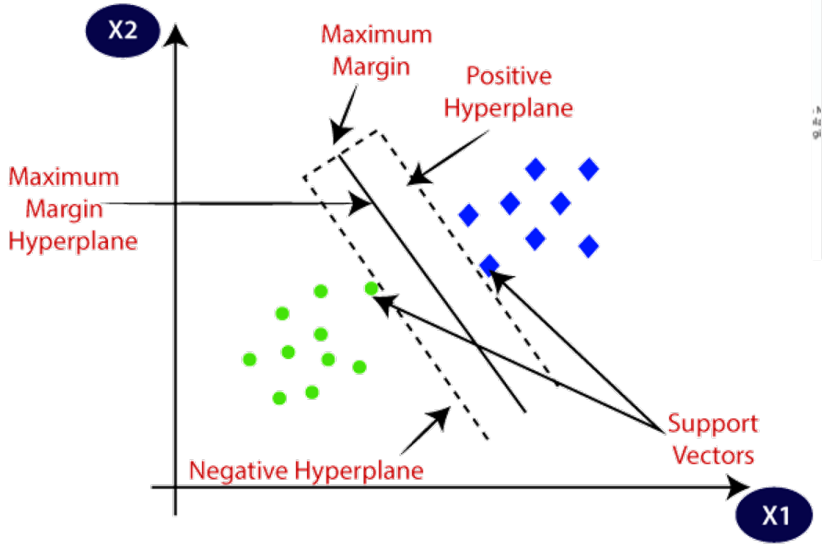


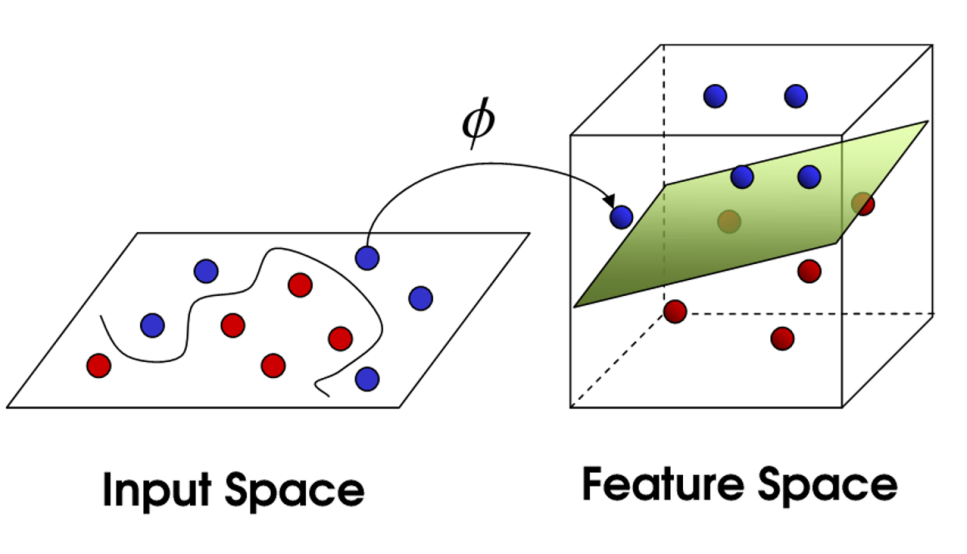


Classifier 6: Support Vector Machines

Key parameters for SVMs:

- One is the kernel function. (Different kernels are suitable for different domains.)
- Gamma controls similarity relations associated with the samples that define support vectors. (How close to these points are other points, after the kernel transformation?)
- C is a regularisation parameter, that adjusts the tradeoff between overfitting and underfitting.





Ensemble classifiers

Some of the most successful classifiers are ensembles of many different classifiers, working separately.

- Often, different classifiers have different strengths and weaknesses, so they can complement one another.
- Practically, ensemble classifiers often just work better.

We'll look at two ensemble methods, both based on decision trees.

- Random forests
- Gradient boosted decision trees

Ensemble classifier 1: Random forests

In the Random forest ensemble classifier:

- We train many independent decision trees, on different, randomly-selected *samples* from the training set.
- Then you make an answer based on responses from all these decision trees.

The guiding principle (I think) is the 'wisdom of the crowd':

- If you ask people to guess how many smarties are in a jar, the answers get better the more people you ask.

Ensemble classifier 2: Gradient-boosted decision trees

In an ensemble classifier, boosting involves training a sequence of classifiers, where each successive classifier is trained to *eliminate the errors* of the previous ones.

In a gradient-boosted ensemble, each subsequent classifier is trained to *correct* the errors of previous ones.

- If Classifier C_1 gets error E_1 in its prediction on input X_1, Classifier C_2 is trained to produce E_1 as its output for X_1.
- Then C_2's output can be *added* to C_1's output, to correct it.

This method is called 'gradient boosting', because each subsequent classifier is like a new step in following the 'gradient' of the ensemble's error, to minimise it.

- A gradient-boosted decision tree just does this boosting with decision trees.