Lecture 1

Machine Learning

what is machine learning?

We define **Intelligence** as "the ability to apply knowledge in order to perform better in an environment." We define **artificial intelligence** as the study and construction of agent programs that perform well in a given environment, for a given agent architecture. An agent is **learning** if it improves its performance on future tasks after making observations about the world.

Machine learning, then, is about making computers modify or adapt their actions (whether these actions are making predictions, or controlling a robot) so that these actions get more accurate, where accuracy is measured by how well the chosen actions reflect the correct ones.

Today we have enteredin the era of big data where continuosly data is produced by millions of computers, smartphones, sensors etc. and are stored on huge data centers. This deluge of data calls for automated methods of data analysis, which is what machine learning provides. From this perspective, we define machine learning as a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty.

Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed.— Arthur L. Samuel, AI pioneer, 1959

Why we want machine to learn?

First, the designers cannot anticipate all possible situations that the agent might find itself in. For example, a robot designed to navigate mazes must learn the layout of each new maze it encounters. Second, the designers cannot anticipate all changes over time; a program designed to predict tomorrow's stock market prices must learn to adapt when conditions change from boom to bust. Third, sometimes human programmers have no idea how to program a solution themselves. For example,

most people are good at recognizing the faces of family members, but even the best programmers are unable to program a computer to accomplish that task, except by using learning algorithms

Example of machine learning application?

Classifiication machine learning algorithm?

- **Supervised Learning:** In supervised learning the machine observes some example input–output pairs and learns a function that maps from input to output. A training set of examples with the correct responses (targets) is provided and, based on this training set, the algorithm generalises to respond correctly to all possible inputs. This is also called learning from **exemplars**. e.g driving under teacher.

- **Unsupervised Learning:** Correct responses are not provided, but instead the algorithm tries to identify similarities between the inputs so that inputs that have something in common are **categorised** together. The statistical approach to unsupervised learning is known as **density estimation**. e.g a taxi agent might gradually develop a concept of "good traffic days" and "bad traffic days"

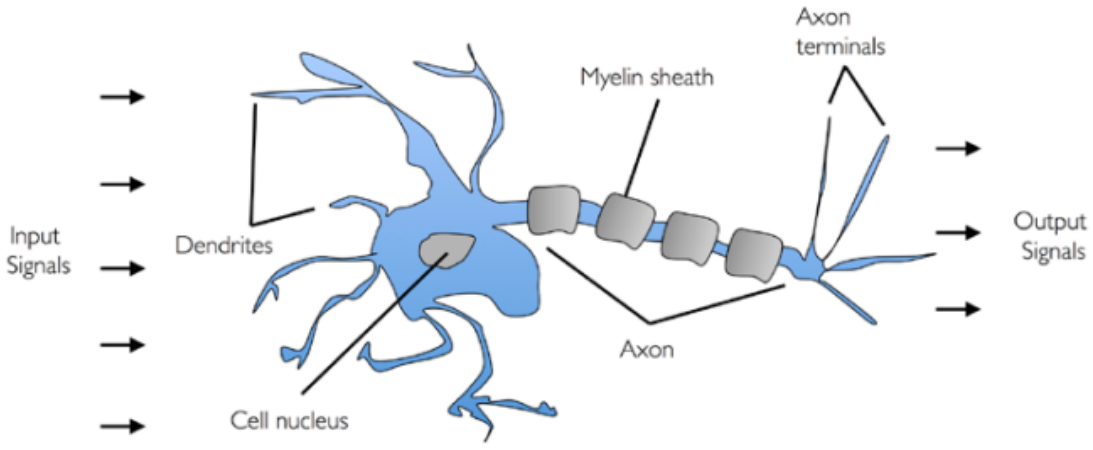
- **reinforcement learning:** the agent learns from a series of reinforcements—rewards or punishments. The algorithm gets told when the answer is wrong, but does not get told how to correct it. It has to explore and try out different possibilities until it works out how to get the answer right. Reinforcement learning is sometime called learning with a **critic**. e.g lack of a tip means bad driving.

- **semi-supervised learning:** Here, the machine is given a large dataset, in which only a few data points are labelled. The algorithm will use clustering techniques (unsupervised learning) to identify groups within the given dataset and use the few labelled data points within each group to provide labels to other data points in the same cluster/group. Noise and lack of labels create

a continuum between supervised and unsupervised learning. e.g identify cats and dogs with few labelled images.

Neural Network and Perceptron Learning

These are nerve cells called **neurons.** There are lots of them (100 billion = 1011 is the figure that is often given) and they come in lots of different types, depending upon their particular task. However, their general operation is similar in all cases: **transmitter chemicals** within the fluid of the brain raise or lower the electrical potential inside the body of the neuron. If this membrane potential reaches some threshold, the neuron **spikes or fires**, and a pulse of fixed strength and duration is sent down the **axon**. The axons divide (arborise) into connections to many other neurons, connecting to each of these neurons in a **synapse**. Each neuron is typically connected to thousands of other neurons, so that it is estimated that there are about 100 trillion (= 1014) synapses within the brain. After firing, the neuron must wait for some time to recover its energy (the refractory period) before it can fire again.



Each neuron can be viewed as a separate processor, performing a very simple computation: deciding whether or not to fire. This makes the brain a massively parallel computer made up of 1011 processing elements

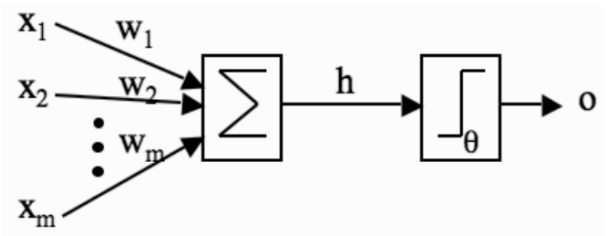
Hebb's Rule

So how does learning occur in the brain? The principal concept is **plasticity:** modifying the strength of synaptic connections between neurons, and creating new connections.

Hebb's rule says that the changes in the strength of synaptic connections are proportional to the correlation in the firing of the two connecting neurons. So if two neurons consistently fire simultaneously, then any connection between them will change in strength, becoming stronger. However, if the two neurons never fire simultaneously, the connection between them will die away.

McCulloch and Pitts Neuron's Model

Trying to understand how the biological brain works, in order to design artificial intelligence (AI), Warren McCulloch and Walter Pitts published the first concept of a simplified brain cell, the so-called **McCulloch-Pitts (MCP) neuron**, in 1943.



McCulloch and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, they are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

Only a few years later, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model (*The Perceptron: A Perceiving and Recognizing Automaton, F. Rosenblatt, Cornell Aeronautical Laboratory, 1957*). With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that would then be multiplied with the input features in order to make the decision of whether a neuron fires (transmits a signal) or not.

Rosenblatt's Perceptron Learning Rule

- **Initialisation**
  - set all of the weights w_{ij} to small (positive and negative) random numbers
- **Training**
  - for T iterations or until all the outputs are correct:
    - for each input vector:
      - compute the activation of each neuron j using activation function g:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } \sum_{i=0}^{m} w_{ij}x_i > 0 \\ 0 & \text{if } \sum_{i=0}^{m} w_{ij}x_i \leq 0 \end{cases}$$

      - update each of the weights individually using:

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i$$

- **Recall**
  - compute the activation of each neuron j using:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } w_{ij}x_i > 0 \\ 0 & \text{if } w_{ij}x_i \leq 0 \end{cases}$$

Machine Learning Process

1. Data Collection and Preparation
2. Feature Selection
3. Algorithm Choice
4. Parameter and Model Selection
5. Training
6. Evaluation

Lecture 2

Supervised Learning

The task of supervised learning is this:

Given a training set of N example input–output pairs

$$(x_1, y_1), (x_2, y_2), ...(x_N, y_N)$$

, where each y_j was generated by an unknown function $y = f(x)$, discover a function h that approximates the true function f.

Here x and y can be any value; they need not be numbers. The function h is a **hypothesis**. Learning is a search through the space of possible hypotheses for one that will perform well, even on new examples beyond the training set. To measure the accuracy of a hypothesis we give it a test set of examples that are distinct from the training set. We say a hypothesis generalizes well if it correctly predicts the value of y for novel examples. Sometimes the function f is stochastic—it is not strictly a function of x, and what we have to learn is a conditional probability distribution, $\mathbf{P}(Y|x)$.

When the output y is one of a finite set of values (such as sunny, cloudy or rainy),the learning problem is called classification, and is called Boolean or binary classification if there are only two values. When y is a number (such as tomorrow's temperature), the learning problem is called regression. (Technically, solving a regression problem is finding a conditional expectation or average value of y, because the probability that we have found exactly the right real-valued number for y is 0.)

Classification

Here the goal is to learn a mapping from inputs x to outputs y, where $y \epsilon \{1, ..., C\}$, with C being the number of classes. If $C = 2$, this is called **binary classification** (in which case we often assume $y \epsilon \{0, 1\}$); if $C > 2$, this is called **multiclass classification**. If the class labels are not mutually exclusive (e.g., somebody may be classified as tall and strong), we call it **multi-label classification**, but this is

best viewed as predicting multiple related binary class labels (a so-called **multiple output model**).

One way to formalize the problem is as function approximation. We assume $y = f(x)$ for some unknown function f, and the goal of learning is to estimate the function f given a labeled training set, and then to make predictions using $\hat{y} = \hat{f}(x)$. Our main goal is to make predictions on novel inputs, meaning ones that we have not seen before (this is called **generalization**), since predicting the response on the training set is easy (we can just look up the answer)

Real-world applications:-

Document classification and email spam filtering: In document classification, the goal is to classify a document, such as a web page or email message, into one of C classes, that is, to compute $p(y = c|x, D)$, where x is some representation of the text. A special case of this is email spam filtering, where the classes are spam $y = 1$ or ham $y = 0$.

Classifying flowers: The goal is to learn to distinguish three different kinds of iris flower, called setosa, versicolor and virginica. Fortunately, rather than working directly with images, a botanist has already extracted 4 useful features or characteristics: sepal length and width, and petal length and width. (Such feature extraction is an important, but difficult, task. Most machine learning methods use features chosen by some human.

Image classification and handwriting recognition We might want to classify the image as a whole, e.g., is it an indoors or outdoors scene? is it a horizontal or vertical photo? does it contain a dog or not? This is called **image classification**. In the special case that the images consist of isolated handwritten letters and digits, for example, in a postal or ZIP code on a letter, we can use classification to perform **handwriting recognition**.

Face detection and recognition A harder problem is to find objects within an image; this is called **object detection** or **object localization**. An important special case of this is face detection. One approach to this problem is to divide

the image into many small overlapping patches at different locations, scales and orientations, and to classify each such patch based on whether it contains face-like texture or not. This is called a **sliding window detector**. The system then returns those locations where the probability of face is sufficiently high. Such face detection systems are built-in to most modern digital cameras; Having found the faces, one can then proceed to perform face recognition, which means estimating the identity of the person . In this case, the number of class labels might be very large. Also, the features one should use are likely to be different than in the face detection problem: for recognition, subtle differences between faces such as hairstyle may be important for determining identity, but for detection, it is important to be invariant to such details, and to just focus on the differences between faces and non-faces

Rigression

Regression is just like classification except the response variable is continuous.

Here are some examples of real-world regression problems.

- Predict tomorrow's stock market price given current market conditions and other possible side information.
- Predict the age of a viewer watching a given video on YouTube.
- Predict the location in 3d space of a robot arm end effector, given control signals (torques) sent to its various motors.
- Predict the amount of prostate specific antigen (PSA) in the body as a function of a number of different clinical measurements.
- Predict the temperature at any location inside a building using weather data, time, door sensors, etc.

Linear Regression Model

One of the most widely used models for regression is known as **linear regression**. This asserts that the response is a linear function of the inputs. This can be written as follows:

$$y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + \epsilon = \sum_{j=1}^{D} w_j x_j + \epsilon$$

where $\mathbf{w}^T$ represents the inner or **scalar product** between the input vector x and the model's **weight vector w**, and ϵ is the **residual error** between our linear predictions and the true response.

Univariate linear regression

A univariate linear function (a straight line) with input x and output y has the form $y = w_1 x + w_0$, where w_0 and w_1 are real-valued coefficients to be learned. We use the letter w because we think of the coefficients as **weights**; the value of y is changed by changing the relative weight of one term or another. We'll define $\mathbf{w}$ to be the vector $[w_0, w_1]$, and define

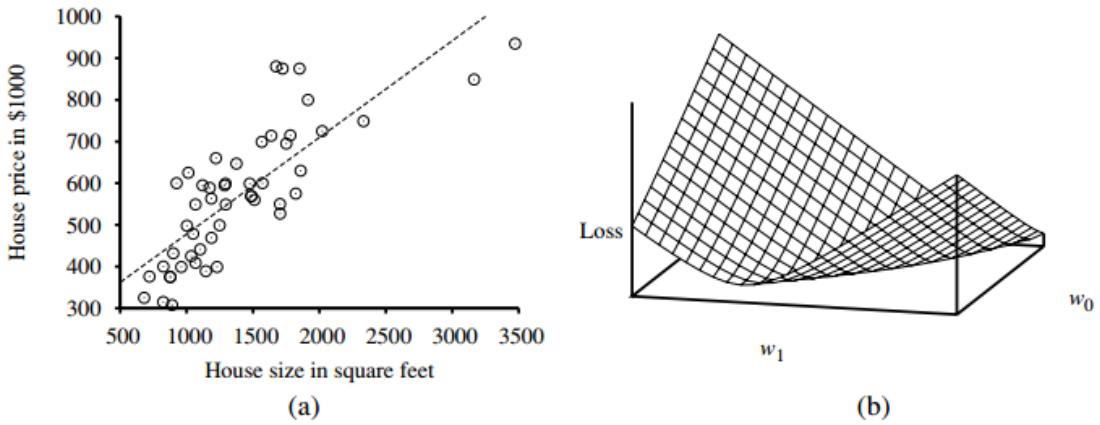
$$h_{\mathbf{w}}(x) = w_1 x + w_0$$

Figure shows an example of a training set of n points in the x, y plane, each point representing the size in square feet and the price of a house offered for sale. The task of finding the $h_{\mathbf{w}}$ that best fits these data is called **linear regression**. To fit a line to the data, all we have to do is find the values of the weights $[w_0, w_1]$ that minimize the empirical loss. It is traditional (going back to Gauss) to use the squared loss function, L_2, summed over all the training examples:

$$\text{Loss}\left(h_{\mathbf{w}}\right) = \sum_{j=1}^{N} L_2\left(y_j, h_{\mathbf{w}}\left(x_j\right)\right) = \sum_{j=1}^{N}\left(y_j - h_{\mathbf{w}}\left(x_j\right)\right)^2 = \sum_{j=1}^{N}\left(y_j - \left(w_1 x_j + w_0\right)\right)^2$$

We would like to find $\mathbf{w}^* = \text{argmin}_{\mathbf{w}}\,\text{Loss}\left(h_{\mathbf{w}}\right)$ The sum $\sum_{j=1}^{N}\left(y_j - \left(w_1 x_j + w_0\right)\right)^2$ is minimized when its partial derivatives with respect to w_0 and w_1 are zero:

$$\frac{\partial}{\partial w_0}\sum_{j=1}^{N}\left(y_j - \left(w_1 x_j + w_0\right)\right)^2 = 0 \text{ and } \frac{\partial}{\partial w_1}\sum_{j=1}^{N}\left(y_j - \left(w_1 x_j + w_0\right)\right)^2 = 0$$



These equations have a unique solution:

$$w_1 = \frac{N\left(\sum x_j y_j\right) - \left(\sum x_j\right)\left(\sum y_j\right)}{N\left(\sum x_j^2\right) - \left(\sum x_j\right)^2}; w_0 = \left(\sum y_j - w_1\left(\sum x_j\right)\right)/N$$

Many forms of learning involve adjusting weights to minimize a loss, so it helps to have a mental picture of what's going on in weight space—the space defined by all possible settings of the weights. For univariate linear regression, the weight space defined by w_0 and w_1 is two-dimensional, so we can graph the loss as a function of w_0 and w_1 in a 3D plot (see Figure). We see that the loss function is **convex**, this is true for every linear regression problem with an $L2$ loss function, and implies that there are no local minima. In some sense that's the end of the story for linear models; if we need to fit lines to data, we apply Equation().

To go beyond linear models, we will need to face the fact that the equations defining minimum loss (as in Equation ()) will often have no closed-form solution. Instead, we will face a general optimization search problem in a continuous weight space. Such problems can be addressed by a hill-climbing algorithm that follows the gradient of the function to be optimized. In this case, because we are trying to minimize the loss, we will use **gradient descent**. We choose any starting point in weight space—here, a point in the (w_0, w_1) plane—and then move to a neighboring point that is downhill, repeating until we converge on the minimum possible loss:

w ← any point in the parameter space

loop until convergence **do**

for each w_i **in** w **do**

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w})$$

The parameter α, which we called the step size is usually called the **learning rate** when we are trying to minimize loss in a learning problem. It can be a fixed constant, or it can decay over time as the learning process proceeds.

For univariate regression, the loss function is a quadratic function, so the partial derivative will be a linear function. Let's first work out the partial derivatives—the slopes—in the simplified case of only one training example, (x, y):

$$\begin{aligned}
\frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w}) &= \frac{\partial}{\partial w_i} \left(y - h_w(x)\right)^2 \\
&= 2\left(y - h_{\mathbf{w}}(x)\right) \times \frac{\partial}{\partial w_i} \left(y - h_{\mathbf{w}}(x)\right) \\
&= 2\left(y - h_{\mathbf{w}}(x)\right) \times \frac{\partial}{\partial w_i} \left(y - (w_1 x + w_0)\right)
\end{aligned}$$

applying this to both w_0 and w_1 we get:

$$\frac{\partial}{\partial w_0} \text{Loss}(\mathbf{w}) = -2\left(y - h_{\mathbf{w}}(x)\right); \quad \frac{\partial}{\partial w_1} \text{Loss}(\mathbf{w}) = -2\left(y - h_{\mathbf{w}}(x)\right) \times x$$

Then, plugging this back into Equation (), and folding the 2 into the unspecified learning rate α, we get the following learning rule for the weights:

$$w_0 \leftarrow w_0 + \alpha\left(y - h_{\mathbf{w}}(x)\right); \quad w_1 \leftarrow w_1 + \alpha\left(y - h_w(x)\right) \times x$$

These updates make intuitive sense: if $h_{\mathbf{w}}(x) > y$, i.e., the output of the hypothesis is too large, reduce w_0 a bit, and reduce w_1 if x was a positive input but increase w_1 if x was a negative input.

The preceding equations cover one training example. For N training examples, we want to minimize the sum of the individual losses for each example. The derivative of a sum is the sum of the derivatives, so we have:

$$w_0 \leftarrow w_0 + \alpha \sum_j \left(y_j - h_{\mathbf{w}}\left(x_j\right)\right); \quad w_1 \leftarrow w_1 + \alpha \sum_j \left(y_j - h_{\mathbf{w}}\left(x_j\right)\right) \times x_j$$

These updates constitute the **batch gradient descent** learning rule for univariate linear regression. Convergence to the unique global minimum is guaranteed (as long as we pick α small enough) but may be very slow: we have to cycle through all the training data for every step, and there may be many steps.

There is another possibility, called **stochastic gradient descent**, where we consider only a single training point at a time, taking a step after each one using Equation (). Stochastic gradient descent can be used in an online setting, where new data are coming in one at a time, or offline, where we cycle through the same data as many times as is necessary, taking a step after considering each single example. It is often faster than batch gradient descent. With a fixed learning rate α, however, it does not guarantee convergence; it can oscillate around the minimum without settling down. In some cases, as we see later, a schedule of decreasing learning rates (as in simulated annealing) does guarantee convergence.

Multivariate linear regression

We can easily extend to **multivariate linear regression** problems, in which each example x_j is an n-element vector. Our hypothesis space is the set of functions of the form

$$h_{sw}(\mathbf{x}_j) = w_0 + w_1 x_{j,1} + \cdots + w_n x_{j,n} = w_0 + \sum_i w_i x_{j,i}$$

The w_0 term, the intercept, stands out as different from the others. We can fix that by inventing a dummy input attribute, $x_{j,0}$, which is defined as always equal to 1. Then h is simply the dot product of the weights and the input vector (or equivalently, the matrix product of the transpose of the weights and the input vector):

$$h_{sw}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^\top \mathbf{x}_j = \sum_i w_i x_{j,i}$$

The best vector of weights, $\mathbf{w}^*$, minimizes squared-error loss over the examples:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j)$$

Multivariate linear regression is actually not much more complicated than the univariate case we just covered. Gradient descent will reach the (unique) minimum of the loss function; the update equation for each weight w_i is

$$w_i \leftarrow w_i + \alpha \sum_j x_{j,i} \left(y_j - h_{\mathbf{w}}(\mathbf{x}_j)\right)$$

It is also possible to solve analytically for the w that minimizes loss. Let y be the vector of outputs for the training examples, and X be the data matrix, i.e., the matrix of inputs with one n-dimensional example per row. Then the solution

$$\mathbf{w}^* = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}$$

minimizes the squared error.

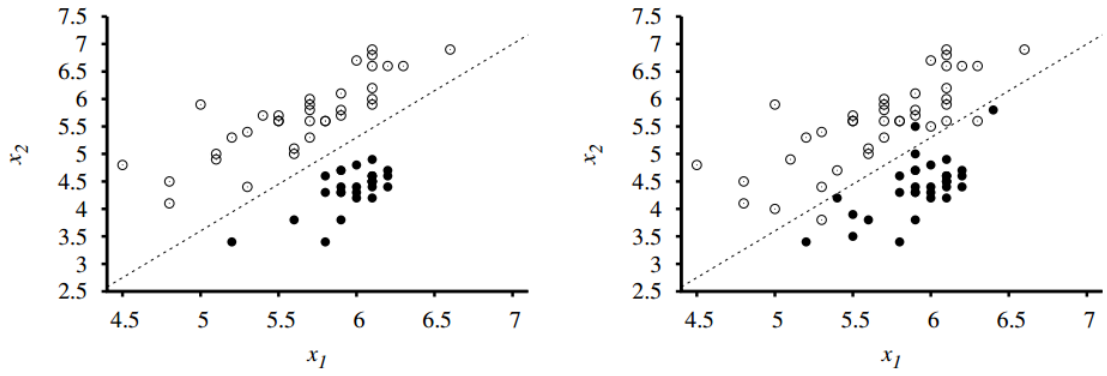
With univariate linear regression we didn't have to worry about **overfitting**. But with multivariate linear regression in high-dimensional spaces it is possible that some dimension that is actually irrelevant appears by chance to be useful, resulting in overfitting. Thus, it is common to use **regularization** on multivariate linear functions to avoid overfitting.

Lecture 3

Logistic Regression

Linear classifiers with a hard threshold

Linear functions can be used to do classification as well as regression. For example, Figure a shows data points of two classes: earthquakes (which are of interest to seismologists) and underground explosions (which are of interest to arms control experts). Each point is defined by two input values, x_1 and x_2, that refer to body and surface wave magnitudes computed from the seismic signal. Given these training data, the task of classification is to learn a hypothesis h that will take new (x_1, x_2) points and return either 0 for earthquakes or 1 for explosions.



A **decision boundary** is a line (or a surface, in higher dimensions) that separates the two classes. In Figure 18.15(a), the decision boundary is a straight line. A linear decision boundary is called a **linear separator** and data that admit such a separator are called **linearly separable**. The linear separator in this case is defined by

$$x_2 = 1.7x_1 - 4.9 \quad \text{or} \quad -4.9 + 1.7x_1 - x_2 = 0$$

The explosions, which we want to classify with value 1, are to the right of this line with higher values of x_1 and lower values of x_2, so they are points for which $-4.9 + 1.7x_1 - x_2 > 0$, while earthquakes have $-4.9 + 1.7x_1 - x_2 < 0$. Using the

convention of a dummy input $x_0 = 1$, we can write the classification hypothesis as

$$h_{\mathbf{w}}(\mathbf{x}) = 1 \text{ if } \mathbf{w} \cdot \mathbf{x} \geq 0 \text{ and } 0 \text{ otherwise.}$$

Alternatively, we can think of h as the result of passing the linear function $\mathbf{w} \cdot \mathbf{x}$ through a **threshold function**:

$$h_{\mathbf{w}}(\mathbf{x}) = \text{Threshold}(\mathbf{w} \cdot \mathbf{x}) \text{ where Threshold}(z) = 1 \text{ if } z \geq 0 \text{ and } 0 \text{ otherwise.}$$

The threshold function is shown in Figure(a).

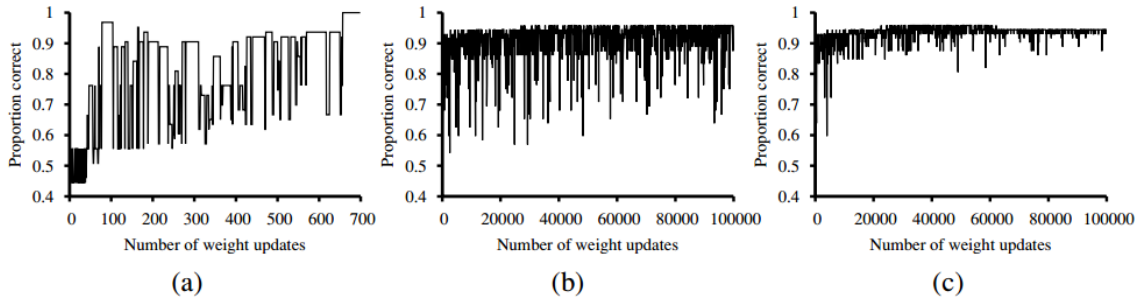
Now that the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ has a well-defined mathematical form, we can think about choosing the weights w to minimize the loss. we did this both in closed form (by setting the gradient to zero and solving for the weights) and by gradient descent in weight space. Here, we cannot do either of those things because the gradient is zero almost everywhere in weight space except at those points where $w \cdot x = 0$, and at those points the gradient is undefined.

There is, however, a simple weight update rule that converges to a solution—that is, a linear separator that classifies the data perfectly–provided the data are linearly separable. For a single example $(\mathbf{x}, y)$, we have

$$w_i \leftarrow w_i + \alpha \left(y - h_{\mathbf{w}}(\mathbf{x})\right) \times x_i$$

which is essentially identical to the Equation (18.6), the update rule for linear regression! This rule is called the **perceptron learning rule**. Because we are considering a 0/1 classification problem, however, the behavior is somewhat different. Both the true value y and the hypothesis output $h_{\mathbf{w}}(\mathbf{x})$ are either 0 or 1, so there are three possibilities:

- If the output is correct, i.e., $y = h_{\mathbf{w}}(\mathbf{x})$, then the weights are not changed.
- If y is 1 but $h_{\mathbf{w}}(\mathbf{x})$ is 0, then w_i is increased when the corresponding input x_i is positive and decreased when xi is negative. This makes sense, because we want to make w · x bigger so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 1.
- If y is 0 but $h_{\mathbf{w}}(\mathbf{x})$ is 1, then w_i is decreased when the corresponding input x_i is positive and increased when xi is negative. This makes sense, because we want to make w · x smaller so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 0.



Typically the learning rule is applied one example at a time, choosing examples at random (as in stochastic gradient descent). Figure(a) shows a **training curve** for this learning rule applied to the earthquake/explosion data shown in (a). A training curve measures the classifier performance on a fixed training set as the learning process proceeds on that same training set. The curve shows the update rule converging to a zero-error linear separator. The "convergence" process isn't exactly pretty, but it always works.

Linear classification with logistic regression

We have seen that passing the output of a linear function through the threshold function creates a linear classifier; yet the hard nature of the threshold causes some problems: the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ is not differentiable and is in fact a discontinuous function of its inputs and its weights; this makes learning with the perceptron rule a very unpredictable adventure. Furthermore, the linear classifier always announces a completely confident prediction of 1 or 0, even for examples that are very close to the boundary; in many situations, we really need more gradated predictions.

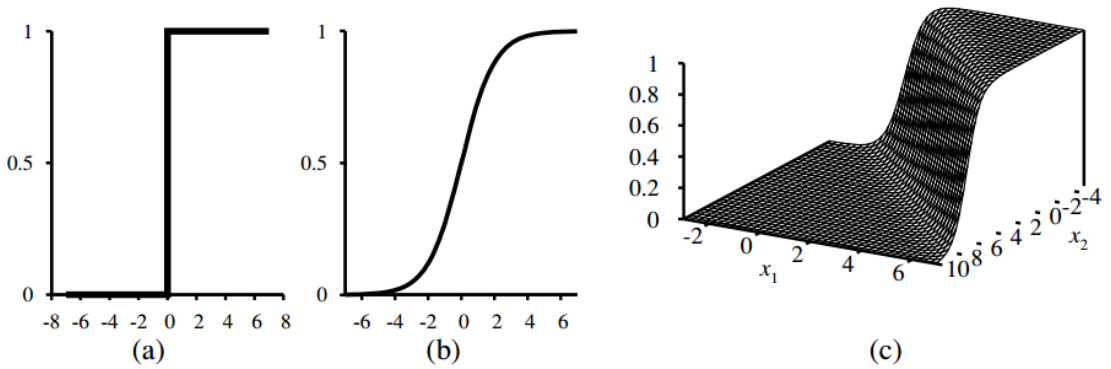
All of these issues can be resolved to a large extent by softening the threshold function— approximating the hard threshold with a continuous, differentiable function.we saw two functions that look like soft thresholds: the integral of the standard normal distribution (used for the probit model) and the logistic function (used for the logit model). Although the two functions are very similar in shape, the logistic function

$$\text{Logistic}(z) = \frac{1}{1 + e^{-z}}$$

has more convenient mathematical properties. The function is shown in Figure(b).

With the logistic function replacing the threshold function, we now have

$$h_{\mathbf{w}}(\mathbf{x}) = \text{Logistic}(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$



An example of such a hypothesis for the two-input earthquake/explosion problem is shown in Figure(c). Notice that the output, being a number between 0 and 1, can be interpreted as a probability of belonging to the class labeled 1. The hypothesis forms a soft boundary in the input space and gives a probability of 0.5 for any input at the center of the boundary region, and approaches 0 or 1 as we move away from the boundary.

The process of fitting the weights of this model to minimize loss on a data set is called **logistic regression**. There is no easy closed-form solution to find the optimal value of w with this model, but the gradient descent computation is straightforward. Because our hypotheses no longer output just 0 or 1, we will use the L2 loss function;

For a single example $(\mathbf{x}, y)$, the derivation of the gradient is the same as for linear regression up to the point where the actual form of h is inserted. (For this derivation, we will need the chain rule:$\partial g(f(x))/\partial x = g'(f(x))\partial f(x)/\partial x$. We have

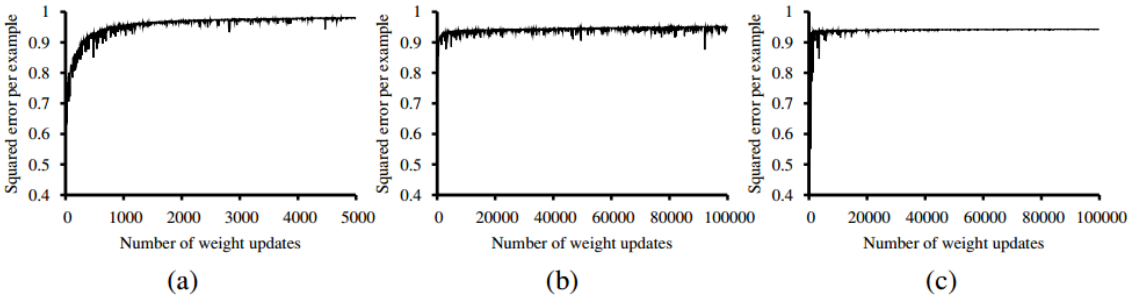
$$\begin{aligned}
\frac{\partial}{\partial w_i} \text{loss}(\mathbf{w}) &= \frac{\partial}{\partial w_i} \left(y - h_{\mathbf{w}}(\mathbf{x})\right)^2 \\
&= 2\left(y - h_{\mathbf{w}}(\mathbf{x})\right) \times \frac{\partial}{\partial w_i} \left(y - h_{\mathbf{w}}(\mathbf{x})\right) \\
&= -2\left(y - h_{\mathbf{w}}(\mathbf{x})\right) \times g'(\mathbf{w} \cdot \mathbf{x}) \times \frac{\partial}{\partial w_i} \mathbf{w} \cdot \mathbf{x} \\
&= -2\left(y - h_{\mathbf{w}}(\mathbf{x})\right) \times g'(\mathbf{w} \cdot \mathbf{x}) \times x_i
\end{aligned}$$

The derivative g' of the logistic function satisfies $g'(z) = g(z)(1 - g(z))$, so we have

$$g'(\mathbf{w} \cdot \mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x})(1 - g(\mathbf{w} \cdot \mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})\left(1 - h_{\mathbf{w}}(\mathbf{x})\right)$$

so the weight update for minimizing the loss is

$$w_i \leftarrow w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})\,(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i$$



Repeating the experiments with logistic regression instead of the linear threshold classifier, we obtain the results shown in Figure. In (a), the linearly separable case, logistic regression is somewhat slower to converge, but behaves much more predictably. In (b) and (c), where the data are noisy and nonseparable, logistic regression converges far more quickly and reliably. These advantages tend to carry over into real-world applications and logistic regression has become one of the most popular classification techniques for problems in medicine, marketing and survey analysis, credit scoring, public health, and other applications.For such learning tasks, the naive Bayes classifier is among the most effective algorithms known.

Lecture 4

Naive Bayes classifier

Bayesian reasoning provides a probabilistic approach to inference. It is based on the assumption that the quantities of interest are governed by probability distributions and that optimal decisions can be made by reasoning about these probabilities together with observed data. It is important to machine learning because it provides a quantitative approach to weighing the evidence supporting alternative hypotheses.

In machine learning we are often interested in determining the best hypothesis from some space, given the observed training data D. One way to specify what we mean by the best hypothesis is to say that we demand the most probable hypothesis, given the data D plus any initial knowledge about the prior probabilities of the various hypotheses in H. Bayes theorem provides a direct method for calculating such probabilities. More precisely, Bayes theorem provides a way to calculate the probability of a hypothesis based on its prior probability, the probabilities of observing various data given the hypothesis, and the observed data itself.

To define Bayes theorem precisely, let us first introduce a little notation. We shall write PQi) to denote the initial probability that hypothesis h holds, before we have observed the training data. PQi) is often called the prior probability of h and may reflect any background knowledge we have about the chance that h is a correct hypothesis. If we have no such prior knowledge, then we might simply assign the same prior probability to each candidate hypothesis. Similarly, we will write P(D) to denote the prior probability that training data D will be observed (i.e., the probability of D given no knowledge about which hypothesis holds). Next, we will write $P(D|h)$ to denote the probability of observing data D given some world in which hypothesis h holds. More generally, we write $P(x|y)$ to denote the probability of x given y. In machine learning problems we are interested in the probability $P(i|D)$ that h holds given the observed training data D. $P(i|D)$ is called the posterior probability of ft, because it reflects our confidence that h holds after we have seen the training data D. Notice the posterior probability $P(i|D)$ reflects the influence of the training data D, in contrast to the prior probability , which is

independent of D.

Bayes theorem is the cornerstone of Bayesian learning methods because it provides a way to calculate the posterior probability $P(i|D)$, from the prior probability $P(i)$, together with$P(D)$ and $P(D|h)$.

Bayes Theorem

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

In many learning scenarios, the learner considers some set of candidate hypotheses H and is interested in finding the most probable hypothesis $h\epsilon H$ given the observed data D (or at least one of the maximally probable if there are several). Any such maximally probable hypothesis is called a maximum a posteriori (MAP) hypothesis. We can determine the MAP hypotheses by using Bayes theorem to calculate the posterior probability of each candidate hypothesis. More precisely, we will say that Hmap is a MAP hypothesis provided

$$\begin{aligned} h_{MAP} &\equiv \underset{h \in H}{\operatorname{argmax}} P(h|D) \\ &= \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)} \\ &= \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h) \end{aligned}$$

Notice in the final step above we dropped the term P(D) because it is a constant independent of h. In some cases, we will assume that every hypothesis in H is equally probable a priori (P(hi) = P(hj) for all ht and hj in //). In this case we can further simplify Equation (6.2) and need only consider the term $P(D|h)$ to find the most probable hypothesis. $P(D|h)$ is often called the likelihood of the data D given h, and any hypothesis that maximizes $P(D|h)$ is called a maximum likelihood (ML) hypothesis, hML.

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(h|D)$$

In order to make clear the connection to machine learning problems, we introduced Bayes theorem above by referring to the data D as training examples of some target function and referring to H as the space of candidate target functions. In fact, Bayes theorem is much more general than suggested by this discussion. It

can be applied equally well to any set H of mutually exclusive propositions whose probabilities sum to one (e.g., "the sky is blue," and "the sky is not blue"). In this chapter, we will at times consider cases where H is a hypothesis space containing possible target functions and the data D are training examples. At other times we will consider cases where H is some other set of mutually exclusive propositions, and D is some other kind of data.

- *Product rule*: probability $P(A \wedge B)$ of a conjunction of two events A and B

$$P(A \wedge B) = P(A|B)P(B) = P(B|A)P(A)$$

- *Sum rule*: probability of a disjunction of two events A and B

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

- *Bayes theorem*: the posterior probability $P(h|D)$ of h given D

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- *Theorem of total probability*: if events $A_1, \ldots, A_n$ are mutually exclusive with $\sum_{i=1}^{n} P(A_i) = 1$, then

$$P(B) = \sum_{i=1}^{n} P(B|A_i)P(A_i)$$

As one might intuitively expect, $P(h|D)$ increases with P(h) and with $P(D|h)$ according to Bayes theorem. It is also reasonable to see that $P(h|D)$ decreases as $P(D)$ increases, because the more probable it is that D will be observed independent of h, the less evidence D provides in support of h.

The naive Bayes classifier is a probabilistic classifier based on Bayes's theorem. Let A and B denote two random events, then

$$p(A|B) \cdot p(B) = p(B|A) \cdot p(A)$$

If the event A can be decomposed into the disjoint events $A_1, ..., A_c$, $p(A_i) > 0$, $i = 1, ..., c$, then

$$p\left(A_i|B\right) = \frac{p\left(A_i\right) \cdot p\left(B|A_i\right)}{\sum_{j=1}^{c} P\left(A_j\right) \cdot P\left(B|A_j\right)}$$

The relevant events considered in classification are "object belongs to class i", or more briefly, just "i", and "object has the feature vector x", or "x". Replacing A_i, A_j and B in by these events yields

$$p\left(i|x\right) = \frac{p\left(i\right) \cdot p\left(x|i\right)}{\sum_{j=1}^{c} P\left(j\right) \cdot P\left(x|j\right)}$$

If the p features in x are stochastically independent, then

$$p(x|i) = \prod_{k=1}^{p} p\left(x^{(k)}|i\right)$$

Inserting this into above yields the classification probabilities of the naive Bayes classifier.

$$p(i|x) = \frac{p(i) \cdot \prod_{k=1}^{p} p\left(x^{(k)}|i\right)}{\sum_{j=1}^{c} p(j) \cdot \prod_{k=1}^{p} p\left(x^{(k)}|j\right)}$$

Given a labeled feature data set $Z = \{(x_1, y_1), \ldots, (x_n, y_n)\} \subset \mathbb{R}^p \times \{1, \ldots, c\}$, the prior probabilities can be estimated from the observed frequencies: $p(i)$ is the relative frequency of class $y = 1$, $i = 1, ..., c$, and the probabilities $p(x^{(k)}|i)$ are the relative frequencies of feature $x^{(k)}$ in class $y = 1$,$i = 1, ..., c$,$k = 1, ..., p$. For a given feature vector x, Equation above yields the classification probabilities for each class. For a deterministic classification the class with the highest probability is selected.

The major advantages of the naive Bayes classifier are its efficiency, because the training data have to be evaluated only once (to compute the relative frequencies), and missing data can be simply ignored. The major disadvantages are that the features must be stochastically independent, which is usually not the case in real-world data, and that the features must be discrete. Continuous features may be used in the naive Bayes classifier after discretization.

Lecture 5

K nearest neighbor

The most basic instance-based method is the k-Nearest Neighbor algorithm.In contrast to learning methods that construct a general, explicit description of the target function when training examples are provided, instance-based learning methods simply store the training examples. Generalizing beyond these examples is postponed until a new instance must be classified. Each time a new query instance is encountered, its relationship to the previously stored examples is examined in order to assign a target function value for the new instance.

This algorithm assumes all instances correspond to points in the n-dimensional space. The nearest neighbors of an instance are defined in terms of the standard Euclidean distance. More precisely, let an arbitrary instance x be described by the feature vector

$$\langle a_1(x), a_2(x), \ldots a_n(x) \rangle$$

where $a_r(x)$ denotes the value of the rth attribute of instance x. Then the distance between two instances x_i, and x_j is defined to be $d(x_i, X_j)$, where

$$d\left(x_i, x_j\right) \equiv \sqrt{\sum_{r=1}^{n} \left(a_r\left(x_i\right) - a_r\left(x_j\right)\right)^2}$$

In nearest-neighbor learning the target function may be either discrete-valued or real-valued. Let us first consider learning discrete-valued target functions of the form $f : \mathbb{R}^n \rightarrow V$ where V is finite set $\{v_1, v_2, v_3 ...\}$ The k-Nearest Neighbor algorithm for approximating a discrete-valued target function is given as:

Training algorithm

- For each training example $< x, f(x) >$, add the example to the list *training examples*.

Classification algorithm

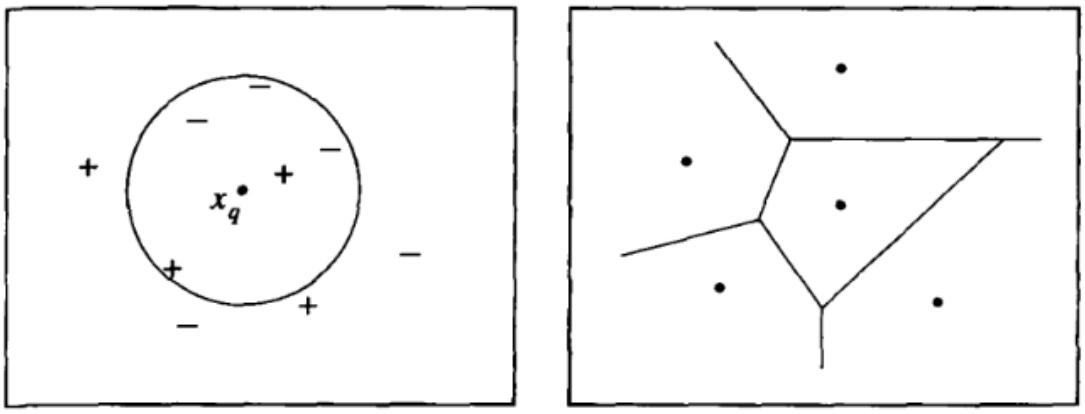
- Given a query instance xq to be classified,

- Let $x_1, ..., x_k$ denote the k instances from *training examples* that are nearest to x_q
- Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

As shown there, the value $\hat{f}(x_q)$returned by this algorithm as its estimate of $f(x)$ is just the most common value of / among the k training examples nearest to xq. If we choose k = 1, then the 1-Nearest Neighbor algorithm assigns to (fx_q) the value ... where ...is the training instance nearest to x_q. For larger values of k, the algorithm assigns the most common value among the k nearest training examples. Figure 8.1 illustrates the operation of the k-Nearest Neighbor algorithm for the case where the instances are points in a two-dimensional space and where the target function is boolean valued. The positive and negative training examples are shown by "+" and "—" respectively. A query point x_q is shown as well. Note the 1-Nearest Neighbor algorithm classifies xq as a positive example in this figure, whereas the 5-Nearest Neighbor algorithm classifies it as a negative example.



What is the nature of the hypothesis space H implicitly considered by the k-Nearest Neighbor algorithm? Note the k-Nearest Neighbor algorithm never forms an explicit general hypothesis regarding the target function . It simply computes the classification of each new query instance as needed. Nevertheless, we can still ask what the implicit general function is, or what classifications would be assigned

if we were to hold the training examples constant and query the algorithm with every possible instance in X. The diagram on the right side of Figure 8.1 shows the shape of this decision surface induced by 1-Nearest Neighbor over the entire instance space. The decision surface is a combination of convex polyhedra surrounding each of the training examples. For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the Voronoi diagram of the set of training examples. The Nearest Neighbor algorithm is easily adapted to approximating continuous-valued target functions. To accomplish this, we have die algorithm calculate the mean value of the k nearest training examples rather than calculate their most common value. More precisely, to approximate a real-valued target function.

$$\hat{f}\left(x_q\right) \leftarrow \frac{\sum_{i=1}^{k} f\left(x_i\right)}{k}$$

Distance-Weighted Nearest Neighbor Algorithm

One obvious refinement to the k-Nearest Neighbor algorithm is to weight die contribution of each of the k neighbors according to their distance to the query point xq, giving greater weight to closer neighbors. For example, in the algorithm , which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from xq.

$$\hat{f}\left(x_q\right) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta\left(v, f\left(x_i\right)\right)$$

where

$$w_i \equiv \frac{1}{d\left(x_q, x_i\right)^2}$$

where w_i is as defined in Equation (8.3). Note the denominator in Equation is a constant that normalizes the contributions of the various weights (e.g., it assures that if $f(x() = c$ for all training examples, then $f(x_q)$ cas well). Note all of the above variants of the /:-Nearest Neighbor algorithm consider only the k nearest neighbors to classify the query point. Once we add distance weighting, there is really no harm

in allowing all training examples to have an influence on the classification of the x_q, because very distant examples will have very little effect on $f(x_q)$. The only disadvantage of considering all examples is that our classifier will run more slowly. If all training examples are considered when classifying a new query instance, we call the algorithm a global method. If only the nearest training examples are considered, we call it a local method. When the rule in Equation is applied as a global method, using all training examples, it is known as **Shepard's method**

The distance-weighted k-Nearest Neighbor algorithm is a highly effective inductive inference method for many practical problems. It is robust to noisy training data and quite effective when it is provided a sufficiently large set of training data. Note that by taking the weighted average of the k neighbors nearest to the query point, it can smooth out the impact of isolated noisy training examples. What is the inductive bias of k-Nearest Neighbor? The basis for classifying new query points is easily understood based on the diagrams in Figure 8.1. The inductive bias corresponds to an assumption that the classification of an instance x_q will be most similar to the classification of other instances that are nearby in Euclidean distance. One practical issue in applying k-Nearest Neighbor algorithms is that the distance between instances is calculated based on all attributes of the instance (i.e., on all axes in the Euclidean space containing the instances). This lies in contrast to methods such as rule and decision tree learning systems that select only a subset of the instance attributes when forming the hypothesis. To see the effect of this policy, consider applying k-Nearest Neighbor to a problem in which each instance is described by 20 attributes, but where only 2 of these attributes are relevant to determining the classification for the particular target function. In this case, instances that have identical values for the 2 relevant attributes may nevertheless be distant from one another in the 20-dimensional instance space. As a result, the similarity metric used by k-Nearest Neighbor—depending on all 20 attributes—will be misleading. The distance between neighbors will be dominated by the large number of irrelevant attributes. This difficulty, which arises when many irrelevant attributes are present, is sometimes referred to as the curse of dimensionality. Nearest-neighbor approaches are especially sensitive to this problem.

Lecture 6

Support Vector Machine

Nonlinear decision boundaries can be handled by algorithms such as support vector machines (SVMs). Support vector machines are a supervised learning algorithm used for solving binary classification and regression problems. The main idea of support vector machines is to construct a hyperplane such that the margin of separation between the two classes is maximized. In this algorithm each of the data points is plotted as a data point in n-dimensional hyperspace. Then construct a hyperplane that maximizes the separation between two classes.

Clearly the class boundaries are not linearly separable. In such a case, SVM uses a technique where two-dimensional representation can be converted to higher dimension data. In this higher dimensional representation, the class boundaries become linearly separable, and SVM classifier can provide a boundary. For example, the data in is transposed to three-dimensional space, with parameters as x, y, and p2xy, and then the data in three dimensions is classified using SVM linear plane classifier. The linear plane is then transposed back to two dimensions to form the boundary as shown by the lines in Fig. 5.3. Section 5.4 will talk some more about SVMs.

The support vector machine or SVM framework is currently the most popular approach for "off-the-shelf" supervised learning: if you don't have any specialized prior knowledge about a domain, then the SVM is an excellent method to try first. There are three properties that make SVMs attractive:

1. SVMs construct a **maximum margin separator**—a decision boundary with the largest possible distance to example points. This helps them generalize well.

2. SVMs create a linear separating **hyperplane**, but they have the ability to embed the data into a higher-dimensional space, using the so-called kernel trick. Often, data that are not linearly separable in the original input space are easily separable in the higherdimensional space. The high-dimensional

linear separator is actually nonlinear in the original space. This means the hypothesis space is greatly expanded over methods that use strictly linear representations.

3. SVMs are a **nonparametric method**—they retain training examples and potentially need to store them all. On the other hand, in practice they often end up retaining only a small fraction of the number of examples—sometimes as few as a small constant times the number of dimensions. Thus SVMs combine the advantages of nonparametric and parametric models: they have the flexibility to represent complex functions, but they are resistant to overfitting.

You could say that SVMs are successful because of one key insight and one neat trick. We will cover each in turn. In , we have a binary classification problem with three candidate decision boundaries, each a linear separator. Each of them is consistent with all the examples, so from the point of view of 0/1 loss, each would be equally good. Logistic regression would find some separating line; the exact location of the line depends on all the example points. The key insight of SVMs is that some examples are more important than others, and that paying attention to them can lead to better generalization.

SVMs use the convention that class labels are +1 and -1, instead of the +1 and 0 we have been using so far. Also, where we put the intercept into the weight vector w (and a corresponding dummy 1 value into xj,0), SVMs do not do that; they keep the intercept as a separate parameter, b. With that in mind, the separator is defined as the set of points $x : w \cdot x + b = 0$. We could search the space of w and b with gradient descent to find the parameters that maximize the margin while correctly classifying all the examples.There is an alternative representation called the dual representation, in which the optimal solution is found by solving

$$\underset{\alpha}{\operatorname{argmax}} \sum_j \alpha_j - \frac{1}{2} \sum_{j,k} \alpha_j \alpha_k y_j y_k \left(\mathbf{x}_j \cdot \mathbf{x}_k\right)$$

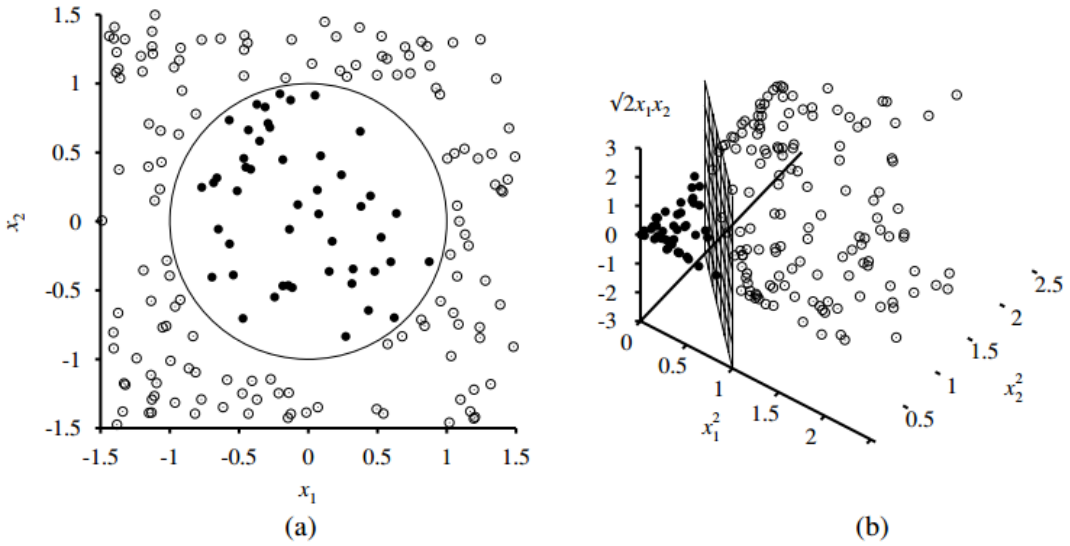
subject to the constraints $\alpha_j \geq 0$ and $\sum_j \alpha_j y_j = 0$ This is a **quadratic programming** optimization problem, for which there are good software packages. Once we have found the vectorα we can get back to w with the equation $\mathbf{w} = \sum_j \alpha_j \mathbf{x}_j$ or

we can stay in the dual representation. There are three important properties of Equation (18.13). First, the expression is convex; it has a single global maximum that can be found efficiently. Second, the data enter the expression only in the form of dot products of pairs of points. This second property is also true of the equation for the separator itself; once the optimal α_j have been calculated, it is

$$h(\mathbf{x}) = \text{sign}\left(\sum_j \alpha_j y_j \left(\mathbf{x} \cdot \mathbf{x}_j\right) - b\right)$$

A final important property is that the weights α_j associated with each data point are zero except for the support vectors—the points closest to the separator. (They are called "support" vectors because they "hold up" the separating plane.) Because there are usually many fewer support vectors than examples, SVMs gain some of the advantages of parametric models.

Support Vector Machine Algorithm



We've talked about SVMs in terms of two-class classification. You might be wondering how to use them for more classes, since we can't use the same methods as we have done to work out the current algorithm. In fact, you can't actually do it in a consistent way. The SVM only works for two classes. This might seem like a major problem, but with a little thought it is possible to find ways around the

- **Initialisation**
  - for the specified kernel, and kernel parameters, compute the kernel of distances between the datapoints
    - the main work here is the computation $\mathbf{K} = \mathbf{X}\mathbf{X}^T$
    - for the linear kernel, return $\mathbf{K}$, for the polynomial of degree d return $\frac{1}{\sigma}\mathbf{K}^d$
    - for the RBF kernel, compute $\mathbf{K} = \exp(-(\mathbf{x} - \mathbf{x}')^2/2\sigma^2)$
- **Training**
  - assemble the constraint set as matrices to solve:

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T t_i t_j \mathbf{K}\mathbf{x} + \mathbf{q}^T\mathbf{x} \text{ subject to } \mathbf{G}\mathbf{x} \leq \mathbf{h}, \mathbf{A}\mathbf{x} = b$$

  - pass these matrices to the solver
  - identify the support vectors as those that are within some specified distance of the closest point and dispose of the rest of the training data
  - compute b^* using equation (8.10)
- **Classification**
  - for the given test data $\mathbf{z}$, use the support vectors to classify the data for the relevant kernel using:
    - compute the inner product of the test data and the support vectors
    - perform the classification as $\sum_{i=1}^{n} \lambda_i t_i \mathbf{K}(\mathbf{x}_i, \mathbf{z}) + b^*$, returning either the label (hard classification) or the value (soft classification)

problem. For the problem of N-class classification, you train an SVM that learns to classify class one from all other classes, then another that classifies class two from all the others. So for N-classes, we have N SVMs. This still leaves one problem: how do we decide which of these SVMs is the one that recognises the particular input? The answer is just to choose the one that makes the strongest prediction, that is, the one where the basis vector input point is the furthest into the positive class region. It might not be clear how to work out which is the strongest prediction. The classifier examples in the code snippets return either the class label (as the sign of y) or the value of y, and this value of y is telling us how far away from the decision boundary it is, and clearly it will be negative if it is a misclassification. We can therefore use the maximum value of this soft boundary as the best classifier.

Lecture 7

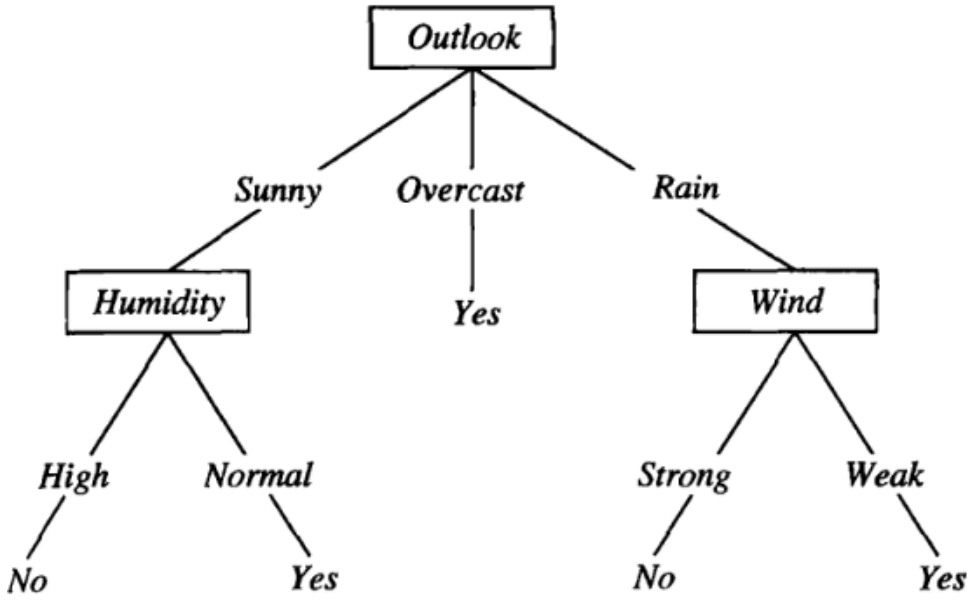
Decision Tree

All previously presented classifiers consider all p features at a time. This may be disadvantageous if not all features are necessary for a good classification, and if the assessment of the feature values causes a high effort. In medical diagnosis, for example, body temperature or blood pressure are measured for many patients because these features are assessed with low effort and yield significant information about the patient's health, whereas more expensive or time-consuming tests are only performed if needed to support a specific diagnosis. Depending on the already observed feature data, additional features may be ranked by importance, and only a subset of all possible features is used. This scheme leads to a *hierarchical structure* of the features, as opposed to the flat featurestructure of the classifiers presented so far. Hierarchical classifiers can be represented by decision trees.

Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some attribute of the instance, and each branch descendingfrom that node corresponds to one of the possible values for this attribute. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node.

This decision tree classifies Saturday mornings according to whether they are suitable for playing tennis.

Most algorithms that have been developed for learning decision trees are variations on a core algorithm that employs a top-down, greedy search through the space of possible decision trees. This approach is exemplified by the ID3 algorithm. ID3, learns decision trees by constructing them top- down, beginning with the question "which attribute should be tested at the root of the tree?" To answer this question, each instance attribute is evaluated using a statistical test to determine how well it alone classifies the training examples. The best attribute is selected and used as



the test at the root node of the tree. A descendant of the root node is then created for each possible value of this attribute, and the training examples are sorted to the appropriate descendant node (i.e., down the branch corresponding to the example's value for this attribute). The entire process is then repeated using the training examples associated with each descendant node to select the best attribute to test at that point in the tree. This forms a greedy search for an acceptable decision tree, in which the algorithm never backtracks to reconsider earlier choices.

Given entropy as a measure of the impurity in a collection of training examples,we can now define a measure of the effectiveness of an attribute in classifyingthe training data. The measure we will use, called information gain, is simply theexpected reduction in entropy caused by partitioning the examples according to this attribute. More precisely, the information gain, Gain(Sy A) of an attribute A, relative to a collection of examples S, is defined as where Values (A) is the set of all possible values for attribute A, and Sv is the subset of S for which attribute A has value v (i.e., $S_v = \{s \epsilon S = v\}$). Note the first term in Equation (3.4) is just the entropy of the original collection S, and the second term is the expected value of the entropy after S is partitioned using attribute A. The expected entropy described by this second term is simply the sum of the entropies of each subset Sv, weighted by the fraction of examples that belong to Sv. Gain(Sy A) is therefore the expected

1. input: $X = \{x_1, \ldots, x_n\} \subset \{1, \ldots, v_1\} \times \ldots \times \{1, \ldots, v_p\}$, $Y = \{y_1, \ldots, y_n\} \subset \{1, \ldots, c\}$
2. create root node W
3. ID3$(X, Y, W, \{1, \ldots, p\})$
4. output: tree with root W

procedure ID3(X, Y, N, I)

1. if I is empty or all Y are equal then terminate
2. compute information gain $g_i(X, Y)$ $\forall i \in I$
3. determine winner feature $j = \text{argmax}\{g_i(X, Y)\}$
4. partition X, Y into v_j disjoint subsets

$$X_i = \{x_k \in X \mid x_k^{(j)} = i\}, \quad Y_i = \{y_k \in Y \mid x_k^{(j)} = i\}, \quad i = 1, \ldots, v_j$$

5. for i with $X_i \neq \{\}, Y_i \neq \{\}$
   a. create new node N_i and append it to N
   b. ID3$(X_i, Y_i, N_i, I \backslash \{j\})$

reduction in entropy caused by knowing the value of attribute A. Put another way, Gain(Sy A) is the information provided about the target function value, given the value of some other attribute A. The value of Gain(S, A) is the number of bits saved when encoding the target value of an arbitrary member of S, by knowing the value of attribute A. For example, suppose 5 is a collection of training-example days described by attributes including Wind, which can have the values Weak or Strong. As before, assume S is a collection containing 14 examples, [9+, 5-]. Of these 14 examples, suppose 6 of the positive and 2 of the negative examples have Wind = Weak, and the remainder have Wind = Strong. The information gain due to sorting the original 14 examples by the attribute Wind may then be calculated as Information gain is precisely the measure used by ID3 to select the best attribute at each step in growing the tree. The use of information gain to evaluate the relevance of attributes is summarized in Figure 3.3. In this figure the information gain of two different attributes, Humidity and Wind, is computed in order to determine which is the better attribute for classifying the training examples shown in As with other inductive learning methods, ID3 can be characterized as searching a space of hypotheses for one that fits the training examples. The hypothesis space searched

by ID3 is the set of possible decision trees. ID3 performs a simple-to- complex, hill-climbing search through this hypothesis space, beginning with the empty tree, then considering progressively more elaborate hypotheses in search of a decision tree that correctly classifies the training data. The evaluation function

Issues In Decision Tree Learning

Practical issues in learning decision trees include determining how deeply to grow the decision tree, handling continuous attributes, choosing an appropriate attribute selection measure,) andling training data with missing attribute values, handling attributes with differing costs, and improving computational efficiency. Below we discuss each of these issues and extensions to the basic ID3 algorithm that address them.

Avoiding Overfitting the Data

We will say that a hypothesis overfits the training examples if some other hypothesis that fits the training examples less well actually performs better over the entire distribution of instances (i.e., including instances beyond the training set).

Definition: Given a hypothesis space H, a hypothesis $h \in H$ is said to overfit the training data if there exists some alternative hypothesis $h' \in H$y such that h has smaller error than h! over the training examples, but h! has a smaller error than h over the entire distribution of instances.

In this case, the ID3 algorithm is applied to the task of learning which medical patients have a form of diabetes. The horizontal axis of this plot indicates the total number of nodes in the decision tree, as the tree is being constructed. The vertical axis indicates the accuracy of predictions made by the tree. The solid line shows the accuracy of the decision tree over the training examples, whereas the broken line shows accuracy measured over an independent set of test examples (not included in the training set). Predictably, the accuracy of the tree over the training examples increases monotonically as the tree is grown. However, the accuracy measured over the independent test examples first increases, then decreases.

Reduced Error Pruning

How exactly might we use a validation set to prevent overfitting? One approach, called reduced-error pruning (Quinlan 1987), is to consider each of the decision nodes in the tree to be candidates for pruning. Pruning a decision node consists of removing the subtree rooted at that node, making it a leaf node, and assigning it the most common classification of the training examples affiliated with that node. Nodes are removed only if the resulting pruned tree performs no worse than the original over the validation set. This has the effect that any leaf node added due to coincidental regularities in the training set is likely to be pruned because these same coincidences are unlikely to occur in the validation set. Nodes are pruned iteratively, always choosing the node whose removal most increases the decision tree accuracy over the validation set. Pruning of nodes continues until further pruning is harmful (i.e., decreases accuracy of the tree over the validation set).

Rule Post-Pruning

In practice, one quite successful method for finding high accuracy hypotheses is a technique we shall call rule post-pruning. Rule post-pruning involves the following steps:

1. Infer the decision tree from the training set, growing the tree until the training data is fit as well as possible and allowing overfitting to occur.

2. Convert the learned tree into an equivalent set of rules by creating one rule for each path from the root node to a leaf node.

3. Prune (generalize) each rule by removing any preconditions that result in improving its estimated accuracy.

4. Sort the pruned rules by their estimated accuracy, and consider them in this sequence when classifying subsequent instances.

Reduced Error Pruning

How exactly might we use a validation set to prevent overfitting? One approach, called reduced-error pruning (Quinlan 1987), is to consider each of the decision nodes in the tree to be candidates for pruning. Pruning a decision node consists of removing the subtree rooted at that node, making it a leaf node, and assigning it the most common classification of the training examples affiliated with that node. Nodes are removed only if the resulting pruned tree performs no worse than the

original over the validation set. This has the effect that any leaf node added due to coincidental regularities in the training set is likely to be pruned because these same coincidences are unlikely to occur in the validation set. Nodes are pruned iteratively, always choosing the node whose removal most increases the decision tree accuracy over the validation set. Pruning of nodes continues until further pruning is harmful (i.e., decreases accuracy of the tree over the validation set).

Rule Post-Pruning

In practice, one quite successful method for finding high accuracy hypotheses is a technique we shall call rule post-pruning. Rule post-pruning involves the following steps:

1. Infer the decision tree from the training set, growing the tree until the training data is fit as well as possible and allowing overfitting to occur.

2. Convert the learned tree into an equivalent set of rules by creating one rule for each path from the root node to a leaf node.

3. Prune (generalize) each rule by removing any preconditions that result in improving its estimated accuracy.

4. Sort the pruned rules by their estimated accuracy, and consider them in this sequence when classifying subsequent instances.

Lecture 8

Random forest algorithm

Random Forests are collection of decision trees. Random Forests are an ensemble classifier using many Decision Tree models. Random Forests are way of averaging multiple Decision Trees built from different parts of the training set with the goal of reducing overfitting by a single Decision Tree. The increased performance comes at a cost of some loss of interpretable and accurate fit to training data. Random Forests use bagging to construct multiple Decision Trees. Bagging or bootstrap aggregation is a technique of sampling a subset of training data with replacement and constructs the model based on the sampled training set. For a large dataset D, it is expected to have 62.3% of unique samples from D in each of the subsets. Random Forests grow many Decision Trees by sampling from the input dataset D for a set of features instead of the training data samples. The process is also called feature bagging. If one of the features is highly correlated, then that feature will be selected by many different trees. This causes the trees to be correlated and thus increases the bias of the algorithm, reducing the accuracy. Typically, for a classification problem with n features originally,

pn features are used in each split. For regression problems, number of features recommended is n/3.

Data Bagging

Consider bagging as putting original dataset into different bags. Data is randomly chosen from the original D dataset and puts into various bags. If you have to create B bags, for each bag, randomly choose some (actually 62.3original dataset D, and put into the bag. In this way, put the data in all the bags. Since this sampling of data is with replacement, the same data can be present in multiple bags. The same data can also potentially go multiple times in the same bag just that duplicate data in the same bag is effectively useless in that bag, from the perspective of creating a decision tree for that bag. Each of the bag so created will form one tree.

Feature Bagging

Each of the B bags so formed still has n features. For each tree, you should create feature bagging, i.e., randomly chose some features, and you will use only those features for creating a decision tree for the data in that bag. For any data bag, create multiple sets of feature bags. Use cross validation to decide which feature bag is most apt for this specific data bag. And use this feature bag for the data to create a Decision Tree. So, each data bag will have a different set of features chosen for creating the Decision Tree.

Cross Validation in Random Forests

In Random Forests, cross validation is estimated internally during the run because of the bagging procedure. For each bag, based on sampling, about 62dataset are used. Hence, about a third of the samples are not present in ith tree construction. These left out data will be used for cross validation, to identify the most useful feature set combination, among the multiple randomly chosen feature bags for that data bag. The proportion of sample classified incorrectly from the cross validation set (for that data bag) over all classifications of the cross validation set is the error estimate of the system. This error estimate is also referred to as out-of-bag (OOB) error estimate. OOB is mean prediction error on each training sample j, using only the trees that did not have sample j in the bootstrap sample.

Prediction

So, at the end of bagging, you now have B Decision Trees, each tree with reduced dataset and reduced feature set. For a new data point, prediction of its value is aggregation of predicted values from the trees. The prediction can be categorical or regression based on the Decision Tree construction. The final result is aggregation of results from all the Decision Trees as

$$f = \frac{1}{B}\sum_{n=1}^{B}(f_n(x))$$

where

f is the final prediction from the Random Forest

B is the number of trees constructed from dataset

n is the index of Decision Tree constructed

f_n is the result from Decision Tree n

x is the input sample that may not be in the training set, for which prediction is desired

Variable Importance

When a large number of variables are involved in decision-making, Random Forests can be used to determine importance of the variables involved and order them with respect to their importance in the decision-making. The importance of variables is derived from mean decrease in impurity of each variable and the degree of interaction of each input variable with other input variables described in context of Decision Tree in Sects describes the variable importance computation using the OOB technique discussed in Sect. 7.5.3. The intuitive notion in determining the variable importance is that if the variable is important, then rearranging the values of the variable in constructing the trees will not reduce the prediction accuracy. In practice, for a variable m, compute the number of correct classifications of the tree for out-of-bag cases. Permute values of variable m and then compute the classification of out-of-bag case for the tree. Compute the difference in number of correct classifications after permutation and before permutation. The average difference over all the trees is the importance score of the variable m.

Determining variable importance is useful in improving the results by constructing trees with subset of variables when large number of variables are involved in decision-making. Variable importance also provides intuitive understanding of importance in applications like genetics and neurosciences.

Disadvantages of Random Forrest

Random Forests can overfit with the data when there are large number of high cardinality categorical variables. For example, with two input variables that have 100 and 200 distinct values, there are 200,000 unique combinations that could lead to a decision. In training the forest, the data will fit to the data too well and may not

perform well with the validation on new data. Random Forests also provide discrete models for prediction. If the response variable is a continuous variable, then there are only n distinct values that are possible through prediction. Other regression mechanisms would provide a model for continuous data prediction. Random Forests unlike Decision Trees are not easy to interpret. As the result is from ensemble of trees, interpretation from intuition point of view is very hard. Random Forests do not support incremental learning very well. With new data they have to be relearned.