Machine Learning by Andrew NG
A Biased Summary

Jordi Busquets Blanco

April 2021

Contents

1 Introduction

This document is a quick summary of the *Coursera* course on Machine Learning, by Andrew NG[1]. It's just a compilation of the main ideas as well as the most relevant formulas. For more detail, please refer to the course notes.

As the title indicates, it is a biased summary in that I have reflected the aspects that I found more interesting or relevant, which of course may not match the interests of other students of the course. Some parts have been expanded with added material, while other parts of the course have been left out. In any case, I hope you find these course notes useful.

1.1 Glossary

Throughout the document, we will use a few common symbols, detailed as follows:

- **m**: the number of training examples.
- **n**: the number of input variables examples, or features.
- **x**: the input variables, or features. Without any subscripts, x represents a vector of vectors, of size m, while $x^{(i)}$ is the i-th vector of n input variables (out of m). Example: we may be trying to estimate property prices from a set of input variables for a given property: square meters, location (longitude, latitude), number of bedrooms, number of bathrooms, whether it has parking or not, etc. All these variables would be the inputs to the training algorithm, the x. When talking about all the x features in matrix form (a mxn matrix, we use capital X.
- **y**: for supervised learning, these are the output values for each example. Without any subscripts, y represents a vector of size m, while $y^{(i)}$ is the i-th output value (out of m). In our example, y would be the known property prices for the training sample.
- $\mathbf{h}_\theta(\mathbf{x})$: the learning algorithm, or hypothesis, which we use to compute y^i from x^i.
- θ: the parameters of our hypothesis, which we need to find as part of the training. Without any subscripts, θ represents a vector, while θ_i is the i-th parameter (out of $n+1$, as we will see).
- $\mathbf{J}(\theta)$: the cost function that we minimize during the learning algorithm.
- α: the learning rate, controlling by how much parameters are allowed to change between iterations of the training algorithm.
- λ: the regularisation parameter, controlling how much weight we give to the regularisation term in the cost function.
- **C**: the regularisation parameter for Support Vector Machines.
- σ: Gaussian kernel parameter to control the smoothness of the function. Used in support Vector Machines. Also used to denote the variance for Gaussian distributions.
- $\mathbf{\Sigma}$: covariance matrix, used in the description of the PCA method and in the multivariate Gaussian distribution.

[1]https://www.coursera.org/learn/machine-learning

2 Supervised Learning

Supervised learning applies when the y values are known for a collection of x values. Our training data set looks like

$$\text{Training set } = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\} \tag{1}$$

In the most general case, each $x^{(i)}$ is an array of n features, $x^{(i)} = \{x_1^{(i)}, ..., x_n^{(i)}\}$, while $y^{(i)}$ is an output vector of K entries, $y^{(i)} = \{y_1^{(i)}, ..., y_K^{(i)}\}$ (for most of the document we discuss cases with $K = 1$). The learning is said to be *supervised* because we know what the desired output is for the training data points, thus allowing us to provide them during the calibration of the algorithm.

2.1 Linear regression

Here we assume that a good model to predict the outputs $y^{(i)}$ is

$$h_\theta(x^{(i)}) = \sum_{j=0}^{n} \theta_j x_j^{(i)} = \theta^T x^{(i)}, \quad x_0^{(i)} = 1 \tag{2}$$

When there is a sole input variable ($n = 1$) we speak of *univariate* linear regression. Otherwise we speak of a *multivariate* linear regression. Note that in either case, we always add the so called **bias** term ($j = 0$), with dummy $x_0^{(i)}$, which enables the model to match the output distribution when $x^{(i)} = 0$.

Our aim when training the model is to find the values of θ that best fit the data. This is a standard optimisation problem, and as such we need to define an **objective** or **cost function**, which is a function constructed in such a way that it has a global minima for the optimal θ. The learning algorithm is thus aimed at finding this global minimum. For our linear regression problem, the objective function is just the sum of squared differences, as follows

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \tag{3}$$

It is clear that the value of $j(\theta)$ will be lowest when $h_\theta(x)$ best matches y. Obviously in order to guarantee a solution to this optimisation problem we need to show that the $J(\theta)$ is convex for each and every θ_j parameter. This is easy to show by computing the second derivative of the cost function and showing that it is always positive. To show this we rewrite the cost function as a function of θ_j only, which makes the calculation easier:

$$\begin{aligned}
J(\theta_j) &= \frac{1}{2m} \sum_{i=1}^{m} (a_i\theta_j + b_i)^2 \\
\frac{\partial J(\theta_j)}{\partial \theta_j} &= \frac{1}{m} \sum_{i=1}^{m} a_i (a_i\theta_j + b_i) \\
\frac{\partial^2 J(\theta_j)}{\partial \theta_j^2} &= \frac{1}{m} \sum_{i=1}^{m} a_i^2 \geq 0 \quad \text{since } a_i \in \mathbb{R}
\end{aligned}$$

This proves the convexity of $J(\theta_j)$ $\forall j = 0, ..., n$. Even more, each term in the sum is positive, which means that each of them is convex individually. Indeed, we know that the sum of convex

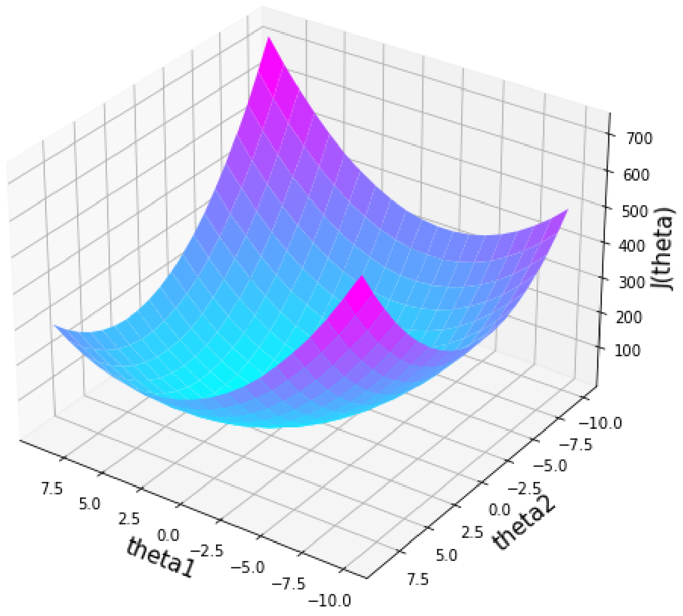

Figure 1: The cost function $J(\theta)$ for a model with 2 input variables x_1 and x_2 and $\theta_0 = 0$.

functions is also convex. Convexity of $J(\theta)$ implies that the function has a minimum, and it is unique and therefore global. Figure 1 shows an example of a linear regression cost function.

As discussed, the training algorithm aims to find the values of θ_j that minimize the cost function:

$$\text{Goal: } \min_\theta J(\theta) \tag{4}$$

The simplest way to solve this problem is to use **gradient descent**, which consists of gradually changing θ in the opposite direction of the value of the derivative of the cost function with respect to θ. In other words, if the derivative of the cost function with respect to θ is positive, we reduce θ, and vice versa. Mathematically, the iterative update rule for each θ_j can be written as

$$\begin{aligned} \theta_j &:= \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \\ &= \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}, \quad x_0^{(i)} = 1 \end{aligned} \tag{5}$$

where α is the so called **learning rate**, determining the size of the updates to θ_j. When α is too small, the rate of convergence will be very slow, but it will be secure. On the other hand, when it is too large, the algorithm will jump around the minimum, struggling to converge, even diverging. Finding the right α is more an art than a science. As we will see later, monitoring he value of $J(\theta)$ as the learning algorithm progresses is crucial to determine whether the learning rate is appropriate.

The correct way to run the above learning algorithm is to apply the update rule for all θ_j **simultaneously**. In other words, during a learning iteration the updated value for θ_j should not be used to update θ_{j+1} during the same iteration.

2.1.1 Feature scaling

For multivariate linear regressions it is advisable to ensure that all variables are comparable in magnitude. This is achieved by shifting and scaling the variables by their mean and standard deviation prior to running the learning algorithm, as follows

$$\hat{x}_j^{(i)} = \frac{x_j^{(i)} - \bar{x}_j}{\sigma_j}, \quad j > 0 \tag{6}$$

where

$$\bar{x}_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)} \tag{7}$$

$$\sigma_j = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(x_j^{(i)} - \bar{x}_j\right)^2} \tag{8}$$

Note that the bias term $x_0^{(i)}$ is not included in $\bar{x}_j$ and σ_j, since it is a synthetic feature and it's always 1 ($\bar{x}_0 = 1$, $sigma_0 = 0$).

2.1.2 Polynomial regression

Sometimes the best fit to the input data is achieved by adding polynomial terms to the regression expression: $J(\theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^{1/2} + ...$ Adding such terms may help improve convergence.

2.1.3 Analytical solution

The above update formulas solve the optimisation problem iteratively. For linear regression, however, an analytical solution exists. In matrix form, this solution reads

$$\theta = (X^T \cdot X)^{-1} \cdot X^T y \tag{9}$$

where here X is a matrix with m rows and $n+1$ columns (with the bias term $x_0^{(i)} = 1$). This method is attractive because there is no need to choose α, and the solution is optimal. However, the drawback is having to compute the inverse matrix $(X^T \cdot X)^{-1}$, which can be very expensive for large values of n, as the computation cost is of order $\mathrm{O}(n^3)$ ($X^T \cdot X$ is an $(n+1)$x$(n+1)$ matrix). The iterative method is recommended for large n, and the analytic solution for small n.

Note that when using the analytical solution we may encounter the problem that $X^T \cdot X$ is **not invertible**. This generally points to one of two potential issues, or both:

- There are redundant features, e.g. we may be using the same input feature in two different units.
- There are too many features, i.e. $m \leq n$. In this case, remove some of them.

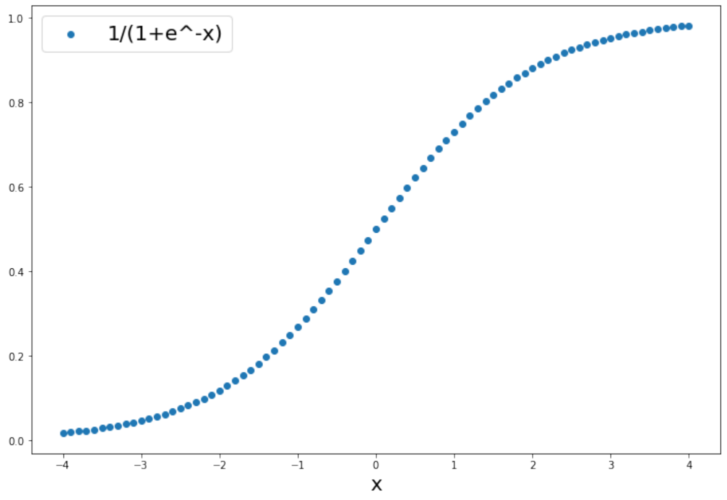

Figure 2: The sigmoid function used in logistic regression.

2.2 Logistic regression

Logistic regression is used when the output of our model, the y, is a binary value, e.g. {yes, no}, $\{1, 0\}$. While linear regression can output any value, here we just need 2, so we want a hypothesis that generates only two values. Therefore, assuming that $y^{(i)} = 0$ or 1, we select a classifier $h_\theta(x)$ such that $0 \leq h_\theta(x) \leq 1$ and we build a model which generates outputs as follows:

$$h_\theta(x) \begin{cases} \geq 0.5 & \text{predicts } y = 1, \\ < 0.5 & \text{predicts } y = 0. \end{cases} \tag{10}$$

One way to achieve this is by using the **logistic regression** model (or sigmoid function), namely

$$\begin{aligned} g(z) &= \frac{1}{1+e^{-z}}, \\ h_\theta(x) &= g(\theta^T x) = g\left(\sum_{j=0}^{n} \theta_j x_j\right), \text{ with } x_0 = 1 \end{aligned} \tag{11}$$

This function approaches 1 as $\theta^T x$ becomes large and positive, while it approaches 0 as $\theta^T x$ becomes large but negative (see fig. 2). In this way, we can interpret $h_\theta(x)$ as the probability that $y = 1$, given x, parametrized by θ, i.e.

$$h_\theta(x) = p(y = 1|x, \theta) \tag{12}$$

The decision boundary, i.e. when $h_\theta(x) = 0.5$, occurs when $\theta^T x = 0$. For logistic regression we need a different cost function than the one we used for linear regression. Now we want $J(\theta)$ to penalise cases when $h_\theta(x) = 1$ if $y = 0$, and vice versa. We do this with the help of logarithms, as follows

$$\text{cost}_{LG}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1, \\ -\log(1 - h_\theta(x)) & \text{if } y = 0. \end{cases} \tag{13}$$

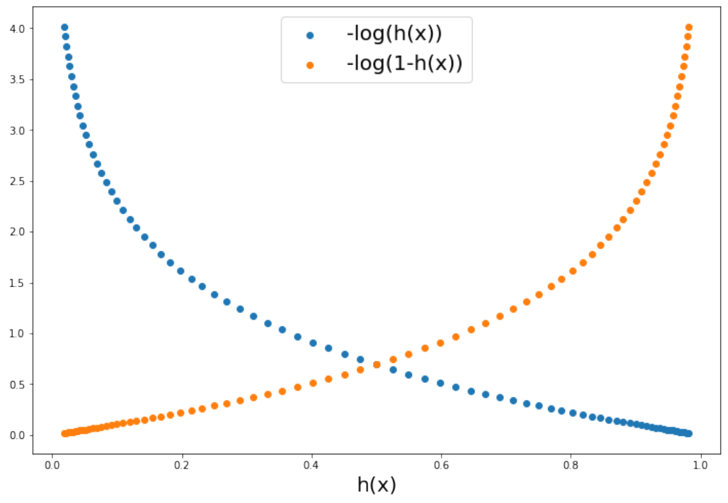

Figure 3: The cost function for $y = 1$ (blue) and $y = 0$ (orange) respectively. Note how the function has a minimum when $y = h_\theta(x)$, as we require.

As can be seen in fig. 3, this function will heavily penalise deviations between y and $h_\theta(x)$. The complete cost function $J(\theta)$ can then be written as a function of $\text{cost}_{LG}(h_\theta(x), y)$, as

$$\begin{aligned} J(\theta) &= \frac{1}{m}\sum_{i=1}^{m}\text{cost}_{LG}(h_\theta(x^{(i)}), y^{(i)}) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right) \end{aligned} \tag{14}$$

As this is again an optimisation problem, we need to show that the cost function is a convex function of each individual θ_i. We do the same exercise as for linear regression and prove this by computing the second derivative and showing that it is positive. We start by looking at the sigmoid function, rewritten as a function of the relevant θ_j, for a given input data point $x^{(i)}$:

$$\begin{aligned} g_i(\theta_j) &= \frac{1}{1+e^{-a_i\theta_j+b_i}} \\ \frac{\partial g_i(\theta_j)}{\partial \theta_j} &= a_i g(\theta_j)(1-g(\theta_j)) \end{aligned}$$

Now we look at the cost function itself, rewritten in the same fashion for simplicity:

$$
\begin{aligned}
J(\theta_j) &= -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log(g_i(\theta_j)) + (1-y^{(i)})\log(1-g_i(\theta_j))\right), \\
\frac{\partial J(\theta_i)}{\partial \theta_j} &= -\frac{1}{m}\sum_{i:y^{(i)}=1}^{m} a_i(1-g_i(\theta_j)) + \frac{1}{m}\sum_{i:y^{(i)}=0}^{m} a_i g_i(\theta_j) \\
\frac{\partial^2 J(\theta_j)}{\partial \theta_j^2} &= \frac{1}{m}\sum_{i:y^{(i)}=1}^{m} a_i^2 g_i(\theta_j)(1-g_i(\theta_j)) + \frac{1}{m}\sum_{i:y^{(i)}=0}^{m} a_i^2 g_i(\theta_j)(1-g_i(\theta_j)) \\
&= \frac{1}{m}\sum_{i=1}^{m} a_i^2 g_i(\theta_j)(1-g_i(\theta_j)) = \frac{1}{m}\sum_{i=1}^{m}\frac{a_i^2 e^{-a_i\theta_j+b_i}}{(1+e^{-a_i\theta_j+b_i})^2} \geq 0
\end{aligned}
$$

As in the linear regression case, each term is the sum over m is positive individually, meaning that each term in the cost function sum is convex too. The goal of the training algorithm is, again, to minimise $J(\theta)$ by modifying the values of θ. Interestingly, the solution to this gradient descent problem can be identically written as the one used for linear regression, namely

$$\theta_j := \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \tag{15}$$

where of course now $h_\theta(x)$ is given by equation 11, and where as usual $x_0^{(i)} = 1$.

2.2.1 Multi-class classification: one-vs-all

When the possible set of values of $Y^{(i)}$ is still discrete but larger than 2, the simplest solution is to fit a logistic regression for each output. In this scenario, and assuming K possible output values, for each output value y_k we fit a regression model $h_{\theta_k,k}(x)$, and we train this model using $y = 1$ for the point with y_k output, and $y = 0$ for all other point with outcomes $y_j,\ j \neq k$. Then, once all models have been trained, given a new data point x we need to run all models, and the model prediction will be chosen looking at which of our K models yields the largest value, i.e.

$$\text{Prediction is } y_k \text{ if } \max_i(h_{\theta_i,i}(x)) \text{ occurs for } i = k \tag{16}$$

2.3 Overfitting and Regularisation

One of the most common problems when fitting a linear or logistic regression is that of overfitting. Overfitting occurs when the parameters θ are not generic enough and, instead, they are too geared towards replicating the training set examples provided. Overfitting is easy to spot because while the error in the training set is small, the error when the model is used on new examples is large. In other words, the model fits the training set very well, but it fails to generalise well to new examples. There are two common solutions for overfitting:

- **Reduce the number of features**: a reduction of degrees of freedom will force the model to be more generic.
- **Regularize the model**: regularisation is a way to change the training algorithm that forces the parameters θ to stay small in magnitude, preventing θ to take large values, which is something commonly seen in overfit models. Regularisation works well when we have models with many features but where all of them contribute a small amount towards the prediction of y.

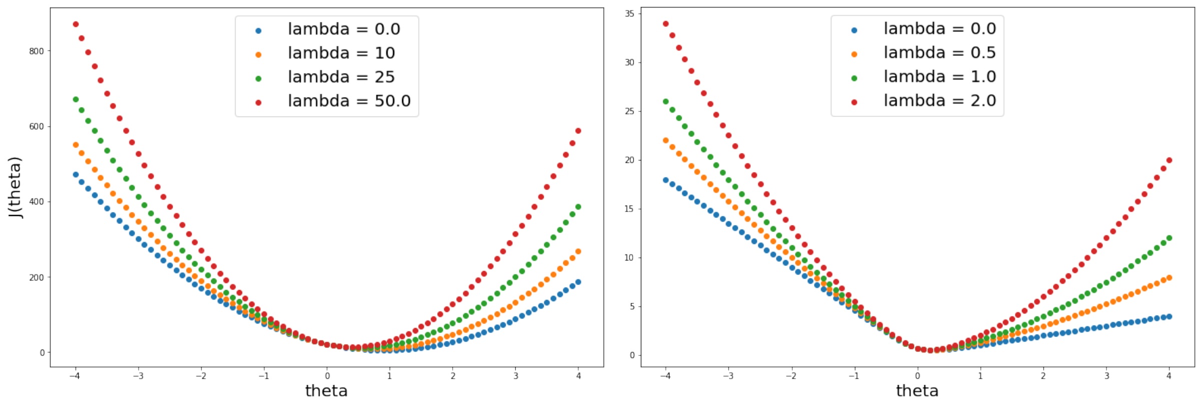

Figure 4: Impact of λ on linear (left) and logistic (right) regression algorithms. Here we look at synthetic problems with a single input variable, and $\theta_0 = 0$. The x axis represents θ_1, and the y axis shows the cost function $J(\theta) = J(\theta_1)$ for multiple values of λ. The larger λ is, the faster the cost function grows with $|\theta|$.

Regularisation is implemented by adding a term to the cost function $J(\theta)$, as follows:

- Linear regression:

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{17}$$

- Logistic regression:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{18}$$

Please note that the bias term $j = 0$ is not included in the regularisation term, meaning that θ_0 is unaffected by it. The effect of this new term is to penalise large values for θ_j, thus forcing the training algorithm to keep their values as low as possible. The choice of λ is very important: a very large value will cause $\theta_j = 0$, which is obviously undesirable; on the other hand, too small a value may not be enough to correct the overfitting. The impact of λ both in linear and logistic regression problems is shown in figure 4.

Another effect of this new term is that it impacts the training algorithm, as it changes the derivative of the cost function with respect to the θ_j parameters. For both linear and logistic, and notwithstanding the different definitions of $h_\theta(x)$, the new update rule now becomes:

$$\theta_0 := \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)} \qquad (\text{with } x_0^{(i)} = 1) \tag{19}$$

$$\theta_j := \theta_j - \frac{\alpha}{m}\left(\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} - \lambda\theta_j\right), \quad j > 0 \tag{20}$$

2.3.1 Analytical solution for linear regression

Even with regularisation, we can still solve analytically for the linear regression case:

$$\theta = \left(X^T \cdot X + \lambda \cdot \tilde{I}\right)^{-1} \cdot X^T y = \left(x^T \cdot x + \lambda \cdot \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}\right)^{-1} \cdot x^T y \tag{21}$$

where, again, X is a matrix with m rows and $n+1$ columns. The *pseudo*-identity matrix $\tilde{I}$ is a $(n+1)$x$(n+1)$ square matrix. It is not an identity matrix because $\tilde{I}_{00} = 0$, since we do not regularize the θ_0 terms. Provided that $\lambda > 0$, the matrix $(X^T \cdot X + \lambda \cdot \tilde{I})$ is not singular, and is thus invertible.

2.4 Neural Networks

Neural networks are algorithms inspired by the human brain. Researchers were motivated by the fact that the human brain is not only very good at learning new skills but also at remembering and recalling large amounts of information.

The design pattern that the brain inspired was one comprised of a large number of simple units with huge connectivity between them (a neuron in the human brain may be connected to as many as 10 thousand others). In these models, as in the human brain, each neuron takes all the input signals from the neurons connected to it and applies some sort of *activation* function, generating an output signal that is then sent to all neurons connected to it. This generates a natural dynamic process throughout the system which ends up with a final output or outputs.

For our purposes, we can think of a neural network as a series of layers of neurons (or **activation units**) where each layer receives inputs from the units in the previous layer, each unit then applies an activation function, and finally all units in the layer send their outputs to the next layer. And so on. Because for our applications we need the system to have an output (either a one value of many), the final layer is called the *output* layer, and it may have 1 or many units depending on the desired output format. The first layer is the input layer, where the model features $x^{(i)}$ are fed into the model. Any layer in between us called a *hidden* layer. This design can be seen in fig. 5. The process of feeding the values of the activation units of one layer into the next layer is called **forward propagation**.

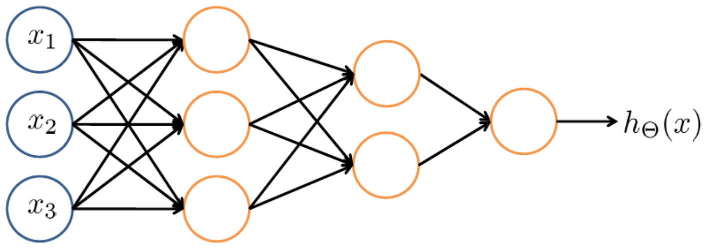

Figure 5: A schematic representation of a neural network: the input layer is how we feed the input features to the model. The output layer generates the model predictions $h_\theta(x)$. The hidden layers apply activation functions on all neurons. Each neuron has its own θ parameters, thus generating a model involving a huge number of degrees of freedom.

The activation function used within the neurons can be anything, as long as it generates a single real number as output. For our purposes, we will consider the sigmoid activation function $g(z)$, as in equation 11. Defining $a_i^{(j)}$ as the i-th activation unit in layer j, we can define the model mathematically as follows

$$\begin{aligned} z_i^{(j+1)} &= (\theta_i^{(j)})^T a^{(j)} = \sum_{l=0}^{n_j} \theta_{il}^{(j)} a_l^{(j)}, \quad a_0^{(j)} = 1 \text{ (\textit{bias} unit in } j\text{-th layer)} && (22) \\ a_i^{(j+1)} &= g\left(z_i^{(j+1)}\right) && (23) \\ h_{\theta,k}^{nn}(x) &= a_k^{(L)} = g\left(z_k^{(L)}\right), \; k = 0, ..., K && (24) \end{aligned}$$

where

- n_j is the number of units in layer j. This implies that n_1 is equal to the number of features x (we count the number of layers from 1).
- $\theta^{(j)}$ is the (n_{j+1})x$(n_j + 1)$ matrix of weights to be used in the activation units in layer $j+1$, using as inputs the outputs of layer j (the $+1$ comes from the bias term in layer $j+1$). Therefore, $\theta_i^{(j)}$ is the vector of parameters linking the outputs of layer j with the i-th unit in layer $j+1$. $\theta_{il}^{(j)}$ is just the l-th element in $\theta_i^{(j)}$.
- L is the total number of layers, including the output layer.
- $h_{\theta,k}^{nn}(x)$ is the model prediction for the k-th output unit in the final (L-th) layer, given the input vector x.
- K is the total number of output units.

Neural networks are thus useful when the number of features is huge (e.g. image recognition), where regressions would be prohibitively expensive. Because of the large number of degrees of freedom they feature, they are also indicated for problems where the classification boundary is highly non-linear, or even disconnected.

2.4.1 Multi-class classification

Because the final layer can have multiple units, neural networks handle multi-class classification out of the box. Just assign each possible output value with an output unit. For a given training example with output $y = y_k$, we just assign $y = \{0, ..., 0, 1, 0, ..., 0\}$, with 1 in the k-th position. As an example, imagine a model that distinguishes images into three categories: cars, motorbikes and bicycles ($K = 3$). Each training example of a car would have as output $y_1 = \{1, 0, 0\}$, for motorbikes we would use $y_2 = \{0, 1, 0\}$, and $y_3 = \{0, 0, 1\}$ for bicycles. Once the model has been trained, for each input vector x the model will generate a vector $h_{\theta,k}^{nn}(x)$. The model prediction will be decided based on how close $h_{\theta,k}^{nn}(x)$ is to one of the valid outputs y_k.

2.4.2 Model training: backward propagation

The cost function for a neural network looks similar to the ones we have already seen for logistic regression, with a couple of differences. The expression is

$$\begin{aligned} J^{nn}(\theta) &= -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K}\left(y_k^{(i)}\log(h_{\theta,k}^{nn}(x^{(i)})) + (1-y_k^{(i)})\log(1-h_{\theta,k}^{nn}(x^{(i)}))\right)\right] \\ &+ \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{n_j}\sum_{j=1}^{n_{j+1}}\left(\theta_{ij}^{(l)}\right)^2 \end{aligned} \quad (25)$$

The main differences with logistic regression are that we now sum over all possible output K values in the first term, and also over all layers in the regularisation term. The usual gradient descent method for the model training requires the computation of $J^{nn}(\theta)$ and $\frac{\partial}{\partial\theta_{ij}^{(l)}}J^{nn}(\theta)$. The calculation of the gradients is somewhat involved than in previous models. To clarify notation, we define the *error* of unit j in layer l for example i as $\delta_j^{(i,l)}$. It is clear that, for each of our training examples, the error on the last layer is trivially computed as

$$\delta_j^{(i,L)} = h_{\theta,j}^{nn}(x^{(i)}) - y_j^{(i)} = a_j^{(i,L)} - y_j^{(i)} \quad (26)$$

where $a_j^{(i,L)}$ is simply $a_j^{(L)}$ for example i. The calculation of $\delta_j^{(i,l)}$ for the hidden layers is also possible but more involved. We present here the matrix expression for them:

$$\begin{aligned} \delta^{(i,l)} &= \left[\left(\theta^{(l)}\right)^T\delta^{(i,l+1)}\right]\circ\left[g'\left(z^{(i,l)}\right)\right] \\ &= \left[\left(\theta^{(l)}\right)^T\delta^{(i,l+1)}\right]\circ\left[g\left(z^{(i,l)}\right)\left(1-g\left(z^{(i,l)}\right)\right)\right] \\ &= \left[\left(\theta^{(l)}\right)^T\delta^{(i,l+1)}\right]\circ\left[a^{(i,l)}\left(1-a^{(i,l)}\right)\right] \end{aligned} \quad (27)$$

where here the symbol $\circ$ denotes element-wise multiplication of vectors, i.e. $u \circ v = \{u_1v_1, ..., u_nv_n\}$. Remember that the $\theta^{(l)}$ are common to all examples and thus there is no need to specify a super-index i for them. One can show that, in the case of no regularisation ($\lambda = 0$), the gradient terms for a given example i can be computed as

$$\left.\frac{\partial J^{nn}(\theta)}{\partial\theta_{kj}^{(l)}}\right|_i = a_j^{(i,l)}\delta_k^{(i,l+1)}, \quad (\lambda = 0) \quad (28)$$

These expressions now provide a way to train the neural network, whereby we run forward propagation for all examples ($i = 1, ..., m$), and then we run **backward propagation** for all of them, computing the gradient for each of them at each layer. In other words, for each example i we need to:

- Set $a^{(i,1)} = x^{(i)}$.
- Perform forward propagation to compute $a^{(i,l)}$ for $l = 2, ..., L$.
- Using $y^{(i)}$, compute $\delta^{(i,L)} = a^{(i,L)} - y^{(i)}$.
- Compute $\delta^{(i,L-1)}$, $\delta^{(i,L-2)}$, ..., $\delta^{(i,2)}$ using equation 27 (the $\delta^{(i,1)}$ are unnecessary).
- Store the result for $\Delta_{kj}^{(i,l)} \equiv a_j^{(i,l)}\delta_k^{(i,l+1)}$.

Finally, the total gradient is given by the expression

$$\frac{\partial J^{nn}(\theta)}{\partial \theta_{kj}^{(l)}} = \frac{1}{m}\sum_{i=1}^{m} \left.\frac{\partial J^{nn}(\theta)}{\partial \theta_{kj}^{(l)}}\right|_i = \begin{cases} \frac{1}{m}\sum_{i=1}^{m} \Delta_{kj}^{(i,l)} & \text{if } j = 0, \\ \frac{1}{m}\sum_{i=1}^{m} \Delta_{kj}^{(i,l)} + \frac{\lambda}{m}\theta_{kj}^{(l)} & \text{if } j \neq 0 \end{cases} \tag{29}$$

With the expression for the derivatives, the algorithm can now run as usual, updating the values of the θ parameters at each iteration, iteratively, until convergence is reached. Given the complexity of these expressions, and how easy it is to make a mistake while implementing them, it is good practice to compare the $\Delta_{kj}^{(i,l)}$ with manual gradient calculation computed using finite differences, but only during training.

2.4.3 Initialisation of θ

The initialisation of θ is no longer trivial. For logistic regressions we could set all values of θ to zero, but this does not work for neural networks as it would generate lots of identical unit values, and the training algorithm would not work. The best way to initialise them is to use random values around zero (e.g. $-\epsilon \leq \theta_{kj}^{(l)} \leq \epsilon$, for a small enough value of ϵ). This breaks the problem's symmetry and enables gradient descent to progress.

2.5 Support Vector Machines

Support Vector Machines, or *SVMs*, are an evolution of logistic regression, also used for classification problems, which under certain circumstances deliver better results than both neural networks and logistic regression, and are computationally more efficient too.

The fundamental difference is the replacement of the sigmoid function in $h_\theta(x)$ with a much simpler expression. In logistic regression, and in the absence of regularisation, the training algorithm will try to make $h_\theta(x) \approx 1$, and thus $\theta^T x^{(i)} \gg 0$, when $y^{(i)} = 1$. Similarly, when $y^{(i)} = 0$, the algorithm will try to fit $h_\theta(x) \approx 0$, and thus $\theta^T x^{(i)} \ll 0$. SVMs modify the cost function terms to make them simpler, which under certain circumstances makes the optimisation problem easier to solve. The idea is to use terms that, although not differentiable, are a close approximation to the logistic regression terms. Table 2.5 shows the cost function term equivalence between SVMs and logistic regression.

output	**Logistic reg**	**SVMs**
y=1	$\log\left(\frac{1}{1+e^{-\theta^T x}}\right)$	$\text{cost}_1^{svm}(\theta^T x) = \max(0, 1 - \theta^T x)$
y=0	$\log\left(1 - \frac{1}{1+e^{-\theta^T x}}\right)$	$\text{cost}_0^{svm}(\theta^T x) = \max(0, \theta^T x + 1)$

Table 1: Cost function terms for logistic regression and SVMs.

The new terms do not require $\theta^T x \gg 0$ ($\ll 0$), when $y = 1$ (0), to obtain $\text{cost}_{1(2)}^{svm}(\theta^T x) = 0$, but simply values above 1 (-1). This facilitates the learning algorithm and may result in simpler solutions to the optimisation problem. One can also show that, although non-differentiable, convexity of $J(\theta)$ is still guaranteed. Figure 6 shows the SVM cost terms alongside the equivalent logistic regression terms.

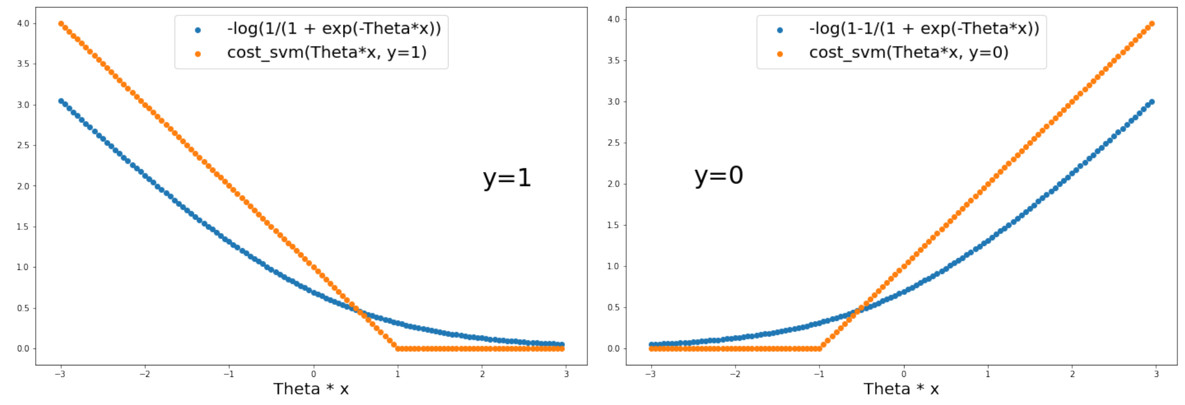

Figure 6: SVM cost function terms for $y = 1$ (left) and $y = 0$ (right) cases. Note that the actual slope in the upward sections of the SVM cost functions is not really relevant; what is important is that the training algorithm does not require very large positive or large negative values for $\theta^T x$ to match $h_\theta(x) = 1$ or 0, but simply values above 1 or below -1 respectively.

The SVM cost function can then be written as

$$J^{svm}(\theta) = C \sum_{i=1}^{m} \left(y^{(i)} \text{cost}_1^{svm}(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{cost}_0^{svm}(\theta^T x^{(i)}) \right) + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2 \tag{30}$$

It's worth making a few comments:

- The minus signs present in the cost function for the logistic regression are now gone, as they have been absorbed by $\text{cost}_1^{svm}(\theta^T x)$ and $\text{cost}_0^{svm}(\theta^T x)$.
- The $\frac{1}{m}$ scaling factors are gone (this is simply SVM convention).
- The regularization parameter λ is not present and, instead, a new parameter C appears in the first term (again, this is SVM convention). Their impact is the same, as they both control the relative importance of each term in the function. For all intents and purposes, one should consider $C \approx \frac{1}{\lambda}$: a large C has the same effect as a small λ, and vice versa.
- For future reference, we define $h_\theta^{svm}(x) \equiv \theta^T x$, as in linear regression.

As usual, the optimisation algorithm aims to minimise $J^{smv}(\theta)$ by changing θ. The minimisation algorithm is more stringent than for logistic regression because for SVMs $h_\theta(x) = 1$ (0) when $\theta^T x \geq 1$ rather than simply $\theta^T x \geq 0$. This generates a model where the decision boundary is far away as possible from the actual points (this distance being known as the **margin**), making SVMs a **large margin classifier**. What this means is that SVMs will optimise to a solution with more robust and intuitively correct boundaries.

2.5.1 Non-linear decision boundaries and Kernels

The question now is how to adapt SVMs to non-linear decision boundary problems. Traditionally, we have seen that our models used standard linear terms in the cost function, namely

$$h_\theta(x) = \theta_0 + \theta_1 x1 + \theta_2 x_2 + ... + \theta_n x_n$$

For non-linear boundaries, polynomial terms were introduced, since they are able to generate non-linearity:

$$h_\theta(x) = \theta_0 + \theta_1 x1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_2 + \theta_5 x_2^2 + ...$$

The question we ask ourselves now is: are there different or better choices for the features, beyond polynomial terms? In other words, can be find functions $f_i(x)$ such that, when used with SVMs (or any other model) provide better predictive models?

$$h_\theta(x) = \theta_0 + \theta_1 f_1(x) + \theta_2 f_2(x) + \theta_3 f_3(x)...$$

The functions $f_i(x)$ are called **kernels** and are generally not of polynomial form. A very common choice is the Gaussian kernel, which is a measure of the *similarity* between the point x and some pre-determined *landmarks* point $l^{(i)}$. In other words, the Gaussian kernel will be maximised when the point x matches the point $l^{(i)}$, and will decrease as x is further and further away from $l^{(i)}$. In mathematical form this is expressed as

$$k_G(x, l^{(i)}) = \text{similarity}(x, l^{(i)}) = \exp\left(-\frac{\| x - l^{(i)} \|_2^2}{2\sigma^2}\right) \tag{31}$$

where $\| x \|_2$ is the standard Euclidean or l_2-norm, i.e. $\| x \|_2 = \sqrt{\sum_i x_i^2}$ (in our kernel definition the norm is squared, so no square root is present). The main idea is that the kernel will be close to 1 when the points x and $l^{(i)}$ are close to each other, and vice versa. The standard deviation σ determines how quickly the value of the kernel decreases towards zero with the distance between the points. Figure 7 shows the impact of σ on the Gaussian kernel. It is also important to note that feature scaling (see sec. 2.1.1) should be performed when using Gaussian kernels, as otherwise the algorithm will give additional weight to features with large values.

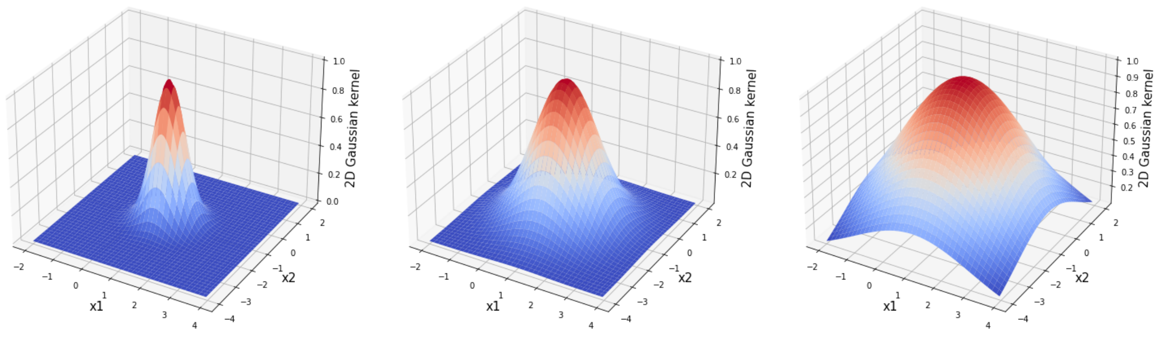

Figure 7: Impact of σ on the Gaussian kernel values. In these examples, l is a two dimensional point, equal to $l = [1, -1]$. The kernel is plotted as a function of $x = [x_1, x_2]$ for $\sigma = 0.5$ (left), $\sigma = 1.0$ (center) and $\sigma = 2.0$ (right).

With the use of Gaussian kernels, $h_\theta^{svm}(x)$ becomes

$$h_\theta^{svm}(x) = \theta_0 + \sum_j \theta_{j=1}^n k_G(x, l^{(j)}) \tag{32}$$

In its most general form, for generic kernels $f_i(x, l^{(j)})$, we introduce a bias term for simplicity and we can write

$$h_\theta^{svm}(x) = \sum_j \theta_{j=0}^n f_j(x, l^{(j)}) \tag{33}$$

The final question is, how should we choose the $l^{(i)}$? Generally, these are chosen from the set of training examples, so that in fact we are modelling the closeness of new input points x to the training data points. In the most general case, we set $l^{(i)} = x^{(i)}$, $\forall i = 1, ..., n$, thus covering the entire training set. The new cost function now reads

$$J^{svm}(\theta) = C \sum_{i=1}^m \left(y^{(i)} \text{cost}_1^{svm}(\theta^T f^{(i)}) + (1 - y^{(i)}) \text{cost}_0^{svm}(\theta^T f^{(i)}) \right) + \frac{1}{2} \sum_{i=1}^m \theta_j^2 \tag{34}$$

where $f^{(i)} \equiv f_i(x, x^{(i)})$. Note also that the second term is now a sum up to m, since the number of θ values is $m + 1$ (i.e. θ, $f^{(i)} \in \mathbb{R}^{m+1}$; the sum excludes the bias term). As usual, given a new point x our model will predict $y = 1$ when $\theta^T f \geq 0$, and vice versa. It is important to understand that when the number of examples m is large, this is a huge function, and potentially very expensive to evaluate. This issue is somewhat lessened by the fact that the SVM cost functions are very cheap to compute. On the other hand, using kernels with logistic regression would be prohibitive given the computational complexity of the cost function in that case.

2.5.2 Other kernels

Many other kernels exist beyond the Gaussian kernel. Some of the most widely used are

- **Gaussian kernel**:
$$k_G(x, l) = \exp\left(-\frac{\| x - l \|_2^2}{2\sigma^2} \right) \tag{35}$$
- **Polynomial kernel**:
$$k_P(x, l) = (x^T l + 1)^p \tag{36}$$
- **RBF kernel**:
$$k_{RBF}(x, l) = \exp\left(-\gamma \left(\| x - l \|_2^2 \right) \right) \tag{37}$$
- **Sigmoid kernel**:
$$k_S(x, l) = \tanh\left(\eta x^T l + \nu \right) \tag{38}$$

Note that not any function can be a well-functioning kernel. Kernels need to fulfil certain properties, given by *Mercer's Theorem.* The first and simpler condition is that kernels should be symmetric with respect to x and l (they should be interchangeable). Secondly, the *kernel matrix* needs to be positive semi-definite for the kernel to be well constructed (this is beyond the scope of this document; for more on kernel matrices in the context of SVMs, please refer to this page).

2.5.3 Multi-class classification

As for logistic regression, with SVMs one can also use the one-vs-all approach to multi-class problems, training K SVMs (one per output value) and then selecting the right answer by looking at which $(\theta^{(k)} f)$ is largest for a new point x.

2.5.4 Logistic regression vs SVMs

Here we provide a few pointers to help decide when to use one or the other method, based on the nature and size of the problem. There is no precise way to select the right model, but from experience we can provide some educated guesses. Using n as the number of features and m as the number of training examples available to us, a rough guide could be as follows:

- For large n relative to m (e.g. $n = 10,000$ and $10 < m < 1,000$), logistic regression or an SVM with *linear* kernel (i.e. $\theta^T x$) are recommended.
- For small n and intermediate m (e.g. $1 < n < 1,000$ and $10 < m < 10,000$), SVM with Gaussian kernel is recommended.
- For small n and large m (e.g. $1 < n < 1,000$ and $m > 50,000$), logistic regression or SVM with linear kernel is recommended.

Note that logistic regression and SVMs without kernel (i.e., with linear kernel) are pretty similar to each other in terms of model prediction. Also, note that neural networks will work in most of these cases, but are likely to take longer to train.

2.6 Recommender systems

Recommender systems are widely used in online services of all kinds to provide suggestions to customers about what to purchase next. From large companies like Netflix or Amazon to the smallest online shop, recommender systems are ubiquitous.

These systems utilise information provided by customers, either in the form of purchased items (which presumably means they like them) or as customer feedback (e.g. the standard one to five stars rating). The challenge is of course that we do not have a rating for every item from every single customer, but on the other hand there are many customers that behave in similar ways. Can we combine all this information to build a recommender system?

To simplify matters, let us consider a movie service that attempts to recommend you new movies to watch. Let's define some notation:

- $\mathbf{n_u}$: number of users of the service.
- $\mathbf{n_m}$: number of movies available in the service.
- $\mathbf{r_{ij}}$: whether or not the user i has rated movie j (values 1 or 0 respectively).
- $\mathbf{y^{(i,j)}}$: the rating for movie i provided by user j (defined only when $r_{ij} = 1$). This could be a continuous value within a range or selected from a discrete set.

The problem we want to solve is how to predict ratings $y^{(i,j)}$ for which $r_{ij} = 0$. These ratings will enable the movie service to recommend new movies to its customers.

2.6.1 Content-based systems

It is somewhat apparent that with the information defined above it is not quite enough to build a recommender system: the r_{ij} and $y^{(i,j)}$ can tell the model that two customers have similar tastes, but we do not know which tastes those are. This means that when a new movie becomes available in the service, we'd need many customers to rate it before we can start recommending it to other customers.

Content-based systems solve this problem by going beyond just looking at customers that provide similar ratings for the same movies. They do this by defining a set of features, and assigning each of them a value for each movie. For example, we could use:

- $\mathbf{x_1^{(i)}}$: amount of action (from 1 to 5) in movie i.
- $\mathbf{x_2^{(i)}}$: amount of romance (from 1 to 5) in movie i.
- $\mathbf{x_3^{(i)}}$: amount of drama (from 1 to 5) in movie i.
- $\mathbf{x_4^{(i)}}$: amount of violence/sex/bad language (from 1 to 5) in movie i.
- etc.

Once we have decided our n features, we can predict a rating for movie i for customer j as $y^{(i,j)} = (\theta^{(j)})^T x^{(i)}$. We find ourselves once more with a linear regression learning algorithm[2] where we need to optimise for $\theta^{(j)}$, $\forall j = 1, ..., n_u$. As usual, $\theta^{(j)} \in \mathbb{R}^{n+1}$ due to the bias term $x_0^{(i)} = 1$. To learn $\theta^{(j)}$ for a given customer j we perform the standard minimisation algorithm, namely

$$\min_{\theta^{(j)}} J_\theta(\theta^{(j)}) = \min_{\theta^{(j)}} \frac{1}{2m^{(j)}} \sum_{i:r_{ij}=1}^{n_m} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2m^{(j)}} \sum_{k=1}^{n} (\theta_k^{(j)})^2 \tag{39}$$

where $m^{(j)} = \sum_i r_{ij}$ is the total number of movies rated by customer j. However, it makes sense to solve this as a a global optimisation problem for all users at once. For convenience, we also drop the $m^{(j)}$ factors, as they do not alter the final result in any way:

$$\min_{\theta^{(1)},...,\theta^{(n_u)}} J_\theta(\theta^{(1)}, ..., \theta^{(n_u)}) = \min_{\theta^{(1)},...,\theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r_{ij}=1}^{n_m} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2 \tag{40}$$

where we have added regularisation terms for good measure. The gradient descent iterative process is written in the usual way as

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \frac{\partial J_\theta(\theta^{(1)}, ..., \theta^{(n_u)})}{\partial \theta_k^{(j)}} \tag{41}$$

where the learning rate α has reappeared. Expanding this expression results in

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \sum_{i:r_{ij}=1}^{n_m} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right) x_k^i \quad \text{(for } k = 0\text{)}, \tag{42}$$

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r_{ij}=1}^{n_m} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right) x_k^i + \lambda \theta_k^{(j)} \right) \quad \text{(for } k \neq 0\text{)}. \tag{43}$$

[2]This could also be solved as a multi-class logistic regression algorithm, if ratings are chosen from a set of discrete values, e.g. from 1 to 5.

2.6.2 Collaborative systems

The content-based approach described in the previous section assumes that we have values for the features for every single movie on offer. However, this may not always be the case immediately. Let's assume that some of the movies (new movies, say) do not have values for some or all of the features. However, assuming that we already have a contents-based algorithm in place, and given that users will quickly start rating the new movie, can we use all that information to predict the values of the features for the new movie? This is what collaborative systems do.

Provided that we have a fairly good initial estimate for $\{\theta^{(1)}, ..., \theta^{(n_u)}\}$, maybe because users told us which types of movies they like, or because we trained a contents-based algorithm with existing information, we can solve for the missing features in the same way as we solved for $\theta^{(j)}$, but optimising for $x^{(i)}$ instead. For a single movie i we solve for

$$\min_{x^{(i)}} J_x(x^{(i)}) = \min_{x^{(i)}} \frac{1}{2} \sum_{j:r_{ij}=1}^{n_u} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (x_k^{(i)})^2 \tag{44}$$

For all movies at once, we solve the following minimisation problem:

$$\min_{x^{(1)},...,x^{(n_m)}} J_x(x^{(1)}, ..., x^{(n_m)}) = \min_{x^{(1)},...,x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r_{ij}=1}^{n_u} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 \tag{45}$$

This is then solved in the standard way, using gradient descent. In the most general case, content-based and collaborative systems are run iteratively, one after another, as an ongoing process of obtaining $\theta^{(j)}$ and $x^{(i)}$, constantly improving the values for the user parameters as well as computing missing movie features: $x \longrightarrow \theta \longrightarrow x \longrightarrow \theta \longrightarrow x....$

Taking this one step further, we could in fact solve for both $\theta^{(j)}$ and $x^{(i)}$ simultaneously, which results in a larger minimisation problem but one that needs to be solved only once. The mathematical expression becomes rather large:

$$\min_{\substack{x^{(1)},...,x^{(n_m)},\\ \theta^{(1)},...,\theta^{(n_u)}}} J_{\theta,x}(x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}) =$$

$$\min_{\substack{x^{(1)},...,x^{(n_m)},\\ \theta^{(1)},...,\theta^{(n_u)}}} \frac{1}{2} \sum_{\substack{(i,j):\\ r_{ij}=1}} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2 \tag{46}$$

Note that when we solve for both $\theta^{(j)}$ and $x^{(i)}$, we do not use bias terms, as the model is learning all features at once and therefore adding dummy $x_0^{(i)} = 1$ terms would not improve the algorithm in any way. This means that both $\theta^{(j)}, x^{(i)} \in \mathbb{R}^n$, and the gradient descent algorithm does not require any special treatment for $k = 0$:

$$x_k^{(i)} := x_k^{(i)} - \alpha \left(\sum_{j:r_{ij}=1}^{n_u} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right) \theta_k^j + \lambda x_k^{(i)} \right), \tag{47}$$

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r_{ij}=1}^{n_m} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right) x_k^i + \lambda \theta_k^{(j)} \right). \tag{48}$$

It is recommended to initialise the $\theta^{(j)}$, as well as all the unknown $x_k^{(i)}$, to small values randomly. Finally, once the minimisation algorithm has converged, we can predict the movie rating for a user j and a movie i with (learned) features $x^{(i)} = \{x_1^{(i)}, ..., x_n^{(i)}\}$ as $(\theta^{(j)})^T x^{(i)}$.

2.6.3 Users without a single rating

Before closing the section on recommender systems, it's worth considering customers that have not provided any ratings at all, i.e. users j for which $r_{ij} = 0$, $\forall i = 1, ..., n_m$. Whether we are solving a contents-based or a collaborative system, we can see that in this case the first sum term in the cost function is zero and thus regularisation will force all $\theta_k^{(j)} = 0$, $\forall k = 1, ..., n$. This means that every single rating predicted for this customer will be 0! This is not ideal because clearly the customer will like some of the movies on offer. It seems more reasonable to give such a customer a rating which is an *average* of all possible ratings (e.g. 3 in a 1 to 5 star ratings system).

This problem is easily solved by applying **mean normalisation**, i.e. subtracting from all $y^{(i,j)}$ the average rating so that the resulting ratings are centred around zero:

$$\hat{y}^{(i,j)} = y^{(i,j)} - \mu_y \tag{49}$$

For instance, given a system with ratings from 1 to 5, the average is $\mu_y = 3$ and therefore $\hat{y}^{(i,j)} = [-2, -1, 0, 1, 2]$. Obviously, now the predicted ratings should also be amended by the average, as

$$\text{predicted } y^{(i,j)} = (\theta^{(j)})^T x^{(i)} + \mu_y \tag{50}$$

This approach will result in an algorithm which treats new customers (with no ratings) as an average customer, with predicted movie ratings equal to μ_y, rather than 0.

3 Unsupervised Learning

In unsupervised learning we do not have the desired outputs for our input variables x. Our training set is just made out of the input features:

$$\text{Training set } = \{x^{(1)}, x^{(2)}, ..., x^{(m)}\} \tag{51}$$

The best we can hope from the model is to get non-trivial insights into the input data, helping us make sense of it by finding patterns that would otherwise remain undetected. Most generally, unsupervised algorithms enable us to *cluster* the data into categories, finding commonalities, trends, etc. Unsupervised algorithms are common in astronomical analysis, market segmentation analysis, social networking, financial crime detection, etc.

Beyond looking at unsupervised learning techniques, in this section we will also explore methods to manipulate data in order to improve its quality, help with its visualisation or simplify the learning algorithms.

3.1 K-means clustering

The most common method of unsupervised learning is K-means clustering. In this algorithm, the user selects how many clusters or categories exist in the data. This is denoted by K. Each cluster has a center point, or **centroid**, and the algorithm aims to allocate every input point x to a cluster. If $x^{(i)} \in \mathbb{R}^n$ (there is no bias term in unsupervised learning), the centroids are denoted as

$$\text{Centroids } = \{\mu_{1,0}, \mu_{2,0}, ..., \mu_{K,0}\}, \quad \mu_{j,l} \in \mathbb{R}^n \tag{52}$$

So how are these centroids chosen? This is precisely the goal of the training algorithm. Initially, the centroids are chosen at random, and then the learning iterative process is as follows:

- Compute the distance from each point $x^{(i)}$ to each centroid j (l denotes the iteration):

  $$d_{j,l}^{(i)} = \parallel x^{(i)} - \mu_{j,l} \parallel_2^2 \tag{53}$$

  Each point is then allocated to a cluster k, chosen to be the one with the closest centroid, i.e. $x^{(i)} \in \text{cluster } k \iff k : \min_j\{d_{j,l}^{(i)}\}$.

- We recompute the centroids as the geometrical center of all the points in each cluster:

  $$\mu_{j,l+1} = \frac{1}{m_{k,l}} \left(\sum_{i:x^{(i)} \in k}^{m} x^{(i)} \right) \tag{54}$$

  where $m_{k,l}$ is the number of examples in cluster k in iteration l.

This process will converge to a final set of centroids. If any of the centroids has no points attached to it, it's ok to drop it. For cases when the points are not very clearly clustered, the algorithm will still succeed, but cluster boundaries may be very close to the actual points and clusters won't be clearly identifiable.

Although we do not need an objective function for our training algorithm, we can write a cost function to minimise. This function is sometimes called a **distortion** function. For a given iteration l, and denoting as $c^{(i)}$ the cluster assigned to point $x^{(i)}$ on that iteration, the cost function reads

$$J_l(c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K) = \frac{1}{m} \sum_{i=0}^{m} \parallel x^{(i)} - \mu_{c^{(i)},l} \parallel_2^2 \tag{55}$$

Note that both of the training algorithm steps help minimising this function: the cluster assignment step minimises $J(\cdot)$ by changing $\mu_{c^{(i)},l}$ while keeping the $\mu_{k,l}$ unchanged, while the centroid recalculation step minimises $J(\cdot)$ by changing the μ_k while keeping the $c^{(i)}$ unchanged. Following these steps consistently ensures a monotonically decreasing $J(\cdot)$.

3.1.1 Randomisation of $\mu_{k,0}$

The initial set of centroids can be initialised by picking K of the training set points $x^{(i)}$. Obviously one expects to have $K < m$, so there should be plenty of points to choose from. Once the points are selected, run the K-means algorithm until convergence. Of course the algorithm may end up in a local minimum, which is why it is recommended to repeat the process multiple times with different randomly selected $\mu_{k,0}$. The final solution, given by $\{c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K\}$, can then be chosen to be that with the lowest final $J(\cdot)$.

3.1.2 Selecting K

What is more challenging is to select the right number of clusters, K. In some cases it's very clear to the user how many clusters are needed (e.g. an algorithm to find out which should be the ranges for T-shirt sizing may want 3 categories from the onset, namely *small* (S), *medium* (M) and *large* (L)), but in other cases this is not so clear cut. The best approach is then to try multiple values of K and see what the convergence value of $J(\cdot)$ looks like as a function of K. One expects $J(\cdot)$ to always decrease with K and asymptotically tend to zero (in the limit, where $K = m$, $J(\cdot) = 0$), but the rate of decrease will slow down with K. A good value for K is that where the rate of decrease of $J(\cdot)$ slows down, which should be visible as a *kink* in the graph. Figure 8 depicts this scenario.

3.2 Dimensionality reduction and PCA

Another common way to gain insights from a data set is to perform what is called **dimensionality reduction**. As the name indicates, this is the process of reducing n, where n is the number of input features in each point $x^{(i)}$. Although it may not be immediately apparent, some of the features in the data may be redundant (e.g. having the same feature both in miles and kilometres) or possibly capture highly correlated quantities (e.g. having two input values denoting the GDP per capita and the mean household income). Dimensionality reduction is the process to remove redundant data from our input set, in order to compress the data, speed up calculations or aid with its visualisation, converting the data from $\{x^{(i)}\} \in \mathbb{R}^n$ to $\{z^{(i)}\} \in \mathbb{R}^{k<n}$. For example, reducing a $n = 3$ data set to another $n = 2$ data set, by collapsing one of the dimensions, will allow us to plot the data in 3D.

Principal Component Analysis, or *PCA*, is the most common dimensionality reduction technique. The objective of PCA is simple: reduce your data from n to k dimensions ($k < n$), by finding k unit[3] vectors $\{u^{(1)}, u^{(2)}, ..., u^{(k)}\}$, onto which to project the data, that minimise the

[3]A unit vector u is a vector with unit norm, i.e. $\parallel u \parallel_2 = 1$.

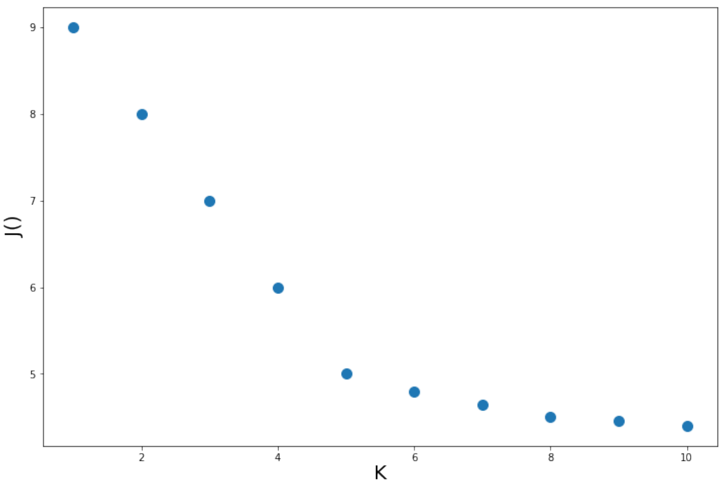

Figure 8: Convergence value of $J(c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K)$ as a function of the number of centroids K. Note that $J(\cdot)$ should always decrease with increasing K, but it's common to find a *kink* in the curve. A good value for K is where the kink is found, 5 in our example.

projection error. Note that k unit vectors $\{u^{(i)}\}$ uniquely identify a k-dimensional surface.

When we reduce the dimension of our data set, we are projecting our original points onto a lower dimensional space determined by those vectors $u^{(i)}$. PCA's algorithm tries to minimise by how much we need to *move* our original points to place them onto the k-dimensional surface. Please note that PCA has nothing to do with linear regression: in linear regression we are trying to predict values for y, while in PCA we are not predicting anything.

To successfully perform PCA, we first need to rescale our data, by subtracting the mean from all our points and, when different features have different scales, by dividing by their standard error to have comparable ranges of values (see sec. 2.1.1 for the mathematical details). Once that's done, the PCA algorithm is as follows:

- Compute the covariance matrix of the rescaled data, i.e.

$$\Sigma_x = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \cdot \left(x^{(i)}\right)^T, \quad \Sigma_x \in \mathbb{R}^{nxn} \tag{56}$$

- Through **Single-Value Decomposition** (or SVD) of the covariance matrix, compute the *eigen*-vectors of the matrix and select the first k (which are the k most relevant). The SVD calculation decomposes the covariance matrix as the product of three other matrices

$$\Sigma = U \cdot S \cdot V \tag{57}$$

where the columns of U hold the *eigen*-vectors of Σ

$$U = \begin{pmatrix} \vdots & \vdots & \dots & \vdots \\ u^{(1)} & u^{(2)} & \dots & u^{(n)} \\ \vdots & \vdots & \dots & \vdots \end{pmatrix} \in \mathbb{R}^{nxn} \tag{58}$$

The new projected points $z^{(i)}$ are obtained by multiplying the reduced matrix U_k (the first k columns of U, i.e. a nxk matrix) by the original points $x^{(i)}$:

$$z^{(i)} = U_k^T \cdot x^{(i)}, \quad z^{(i)} \in \mathbb{R}^k \tag{59}$$

3.2.1 The choice of k

Of course, the projection of the original data will cause the loss of some of the information in the $x^{(i)}$ points, since we have projected them onto a lower dimensional space. The inverse process, therefore, will not recover the $x^{(i)}$ points but an approximation of those points instead:

$$\tilde{x}_k^{(i)} = U_k \cdot z^{(i)}, \quad \tilde{x}_k^{(i)} \in \mathbb{R}^n \tag{60}$$

The values of $\tilde{x}_k^{(i)}$ are useful to help us decide what is the most appropriate k to pick. There is no right or wrong answer when choosing k: the closer k gets to n, the closer $z^{(i)}$ will get to the original $x^{(i)}$. As we make k smaller, more of the original information is lost. So the way to choose k should be based on how much of the original information we want to retain after the projection. One way to measure this is through the average squared projection error, δ_k namely

$$\delta_k = \frac{\frac{1}{m}\sum_{i=1}^{m} \parallel x^{(i)} - \tilde{x}_k^{(i)} \parallel_2^2}{\frac{1}{m}\sum_{i=1}^{m} \parallel x^{(i)} \parallel_2^2} \tag{61}$$

where the denominator is colloquially denoted as the *variance* of the original data. With this definition, we then select the smallest value of k that guarantees $\delta_k < \epsilon$, where ϵ is the desired upper bound for the error. δ_k is thus a measure of the variance lost during the projection from n to k dimensions. A standard value for ϵ is 1%.

3.2.2 Advice when applying PCA

Once we have computed the $z^{(i)}$, we can then proceed to utilise these projected data to train a supervised or unsupervised algorithm, or to visualize the data. Note, however, that PCA should only be run on the training set (i.e., the SVD should be run on the covariance matrix of the training data only).

PCA is also not a solution for overfitting. Indeed, PCA does not make use of the output values $y^{(i)}$, so it can't possibly help with fitting it better. In other words, PCA is not a way to reduce the number of features, but to capture as much information as possible from the original features with the least number of inputs possible. To solve overfitting, use regularisation instead.

Finally, it is worth pointing out that PCA should not be used by default: if your computational power and memory availability can handle the original data, it's always best to try with the raw data first, avoiding the added complexity of running PCA. Only if that fails, then should PCA be implemented, and the $z^{(i)}$ used instead for training.

3.3 Anomaly detection

Anomaly detection, or outlier detection, is the process of identifying data points that lie far from the average. These points may skew your model results in undesired ways, so it is useful to remove these points from the data set before training your algorithm. In other cases, the problem we are trying to solve is in fact the detection of these point, e.g. financial crime and fraud detection, machinery malfunction detection, etc.

As usual, we start with our training data set:

$$x = \{x^{(1)}, x^{(2)}, ..., x^{(m)}\} \tag{62}$$

Each of the $x^{(i)}$ is a vector of n features. The question we are trying to solve is how can we identify which points $x^{(i)}$ are anomalies? A simple way to do this is to use the probability density for the data points: if we know what is the probability of a certain point to appear in the set, $p(x^{(i)})$, we can consider as anomaly any point with $p(x^{(i)}) < \epsilon$. Of course, the probability density function (or *PDF*) for the data is not generally known, but we model it by trying to calibrate well-known distributions (e.g. Gaussian) to the data. Even if the match is approximate, chances are that it is good enough for anomaly detection.

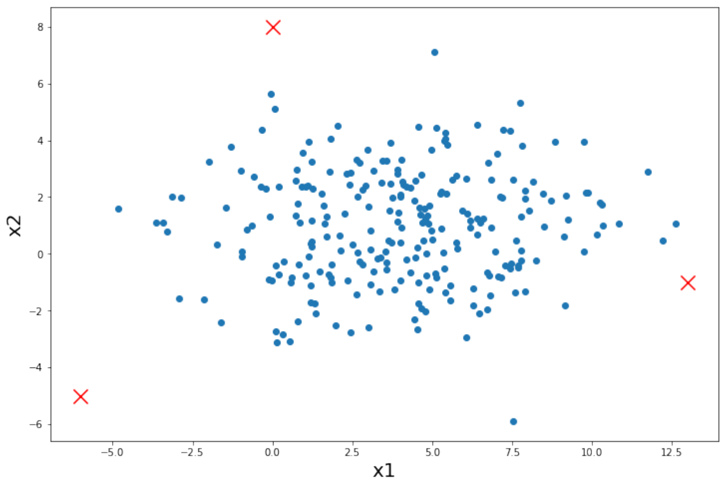

Figure 9: Anomaly detection is the process of identifying points within the data set that fall far from the data set average. In this example, the red crosses are candidates for being anomalies within the data set.

3.3.1 Using Gaussian distributions

One of the most common probability distributions is the Gaussian distribution, denoted $\mathcal{N}(\mu, \sigma^2)$, with mean μ and variance σ^2. In this distribution, the probability density of a given value x is given by

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{63}$$

Figure 10 shows this well-known distribution, for zero mean and unit variance. The calibration of the mean and variance of the distribution for a set of values $\{x^{(i)}\}$ is computed as per usual:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \tag{64}$$

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu)^2 \tag{65}$$

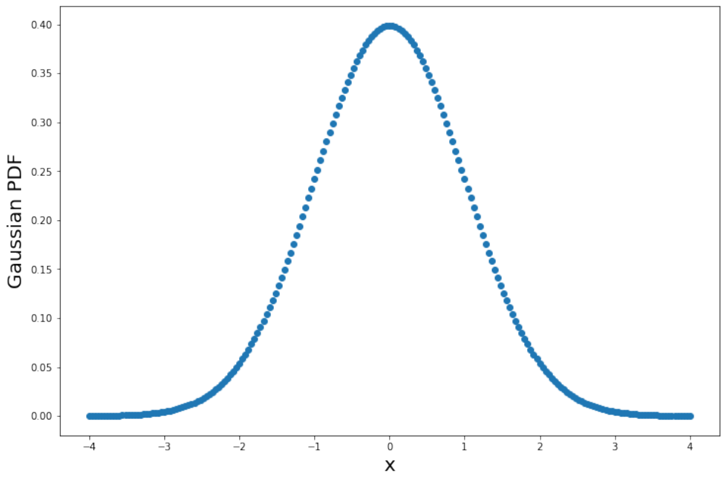

Figure 10: One-dimensional Gaussian PDF for $\mu = 0$ and $\sigma^2 = 1$.

We can consider as anomalies any points that are far on the tails of the distribution, which are those with $p(x; \mu, \sigma^2) < \epsilon$, for a small value of ϵ. For data sets with multiple features, the PDF is just the product of all individual PDFs:

$$p(\{x_1, x_2, ..., x_n\}) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) \tag{66}$$

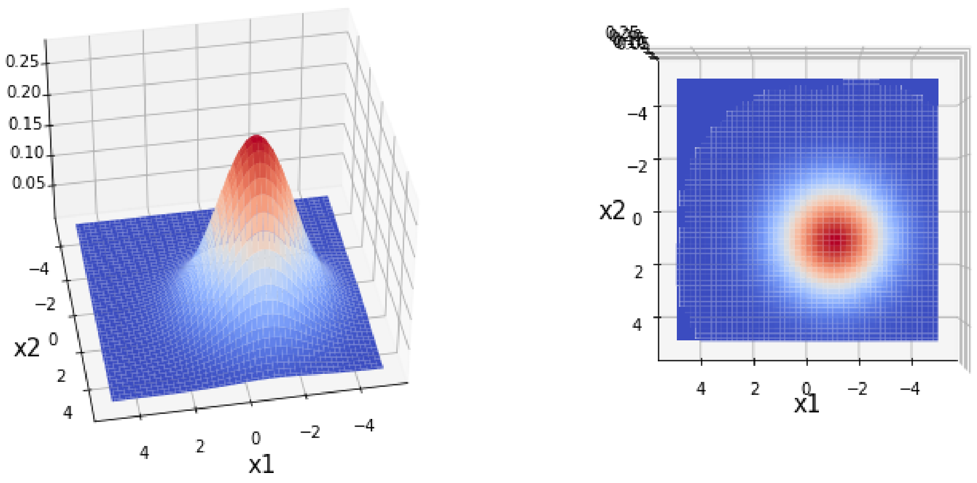

Figure 11: Two-dimensional Gaussian PDF for $\mu = \{1, -1\}$ and $\sigma^2 = \{2, 2\}$.

Figure 11 shows an example of a 2D Gaussian PDF. Anomalies are detected in exactly the same way as before, namely by finding points with $p(\{x_1, x_2, ..., x_n\}) < \epsilon$. Of course, in the above we have made a couple of important assumptions:

- All features x_j fit to a Gaussian PDF. When this is not always the case.
- All features are independent from each other, i.e. there is a zero correlation between them.

However, none of these are necessarily obvious, or even common. Let us go through both of them in more detail and see if we can overcome them.

When the distribution of points for a given feature does not fall on a Gaussian distribution, we can explore mathematical transformations to obtained features that do. For instance, we can take the logarithm, the exponential, square its value, etc. in the hope that we will end up with a distribution resembling a Gaussian (see fig 12 for an example). In any case, a perfect match is not required, and an approximated match will suffice.

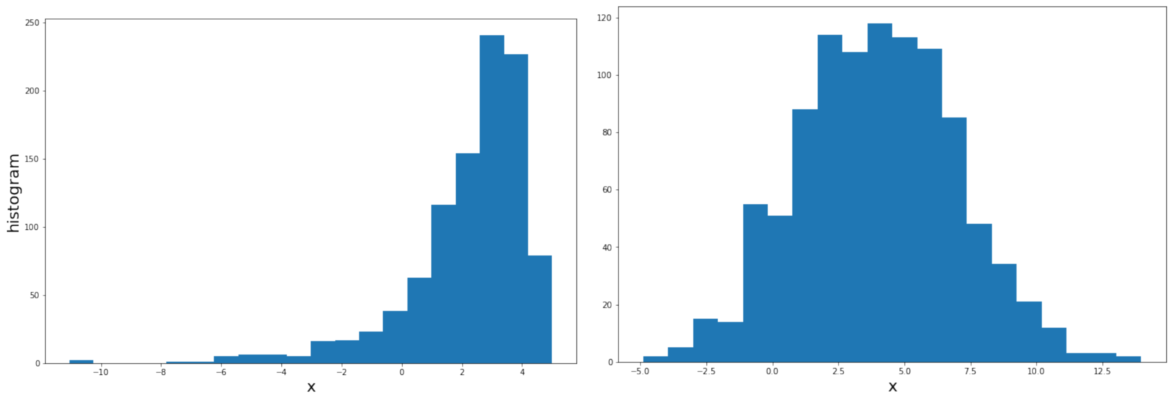

Figure 12: An example of how to convert a non-Gaussian distribution into an approximated Gaussian distribution. The left plot shows the histogram of a particular feature x, clearly non-Gaussian. However, the histogram looks a lot closer to a Gaussian when we plot the histogram for $log(x^2)$.

The second approximation (feature independence) is much harder to fake. When the features are correlated, the model we have proposed does not work and, instead, we should use a **multivariate** Gaussian distribution, which takes correlation into account. Given feature input data points $x \in \mathbb{R}^n$, we need to compute the mean array and covariance matrices, $\mu \in \mathbb{R}^n$ and $\Sigma \in \mathbb{R}^{nxn}$ (see equation 56 for a mathematical definition of covariance). Once these have been computed, the PDF can be written as

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) \tag{67}$$

where $|\Sigma|$ is the determinant of the covariance matrix. One can easily show that when the covariance matrix Σ is equal to the identity matrix we recover the feature independence case shown in eq. 66. The mathematical expression in the exponent contains all the cross terms, weighted by the covariance terms in the matrix. These terms allow for distributions which are not symmetrical with respect to any of the n dimensions in the feature input data points x, as shown in figures 13 for a 2D multivariate Gaussian distribution.

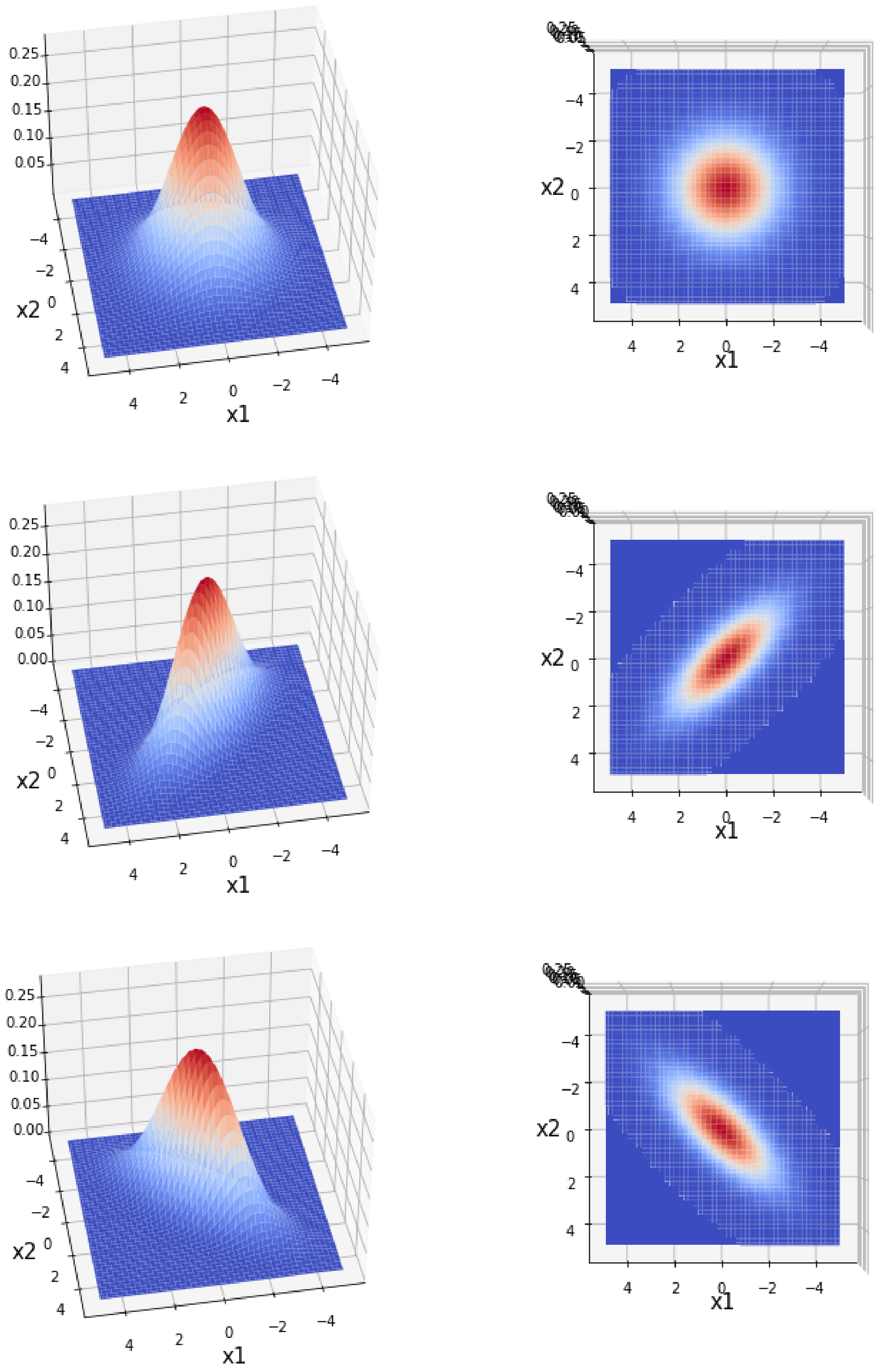

Figure 13: Examples of two-dimensional multivariate Gaussian distributions. In all of them the mean is $\mu = \{0, 0\}$. The covariance matrices Σ are $\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$ (top), $\begin{pmatrix} 2 & 1.5 \\ 1.5 & 2 \end{pmatrix}$ (middle) and $\begin{pmatrix} 2 & -1.5 \\ -1.5 & 2 \end{pmatrix}$ (bottom).

3.3.2 Training an anomaly detection algorithm

Although we have been looking in this section at how to identify anomalies within a data set, note that we could use this information to train a supervised learning model that detects anomalies. We could proceed as follows:

- Select a training set *without* any anomalies, and calibrate the Gaussian distributions $p(x_i)$ for the points in it, obtaining μ_i and σ_i^2 in the process, or the covariance matrix Σ if the features are correlated.
- Select a different data set with some anomalies, and use it to select a value of ϵ that successfully predicts these anomalies (this set us called the cross-validation set, see section 5.1).
- Use a third test (or testing set), also with anomalies, to test the quality of the model, checking that it predicts the anomalies correctly. In the case that anomalies are very rare, we need to use evaluation metrics appropriate for imbalanced data sets, such as the F_1 score (see section 5.3 for more on this).

Such a model can then be used to detect anomalies via inference for new data points, without the need to compute Gaussian probabilities, although a periodic recalibration of the model is advisable. The multivariate Gaussian model is better at capturing correlations between the features but it is more computationally expensive due to the matrix inversion involved.

The question that remains is how to decide whether to use a supervised learning model or the standard probability-based anomaly detection approach. The answer depends on the data: probabilistic anomaly detection is recommended when there is a small amount of anomalies, so that it would be hard for an algorithm to capture their characteristics. Also, we use the probability-based method when new anomalies may appear that are nothing alike the previous ones. Examples include fraud detection, data center monitoring or manufacturing defect detection. On the other hand, a supervised learning algorithm is recommended when there are large numbers of both valid and anomaly examples, so that the algorithm can get a sense of what anomalies look like, and where future new anomalies are similar to previous ones. Examples include spam detection, cancer diagnosis or weather prediction.

4 Large scale machine learning

Machine learning models work so well nowadays because of the huge computational power we have access to, but also because of the enormous amounts of data available. Ultimately, the best ML algorithms are not necessarily those with the smartest underlying models but those with the most data. The challenge then becomes how to manage the data and the computational cost of training such large systems.

4.1 Batch, stochastic and mini-batch gradient descent

Performing gradient descent for each θ_j involves a sum over all m examples:

$$\begin{aligned} h_\theta(x) &= \sum_{j=0}^{n} \theta_j x_j, \\ J(\theta) &= \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2, \\ \theta_j &:= \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}, \quad \forall j = 1, ..., n. \end{aligned}$$

When we have hundreds of millions of examples, summing over them becomes very computationally expensive. When the training is done using the full sum over all m examples it is called **batch** gradient descent. **Stochastic** gradient descent, on the other hand, updates the θ_j using 1 example at a time, iterating over all examples with an outer loop instead:

$$\text{for } i = 1, ..., m \quad \left\{\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}, \quad \forall j = 1, ..., n\right\} \tag{68}$$

Basically, we update each θ_j by a small amount at each step (i.e. the term of the derivative with respect to θ impacting $x^{(i)}$ only). This means that the cost function $J(\theta)$ may not decrease at each iteration step, but it should end up converging to the solution nonetheless. The goal of stochastic gradient descent is to simplify each update, although normally the loop over all m examples is needed between 1 and 10 times before convergence is reached. This convergence should be monitored, and α modified as iterations progress to aid convergence. Rather than plotting $J(\theta)$, in stochastic gradient descent it makes more sense to plot the average of the previous N values of $(h_\theta(x^{(i)}) - y^{(i)})^2$ (1000, say), and see how this evolves over time. If it increases, the model is diverging and α should be reduced.

An intermediate approach between batch and stochastic gradient descent is **mini-batch** gradient descent, where b examples are used in each update step, making sure that all examples are used:

$$\text{for } k = 0, ..., \frac{m}{b} - 1 \quad \left\{\theta_j := \theta_j - \alpha \sum_{i=kb}^{(k+1)b-1} (h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}, \quad \forall j = 1, ..., n\right\} \tag{69}$$

One of the benefits of mini-batch gradient descent is that it allows for a certain amount of vectorisation, as for batch gradient descent, without having to sum over all examples. The value of b can be anything between 1 and a small percentage of m.

4.2 Online learning

We denote as **online** learning the calibration of machine learning models that occurs perpetually, consuming new data points as they become available in real time. Online learning is used when we are interested in the real time changes in customer behaviour, say, or when the data comes in constantly and we want to capture any new features as they manifest themselves in the data. In such cases, models are being recalibrated with each point that becomes available. Given a new data point $(\hat{x}, \hat{y})$, we recalibrate the model as

$$\theta_j := \theta_j - \alpha(h_\theta(\hat{x}) - \hat{y})\hat{x}_j, \quad \forall j = 1, ..., n \tag{70}$$

Note that each data point is used just once and then thrown away. This is ideal when you have a constant stream of data, allowing you to have a model which is always up to date, and without the burden of having to store any training data set.

4.3 Machine learning pipelines

In real world applications, the tasks that require machine learning models are complicated enough to require multiple machine learning models applied in sequence. We denote this as a **machine learning model pipeline**. As an example, imagine a system that is able to identify text found in photographs. Such a system requires multiple models as part of the overall pipeline, namely:

- **Text detection**: a model which scans the photograph and identifies areas with text in them.
- **Character segmentation**: once we have detected the areas of the picture with text in them, we need to identify the characters, this model splits these areas into separate boxes with a single character in each of them.
- **Character recognition**: finally, this algorithm processes each of those single characters and identifies them.

These pipelines present the challenge that once the whole pipeline is in place, and assuming that the overall performance is not ideal, it is hard to know in which of the models it is best to spend our development time. A way to do this is by running a **ceiling analysis**, which compiles the overall accuracy of the system provided the first n components are working perfectly. In our example above, the ceiling analysis would be as follows:

- Current overall performance: 73%.
- Overall performance if text detection was perfect: 89%.
- Overall performance if text detection and character segmentation were perfect: 90%.
- Overall performance if text detection, character segmentation and character recognition were perfect: 100%.

It is clear from the analysis that we should spend time improving text detection and character recognition, but not much will be gained from improving character segmentation. The initial and final numbers in the analysis are easy to come by, while the intermediate values may require some manual intervention to *simulate* perfect model performance. Ceiling analysis is highly recommended in order to avoid wasting time improving models that will have small overall impact.

5 Model validation

5.1 Training, cross-validation and testing data sets

How do we know if our machine learning model is performing well? Rather than checking how well it reproduces the input examples in the training data set, what we do is to split all our data into two non-overlapping sets, the **training** and the **testing** sets. Then, we train the model using the training set by minimising $J_{train}(\theta)$ and check its quality on the testing set, which in effect tests the model on examples that the model has not seen before. The quality of the model is thus checked by computing the cost function on the test set, i.e. $J_{test}(\theta)$.

But given a machine learning model with sub-par performance, what should you focus on to improve it? You have many options:

- Get more training examples (increase m).
- Reduce the number of features (reduce n).
- Increase the number of features (increase n).
- Add polynomial features (x^2, $x^{1/2}$,...).
- Decrease λ.
- Increase λ.
- etc.

Some of these approaches may take some time to implement, so it makes sense to decide carefully. So it makes sense to try a few of them and see which one performs best. These models may differ on the number of features, the number of hidden layers or number of units per layer (in the case of neural networks), the value of λ, etc. So how do we decide which one is best? The main idea here is that the training set should not be used to select the model, as we may be selecting a model simply because it works well for that set. Equally, the testing set should not be used either as this is the set that is reserved to compute overall performance of the model. Therefore, the solution is to have a third set, the **cross-validation** set. In a nutshell:

- **Train**: train all your models using the training set.
- **Cross validation**: compute the error for those models using the cross-validation set, and select the model with the lowest error.
- **Test**: compute the quality of the selected model using the testing set.

A rough guide for the data set split would be 60%-20%-20%, but this is variable and dependent, for example, on how many data points you have access to in the first place.

5.2 Bias vs variance - learning curves

Here we define two important concepts:

- **Bias**: bias is a measure of how well a model performs on the training set. A low variance indicates a very good match of the training examples, and vice versa. Note that low variance does not mean that the model is good, as it may have overfit the training set.

- **Variance**: variance is a measure of how well a model performs on the cross-validation or testing sets. High variance means that the model is not matching the cross-validation or testing set data well, and vice versa.

These two concepts are very useful when diagnosing a problem with our model, and thus when deciding what to work on next. For example:

⇒ **Low bias and high variance is a symptom of model overfitting** of the training set.

⇒ **High bias and high variance is a symptom of model underfitting** of the training set.

⇒ **Low bias and low variance is a symptom of a good model**.

Plotting the cost function on the different data sets will provide valuable insights into which models is best. An example of this analysis can be seen in figure 14. In this graph we see the error for a series of linear regression polynomial models, both on the training and the cross-validation data sets. The error on the training set monotonously decreases with the degree of the polynomial, as higher degree polynomials match the data better and better. However, on the cross-validation set we can see that high polynomial degrees don't do so well for unseen examples. This indicates that for high-degree polynomials our algorithm is overfitting the training set. Separately, low polynomial degrees underfit the data in both sets. Crucially, the cross-validation curve reveals the optimal polynomial degree, around 5 in this case.

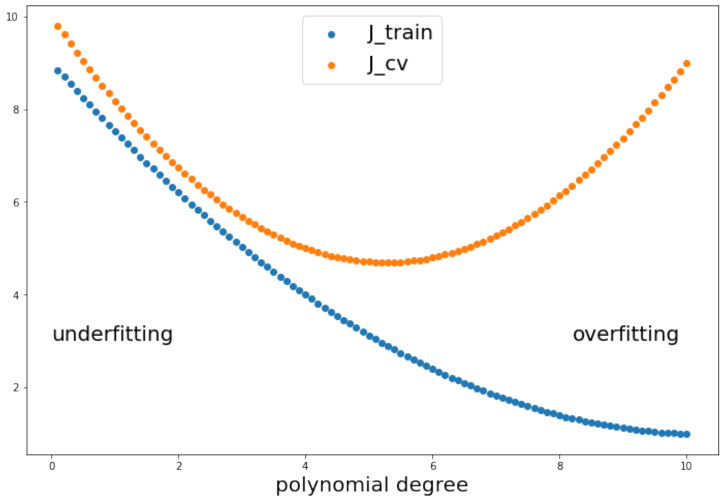

Figure 14: An example analysis to evaluate a series of models with different polynomial degrees. Low polynomial degree models display high bias and high variance models (underfitting), while high models with polynomial degrees show a low bias but high variance (overfitting). An intermediate polynomial degree ($p = 5$ in this case) has the overall best performance in the cross-validation set.

A similar picture would arise when changing the regularisation parameter λ, although the plot for the training set would be inverted, as overfitting would occur for low λ, and vice versa.

Another interesting plot to draw is the cost function for the training and cross-validation sets as a function of the number of examples m. This also helps diagnose whether we have a high bias or high variance problem in our model. High bias problems are shown by both cost functions approaching a high value asymptotically, as m gets large: in this case, the error does not go down with m and thus the model is not fitting the data well. On the other hand, high variance is represented by a low asymptotic value for the training set but a large value for the cross-validation set, meaning that the model fits the training data very well, but not so the cross-validation data. While high variance problems are likely to be solved by increasing m, high bias problems are unlikely to. Examples of these plots are shown in fig. 15.

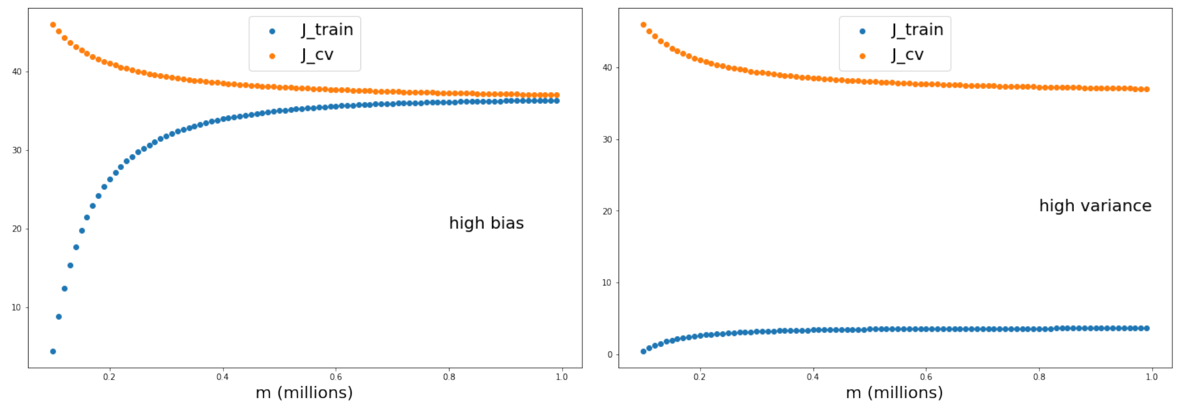

Figure 15: Learning curves as a function of the data set size, m. These examples correspond to high bias (left) and high variance (left) respectively. See main text for more detail.

Once we have plotted these learning curves, we are in a position to decide what to work on next to improve our model. Table 5.2 presents the most common solutions to the high bias and variance problems.

High bias	**High variance**
add more features	add more training examples
add polynomial features	remove features
decrease λ (non-SVM)	increase λ (non-SVM)
increase C (SVMs)	decrease C (SVMs)
decrease σ (Gaussian kernel)	increase σ (Gaussian kernel)

Table 2: Common solutions to the high bias and variance problems.

5.3 Imbalanced data sets

An imbalanced data set is a data set whereby the frequency of the different outputs is not comparable. In other words, some outputs are a lot more likely than others. For instance, if we are training a model to detect spam e-mail, we have an imbalanced data set because the vast majority of e-mails are not spam.

Imbalanced data sets need to be treated carefully because the standard measure for model quality do not work as one would expect. For instance, imagine a model that predicts cancer from a scan image. Let's imagine that the model has a 1% error *only*. This may sounds like a great model, but we really do not know how good it is because this depends on what is the proportion of cancer cases among the general population. Let's assume that it is 0.5%. This means that on a sample of 1 million people, 5,000 people have cancer. Our model would thus correctly detect 4,950 of those cancer cases (50 of them would be missed due to the 1% error on the model). Equally, the model would also incorrectly detect 9,950 cases of cancer (1% of 995,000 healthy individuals). Therefore, our model would predict 14,900 cancer cases, but only 33% of them (4,950/14,900) would actually be real cases of cancer! The model does not seem so good anymore.

	$y = 1$	$y = 0$
$h_\theta(x) = 1$	true positives	false positives
$h_\theta(x) = 0$	false negatives	true negatives

Table 3: True positives and negatives are cases when the model predicts the correct outcome, and vice versa.

This example shows that using the standard error measure on the test data set is not a good indicator of model quality. Instead, in such cases it is recommended to use **precision** and **recall** (see table 5.3 to see the definitions of these quantities). Assuming that the rare case is specified with $y = 1$ and the common case with $y = 0$. Then, precision P and recall R are defined as:

$$P = = \frac{\text{True positives}}{\text{Predicted positives}} = \frac{\text{True positives}}{\text{True positives + False positives}} \tag{71}$$

$$R = = \frac{\text{True positives}}{\text{Actual positives}} = \frac{\text{True positives}}{\text{True positives + False negatives}} \tag{72}$$

Note that $0 \leq P, R \leq 1$. For imbalanced data sets, the model needs to yield high values for both P and R for it to be considered good. For example, a cancer detection model that always predicts no cancer would have $P = R = 0$, thus making it clearly a bad model. For the model we used above as example, one can compute $P = 4,950/(4,950 + 9,950) \simeq 0.33$ and $R = 4,950/(4,950 + 50) = 0.99$: although recall is pretty good, precision is quite low.

There is generally a trade-off between precision and recall. Measures that increase one tend to decrease the other one. In our cancer detection model, if we err on the side of caution and make the model predict cancer more often, we will find more true positives, but the number of false positives will also go up. This will increase recall, but decrease precision. If, on the other hand, we change the model to avoid detecting false positives at the cost of missing true positives, we will increase precision but decrease recall.

Another question that arises is what are good values for precision and recall? When should we consider our model to be acceptable? The best way to assess this is to use a single numerical measure of quality, combining P and R into a single number. This number is called the F_1 score, and it is computed as the harmonic mean of P and R:

$$F_1 = \frac{1}{\frac{1}{R} + \frac{1}{P}} = \frac{2PR}{R + P} \tag{73}$$

The best model will be the one with largest F_1 score. A plot of the F_1 score as a function of R and P can be seen in fig. 16, alongside a plot of the arithmetic mean ($(R + P)/2$) for comparison: F_1 score is favoured over the arithmetic mean because it penalises more harshly models whereby either P or R are low.

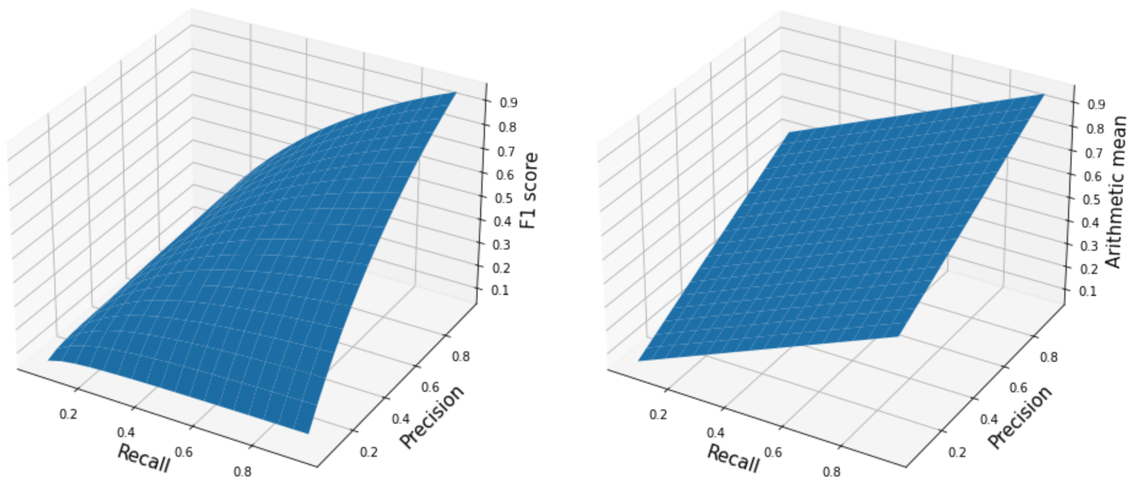

Figure 16: Harmonic (left, also called F_1 score) vs arithmetic (right) mean of recall R and precision P: the harmonic mean punishes more heavily cases when one of the two quantities is small.