These are notes based on the Stanford machine learning coursera course by Andrew Ng. They are not final yet and I will update them as I go through the course. These notes will be improved and material from outside the course will be added once I have finished the course.

1 Linear Regression

1.1 Gradient descent

Let $\theta = (\theta_0, \theta_1, \cdots, \theta_n)$ (column vector) be the parameters and $x^{(i)} = (x_0^{(i)}, \cdots, x_n^{(i)})$ (also column vector) be the features of the ith training example. So we have n features, and also we use m to denote the number of trainig examples. Our prediction of the outcome variable given the ith training example is now given by

$$\begin{aligned} \hat{y}(x^{(i)})_\theta &= \theta^T x^{(i)} \\ &= \theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)} \end{aligned} \tag{1}$$

where we chose $x_0 = 1$ so that θ_0 just becomes a constant contribution.

Generally, we have m samples $(x^{(1)}, y^{(1)}), \cdots, (x^{(m)}, y^{(m)})$ and n features, i.e.

$$x^{(i)} = \begin{pmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{pmatrix} \in \mathbb{R}^{n+1} \tag{2}$$

and we define $X \in \mathbb{R}^{m \times n+1}$ to be the matrix which contains all the features of all the training examples:

$$X = \begin{pmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(m)})^T \end{pmatrix} \tag{3}$$

Here $(x^{(1)})^T$ is a row vector, etc.. We can now write the linear model as

$$\hat{y}(X)_\theta = X\theta \tag{4}$$

which itself is a m-dimensional column vector, with each entry the prediction of a given training sample.

We now define the cost function as

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})^2 \tag{5}$$

where $y^{(i)} = (y_1^{(i)}, y_2^{(i)}, \cdots, y_m^{(i)})$ are the actual values. This gives the cost for a given training example $x^{(i)}$.

For updating our parametes using gradient descent, we use the following rule:

$$\theta_k \to \theta_k - \alpha\, \partial_{\theta_k} J(\theta) \tag{6}$$

$$= \theta_k - \alpha \frac{1}{m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})\, x_k^{(i)} \tag{7}$$

where α is the learning rate.

Let's write this in a more general form where $\hat{y}_\theta(x^{(i)})$ denotes our prediction given x and θ:

$$\theta_k \to \theta_k - \alpha \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_\theta(x^{(i)}) - y^{(i)})\, x_k^{(i)} \tag{8}$$

1.2 Feature Scaling

If one feature takes on much larger values than the other, then with e.g. two features get this very stretched gaussian kind of thing. So very thin in one direction, then it can take ages for gradient descend to reach the minimum. So we need to rescale the features e.g. using mean normalization. This ensures that all features more or less lie in the same range (it doesn't have to be exactly the same, it doesn't have to be exactly between 0 and 1, but just roughly all on the same scale). Mean normalization does the following:

$$x_i \to \frac{x_i - \mu_i}{s_i} \tag{9}$$

where $\mu_i = 1/n \sum_j x_j^{(i)}$ and s_i is the standard deviation of the $x^{(i)}$ vector , $j = 1 \dots n$ (n is the total number of samples we have of this feature $x^{(i)}$).

1.3 Choosing a learning rate

Can plot cost function as function of the number of iterations (Figure 1). Every iteration of gradient descent we update our parameter vector, and then for the new parameter vector we just evaluate the cost function which is then $J(\theta)$. As we continue with our gradient descent, we move to lower values of the cost so the line decreases and at some point converges (I guess we're not doing stochastic gradient descent because the line is always going down). Can look at this figure when you've done your gradient descent to see whether you converged properly. Can also set a convergent criterion (so e.g. if J changes by less than 10^{-3} then you're converged).

Of course a too small learning rate is also not great as the convergence is really slow. Plotting graphs like the ones below is good to see what's going on. Also, *just try a set of different learning rates, e.g.* $\alpha = \cdots, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, \cdots$ *and make a plot for each of them. Pick slightly smaller than largest reasonable value.*

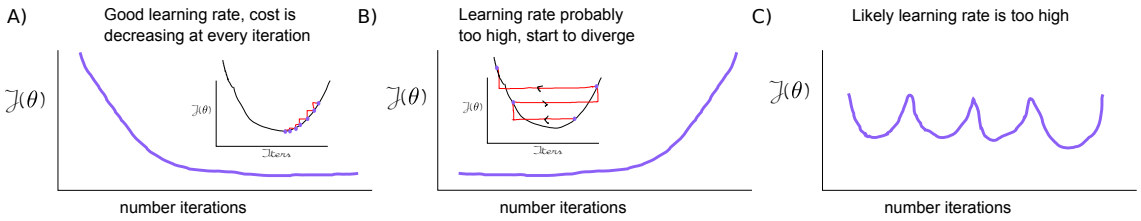

Figure 1: **Convergence** This is determinsitic gradient descent so if done properly the cost should always go down. **A)** The cost decreases at every iteration and the algorithm is sucessfully approaching the optimum. **B)** The cost is diverging and it is likely that this is caused by a learning rate that is too large. **C)** This oscillating behaviour of the cost as function of the number of iterations is also often an indicating of a too large learning rate.

1.4 Choosing features

It's important to choose good features. Suppose you want to sell a house and you have the width of the house and the depth of the house (so from front to back). Then could say $\hat{y}(x) = \theta_0 + \theta_1 \text{width} + \theta_2 \text{depth}$. But, could make a new feature, e.g. the area, so $A = \text{depth} \times \text{width}$. Then could do $\hat{y}(x) = \theta_0 + \theta_1 A$. This might be a better model.

1.4.1 Polynomial regression

Instead of linear might do something like $\hat{y}(x) = \theta_0 + \theta_1 x + \theta_2 x^2$ or do a third order one or whatever. Suppose we want our model to depend on the size of a house in a cubic way, so $\hat{y}(x) = \theta_0 + \theta_1 (\text{size}) + \theta_2 (\text{size})^2 + \theta_3 (\text{size})^3$. Then we can just define our features as $x_1 = (\text{size}), x_2 = (\text{size})^2, x_3 = (\text{size})^3$. *So can use linear regression to fit a cubic model to the data.*. Of course can do all kinds of things. It's good to think of what kind of shape you expect the data to be and then select features blah. Note that in this case feature scaling becomes really important, as e.g. x_3 in the example is the size cubed so if the size varies over a large scale this x_3 will vary over a huge scale, etc.. So be extra careful.

There are algorithms that automatically choose features for you, so that's good :).

1.5 Normal equation

The normal equation lets us solve for the optimal θ, so we don't even have to do gradient descent anymore.

Suppose the cost function is $J(\theta)$ and we have $\theta \in \mathbb{R}^{n+1}$. Then of course we just do

$$\partial_{\theta_j} J(\theta) = 0 \quad \forall j \tag{10}$$

and solve for all the θ_i.

In general, we can write X for a matrix of dimension $m \times n+1$, each rows is a training sample and each column is a feature. X was defined previously in Eq. (3). The value of θ which minimizes the cost function is:

$$\theta = (X^T X)^{-1} X^T y \tag{11}$$

where y is a vector with the actual observed values. So the structure of X is, e.g. if we have 3 features (x_0 doesn't count, it is just set to 1) and 4 samples $X = \begin{pmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ x_0^{(2)} & x_1^{(2)} & x_2^{(2)} & x_3^{(2)} \\ x_0^{(3)} & x_1^{(3)} & x_2^{(3)} & x_3^{(3)} \\ x_0^{(4)} & x_1^{(4)} & x_2^{(4)} & x_3^{(4)} \end{pmatrix}$ and $y = (y^{(1)}, y^{(2)}, y^{(3)}, y^{(4)})$ (a column vector).

Of course if we use the normal equation we don't have to rescale the features because we're not doing any gradient descent.

Why don't we just use $\theta = X^{-1}y$? Because X may not be invertible. One can easily show that the normal equation is equivalent to $\theta = X^{-1}y$ while not having to directly take the inverse of X. We see that $(X^T X)^{-1} X^T = X^{-1}(X^T)^{-1} X^T = X^{-1}$. It turns out that $X^T X$ is almost always invertible (at least it's a square matrix whereas X may not be).

1.5.1 When use the normal equation

With gradient descent, we need to choose the learning rate α which can be annoying. Also, we might need many iterations so it could be slow.

However, gradient descent works well with a large number of features (say, millions). The normal equation doesn't work well if n is large, it is expensive to calculate $(X^T X)^{-1}$, the matrix $(X^T X)$ is an $n \times n$ matrix. The cost of inverting the matrix goes as $O(n^3)$ so obviously for large n it is very slow to compute this $(X^T X)$ quantity. Gradient descent goes as $O(kn^2)$.

So, for small n use the normal equation, for large n do gradient descent. What is large? If $n = 1000$ still use normal equation, if n is around 10^5 or 10^6 then maybe start using gradient descent.

1.5.2 What if cannot invert?

$(X^T X)^{-1}$ might not exist if $(X^T X)$ is not invertible (i.e. it is singular/degenerate). *This apparently is pretty rare.* Even if it's not invertible, you can usually in programming languages still calculate $(X^T X)^{-1}$ numerically somehow.

If $(X^T X)$ is not invertible this could be caused by redundant features (e.g. one features gives length in meters and another feature gives length in feet, so these are completely dependent). Also, can happen if there are too many features (if e.g. $m \leq n$, try to fit many parameters with just a few training examples) and deleting some might work, or using regularization might work (Section 3.2.3) . Regularization allows you to have a lot of features even with few training examples.

2 Classification: Logistic Regression

Examples of classification problems are whether an email is spam or not, whether a tumor is benign or not, whether an image is a cat, dog, horse or goat, etc.. Logistic *regression* is a *classification* algorithm (slightly confusing).

2.1 Two classes

When we have two classes, the true output is either 0 or 1. We will choose our model such that our predictions always lie between 0 and 1.

We define

$$g(z) = \frac{1}{1+e^{-z}} \tag{12}$$

which is illustrated in Figure 2A. This is of course just a sigmoid. Now our prediction model is given by

$$\hat{y}(x)_\theta = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}} \tag{13}$$

Note that $\theta^T x$ doesn't have to be linear, we can define e.g. $x_3 = x_1^2$ (see later examples).

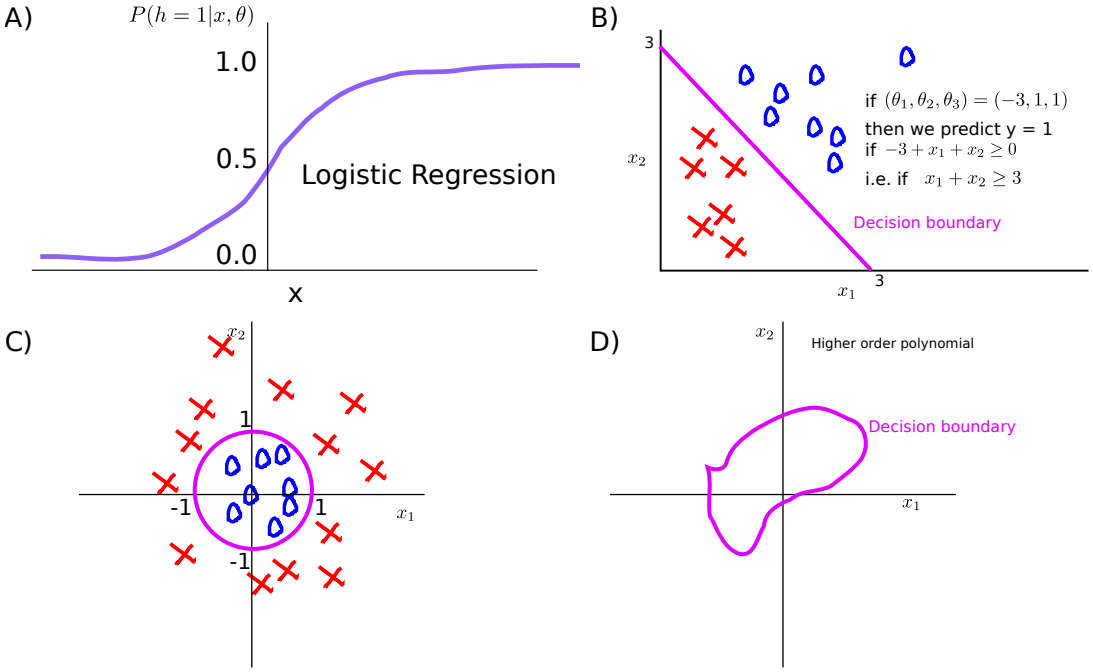

Figure 2: **Logistic Regression A)** Given some feature input x, we get the logistic thingie as an output and this gives us the probability that the output is 1. **B)** Decision boundary.

In a simple example for tumor being benign or not, the actual output is either 0 or 1. The output we get can be interpreted as the probability of the output being 1 (which we here choose to be malignant).

We can now decide to predict an outcome of '1' when $\hat{y} \geq 0.5$. This occurs whenever $\theta^T x \geq 0$.

2.1.1 Examples

Suppose we have $\hat{y} = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$ with data as shown in Figure 2B. Using $\theta = (-3, 1, 1)$ works in this exampe as it separates based on the line $x_1 + x_2 \geq 3$. This is exactly a perceptron (see later). The boundry defined by that line is called a *decision boundary*.

We can also say $\hat{y} = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$ with e.g. $\theta = (-1, 0, 0, 1, 1)$ in which case we will predict '1' if $x_1^2 + x_2^2 \geq 1$, i.e. we can separate the data shown in Figure 2C. Note that we don't really have to think much about the logistic curve itself, in the end the prediction just depends on whether $\theta^T x \geq 0$. The nice thing about the sigmoid is that we can then get probabilities out of it, so we might predict '1' but we can then say that we're e.g. 70% sure.

Higher order polynomials can give more complex decision boundaries (Figure 2D).

2.1.2 Fitting parameters

Given a training set $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$ and let $x^{(i)} \in \mathbb{R}^n, y^{(i)} \in \{0, 1\} \forall i$, our predictions are given by $\hat{y}(x^{(i)}) = 1/(1 + e^{-\theta^T x^{(i)}})$, then how do we choose the parameters θ?

We write the cost function as $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(\hat{y}(x^{(i)}), y^{(i)})$ with . Before, with linear regression, we had $\text{Cost}(\hat{y}(x^{(i)}), y^{(i)}) = \frac{1}{2}(\hat{y}(x^{(i)}) - y^{(i)})^2$. Let's for simplicity write this as $\text{Cost}(\hat{y}(x), y) = \frac{1}{2}(\hat{y}(x) - y)^2$. Can we still use this cost function? No.

When using that square cost function from our linear regression, the actual cost function becomes quite complicated because our $\hat{y}(x)_\theta$ is itself a sigmoid. It turns out that the cost function $J(\theta)$, when plugging the sigmoid into the quadratic equation, becomes non-convex. This means we can get stuck in local minima (Figure 3A, B).

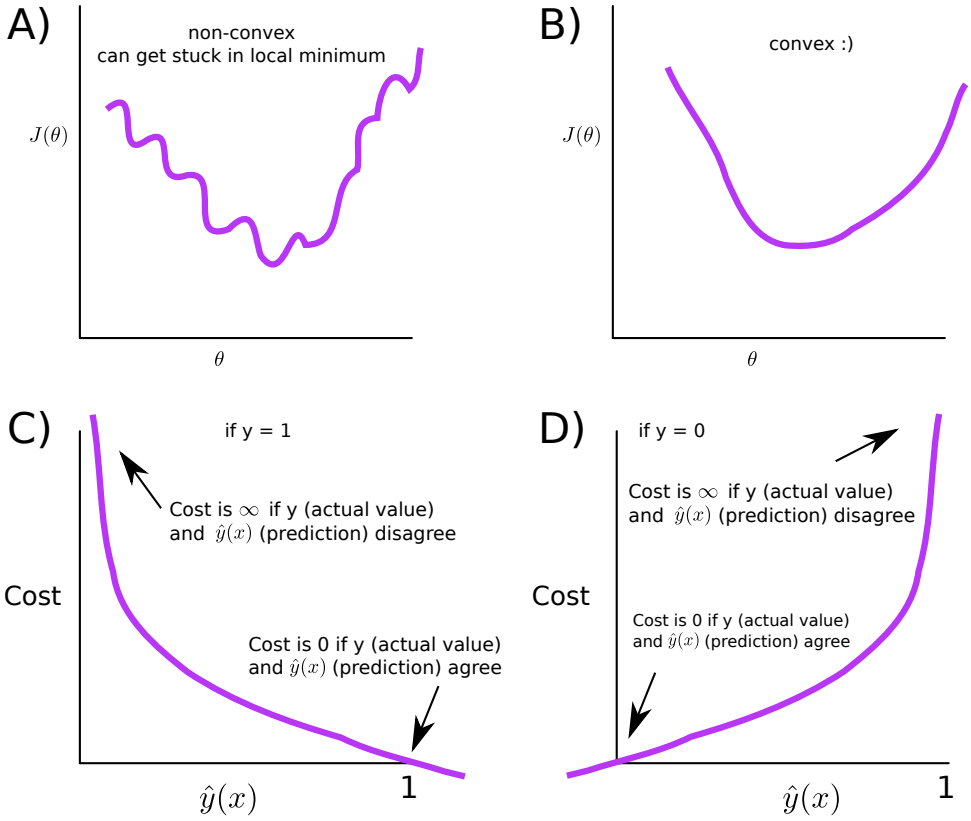

Figure 3: **Pick nice cost function A, B)** If we pick the same cost function as for linear regression, then our overall $J(\theta)$ as a function of θ will be quite complex and non-convex. This is not what we want, we want a nice looking $J(\theta)$. **C, D)** When picking the cost function as specified in Eq. (14), we get some nice properties.

What we do, is the following:

$$\text{Cost}(\hat{y}(x), y) = \begin{cases} -\log(\hat{y}(x)) & \text{if } y = 1 \\ -\log(1 - \hat{y}(x)) & \text{if } y = 0 \end{cases} \tag{14}$$

which is shown in Figure 3C, D. This example was just with a single training example. In general it works as well, the cost function $J(\theta)$ is convex and all is well. Remember that we had assumed that the actual output y is either 0 or 1, so this particular approach we outlined only works if there are 2 classes.

2.1.3 Simpler cost function

We can simplify the cost function given in Eq. (14), namely:

$$\text{Cost}(\hat{y}(x), y) = -y \log(\hat{y}(x)) - [1 - y] \log(1 - \hat{y}(x)) \tag{15}$$

This works of course because y is either 0 or 1 so one of the terms is zero in either case.

The cost function is therefore:

$$J(\theta) = -\frac{1}{m}\Big[\sum_{i=1}^{m} y^{(i)}\log(\hat{y}(x^{(i)})) + (1-y^{(i)})\log\Big(1-\hat{y}(x^{(i)})\Big)\Big] \tag{16}$$

Why this particular function? This cost function can be derived using maximum likelihood estimation, it also has the nice property that it is convex. Basically everyone uses this cost function when fitting logistic regression models. It's not as random as it seems here.

Now what we want of course is find the parameters θ that minimize $J(\theta)$, and then when we observe a new feature vector we can predict the outcome which is $\hat{y}(x) = 1/(1+e^{-\theta^T x})$ which is the probability that $y = 1$. Again, we use gradient descent like before. The rule we use is exactly the same as the one in Eq. (6), and after computing $\partial_\theta J(\theta)$ we get:

$$\theta_k \to \theta_k - \frac{\alpha}{m}\sum_{i=1}^{m}(\hat{y}(x^{(i)}) - y^{(i)})\, x_k^{(i)} \tag{17}$$

This is actually exactly the same equation as the one for linear regression (Eq. (8)). The only thing that has changed is the definition of $\hat{y}(x)$ itself.

Again, we should use feature scaling when implementing logistic regression, etc..

2.1.4 Beyond gradient descent

What we want: Find the best θ. Gradient descent keeps updating θ and if we also want to check whether convergence occurred (which means we want to keep track of the actual values of $J(\theta)$ as we go through the iterations) we have to have code that evaluates $J(\theta)$ and code that updates the θ values.

When having these two bits of code, other algorithms can also be used to find the optimal θ, e.g. conjugate gradient, BFGS, L-BFGS. Why would we want to use these? Usually we would not need to pick the learning rate α. These algorithms do something clever (a line search algorithm) and they automatically pick a learning rate α and they might pick a different one for every iteration (so adaptive). They often converge much faster than gradient descent. The disadvantage is that they are more complex. You shouldn't implement them yourself as they are written by people who actually know about numerical computing and they have been optimized to run quickly.

2.2 Multiple Classes

When we have n classes, we can divide the problem in n sets of 2-classes problems: this is called *One-vs-all*. We just take all data points belonging to e.g. class 0 and treat all the others points as belonging to a single other class. Now we have two classes (class 0 vs not class 0) and we can perform logistic regression to obtain the probability that any new data point belongs to our class 0. We can do the same for all the other classes (e.g. class 1 vs not class 1). This is illustrated in Figure 4.

When a new data point x is presented, we can now evaluate the following n probabilities: $P(x \in \text{class}\,0), P(x \in \text{class}\,1), \cdots P(x \in \text{class}\,n)$. The predicted class is now simply the one with the maximum probability.

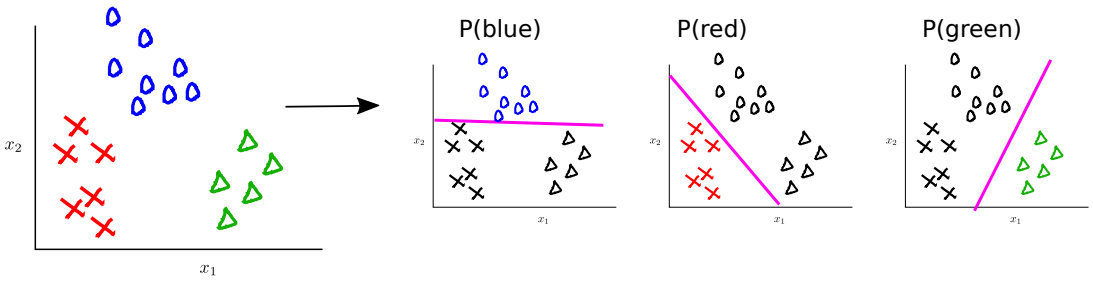

Figure 4: **Logistic Regression multiple classes** Take all the training examples belonging to a specific class and now treat all of the other data as belonging to a single different class. So here our original classes are red, blue and green, but we treat it as three classification problems: red vs the rest, blue vs the rest, and green vs the rest. Each of the individual classifications allows us to calculate the probability of belonging to that specific class when a new data point is presented.

3 Overfitting and underfitting

An underfitting model is said to have a *high bias*, which refers to the algorithm having a very strong bias to use a particular model (e.g. a linear one in Fig. 5) regardless of the actual data.

If we are overfitting we might be able to correctly predict each data point in our training set, but we have poor out-of-sample performance (Figure 5). A model that overfits is said to be *high variance*. It can almost fit any function, the space of possible hypotheses is too large, too variable.

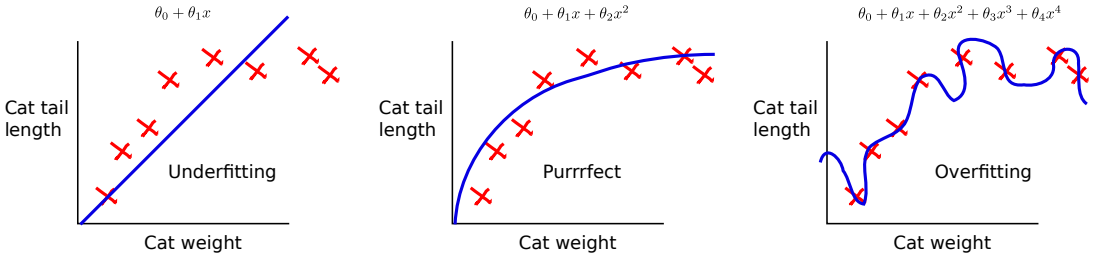

Figure 5: **Under- and overfitting** The plots speak for themselves.

3.1 How do we know we are overfitting?

One option would be to throw away some of the features, though this is not great because we're actually throwing away data. This is never good

Another option is regularization. We keep all the features but we reduce the magnitude of the parameters θ. This works well when we have a lot of features, each contributing a little bit to our prediction $\hat{y}$.

3.2 Regularization

Suppose we do have some 'complicated' linear regression model, i.e. $\hat{y} = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$ and we want to find the optimal θ. Using our normal cost function defined in Eq. (5), finding the optimal θ means minimizing $\sum_{i=1}^{m} (\hat{y}_\theta(x^{(i)}) - y^{(i)})^2$. Now what if instead we penalize large values for the parameters θ_3 and θ_4 by adding terms to the cost function. For example, we can find the θ that minimizing the following expression:

$$\frac{1}{2m} \sum_{i=1}^{m} \left(\hat{y}_\theta(x^{(i)}) - y^{(i)} \right)^2 + 1000\,\theta_3^2 + 1000\,\theta_4^2 \tag{18}$$

This will mean our model $\hat{y} = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$ becomes more or less a quadratic function because the optimal θ_3 and θ_4 will be very small (assuming 1000 is large compared to the actual x values used).

The idea is as follows: small parameters $\theta_0, \theta_1, \cdots, \theta_n$ values lead to a simpler hypothesis and the model becomes less prone to overfitting. Now one may wonder why making *all* parameters smaller still leads to a simpler hypothesis (clearly if we make the higher order terms smaller then yes our function is simpler, but if we make all of them smaller instead of specifically the higher order ones?), but this is true anyway. Just try.

3.2.1 Examples

Suppose we have 100 features, $x_1, x_2, \cdots, x_{100}$ and we have 100 parameters $\theta_1 \cdots \theta_{100}$. As it is, we don't even know which parameters multiply higher order features (it may be e.g. that $x_5 = x_1^3$) so we cannot penalize only higher order terms even if we wanted to. The way in which we will modify the cost function is:

$$J(\theta) = \frac{1}{2m} \left(\sum_{i=1}^{m} \left((\hat{y}(x^{(i)}) - y^{(i)})^2 \right) + \lambda \sum_{i=1}^{n} \theta_i^2 \right) \tag{19}$$

The last term is called the regularization term and it just adds a cost to having large values of θ. All values of θ are regularized except for θ_0 (that's a convention, it doesn't really make any difference). The parameter λ is called the *regularization parameter*. λ controls the trade-off between fitting the training

set well and keeping the parameters small. If λ is too small we might overfit, if λ is too large we will underfit as we fail to fit the training data (very high λ leads to a prediction of θ_0, so the 'bias' of the cat tail length being θ_0 is very high). Of course we know better, as baby kittens obviously have shorter tails than adult cats.

If we were to use regularization on Figure 5 then the graph on the right will look more to the one in the middle. It won't be completely quadratic, but the higher order terms will have a relatively lower contribution, it will basically become a wobbly version of the middle graph.

3.2.2 Regularized linear regression

Our cost function is given by Equation (19) and we update our parameters θ using the rule

$$\begin{aligned}\theta_k &\to \theta_k - \frac{\alpha}{m}\left(\sum_{i=1}^{m}(\hat{y}(x^{(i)}) - y^{(i)})\, x_k^{(i)} + \lambda\theta_k\right) \\ &= \theta_k\left(1 - \alpha\frac{\lambda}{m}\right) - \frac{\alpha}{m}\sum_{i=1}^{m}(\hat{y}(x^{(i)}) - y^{(i)})\, x_k^{(i)}\end{aligned} \tag{20}$$

This can be trivially shown by calculating $\partial_{\theta_k} J(\theta)$. Note that the above equations only hold for $k \neq 0$, when $k = 0$ the term with λ is dropped because θ_0 is not regularized.

The term $\left(1 - \alpha\frac{\lambda}{m}\right)$ is usually quite close to one (if e.g. many training examples m and a small learning rate α). So the effect is that instead of just θ_k as the first term (as in Eq. (6)) we use something like $0.99\theta_k$. So on every iteration the parameters θ_k are shrunk a little bit and then the 'normal' update term is added (i.e. the $-\alpha\sum$ etc..).

The normal equation also changes when we use regularization, it becomes:

$$\theta = \left(X^T X + \lambda \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}\right)^{-1} X^T y \tag{21}$$

The derivation of this is shown in XXX. The matrix is a $(n+1)\times(n+1)$-dimensional matrix.

3.2.3 Non-invertibility

Suppose that $m \leq n$, i.e. our number of training examples is less than the number of features. In this case, the matrix $X^T X$ is non-invertible. However, if $\lambda > 0$ then because we now take the inverse of $(X^T X + \lambda \cdot \text{Matrix})$ and that matrix *is* invertible. So regularization takes care of any non-intvertibility problems of $X^T X$!.

3.2.4 Regularized Logistic Regression

Our non-regularized cost function for logistic regression is provided in Eq. (16), the regularized cost function has an additional term which is exactly the same as the term we added for linear regression, i.e. $\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$. So again, all we do is add a term which penalizes large parameter values. The update rule for θ is again given by Eq. (20), with the only difference that now $\hat{y}(x^{(i)})$ is a logistic function whereas for linear regression is was a linear function (just like Eqs. (8) and (17) were the same). This of course makes a lot of sense as we wrote things down in a general way.

4 Neural Networks

Why do we want to use neural networks? In many problems, e.g. when recognizing images, the number of features can be very large. For example, a 200 by 400 pixel image already has $200 \times 400 \times 3 = 240000$ features (the 3 because of RBG), and that's only the actual pixel values themselves. If we want to fit a model that is nonlinear, we would need to take nonlinear combinations of features, e.g. $x_1 x_2, x_1 x_2^2, x_3^2$ etc.. (which is what we did to be able to separate the classes shown in Figure 2C. With so many features, linear or logistic regression is not a good option.

Neural networks were developed to simulate actual neural networks in the brain. Figure 6A,B shows a cartoon of actual neurons and their (basic) neural network representation. This figure just shows one layer, but we can have more (Figure 6C). Anything that is not an input or output layer is called a 'hidden layer'.

As shown in the figure, the notation $a_i^{(j)}$ denotes the activation of unit i in layer j. This is just the output value of that particular unit and goes along the axon to neurons in the next layer where the signal is combined with other incoming signals (in a weighted manner, as the synapses carry weight) and then this combined weighted signal goes through the activation function of these next-layer neurons. Each of the neurons in the next layer has its own weights so even though they all receive the same inputs, their outputs will generally be different.

The activations in Figure 6C are calculated as follows:

$$\begin{aligned} a_1^{(2)} &= g(\theta_{10}^{(1)}x_0 + \theta_{11}^{(1)}x_1 + \theta_{12}^{(1)}x_2 + \theta_{13}^{(1)}x_3) \\ a_2^{(2)} &= g(\theta_{20}^{(1)}x_0 + \theta_{21}^{(1)}x_1 + \theta_{22}^{(1)}x_2 + \theta_{23}^{(1)}x_3) \\ a_3^{(2)} &= g(\theta_{30}^{(1)}x_0 + \theta_{31}^{(1)}x_1 + \theta_{32}^{(1)}x_2 + \theta_{33}^{(1)}x_3) \\ \hat{y}_\theta(x) = a_1^{(3)} &= g(\theta_{10}^{(2)}a_0^{(2)} + \theta_{11}^{(2)}a_1^{(2)} + \theta_{12}^{(2)}x_2 + \theta_{13}^{(2)}x_3) \end{aligned} \tag{22}$$

with $g()$ the activation function which can e.g. by the sigmoid we used previously, $g(z) = 1/(1+e^{-z})$. In this example, $\theta^{(1)} \in \mathbb{R}^{3\times 4}$ and $\theta^{(2)} \in \mathbb{R}^{1\times 4}$ meaning we have a single output neuron which itself received four inputs. The indexing is perhaps a little confusing as terms like θ_{00} don't really exist, which is just because the 0 units don't have an input themselves, they are biases and their output is always 1. In general, if a network has s_j units in layer j, and s_{j+1} units in layer $j+1$, then $\theta^{(j)} \in \mathbb{R}^{s_{j+1}\times(s_j+1)}$ (note that the 0 units do not count as a unit when counting the number of units in layer j, so in the example in Figure 6C we have three units in both layer 1 and 2).

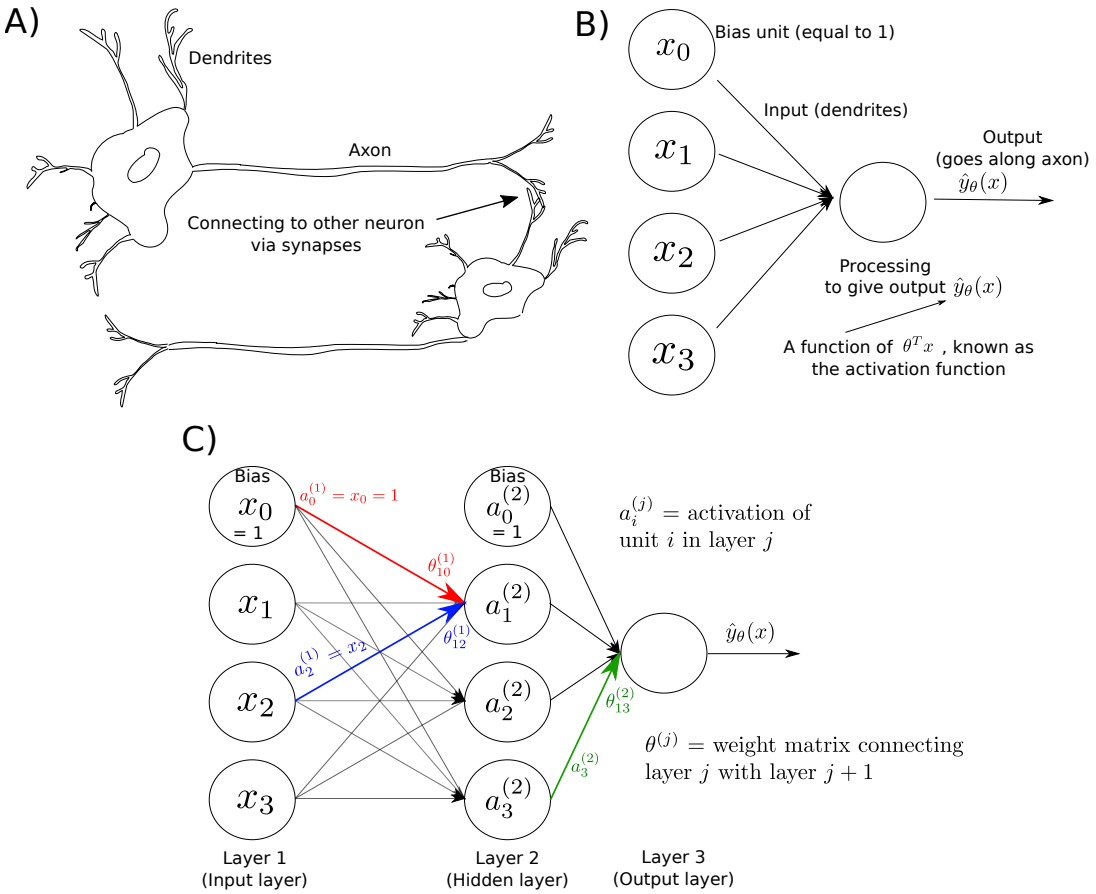

Figure 6: **Neural network representation A)** A cartoon of actual neurons. **B)** The basic units of a neural network and their relation to actual neurons. **C)** An example of a neural network with a single hidden layer and a single output neuron.

We can write the above example in a more compact form, namely

$$a^{(2)} = g(\theta^{(1)}a^{(1)})$$
$$a^{(3)} = g(\theta^{(2)}a^{(2)}) \tag{23}$$

so we get

$$a^{(i)} = g(\theta^{(i-1)}a^{(i-1)}) \tag{24}$$

4.1 Some intuition

The output neuron does its processing in a similar way as logistic regression, in inputs are now not x and θ but the activations $a^{(2)}$ and θ. So it's like the neural network still does logistic regression (as its final answer basically) but it chooses its *own* features to use rather than simply using the input values x. You can get some complex features (i.e. complex combinations of the x values) by having these θ matrices. The more hidden layers, the more weird combinations and weighting etc.. we can make from the intputs x. So neural networks 'create their own feature vectors' on which they then perform logistic regression (well, if we use the sigmoid as the final activation function). Of course the output layer can be more than just a single neuron, but the intuition remains the same.

4.2 Forward propagation

4.3 Back propagation

5 How to improve

If you've implemented a machine learning algorithm, you've trained it but then when you test it on other samples (which were not in your training data) it turns out to perform very badly. What do you do?

There are diagnostic tests you can run, which can prevent you from wasting time on trying to improve your algorithm in a way that isn't fruitful. For example, sometimes more trainig data does not help and you are wasting time by trying to collect more data. A diagnostic test may tell you in advance that finding more training data is unlikely to be helpful.

5.1 Training, (cross) validation and test sets

You might need to chose whether to fit a linear function, quadratic, cubic, etc.. when you are using e.g. linear or logistic regression. One could fit all models to the training data to obtain, for each model, parameters $\theta^{(d)}$ were d denotes the degree of the polynomial that was used to get that parameter estimate.

One can then calculate the error of the models on a test set, and pick the model with the lowest test set error. Now we want to ask how well this picked model generalizes. To do this, one can NOT look at the error on the test set, because the value of d was chosen already using the test set data. The test set error is likely to be an overly optimistic estimate of the generalization error. This is why we split in training, validation and test (instead of just training and test). It's typical to divide the data (0.6, 0.2, 0.2) between training, validation and test sets.

To select our polynomial degree, we use the validation set to select the d with minimum error (so minimum validation error). Then to get the generalized error of that particular polynomial we use the test set.

5.2 Over- and underfitting

As shown in Figure 5, we might overfit (high variance) or underfit (high bias) our model. From Figure 7 we see that we are likely to be *underfitting* if $J_{\text{train}}(\theta) = \text{high}, J_{CV}(\theta) \approx J_{\text{train}}$ (where $J_{CV}(\theta)$ is the error using the cross-validation set). We are likely to be *overfitting* if $J_{\text{train}}(\theta) = \text{low}, J_{CV}(\theta) \gg J_{\text{train}}(\theta)$

5.3 Choose the regularization parameter λ

One can use the same method we used for determining what degree polynomial to use. Basically, use the model with the lowest validation error.

Create some range of values of λ, e.g. $\lambda = 0, 0.01, 0.02, \cdots, \approx 10$. For each of these models, obtain the parameter fit θ. Using these fitted θ values, calculate the errors using the cross validation set and

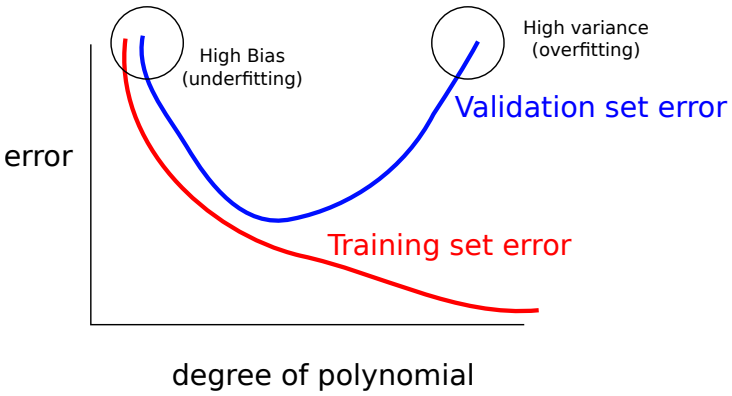

Figure 7: **Diagnostic for over- and underfitting** By comparing the errors from the training and validation sets, one can determine whether it is likely that we're over- or underfitting.

pick the model with the lowest cross-validation set error. We can then obtain the test set error of that particular model to get an idea of the out-of-sample performance.

When plotting the training and validation errors against λ, we get a something similar to Figure 7. The training error will increase with λ, the cross validation error will first decrease and then increase. Overfitting occurs at very small λ, underfitting occurs at very large *lambda*.

5.4 Learning curves

A learning curve is a plot of the training and cross validation errors as a function of the size of the training set (i.e. the number of training examples).

When you have high bias (underfitting) more training data does not improve your errors, whereas when you're overfitting more data does decrease your validation error (Figure 8).

Learning curves are useful to again get a better idea of whether we are over- or underfitting and whether collecting more data would help.

5.5 So, what do we do?

Option	When useful
Getting more training examples	This can help fix overfitting (high variance)
Try smaller sets of features	This can help fixing overfitting (high variance). There is no point in carefully selecting a smaller set of features when you are underfitting.
Try getting additional features	Usually this could help fix underfitting (high bias)
Try adding polynomial features (x_1^2, x_2^2, etc)	Can help fixing underfitting (high bias)
Try decreasing λ	Can help fixing underfitting (high bias)
Try increasing λ	Can help fixing overfitting (high variance)

Table 1: **Overview of when to use certain methods to improve performance** There are several ways in which the performance of our model can be improved, some are listed here. We also describe the situations in which these 'improvements' would or wouldn't work.

When using neural networks, a small network is nice because it is computationally cheaper but it is prone to underfitting. Larger networks are more computationally expensive and can be prone to overfitting. Regularization can be used to reduce overfitting. You can try to use several neural networks, each with a different number of hidden layers, then pick the one with the smallest cross-validation error.

5.6 Good way to start

A good way to start a new project may be to

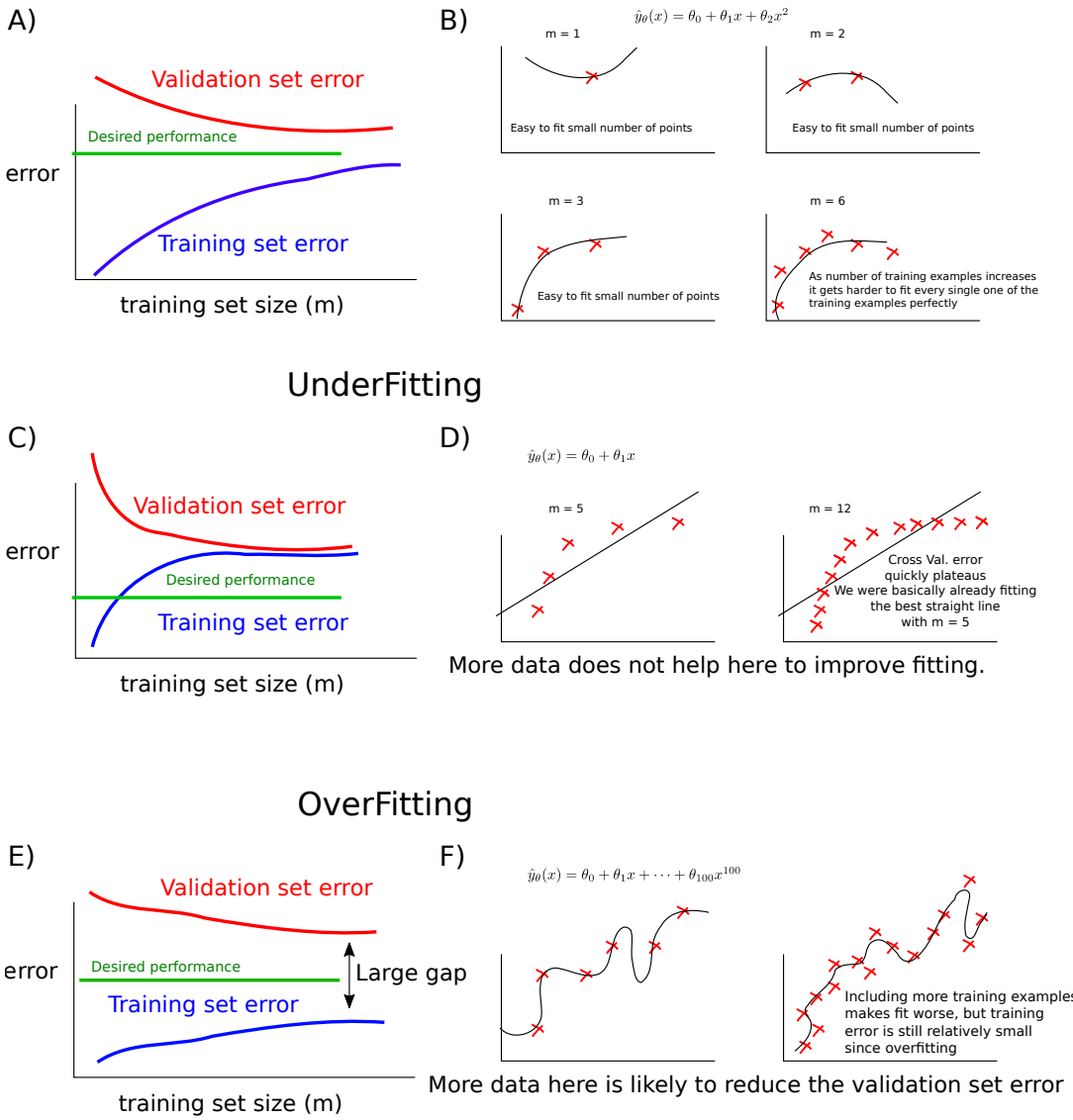

Figure 8: **Learning curve** Here we compare learning curves in 'good' (A,B), underfitting (C,D) and overfitting (E, F) scenarios. The figures on the right are meant to motivate some of the shapes of the learning curves on the left.

- Create a quick and dirty model that runs without errors, check the accuracy on the cross-validation set
- Plot learning curves, etc.. to find out whether the model may be over- or underfitting. Improve if necessary
- Look at some examples it got wrong and see you can come up with better features. Look at some examples that it got correct. You might find that some features didn't really help in getting the images correct, those could e.g. be dropped (if you want to drop features).
- If there's a fast way of checking whether including something new (like 'stemming' which can treat words like 'discount', 'discounts', 'discounted' etc.. as the same word, which might be useful in natural language processing) then you can just compare the cross-validation error with and without this new feature and see whether it's worth using it.

5.7 Skewed classes

Suppose you want to determine whether a change you just made to your model is actually an improvement. You look at the cross-validation error and the error has reduced from 0.8% to 0.5%. Great! Or

not? If one class is much more represented than another, then it can be hard to determine whether the change has actually improved the model by merely looking at a single number.

For example, suppose we are creating a model for breast cancer classification in which we want to predict whether or not someone has breast cancer, but only 0.5% of the samples in our validation set correspond to someone having breast cancer. Then a classifier that always predicts no cancer only has a 0.5% error. This means that if we changed our model to only predict 'no cancer', we also would have reduced our error from 0.8% to 0.5% but it would obviously be a bad change. Basically, when the classes are skewed it could be that your lower error is 'accidental' and does not really reflect whether an actual improvement was made or not. So what do we do in this case?

One good option is to look at the *precision* and *recall*. The precision is the number of true positives divided by the number of predicted positives, which is also just $true\,pos/(true\,pos + false\,pos)$. Suppose we classify whether a person has breast cancer or not, false positives would be when we tell someone he/she has cancer while they actually don't.

The recall is the fraction of all the actual positives that we actually classified as positives (which is $truepos/(true\,pos + false\,neg)$. A high recall is good. In the example sketched above, our recall would be zero which immediately tells us that the classifier is bad. Note that our recall is only zero if we represent the underrepresented class as '1' (positive).

Supose we only want to tell people they have cancer when we are 70% sure they actually have cancer. In a logistic regression model, this would mean predicting '1' if $\hat{y}_\theta(x) \geq 0.7$ and '0' otherwise (instead of setting the threshold at 0.5. This will increase the precision but lower the recall. Similarly, setting the threshold lower than 0.5 leads to higher recall and lower precision. Which one you want to be higher depends on the individual problem. When the threshold is at 0.5 our recall and precision have a more intermediate values.

The disadvantage of only looking at the precision (P) and recall (R) is that now we have two numbers. Is it better to have a P and R of 0.5 and 0.4, to of 0.7 and 0.1? One way to combine precision and recall is called the F_1*-score* (also known as the F-score):

$$F_1 = 2\frac{PR}{P+R} \tag{25}$$

Then pick the choice with the highest F_1-score. So one could pick a range of threshold, evaluate the cross-validation F_1-score and automatically pick the threshold with the highest score. In this way you don't have to manually tweak it or think about it too much. The F-score is just one particular example of choosing combining the precision and recall, there are many others. If a human can predict the answer, a machine could as well if it has access to enough training data (as humans are of course just trained 'computers' themselves).

5.8 Does more data help?

A study in 2001 showed that a bunch of different algorithms performed very similarly and their accuracy increased monotonically with the amount of training data. This suggests that even a 'bad' model with a lot of data can beat a much better model with less data. This led to the saying

"*'It's not who has the best algorithm that wins. It's who has the most data.'*"

How do we know whether a problem can benefit from more data? If this is the case then we might want to spend more time collecting more data instead of perfecting the model.

On good indication is whether a human expert can confidently predict the correct output given the input. For example, suppose you want to fill in one of the words 'too, two, to' in the following sentence: For breakfast, I ate $\cdots$ eggs. The correct answer is clearly 'two'. However, if the problem is to predict the house price based on only its size, then even humans may not do this very well. There are many other factors (e.g. location, number of bedrooms, how modern it is, whether there is a garden) that determine house price. So in this case the features we have are actually not sufficient themselves to get the correct answer.

If the features *can* accurately predict the correct outcome, then when we have a model with many parameters (which will have low bias) and a really large training set (which gives us low variance), we are in a good place.

6 Support Vector Machines

- Adjusted (simplified) logistic regression

- Slightly different formulation, use parameter C instead of λ, is roughly corresponds to $1/\lambda$
- SVMs do not give probabilities as outcomes (as linear regression models do do) but they just output the predicted class
- Is also called a large margin classifier (which is technically only true if C is very large)
- If C is very large, the decision boundary generated is very sensitive to outliers. Also if the data is not linearly seperable, a large value of C doesn't work well (it really tries to fit the exact data and is not good at dealing with noise). A smaller C is able to ignore outliers

6.1 Cost function for SVM

6.2 Kernels

One way we dealt with nonlinear decision boundaries is to create new features by creating a polynomial of the components of the input vectors. For example, if the original ith training example was a vector $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, x_3^{(i)})$, we create a new set of features, which we will call $f^{(i)}$, given by e.g.

$$\begin{aligned} f_1^{(i)} &= x_1^{(i)} & f_2^{(i)} &= x_2^{(i)} \\ f_3^{(i)} &= x_3^{(i)} & f_4^{(i)} &= x_4^{(i)} \\ f_5^{(i)} &= (x_1^{(i)})^2 & f_6^{(i)} &= (x_2^{(i)})^2 \\ f_7^{(i)} &= (x_3^{(i)})^2 & f_8^{(i)} &= x_1^{(i)} x_2^{(i)} \\ etc... & & & \end{aligned} \tag{26}$$

For each new training example x, we have a vector with all these feature values, then calculate the cost of that training example given our weights, do that for each training example to get the overall cost, adjust weights, etc..

Though we can get some complex looking decision boundary, polynomials are of course not the only (and it turns out not the best) way in which we might create a nonlinear function. One common way of creating new features is to use *gaussian kernels*. The features now do not include the components of the training examples themselves anymore, but are formed by creating guassians around each of the sample vectors $x^{(i)}$ and evaluating the guassian function at each of the training examples. To make this more concrete, we obtain the following features:

$$\begin{aligned} f_1^{(i)} &= e^{(x^{(i)} - x^{(1)})^2/(2\sigma^2)} \\ f_2^{(i)} &= e^{(x^{(i)} - x^{(2)})^2/(2\sigma^2)} \\ &\vdots \\ f_m^{(i)} &= e^{(x^{(i)} - x^{(m)})^2/(2\sigma^2)} \end{aligned} \tag{27}$$

Now the number of features we have is equal to the number of training examples. Each feature is a measure of the distance between the current training example and all the other training examples. Note that, for each training example i, one of the features will be equal to 1 ($f_i^{(i)} = 1$). Also, a feature $f_0^{(i)} = 1$ is added to represent the intercept (like θ_0 previously), meaning that the feature vector is $(m+1)$-dimensional: $f^{(i)} \in \mathbb{R}^{m+1}$. This is illustrated in Figure 9.

Similar as before, we will calculate the quantity $\theta^T f^{(i)}$ to determine which class to predict for intput i. If there are only two classes, we predict '1' if $\theta^T f^{(i)} \geq 0$. If there are more than two classes we can apply the 1-vs-all method in a similar manner as before (i.e. train K SVMs where K is the number of classes such that we get K parameters vectors $\theta^{(1)} \cdots \theta^{(K)}$ and choose the class j with the largest $(\theta^{(j)})^T x$ (for some training example x)).

We can also use kernels for logistic regression, but for logistic regression the algorithm then becomes very slow. SVMs are implemented in certain ways which makes them good for using kernels (I know this sounds a bit vague but it's just the way it is).

If the parameter σ^2 is large, the features vary more smoothly as we update them because the slope of the gaussian at a certain distance from the centre is smaller compared to having a small σ^2. The model we fit with a large σ^2 tends to have a relatively high bias and low variance, and vice versa for a model with small σ^2.

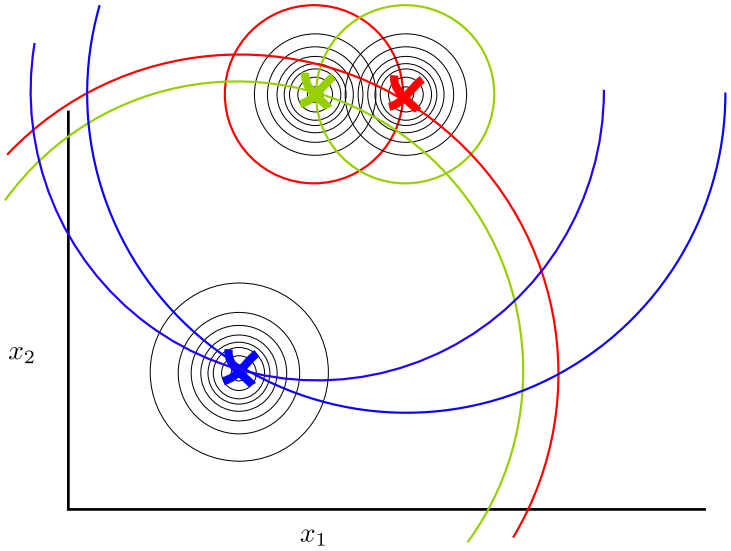

Figure 9: **Gaussian kernel features** This cartoon illustrates how the features of a SVM with gaussian kernels are constructed. Consider the red training example. The features corresponding to this red example are the values of the gaussians centered at each of the training examples (including itself) at the position of the red example itself. These are indicated by the red circles (and the gaussian centered at the red example itself represents a feature value of 1). The same is true for the other training examples. For example, if we denote the red, green and blue training examples by $x^{(1)}, x^{(2)}$ and $x^{(3)}$, then $f_3^{(1)}$ has a small value whereas $f_2^{(1)}$ is larger. Each training example here is represented by a 4-dimensional feature vector.

To actually find the optimal parameters θ, nobody really writes their own code to do this, there are researches who wrote libraries that numerically optimise this optimisation, so just use a library! Most packes have built-in multi-class classification so we can just provide it with the number of classes we have. So do we even need to do something ourselves? Well, we do need to make some educated guess for C and the kernel (as well as the parameters that come with it). One can use the gaussian kernel described above, but also a linear kernel (which is basically 'no kernel' and just e.g. calculates whether $\theta^T x \geq 0$) or some other kernel such as a chi-square kernel, string kernel, histogram intersectio kernel, ... If the dimensionality of the inputs is large, a linear model might be better otherwise we could be overfitting. When using a gaussian kernel, e.g. if we have many training examples and the dimensionality of the inputs is relatively small, we need to choose a σ^2.

6.3 Note on feature confusion

The term 'feature' is often use to denote an independent component of the training examples. Suppose we have a vector $x^{(i)} \in \mathbb{R}^6$ that is given to us, then we would say that there are 6 features. But then for the actual model training we can create additional features (or remove features). The eventual features we use for the training could be called 'training features' which can include the 'features' (as with logistic and linear regression) or not (with SVMs with a gaussian kernel). So don't be too confused whenever the word 'feature' is used in slightly different settings.

6.4 Model selection tips

Let n denote the dimensionality of the input vectors that are given in some dataset, and m the number of such vectors in the dataset. If n is small and m is intermediate, then it is good to use a SVM with Gaussian kernel. This could be e.g. when n is roughly $1-1000$) and m somewhere between $(10-10000)$. If n is small and m is large (e.g. $n = 1-1000$ and $m = 50000+$ then we might want to use logistic regression or a SVM with linear kernel and also increase the number of features (so let the 'training features' be larger than n by e.g. taking a high-degree polynomial with logistic regression). A neural network usually works well but may be slower to train.

7 K-means clustering

7.1 The algorithm

7.1.1 Cost function

The K-means clustering algorithm, like all the other algorithms discussed above, is minimising a cost function. What is this cost function? It is given by:

$$J(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K) = \frac{1}{m} \sum_{i=1}^{m} \left\| x^{(i)} - \mu_{c^{(i)}} \right\|^2 \tag{28}$$

where $c^{(i)}$ denotes the cluster index (which is in $(1, 2, \ldots K)$) to which training example $x^{(i)}$ is currently assigned, $\mu_k \in \mathbb{R}^n$ denotes the cluster centroid (i.e. the coordinates) of cluster k, and $\mu_{c^{(i)}}$ denotes the centroid of the cluster to which example $x^{(i)}$ has been assigned. For example, if example $x^{(3)}$ is assigned to cluster 9, we have $c^{(3)} = 9$ and $\mu_{c^{(3)}} = \mu_9$. This cost function is called the *distortion cost function*. All the cost function is doing is adding the squared distances between each of the training examples and their corresponding cluster centroid (Figure 10).

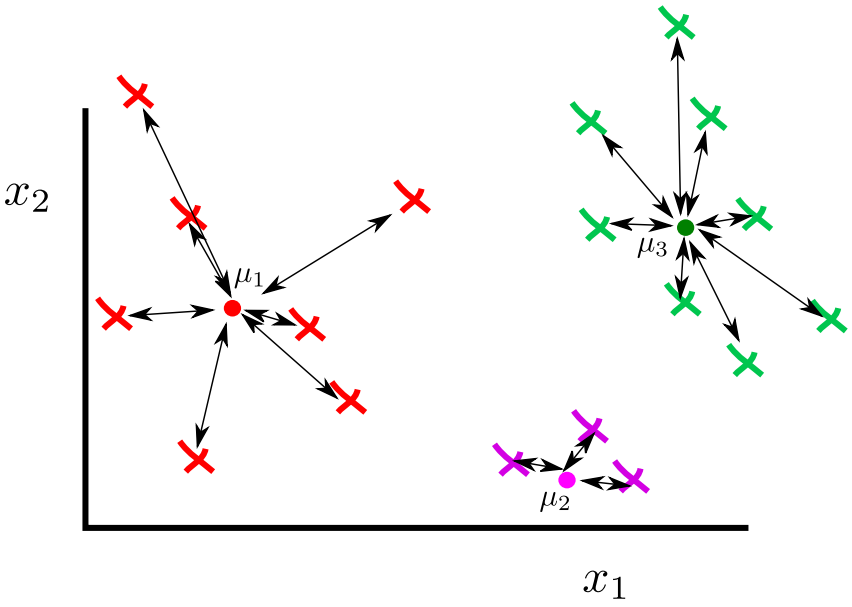

Figure 10: **K-means clustering cost function** Here we illustrate the terms in the cost function of the K-means clustering algorithm. For each data point, the squared distance between this point and its cluster centroid (i.e. the centroid of the cluster to which it is currently assigned) is calculated. All these distances are added to give the cost of the current configuration.

We can now write the K-means clustering algorithm as:

```
Repeat {
   for i = 1 to m
        Set c^(i) to be the index of the cluster centroid
        closest to x^(i) (c^(i) ∈ [1, K])
   for k = 1 to K
        Set μ_k to be the mean of the points
        assigned to cluster k
}
```
(29)

The first loop over i minimises $J()$ while keeping $\mu_1, \ldots, \mu_k$ fixed. So it finds the values $c \in 1, \ldots c \in m$ such that $J()$ is minimised. In the second loop, the values $\mu_1, \ldots, \mu_k$ are then chosen to minimise $J()$. So first we minimise J with respect to c, then with respect to μ. This can be proven :p.

7.1.2 Randomly initialising cluster centroids

How do we initialise our cluster centroids? Suppose we have picked a valuve for K, we can then just randomly pick K training examples and use those coordinates to initialise the cluster centroids. This is a common method. The final solution can depend on the initial random cluster assignment and it's possible to get stuck in a local minimum of the distortion cost function. We can simply perform our entire K-means algorithm many times (e.g. a hundred times) and calculate the cost at the end of each of the repeats. Then pick the clustering that gave the lowest cost value. The higher the number of clusters, the smaller the probability of being stuck in a local minimum.

7.2 How do we choose K?

Unfortunately, there is no general method to determine the 'best' value for K. Here we briefly mention three possible methods to choose K, though there are others of course. Firstly, we can visualise the data in some way and pick a value of K by hand.

Another approach one can take is to maximise the Bayesian Information Criterion (BIC). The BIC is a criterion for model selection among a finite set of models. BIC penalises the use of more parameters as a model with many parameters can overfit the data and will, if its extra parameters are not penalised, always be the better model. The Akaike information criterion (AIC) can also be used. BIC penalises the number of parameters in the model more than AIC does.

$$BIC(M|X) = k\log(m) - 2\mathcal{L}(X|M,\theta) \tag{30}$$

where $\mathcal{L}(X|M,\theta)$ is the likelihood of the dataset X given the model M and parameters θ, k is the number of parameters in the model M, and m is the number of datapoints we have.

Finally, we may apply the *elbow method*. Like in the method above, we try out various values of K and plot the cost function as a function of K. Sometimes the plot looks like an elbow and the number of clusters that should be picked is the location of the elbow. However, this method is not used a lot because often there's not a clear elbow point.

Basically, some common sense and some trial-and-error should be used to choose the number of clusters that is preferred.

8 Large data sets

For very large data sets (e.g. hundreds of million or more training examples), using logistic or linear regression takes a long time. By plotting the learning curve we might find out whether using less training examples doesn't make much of a difference and we might just only use a subset of them (although I still have some feeling that throwing away data is throwing away information, which is always bad!). We here discuss some ways of dealing with big datasets.

8.1 Stochastic gradient descent

For deterministic gradient descent, we compute the derivative of the cost function with respect to θ at each iteration. This means we need to sum over all training examples (see Eq. (17)) which will be very inefficient. This particular form of gradient descent is called *batch gradient descent* because we use the whole batch (the set of training examples) to compute the derivatives. After all that work of calculating all these gradients, we have taken 1 step. We need to do that again and again and again.

In stochastic gradient descent, we ..... Instead of first summing over all training examples and then taking 1 step, we just use a single training example and make a step (based on evaluating the derivative of the cost function but without looping over that i), then look at another training example and take a step, etc... Does the 'stochastic' bit mean that we randomly pick the training example we want to use to evaluate the cost function, meaning that we might pick the same one multiple times in a row (theoretically)? ah no I think it's because we don't necessarily always move downhill so the trajectory towards the global minimum looks a bit random. It wanders around in the region close to the global minimum and doesn't necessarily stay there (since we're not optimising over all parameters simultaneously, we're doing it 1-by-1).

8.2 Mini-batch gradient descent

Stochastic gradient descent is in fact a form of mini-batch gradient descent with a batch size of 1, whereas batch gradient descent itself has a batch size of m (all examples). In mini-batch gradient descent, as we might have expected, we use some batch size $1 < b < m$. Mini-batch gradient descent can be faster than stochastic gradient descent because we walk on the cost function space less randomly. Every step takes more time compared to stochastic gradient descent, but we walk towards the minimum in a more efficient way.

Mini-batch gradient descent generally only outperforms stochastic gradient descent if we have a nice vectorised implementation of our gradient descent which will allow you to partially parallelise the gradient computations over the b examples.

9 Principal Component Analysis

Sometimes we might want to reduce the number of features in our dataset. We'll discuss some examples for 2D and 3D data because it's easier to interpret, but of course these ideas will then be generalised to much higher dimensional data.

Suppose we have 2D data and want to convert this into 1D data. For example, we have two features which are highly correlated with each other. Though our data is 2D, it effectively lies in a 1D space, i.e. a line. We can represent each of the datapoints as just a single cooredinate, namely its position on the line. How do we choose this line? There are various ways of doing this, one of them is Principal Component Analysis, discussed below.

A similar story holds for reducing 3D data to 2D data. If the 3D data actually lies roughly in a plane, we can project the data onto that plane. Then we only need two coordinates to determine the location of each of the datapoints on the plane.

We might also want to use dimensionality reduction to visualise data. If we have 50D data and we want to reduce it to 2D in order to plot it, then some dimensionality reduction method can be used. Now what do the new two components mean? That's not always easy to interpret. By looking at which data point are plotted where in the space we may interpret the meaning of the two features ourselves.

9.1 The algorithm

Principal Component Analysis (PCA) tries to find a lower-dimensional surface on which to project the data, while minimising the projection error. The projection error is often taken to be the sum of squared distances from the original data points to their projected locations. Before we do PCA, features should be normalised.

In general we have n-dimensional data and we want to reduce it to k dimensions. This means we have to find k orthogonal vectors, $u^{(1)}, u^{(2)}, \dots u^{(k)}$ which span the k-dimensional surface onto which the data is projected, while minimising the projection error.

We want to go from n to k dimensions. First, we normalise the data for each feature, by subtracting the mean and dividing by the standard deviation (Eq. (9)). Then, we compute the covariance matrix:

$$\Sigma = \frac{1}{m} \sum_{i=1}^{n} \left(x^{(i)}\right) \left(x^{(i)}\right)^T \tag{31}$$

and find the eigenvectors of this matrix. The covariance matrix is always symmetric positive semidefinite. Singular value decomposition (svd) can be used to get the eigenvalues (just use some library in whatever language we're using). Now we suppose that we have a matrix $U \in \mathbb{R}^{n \times n}$ whose columns represent the eigenvalues of Σ. Equation (31) can be written in vectorised form as $\Sigma = 1/m X^T X$ with X a matrix with each row representing a particular training example (i.e. $X \in \mathbb{R}^{m \times n}$).

We can now pick the first k columns of the matrix U, which we call $U_{reduce} \in \mathbb{R}^{n \times k}$. Then we calculate

$$Z = X U_{reduce} \tag{32}$$

which itself has dimension $(m \times n) \times (n \times k) = m \times k$. Each row of the matrix Z represents a single training example, which now itself is a vector of length k.

9.2 How to go back to our original data?

When performing PCA, we have reduced the number of dimensions and now represent our training examples by $z^{(i)} \in \mathbb{R}^k$. Given these vectors z, how do we go back to our original data vectors x? We can of course just do $\tilde{X} \equiv ZU_{reduce}^{-1} = ZU_{reduce}^T \approx X$, where we used the fact that U is an orthogonal matrix. The matrix $\tilde{X}$ will not exactly be X, but if the projection error was small then it's a reasonable approximation.

The projection error which PCA miminises is given by

$$\text{Proj. error} = \frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - \tilde{x}^{(i)}\right\|^2 \tag{33}$$

9.3 How do we choose the number of principal components?

How do we choose k, the number of dimensions we want to have? Typically, k is chosen to be the smallest value which satisfies the following condition:

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - \tilde{x}^{(i)}\right\|^2}{\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)}\right\|^2} \leq 0.01 \text{ (or 0.05, or 0.1, etc..)} \tag{34}$$

Here the denominator represents the total variation in the data, i.e. how spread out are the original training examples we had. Note that we had normalised the data so the training examples should lie around the origin, the denominator then just calculates how far away from the origin our datapoints lie. The condition above means that we retain 99% of the variance that was present in our original dataset. The value 0.01 can be chosen differently, typically one would say 'I want to have the k that retains y% of the variance' for some y. Sometimes quite a lot of dimensions can be lost while still retaining 90% of the variance.

One way of finding k is of course to just start with some low value and increase it until for the first time we satisfy the condition above. However, one can imagine this is quite inefficient. When using singular value decomposition, a diagonal matrix S is often returned. This matrix S can be used to calculate the fraction in Eq. (34) for every value of $k \in [1, n]$. To be precise, the fraction in Eq. (34) for a particular value of k is given by

$$1 - \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \tag{35}$$

We can therefore just calculate this value for different values of k and find the lowest k for which the above equation is less than whatever value we want to be less than to. In summary, we calculate Σ, use singular-value decomposition, use S to find our preferred value of k, then use U to construct the new data vectors z.

9.4 Example of when we could use PCA

Suppose we want to recognise images which are e.g. 200×200 pixels, which means we have $200 \times 200 \times 3 = 1.2e5$ parameters (if they are colour images). We want to reduce this number. If original labelled trainingset is given by $\left\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\right\}$, then we use PCA to map our inputs to a lower-dimensional space:

$$\begin{gathered} x^{(1)}, x^{(2)}, \ldots, x^{(m)} \in \mathbb{R}^{1.2e5} \\ \downarrow PCA \\ z^{(1)}, z^{(2)}, \ldots, z^{(m)} \in \mathbb{R}^{1000} \end{gathered} \tag{36}$$

meaning our new training set is given by $\left\{(z^{(1)}, y^{(1)}), \ldots, (z^{(m)}, y^{(m)})\right\}$. Of course when predict outputs in a test-set, the original test-set datapoint is also first mapped to a lower-dimensional version. The PCA algorithm which finds the matrix U_{reduce} which minimises the projection error should be used on the training set only, then the cross-validation and test set data can be transformed using that U_{reduce} found. Using a powerful GPU to speed up our calculations is of course preferable than throwing away some of our input information. Only use PCA when your learning is really not fast enough for your purposes or if there isn't enough memory to use the original data.

10 Anomaly detection

We are given some dataset $\{x^{(1)}, \dots x^{(m)}\}$ which are assumed to be 'normal' data points (i.e. not anomalous), and we want to know whether a new point x_{test} is anomalous. To do this, we come up with some model $P(x)$ denoting the probability that x occurs. Now we define x_{test} to be anomalous if $P(x_{test}) < \epsilon$ for some small value ϵ.

On common application of anomaly detection is fraud detection. Features of user activities can be obtained, such as: how often does this user log in? How many times per minute does the user click on links? How long does the user stay logged in for? What is the user typing speed? How many blog posts has the user made? etc.. Based on values for these features from many users, one can construct a model of 'normal' behaviour. Unusual users can then be identified.

Other examples of applications are manufacturing (is a car or airplane working properly?) and monitoring computeres in data centres (finding machines that may be about to go down).

10.1 The algorithm

Suppose we have a training set $\{x^{(1)}, \dots x^{(m)}\}$ where each example $x \in \mathbb{R}^n$. The probability of observing example $x^{(i)}$ is the product of the probabilities of observing each of the feature values of $x^{(i)}$ (if these features are independent). If we assume each of the features is normally distributed, we obtain

$$P(x^{(i)}) = \prod_{j=1}^{n} P(x_j^{(i)} | \mu_j, \sigma_j^2) \tag{37}$$

where $\mu_j = \sum_{i=1}^{m} x_j^{(i)}$ and $\sigma_j^2 = \sum_{i=1}^{m} (x_j^{(i)} - \mu_j)^2$. Note that we assumed that the features are independent of each other, which may not be true in general. However, even in cases where this is not true it is often assumed it is and the results can still be good. It seems though that in that case we can calculate covariances between features and use those to calculate the joint density.

We use the following steps to create our anomaly detection algorithm:

1. Choose features x_i for $i \in [1, n]$ that may be indicative of anomalous examples.
2. Get yourself a dataset of m training examples, each having the n features chosen above
3. Find the parameters $\mu_1, \dots \mu_n, \sigma_1^2, \dots \sigma_n^2$, i.e. the means and variances of each of the feature values.
4. Given a new example x, we now compute $P(x) = \left(\frac{1}{\sqrt{2\pi\sigma_j^2}}\right)^n \prod_{j=1}^{n} \exp(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2})$. The new example is an anomaly if $P(x) \leq \epsilon$