Zusammenfassung Machine Learning HS 17

January 4, 2018

1 Supervised Learning

Given some data, we want to be able to predict the outcome. E.g. when we have the Living area, we may want to predict the price of a house as a function of the size of the living area.

To establish a notation, we'll use $x^{(i)}$ to denote the input variables (the living area in our example). These are also called **input features**. We'll use $y^{(i)}$ to denote the output, which is also called **target variable**. A pair $(x^{(i)}, y^{(i)})$ is then called a **training example** and the whole list of training examples we're using is called the **training set**. The i says that this is the i-th training example.

Our goal is, given a training set, to find a function h so that $h(x)$ predicts y "accuratly". This function $h(x)$ is called a **hypothesis**.

`../../Machine Learning/hypothesis.png`

When the target variable that we're trying to predict is continuous (can take on infinitely many, uncountable values), we call the learning problem a **regression problem**. When y can take on only a small number of discrete values, we call it a **classification problem.**

1.1 Linear Regression

To perform supervised learning, we must decide how to represent our hypothesis h. As an inital choice, we're going to approximate y as a linear function of x:

$$h_\Theta(x) = \sum_{i=0}^{n} \Theta_i x_i = \Theta^T x$$

The Θ_i's are the **parameters** or the **weights** which parameterize our functions mapping from the **feature space** X to the **target space** Y. Note, that x_0(also called the **intercept term**) is always equal to 1 to simplify our notation. On the right-hand side of the equation above, we use both $\Theta$and x as vectors to simplify the math.

Now how are we going to learn our parameters Θ? We will try to make $h(x)$ to be as close to y as possible for the training examples we have. To do this, we define the **cost function**, which measures how close the $h(x^{(i)})$'s are to the corresponding $y^{(i)}$:

$$J(\Theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2$$

1.1.1 LMS algorithm

We want to choose Θso as to make $J(\Theta)$ as small as possible. We start by choosing some initial guess for $\Theta$and then we repeatedly change $\Theta$to minimize $J(\Theta)$ step by step. We now look at the **gradient descent** algorithm that does exactly that:

$$\Theta_j := \Theta_j - \alpha\frac{\partial}{\partial}J(\Theta)$$

where $\alpha$is the **learning rate**. This way w're going stepwise in the direction of steepest descrease of J. For a single training example, we get the update rule:

$$\Theta_j := \Theta_j + \alpha(y^{(i)} - h_\Theta(x^{(i)}))x_j^{(i)}$$

This rule is called the **LMS (least mean square)** update rule. It is proportional to the error term $(y^{(i)} - h_\Theta(x^{(i)}))$ so for small errors w're only going to make small adjustments whereas we're going to make bigger adjustments for bigger errors.

The adjustment to the whole training set is quite simple:

```
Loop {
    for i=1 to m, {
        $\Theta_j := \Theta_j + \alpha(y^{(i)} - h_\Theta(x^{(i)}))x_j^{(i)}$ (for every j)
    }
}
```

We repeatedly run through the training set and for each training example we update all he parameters. This algorithm is called **stochastic gradient descent** or **incremental gradient descent**.

Another option is the **batch gradient descent**:

```
Repeat until convergence{
    $\Theta_j := \Theta_j + \alpha\sum_{i=1}^{m}(y^{(i)} - h_\Theta(x^{(i)}))x_j^{(i)}$
}
```

Stochastic gradient is often preferred to batch gradient because whereas batch gradient has to scan over the whole training set (which can get costly for large m), stochastic gradient can start right away with the first training example. Note, that stochastic gradient doesn't reach a global minimum (as batch gradient) but oscillates around an optimal solution.

1.2 The normal equations

We will now see another way to minimize J. In this method we' re going to take the derivatives of J with respect to the Θ_j's and set them to zero. First we're repeating some rules to simplify our math.

1.2.1 Matrix derivatives

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{m1}} & \cdots & \frac{\partial f}{\partial A_{mn}} \end{bmatrix}$$

$$tr(A) = tr(A^T)$$

$$tr(ABCD) = tr(DABC) = tr(CDAB) = tr(BCDA)$$

1.2.2 Least squares revisited

Given a training set, we define the **design matrix** X to be the matrix containing all training examples, where the i-th training example is the i-th row.

$$X = \begin{bmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix}$$

Let $\overrightarrow{y}$ be the m-dimensional vector containing all target values from the training set. Using math rules and the formula we got in the last section we get

$$\nabla_\Theta J(\Theta) = X^T X \Theta - X^T y$$

Therefore the value of Θthat minimizes $J(\Theta)$ is given by the equation

$$\Theta = (X^T X)^{-1} X^T y$$

1.3 Probabilistic interpretation

We will now look at linear regression under terms of probabilistic assumptions. To start, we assume that the target variable and the inputs are related via

$$y^{(i)} = \Theta^T x^{(i)} + \epsilon^{(i)}$$

where $\epsilon^{(i)}$ is an error term that is distributed IID (independently and identically distributed) according to a Gaussian distribution with mean 0 and variance σ^2. We write this as "$\epsilon^{(i)} \sim N(0, \sigma^2)$." This implies that

$$p(y^{(i)}|\, x^{(i)}; \Theta) = \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{(y^{(i)} - \Theta^T x^{(i)})^2}{2\sigma^2}\right)$$

This follows from the densitiy of $\epsilon^{(i)}$. The notation $p(y^{(i)}|\, x^{(i)}; \Theta)$ indicates the distribution of $y^{(i)}$given $x^{(i)}$, parameterized by Θ. Getting back to our whole training set we can also write the probability of the whole data as $p(y|\, X; \Theta)$. When we wish to see this as a function of Θ, we will call it the **likelihood** function:

$$L(\Theta) = L(\Theta; X, y) = p(y|\, X; \Theta)$$

Because of our assumption of independence for the $\epsilon^{(i)}$, this can also be written as

$$L(\Theta) = \prod_{i=1}^{m} p(y^{(i)}|\, x^{(i)}; \Theta) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{(y^{(i)} - \Theta^T x^{(i)})^2}{2\sigma^2}\right)$$

The principal of **maximum likelihood** says that we should choose Θso as to make the data as high probability as possible. Therefore we should choose $\Theta$to maximize $L(\Theta)$. To make our calculations simpler we will instead of maximizing $L(\Theta)$maximize a strictly increasing function of $L(\Theta)$, in our case the **log likelihood** $l(\Theta)$:

$$l(\Theta) = log\, L(\Theta) = m\, log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^{m} (y^{(i)} - \Theta^T x^{(i)})^2$$

Hence, maximizing $l(\Theta)$gives the same answer as maximizing

$$\frac{1}{2} \sum_{i=1}^{m} (y^{(i)} - \Theta^T x^{(i)})^2,$$

which is our original $J(\Theta)$, the cost function from least-square.

2 Classification and logistic regression

Classification is quite similar to the regression problem, but now our values y take on only a small number of discrete values. To start, we will focus on the **inary classification** problem, where y can only take on 1 and 0. Given $x^{(i)}$, the corresponding $y^{(i)}$is also called the **label** for the training example.

2.1 Logistic regression

If we would use our already known linear regression algorithm to predict y given x, we would see that this method performs rather poorly. It also makes no sense that $h_\Theta(x)$ can take on values outside of $[0,1]$. To fix this we will change our hypothesis:

$$h_\Theta(x) = g(\Theta^T x) = \frac{1}{1+e^{-\Theta^T x}}$$

where

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

is called the **logistic function** or the **sigmoid function**. As you can see, $g(z)$ tends to 1 for large values of z and to 0 for small values of z. The derivative of the sigmoid function is

$$g'(z) = g(z)(1-g(z)).$$

Similar as before we will now make a set of probabilistic assumptions and then fit our parameters via maximum lieklihood. Let us assume that

$$P(y=1|\,x;\Theta) = h_\Theta(x)$$

$$P(y=0|\,x;\Theta) = 1 - h_\Theta(x)$$

Therefore

$$p(y|x;\Theta) = (h_\Theta(x))^y(1-h_\Theta(x))^{1-y}$$

Again, assuming that our m training examples ar IID, the likelihood of the parameters is

$$L(\Theta) = \prod_{i=1}^{m}(h_\Theta(x^{(i)}))^{y^{(i)}}(1-h_\Theta(x^{(i)}))^{1-y^{(i)}}$$

As before, we will again maximize the log likelihood

$$l(\Theta) = log\,L(\Theta) = \sum_{i=1}^{m} y^{(i)} log\,h(x^{(i)}) + (1-y^{(i)}) log(1-h(x^{(i)}))$$

How do we maximize the likelihood? Similar as with linear regression we can use gradient ascent. In vectorial notation, our updates will be given by $\Theta := \Theta + \alpha\nabla_\Theta l(\Theta)$ where

$$\frac{\partial}{\partial\Theta_j} l(\Theta) = (y - h_\Theta(x))x_j$$

This gives us therefore the stochastic gradient ascent rule

$$\Theta_j := \Theta_j + \alpha(y^{(i)} - h_\Theta(x^{(i)}))x_j^{(i)}$$

2.2 The perceptron learning algorithm

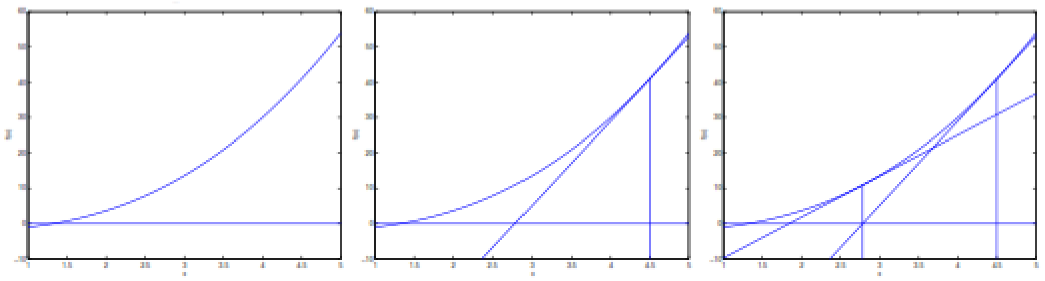
We want to be able to "force" our algorithm to output values that are either exactly 1 or 0. We can do this by modifying $g(z)$so that it's outputing 1 if $z \geq 0$ and 0 if $z < 0$. If we then let $h_\Theta(x) = g(\Theta^T x)$ and use the same update rule as above, we have the **perceptron learning algorithm**.

2.3 Another algorithm for maximizing $l(\Theta)$

First, consider Newton's method for finding a zero of a function. The update rule here is

$$\Theta := \Theta - \frac{f(\Theta)}{f'(\Theta)}$$

Here, we're approximating the minimum by taking the tangent of the function, going through the current guess of Θ. The point where this tangent is zero becomes our new guess.



This method gives us a way to get to $f(\Theta) = 0$. But we want to use it to maximize a function. The maxima of this function l correspond to points where the first derivative is zero. Therefore we obtain the update rule

$$\Theta := \Theta - \frac{l'(\Theta)}{l''(\Theta)}$$

The generalization if this method to the multidimensional setting is given by

$$\Theta := \Theta - H^{-1}\nabla_\Theta l(\Theta)$$

where H is the Hessian which is given by

$$H_{ij} = \frac{\partial^2 l(\Theta)}{\partial\Theta_i \partial\Theta_j}.$$

Usually, Newton's method converges much faster than gradiant descent and needs fewer iterations. But for large numbers of features this method can get very costly because the inverse of a big matrix needs to be calulated. When Newton's method is applied to maximize the logistic regression log likelihood function $l(\Theta)$, the resulating method is also called **Fisher scoring**.

3 Generative Learning algorithms

So far we always modelled $p(y|x)$. Now we want to learn $p(x|y)$. For example, if y indicates whetere an example is a dog (0) or an elephant (1), then $p(x|y=0)$ models the distribution of the dogs' features. After modeling $p(y)$ (called **class priors**) and $p(x|y)$ our algorithm can then use Bayes rule to derive the posterior distribution on y given x:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

3.1 Gaussian discriminant analysis GDA

This model assumes that $p(x|y)$ is distributed according to a multivariate normal distribution.

3.1.1 Multivariate Normal Distribution

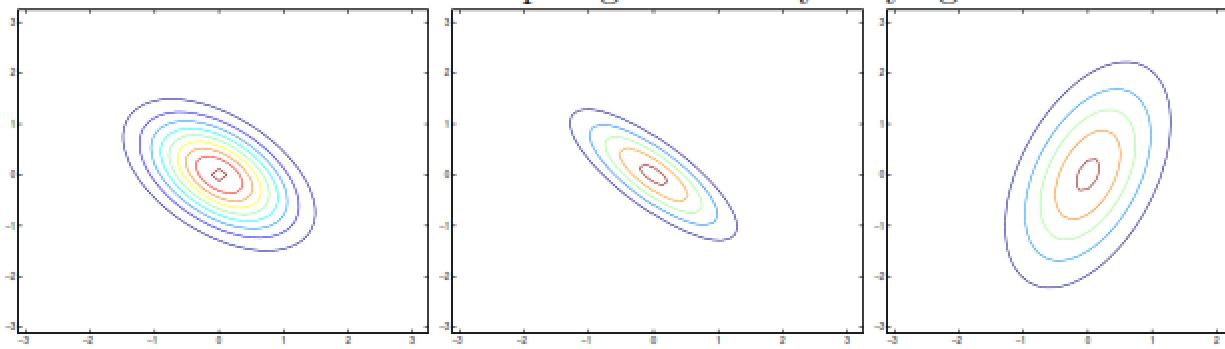
The multivariate normal distribution in n-dimensions is parameterized by a **mean vector** $\mu \in \mathbb{R}^n$and a **covariance matrix** $\Sigma \in \mathbb{R}^{n\,x\,n}$ where $\Sigma$is symmetric and positive definitve. The density (also written as $N(\mu, \Sigma)$) is given by

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

The mean of this density is of course μ. The **covariance** of a vector-valued random variable Z is defined as Cov(Z)=E[(Z-E[Z])(Z-E[Z])^T] If X is distributed according to a multivariate normal distribution, then $Cov(X) = \Sigma$.

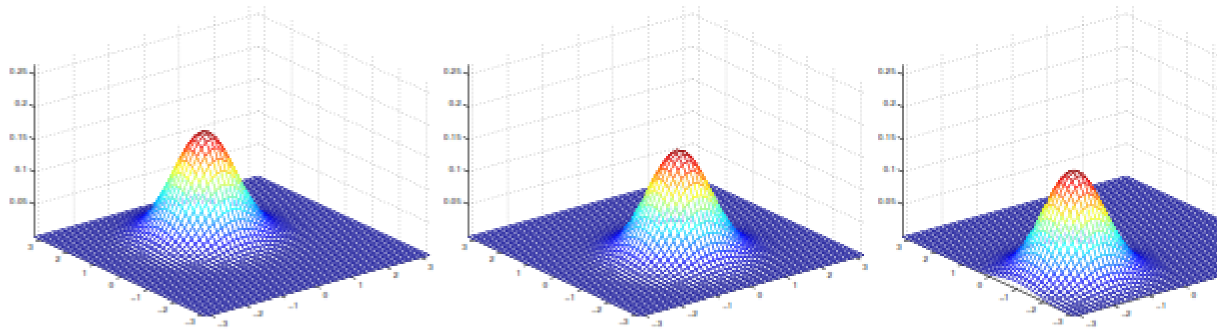
Graphically spoken, the covariance matrix changes the shape of the ellipsoid where mean moves the center:

Here's one last set of examples generated by varying Σ:



The plots above used, respectively,

$$\Sigma = \begin{bmatrix} 1 & \text{-}0.5 \\ \text{-}0.5 & 1 \end{bmatrix}; \;\; \Sigma = \begin{bmatrix} 1 & \text{-}0.8 \\ \text{-}0.8 & 1 \end{bmatrix}; \;\; .\Sigma = \begin{bmatrix} 3 & 0.8 \\ 0.8 & 1 \end{bmatrix}.$$



The figures above were generated using $\Sigma = I$, and respectively

$$\mu = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; \;\; \mu = \begin{bmatrix} \text{-}0.5 \\ 0 \end{bmatrix}; \;\; \mu = \begin{bmatrix} \text{-}1 \\ \text{-}1.5 \end{bmatrix}.$$

3.1.2 The Gaussian Discriminant Analysis model

The model is:

$$y \sim Bernoulli(\phi)$$

$$x|y = 0 \sim N(\mu_0, \Sigma)$$

$$x|y = 1 \sim N(\mu_1, \Sigma)$$

The parameters of our model here are $\phi, \Sigma, \mu_0$and μ_1.The log-likelihood of the data is given by

$$l(\phi, \mu_0,\mu_1, \Sigma) = log \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}; \phi, \mu_0, \mu_1, \Sigma)$$

When we maximize l with respect to the parameters we find he maximum likelihood estimates of the parameters to be:

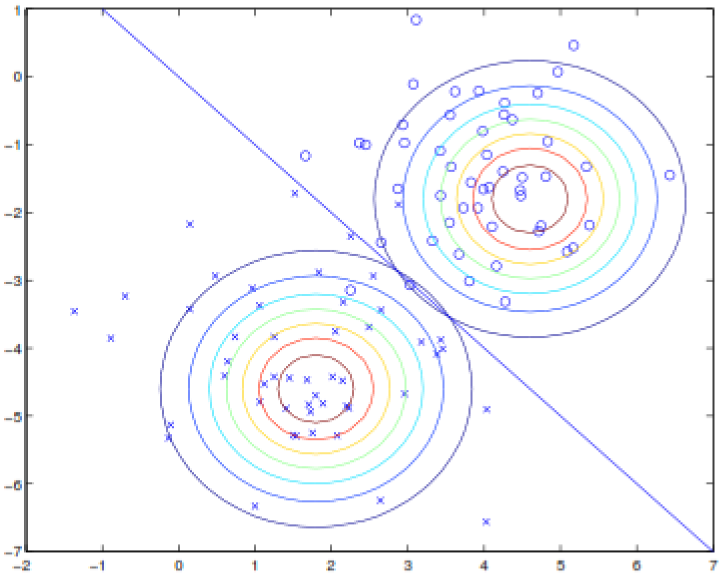
$$\phi = \frac{1}{m} \sum_{i=1}^{m} 1\{y^{(i)} = 1\}$$

$$\mu_0 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}}$$

$$\mu_1 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}$$

$$\Sigma = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - u_{y^{(i)}})^T$$

The following image shows what the algorithm is doing:



The data has been fit to the two gaussians we see here. One is for the data that is an x, the other is for the data that is a circle. Also shown is the decision boundardy where $p(y=1|x)=0.5$. On one side we predict $y=1$, on the other we predict $y=0$. The gaussians have the same shape because they have the same covariance matrix but their mean $\mu_0,\mu_1$is different .

3.1.3 Discussion: GDA and logistic regression

We can view $p(y=1|x;\phi,\mu_0,\mu_1,\Sigma)$ as a function of x and will find that in can be expressed in the form

$$p(y=1|x;\phi,\Sigma,\mu_0,\mu_1)=\frac{1}{1+exp(-\Theta^T x)}$$

where $\Theta$is some appropriate function of the remainin parameters. This is the form that logistic regression used to model $p(y=1|x)$. When do we prefer one model over the other?

GDA makes *stronger* modeling assumptions than logistic regression does. It turns out that when these assumptions are correct, GDA will find better fits to the data. This is especially the case when $p(x|y)$ is gaussian. But by making weaker assumptions, logistic regression is more *robust* and less sensitive to wrong assumptions than GDA. So if we assumed that $p(y|x)$ is logistic and $x|y\sim Poisson$ then GDA may not do well on such non-Gaussian data. GDA makes more assumptions and is more data-efficient when the modeling assumptions are (at least approximately) correct. Logistic regression makes weaker assumptions and is more robust. Especially in non-Gaussian data, logistic regression will do better than GDA.

3.2 Naive Bayes

GDA works with continuous, real-valued vectors. This is a different learning algorithm where the x_i's are discrete-valued. To demonstrate, assume we're gonna build an spam-filter.

We will represent an email via a feature vector whose length is equal to the length of the dictionary. If an email contains word number i in the dictionary, then $x_i=1$, otherwise $x_i=0$. The words in the feature vector are called **vocabulary**. We want to model $p(x|y)$. This means that given spam or not spam, how possible is it that word i is going to show up. To model $p(x|y)$ we're going to make a very strong assumption. We will assume that the x_i's are conditionally independent given y. This assumption is called the **Naive Bayes (NB) assumption** and the resulting algorithm is called the **Naive Bayes classifier.** This means that when we have $y=1$, the knowledge of x_{2987}makes no effect on the knowledge of x_{19}. This can also be written as $p(x_{2987}|y)=p(x_{2987}|y,x_{19})$ (this is different from saying they are independet what would mean $p(x_{2987})=p(x_{2987}|x_{19})$).

We now have $p(x_1,\ldots,x_n|y)=\prod_{i=1}^{n}p(x_i|y)$

Note that even though we made a strong assumptions, NB still works well on many problems.

Our model is parameterized by $\phi_{i|y=1} = p(x_i = 1|y = 1)$, $\phi_{i|y=0} = p(x_i = 1|y = 0)$ and $\phi_y = p(y = 1)$. When we write down the joint likelihood of the data we get

$$L(\phi_y, \phi_{i|y=0}, \phi_{i|y=1}) = \prod_{i=1}^{m} p(x^{(i)}, y^{(i)})$$

Maximizing this results in the maximum likelihood estimates:

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}$$

(How many times does the word show up in spam email in contrast to all spam email?)

$$\phi_{j|y=0} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}}$$

(How many times does the word show up in non-spam email in contrast to all non-spam email?)

$$\phi_y = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}{m}$$

Now that we have fit all our data, to make a prediction we simply have to calculate

$$p(y = 1|x) = \frac{p(x|y = 1)p(y = 1)}{p(x)} = \frac{(\prod_{i=1}^{n} p(x_i|y = 1))p(y = 1)}{(\prod_{i=1}^{n} p(x_i|y = 1))p(y = 1) + (\prod_{i=1}^{n} p(x_i|y = 0))p(y = 0)} = \frac{(\prod}{(\prod_{i=1}^{n} \phi_{i|y=1})}$$

Note that even though this example was with binary values, the generalization to a case where $x_i$can take on values in {1,2,...,k} is straightforward. We could just model $p(x_i|y)$ as multinomial rather than as Bernoulli. It is in practice quite common to **discretize** and then use Naive Bayes. When original, contiuous-valued attributes are not well-modeled by a multivariate normal distribution, discretizing the features and using Naive Bayes will often result in a better classifier.

3.2.1 Laplace smoothing

Assume a word our algorithm has never seen shows up. Because it has not seen it before it will decide the probability to be zero for spam and non-spam mail what will will lead to $\frac{0}{0}$ division: Our algorithm doesn't know how to predict this case. To take account of such new words we can use **Laplace smoothing:**

$$\phi_j = \frac{(\sum_{i=1}^m 1\{z^{(i)} = 1\}) + 1}{m+k}$$

where k is the maximum value z can take on. Returning to our Naive Bayes classifier with Laplace smoothing we get:

$$\phi_{j|y=1} = \frac{\sum_{i=1}^m 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\} + 1}{\sum_{i=1}^m 1\{y^{(i)} = 1\} + 2}$$

$$\phi_{j|y=0} = \frac{\sum_{i=1}^m 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\} + 1}{\sum_{i=1}^m 1\{y^{(i)} = 0\} + 2}$$

3.2.2 Event models for text classification

We'll look at one more model for text classification. In the last chapter we assumed that first it is randomly determined whether the next mail is from a spammer or a non-spammer. Then the person sending the mail runs through the dictionary deciding whether to use or not use the word i according to the probabilities $\phi_{i|y}$. Now we take a look at the **multinomial event model.**

$x_i$will now denote the identitiy of the i-th word in the email. Thus, $x_i$can now take on values from 1 to $|V|$ where V is the dictionary. $(1, 17, ...)$ means the first word in the mail is the first from the dictionary, the second word is the 17-th from the dictionary, etc. Now again, it is first decided whether it is a spam mail or not. Then, the sender of the email writes the email by generating $x_1$from some multinomial distribution over words, then chooses $x_2$the same way but independently, etc. The overall probability of a message is then given by $p(y)\prod_{i=1}^n p(x_i|y)$. This looks similar as before but the terms in the formula mean now different things and in particular, $x_i|y$ is now multinomial, rather than Bernoulli.

The parameters are $\phi_y = p(y)$, $\phi_{i|y=1} = p(x_j = i|y = 1)$,$\phi_{i|y=0} = p(x_j = i|y = 0)$

The likelihood is then given by

$$L(\phi, \phi_{i|y=1}, \phi_{i|y=0}) = \prod_{i=1}^m (\prod_{j=1}^{n_i} p(x_j^{(i)}|y; \phi_{i|y=1}; \phi_{i|y=0}))p(y^{(i)}|\phi).$$

Maximizing this yield the maximum likelihhod estimates for the parameters:

$$\phi_{k|y=1} = \frac{\sum_{i=1}^m \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 1\}}{\sum_{i=1}^m 1\{y^{(i)} = 1\}n_i}$$

$$\phi_{k|y=0} = \frac{\sum_{i=1}^m \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 0\}}{\sum_{i=1}^m 1\{y^{(i)} = 0\}n_i}$$

$$\phi_y = \frac{\sum_{i=1}^m 1\{y^{(i)} = 1\}}{m}$$

We can also apply Laplace smoothing (what is needed in practice for good performance):

$$\phi_{k|y=1} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 1\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} n_i + |V|}$$

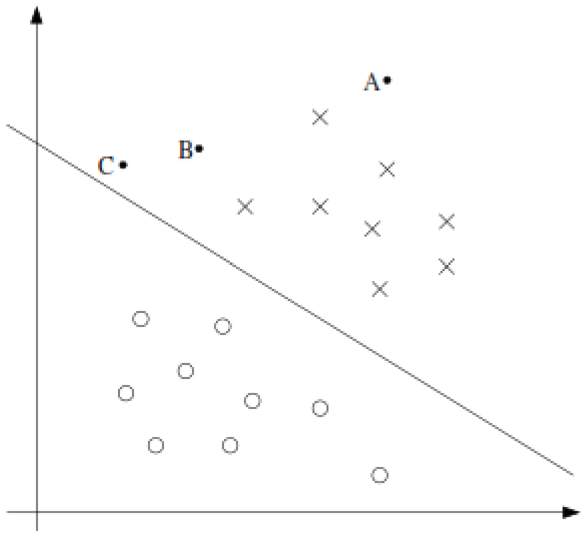
$$\phi_{k|y=0} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 0\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} n_i + |V|}$$

4 Support Vector Machines

4.1 Margins: Intuition

Consider logistic regression. We would predict "1" on an input 1 iff $h_\Theta(x) \geq 0.5$ or equivlently iff $\Theta^T x \geq 0$. The larger $\Theta^T x$ is, the higher our degree of "confidence" that the label is +1+. Informally we could also say, that $y = 1$ if $\Theta^T x \gg 0$. Similarly we could say that $y = 0$ if $\Theta^T x \ll 0$. Therefore we could say informally that we found a good fit if we can find Θso that $\Theta^T x \gg 0$ whenever $y^{(i)} = 1$, and $\Theta^T x \ll 0$ whenever $y^{(i)} = 0$.

For another type of intuition, consider the following figure where x's represent positive training examples and y's negative training examples. The line is also called **separating hyperplane**.



Making a prediction for A should be much easier and more confident than one for C. It would be nice to have a classifier that allows us to make correct and confident predictions on the training examples.

4.2 Notation

To make things easier we'll consider a linear classifier for a binary classification problem with labels $y \in \{-1, 1\}$. We will also use parameters w, b and write our classifier as $h_{w,b}(x) = g(w^T x + b)$ where $g(z) = 1$if $z \geq 0$and 0 otherwise. So, b take the role of $\Theta_0$and w the role of $[\Theta_1, \ldots, \Theta_n]^T$.

4.3 Functional and geometric margins

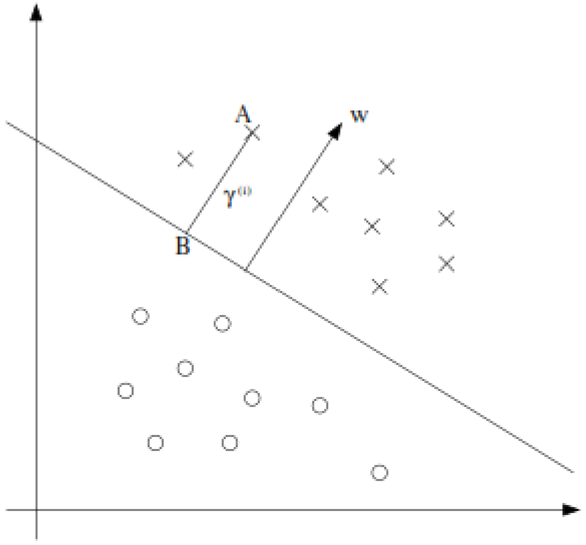
We define the **functional margin** of (w, b) with respect to a training example i

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x + b)$$

Note, that when $y^{(i)}$is 1, then for the functional margin to be large, $(w^T x + b)$ needs to be a large positive number. Similar for $y^{(i)} = -1$it needs to be a large negative number. Moreover, if $y^{(i)}(w^T x + b) > 0$, then our prediction on this example is correct. The problem here is that scaling our w, b does not change our hypothesis $h_{w,b}$since it only depends on the sign. But scaling affects our functional margin. It might make sense to normalize this so we can't exploit this freedom. We'll come back later to this. For a whole training set S we set the overall function margin to be the smallest of the functional margins:

$$\hat{\gamma} = min\hat{\gamma}^{(i)}$$

Let's talk about **geometric margins**. Consider the following figure:



We want to be able to find the value of $\gamma^{(i)}$. We know that B is given by $x^{(i)} - \gamma^{(i)} \cdot w/||w||$. It also has to satisfy the equation of the decision boundary $w^T x + b = 0$. Hence,

$$w^T \left(x^{(i)} - \gamma^{(i)} \frac{w}{||w||} \right) + b = 0$$

Solving for $\gamma^{(i)}$ yields

$$y^{(i)} = \left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||}$$

More generally, to consider negative y

$$y^{(i)} = y^{(i)} \left(\left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||} \right)$$

Now scaling w, b does not change the margin. Again we define the geometric margin as the smallest of the geometric margins of the training set.

4.4 The optimal margin classifier

It seems that a good way is to find a decision boundary that maximizes the margin. For this (and now), we'll assume that our data is linear separable. This poses the following optimization problem:

$$max_{\gamma,w,b} \gamma$$

$$s.t.\ y^{(i)}(w^T x^{(i)} + b) \geq \gamma,\ \ i = 1, \ldots, m$$

$$||w|| = 1$$

We want to maximize γ. The last constraint ensures that the functional margin equals the geometric margin. Solving this problem will result with (w, b) with the largest possible geometric margin. This problem is quite hard to solve, so we're going to transform it a bit:

$$max_{\gamma,w,b} \frac{\hat{\gamma}}{||w||}$$

$$s.t.\ y^{(i)}(w^T x^{(i)} + b) \geq \hat{\gamma},\ \ i = 1, \ldots, m$$

We can do this because of $\gamma = \frac{\hat{\gamma}}{||w||}$. This is still a bit nasty, so we keep going with a bit of scaling. Since we are now allowed to scale we can set $\hat{\gamma} = 1$ and rescale (w, b)what leads to:

$$min_{\gamma,w,b} \frac{1}{2} ||w||^2$$

$$s.t.\ y^{(i)}(w^T x^{(i)} + b) \geq 1,\ \ i = 1, \ldots, m$$

This can be solved efficiently and is called the **optimal margin classifier**.

4.5 Lagrange duality

This part is about solving constrained optimization problems. Cosnider a problem of the following form:

$$min_w \ f(w)$$

$$s.t. \ h_i(w) = 0, \ i = 1, \ldots, l$$

We now define the **Lagrangian** to be

$$L(w, \beta) = f(w) + \sum_{i=1}^{l} \beta_i h_i(w)$$

The β_i are called the **Lagrange multipliers**. We would next find and set the partial derivatives of the Langragian to zero:

$$\frac{\partial L}{\partial w_i} = 0; \ \frac{\partial L}{\partial \beta_i} = 0$$

and solve for w and β.

We will now generalize this to be also able to solve problems with inequaility constraints. The following will be called the **primal** optimization problem:

$$min_w \ f(w)$$

$$s.t. \ g_i(w) \leq 0, \ i = 1, \ldots, k$$

$$h_i(w) = 0 \ i = 1, \ldots, l$$

To solve this we define the **generalized Lagrangian**

$$L(w, \alpha, \beta) = f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w)$$

Consider now the quantity

$$\Theta_P(w) = \underset{\alpha, \beta : \alpha_i \geq 0}{max} L(w, \alpha, \beta)$$

If, for a given w any of our constraints is hurt, this quanitity will be ∞. If the constraints are satisfied, then $\Theta_P(w) = f(w)$. Thus, $\Theta_P$takes on the value of our objective if the constraints are satisfied and ∞if they are not satisfied. Therefore we can consider the following optimization problem

$$\underset{w}{min} \Theta_p(w) = \underset{w}{min} \underset{\alpha, \beta : \alpha_i \geq 0}{max} L(w, \alpha, \beta)$$

and see that it's our original optimization problem. We define the optimal value of the objective to be $p^* = \underset{w}{min} \Theta_p(w)$ and vall this the **value** of the primal problem.

Now we look at a different problem. We define:

$$\Theta_D(\alpha, \beta) = \underset{w}{min} L(w, \alpha, \beta)$$

Here, the D subscript stands for **dual**. We are now optimizing with respect to w, whereas we were optimizing with respect to α, β in the primal problem. We can now pose the dual optimization problem:

$$\underset{\alpha,\beta:\alpha_i \geq 0}{max} \Theta_D(w) = \underset{\alpha,\beta:\alpha_i \geq 0}{max} \underset{w}{min} L(w, \alpha, \beta)$$

We also define the optimal value of the dual problem's objective to be $d^* = max_{\alpha,\beta:\alpha_i \geq 0} \Theta_D(w)$. It can easily be show that

$$d^* = max_{\alpha,\beta:\alpha_i \geq 0} \Theta_D(w) \leq min \Theta_p(w) = p^*$$

However, under certain conditions we will have $d^* = p^*$. We suppose now that f and the g_i's are convex and the h_i's affine ($h_i(w) = a_i^T w + b$), linear with intercept term) and we assume that the constraints on $g_i$are feasible. Under these assumptions, there must exist w^*, α^*, β^*, so that $w^*$is the solution to the primal problem and $\alpha^*, \beta^*$are the solution to the dual problem and moreover $p^* = d^* = L(w^*, \alpha^*, \beta^*)$. Also, w^*, α^*, β^*satisfy the **Karush-Kuhn-Tucker (KKT) conditions**, which follow and if they satisfy them, then they're a solution to the primal and dual problems:

$$\frac{\partial}{\partial w_i} L(w^*, \alpha^*, \beta^*) = 0, \ i = 1, \ldots, n$$

$$\frac{\partial}{\partial \beta_i} L(w^*, \alpha^*, \beta^*) = 0, \ i = 1, \ldots, l$$

$$\alpha_i^* g_i(w^*) = 0, \ i = 1, \ldots, k$$

$$g(w^*) \leq 0, \ i = 1, \ldots, k$$

$$\alpha^* \geq 0, \ i = 1, \ldots, k$$

$\alpha_i^* g(w^*) = 0$ also means that if $\alpha_i^* > 0$, then $g_i(w^*) = 0$. We will use this later.

4.6 Optimal margin classifiers

Remember the primal optimization problem we posed for finding the optimal margin classifier:

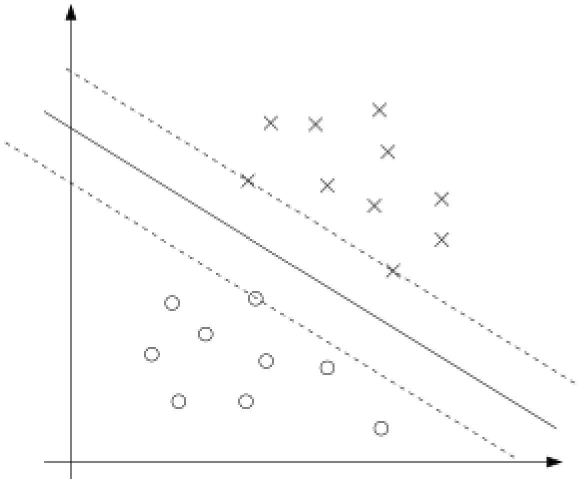
$$min_{\gamma,w,b}\frac{1}{2}||w||^2$$

$$s.t.\ y^{(i)}(w^Tx^{(i)}+b)\geq 1,\ i=1,\ldots,m$$

We can write the constraints as

$$g_i(w)=-y^{(i)}(w^Tx^{(i)}+b)+1\leq 0$$

We have one such constraint per training example. Note that from the KKT condition we mentioned, we will have $\alpha_i>0$ for only the training examples where the functional margin is exactly 1, because only there $g_i(w)=0$. Look at the figure below



You can see exactly three points on the dashed line which signals the points where the functional margin is equal to 1. The points on these lines (where the α_i are non-zero) are called **suport vectors**.

When we construct the Lagrangian for our optimization problem we have:

$$L(w,b,\alpha)=\frac{1}{2}||w||^2-\sum_{i=1}^{m}\alpha_i[y^{(i)}(w^Tx^{(i)}+b)-1]$$

We have only α_i, but no $\beta_i$because we have only inequality constraints. Let's find the dual form. We start by setting the derivatives of L with respect to w,b to zero:

$$\nabla_wL(w,b,\alpha)=w-\sum_{i=1}^{m}\alpha_iy^{(i)}x^{(i)}=0$$

This implies that

$$w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} = 0$$

For the deratives with respect to b we get

$$\frac{\partial}{\partial b} L(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i y^{(i)} = 0$$

If we put these back in the Lagrangian, simplify and remember the last result we got we obtain

$$L(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)}$$

Putting tis together with the constraint that $\alpha_i \geq 0$, we get the following dual optimization problem:

$$max_\alpha W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)}$$

$$s.t. \ \alpha_i \geq 0$$

$$\sum_{i=0}^{m} \alpha_i y^{(i)} = 0$$

The KKT conditions are all satisfied. We'll talk later about how to get the optimal values for the α_i. But if we had them we could solve for the optimal w and then, having this solve for the optimal value of the intercept b, that would be

$$b^* = -\frac{max_{i:y^{(i)}=-1} w^{*T} x^{(i)} + min_{i:y^{(i)}=1} w^{*T} x^{(i)}}{2}$$

Before we finish here we look at how we could make a prediction. Having fit the parameters, we would calculate $w^T x + b$ and predict $y = 1$iff it is bigger than 0. This quantity can also be written as

$$w^T x + b = \sum_{i=1}^{m} \alpha_i y^{(i)} < x^{(i)}, x > +b$$

where all but the suport vectors are zero. This will help making this algorithm fast and efficient.

4.7 Kernels

In the beginning we had our example where we had the living area as input x. We then considered using the features $x, x^2, x^3$and some cubic form. To be able to distinguish we will further call the original input value the **input attributes** of a problem. When they get somewhat mapped (could also just be 1:1) and then used in our learning algorithm we'll call those **input features**. We also denote $\phi$the **feature mapping**, which maps from the attributes to the features. In the mentioned example we had:

$$\phi(x) = \begin{bmatrix} x \\ x^2 \\ x^3 \end{bmatrix}$$

In our SVM, we may want to work with such mapped features. To do so we can simply use $\phi(x)$ instead of x. Since the algorithm has been written in terms of inner products $< x, z >$ we would replace these with $< \phi(x), \phi(z) >$. Given a feature mapping ϕ, we fedine the corresponding **Kernel** to be

$$K(x, z) = \phi(x)^T \phi(z)$$

Then, everywhere we had $< x, z >$in our algorithm we replace it with $K(x, z)$.

Now, given ϕwe could easily compute $K(x, z)$ by finding $\phi(x)$ and $\phi(z)$ and taking their inner product. But more interesting is that often, $K(x, z)$ may be very inexpensive to calculate even though $\phi(x)$ may be very expensive (e.g. because it's high-dimensional). By using an efficient way to calculate $K(x, z)$ in our algorithm, we can get the SVMs to learn in high dimensional feature space given by ϕwithout ever having to explicitly use $\phi(x)$.

Let's see an example. Suppose $x, z \in \mathbb{R}^n$and consider

$$K(x, z) = (x^T z)^2$$

We can also write this as

$$K(x, z) = \sum_{i,j=1}^{n} (x_i x_j)(z_i z_j)$$

Only calculating $\phi(x)$, here in the case of $n = 2$

$$\phi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_2 x_1 \\ x_2 x_2 \end{bmatrix}$$

Whereas calculating $K(x, z)$ only required $O(n)$ time, just calculating $\phi(x)$ took $O(n^2)$ time. Even though we mapped to a higher dimensional space, we can still work in linear time.

Now, let's talk about a different, intuitive view of kernels. Intuitively we might expect $K(x, z)$ to be large when $\phi(x)$ and $\phi(z)$ are close together (because the angle in between is small) and small when they are far apart. So we can think of the kernel $K(x,z)$ as a measurment of how similar $\phi(x)$ and $\phi(z)$ are. Given this intuition we might choose

$$K(x,z) = exp\left(-\frac{||x-z||^2}{2\sigma^2}\right).$$

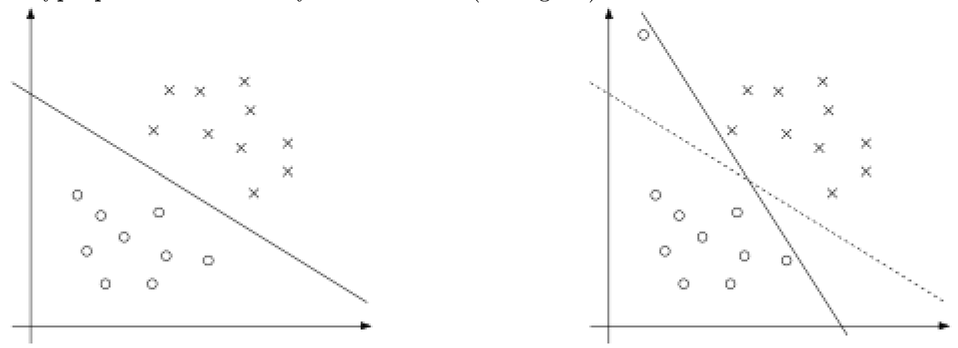
This kernel is called the **Gaussian kernel** and is a valid kernel. But how can we generally decide if a kernel is valid?

We denote the **Kernel matrix** to be the matrix defined by $K_{ij} = K(x^{(i)}, x^{(j)})$ where the $x^{(i)}$come from a set of points. Now, K is a valid kernel if and only if its matrix is *positive semi-definite.*

Shortly: Kernels let you work efficiently in high dimensional space.

4.8 Regularization and the non-seperable case

So far, we've only worked in cases where the data was linearly seperable. But this might not be the case or you could also just have small outliners that change your hyperplane dramatically to the worse (see figure)



To make the algorithm work for non-linearly seperable datasets as well as be less sensitive to such outliners, we reformulate our optimization using l_1-**regularization**:

$$min_{\gamma,w,b}\frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i$$

$$s.t.\ y^{(i)}(w^Tx^{(i)}+b) \geq 1-\xi_i,\ i=1,\dots,m$$

$$\xi_i \geq 0\ i=1,\dots,m$$

Examples are now allowed to have a functional margin less than 1, and if an example has such a functional margin $1-\xi_i$, we would pay a cost to our objective function being increased with $C\xi_i$. C controls the relative weighting between the two goals of keeping the objective function small and ensuring most examples have a functional margin of at least 1.

We can again form the Lagrangian:

$$L(w,b,\alpha) = \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i[y^{(i)}(w^Tx^{(i)}+b)-1+\xi_i] - \sum_{i=1}^{m} r_i\xi_i$$

The α_i's and r_i's are our Lagrange multipliers (≥ 0). We then obtain the dual form of the problem:

$$max_\alpha W(\alpha) = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} y^{(i)}y^{(j)}\alpha_i\alpha_j(x^{(i)})^Tx^{(j)}$$

$$s.t.\ 0 \leq \alpha_i \leq C$$

$$\sum_{i=0}^{m}\alpha_i y^{(i)} = 0$$

Applying the l_1-regulariztation, the only change here is that the $\alpha_i$also have to be less or equal to C. Also, the KKT dual-complementarity conditions are:

$$\alpha_i = 0 \Rightarrow y^{(i)}(w^Tx^{(i)}+b) \geq 1$$

$$\alpha_i = C \Rightarrow y^{(i)}(w^Tx^{(i)}+b) \leq 1$$

$$0 < \alpha_i < C \Rightarrow y^{(i)}(w^Tx^{(i)}+b) = 1$$

4.9 The SMO algorithm

The **sequential minimal algorithm SMO** gives an efficient way of solvinf the dual problem in SVMs.

4.9.1 Coordinate ascent

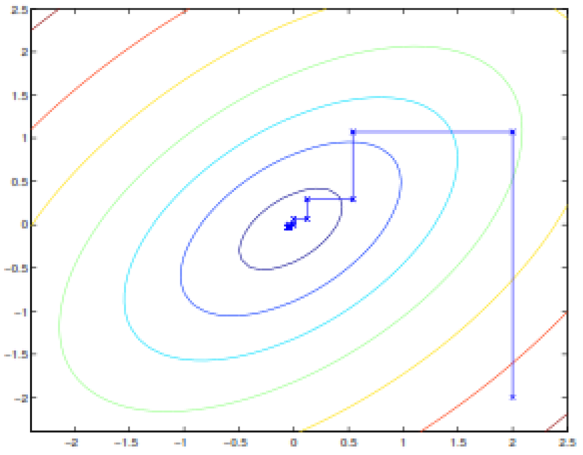
Consider trying to solve the unconstrained optimization problem

$$\underset{\alpha}{max} W(\alpha_1, \ldots, \alpha_m)$$

Here we think of W as some funciton of the parameters $\alpha_i$and ignore every ressemblance or similarity. We're going to consider a new algorithm called **coordinate ascent**:

```
Loop until convergence: {
    For $i = 1, \ldots, m${
        $\alpha_i := arg\,max_{\hat{\alpha}_i} W(\alpha_1, \ldots, \alpha_{i-1}, \hat{\alpha}_i, \alpha_{i+1}, \ldots, \alpha_m)$
    }
}
```

Thus, for each loop we hold all except α_i fixed and reoptimize W with respect to just the parameter α_i. The order in which we choose the $\alpha$to be optimized can be to our likings (e.g. the one we expect the largest increase for $W(\alpha)$). When the innermost $arg\,max$ can be calculated efficiently, coordinate ascent can be fairly efficient. Following is a figure of some coordinate ascent example:



4.9.2 SMO

Again, this is the dual optimization problem we want to solve:

$$max_{\alpha} W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)}$$

$$s.t.\ 0 \le \alpha_i \le C$$

$$\sum_{i=0}^{m} \alpha_i y^{(i)} = 0$$

Assume we have a set of α_i that satisfy this constraint. When we want to update some objects of the α_i, we have to update at least two of them simultaneously to keep satisfying the constraint. We will do the following in the SMO:

Repeat till convergence {

1. Select some pair $\alpha_i$and α_j to update next (using a heuristic that picks the two that will allow us to make the biggest progress)

2. Reoptimize $W(\alpha)$ with respect to $\alpha_i$and α_j while holding all other α_k fixed.

}

To check for convergence we can check whether the KKT conditions are satisfied within some tolerance convergence parameter *tol* which is typically set around 0.001 to 0.01.

The key to efficency is to be able to compute the update efficiently. Let's shortly sketch this update:

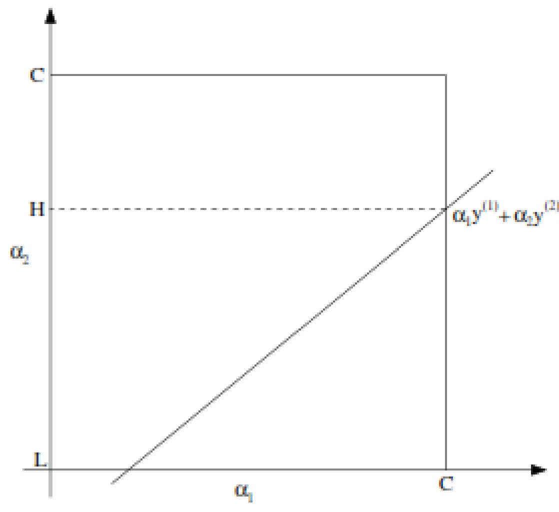
Let's say in our current step we want to optimize α_1, α_2. We hold all other fixed and we know that

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = -\sum_{i=3}^{m} \alpha_i y^{(i)}$$

The right side is fixed so we can denote it by some constant ζ:

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$$

We can picture the constraints as follows:



From the constraints, we know that $\alpha_1$and $\alpha_2$must lie within the box [0,C] x [0,C]. Also we know that they both must lie on the line plotted. We rewrite our equation a bit:

$$\alpha_1 = (\zeta - \alpha_2 y^{(2)}) y^{(1)}$$

Hence, the objective function can be written as

$$W(\alpha_1, \ldots, \alpha_m) = W((\zeta - \alpha_2 y^{(2)}) y^{(1)}, \alpha_2, \ldots, \alpha_m)$$

This is just some quadratic function in α_2, so we can express it as $a\alpha_2^2 + b\alpha_2 + c$ for some appropriate a,b and c. We can easily calculate the maximum value for this and denote it as $\alpha_2^{new,unclipped}$uf we ignore the box constraint. Now we just have to make sure it lies within the minimum and maximum it may take (clip inside the [L,H] interval):

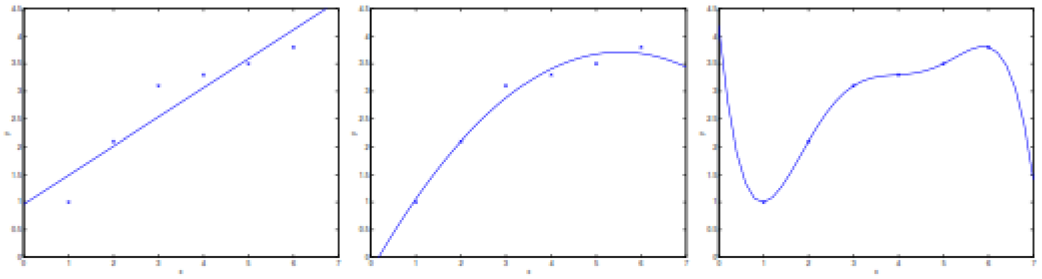
$$\alpha_2^{new} = \begin{cases} H & if\ \alpha_2^{new,unclipped} > H \\ \alpha_2^{new,unclipped} & if\ L \leq \alpha_2^{new,unclipped} \leq H \\ L & if\ \alpha_2^{new,unclipped} < L \end{cases}$$

Having this we can just plug it back into our equation and solve for α_1^{new}.

5 Learning Theory

5.1 Bias/variance tradeoff

Using linear regression, we might think about whether to use "simple" model like a linear one or more complex like the polynomial one. Consider the following example:



Fitting a 5th order polynomial does not result in a good model. Even though it would do very well on predicting training example it would do a bad job on new examples. It **generalizes** badly. The **generalization error** of a hypothesis is its expected error on examples not necessarily in the training set. Informally, we define the **bias** of a model to be the expected generalization error, even if we fit it to a huge training set. Thus, in the problem above, the linear model suffers from large bias and may underfit the data.

The second component to the generalization error, consisting of the **variance** of a model fitting procedure. Especially, when fitting high-order polynomial like the right-most figure, there is a large risk that we're fitting patterns in the data that happen to be only present in our training examples but do not reflect the relationship between x and y. By fitting these patterns we might again obtain a model with large generalization error. In this case, we would say the model has large variance.

Often, there is a tradeoff between bias and variance. If our model has too few parameters, it may have large bias and small variance, but if it has too many parameters, it may have small bias and large variance. In the figures above, the quadratic function does best in predicting new examples.

5.2 Preliminaries

This chapter will help us hone our intuition and derive rules of thumb about how to best apply learning algorithms in different settings. We will also try to

formalize the bias/variance tradeoff and answer other questions.
Lemma. Let $A_1, \dots, A_k$ be k different events (that may not be independent). Then

$$P(A_1 \cup \cdots \cup A_k) \leq P(A_1) + \cdots + P(A_k).$$

Lemma. Let $Z_1, \dots, Z_m$ be m independent and iid random variables drawn from a Bernoulli(ϕ) distribution. I.e. $P(Z_i = 1) = \phi$ and $P(Z_i = 1 - \phi)$. Let $\hat{\phi} = (1/m)\sum_{i=1}^{m} Z_i$ be the mean of these random variables and let any $\gamma > 0$ be fixed. Then

$$P(|\phi - \hat{\phi}| > \gamma) \leq 2exp(-2\gamma^2 m)$$

This lemma (which is also called the **Chernoff bound**) says that if we take $\hat{\phi}$-the average of m Bernoulli(ϕ) random variables- to be our estimate of ϕ, then the probability of being far from the true value is small, so long as m is large. You can also say that if you have a biased coin which has a chance of $\phi$for heads and you toss it m times and calculate the fraction of times it came up heads, tat will be a good estimate of ϕ(as long as m is large).

We will stick to binary classification in which the labels are $y \in \{0, 1\}$. All our conclusions can be generalized.

We assume that we are given a training set S of size m, where the training examples $(x^{(i)}, y^{(i)})$ are drawn iid from some probability distribution D. For a hypothesis h we define the **training error** (also called **emiprical risk** or **empirical error**) to be

$$\hat{}(h) = \frac{1}{m}\sum_{i=1}^{m} \hat{1}\{h(x^{(i)} \neq y^{(i)}\}$$

If we want to make the dependence of the training set clear we can also write $\hat{\epsilon}_S(h)$. We also define the generalization error to be

$$\epsilon(h) = P_{(x,y)\sim D}(h(x) \neq y).$$

This is the probability that, if we now draw a new example from a distribution D, we will misclassify it. Note that we assume that all our training data is drawn from the same distribution D.

Consider the setting of linear classification and let $h_\Theta(x) = 1\{\Theta^T x\}$. A way to fit the parameters $\Theta$is to mimimize the training error and pick

$$\hat{\Theta} = arg\ min_\Theta \hat{\epsilon}(h_\Theta)$$

We call this the **empirical risk minimization ERM** and the resulting hypothesis output by the learning algorithm is $\hat{h} = h_{\hat{\Theta}}$.

We will further abstract away from specific parameterization of hypotheses and from wether we're using a linear classifier. We define the **hypothesis class** H used by a learning algorithm to be the set of all classifiers considered by it. For linear classification $H = \{h_\Theta : h_\Theta(x) = 1\{\Theta^T x\}, \Theta \in \mathbb{R}^{n+1}\}$is the set of all classifiers over the domain of inputs X where the decision boundary is linear. If we were looking at, say, neural networks then we could let H be the set of all classifiers representable by some neural network architecture.

Empirical risk minimization can now be thought as the minimization over the class of functions H, in which the learning algorithm picks the hypothesis

$$\hat{h} = arg \min_{h \in H} \hat{\epsilon}(h)$$

5.3 The case of finite H

We consider now a learning problem in which we have a finite hypothesis class $H = \{h_1, \dots h_k\}$ consisting of k hypotheses. Thus, H is a set of functions mapping from X to $\{0,1\}$ and empirical risk minimization picks the one with the smallest error. To be able to give a guarrantee on the generalization error by first showing that $\hat{\epsilon}(h)$ is a reliable estate of $\epsilon(h)$ for all hypotheses h and second showing that this implies an upper bound on the generalization error on $\hat{h}$.

Take any fixed $h_i \in H$. Consider a random variable Z: We're going to sample $(x, y) \sim D$. Then we set $Z = 1\{h_i(x) \neq y\}$. In other words, Z indicates whether h_i misclassifies it. We also define $Z_j = 1\{h_i(x^{(j)} \neq y^{(j)}\}$.

The misclassification probability on a randomy drawn example $(\epsilon(h))$ is exactly the expected value of Z . Moreover, the training error can be written as

$$\hat{\epsilon}(h_i) = \frac{1}{m} \sum_{j=1}^{m} Z_j$$

Thus, $\hat{\epsilon}$ is exactly the mean of the m random variables Z_j that are drawn iid from a Bernoulli distribution with mean $\epsilon(h_i)$. Applying the heffding inequality, we obtain

$$P(|\epsilon(h_i) - \hat{\epsilon}(h_i)| > \gamma) \leq 2exp(-2\gamma^2 m)$$

This shows that $\epsilon(h_i)$ will be close to $\hat{\epsilon}(h_i)$. We want to show that this is true for all hypotheses. Let $A_i$denote the event that $|\epsilon(h_i) - \hat{\epsilon}(h_i)| > \gamma$. Through the union bond lemma and the equation above, we get that

$$P(\neg\exists h \in H : |\,\epsilon(h_i) - \hat{\epsilon}(h_i)| > \gamma) = 1 - 2k\, exp(-2\gamma^2 m)$$

What we did was, for particular values of m and γ. , given a bound on probability that $|\epsilon(h) - \hat{\epsilon}(h)| > \gamma$. Now, given γ and some $\delta > 0$, how large must m be before we can guarantee that with probability of at least $1 - \delta$, training error will be within $\gamma$of generalization error? By setting $\delta = 2k\, exp(-2\gamma^2 m)$ and solving for m we get that if

$$m \geq \frac{1}{2\gamma^2} log \frac{2k}{\delta},$$

then with probabilty of at least $1-\delta$, we have that $|\epsilon(h)-\hat{\epsilon}(h)| \leq \gamma$ for all hypotheses (or that the probability of the difference of the errors being greater than γ is at most δ). The training set size m that a certain method or algorithm requires to achieve a certain level of performance is also called the algorithm's **sample complexity**.

Similarly, we can also hold m and δ fixed and show that with probability $1-\delta$, we have that for all hypotheses

$$|\epsilon(h)-\hat{\epsilon}(h)| \leq \sqrt{\frac{1}{2m} log \frac{2k}{\delta}}.$$

We assume now that uniform convergence holds and define $h^* = arg\, min_{h \in H} \hat{\epsilon}(h)$ to be the best possible hypothesis in H. It makes sense to compare with h^*since this is the best possible hypothesis. Pick $\hat{h} = arg\, min_{h \in H} \hat{\epsilon}(h)$. We have now:

$$\epsilon(\hat{h}) \leq \epsilon(h^*) + 2\gamma$$

Theorem. Let $|H| = k$ and any m, δ be fixed. Then with probability of at least $1-\delta$, we have that

$$\epsilon(\hat{h}) \leq \left(\underset{h \in H}{min} \epsilon(h) \right) + 2\sqrt{\frac{1}{2m} log \frac{2k}{\delta}}$$

Note, that this also quantifies our statement about bias/variance. If we switch from our hypothesis class H to a bigger class $H'$with $H' \supseteq H$, then our first term (the bias) decreases (because we're taking the min from more hypotheses), but the second term (the variance) increases (because k gets bigger).

By holding γ and δfixed and solving for m, we can also obtain the complexity bound:

Corollary. Let $|H| = k$ and let any δ, γ be fiexed. Then for $\epsilon(\hat{h}) \leq min_{h \in H} \epsilon(h) + 2\gamma$ to hold with probability at least $1-\delta$, it suffices that

$$m \geq O\left(\frac{1}{\gamma^2} log \frac{k}{\delta} \right)$$

5.4 The case of inifinite H

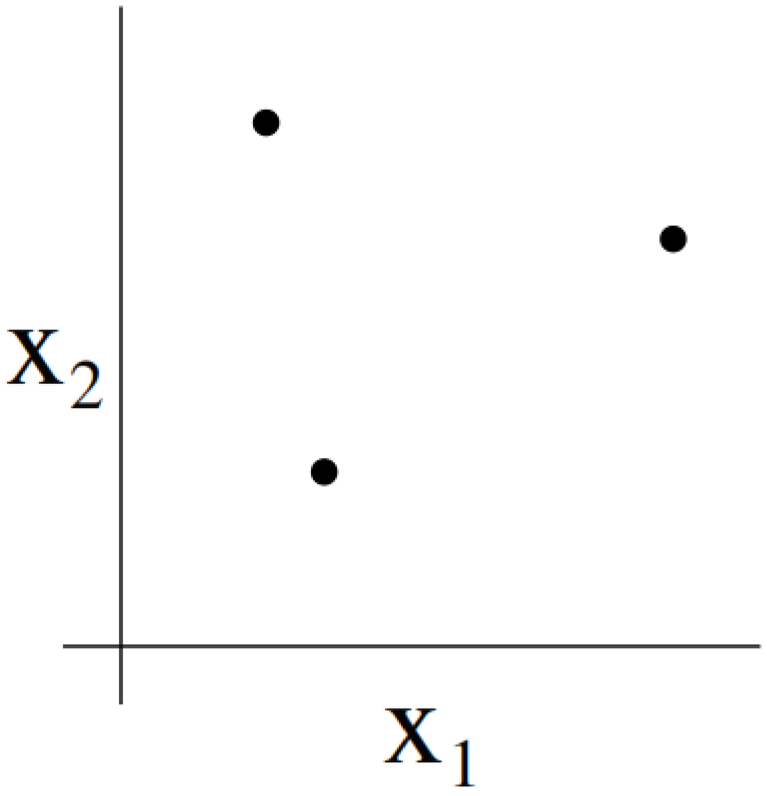
Many hypothesis classes actually contain an infinite number of functions. Suppose now, that we have an H, that is parameterized by d real numbers. We'd be using 64 bit to represent a float and this means our learning algorithm is parameterized by $64d$ bits. Thus our hypothesis class consists at most $k = 2^{64d}$different hypotheses. To guarantee that $\epsilon(\hat{h}) \leq \epsilon(h^*) + 2\gamma$ to hold with probability at least $1-\delta$, it suffices that $m = O_{\gamma,\delta}(d)$ (linear, parameterized by γ and δ). This means to do well we're going to need on the order of a linear number of training examples in$d$.

To derive a more satisfying argument, we need more definitions.

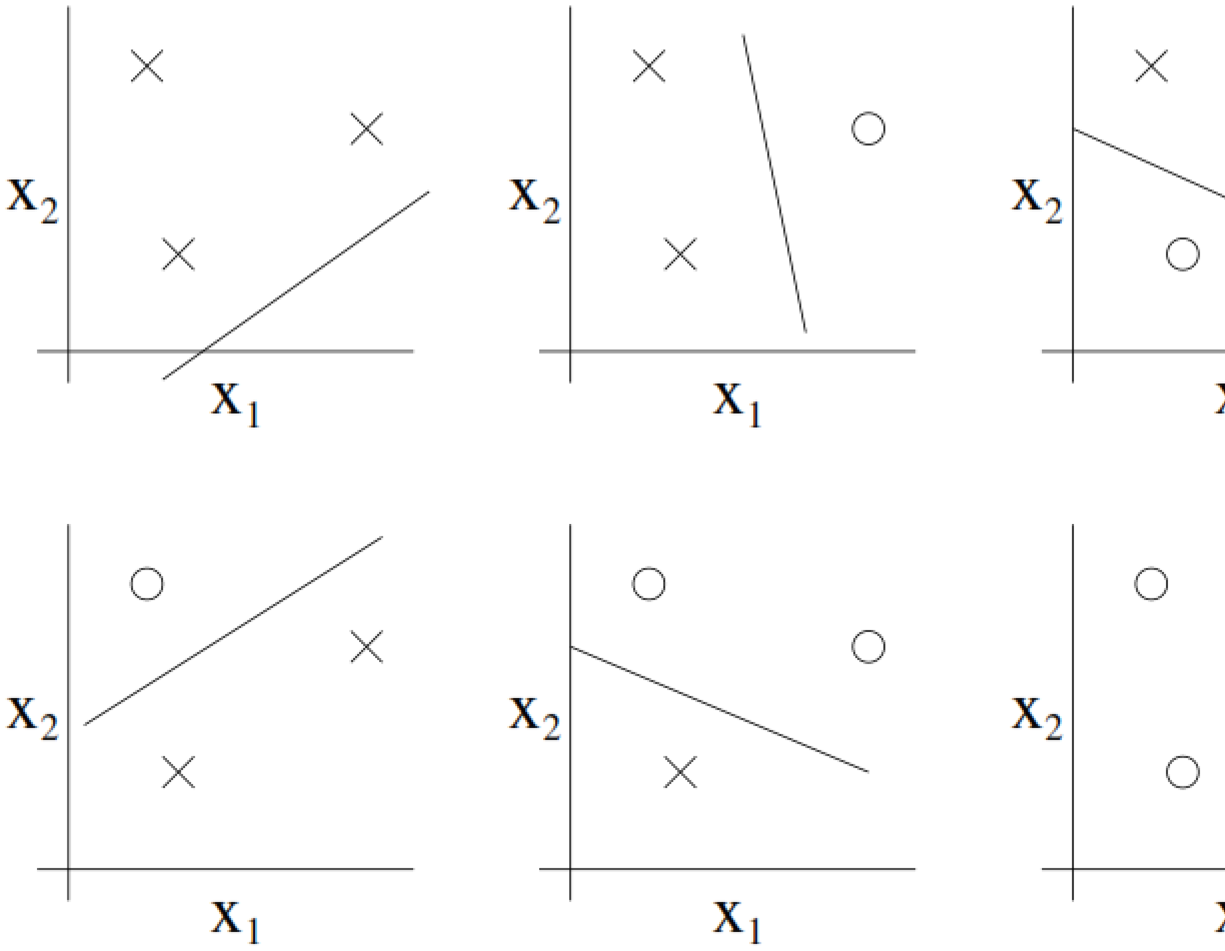
Given a set $S = \{x^{(i)}, \ldots, x^{(d)}\}$ (no relation to the training set) of points $x^{(i)} \in X$, we say that H **shatters** S, if H can realize any labeling on S. I.e. if for any set of labels $\{y^{(1)}, \ldots, y^{(d)}\}$, there exists some $h \in H$ so that $h(x^{(i)}) = y^{(i)}$for all $i = 1, \ldots, d$.

Given a hypothesis class H, we then define its **Vapnik-Chervonenkis dimension VC(H)**, to be the size of the largest set that is shattered by H.

For instance, consider the following figure:



Can the set H of linear classifiers in two dimensions ($h(x) = 1\{\Theta_0 + \Theta_1 x_1 + \Theta_2 x_2 \geq 0\}$) shatter the set above? Yes, it can:



But it is possible to show that there is no set of 4 points that this hypothesis class can shatter. Thus, VC(H)=3.

To prove that VC(H) is at least d, we need to show that there's at least *one* set of size d that H can shatter.

Theorem. Let H be given, and let $d = VC(H)$, Then with probability at least $1 - \delta$, we have that for all $h \in H$

$$|\epsilon(h) - \hat{\epsilon}(h)| \le O\left(\sqrt{\frac{d}{m} log \frac{m}{d} + \frac{1}{m} log \frac{1}{\delta}}\right)$$

And

$$|\epsilon(\hat{h}) \le \epsilon(h^*) + O\left(\sqrt{\frac{d}{m} log \frac{m}{d} + \frac{1}{m} log \frac{1}{\delta}}\right).$$

In other words, if a hypothesis class has finite VC dimension, then the uniform

convergence occurs as m becomes large.
Corollary. For $|\epsilon(h) - \hat{\epsilon}(h)| \leq \gamma$ to hold for all hypotheses with probability at least $1 - \delta$, it suffices that $m = O_{\gamma,\delta}(d)$.

In other words, the number of training examples to learn "well" using H ist linear in the VC dimension of H. It also turns out, that for "most" hypothesis classes, the VC dimension is also roughly linear in the number of parameters. So, the number of training examples needed is usually roughly linear in the number of parameters of H.

6 Regularization and model selection

How do we select among different models for a learning problem? How can we automatically select a model that represents a good tradeoff between bias and variance?

For the sake of concrectness, we assume we have some finite set of models $M = \{M_1, \dots, M_d\}$that we're trying to select among.

6.1 Cross validation

Given a training set S, our intuition might be to train each model on S and then choose the hypothesis with the smallest training error. This does not work. Consider choosing the order of a polynomial. This algorithm would always choose the highest order possible, leading to high variance.

A better algorithm is **hold-out cross validation** (also called **simple cross validation**):

1. Randomly split S into S_{train}(maybe 70% of the data) and S_{CV}(the remaining data). S_{VC} is called the hold-out cross validation set.

2. Train each model M_i on $S_{train}$to get some hypothesis h_i.

3. Select the hypothesis h_i that has the smallest error $\hat{\epsilon}_{S_{CV}}(h_i)$ on the hold-out corss validation set.

By testing on a set that the models were not trained on, we obtain a better estimate of our hypotheses' true generalization error. Usually somewhere between 25% and 33% of the data is used in the hold-out cross validation set. Optionally we can also make a fourth step where after we choose our model we retrain it on the entire training set S.

The disadvantage here is that we're "wasting" about 30% of our data. There will definitely be a point where we can only train of $0.7m$, instead of m training examples. So, in learning problems where we have few data (say, $m = 20$) we would rather use another method, called **k-fold cross validation**:

1. Randomly split S into k disjoint subsets of m/k training examples each. We'll call these subsets $S_1, \dots, S_k$.

2. For each model M_i, we evaluate it as follows:
   For $j = 1, \dots, k$
   Train the model $M_i$on all data except S_j to get some hypothesis h_{ij}. Test this hypothesis h_{ij} on S_j to get $\hat{\epsilon}_{s_j}(h_{ij})$.
   The estimated generalization error of the model M_j is then calculated as the average of the $\hat{\epsilon}_{S_j}(h_{ij})$ (averaged over j).

3. Pick the model M_i with the lowest estimated generalization error and retrain it on the whole training set. The resulting hypothesis is then output as our final answer.

A ususal choice for k as a number of folds would be around $k = 10$. While the fraction of data held out is now only $1/k$ -and therefore much smaller-, this method may also be more computionally expensive.

When we're using the extreme choice of $k = m$ to leave out as little data as possible, we call this method the **leave-one-out cross validation .**

These methods can also be used to evaluate a single model or algorithm. If we want to estimate our future performance, cross validation can give a reasonable way of doing so.

6.2 Feature selection

Imagine you have a supervised learning problem where the number of features n is very large but you suspect that only a small number of features is actually relevant. Even with a linear algorithm, overfitting would still be a potential problem unless the training set is fairly large. We want to choose only the features that are relevant to us. A possible way of doing so is called **forward search**:

1. Initialize $F = \emptyset$

2. Repeat {
   (a) For $i = 1, \dots, n$ if $i \notin F$, let $F_i = F \cup \{i\}$and use some version of cross validation to evaluate features F_i(I.e. train you algorithm only on the features in $F_i$and estimate its generalization error.)
   (b) Set F to be the best feature subset found on step (a).
   }

3. Select and output the best feature subset that was evaluated during the entire search procedure.

The outher loop can be done as many times as the number of features wanted or till all features are chosen. This algorithm is an instantiation of **wrapper model feature selection**, since it "wraps" around the algorithm and repeatedly calls the algorithm to evaluate how well it does with different subsets. Similarly we can also do **backward search** where we start with the entire set and delete one by one, till we are satisfied. These algorithms work usually quite well but can be computationally expensive.

Filter feature selection methods give heurisitc, but cheaper ways of chhosing feature subsets. The idea is to calculate a simple score that tells us how informative each fature is about the labels. Then we just choose the k features with the best scores.

In practice, a common way of doing so is to choose $S(i)$ to be the **mutual information** $MI(x_i, y)$ between $x_i$and y:

$$MI(x_i, y) = \sum_{x_i \in \{0,1\}} \sum_{y \in \{0,1\}} p(x_i, y) log \frac{p(x_i, y)}{p(x_i)p(y)}$$

Note, that this equation assumes that the $x_i$and y are binary-valued. The probabilities can be estimated according to their empirical distributions on the training set. Intuitively, this gives a measure of how different the probability distributions $p(x_i, y)$ and $p(x_i)p(y)$ are. If they are independent, then $p(x_i, y) = p(x_i)p(y)$ and the score would be small.

A way to choose how many features you want to use is again cross validation.

6.3 Bayesian statistics and regularization

This will be another tool against overfitting. For maximum likelihood, we chose our parameters according to

$$\Theta_{ML} = \underset{\Theta}{argmax} \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}; \Theta).$$

In our previous discussions, $\Theta$is not random, just unknown and it's our just to come up with procedures to estimate it. An alterative approach is the **Bayesian** view, where we think of Θas a random variable with unkown value. We would specify a **prior distribution** $p(\Theta)$ on Θthat expresses our prior beliefs. Given a training set, we can then calculate the posterior beliefs on the parameters

$$p(\Theta|S) = \frac{p(S|\Theta)p(\Theta)}{p(S)} = \frac{(\prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, \Theta))p(\Theta)}{\int_{\Theta}(\prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, \Theta)p(\Theta))d\Theta}$$

Here, $p(y^{(i)}|x^{(i)}, \Theta)$ comes from our model. For example, using Bayesian logistic regression: $p(y^{(i)}|x^{(i)}, \Theta) = h_\Theta(x^{(i)})^{y^{(i)}}(1 - h_\Theta(x^{(i)}))^{(1-y^{(i)}}$. To make a prediction we can compute the posterioir distribution on the class label:

$$p(y|x, S) = \int_{\Theta} p(y|x, \Theta)p(\Theta|S)d\Theta$$

This procedure is thought of as "fully Bayesian" prediction. Unfortunatelly, it is usually very computationally expensive. Thus, in praxis we will usually approximate the posterior distribution for Θ. The **MAP (maximum a posteriori)** estimate for Θ is given by

$$\Theta_{MAP} = \underset{\Theta}{argmax} \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, \Theta)p(\Theta).$$

7 k-means

In a clustering problem we are given a training set and want to group the data into cohesive clusters. $x^{(i)} \in \mathbb{R}^n$as usual but no labels $y^{(i)}$ are given. This means it's an unsupervised learning probelm. The k-means clustering algorithm is as follows:

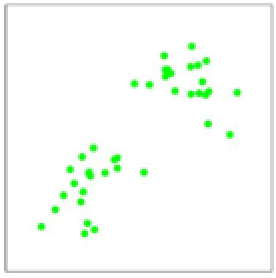
1. Initialize **cluster centroids** $\mu_1, \ldots, \mu_k$randomly.
2. Repeat until convergence: $\{$
   For every i, set

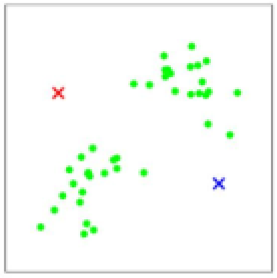
   $$c^{(i)} = \underset{j}{argmin} ||x^{(i)} - \mu_j||^2$$

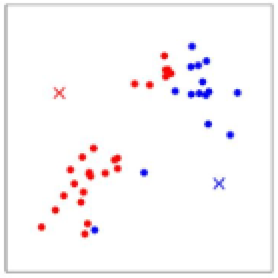
   Assign the training example to the closest cluster centroid.
   For each j, set

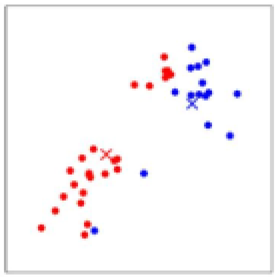
   $$\mu_j = \frac{\sum_{i=1}^{m} 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{c^{(i)} = j\}}$$

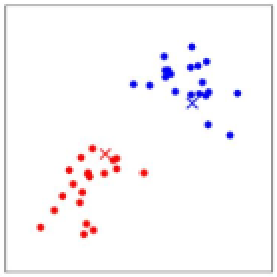
   Move the cluster centroids to the mean of their assigned training examples.

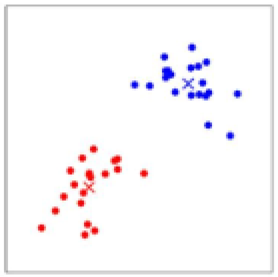

(a)


(b)


(c)


(d)


(e)


(f)

Let us define the **distortion function** to be:

$$J(c, \mu) = \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

Thus, J measures the sum of euclidean distances between each training examples and its assigned cluster centroid. It can be shown that k-means is coordinate descent on J. The inner loop of the k-means algorithm minimizes J with respect to c while holding μfixed and the then minimizes J with respect to μwhile holding c fixed. Thus, J must monotonically decrease and the value of J must converge.

The distortion function J is a non-convex function, what means that it's not guaranteed to converge in a global minimum but can also converge in a local minimum. Therefore if you're worried about getting stuck in a bad local minimum, a common thing is to run k-means many times and then pick the one that gives the lowest distortion $J(c, \mu)$.

8 Mixture of Gaussians and the EM algorithm

We are in an unsupervised learning setting (no labels) and have a training set as usual. We want to model the data by specifying a joint distribution $p(x^{(i)}, z^{(i)}) = p(x^{(i)}|z^{(i)})p(z^{(i)})$. Here, $z^{(i)} \sim Multinomial(\phi)$ where $\phi_j = p(z^{(i)} = j)$ and $x^{(i)}|z^{(i)} \sim N(\mu_j, \Sigma_j)$. k denotes the number of values that the $z^{(i)}$can take on. Thus, our model posits that each $x^{(i)}$was generated by randomly choosing $z^{(i)}$from $\{1, \ldots, k\}$ and then $x^{(i)}$was drawn from one of the k Gaussians, depending on $z^{(i)}$. This is called the **mixture of Gaussians** model.

The parameters of our models are ϕ, μ, Σ and the likelihood of our data

$$l(\phi, \mu, \Sigma) = \sum_{i=1}^{m} log \sum_{z^{(i)}=1}^{k} p(x^{(i)}|z^{(i)}; \mu, \Sigma)p(z^{(i)}, \phi).$$

The problem now is that we can't find the maximum likelihood estimates in closed form by setting the derivative to zero.

The random variable $z^{(i)}$ indicate which of the k Gaussians each $x^{(i)}$had come from. Now our problem is that we don't know the $z^{(i)}$'s . We can now use the EM algorithm.

In the E-step it tries to guess the values of the $z^{(i)}$'s. In the M-step we pretend that the guesses are correct and then we can maximize:

Repeat until convergence: {

(E-step) For each i, j, set:

$$w_j^{(i)} := p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma)$$

(M-step) Update the parameters:

$$\phi_j := \frac{1}{m} \sum_{i=1}^{m} w_j^{(i)}$$

$$\mu_j := \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}}$$

$$\Sigma_j := \frac{\sum_{i=1}^{m} w_j^{(i)}(x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\Sigma_{i=1}^{m} w_j^{(i)}}$$

In the E-step, we calculate the posterior probability of the $z^{(i)}$ given the $x^{(i)}$and using the current setting of our parameters, we obtain:

$$p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma) = \frac{p(x^{(i)}|z^{(i)} = j; \mu, \Sigma)p(z^{(i)} = j; \phi)}{\sum_{l=1}^{k} p(x^{(i)}|z^{(i)} = l; \mu, \Sigma)p(z^{(i)} = l; \phi)}$$

Here, $p(x^{(i)}|z^{(i)} = j; \mu, \Sigma)$ is given by evaluating the density of a gaussian with mean μ_j and covariance $\Sigma_j$at $x^{(i)}$; $p(z^{(i)} = j; \phi)$ is given by $\phi_j$and so on. The values $w_j^{(i)}$calculated in the E-step represent the soft guesses for the values of $\underline{z^{(i)}}$.

The difference to the k-means algorithm is where we had hard assignments for the cluster centroids, we now have soft assignements. It is also susceptible to local minima.

9 The EM algorithm

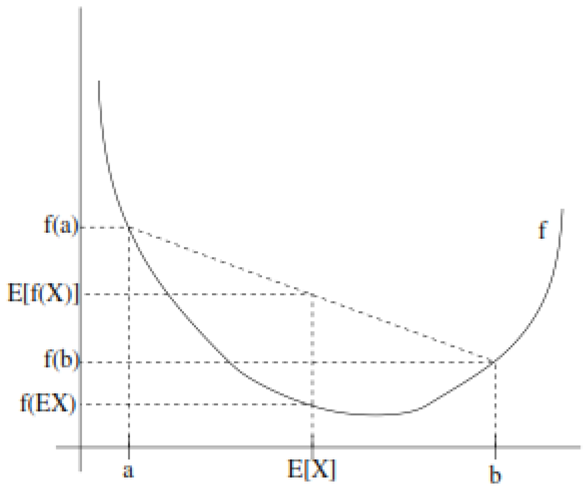
9.1 Jensen's inequality

Let f be a function whose domain is the set of the real numbers. We say that f is convex if $f'''(x) \geq 0$ for all values of x. For vector-valued x, this can be generalized to its hessian H being positive semi-definite. f is **strictly convex** if $f'''(x) > 0$ for all x or if H is strictly positive semi-definite.

Theorem. Let f be a convex function and let X be a random variable. Then

$$E[f(X)] \geq f(E[X])$$

Moreover, if f is strictly convex, then $E[f(X)] = f(E[X])$ only if $X = E[X]$ with probability 1.



Remark: This works the same way with concave function but the $\geq$is exchanged with $\leq$.

9.2 The EM algorithm

Suppose we have an estimation problem with a training set consisting of m independent examples. We wish to fit the parameters of a model $p(x,z)$ to the data, where the likelihood is given by

$$l(\Theta)=\sum_{i=1}^{m} log \sum_{z} p(x,z;\Theta)$$

If the $z^{(i)}$were observed, this wouldn't be much of a problem. Now, EM algorithm gives an efficient method for maximum likelihood estimation. Our strategy will be to repeatedly construct a lower-bound on l (E-step) and then optimize that lower-bound (M-step).

For each i, let $Q_i$be some distribution over the z's ($\sum_z Q_i(z)=1; Q_i\geq 0$). Consider:

$$\sum_{i} log\, p(x^{(i)};\Theta)\geq \sum_{i}\sum_{z^{(i)}} Q_i(z^{(i)}) log\frac{p(x^{(i)},z^{(i)};\Theta)}{Q_i(z^{(i)})}$$

Now, for any set of distributions Q_i, this formula gives a lower bound on $l(\Theta)$. It seems natural to make the lower-bound tight at the value of Θ. To make the bound tight, we need for the step involving Jensens ineuqality to hold with equality at our value of Θ. We require that

$$\frac{p(x^{(i)},z^{(i)};\Theta)}{Q_i(z^{(i)})}=c$$

for some constant c that does not depend on $z^{(i)}$. This is easily accomplished by choosing

$$Q_i(z^{(i)}) \propto p(x^{(i}, z^{(i)}, \Theta)$$

Because $\sum_z Q_i(z^{(i)}) = 1$, we know that

$$Q_i(z^{(i)}) = p(z^{(i)}|x^{(i)}; \Theta)$$

Thus, we can simply set the Q_i's to be the posterior distribution of the $z^{(i)\prime}s$ given $x^{(i)}$and the setting of the parameter Θ. Now, for the choice of the Q_i, we got an equation before that gives a lower-bound on the loglikelihood we're trying to maximize. This is the E-setp. In the M-step, we then maximize our formula with repsect to obtain a new setting of the Θ's. Repeatedly carrying these two steps out gives us the EM-algorithm:

```
Repeat until convergence {
    (E-step) For each $i$, set
        $Q_i(z^{(i)}) := p(z^{(i)}|x^{(i)}; \Theta)$
    (M-step) Set
        $\Theta := argmax_{\Theta} \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) log \frac{p(x^{(i)}, z^{(i)}; \Theta)}{Q_i(z^{(i)})}$
}
```

Through Jensen's inequaliy we know that $l(\Theta^{(t)}) \leq l(\Theta^{(t+1)})$ and therefore that the EM monotonically improves the log-likelihood. EM also causes the likelihhod to converge monotonically. A good moment to stop repeating the steps could be when we don't improve more than some tolarance parameter.

9.3 Mixture of Gaussians revisited

Let's go briefly back to the MoG. Following our algorithm derivation above, for the E-step we simply calculate

$$w_j^{(i)} = Q_i(z^{(i)} = j) = P(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma)$$

Here, the middle part of the equation denotes the probability of $z^{(i)}$taking the value j under the distribution Q_i. Next, in the M-step, we need to maximize with respect to our parameters ϕ, μ, Σ, the quantity

$$\sum_{i=1}^{m} \sum_{z^{(i)}} Q_i(z^{(i)}) log \frac{p(x^{(i)}, z^{(i)}; \phi, \mu, \Sigma)}{Q_i(z^{(i)})} = \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} log \frac{\frac{1}{(2\pi)^{n/2}|\Sigma_j|^{1/2}} exp(-\frac{1}{2}(x^{(i)} - \mu_j)^T \Sigma_j^{-1} (x^{(i)} - \mu_j)) \cdot \phi_j}{w_j^{(i)}}$$

Let's maximize with respect to μ_l. If we take the derivative with respect to $\mu_l$we get:

$$\nabla_{\mu_l} \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} log \frac{\frac{1}{(2\pi)^{n/2}|\Sigma_j|^{1/2}} exp(-\frac{1}{2}(x^{(i)} - \mu_j)^T \Sigma_j^{-1}(x^{(i)} - \mu_j)) \cdot \phi_j}{w_j^{(i)}} = \sum_{i=1}^{m} w_l^{(i)} (\Sigma_l^{-1} x^{(i)} - \Sigma_l^{-1} \mu_l)$$

Setting this to zero and solving for $\mu_l$yields the update rule

$$\mu_l = \frac{\Sigma_{i=1}^{m} w_l^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_l^{(i)}}$$

We'll leave out the rest, letting this part serve as an example.

10 Factor analysis

When we have data that comes from a mixture of Gaussians, the EM algorithm can be applied to fit a mixture model. In this setting we usually have problems wth sufficient data to discern the multiple-Gaussians structure ($m \gg n$, much more data than features). Imagine a setting where you have much more features than data ($n \gg m$). It might be difficult to model the data even with a single gaussian, much more with a mixture of gaussians. Using the same maximum likelihood estimators as in EM-MoG, we would get a singular covariance matrix. Therefore its inverse does not exist and the density of a multivariate Gaussian distribution can not be calculated.

Generally spoken, unless m exceeds n by some reasonable amount, the maximum likelihood estimators of the mean and covariance may be poor. In this chapter we will show a way to fit a reasonable Gaussian model to such data.

10.1 Restrictions of Σ

If we do not have sufficient data we may place some restrictions on the space of matrices Σwe will consider. For instance, we might consider to fit a covariance matrix Σ that is diagonal. The maximum likelihood estimate of the covariance matrix is given by the diagonal matrix Σ satisfying

$$\Sigma_{ij} = \frac{1}{m} \sum_{i=1}^{m} (x_j^{(i)} - \mu_j)^2$$

Thus, $\Sigma_{ij}$is just the empirical estimate of the variance of the j-th coordinate of the data. Recall that the contours of a Gaussian data are ellipses. A diagonal Σ corresponds to a Gaussian where the major axes of these ellipses are axis-algined.

Smoetimes, we may place a further restriction on the covariance matrix that not only must it be diagonal but its diagonal entries must all be equal. In this setting, we have $\Sigma = \sigma^2 I$ where $\sigma^2$is the parameter under our control. The maximum likelihood estimate of σ^2can be found to be:

$$\sigma^2 = \frac{1}{mn}\sum_{j=1}^{n}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$$

This corresponds to Gaussians whose densities have contours that are circles (or spheres/hyperplanes in higher dimensions as 2).

Without the constraints we needed $m \geq n{+}1$ to be sure that $\Sigma$is not singular. Under both of our two constraints we may obtain non-singular Σ when $m \geq 2$.

However, with these restrictions we also model the coordinates $x_i, x_j\ i \neq j$ to be uncorrelated and independent. Often, there are some interesting correlations that we want to capture. Using these restrictions would leave us unable to do so.

10.2 Marginals and conditionals of Gaussians

Suppose we have a vector-valued random variable

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

where $x_1 \in \mathbb{R}^r, x_2 \in \mathbb{R}^s, x \in \mathbb{R}^{r+s}$. Suppose $x \sim N(\mu, \Sigma)$, where

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix},\ \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$$

Here, $\mu_1 \in \mathbb{R}^r, \mu_2 \in \mathbb{R}^s, \Sigma_{11} \in \mathbb{R}^{rxr}, \Sigma_{12} \in \mathbb{R}^{rxs}$and so on. Note that since covariance matrices are symmetric, $\Sigma_{12} = \Sigma_{21}^T$.

Under our assumptions, $x_1$and x_2 are jointyl multivariate Gaussians. It is not hard to see that $E[x_1] = \mu_1$ and $Cov(x) = E[(x_1 - \mu_1)(x_1 - \mu_1)] = \Sigma_{11}$

Since marginal distributions of Gaussians are themselves Gaussian, we therefore have that the marginal distribution of $x_1$is given by $x_1 \sim N(\mu_1, \Sigma_{11})$. It can also be shown that $x_1|x_2 \sim N(\mu_{1|2}, \Sigma_{1|2})$, where

$$\mu_{1|2} \ = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2)$$

$$\Sigma_{1|2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}$$

10.3 The Factor analysis model

We posit a joint distribution on (x, z) as follows, where $z \in \mathbb{R}^k$ is a latent random variable:

$$z \sim N(0, I)$$

$$x|z \sim N(\mu + \Lambda z, \psi)$$

Here, the parameters of our model are the vector $\mu \in \mathbb{R}^n$, the matrix $\Lambda \in \mathbb{R}^{nxk}$, and the diagonal matrix $\psi \in \mathbb{R}^{nxn}$. The value of k is usually chosen to be smaller than n.

Thus, we imagine that each datapoint $x^{(i)}$is generated by sampling a k dimension multivariate Gaussian $z^{(i)}$. Then, it is mapped to a k-dimensional affine space by computing $\mu+\Lambda z^{(i)}$. Lastly $x^{(i)}$is generated by adding covariance noise ψ to this computation.

Equivalently, we can therefore also define the factor analysis model according to

$$z \sim N(0, I)$$

$$\epsilon \sim N(0, \psi)$$

$$x = \mu + \Lambda z + \epsilon$$

where ϵ and z are independent.

What distribution does our model exactly define? Our random variables z and x have a joint Gaussian distribution

$$\begin{bmatrix} z \\ x \end{bmatrix} \sim N(\mu_{zx}, \Sigma)$$

We will now find μ_{zx} and Σ.

We know that $E[z] = 0$ from the fact that $z \sim N(0, I)$. Also, we have that

$$E[x] = E[\mu + \Lambda z + \epsilon] = \mu + \Lambda E[z] + E[\epsilon] = \mu$$

Putting these together, we otain

$$\mu_{zx} = \begin{bmatrix} \overrightarrow{0} \\ \mu \end{bmatrix}$$

Next, to find Σ we need to calculate $\Sigma_{zz} = E[(z-E[z])(z-E[z])^T]$, $\Sigma_{zx} = E[(z-E[z])(x-E[x])^T]$, $\Sigma_{xx} = E[(x-E[x])(x-E[x])^T]$ (the upper-left, upper-right and lower-right block). Now, since $z \sim N(0, I)$, we find that $\Sigma_{zz} = Cov(z) = I$. Also,

$$\Sigma_{zz} = E[(z-E[z])(x-E[x])^T] = E[z(\mu+\Lambda z+\epsilon-\mu)^T] = E[zz^T]\Lambda^T+E[z\epsilon^T] = \Lambda^T$$

Similarly, we can find

$$\Sigma_{xx} = E[(x-E[x])(x-E[x])^T] = E[(\mu+\Lambda z+\epsilon-\mu)(\mu+\Lambda z+\epsilon-\mu)^T] = E[\Lambda zz^T\Lambda^T+\Lambda z\epsilon^T+\Lambda^T z^T\epsilon+\epsilon\epsilon^T] = \Lambda E[zz^T$$

Putting everything together, we therefore have that

$$\begin{bmatrix} z \\ x \end{bmatrix} \sim N\left(\begin{bmatrix} \vec{0} \\ \mu \end{bmatrix}, \begin{bmatrix} I & \Lambda^T \\ \Lambda & \Lambda\Lambda^T + \psi \end{bmatrix}\right)$$

We also get that the marginal distribution of x is given by $x \sim N(\mu, \Lambda\Lambda^T + \psi)$. Thus, given a training set, we can write down the log likelihood of the parameters:

$$l(\mu, \Lambda, \psi) = log \prod_{i=1}^{m} \frac{1}{(2\pi)^{n/2}|\Lambda\Lambda^T + \psi|} exp\left(-\frac{1}{2}(x^{(i)} - \mu)^T(\Lambda\Lambda^T + \psi)^{-1}(x^{(i)} - \mu)\right).$$

10.4 EM for factor analysis

Maximizing the formula above is quite hard. Therefore we will instead use the EM algorithm, which we're gonna derive in this section.

The derivation for the E-step is easy. We need to compute $Q_i(z^{(i)}) = p(z^{(i)}|x^{(i)}; \mu, \Lambda, \psi)$. By inserting our previous results into the formulas for finding the conditional distribution of a Gaussian, we find that $z^{(i)}|x^{(i)}; \mu, \Lambda, \psi \sim N(\mu_{z^{(i)}|x^{(i)}}, \Sigma_{z^{(i)}|x^{(i)}})$,where

$$\mu_{z^{(i)}|x^{(i)}} = \Lambda^T(\Lambda\Lambda^T + \psi)^{-1}(x^{(i)} - \mu)$$

$$\Sigma_{z^{(i)}|x^{(i)}} = I - \Lambda(\Lambda\Lambda^T + \psi)^{-1}\Lambda$$

So, using these definitions, we have

$$Q_i(z^{(i)}) = \frac{1}{(2\pi)^{k/2}|\Sigma_{z^{(i)}|x^{(i)}}|^{1/2}} exp\left(-\frac{1}{2}(z^{(i)} - \mu_{z^{(i)}|x^{(i)}})^T \Sigma^{-1}_{z^{(i)}|x^{(i)}}(z^{(i)} - \mu_{z^{(i)}|x^{(i)}})\right)$$

Now to the M-step. We need to maximize

$$\sum_{i=1}^{m} \int_{z^{(i)}} Q_i(z^{(i)}) log \frac{p(x^{(i)}, z^{(i)}; \mu, \Lambda, \psi)}{Q_i(z^{(i)})} dz^{(i)}$$

with respect to the parameters μ, Λ, ψ. We'll only do $\Lambda$for now.

We can simplify the equation above to

$$\sum_{i=1}^{m} E_{z^{(i)} \sim Q_i}[log\, p(x^{(i)}|z^{(i)}; \mu, \Lambda, \psi) + log\, p(z^{(i)}) - log\, Q_i(z^{(i)})]$$

The "$z^{(i)} \sim Q_i$" subscript indicates that the expectation is with respect to $z^{(i)}$drawn from Q_i. We will leave this out if not necessary. Dropping terms that do not depend on our parameters, we find that we need to maximize

$$\sum_{i=1}^{m} E[log\, p(x^{(i)}|z^{(i)}; \mu, \Lambda, \psi)] = \sum_{i=1}^{m} E\left[-\frac{1}{2}log|\psi| - \frac{n}{2}log(2\pi) - \frac{1}{2}(x^{(i)} - \mu - \Lambda z^{(i)})^T\psi^{-1}(x^{(i)} - \mu - \Lambda z^{(i)})\right]$$

We maximize this now with respect to Λ. Taking the derivatives on the parts depending on $\Lambda$and using some trace rules, we get:

$$\nabla_\Lambda \sum_{i=1}^{m} -E\left[\frac{1}{2}(x^{(i)}-\mu-\Lambda z^{(i)})^T\psi^{-1}(x^{(i)}-\mu-\Lambda z^{(i)})\right] = \sum_{i=1}^{m} E[-\psi^{-1}\Lambda z^{(i)}z^{(i)^T} + \psi^{-1}(x^{(i)}-\mu)z^{(i)^T}]$$

Setting this to zero and simplifying, we get:

$$\sum_{i=1}^{m} \Lambda E_{z^{(i)}\sim Q_i}\left[z^{(i)}z^{(i)^T}\right] = \sum_{i=1}^{m}(x^{(i)}-\mu)E_{z^{(i)}\sim Q_i}[z^{(i)^t}]$$

Solving for Λ, we obtain

$$\Lambda = \left(\sum_{i=1}^{m}(x^{(i)}-\mu)E_{z^{(i)}\sim Qi}\left[z^{(i)T}\right]\right)\left(E_{z^{(i)}\sim Qi}\left[z^{(i)}z^{(i)T}\right]\right)^{-1}$$

This is quite the close relationship to the normal equation we got for least squares regression ("$\Theta^T = (y^TX)(X^TX)^{-1}$. The analogy here is, that the x's are a linear function of the z's. Given the "guesses" for the z that the E-step has found, we will now try to estimate the unknown linearity Λrelating the x's and z's. To complete our M-step update, lets work out the values of the expectations we got above. From our definition of Q_i being Gaussian with mean $\mu_{z^{(i)}|x^{(i)}}$, we easily find

$$E_{z^{(i)}\sim Qi}\left[z^{(i)T}\right] = \mu^T_{z^{(i)}|x^{(i)}}$$

$$E_{z^{(i)}\sim Qi}\left[z^{(i)}z^{(i)T}\right] = \mu_{z^{(i)}|x^{(i)}}\mu^T_{z^{(i)}|x^{(i)}} + \Sigma_{z^{(i)}|x^{(i)}}$$

Substituting this back into our previous equation, we get the M-step update for Λ:

$$\Lambda = \left(\sum_{i=1}^{m}(x^{(i)}-\mu)\mu^T_{z^{(i)}|x^{(i)}}\right)\left(\sum_{i=1}^{m}\mu_{z^{(i)}|x^{(i)}}\mu^T_{z^{(i)}|x^{(i)}} + \Sigma_{z^{(i)}|x^{(i)}}\right)^{-1}$$

It is important to note that there is the term $\Sigma_{z^{(i)}|x^{(i)}}$.This is the covariance in the posterior distribution $p(z^{(i)}|x^{(i)})$. The M-step must take this uncertainity about $z^{(i)}$into account. A common mistake in deriving EM is to assume that in the E-step we only need to calculate the expectation $E[z]$ of the latent random variable z and then plug it into the optimization in the M-step everywhere z occurs. While this works in simple problems such as the mixture of gaussians, we saw here that $E[zz^T]$ and $E[z]E[z]^T$ differ by $\Sigma_{z|x}$. Thus, the M-step update must take the covariance of z in the posterior distribution $p(z^{(i)}|x^{(i)})$into account.Without making the entire derivation:

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

Since this doesn't change as the parameters are varied (only dependent on the $x^{(i)}$), we can compute this only once.

$$\Phi = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}x^{(i)^T} - x^{(i)}\mu^T_{z^{(i)}|x^{(i)}}\Lambda^T - \Lambda\mu_{z^{(i)}|x^{(i)}}x^{(i)^T} + \Lambda(\mu_{z^{(i)}|x^{(i)}}\mu^T_{z^{(i)}|x^{(i)}} + \Sigma_{z^{(i)}|x^{(i)}}\Lambda^T$$

We then set $\psi_{ii} = \Phi_{ii}$to extract the diagonal entries.

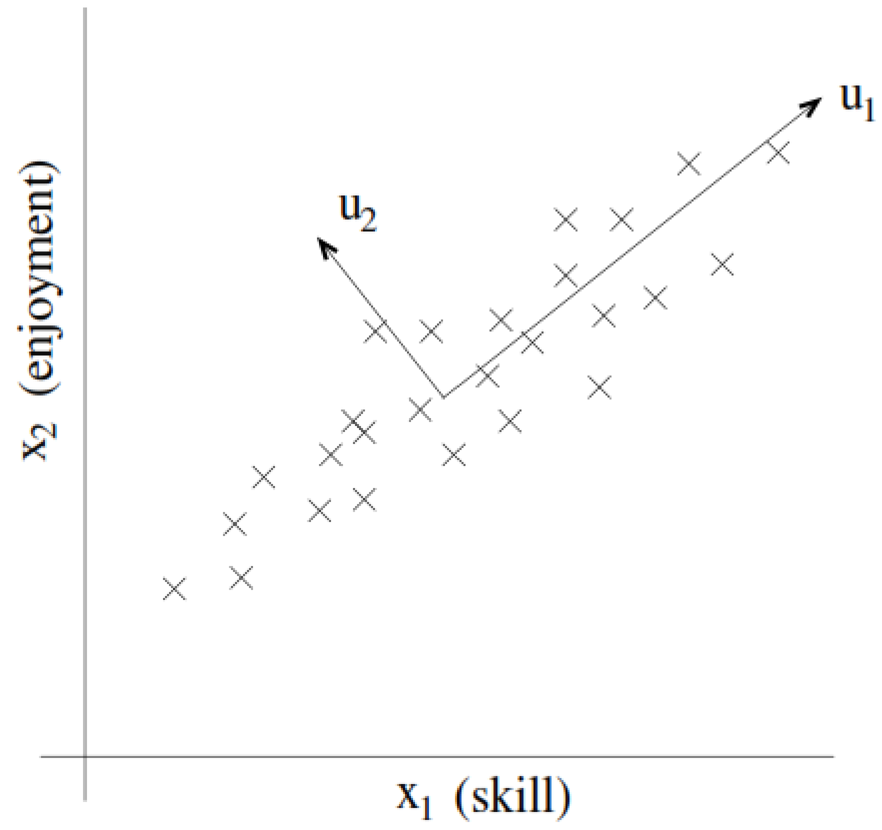
11 Principal component analysis (PCA)

In the factor analysis discussion, we gave a way to model data as "approximately" lying in some k-dimension subspace where $k \ll n$. We imageined that each point $x^{(i)}$ was created by first generating some $z^{(i)}$ lying in the k-dimension affine space $\{\Lambda z+\mu$;z$\in \mathbb{R}^{k}\}$, and then adding Φ-covariance noise. This is based on probability and parameter estimation used the EM algorithm.

In the PCA, we will also try to identify the subspace in which the data approximately lies. However, here we're going to do this more directly and instead of EM we'll only use eigenvector calculation.

Suppose we're given a dataset $\{x^{(i)}; i = 1, \dots, m\}$ of attributes of m different types of cars. These attributes (x_j) are attributes such as turn radius, maximum speed, etc. Unknown to us is, that two different attributes - some $x_i$and x_j- give a car's maximum speed in kmh, respectively mph. That means we have two attributes that are almost linearly dependent. Thus, the overall data lies approximately on an $n-1$ subspace (we can "cut away" one of max speeds and don't lose any information when modeling).

Another exmaple: Consider a dataset from a survey with pilots of radio-controlled helicopters where $x_1^{(i)}$measures the skill of pilot i and $x_2^{(i)}$how much pilot i enjoys fyling. Because RC helicopters are quite difficult to fly, only the most commited pilots become actually good pilots. Therefore we can assume, that the attributes x_1 and x_2 are strongly correlated. We might posit that the data lies in some subspace along an axis (the $u_1$direction in the figure below) capturing the intristic piloting "karma" of a person:



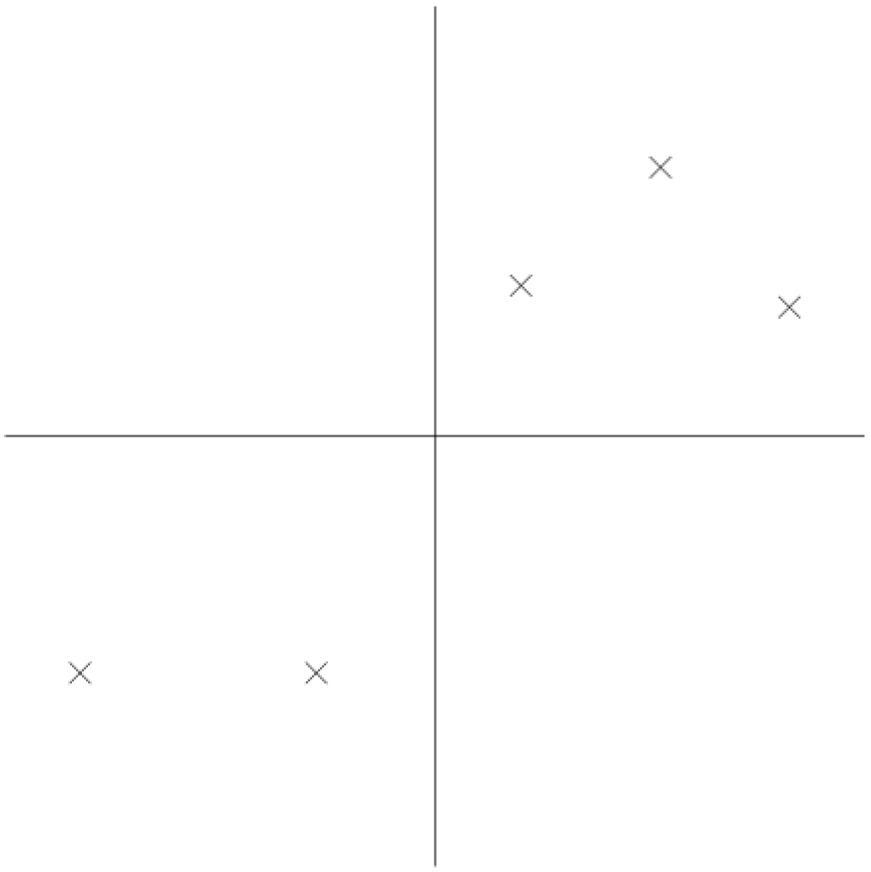
Before we run PCA per so, typically we first pre-process the data to normalize its mean and variance:

1. Let $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$
2. Replace each $x^{(i)}$ with $x^{(i)} - \mu$
3. Let $\sigma_j^2 = \frac{1}{m}\sum_i (x_j^{(i)})^2$
4. Repleace each $x_j^{(i)}$ with $x_j^{(i)}/\sigma_j$

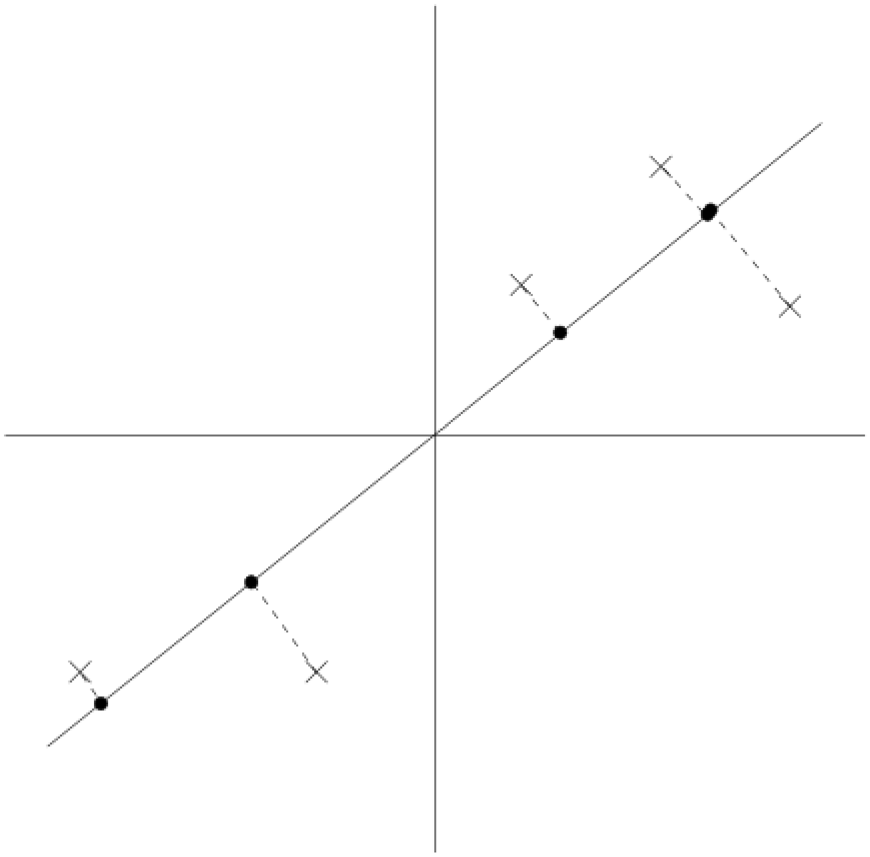
Steps (1-2) zero out the mean of the data and may be omitted for data with zero mean. Steps (3-4) rescale each coordinate to have unit variance, which ensures that different attributes are all treated on the same scale. If $x_1$was car's maximum speed in mph (taking values in high tens to low hundreds) and x_2 were the number of seats (ususally less than 10), then this renormalization rescales the different attributes to make them more comparable. If we have a priori knowledge that the attributes are all on the same scale we can leave out these steps.

Now how do we calculate the direction u in which the data approximately lies. A way to pose this problem is finding the unit vector u, so that when the data is projected onto the direction corresponding to u , the dariance of the projected data is maximized. Intuitively, the data starts off with some variance in it. We would like to choose a direction u, so that if we approximate the data as lying in this direction/subspace corresponding to u, as much as possible of the variance is still retained.

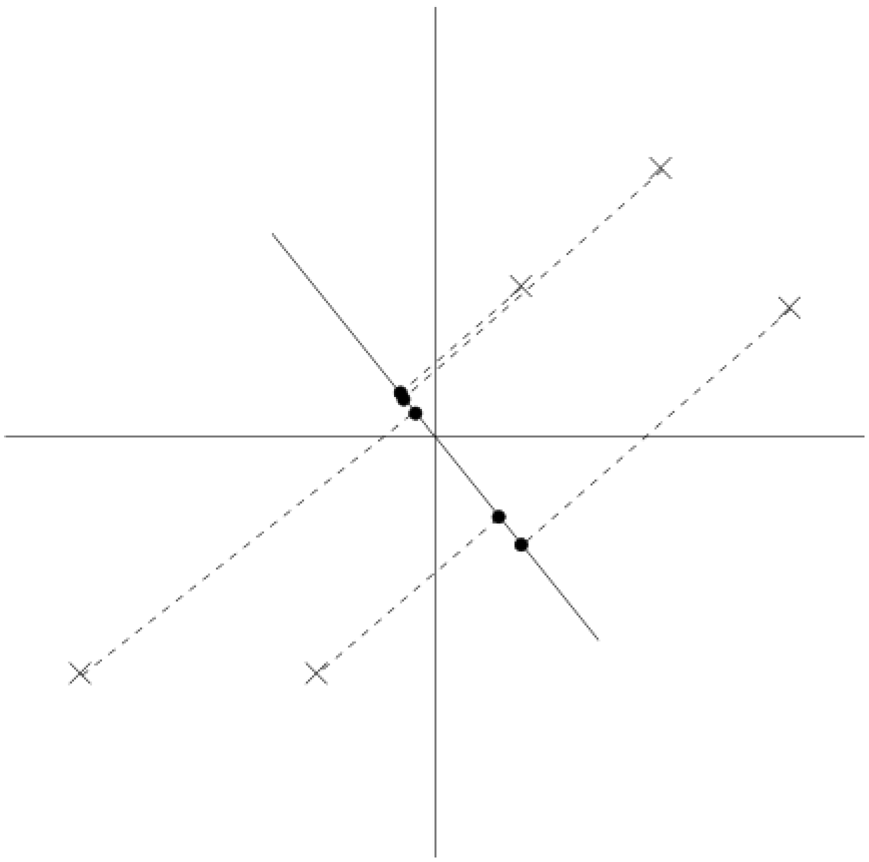
Consider the following, already normalized dataset:



Now supoose we pick u to correspond the direction shown in the figure below. The circles denote the projections of the original data onto this line:



We see that the projected data still has a fairly large variance and the points tend to be far from zero. In contrast, suppose we had instead picked the following direction:



Here, the projections have a smaller variance and are much closer to the origin. We would like to automatically choose the direction of the first figure. To formalize this, note that given a unit vector u and a point x, the length of the projection of x onto u is given by $x^T u$. I.e. if $x^{(i)}$is a point in our dataset, then its projection onto u has distance $x^T u$ from the origin. Hence, to maximize the variance of the projections, we would like to chhose a unit-length u so as to maximize

$$\frac{1}{m}\sum_{i=1}^{m}(x^{(i)^T}u)^2 = u^T\left(\frac{1}{m}\sum_{i=1}^{m}x^{(i)}x^{(i)T}\right)u$$

We easily recognize that maximizing this (with $||u||_2 = 1$) gives the principal ("largest) eigenvector of $\Sigma = \frac{1}{m}\sum_{i=1}^{m}x^{(i)}x^{(i)^T}$, which is just the empirical covariance matrix of the data (assuming it has zero mean).

To summarize, we have found that if we wish to find a 1-dimensional subspace with which to approximate the data, we should choose u to be the eigenvector of Σ. More generally, if we wish to project our data into a k-dimensional subspace $(k < n)$, we should choose $u_1, \ldots, u_k$ to be the top k eigenvectors of Σ. The u_i's form a new, orthogonal basis for the data. Then, to represent $x^{(i)}$ in this basis, we need only compute the corresponding vector

$$y^{(i)} = \begin{bmatrix} u_1^T x^{(i)} \\ \vdots \\ u_k^T x^{(i)} \end{bmatrix} \in \mathbb{R}^k$$

Thus, whereas $x^{(i)} \in \mathbb{R}^n$, the vector $y^{(i)} \in \mathbb{R}^k$ now gives a k-kimdensional approximation for $x^{(i)}$. PCA is therefore also referred to as dimensionality reduction algorithm. The vectors $u_1, \dots, u_k$ are called the first k principal components of the data.

12 Reinforcement Learning and Control

In supervised learning, our algorithms had some labels y that indicated whether our algorithm was doing well. In many sequential decision maxing and control problems, it is difficult to provide this kind of information. Imagine a four-legged robot trying to walk. Inititally we have no idea what the "correct" actions to take are. In the reinforcement learning framework, we will provide some reward function that indicates to our learning agent when it is doing poorly and when it is doing well. In the example with the robot, the reward function might give positive rewards for moving forwards and negative rewareds for moving backwards or falling over.

12.1 Markov decision processes MDP

A Markov decision process is a tuple $(S, A, \{P_A\}, \gamma, R)$, where:

- S is a set of **states** (For example in autonomous helicopter flight, S might be the set of all possible orientations and positions of the helicopter)
- A is a set of **actions** (For example the set of all possible directions in which you can push the helicopter's control stick)
- P_{sa} are the **state transition probabilities**. For each state $s \in S$ and action $a \in A$, P_{sa} is a distribution over the state space. Briefly, P_{sa} gives the distribution over what states we will transition to if we take action a in state s.
- $\gamma \in [0, 1)$ is called the **discount factor**.
- $R : S \times A \mapsto \mathbb{R}$ is the **reward function** (sometimes only written as $R : S \mapsto \mathbb{R}$).

We start in some state s_0 and get to choose some action a_0 to take in the MDP. Based on this choice, the state of the MDP randomly transitions to some successor state s_1, drawn according to $s_1 \sim P_{s_0 a_0}$. Then we get to pick another action a_0, MDP randomly transitions to some successor state $s_2 \sim P_{s_0 a_0}$.

$$\begin{tikzcd} s_0 \arrow[r, "a_0"] & s_1 \arrow[r, "a_1"] & s_2 \arrow[r, "a_2"] & s_3 \arrow[r, "a_3"] & \dots \end{tikzcd}$$

Our total payoff is then given by

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \dots$$

Or when we are writing the reward as a function of the states only

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$$

Our goal in reinforcement learning is to choose actions over time as to maximize the expected value of the total payoff:

$$E\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots\right]$$

Note that the reward at timestep t is **discounted** by a factor of γ^t. Thus, to make this expectation large, we would like to get positive feedback as soon as possible (and postpone negative rewards as long as possible).

A **policy** is any function $\pi : S \mapsto A$ mapping from the states to the actions. We say that we are **executing** some policy π if, whenever we are in state s, we take action $a = \pi(s)$. We also define the **value function** for a policy π according to

$$V^\pi(s) = E\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots | s_0 = s, \pi\right]$$

$V^\pi(s)$ is simply the expected sum of discounted rewards upon starting in state s and taking actions according to π.

Given a fixed policy π, it's value function V^π satisfies the **Bellman euqations:**

$$V^\pi(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^\pi(s')$$

This says that the expected sum of discounted rewards $V^\pi(s)$ for starting in s consists of two terms: First, the **immediate reward** $R(s)$ that we get rightaway for starting in state s and second, the expected sum of future rewards. We see that we can rewrite this second term as $E_{s' \sim P_{s\pi(s)}}\left[V^\pi(s')\right]$. This is the expected sum of discounted rewards for starting in state s', where s' is distributed according to $P_{s\pi(s)}$ which is the distribution over where we will end up taking the first action $\pi(s)$ in the MDP from state s.

Bellman's equation can be used to efficiently solve for V^π.In a finite-state MDP, we can write down one such equation for $V^\pi(s)$ for every state s. This gives us a set of $|S|$ linear equations in $|S|$ variables which can be efficiently solved for the $V^\pi(s)$'s.

We also define the **optimal value function** according to

$$V^*(s) = max_\pi V^\pi(s)$$

This is the best possible expected sum of discounted rewards that can be attained using any policy. There is also a version of Bellman's euqation for the optimal value:

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s')V^*(s')$$

The first part is the immediate reward as before. The second term is the maximum over all actions a of the expected future sum of discounted rewards we'll get after action a. In other words, you choose the action that will lead to the most profit from the second state on.

We also define a policy $\pi^* : S \mapsto A$ as follows:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{sa}(s')V^*(s')$$

So the best policy is the one, that chooses the action leading to the biggest profit.

Further, we have

$$V^*(s) = V^{\pi^*}(s) \geq V^{\pi}(s)$$

An interesting property of $\pi^*$is, that it's the optimal policy for all states s. This means that we can start in any initial state in our MDP and use the same policy.

12.2 Value iterations and policy iteration

This section is a description of two efficient algorithms for solving finite-state MDPs. We will consider only MDPs with finite state and action spaces.

The first algorithm, **value iteration:**

1. For each state s, initialize $V(s) := 0$

2. Repeat until convergence {
   For every state, update $V(s) := R(s) + \max_{a \in A} \sum_{s' \in S} P_{sa}(s')V^*(s')$
   }

This algorithm can be thought of as repeatedly tryin to update the estimated value function using Bellman Equations.

There are two ways of performing the updates of the inner loop. In the first, we first compute the new values of $V(s)$ for every state s and then overwrite all the old values with the new values. This is called a **synchronous** update. Alternatively, we can also perfprm **asychronous** updates. Here, we would loop over the states, updating the values one at a time.

Under either synchronous or asynchronous updates, it can be shown that value iteration will cause V to converge to V^*. Having found V^*, we can then use the equation in the previous section to find the optimal policy.

The second way is the **policy iteration** algorithm which proceeds as follows:

```
1. Initialize $\pi$ randomly.
2. Repeat until convergence {
   (a) Let $V := V^{\pi}$
   (b) For each state $s$, let $\pi(s) := arg\max_{a\in A} \sum_{s'} P_{sa}(s')V(s')$
```

The inner loop repeatedly computes the value function for the current policy and then updates the policy using the current value function. Step (a) can be done via solving Bellman's equations as described earlier.

After at most a finite number of iterations of this algorithm, V will converge to $V^*$and π will converge to π^*.

Both algorithms are standard for solving MDPs and there isn't universal agreement over which is better. For small MDPs, policy iteration is often very fast and converges with very few iterations. However, for MDPs with large state spaces, solving for V^πexplicitly would involve solving a large system of linear quations and could be difficult. In these problems, value iteration may be preffered. For this reason, in practice value iteration seems to be used more often than policy iteration.

12.3 Learning a model for an MDP

So far in our discussion we assumed that we know the state transition probabilites and rewards. Often, we don't have this values given and must estimate them from the data (Usually, S, A and γ are known).

For example, assume that for a problem we have a number of trials in the MDP as follows:

$$s_0^{(1)} \overset{a_0^{(1)}}{\to} s_1^{(1)} \overset{a_1^{(1)}}{\to} \dots$$

$$s_0^{(2)} \overset{a_0^{(2)}}{\to} s_1^{(2)} \overset{a_1^{(2)}}{\to} \dots$$

Here, $s_i^{(j)}$ is the state we were at time i of trial j and $a_i^{(j)}$ is the corresponding action that was taken from that state. In Practice, this runs untill we cancel it or it terminates.

Given this experiecce in the MDP consisting of a number of trials, we can then easily derive the maximum likelihood estimates for the state transition probabilities:

$$P_{sa}(s') = \frac{\#times\ we\ took\ action\ a\ in\ state\ s\ and\ got\ to\ s'}{\#times\ we\ took\ action\ a\ in\ state\ s}$$

Or if the ratio above is "0/0" - we never have taken action a in state s before- we can simply estimate it to be 1/|S|.

If we get new experience in the sense of more trials, we can simply keep the ratio above and add the new trials.

Using a similar procedure, if R is unknown, we can also pick our estimate of the expected immediate reward $R(s)$ in state s to be the average reward observed in state s.

Having learned a model for the MDP, we can then use either value iteration or policy iteration to solve the MDP using the estimated transition probabilites and rewards. For example, this is a possible algorithm for learning in an MDP with unkown state transition probabilities:

1. Initialize π randomly.

2. Repeat {
   (a) Execute π in the MDP for some number of trials.
   (b) Using the accumulated eperience in the MDP, update our estimates for P_{sa} (and R if applicable).
   (c) Apply value iteration with the estimated state transition probabilities ad rewards to get a new estimated value function V.
   (d) Update π to be the greedy policy with respect to $|V|$.
   }

In this particular algorithm, we can in the inner loop instead of initializing value iteration with $V = 0$ continue with the value from the last step and make it converge faster.

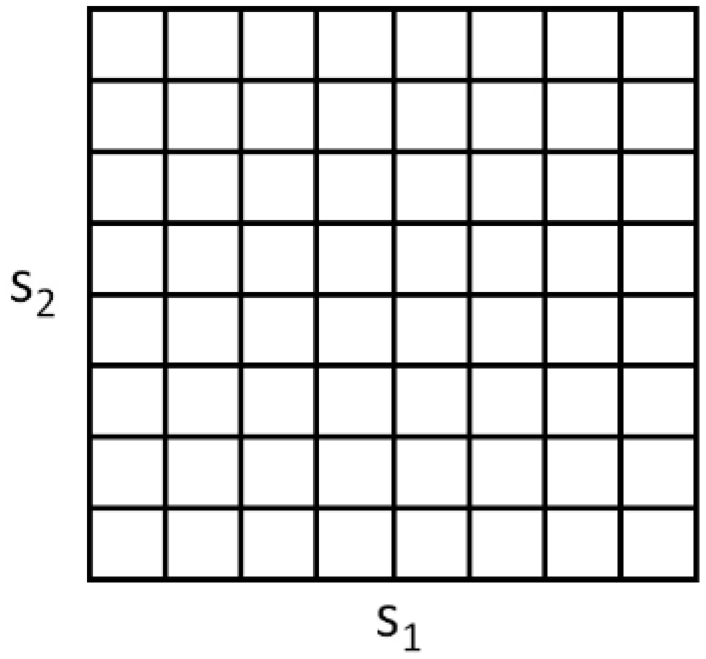
12.4 Continuous state MDPs

We will now discuss MDPs which may have an infinite number of states. For example, for a car, we might represent the state as $(x, y, \theta, \dot{x}, \dot{y}$, comprising its position (x, y); orientation ϑ; velocity in the x and y directions $\dot{\text{x}}$ and $\dot{\text{y}}$; and angular velocity $\dot{\vartheta}$. Hence, $S = \mathbb{R}^6$ is an infinite set of states, 7 $\dot{\vartheta}$), because there is an infinite number of possible positions and orientations for the car.

12.4.1 Discretization

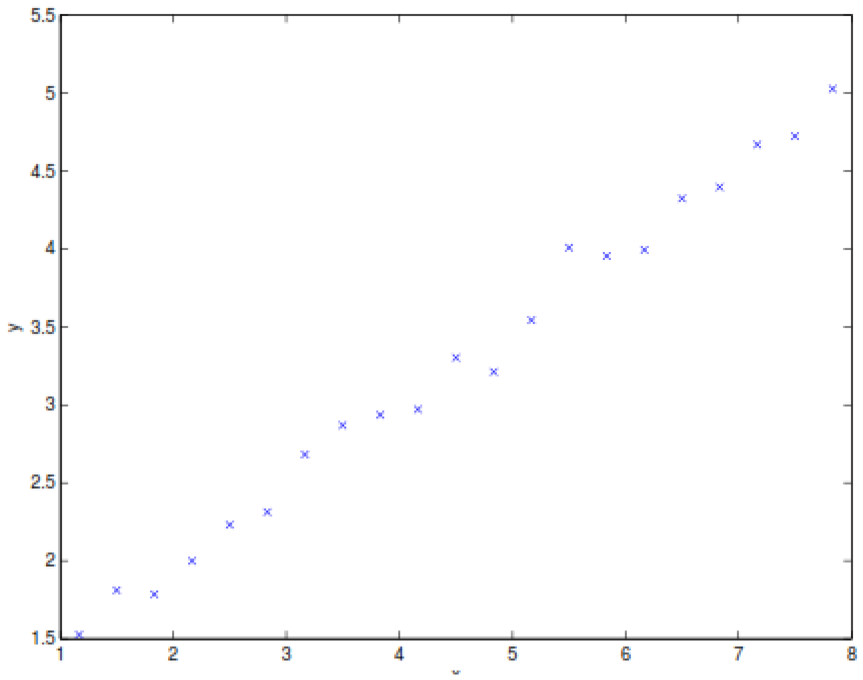
The perhaps simplest way is to discretize the state space and then to use an algorithm like value iteration or policy iteration as described previously.

For example, if we have 2d states (s_1, s_2), we can use a grid to discretize the state space:

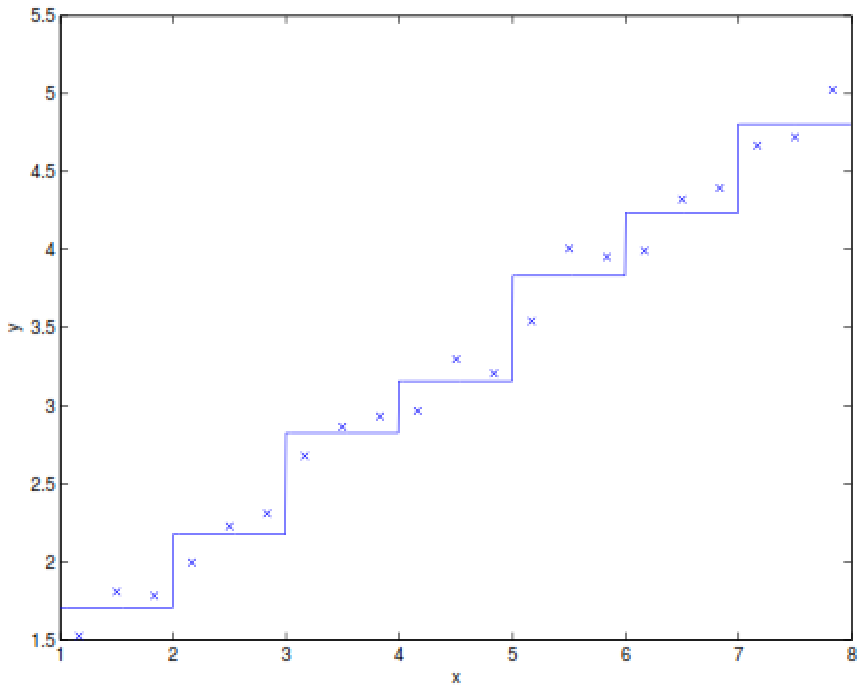


Here, each grid cell represents a seperate discrete state $\overline{s}$. We can then approximate the continuous-state MDP via a discrete-state one$(\overline{S}, A, \{P_{\overline{s}a}\}, \gamma, R)$, where $\overline{S}$ is the set of discrete states, $\{P_{\overline{s}a}\}$ are our state transition probabilities over the discrete states and so on. We can then use value or policy iteration as before. When our actual system is in some coninuous-valued state $s \in S$ and we need to pick an action to exectue, we compute the corresponding discretized state $\overline{s}$ and execute action $\pi^*(\overline{s})$.

This can work well for many problems. But there are two downsides. First, it uses a farily naive representation for V^* (and π^*). It assumes that the value fucntion takes on a constant value over each of the discretization intervals. To better understand this, consider a supervised learning problem of fitting a function to this dataset:



Linear regression would obviously do fine on this problem. However, if we instead discretize the x-axis and then use a representation that is piecewise constant in the intervals, our fit to the data would look like this:



It results in little smoothing over the inputs and no generalization over the different grid cells. We would need a very fine discretization (very small grid cells) to get a good approximation.

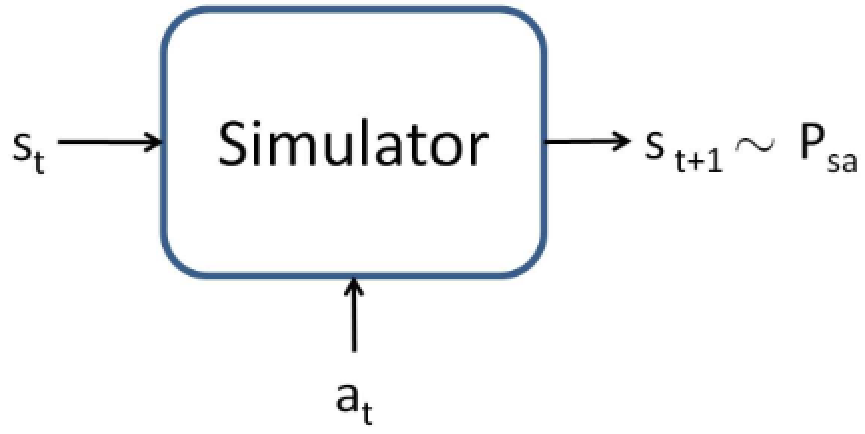
A second downside of this representation is called the **curse of dimensionality**. Suppose $S = \mathbb{R}^n$ and we discretize each of the n states into k values. Then the total number of sicrete states we have is k^n. Thus, it does not scale well to large problems.

As a rule of thumb, discretization usueally works extremely well for 1d and 2d problems. With some tricks it can still be efficient for up to 4d, but more is usually not working.

12.4.2 Value function approximation

This is an alternative method for finding policies in continuous-state MDPs, in which we approximate V^*directly, without resorting to discretization.

Using a model or simulator To develop such an algorithm, we will assume that we have a **model** or **simulator** for the MDP. Informally, a simulator is a black-box that takes as input any state s_t and action a_t and outputs a next state s_{t+1}, sampled according to the state transition probabilities $P_{s_t a_t}$:



A method to get such a model is to use off-shelf software. An alternative way to get a model is to learn one from data collected in the MDP.

Suppose we execute m **trials** in which we repeatedly take actions in an MDP, each trial for T timesteps. We would then observe m state sequences like the following:

$$s_0^{(1)} \overset{a_0^{(1)}}{\rightarrow} s_1^{(1)} \overset{a_1^{(1)}}{\rightarrow} \dots \overset{a_{T-1}^{(1)}}{\rightarrow} s_T^{(1)}$$

$$s_0^{(2)} \overset{a_0^{(2)}}{\rightarrow} s_1^{(2)} \overset{a_1^{(2)}}{\rightarrow} \dots \overset{a_{T-1}^{(2)}}{\rightarrow} s_T^{(2)}$$

$$\vdots$$

$$s_0^{(m)} \overset{a_0^{(m)}}{\rightarrow} s_1^{(m)} \overset{a_1^{(m)}}{\rightarrow} \dots \overset{a_{T-1}^{(m)}}{\rightarrow} s_T^{(m)}$$

We can then apply a learning algorithm to predict s_{t+1}as a function of s_t and a_t.

For example, one may choose to learn a linear model of the form

$$s_{t+1} = As_t + Ba_t$$

using an algorithm similar to linear regression. Here, the parameters of the model are the matrices A and B and we can estimate them using the data collected from our m trials by picking

$$arg\min_{A,B} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - \left(As_t^{(i)} + Ba_t^{(i)} \right) \right\|^2$$

Having learned A and B, one option is to build a **deterministic** model in which given an input $s_t$and a_t, the output s_{t+1} is exactly determined.

Alternatively, we may also build a **stochastic** model, in which s_{t+1} is a random function of the inputs by modeling it as

$$s_{t+1} = As_t + Ba_t + \epsilon_t,$$

where ϵ_t is a noise term, usually modeled as $\epsilon_t \sim N(0, \Sigma)$.

Of course, also non-linear functions are possible.

Fitted value iteration We now describe the **fitted value iteration** algorith for approximating the value function of a continuous state MDP. We will assume that the problem has a continuous state space $S = \mathbb{R}^n$ but that the action space A is small and discrete.

Recall that in value iteration we would like to perform the update

$$V(s) := R(s) + \gamma \underset{a}{max} E_{s' \sim P_{sa}}[V(s')]$$

The idea of fitted value iteration is that we are going to approximatley carry out this step over a finite sample of states $s^{(1)}, \ldots, s^{(m)}$. We will use linear regression (a supervised learning algorithm) to approximate the value function as a linear or non-linear function of the states:

$$V(s) = \Theta^T \phi(s).$$

Here, ϕ is some appropriate feature mapping of the states.

For each state s in our finite sample of m states, fitted value iteration will first compute a quantity $y^{(i)}$, which will be our approximation to $R(s) + \gamma \underset{a}{max} E_{s' \sim P_{sa}}[V(s')]$. Then, it will aplly a supervised learning algorithm to try to get $V(s)$ close to $R(s) + \gamma \underset{a}{max} E_{s' \sim P_{sa}}[V(s')]$ (or in other words to try to get $V(s)$ close to $y^{(i)}$).

In detail:

1. Randomly sample m states $s^{(1)}, \ldots, s^{(m)} \in S$
2. Initialize $\Theta := 0$
3. Repeat {

```
For $i = 1, \ldots, m\{$
For each action $a \in A\{$
Sample $s'_1, \ldots s'_k \sim P_{s^{(i)}a}$ (using a model of the MDP).
Set $q(a) = \frac{1}{k} \sum_{j=1}^{k} R(s^{(i)}) + \gamma V(s'_j)$ Hence, $q(a)$ is an estimate of $R(s^{(i)}) + \gamma \underset{a}{max} E_{s' \sim P_{s^{(i)}a}}[V(s')]$
}
Set $y^{(i)} = max_a q(a)$
}
Set $\Theta := arg\, min_\Theta \frac{1}{2} \sum_{i=1}^{m} (\Theta^T \phi(s^{(i)}) - y^{(i)})^2$
}
```

Above, we had written out fitted value using linear regression. That step is completely analogous to a standard supervised learning problem, in which we have a training set and want to learn function mapping from x to y. The only difference is taht here s plays the role of x.

Unlike value iteration over a discrete set of states, fitted value iteration cannot be proven to converge. However, in practice it often does.

Finally, fitted value iteration outputs V, which is an approximation to V^*. This implicitly defines our policy. When our system is in some state s and we need to choose an action, we ould like to choose the action

$$arg\, max_a E_{s' \sim P_{sa}}[V(s')]$$

The process for approximating this is similar to the inner loop of fitted value iteration, where for each iteration, we sample $s'_1, \ldots, s'_k \sim P_{sa}$ to approximate the expectation.

In practice there're often other ways to approximate this step as well. Another common case is if the simulator is of the form $s_{t+1} = f(s_t, a_t) + \epsilon_t$ where f is some deterministic function of the states and ϵ is zero-mean Gaussian noise. In this case, we can pick the action given by

$$arg\, max_a V(f(s,a))$$

In other words, here we are just setting $\epsilon_t = 0$ and $k = 1$. This can be dervied from the quation above using the approximation

$$E_{s'}[V(s')] \simeq V(E_{s'}[s']) = V(f(s,a))$$

However, for problems that don't lend themselves to such approximations, having to sample $k|A|$ states using the model, in order to approximate the expectation above can be computationally expensive.