Freie Universität Berlin

Data Science Lecture 13 😢

18.07.2014

Dr. Kashif Rasul
kashif @krasul #167

Shoaib Burq
sabman @sabman #167

Welcome to Data Science Lecture 13 the last lecture 😢

Last Time

- Neural Networks
- AWS Setup and codes

Last time we did neural networks and in particular Convolutional Neural Networks and gave you an intuition of why they could work for learning non-linear features as well as some techniques for regularisation as well as making the computations faster.

You should all have your AWS credit codes now too.

Today

- Spark for Data Analysis
- Probabilistic Graphical Models

Today we will look at Spark for data analysis and Probabilistic Graphical Models: you can get more details from this pdf: http://research.microsoft.com/en-us/um/people/cmbishop/prml/pdf/Bishop-PRML-sample.pdf

Probabilistic Graphical Models is again a huge topic and we can easily do a whole course on this. So we will touch on it briefly.

Data Analysis with Spark

- Spark: cluster computing platform which is fast and general-purpose
- Extends MapReduce to more types of computations
- Comes as a unified stack for Stream computing, graph processing and Machine Learning components

Apache Spark is a cluster computing platform designed to be fast and general-purpose. On the speed side, Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing. Speed is important in processing large datasets as it means the difference between exploring data interactively and waiting minutes between queries, or waiting hours to run your program versus minutes. One of the main features Spark offers for speed is the ability to run computations in memory, but the system is also faster than MapReduce for complex applications running on disk.

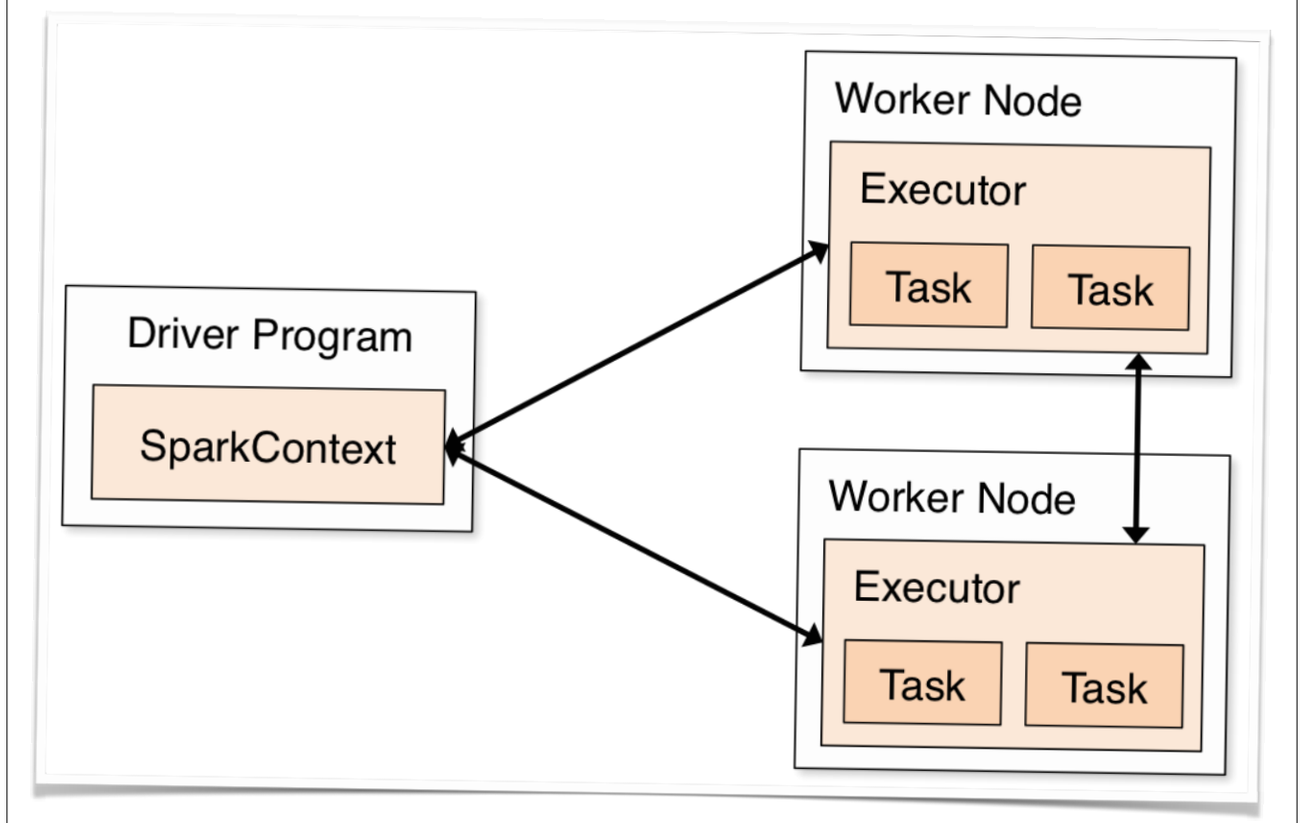
Used for what?

- Analyse big data: queries are interactive
- Sparks' speed and APIs are easy to use with little code
- Parallelise tasks across many nodes and hide the complexity of the network communication and distributed systems

```
➜ ./bin/pyspark
...

>>> lines = sc.textFile("README.md")

>>> lines.count()
127

>>> lines.first()
u'# Apache Spark'
```


In Spark we express our computation through operations on distributed collections that are automatically parallelised across the cluster. These collections are called a Resilient Distributed Datasets, or RDDs. RDDs are Spark's fundamental abstraction for distributed data and computation.

Before we say more about RDDs, let's create one in the spark shell from a local text file and do some very simple ad-hoc analysis by following the example below. Note the operations on the RDD happen in parallel.



At a high level, every Spark application consists of a driver program that launches various parallel operations on a cluster. The driver program contains your application's main function and defines distributed datasets on the cluster, then applies operations to them. Above we used the spark shell as the driver program.

Driver programs access spark through a SparkContext object which is a connection to a computing cluster. Once you have a SparkContext, you can use it to build resilient distributed datasets, or RDDs. In the example above, we called SparkContext.textFile to create an RDD representing the lines of text in a file. We can then run various operations on these lines, such as count().

To run these operations, driver programs typically manage a number of nodes called executors. For example, if we were running the count() above on a cluster, different machines might count lines in different ranges of the file.

```
>>> lines = sc.textFile("README.md")

>>> pythonLines = lines.filter(lambda line: "Python" in line)

>>>  pythonLines.first()
u'## Interactive Python Shell’

➜ ./bin/park-submit --master local[4] SimpleApp.py
...
```


Finally, a lot of Spark’s API revolves around passing functions to its operators to run them on the cluster. For example, we could extend our README example by filtering the lines in the file that contain a word, such as “Python”.

The final piece is how to use Spark in standalone programs. In python we simply write our application as a python script and run it using a special bin/spark-submit script included in Spark. This scripts sets up all that is needed for the python script to run.

```python
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("My App")
sc = SparkContext(conf)

...

sc.stop()
```


To create a program then we need to first creating a SparkConf object to configure your application, and then building a SparkContext for it in our main python script.

A cluster URL, namely “local” in these examples, tells Spark how to connect to a cluster. “local” is a special value that runs Spark on one thread on the local machine, without connecting to a cluster.

An application name, namely “My App” in these examples. This will identify your application on the cluster manager’s UI if you connect to a cluster.

We shut down Spark by calling stop() on the SparkContext object.

RDD Basics

- Create some input RDDs from external data
- Transform them to define new RDDs using transformations like `filter()`
- Ask Spark to `persist()` any intermediate RDDs that will need to be reused
- Launch actions such as `count()` and `first()` to kick off a parallel computation, which is then optimised and executed by Spark

An RDD in Spark is simply a distributed collection of objects. Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster. In Spark all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result.

And now let's go through the Spark work flow in a bit more detail.

Creating RDDs

- Loading an external dataset
- Take an existing in-memory collection and pass it to SparkContext's `parallelize` method:

```
lines = sc.parallelize(["pandas", "i like pandas"])
```


This second approach is very useful when learning Spark, since you can quickly create your own RDDs in the shell and perform operations on them. However, outside of prototyping and testing, this is not widely used since it requires you have your entire dataset in memory on one machine.

RDD Ops

- Transformations: operations on RDDs that return a new RDD
- Actions: operations that return a final value to the driver program

Transformed RDDs are computed lazily, only when you use them in an action. Many transformations are element-wise, that is they work on one element at a time, but this is not true for all transformations.

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)

errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```


Suppose that we have a log file, log.txt, with a number of messages, and we want to select only the error messages. We can use the filter transformation.

Note that the filter operation does not mutate the existing inputRDD. Instead, it returns a pointer to an entirely new RDD. inputRDD can still be re-used later in the program, for instance, to search for other words.

Finally we use another transformation called union(). Union operates on two RDDs.

Spark keeps track of the set of dependencies between different RDDs, called the lineage graph and uses this information to compute RDDs on demand and recover data if a persistent RDD is lost.

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"

for line in badLinesRDD.take(10):
  print line
```


Actions force the evaluation of the transformations required for the RDD they are called on, since they are required to actually produce output.

We might want to print out some information about the badLinesRDD. To do that, we'll use two actions, count() and take().

RDDs also have a collect() function to get the entire RDD which might be useful if the filter is tiny, but in general you would not use collect() for big data.

In real-world applications we would write data out to a distributed storage systems such as HDFS or Amazon S3.

Finally, lazy evaluation can be counter intuitive but if you know functional languages then it might be more familiar. An operation is not performed immediately, instead Spark internally records meta-data to indicate this operation has been requested.

We can think of RDDs are consisting of instructions on how to compute the data that we build up through transformations. The main advantage of this lazy loading is to reduce the number of passes it has to take over our data by grouping operations together, a big advantage over pure Hadoop MapReduce.

```python
word = rdd.filter(lambda s: "error" in s)

def containsError(s):
  return "error" in s
word = rdd.filter(containsError)

class WordFunctions(object):
  ...
  def getMatchesNoReference(self, rdd):
    # Safe: extract only the field we need into
    # a local variable query = self.query
    return rdd.filter(lambda x: query in x)
```


We can pass functions to transforms too. In python we have 3 options. For shorter function we can pass in lambda expressions. We can also pass in top-level functions or locally defined functions.

One issue to watch out for when passing functions if that if you pass functions that are members of an object, or references to fields in an object this results in sending in the entire object, which can be much larger than just the bit of information you need. Sometimes this can also cause your program to fail, if your class contains objects that Python can’t figure out how to pickle.

Instead, just extract the fields you need from your object into local variable and pass that in. Also it forces users to write smaller manageable operations rather than a single complex MapReduce application.

```python
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
  print "%i " % (num)


lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first() # returns "hello"
```


The two most common transformations you will likely be performing on basic RDDs are map, and filter. Map takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD. The filter transformation take in a function and returns an RDD which only has elements that pass the filter function.

Sometimes we want to produce multiple output elements for each input element. The operation to do this is called flatMap. Like with map, the function we provide to flat Map is called individually for each element in our input RDD. Instead of returning a single element, we return an RDD which consists of the elements from all of the iterators.

RDDs support many of the operations of mathematical sets, such as union and inter– section, even when the RDDs themselves not properly sets.

```python
sum = rdd.reduce(lambda x, y: x + y)

sumCount = nums.aggregate((0, 0),
            (lambda x, y: (x[0] + y, x[1] + 1),
            (lambda x, y: (x[0] + y[0], x[1] + y[1]))))
return sumCount[0] / float(sumCount[1])
```


The most common action on basic RDDs you will likely use is reduce. Reduce takes in a function which operates on two elements of the same type of your RDD and returns a new element of the same type. A simple example of such a function is + , which we can use to sum our RDD.

Similar to reduce is fold which also takes a function with the same signature as needed for reduce, but also takes a “zero value” to be used for the initial call on each partition. The zero value you provide should be the identity element for your operation.

The aggregate function frees us from the constraint of having the return the same type as the RDD which we are working on. Above we can use aggregate to compute the average of a RDD avoiding a map before the fold.

Level	Space Used	CPU time	In memory	On Disk	Nodes with data	Comments
MEMORY_ONLY	High	Low	Y	N	1	
MEMORY_ONLY_2	High	Low	Y	N	2	
MEMORY_ONLY_SER	Low	High	Y	N	1	
MEMORY_ONLY_SER_2	Low	High	Y	N	2	
MEMORY_AND_DISK	High	Medium	Some	Some	1	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_2	High	Medium	Some	Some	2	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	1	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER_2	Low	High	Some	Some	2	Spills to disk if there is too much data to fit in memory.
DISK_ONLY	Low	High	N	Y	1	
DISK_ONLY_2	Low	High	N	Y	2	

As discussed earlier, Spark RDDs are lazily evaluated, and sometimes we may wish to use the same RDD multiple times. If we do this naively, Spark will recompute the RDD and all of its dependencies each time we call an action on the RDD. This can be especially expensive for iterative algorithms, which look at the data many times.

We an ask Spark to persist the data. When we ask Spark to persist an RDD, the nodes that compute the RDD store their partitions. If a node that has data persisted on it fails, Spark will recompute the lost partitions of the data when needed. We can also replicate our data on multiple nodes if we want to be able to handle node failure without slowdown.

This shows the levels of Spark persistance.

```python
input.map(lambda x: (x.split(" ")[0], x))

result = pair.filter(lambda x: len(x[1]) < 20)

rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] +
y[0], x[1] + y[1]))
```


Spark provides special operations on RDDs containing key-value pairs called Pair RDDs. These are useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network. There are a number of ways to get Pair RDDs in Spark.

For the functions on keyed data to work we need to make sure our RDD consists of tuples. Pair RDDs are allowed to use all the transformations available to standard RDDs. Since Pair RDDs contain tuples, we need to pass functions that operate on tuples rather than on individual elements.

Sometimes working with these pairs can be awkward if we only want to access the value part of our Pair RDD.

When datasets are described in terms of key-value pairs, it is common to want to aggregate statistics across all elements with the same key. reduceByKey is quite similar to reduce, both take a function and use it to combine values. This runs several parallel reduce operations, one for each key in the dataset, where each operation combines values together which have the same key.

There is also foldByKey and reduceByKey along with mapValues.

```
rdd = sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y:
                                                 x + y)
```

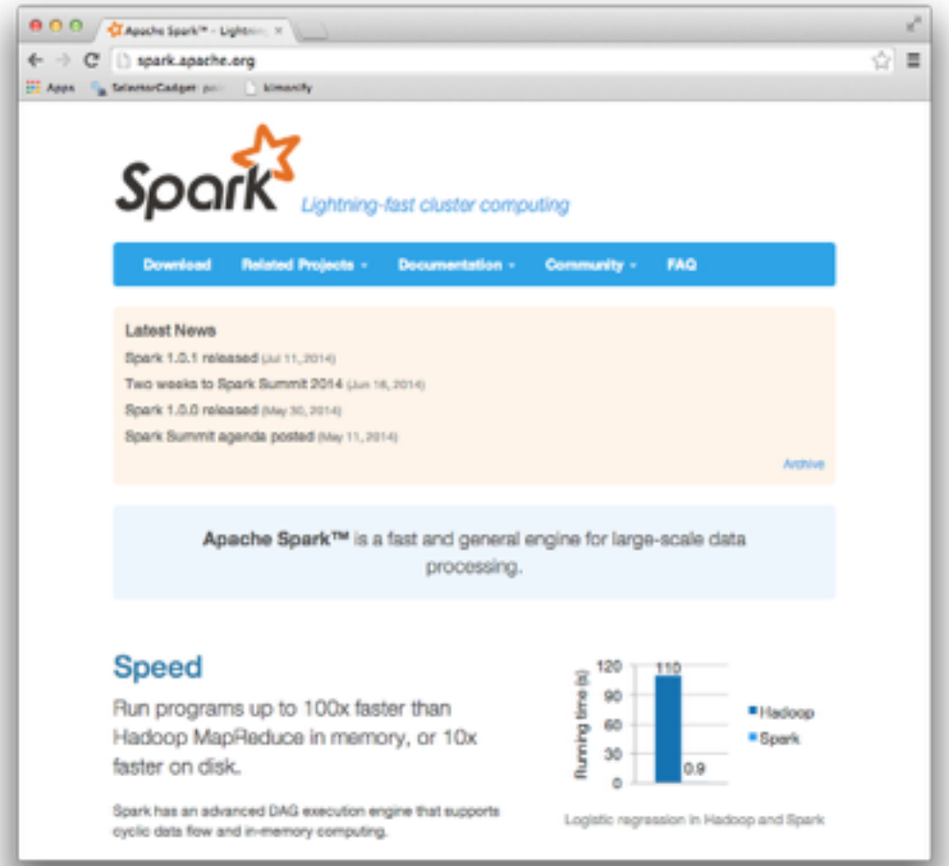

We can use a similar approach to also implement the classic distributed word count problem. We will use flatMap so that we can produce a Pair RDD of words and the number 1 and then sum together all of the words using reduce ByKey like in our previous example.

We can also do groupByKey and Joins as well on Pair RDDs.

```
import csv
import StringIO

...

def loadRecord(line):
  """Parse a CSV line"""
  input = StringIO.StringIO(line)
  reader = csv.DictReader(input, fieldnames=[“name",
                                             "favouriteAnimal"])
  return reader.next()

input = sc.textFile(inputFile).map(loadRecord)
```

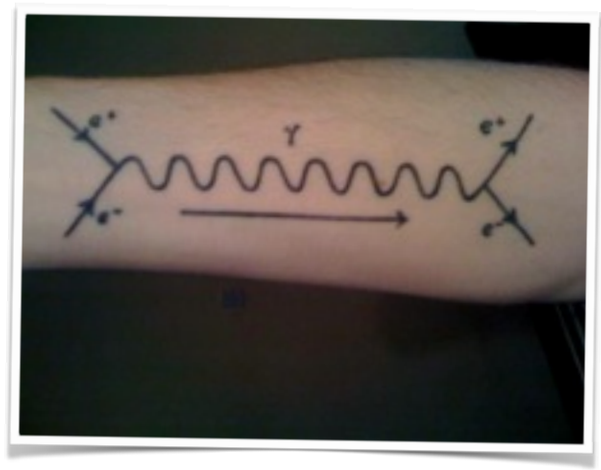

If your CSV data happens to not contain newlines in any of the fields, you can load your data with textFile and parse it.

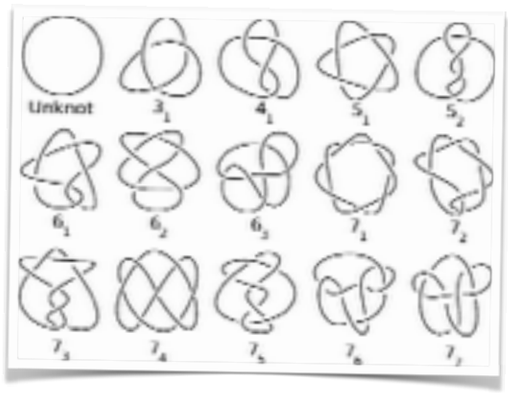
If there are embedded newlines in fields we will need to load each file in full and parse the entire segment. This is unfortunate as if each file is large this can easily introduce bottlenecks in loading and parsing. And we can similarly read and write form JSON and databases, and different files.

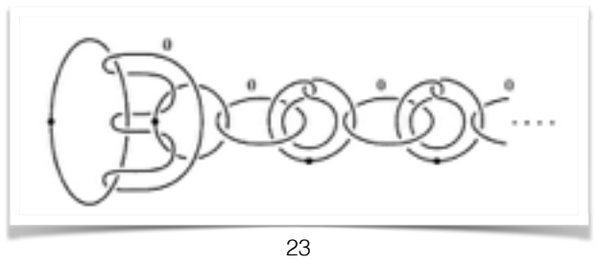


You can get download and get started by visiting the Spark website at http://spark.apache.org/ and we encourage you to tryout Spark on AWS as well and perhaps use it for HW3.

Why Graphical Models?





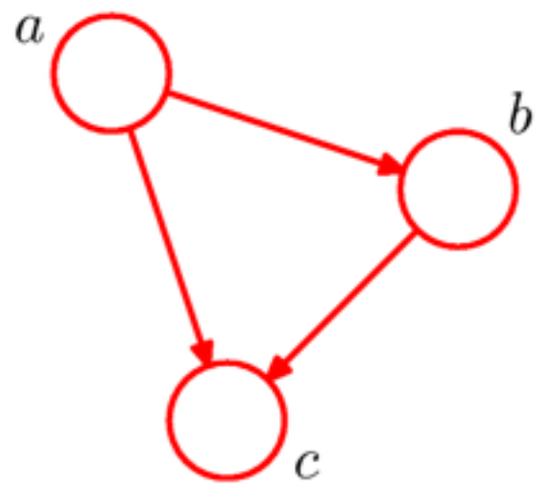


Humans find reasoning about and manipulating graphical representations or abstractions of complex ideas much easier than reasoning about and manipulating the complex ideas itself in their brain or on paper. Examples of these abstractions in physics are Feynman diagrams. In a branch of mathematics these knot diagrams help the experts to reason about these complex structures. Lastly another example is Kirby Diagrams which Mathematicians use to describe four dimensional space. Chemistry and other fields have their equivalent pictorial abstractions.

So when it comes to statistical learning, it seems only logical to look for a graphical representation of a probability distribution that allows one to easily manipulate and reason about these distributions. That is what Graphical Models provide.

Bayesian Networks

- p(a,b,c) = p(c|a,b) p(b|a) p(a)
- random variable -> node
- associate each node with the conditional probability for its variable
- then for each node and for each corresponding conditioning variable add a directed edge from conditioning to conditioned

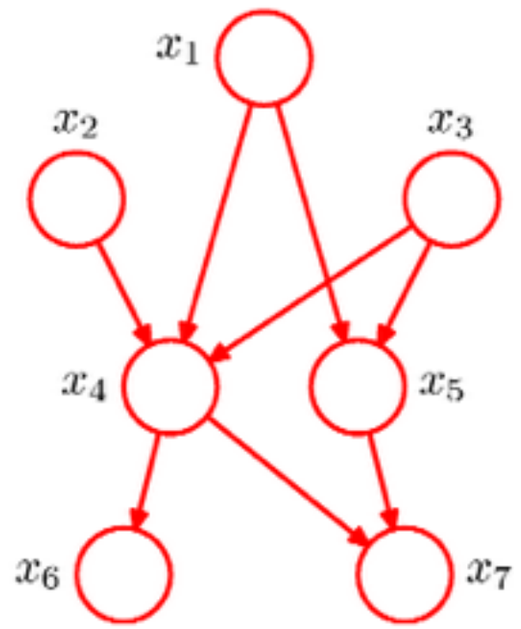


Consider a probability distribution p(a,b,c) over three random variables a, b, and c . Using the product rule of probability twice, we can rewrite p(a,b,c) as follows:

Now we can introduce a graphic form for the right hand side of this equation.

The distribution p(a) is not conditioned; thus, there are no directed edges going in to node a . The distribution p(b|a) is conditioned on a; thus, there is a directed edge from node a to node b. The distribution p(c|a,b) is conditioned on a and b; thus, there is a directed edge from node a to node c and a directed edge from node b to node c .

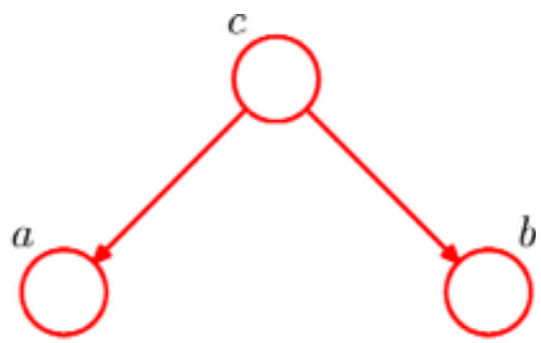
- $p(x_1)\ p(x_2)\ p(x_3)$
  $p(x_4|x_1,x_2,x_3)$
  $p(x_5|x_1,x_3)\ p(x_6|x_4)$
  $p(x_7|x_4,x_5)$



We can explore more complicated diagrams. For example the distribution described by this graph is given on the right. One can form an infinite number of such graphs: nodes representing random variables and directed edges conditioning. Generically such a graph goes under the name of a Bayesian network, and so let us study Bayesian networks in some detail.

Conditional Independence

- Independent: p(a,b)=p(a)p(b)
- conditionally independent: p(a,b|c) = p(a|c) p(b|c)



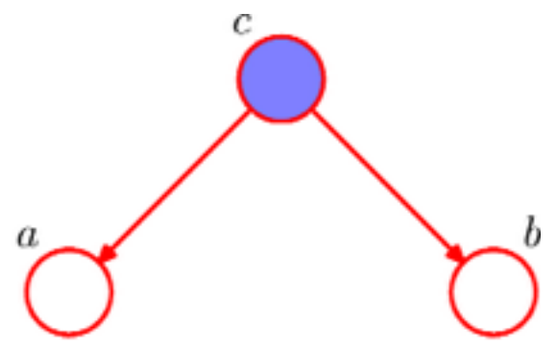
These graphical models help one reason about "independence" and "conditional independence". Recall the definitions of independence above.

This notion of independence can be generalised to the case of conditional distributions as well. Given the random variable c two random variables a and b are conditionally independent if the second equation holds. Often one is given a joint probability distribution and one needs to determine if two, or more, random variables in the joint distribution are independent or conditionally independent. Given a graphical model for the distribution, this question can be answered relatively easily, as we will see.

First consider the probability distribution depicted by this graphical model. It corresponds to the equation p(a,b,c) = p(a|c) p(b|c) p(c). Let us assume that c is not observed and, in the context of this assumption, examine whether a and b are independent. Using the sum rule of probability we can write the distribution p(a,b) as $\sum_c$p(a,b,c). Substituting the form of the distribution p(a,b,c) gleaned from the graphical model, we have p(a,b) = $\sum_c$ p(a|c) p(b|c) p(c). So generically this distribution does not factor into a product of p(a) and p(b) and is not independent in general.

c is observed

- An observed node is shaded
- p(c)≠0 since it was observed
- p(a,b|c)=p(a|c) p(b|c) => a and b are conditionally independent
- Tail-to-Tail rule



Consider now the case in which c is observed. If c is observed, then a and b are conditionally independent: p(a,b|c)= p(a|c) p(b|c).

This gives rise to the tail-to-tail rule of conditional independence: if an observed node is connected to the tails of two directed edges then the two nodes at directed edges' heads are independent.

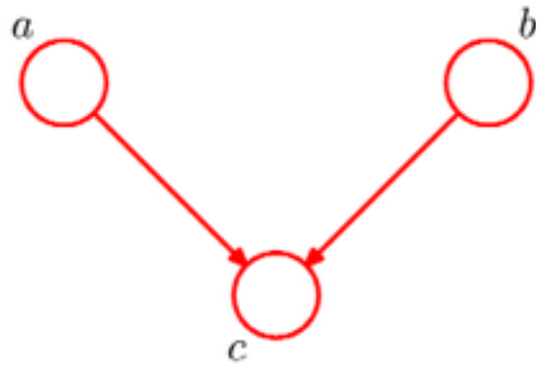
Head-To-Tail

- p(a,b,c) = p(a) p(c|a) p(b|c)
- c is not observed: are a and b are independent?
- c is observed: are a and b independent?

This is the head-to-tail rule of conditional independence. If an observed node is connected to the head of one directed edge and the tail of another then the two nodes at the edges' other end are independent.

Head-To-Head

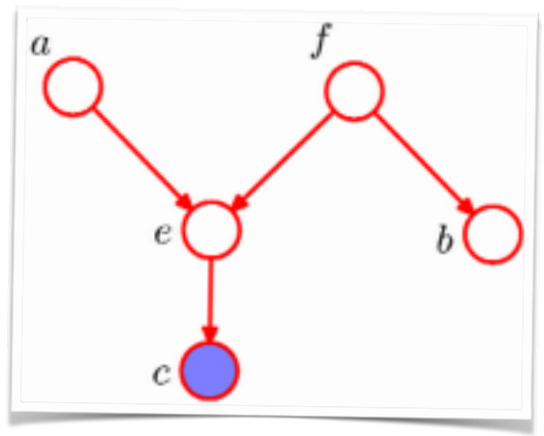
- p(a,b,c) = p(a) p(b) p(c|a,b)
- distribution p(a,b) = p(a) p(b) => a and b independent
- c observed => a and b not independent

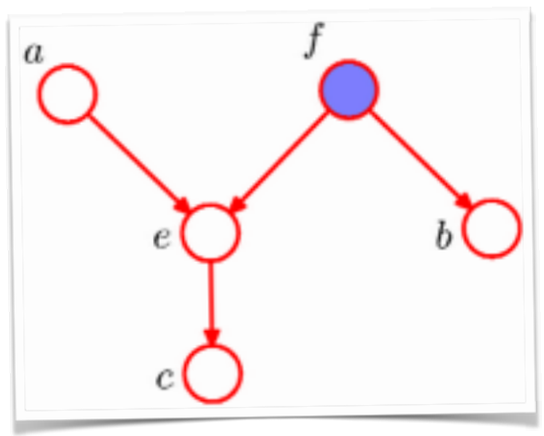


This is exactly opposite of the tail-to-tail and head-to-tail cases where, if c was not observed, a and b were not independent. Now when c is not observed, a and b and independent.

When c is observed then a and b are not independent. This is the Head-To-Head rule of conditional independence: if an observed node is connected to the head of two directed edges then the two nodes at the edges' other ends are dependent. More over if the node is connected to the head of two two directed edges, c, above and even if any of c’s children are observed then a and b are dependent.

Generally if two, or more, nodes are conditionally independent according to these rules, the nodes are said to be d-seperated. And there is a d-seperation theorem.



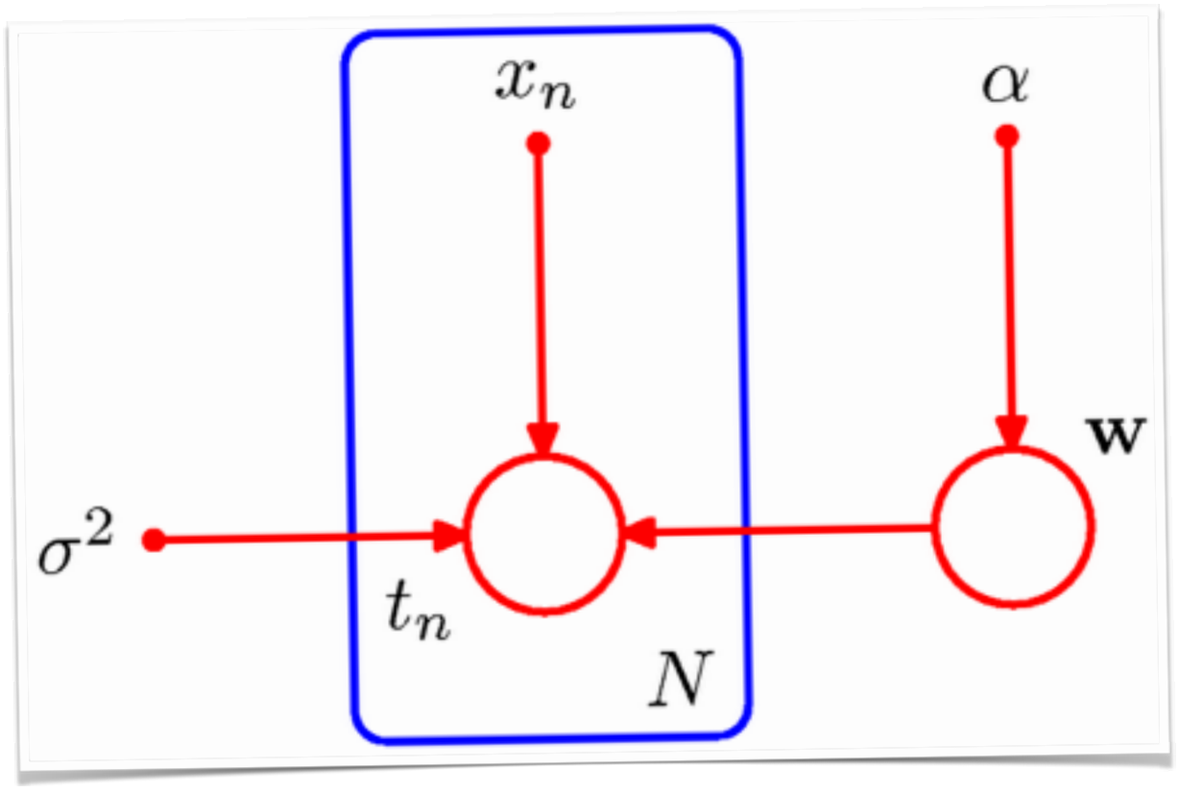


So this principle now lets us ask whether a particular conditional independence statement, A⫫B | C holds for a given directed acyclic graph. To do so consider all possible paths from any node in A to any node in B. Any such path is said to be blocked if it includes a node such that either
1. the arrows on the path meet either head-to-tail or tail-to-tail at the node, and the node is in the set C
2. the arrows meet head-to-head at the node, and neither the node, nor any of its descendants, is in the set C.

If all paths are blocked, then A is said to be d-separated from B by C, and the joint distribution over all of the variables in the graph will satisfy A ⫫ B | C.

In the above left image the path from a to b is not blocked by node f because it is a tail-to-tail node for this path and is not observed, nor is it blocked by node e. So the conditional independence a ⫫ b | c does not follow.

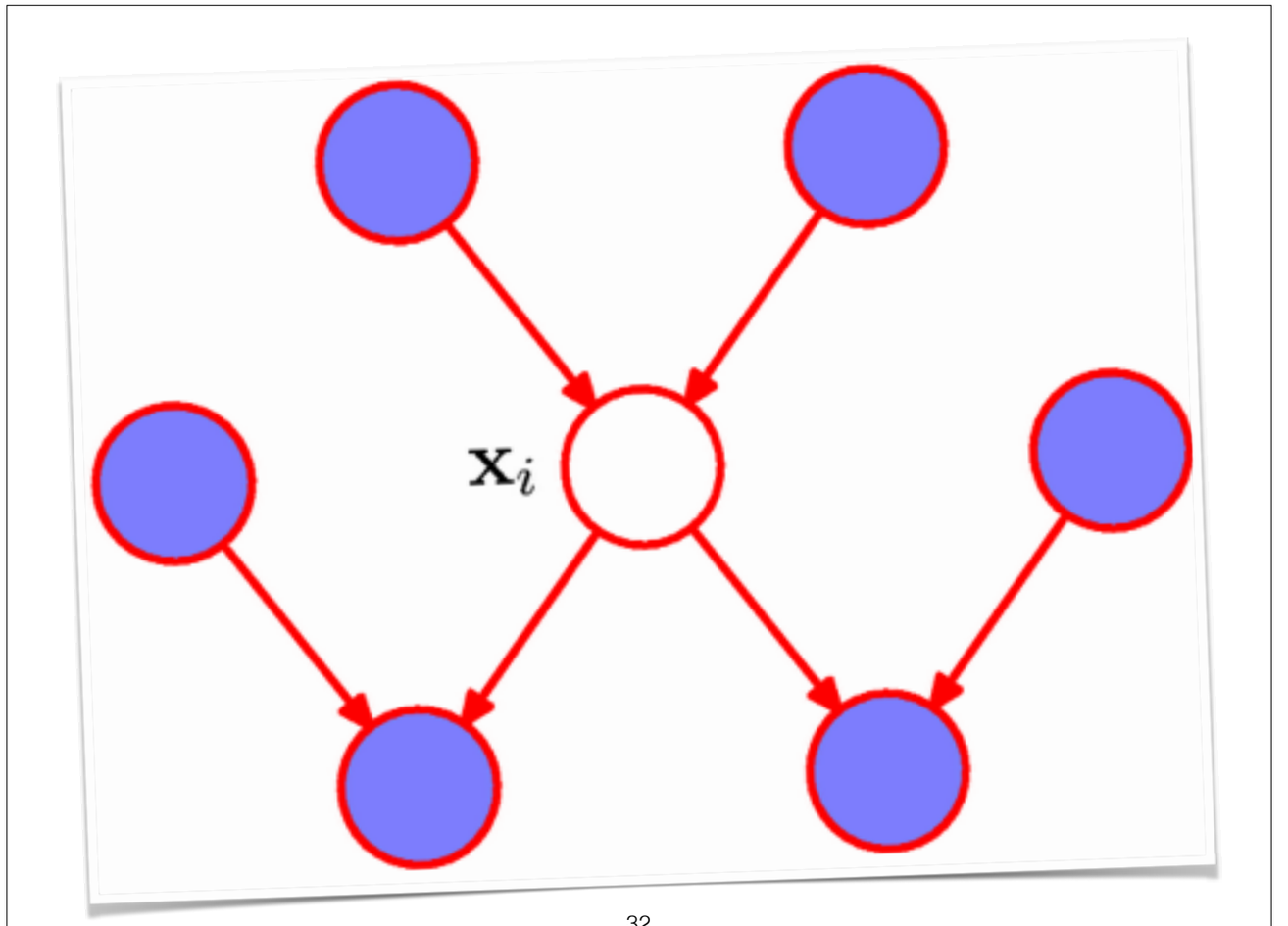
However in the right image, the path from a to b is blocked by node f and the conditional independence a ⫫ b | f property will be satisfied.



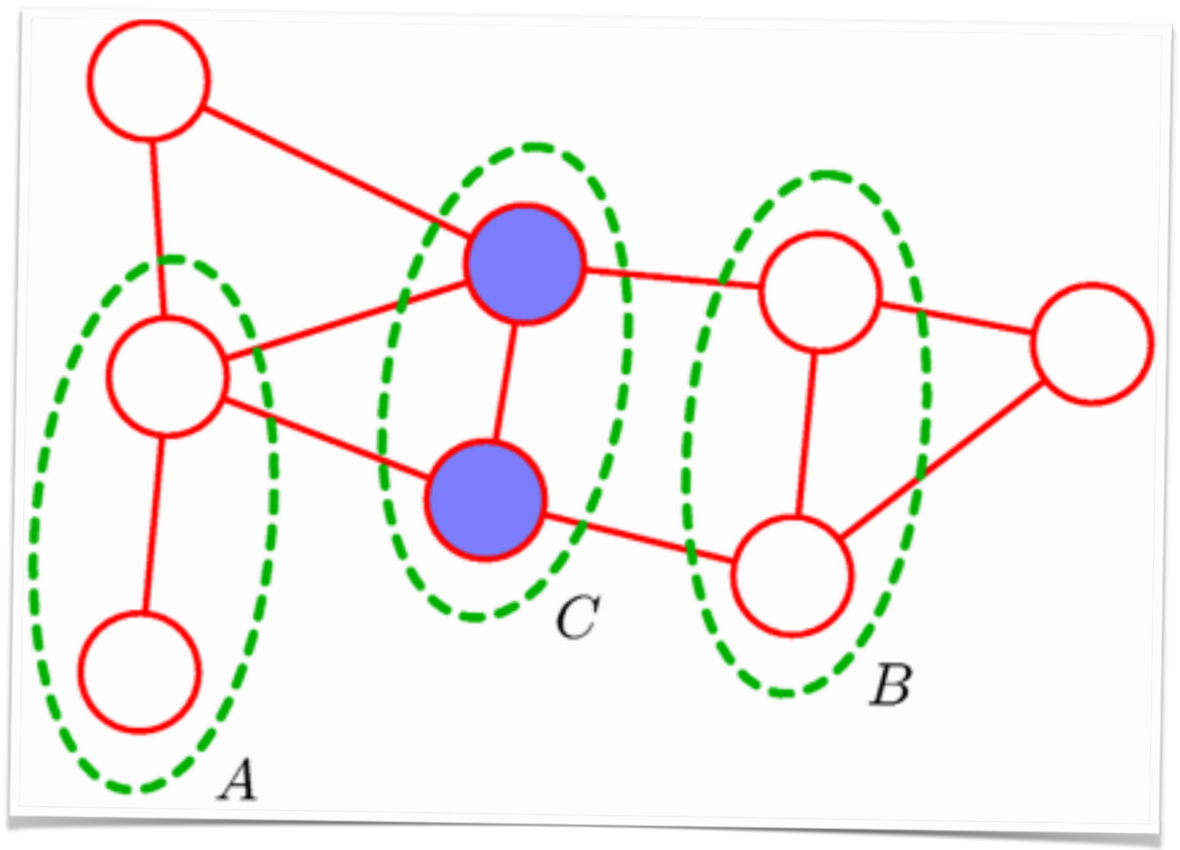
Here is an illustration of these graphical models to describe Bayesian polynomial regression. w are the coefficients and $t = (t_1,\ldots,t_N)^T$ are the observed data and the input data are the $x = (x_1,\ldots,x_N)$. The noise variance is σ^2 , and the hyperparameter α representing the precision of the Gaussian prior over w, all of which are parameters of the model rather than random variables.

Here we are representing many repeated units in this plate notation.

For the purpose of d-seperation these hyperparameter α and σ^2 behave in the same was as observed nodes but have no parents and all paths from them will be tail-to-tail and hence blocked. Consequently they play no role in d-separation.



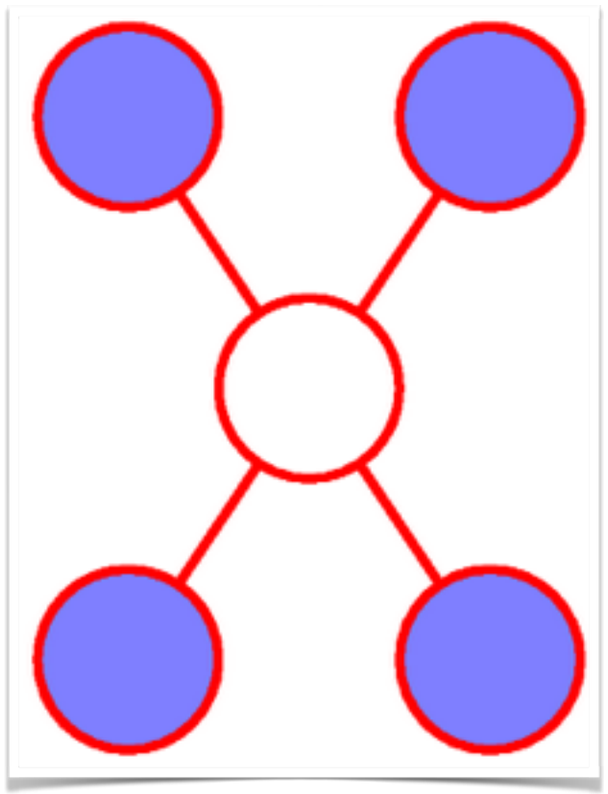
In a large model, we will often find it useful to find the minimal set of nodes on which a given node depends on. This set is called a Markov Blanket. It has the property that the conditional distribution of x_i conditioned on all the remaining variables is dependent only on the variables in the Blanket. Thus we can simplify complicated conditional expressions and use the graph to work it out quickly for a particular model.



One could also consider undirected edges too. However, it's not immediately apparent what a graph with undirected edges would correspond to. It turns out that one can actually make sense of undirected graphs, and doing so leads to a set of models called Markov random fields.

As one will remember, the head-to-head rule was a bit confusing. It seemed more or less the exact opposite of the tail-to-tail and head-to-tail rule. One might hope that the complexity of this head-to-head rule could be simplified if there was some way of "just making all of the dependency rules look the same." One way of doing this would be to use only undirected edges. Then, as there is no head or tail of an edge, all of the rules head-to-head, head-to-tail, and tail-to-tail will "look the same." In fact this intuition actually is correct, as we will now see.

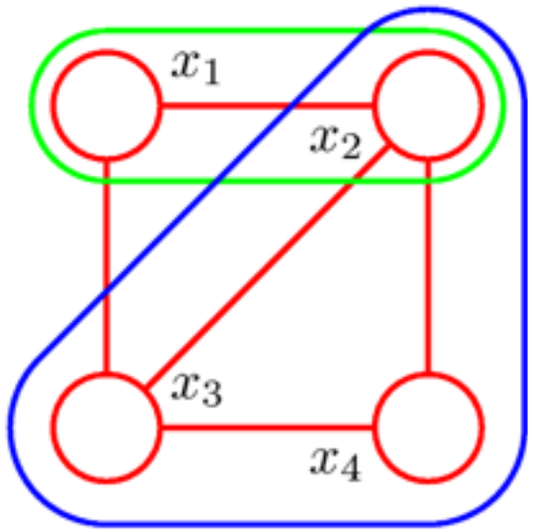
Assume we are given three sets of random variables denoted by A, B, and C and asked to determine if A and B are conditionally independent given C. One only needs to consider all paths from nodes in A to nodes in B . If all such paths contain at least one node in C , then A and B are conditionally independent given C. However, if at least one such path does not contain a node in C, then A and B need not be conditionally independent. This is much simpler than the directed case.



As a result of these conditional independence properties, determination of the Markov Blanket for a node in an undirected graph is relatively simple. For example above the Markov Blanket of the middle node is the set of nodes that share an edge with the middle node.

Joint Distribution

- undirected graph -> some factorisation of the joint probability but which one?
- cliques: set of all fully connected nodes
- factors are functions of the variables in the maximal cliques
- $p(x) = Z^{-1} \prod_c \Psi_c(x_c)$



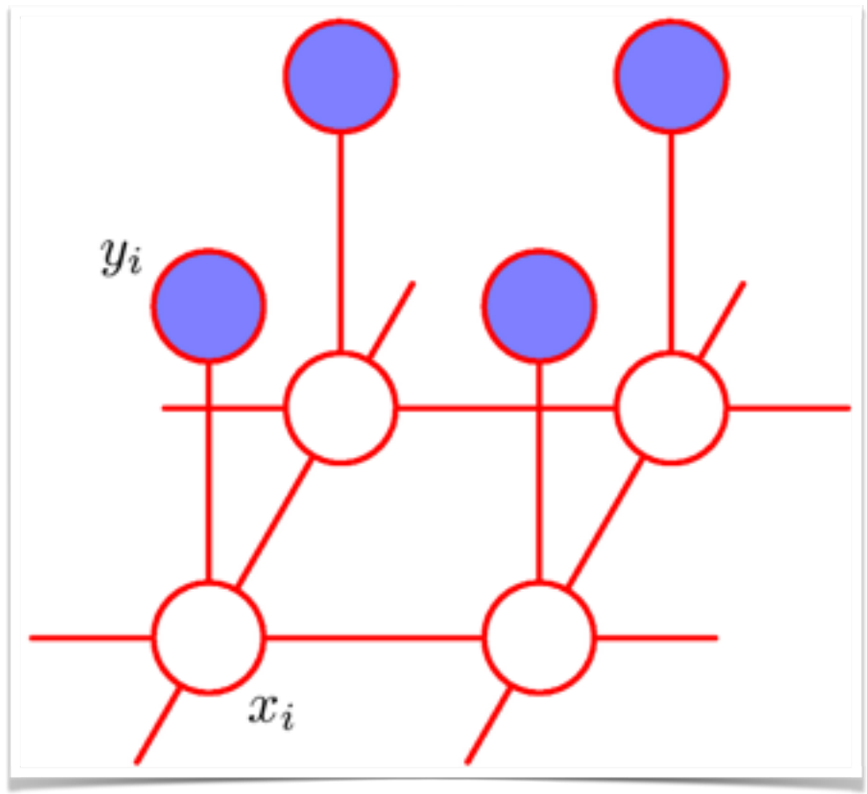
We still haven't examined how to write the joint distribution corresponding to an undirected graph. An undirected graph will correspond to some factorisation of the joint probability; the question is: Which factorisation?

If we consider two nodes x_i and x_j that are not connected by a link, then these variables must be conditionally independent given all other nodes in the graph. So the factorisation of the joint distribution must be such that x_i and x_j do not appear in the same factor for this conditional independence to hold for all possible distributions belonging to the graph.

This leads us to consider clique: subsets of the nodes in a graph such that there exists a link between all pairs of nodes in the subset. So all the nodes in the clique are fully connected. A maximal clique is a clique that stops being one if another node is added.

We can define the factors in the decomposition of the joint distribution to be functions of the variables in the maximal cliques. In other words the joint distribution is written as a product of potential functions over the maximal cliques of a graph.

By considering potential functions ≥ 0 we ensure $p(x) \geq 0$. Given this restriction we can make a precise relationship between factorisation and conditional independence.



We can illustrate the application of undirected graphs using an example of noise removal from a binary image. Let the y_i be binary pixel values -1 or +1. Suppose the image is obtained by taking a noise free image x_i and flipping the sign of the pixels with some probability. For small noise the x_i and y_i are correlated and we also know that neighbouring pixels x_i and x_j in an image are correlated. This can be captured in this markov random field model.

This graph has two types of cliques, each of which contains two variables. The cliques of the form $\{x_i,y_i\}$ and $\{x_i,x_j\}$ and we can associate energies on them and want these to be small when the signs are same and large when the signs are opposite. Once we have a complete energy function for the model we can use it as our potential function to define a joint distribution over x and y: p(x,y).

Now we fix the elements of y to the observed values and this defines a conditional p(x|y) over noise-free images. We then iteratively find the x with the lowest energy starting with an initial guess of x = y.

Undirected vs Directed

Consider first the problem of taking a model that is specified using a directed graph and trying to convert it to an undirected graph. In some cases this is straightforward.

In the undirected graph, the maximal cliques are simply the pairs of neighbouring nodes. We can generalise this construction to convert any distribution specified by a factorisation over a directed graph into one specified by a factorisation over an undirected graph.

The process of converting a directed graph into an undirected graph plays an important role in exact inference techniques. Going the other way is less common.

Inference

- Use graphical structure to find efficient algorithms for inference
- Make the structure of those algorithms transparent
- Many algorithms can be expressed in terms of messages around the graph
- With just two nodes we use Bayes' Theorem

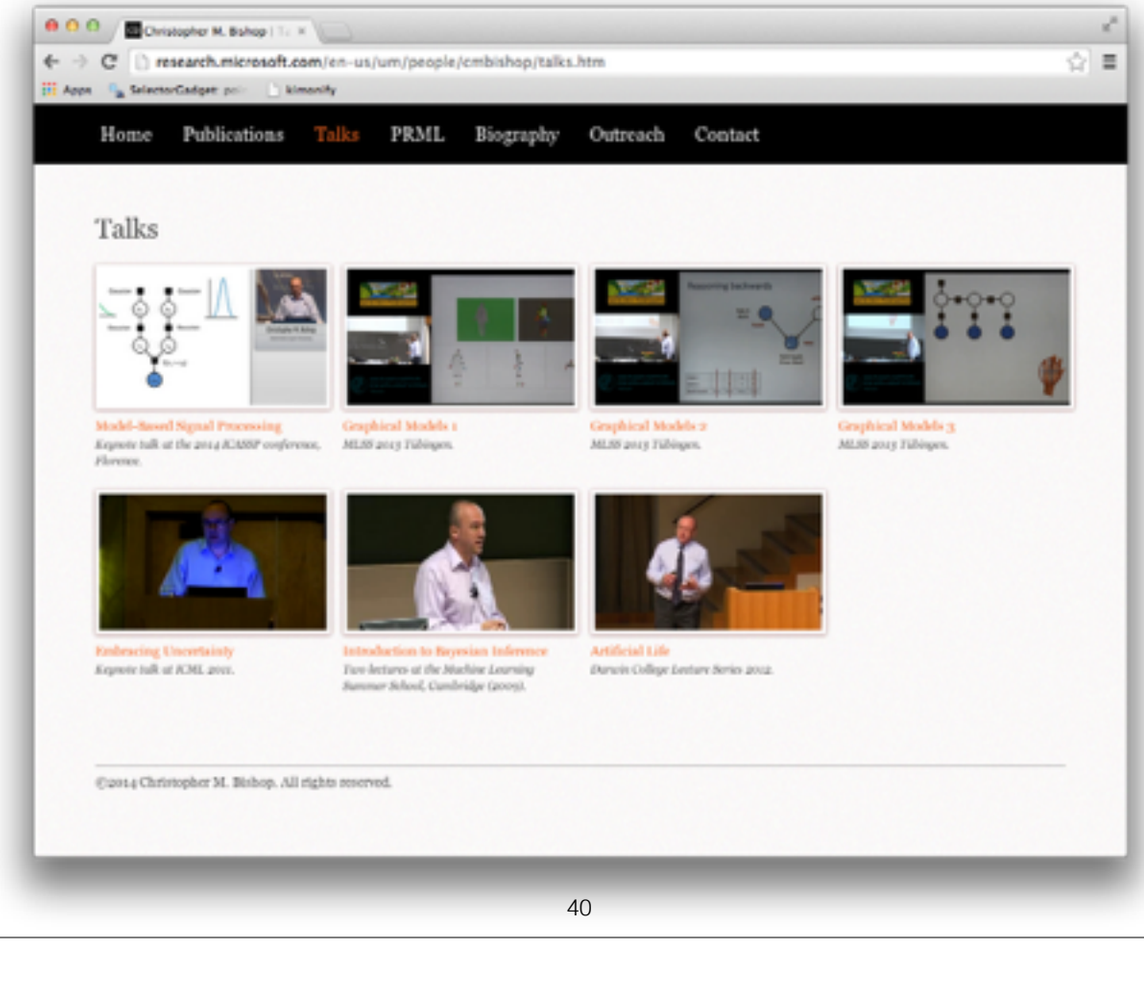
We turn now to the problem of inference in graphical models, where some of the nodes in a graph are clamped to observed values, and we wish to compute the posterior distributions of one or more subsets of other nodes.

Inference on a chain

Now consider a more complex problem involving the chain of nodes of the form shown above. Let us consider the inference problem of finding the marginal distribution $p(x_n)$ for a specific node x_n. Since there are no observed nodes p(xn) is a sum that grows exponentially was we increase the length N.

A more efficient algorithm can be obtained by conditional independence property of this graphical model and we end up doing N-1 operations or linear in the length of the chain. So we are able to exploit the many conditional independence properties of this simple graph in order to obtain an efficient calculation.

This touches then on ideas of Markov Chains.



You learn more from some online talk as well here: http://research.microsoft.com/en-us/um/people/cmbishop/talks.htm