Chapter No. 2
"Directed Graphical Models"
In this package, you will find:

The author’s biography

A preview chapter from the book, Chapter no.2 "Directed Graphical Models"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Kiran R Karkera is a telecom engineer with a keen interest in machine learning. He has been programming professionally in Python, Java, and Clojure for more than 10 years. In his free time, he can be found attempting machine learning competitions at Kaggle and playing the flute.

I would like to thank the maintainers of Libpgm and OpenGM libraries, Charles Cabot and Thorsten Beier, for their help with the code reviews.

For More Information:

Building Probabilistic Graphical Models with Python

In this book, we start with an exploratory tour of the basics of graphical models, their types, why they are used, and what kind of problems they solve. We then explore subproblems in the context of graphical models, such as their representation, building them, learning their structure and parameters, and using them to answer our inference queries.

This book attempts to give just enough information on the theory, and then use code samples to peep under the hood to understand how some of the algorithms are implemented. The code sample also provides a handy template to build graphical models and answer our probability queries. Of the many kinds of graphical models described in the literature, this book primarily focuses on discrete Bayesian networks, with occasional examples from Markov networks.

What This Book Covers

Chapter 1, Probability, covers the concepts of probability required to understand the graphical models.

Chapter 2, Directed Graphical Models, provides information about Bayesian networks, their properties related to independence, conditional independence, and D-separation. This chapter uses code snippets to load a Bayes network and understand its independence properties.

Chapter 3, Undirected Graphical Models, covers the properties of Markov networks, how they are different from Bayesian networks, and their independence properties.

Chapter 4, Structure Learning, covers multiple approaches to infer the structure of the Bayesian network using a dataset. We also learn the computational complexity of structure learning and use code snippets in this chapter to learn the structures given in the sampled datasets.

Chapter 5, Parameter Learning, covers the maximum likelihood and Bayesian approaches to parameter learning with code samples from PyMC.

Chapter 6, Exact Inference Using Graphical Models, explains the Variable Elimination algorithm for accurate inference and explores code snippets that answer our inference queries using the same algorithm.

For More Information:
Chapter 7, Approximate Inference Methods, explores the approximate inference for networks that are too large to run exact inferences on. We will also go through the code samples that run approximate inferences using loopy belief propagation on Markov networks.

Appendix, References, includes all the links and URLs that will help to easily understand the chapters in the book.

For More Information:
Directed Graphical Models

In this chapter, we shall learn about directed graphical models, which are also known as Bayesian networks. We start with the what (the problem we are trying to solve), the how (graph representation), the why (factorization and the equivalence of CPD and graph factorization), and then move on to using the Libpgm Python library to play with a small Bayes net.

Graph terminology

Before we jump into Bayes nets, let's learn some graph terminology. A graph $G$ consists of a set of nodes (also called vertices) $V = \{V_1, V_2, ..., V_n\}$ and another set of edges $E = \{E_1, E_2, ..., E_m\}$. An edge that connects a pair of nodes $V_i, V_j$ can be of two types: directed (represented by $V_i \rightarrow V_j$) and undirected (represented by $V_i - V_j$). A graph can also be represented as an adjacency matrix, which in the case of an undirected graph, if the position $G(i,j)$ contains 1, indicates an edge between $i$ and $j$ vertices. In the case of a directed graph, a value of 1 or -1 indicates the direction of the edge.

In many cases, we are interested in graphs in which all the edges are either directed or undirected, leading to them being called directed graphs or undirected graphs, respectively.

The parents of a $V_i$ node in a directed graph are the set of nodes that have outgoing edges that terminate at $V_i$.

The children of the $V_i$ node are the set of nodes that have incoming edges which leave $V_i$.

The degree of a node is the number of edges it participates in.

A clique is a set of nodes where every pair of nodes is connected by an edge. A maximal clique is the one that loses the clique property if it includes any other node.
If there exists a path from a node that returns to itself after traversing the other nodes, it is called a cycle or loop.

A **Directed Acyclic Graph (DAG)** is a graph with no cycles.

A **Partially Directed Acyclic Graph (PDAG)** is a graph that can contain both directed and undirected edges.

A forest is a set of trees.

**Python digression**

We will soon start to explore GMs using Python, and this is a good time to review your Python installation. The recommended base Python installation for the examples in this book is IPython, which is available for all platforms. Refer to the IPython website for platform-specific documentation.

We shall also use multiple Python libraries to explore various areas of graphical models. Unless otherwise specified, the usual way to install Python libraries is using `pip install <packagename>` or `easy_install <packagename>`.

To use the code in this chapter, please install Libpgm (https://pypi.python.org/pypi/libpgm) and scipy (http://scipy.org/).

**Independence and independent parameters**

One of the key problems that graphical models solve is in defining the joint distribution. Let’s take a look at the job interview example where a candidate with a certain amount of experience and education is looking for a job. The candidate is also applying for admission to a higher education program.

We are trying to fully specify the joint distribution over the job offer, which (according to our intuition) depends on the outcome of the job interview, the candidate's experience, and his grades (we assume that the candidate's admission into a graduate school is not considered relevant for the job offer). Three random variables \{Offer, Experience, Grades\} take two values (such as yes and no for the job offer and highly relevant and not relevant for the job experience) and the interview takes on three values, and the joint distribution will be represented by a table that has 24 rows (that is, $2 \times 2 \times 2 \times 3$).
Each row contains a probability for the assignment of the random variables in that row. While different instantiations of the table might have different probability assignments, we will need 24 parameters (one for each row) to encode the information in the table. However, for calculation purposes, we will need only 23 independent parameters. Why do we remove one? Since the sum of probabilities equals 1, the last parameter can be calculated by subtracting one from the sum of the 23 parameters already found.

<table>
<thead>
<tr>
<th>Experience</th>
<th>Grades</th>
<th>Interview</th>
<th>Offer</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.0800</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.0720</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.1080</td>
</tr>
</tbody>
</table>

Rows elided

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

The preceding joint distribution is the output of the `printjointdistribution.ipynb` IPython Notebook, which prints all permutations of the random variables' experience, grades, interview, and offer, along with their probabilities.

It should be clear on observing the preceding table that acquiring a fully specified joint distribution is difficult due to the following reasons:

- It is too big to store and manipulate from a computational point of view
- We'll need large amounts of data for each assignment of the joint distribution to correctly elicit the probabilities
- The individual probabilities in a large joint distribution become vanishingly small and are no longer meaningful to human comprehension

How can we avoid having to specify the joint distribution? We can do this by using the concept of independent parameters, which we shall explore in the following example.

Directed Graphical Models

The joint distribution $P$ over Grades and Admission is (throughout this book, the superscript 0 and 1 indicate low and high scores) as follows:

<table>
<thead>
<tr>
<th>Grades</th>
<th>Admission</th>
<th>Probability $(S,A)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^0$</td>
<td>$A^0$</td>
<td>0.665</td>
</tr>
<tr>
<td>$S^0$</td>
<td>$A^1$</td>
<td>0.035</td>
</tr>
<tr>
<td>$S^1$</td>
<td>$A^0$</td>
<td>0.06</td>
</tr>
<tr>
<td>$S^1$</td>
<td>$A^1$</td>
<td>0.24</td>
</tr>
</tbody>
</table>

When we reason about graduate admissions from the perspective of causality, it is clear that the admission depends on the grades, which can also be represented using the conditional probability as follows:

$$P(S,A) = P(S)P(A|S)$$

The number of parameters required in the preceding formula is three, one parameter for $P(S)$ and two each for $P(A|S^0)$ and $P(A|S^1)$. Since this is a simple distribution, the number of parameters required is the same for both conditional and joint distributions, but let's observe the complete network to see if the conditional parameterization makes a difference:

For More Information:
How do we calculate the number of parameters in the Bayes net in the preceding diagram? Let's go through each conditional probability table, one parameter at a time. Experience and Grades take two values, and therefore need one independent parameter each. The Interview table has 12 (3 x 4) parameters. However, each row sums up to 1, and therefore, we need two independent parameters per row. The whole table needs 8 (2 x 4) independent parameters. Similarly, the Offer table has six entries, but only 1 independent parameter per row is required, which makes 3 (1 x 3) independent parameters. Therefore, the total number of parameters is 1 (Experience) + 1 (Grades) + 12 (Interview) + 3 (Offer) amount to 17, which is a lot lesser than 24 parameters to fully specify the joint distribution. Therefore, the independence assumptions in the Bayesian network helps us avoid specifying the joint distribution.

The Bayes network
A Bayes network is a structure that can be represented as a directed acyclic graph, and the data it contains can be seen from the following two points of view:

- It allows a compact and modular representation of the joint distribution using the chain rule for Bayes network
- It allows the conditional independence assumptions between vertices to be observed

We shall explore the two ideas in the job interview example that we have seen so far (which is a Bayesian network, by the way).

The modular structure of the Bayes network is the set of local probability models that represent the nature of the dependence of each variable on its parents (Koller et al 3.2.1.1). One probability distribution each exists for Experience and Grades, and a conditional probability distribution (CPD) each exists for Interview and Offer. A CPD specifies a distribution over a random variable, given all the combinations of assignments to its parents. Thus, the modular representation for a given Bayes network is the set of CPDs for each random variable.

The conditional independence viewpoint flows from the edges (our intuition draws) between different random variables, where we presume that a call for a job interview must be dependent on a candidate's experience as well as the score he received in his degree course, and the probability of a job offer depends solely on the outcome of the job interview.
The chain rule

The chain rule allows us to define the joint distribution as a product of factors. In the job interview example, using the chain rule for probability, we can write the following formula:

\[ P(E, G, I, O) = P(E) \times P(G | E) \times P(I | E, G) \times P(O | E, G, I) \]

In the preceding formula, \( E, G, I, \) and \( O \) stand for Experience, Grades, Interview, and Offer respectively. However, we can use the conditional independence assumptions encoded by the graph to rewrite the joint distribution as follows:

\[ P(E, G, I, O) = P(E) \times P(G) \times P(I | E, G) \times P(O | I) \]

This is an example of the chain rule for Bayesian networks. More generally, we can write it as follows:

\[ P(X_1, X_2, \ldots, X_n) = \prod_i P(X_i | Par_G(X_i)) \]

Here, \( X_i \) is a node in the graph \( G \) and \( Par_G \) are the parents of the \( X_i \) node in the graph \( G \).

Reasoning patterns

In this section, we shall look at different kinds of reasoning used in a Bayes network. We shall use the Libpgm library to create a Bayes network. Libpgm reads the network information such as nodes, edges, and CPD probabilities associated with each node from a JSON-formatted file with a specific format. This JSON file is read into the NodeData and GraphSkeleton objects to create a discrete Bayesian network (which, as the name suggests, is a Bayes network where the CPDs take discrete values). The TableCPDFactorization object is an object that wraps the discrete Bayesian network and allows us to query the CPDs in the network. The JSON file for this example, job_interview.txt, should be placed in the same folder as the IPython Notebook so that it can be loaded automatically.

The following discussion uses integers 0, 1, and 2 for discrete outcomes of each random variable, where 0 is the worst outcome. For example, \( Interview = 0 \) indicates the worst outcome of the interview and \( Interview = 2 \) is the best outcome.

For More Information:
Causal reasoning

The first kind of reasoning we shall explore is called causal reasoning. Initially, we observe the prior probability of an event unconditioned by any evidence (for this example, we shall focus on the Offer random variable). We then introduce observations of one of the parent variables. Consistent with our logical reasoning, we note that if one of the parents (equivalent to causes) of an event is observed, then we have stronger beliefs about the child random variable (Offer).

We start by defining a function that reads the JSON data file and creates an object we can use to run probability queries. The following code is from the Bayes net-Causal Reasoning.ipynb IPython Notebook:

```python
from libpgm.graphskeleton import GraphSkeleton
from libpgm.nodedata import NodeData
from libpgm.discretebayesiannetwork import DiscreteBayesianNetwork
from libpgm.tablecpdfactorization import TableCPDFactorization

def getTableCPD():
    nd = NodeData()
    skel = GraphSkeleton()
    jsonpath="job_interview.txt"
    nd.load(jsonpath)
    skel.load(jsonpath)
    # load bayesian network
    bn = DiscreteBayesianNetwork(skel, nd)
    tablecpd=TableCPDFactorization(bn)
    return tablecpd
```

We can now use the specificquery function to run inference queries on the network we have defined. What is the prior probability of getting a $P(\text{Offer} = 1)$ Offer? Note that the probability query takes two dictionary arguments: the first one being the query and the second being the evidence set, which is specified by an empty dictionary, as shown in the following code:

```python
tcpd=getTableCPD()
tcpd.specificquery(dict(Offer='1'),dict())
```

The following is the output of the preceding code:

```
0.432816
```

It is about 43 percent, and if we now introduce evidence that the candidate has poor grades, how does it change the probability of getting an offer? We will evaluate the value of $P(\text{Offer} = 1 | \text{Grades} = 0)$, as shown in the following code:

```python
tcpd=getTableCPD()
tcpd.specificquery(dict(Offer='1'),dict(Grades='0'))
```

For More Information:
The following is the output of the preceding code:

0.35148

As expected, it decreases the probability of getting an offer since we reason that students with poor grades are unlikely to get an offer. Adding further evidence that the candidate's experience is low as well, we evaluate \( P(Offer = 1 | Grades = 0, Experience = 0) \), as shown in the following code:

```python
tcpd=getTableCPD()
tcpd.specificquery(dict(Offer='1'),dict(Grades='0',Experience='0'))
```

The following is the output of the preceding code:

0.2078

As expected, it drops even lower on the additional evidence, from 35 percent to 20 percent.

What we have seen is that the introduction of the observed parent random variable strengthens our beliefs, which leads us to the name causal reasoning.

In the following diagram, we can see the different paths taken by causal and evidential reasoning:

![Diagram showing causal and evidential reasoning paths]
Evidential reasoning

Evidential reasoning is when we observe the value of a child variable, and we wish to reason about how it strengthens our beliefs about its parents. We will evaluate the prior probability of high Experience \( P(\text{Experience} = 1) \), as shown in the following code:

```python
tcpd = getTableCPD()
tcpd.specificquery(dict(Experience='1'), dict())
```

The output of the preceding code is as follows:

0.4

We now introduce evidence that the candidate's interview was good and evaluate the value for \( P(\text{Experience} = 1 | \text{Interview} = 2) \), as shown in the following code:

```python
tcpd = getTableCPD()
print tcpd.specificquery(dict(Experience='1'), dict(Interview='2'))
```

The output of the preceding code is as follows:

0.864197530864

We see that if the candidate scores well on the interview, the probability that the candidate was highly experienced increases, which follows the reasoning that the candidate must have good experience or education, or both. In evidential reasoning, we reason from effect to cause.

Inter-causal reasoning

Inter-causal reasoning, as the name suggests, is a type of reasoning where multiple causes of a single effect interact. We first determine the prior probability of having high, relevant experience; thus, we will evaluate \( P(\text{Experience} = 1) \), as shown in the following code:

```python
tcpd = getTableCPD()
tcpd.specificquery(dict(Experience='1'), dict())
```

The following is the output of the preceding code:

0.4
Directed Graphical Models

By introducing evidence that the interview went extremely well, we think that the candidate must be quite experienced. We will now evaluate the value for $P(\text{Experience } = 1 | \text{Interview } = 2)$, as shown in the following code:

```python
tcpd = getTableCPD()
tcpd.specificquery(dict(Experience='1'), dict(Interview='2'))
```

The following is the output of the preceding code:

0.864197530864

The Bayes network confirms what we think is true (the candidate is experienced), and the probability of high experience goes up from 0.4 to 0.86. Now, if we introduce evidence that the candidate didn't have good grades and still managed to get a good score in the interview, we may conclude that the candidate must be so experienced that his grades didn't matter at all. We will evaluate the value for $P(\text{Experience } = 1 | \text{Interview } = 2, \text{Grades } = 0)$, as shown in the following code:

```python
tcpd = getTableCPD()
tcpd.specificquery(dict(Experience='1'), dict(Interview='2', Grades='0'))
```

The output of the preceding code is as follows:

0.909090909091

This confirms our hunch that even though the probability of high experience went up only a little, it strengthens our belief about the candidate's high experience. This example shows the interplay between the two parents of the Job interview node, which are Experience and Degree Score, and shows us that if we know one of the causes behind an effect, it reduces the importance of the other cause. In other words, we have explained the poor grades on observing the experience of the candidate. This phenomenon is commonly called explaining away.

The following diagram shows the path of interaction between nodes involved in inter-causal reasoning:
Bayes networks are usually drawn with the independent events on top and the influence flows from top to bottom (similar to the job interview example). It may be useful to recall causal reasoning flows from top to bottom, evidential reasoning flows from bottom to top, and inter-causal reasoning flows sideways.

**D-separation**

Having understood that the direction of arrows indicate that one node can influence another node in a Bayes network, let's see how exactly influence flows in a Bayes network. We can see that the grades eventually influence the job offer, but in the case of a very big Bayes network, it would not help to state that the leaf node is influenced by all the nodes at the top of the Bayes network. Are there conditions where influence does not flow? We shall see that there are simple rules that explain the flow of influence in the following table:

<table>
<thead>
<tr>
<th>No variables observed</th>
<th>Y has been observed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \leftarrow Y \leftarrow Z$</td>
<td>$X \leftarrow Y \leftarrow \oplus$</td>
</tr>
<tr>
<td>$X \rightarrow Y \rightarrow Z$</td>
<td>$X \rightarrow Y \rightarrow Z \oplus$</td>
</tr>
<tr>
<td>$X \leftarrow Y \rightarrow Z$</td>
<td>$X \leftarrow Y \rightarrow Z \ominus$</td>
</tr>
<tr>
<td>$X \rightarrow Y \leftarrow Z$</td>
<td>$X \rightarrow Y \leftarrow Z \ominus$</td>
</tr>
</tbody>
</table>

For More Information:
Directed Graphical Models

The preceding table depicts the open and closed active trails between three nodes $X$, $Y$, and $Z$. In the first column, no variables are observed, whereas in the second column, $Y$ has been observed.

We shall first consider the case where no random variables have been observed. Consider the chains of nodes in the first column of the preceding table. Note that the rules in the first three rows allow the influence to flow from the first to the last node.

Influence can flow along the path of the edges, even if these chains are extended for longer sequences. It must be pointed out that the flow of influence is not restricted by the directionality of the links that connect them.

However, there is one case we should watch out for, which is called the V-structure, $X \rightarrow Y \leftarrow Z$—probably called so because the direction of edges is pointed inwards.

In this case, the influence cannot flow from $X$ to $Z$ since it is blocked by $Y$. In longer chains, the influence will flow unless it is obstructed by a V-structure.

In this case, $A \rightarrow B \rightarrow X \leftarrow Y \rightarrow Z$ because of the V-structure at $X$ the influence can flow from $A \rightarrow B \rightarrow X$ and $X \leftarrow Y \rightarrow Z$ but not across the node $X$.

We can now state the concept of an active trail (of influence). A trail is active if it contains no V-structures, in the event that no evidence is observed. In case multiple trails exist between two variables, they are conditionally independent if none of the trails are active.

For More Information:
Let's now look at the second case where we do have observed evidence variables. It is easier to understand if we compare the case with the previous chains, where we observe the random variable $Y$.

The smiley trails shown in the previous table indicate an active trail, and the others indicate a blocked trail. It can be observed that the introduction of evidence simply negates a previously active trail, and it opens up if a previously blocking V-structure existed.

We can now state that a trail given some evidence $Z$ will be active if the middle node or any of its descendants in any V-structure (for example, $Y$ or its descendants in $X \rightarrow Y \leftarrow Z$) is present in the evidence set $Z$. In other words, observing $Y$ or any of its children will open up the blocking V-structure, making it an active trail.

Additionally, as seen in the following table, an open trail gets blocked by introduction of evidence and vice versa.

### The D-separation example

In this section, we shall look at using the job candidate example to understand D-separation. In the process of performing causal reasoning, we will query for the job offer and shall introduce the observed variables in the parents of the job offer to verify the concepts of active trails, which we have seen in the previous section. The following code is from the D-separation.ipynb IPython Notebook.

---

For More Information:
Directed Graphical Models

We first query the job offer with no other observed variables, as shown in the following code:

```python
getTableCPD().specificquery(dict(Offer='1'),dict())
```

The output of the preceding code is as follows:

0.432816

We know from the active trail rules that observing Experience should change the probability of the offer, as shown in the following code:

```python
getTableCPD().specificquery(dict(Offer='1'),dict(Experience='1'))
```

The output of the preceding code is as follows:

0.6438

As per the output, it changes. Now, let's add the Interview observed variable, as shown in the following code:

```python
getTableCPD().specificquery(dict(Offer='1'),dict(Interview='1'))
```

The output of the preceding code is as follows:

0.6

We get a slightly different probability for Offer. We know from the D-separation rules that observing Interview should block the active trail from Experience to Offer, as shown in the following code:

```python
getTableCPD().specificquery(dict(Offer='1'),dict(Interview='1',Experience='1'))
```

The output of the preceding code is as follows:

0.6

Observe that the probability of Offer does not change from 0.6, despite the addition of the Experience variable being observed. We can add other values of Interview object's parent variables, as shown in the following code:

```python
query=dict(Offer='1')
results=[getTableCPD().specificquery(query,e) for e in [dict(Interview='1',Experience='0'),
dict(Interview='1',Experience='1'),
dict(Interview='1',Grades='1'),
dict(Interview='1',Grades='0')]]
print results
```

For More Information:
The output of the preceding code is as follows:

\[0.6, 0.6, 0.6, 0.6\]

The preceding code shows that once the Interview variable is observed, the active trail between Experience and Offer is blocked. Therefore, Experience and Offer are conditionally independent when Interview is given, which means observing the values of the interview's parents, Experience and Grades, do not contribute to changing the probability of the offer.

### Blocking and unblocking a V-structure

Let's look at the only V-structure in the network, \( \text{Experience} \rightarrow \text{Interview} \leftarrow \text{Grades} \), and see the effect observed evidence has on the active trail.

```
getTableCPD().specificquery(dict(Grades='1'),dict(Experience='0'))
getTableCPD().specificquery(dict(Grades='1'),dict())
```

The result of the preceding code is as follows:

0.3
0.3

According to the rules of D-separation, the interview node is a V-structure between Experience and Grades, and it blocks the active trails between them. The preceding code shows that the introduction of the observed variable Experience has no effect on the probability of the grades.

```
getTableCPD().specificquery(dict(Grades='1'),dict(Interview='1'))
```

The following is the output of the preceding code:

0.413016270338

The following code should activate the trail between Experience and Grades:

```
getTableCPD().specificquery(dict(Grades='1'),dict(Interview='1',Experience='0'))
getTableCPD().specificquery(dict(Grades='1'),dict(Interview='1',Experience='1'))
```

The output of the preceding code is as follows:

0.588235294118
0.176470588235

The preceding code now shows the existence of an active trail between Experience and Grades, where changing the observed Experience value changes the probability of Grades.

For More Information:

**Factorization and I-maps**

So far, we have understood that a graph G is a representation of a distribution P. We can formally define the relationship between a graph G and a distribution P in the following way.

If G is a graph over random variables $X_1, X_2, ..., X_n$, we can state that a distribution P factorizes over G if $P(X_1, X_2, ..., X_n) = \prod P(X_i | \text{Par}_G(X_i))$. Here, $\text{Par}_G(X_i)$ are the parent nodes of $X_i$. In other words, a joint distribution can be defined as a product of each random variable when its parents are given.

The interplay between factorization and independence is a useful phenomenon that allows us to state that if the distribution factorizes over a graph and given that two nodes $X, Y \mid Z$ are D-separated, the distribution satisfies those independencies ($X, Y \mid Z$).

Alternately, we can state that the graph G is an **Independency map (I-map)** for a distribution P, if P factorizes over G because of which we can read the independencies from the graph, regardless of the parameters. An I-map may not encode all the independencies in the distribution. However, if the graph satisfies all the dependencies in the distribution, it is called a **Perfect map (P-map)**. The graph of the job interview is an example of an I-map.

**The Naive Bayes model**

We can sum this up by saying that a graph can be seen from the following two viewpoints:

- **Factorization**: This is where a graph allows a distribution to be represented
- **I-map**: This is where the independencies encoded by the graph hold in the distribution

The Naive Bayes model is the one that makes simplistic independence assumptions. We use the Naive Bayes model to perform binary classification. Here, we are given a set of instances, where each instance consists of a set of features $X_1, X_2, ..., X_n$ and a class $y$. The task in classification is to predict the correct class of $y$ when the rest of the features $X_1, X_2, ..., X_n$ are given.

For example, we are given a set of newsgroup postings that are drawn from two newsgroups. Given a particular posting, we would like to predict which newsgroup that particular posting was drawn from. Each posting is an instance that consists of a bag of words (we make an assumption that the order of words doesn't matter, just the presence or absence of the words is taken into account), and therefore, the $X_1, X_2, ..., X_n$ features indicate the presence or absence of words.

Here, we shall look at the Naive Bayes model as a classifier.

For More Information:  
The difference between Naive Bayes and the job candidate example is that Naive Bayes is so called because it makes naïve conditional independence assumptions, and the model factorizes as the product of a prior and individual conditional probabilities, as shown in the following formula:

\[ P(C, X_1, X_2, \ldots, X_n) = P(C) \prod_{i=2}^{n} P(X_i | C) \]

Although the term on the left is the joint distribution that needs a huge number of independent parameters \(2^{n+1} - 1\) if each feature is a binary value, the Naive Bayes representation on the right needs only \(2n+1\) parameters, thus reducing the number of parameters from exponential (in a typical joint distribution) to linear (in Naive Bayes).

In the context of the newsgroup example, we have a set of words such as \{atheist, medicine, religion, anatomy\} drawn from the alt.atheism and sci.med newsgroups. In this model, you could say that the probability of each word appearing is only dependent on the class (that is, the newsgroup) and independent of other words in the posting. Clearly, this is an overly simplified assumption, but it has been shown to have a fairly good performance in domains where the number of features is large and the number of instances is small, such as text classification, which we shall see with a Python program.

Once we see a strong correlation among features, a hierarchical Bayes network can be thought of as an evolved version of a Naive Bayes model.
The Naive Bayes example

In the Naive Bayes example, we will use the Naive Bayes implementation from Scikit-learn—a machine learning library to classify newsgroup postings. We have chosen two newsgroups from the datasets provided by Scikit-learn (alt.atheism and sci.med), and we shall use Naive Bayes to predict which newsgroup a particular posting is from. The following code is from the Naive Bayes.ipynb file:

```python
from sklearn.datasets import fetch_20newsgroups
import numpy as np
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics,cross_validation
from sklearn.feature_extraction.text import TfidfVectorizer
cats = ['alt.atheism', 'sci.med']
newsgroups= fetch_20newsgroups(subset='all',remove=('headers', 'footers', 'quotes'), categories=cats)
```

We first load the newsgroup data using the utility function provided by Scikit-learn (this downloads the dataset from the Internet and may take some time). The newsgroup object is a map, the newsgroup postings are saved against the data key, and the target variables are in newsgroups.target, as shown in the following code:

```python
newsgroups.target
```

The output of the preceding code is as follows:

```python
array([1, 0, 0, ..., 0, 0, 0], dtype=int64)
```

Since the features are words, we transform them to another representation using Term Frequency-Inverse Document Frequency (Tfidf). The purpose of Tfidf is to de-emphasize words that occur in all postings (such as "the", "by", and "for") and instead emphasize words that are unique to a particular class (such as religion and creationism, which are from the alt.atheism newsgroup). We can do the same by creating a TfidfVectorizer object and then transforming all the newsgroup data to a vector representation, as shown in the following code:

```python
vectorizer = TfidfVectorizer()
vectors = vectorizer.fit_transform(newsgroups.data)
```

Vectors now contain features that we can use as the input data to the Naive Bayes classifier. A shape query reveals that it contains 1789 instances, and each instance contains about 24 thousand features, as shown in the following code. However, many of these features can be 0, indicating that the words do not appear in that particular posting:

```python
vectors.shape
```
The output of the preceding code is as follows:

\((1789, \ 24202)\)

Scikit-learn provides a few versions of the Naive Bayes classifier, and the one we will use is called \texttt{MultinomialNB}. Since using a classifier typically involves splitting the dataset into train, test, and validation sets, then training on the train set and testing the efficacy on the validation set, we can use the utility provided by Scikit-learn to do the same for us. The \texttt{cross_validation.cross_val_score} function automatically splits the data into multiple sets and returns the \(f_1\) score (a metric that measures a classifier's accuracy), as shown in the following code:

```python
clf = MultinomialNB(alpha=.01)
print "CrossValidation Score: ", np.mean(cross_validation.cross_val_score(clf,vectors, newsgroups.target, scoring='f1'))
```

The output of the preceding code is as follows:

```
CrossValidation Score:  0.954618416381
```

We can see that despite the assumption that all features are conditionally independent when the class is given, the classifier maintains a decent \(f_1\) score of 95 percent.

**Summary**

In this chapter, we learned how conditional independence properties allow a joint distribution to be represented as the Bayes network. We then took a tour of types of reasoning and understood how influence can flow through a Bayes network, and we explored the same concepts using Libpgm. Finally, we used a simple Bayes network (Naive Bayes) to solve a real-world problem of text classification.

In the next chapter, we shall learn about the undirected graphical models or Markov networks.

For More Information:

Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.