

Probability Theory: Creating Bayesian Models

Bayesian models are the perfect tool for use-cases where there are multiple easily observable outcomes and hard-to-diagnose underlying causes, using a combination of graph theory and Bayesian statistics. Use this course to learn more about stating and interpreting the Bayes theorem for conditional probabilities. Discover how to use Python to create a Bayesian network and calculate several complex conditional probabilities using a Bayesian machine learning model. You'll also examine and use naive Bayes models, which are a category of Bayesian models that assume that the explanatory variables are all independent of each other. Once you have completed this course, you will be able to identify use cases for Bayesian models and construct and effectively employ such models.

Table of Contents

- [1. Video: Course Overview \(it_daprthdj_03_enus_01\)](#)
 - [2. Video: Bayes Theorem \(it_daprthdj_03_enus_02\)](#)
 - [3. Video: Bayesian Networks \(it_daprthdj_03_enus_03\)](#)
 - [4. Video: Using the Chain Rule with Bayesian Networks \(it_daprthdj_03_enus_04\)](#)
 - [5. Video: Creating a Bayesian Network Model \(it_daprthdj_03_enus_05\)](#)
 - [6. Video: Associating Probabilities with Bayesian Networks \(it_daprthdj_03_enus_06\)](#)
 - [7. Video: Computing Probabilities from Bayesian Networks \(it_daprthdj_03_enus_07\)](#)
 - [8. Video: Creating Bayesian Machine Learning Models \(it_daprthdj_03_enus_08\)](#)
 - [9. Video: Predicting Values Using a Bayesian Model \(it_daprthdj_03_enus_09\)](#)
 - [10. Video: Interpreting Probabilities Generated by Bayesian Models \(it_daprthdj_03_enus_10\)](#)
 - [11. Video: Understanding and Creating Naive Bayes Models \(it_daprthdj_03_enus_11\)](#)
 - [12. Video: Testing Naive Bayes Machine Learning Models \(it_daprthdj_03_enus_12\)](#)
 - [13. Video: Course Summary \(it_daprthdj_03_enus_13\)](#)
- [Course File-based Resources](#)

1. Video: Course Overview (it_daprthdj_03_enus_01)



Objectives

- *discover the key concepts covered in this course*

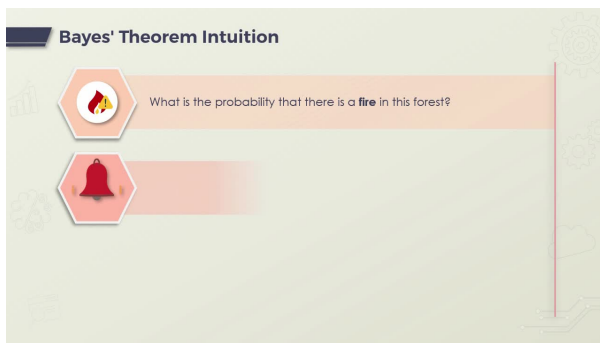
[Video description begins] *Topic title: Course Overview.* [Video description ends]

Hi and welcome to this course, Creating Bayesian Models. [Video description begins] *Your host for this session is Vitthal Srinivasan. He is a Software engineer and big data expert.* [Video description ends] My name is Vitthal Srinivasan and I will be your instructor for this course. A little bit about myself first. I did my Master's from Stanford University and have worked at various companies, including Google and Credit Suisse. I presently work for Loonycorn, a studio for high-quality video content.

Consider a situation where we have multiple symptoms that is easily observable outcomes and multiple possible illnesses that is hard to diagnose underlying causes. Given a set of symptoms, how likely is a specific illness. Bayesian models are just the right tool for such use cases using a combination of graph theory and Bayesian statistics. You will start this particular course by stating and interpreting Bayes theorem for conditional probabilities.

You'll then use Python to create a Bayesian network and calculate several complex conditional probabilities using a Bayesian machine learning model. Finally, you will define and use Naive Bayes models, which are a category of Bayesian models that assume that the explanatory variables are all independent of each other. By the time you finish this course, you will be able to identify use cases for Bayesian models and construct and effectively employ such models.

2. Video: Bayes Theorem (it_daprthdj_03_enus_02)



Objectives

- *define and understand the Bayes theorem*

[Video description begins] *Topic title: Bayes Theorem. Your host for this session is Vitthal Srinivasan.* [Video description ends]

We'll now turn our attention to a really important result in probability known as Bayes' Theorem. Bayes' Theorem is so important and so groundbreaking that it's almost given rise to its own branch of statistics. Bayesian statistics, which are founded on Bayes' Theorem, differ from traditional frequentist statistics because in Bayesian statistics, we treat any estimate of a particular value as something that has to be updated as new information relevant to that estimate comes in.

In contrast, in frequentist statistics probabilities are viewed as the limits of the relative frequencies of an event after an infinite number of trials. Without getting too deep into the difference between Bayesian and frequentist statistics, let's try and quickly understand Bayes' Theorem, the famous result which is responsible for all of this. Bayes' Theorem describes the probability of an event, based on prior knowledge of conditions that might be related to the event.

We'll go ahead and see how this Bayes' Theorem translates this intuition into a mathematical formula. Let's try and do this using a real-world example. Consider a situation where we are observing conditions related to temperature and other atmospheric conditions near a forest. We are trying to estimate the probability of a fire in this forest. Let's say that this probability is P . This is the probability that we are now going to try and estimate.

At this point, we don't really have any information about this specific forest yet. So our first estimate of P is going to probably come from overall estimates of probabilities of forest fires. And that initial value of P is called the prior. The prior probability is something that we assign before we have any specific information about the exact problem that we are trying to solve.

However, we now start monitoring the readings coming in from various instruments. And these give us additional bits of information which allow us to update our estimate of P . So, for instance, maybe we spot smoke somewhere in the forest. This is an important piece of new information. And it causes us to update our estimate of P upwards. So we are increasing our estimate that there is indeed a fire in this forest.

But this is not the end of our observation process. Let's say that an hour passes and we see that it's been raining for that entire period. Now our probability estimate is going to go down. So at each point, as new information comes in about events related to what we are trying to forecast, we are going to update our probability estimate. This is the foundation of Bayesian statistics and all of this is based on Bayes' Theorem.

So Bayes' Theorem is a way for us to calculate probabilities of an event when we know the probabilities of related events. And the precise form of Bayes' Theorem is now visible on-screen. The probability of A given B is the probability of A multiplied by the probability of B given A divided by the probability of B . Now, you might think that this is a somewhat circular definition because on the left-hand side of the equal to sign, we have the probability of A given B , and on the right-hand side, we have the probability of B given A .

But in reality, this is actually a very easy formula to compute because it's often the case that the probability of B given A is a lot more readily available than the probability of A given B . This formula for Bayes' Theorem is an important one. So let's break it down. $P(A | B) = P(A) * P(B | A)$ divided by $P(B)$. What's on the left-hand side of the equal to sign is the probability of A given B . This is what we are trying to find.

And you can think of this as the probability of there being a forest fire given that we've spotted smoke and given that it's been raining for the past hour. So in general, $P(A | B)$ is what we are trying to estimate. And it answers the question, how often does A occur given that B has occurred? Now, $P(B | A)$ measures something quite different.

It tells us how often does B occur given that A has occurred. So in our forest fire example, $P(B | A)$ would be the probability that we see smoke and that it's been raining for the past hour given that there is a fire in the forest. And there are still two more probabilities here. $P(A)$ which is how often does A occur on its own, and $P(B)$, how often does B occur on its own.

So here, for instance, maybe A is the probability of a fire in the forest and B is the probability of seeing smoke in the forest. To keep things simple for a moment, let's forget about the fact that it rained for the past hour, and just focus on the one bit of additional information that's available to us, which is that smoke was spotted. Let's now start to plug in some of these calculations.

So here, let the event A be the event that there is a fire and the event B that we've spotted smoke. $P(A)$ is the probability of a fire in the absence of any additional information. And this probability is probably available from historic data; how often we've actually observed fires in this forest or other forests like this. $P(B)$ is the probability of spotting smoke.

And this, again, is something that we can probably measure quite easily using historic records. How often was smoke observed in this particular location or from this observation post? So here $P(A)$ is $P(\text{fire})$ and $P(B)$ is $P(\text{smoke})$. Where it gets interesting is where we try to compute the probability of fire given that smoke was seen. So $P(\text{fire}|\text{smoke})$ is what we are really trying to estimate.

Given that we have observed smoke, what's the probability that there is a fire? And this probability takes into account the crucial additional information which we have at our disposal, which is that smoke was indeed observed. And the last relevant probability here is the probability

of smoke given fire. So this probability is trying to estimate the probability of smoke being observed if a fire has actually occurred.

Now, this probability is also something that's probably available much more easily than the probability of fire given smoke. The whole point of Bayes' Theorem is that the probability of fire given smoke, which we'd like to estimate is a lot harder to observe than any of the other three probabilities. And this includes the probability of smoke given fire.

Now, let's try and find each of these individual probabilities and use them in order to calculate the probability of fire given that we've observed smoke. Using historic data, we find that probability of dangerous fires in general in this area is pretty low, just 2%. The probability of spotting smoke is pretty high because there are lots of relatively innocuous sources of smoke other than forest fires, such as, for instance, campfires and barbeques.

So the probability of spotting smoke works out to be 15%. This probability is also available easily in the form of log entries from the park rangers' notes. Now the third and most important bit of probability here is the probability that smoke is seen if there is a fire. And that probability is 98%. And this comes from long-term studies of forest fires which show that only a very small proportion of forest fires do not lead to smoke being visible at a certain distance.

So the probability of a forest fire emitting visible smoke is 98%. Now, let's plug all of these values in to our Bayes' formula. $P(\text{fire}) = 2\%$, $P(\text{smoke}) = 15\%$, and $P(\text{smoke} | \text{fire}) = 98\%$. And when we perform this calculation, we find that the probability of fire given that we observed smoke is 2 multiplied by 98% divided by 15%.

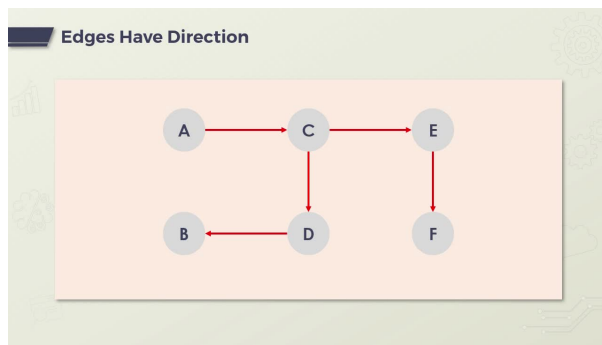
We've dropped the percent signs from the formula because we have percentages in the numerator as well as the denominator. And when we perform this calculation, we find that $P(\text{fire} | \text{smoke})$ works out to 13.06%. Now, this is not an extremely high probability, but it's about six and a half times higher than the probability of a fire by itself.

So the fact that we observed smoke increased the chances that there was an underlying fire from 2% to 13.06%. That's an increase by a factor of almost 6.5. Now, in addition to the formula for Bayes' Theorem, we've also got to be familiar with specific terms for each of these elements.

The probability of fire given smoke, that's what we are trying to estimate, is the Posterior probability. The probability of fire in the absence of any other information is called the Prior probability. The probability of smoke given fire is called the Likelihood. And finally, the probability of smoke is referred to as the Evidence. These are all pretty common terms, and most especially the terms that you'll find are the Prior and the Posterior probabilities.

Both the Prior and the Posterior give us the probability of a fire. However, the Prior gives us the unconditional probability. That's a probability that we have a fire in the absence of any other information. The Posterior probability as its name would suggest, represents our revised probability estimate after we incorporate all additional information that we have at our disposal. And Bayes' Theorem, and indeed Bayesian statistics, is all about making use of Posterior probabilities rather than Prior probabilities.

3. Video: Bayesian Networks (it_daprthdj_03_enus_03)



Objectives

- *enumerate the architecture of Bayesian networks*

[Video description begins] *Topic title: Bayesian Networks. Your host for this session is Vitthal Srinivasan.* [Video description ends]

In our previous example, we made use of information about smoke and fires in order to predict the probability that we have a forest fire on our hands given that we've spotted smoke. Along the way, we noted that there were many other innocuous causes of smoke, such as campfires and barbeques as well. This leads to an interesting class of problems.

What if there is a phenomenon which has multiple different possible causes? And what if we'd like to find the probability of that phenomenon across all of those causes? This is where a tool known as the Bayesian network can come in handy. A Bayesian network builds upon the idea of Bayes' theorem to give us a probabilistic model. This is a model expressed in the form of a graph.

And that graph specifically is a directed-acyclic graph. In a Bayesian network, we have a set of nodes representing random variables, and then we have edges connecting those nodes where the edges contain probability information. Let's go ahead and see how a Bayesian network can be used. A good place to start is with the idea of a directed-acyclic graph. On-screen now you can see a graph.

And this graph consists of nodes as well as edges. The nodes are the circles corresponding to A, C, E, B, D and F. And those nodes are linked by edges and crucially, those edges have a direction. You can see that every one of these edges is represented by a ray that is, there is an arrowhead at one end of the edge, but not at the other.

The term directed in the name directed-acyclic graph, comes from the fact that all of these edges have directions. And the term acyclic comes from the fact that this graph has no cycles. There's no way to start from one node, walk along edges in the graph, and find yourself back at that same node. Such a graph would have been set to contain cycles, and a directed-acyclic graph does not contain any cycles.

So that's what a directed-acyclic graph is. So let's now turn our attention to a directed-acyclic graph as a probabilistic graphical model. In our DAG or directed-acyclic graph, we continue to have nodes, but these nodes will now represent random variables, specifically, Bayesian random variables.

Examples of such random variables could be observable quantities, latent variables which cannot be directly observed, but which can be inferred from the state of the system, unknown parameters or hypotheses that we would like to evaluate. The edges are going to represent conditional dependencies. Consider for instance, the edge between A and C. This edge links A and C, and this tells us that the node C depends on the node A.

Or in other words, A can cause C. In similar fashion, the directed edge going from E to F tells us that E can cause F. Nodes that are not linked to each other, such as, for instance, C and F represent

conditionally independent variables. So here, for instance, the nodes B and F are not connected to each other, which means that B is conditionally independent of F.

In similar fashion, there's no link between D and E, and this in turn means that D is conditionally independent of E. Before we move on, a quick word about the causal relationships encoded within this network. You can see here that, for instance, A seems to cause C and C in turn seems to cause E and D. It's very important to keep in mind, however, that these causal relationships are not deterministic.

So if, for instance, A were to occur, we cannot say that C is sure to occur. Rather, these causal relationships are all probabilistic. If A occurs, there is a certain probability that C will occur. And if C occurs, then there is a corresponding probability that its dependents will occur. All of this information is associated with a probability function, and that probability function, which is usually represented in the form of a table, is going to be associated with each node.

So in our Bayesian network, every one of the nodes represents a Bayesian random variable, and every one of those nodes is also going to be associated with a probability function table. That probability function, of course, like any function, has inputs and outputs. And the inputs into the function are going to be values of all of the parent nodes for this particular node.

So all of the inputs into the probability function are going to be values for all of those causal variables which might impact this particular node. And the output is going to be another probability table with all of the possible values of this node and their associated probabilities. Where Bayesian network really comes in handy is in use cases where we would like to take an event that has occurred and then predict the likelihood that a specific underlying cause was responsible.

So, again, we use the Bayesian network when we have the final effect and where we'd like to predict or attribute that effect across many possible different causes. And we are interested in finding the contributing factor which most directly drove the output. Consider, for instance, a scenario where we have a large number of symptoms.

These have been observed in one or more patients, and then we also have a large number of possible illnesses or diseases. The symptoms are the effects. They are all observable. The diseases are all of the causes. And what we'd like to do is, given a set of symptoms, we'd like to find the probabilities that these symptoms are caused by different underlying diseases.

This is a powerful use for Bayesian networks, and this is what we'll be implementing in the demos coming up ahead. For now, however, let's work with a simple conceptual example specifically related to wet grass with rain and the sprinkler. And this is the same example which has been worked out on the Wikipedia page for Bayesian networks. The problem setup is as follows.

We have a patch of grass such as a lawn, and we can test or observe whether that grass is wet or not. If we do happen to find that the grass was wet, we'd like to know whether that wet grass was caused by a sprinkler system being active or whether it was merely caused by rain. So we have one observable effect, which is the wet grass, and we have two possible underlying causes, the active sprinkler system and the presence of rain.

To use a Bayesian network, we'll set up a directed-acyclic graph, and in that graph we have three nodes: the state of the sprinkler, whether it's rained or not, and the state of the grass. Here on-screen now is our first attempt at modeling the system using a Bayesian network. You can see that we've initially modeled arrows or edges which go from the nodes for the sprinkler and the rain to the wet grass.

We don't yet have any arrow leading from rain to the sprinkler. This set up, as it stands right now, suggests that the wet grass may have been caused either by the sprinkler or by the rain. The nodes for the sprinkler and the rain don't have any parent nodes. That is, there are no edges currently leading into either of these. And that means that we have the sprinkler system as well as the rain as conditionally independent of each other.

However, after thinking about it a little bit, we tweak our DAG. We decide that when it's raining, we are unlikely to turn on the sprinkler, and that causes us to have an arrow which goes from the node for rain to the node for the sprinkler. Note that there is no arrow going in the other way. So the node for the rain is still conditionally independent of everything else.

There's no external factor in our system which explains whether it's going to rain or not. However, if it does rain, then the probability that the sprinkler is going to be on does indeed change. And now we have a DAG in which we have relationships from rain to the sprinkler, from rain to the wet grass, and also from the sprinkler to the wet grass. Now that we've got the direction of the causality right, let's focus on each of the nodes.

We decide that each one of these three nodes can be modeled as a Boolean variable. Remember that a node in a Bayesian network represents any Bayesian random variable. And here we've chosen for these nodes to represent Boolean variables. So each of these variables has two possible values, true and false. Now that we know the states of each of these variables, we've got to set up the conditional probability tables.

Let's start with the conditional probability table for rain. There's nothing in our system which explains whether it's going to rain or not, and so this table is pretty simple. There are only two possible values, true or false. And from past data we find that there is a 20% probability that it is raining, and an 80% probability that it's not raining. So that does it for the conditional probability table for the node rain.

Next, let's set up a similar table for the sprinkler node. Now, this sprinkler node is not like the rain node because it has one input and that input is whether it's raining or not. So you can see here that we have rows which correspond to whether it's raining or not, and then we have columns, corresponding to whether the sprinkler is on or not.

If it's not raining, so R is equal to F , then the probability of the sprinkler being on is 0.4. A different way of putting this is that the conditional probability of the sprinkler being on, given that it's not raining, is 0.4, that is 40%. If it's not raining, then the conditional probability of the sprinkler being off is 0.6, that is 60%. Now if it is raining, then there's only a 1% probability that the sprinkler is going to be on, and a 99% probability that the sprinkler is going to be off.

So this conditional probability table is a function. This function has one input. That input tells us whether it's raining or not. And there's one output which tells us whether the sprinkler is on or not. The input has two possible values. The output also has two possible values. And so there are four possible combinations, 2 raised to 2.

And that's why this table has four values embedded within it. If we now turn our attention to the grass, we are going to have a similar conditional probability table there as well, but this one is going to have not two, not four, but eight possible probabilities. And that's because the conditional probability of the grass being wet depends on two inputs, whether the sprinkler is on and whether it's raining.

Each of those two inputs has two possible values. So that leads to four possible input combinations. Then there's also the output, which is whether the grass is wet or not. That output has two values of its own. And so we have a table with four multiplied by two, that is eight probabilities. This conditional probability table tells us that if the sprinkler is off and that it's not raining, then there is no chance at all, there is a zero probability that the grass is going to be wet.

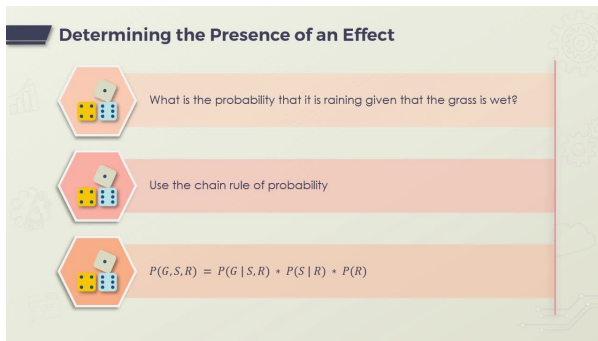
On the other hand, if the sprinkler is on and it's raining, there is a 99% probability that the grass is going to be wet. [Video description begins] *If the Sprinkler is off and it is Raining, there is an 80% probability for the grass to be wet. If the Sprinkler is on and it is not Raining, there is a 90% probability for the grass to be wet.* [Video description ends] Now the table, which gives probabilities for the grass being wet or not, has complimentary outcomes.

So, for instance, if the grass being wet is equal to true, and that probability is 0, then the probability that the grass is not wet is going to be 1. That's why every one of the rows in this

particular table sums to 1. We have two numbers which represent the probabilities of complementary events, and those two probabilities have got to sum to 1.

We've now set up all of the conditional probability tables as well as the directed-acyclic graph. What's left for us to do is to actually make use of these probabilities in order to attribute causality. That is, in order to make statements like the grass is wet and this is caused by the sprinkler with a probability of X%.

4. Video: Using the Chain Rule with Bayesian Networks (it_dapnthdj_03_enus_04)



Objectives

- *use the chain rule with Bayesian networks*

[Video description begins] *Topic title: Using the Chain Rule with Bayesian Networks. Your host for this session is Vitthal Srinivasan.* [Video description ends]

Now that we've set up our probability table, let's go ahead and actually use it in order to answer some questions about our system. Here, the one question that we are going to try and answer is, what is the probability that it is raining given that the grass is wet? So you can see here again that we are going from an easily observable effect and trying to attribute it to an underlying cause.

It's easy to observe that the grass is wet. What we are trying to do is now attribute the wetness of the grass to the specific underlying cause of the rain. Now, of course, this is exactly what Bayes' Theorem is all about. But now, we have an additional twist because there are different possible causes for this observable effect. And so we are going to make use of our Bayesian network and we are going to couple that Bayesian network with the repeated use of the chain rule.

We've encountered the chain rule multiple times, but let's restate it here in this context. Given any possible combination of values for the grass, the sprinkler and the rain, we can use the chain rule, as you see on screen now. So on the left-hand side of the equal to sign, we have the joint probability for a specific combination of values for the grass, sprinkler and rain.

On the right-hand side of the equal to sign, we first have the value of the observed state of the grass, given values for the sprinkler and the rain. We multiply that by the conditional probability of the given state of the sprinkler, given the state of the rain. And finally we multiply that by the probability of it's raining. And so when we combine all of these terms, we get the joint probability for any combination that is for any state of the system as a whole.

Now, let's come back to the question that we were trying to answer. That question was, what is the probability that it is raining, given that the grass is wet? And this is expressed by the expression, $P(R = T \mid G = T)$ that is, that rain is equal to true, given that $G = T$ which means that given that the grass is wet as well. So this is what we are trying to find.

We are trying to find the conditional probability that rain is equal to true given that grass wetness is equal to true. And when we apply the formula for conditional probability, we can see that we

have an expression where in the numerator, we have $P(G = T, R = T)$. So this is the joint probability that the grass is wet and that it's raining, divided by the probability that the grass is wet.

Let's focus for now on the numerator of this expression. You can see that we don't have a term in there for the value of S , the sprinkler system. All that we care about is that the grass wetness is equal to true, and that rain is equal to true. But that, in turn, means that this expression is the sum of two different states of the world.

In both states of the world, we have $G = T$ and $R = T$, but in one of the states of the world, $S = T$, and in the other S is equal to false. And we sum these using the sigma operation. So we've now re-expressed the numerator as the sum of two terms. And each of those two terms has explicit values for all of our state variables, the grass, the sprinkler, and the rain.

Let's now do something similar for the denominator. The denominator currently merely has $G = T$. So in the denominator, we have the probability that the grass is wet, but that grass could be wet along with different combinations of the sprinkler system and the rain. And in particular, there are four such combinations with the sprinkler system being set to true or false, and the rain also being set to true or false.

And that's why now the denominator also, is the summation of four different terms. Please note that in the denominator, we are summing together four different probabilities. These four probabilities correspond to S equal to T and F , as well as R equal to T and F . So that gives us four combinations.

In the numerator we are summing together two probabilities, in both of which we have $G = T$ and $R = T$, but in one of which we have $S = T$, and in the other we have $S = F$. So now that we have this expanded expression for the conditional probability that we are interested in, we can now go ahead and refer to our Bayesian network table. We are going to do just that in a moment, but first, let's re-express this calculation as an explicit sum, just because the sigma notation is sometimes a little hard to interpret.

So now in the numerator, for instance, we are adding together two probabilities. These correspond to two different states of the system. The first is $G = T$, $S = T$, and $R = T$, and the second is $G = T$, $S = F$, and $R = T$. Let's hone in on exactly one of these terms and see how we can compute it using our Bayesian network. This is the first probability where $G = T$, $S = T$, $R = T$.

So this is the probability that the grass is wet, the sprinkler is on and that it's raining. And this in turn can be computed using the chain rule. So we first take the probability that the grass is wet, given that the sprinkler is on, and that it's raining. We multiply that by the probability that the sprinkler is on, given that it's raining, and finally, we multiply all of this by the probability that it's raining.

Thankfully, these probabilities are readily available in our Bayesian network and the corresponding tables. The probability that it's raining is available in the table for the rain node. There, we can see that the probability that $R = T$ is 0.2. Then let's come to the table for the sprinkler node. There the probability that the sprinkler is on given that it's raining is equal to 0.01. This corresponds to the probability that $S = T$, given that $R = T$.

And the third probability, which is the probability that the grass is wet, given that the sprinkler is on and that it's raining, is also available from the third node corresponding to the grass. That probability is readily available as well, it's 0.99. And so we plug these numbers into the chain rule. And we get the probability that the grass is wet, given that the sprinkler is on and that it's raining is equal to $0.2 * 0.01 * 0.99$.

When we work out this arithmetic, it works out to 0.00198. This gives us one probability, which corresponds to one term in our overall calculation. So we take a step back and plug this into our overall formula. You can see that the value of the highlighted term is 0.00198. In similar fashion, we can go ahead and compute the other probabilities in this formula on-screen now.

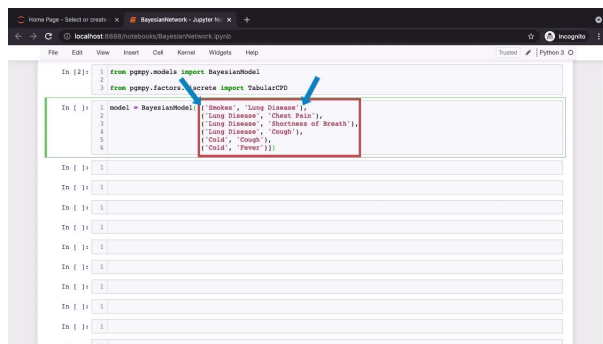
For instance, the probability that the grass is wet, given that the sprinkler system is off, and that it is raining, is equal to 0.1584. And that's the other probability that we have in the numerator. In similar fashion, in the denominator, we have four distinct probabilities. These correspond to the states of the world, which we have represented with the subscripts, TTT, TTF, TFT, and TFF. [Video description begins] *TTT corresponds to 0.00198, TTF corresponds to 0.288, TFT corresponds to 0.1584, and TFF corresponds to 0.0.* [Video description ends]

TTT indicates a state of the world where the grass is wet, the sprinkler is on, and it's raining. TTF is where the grass is wet, the sprinkler is on, but it's not raining. TFT is where the grass is wet, the sprinkler is off and it is raining. And TFF is where the grass is wet, the sprinkler is off, and it's not raining. That last probability is exactly zero.

And when we plug all of these numbers in, we can find the probability that R is equal to true, given that G is equal to true, to be equal to 0.36, that is approximately 36%. And so in this way, we've answered the question that we set out to. Given that the grass is wet, what's the probability that this wetness of the grass was caused by it's raining?

And the answer is about 36%. With this we get to the end of our exploration of the theoretical aspects. We've seen how Bayesian networks are a great tool for attributing causality, where we have an observed effect, and we'd like to see what its possible causes are.

5. Video: Creating a Bayesian Network Model (it_daprthdj_03_enus_05)



Objectives

- *create probability tables for a Bayesian network*

[Video description begins] *Topic title: Creating a Bayesian Network Model. Your host for this session is Vitthal Srinivasan.* [Video description ends]

In this demo, we are going to build a Bayesian network using some pretty cool Python libraries and then use those in order to compute or attribute the probabilities of illnesses given symptoms, which we observe. Let's go ahead and plunge right in. This is a brand new Jupyter notebook. So we start with all of the required import statements.

So we have all of the usual suspects, such as numpy, pandas, matplotlib, and seaborn, and then most importantly, we have pgmpy. This is something that is going to be absolutely key in our construction of this Bayesian model. Next from pgmpy.models, we import BayesianModel, and we also import a construct known as the TabularCPD. CPD stands for Conditional Probability Distribution.

Once we have these available for use, let's define our model. We are going to start by defining all the nodes in the Bayesian network and these nodes fall into three categories. We have one behavior, which is 'Smokes'. Then we have multiple illnesses which are 'Lung Disease' and 'Cold'. And finally we have multiple symptoms which include 'Chest Pain', 'Shortness of Breath', 'Cough' and 'Fever'.

So again, we have one behavior, two illnesses and four symptoms. Now, if you look closely, you'll see that we've actually specified these in the form of tuples, specifically in the form of pairs. And each one of these pairs corresponds to an edge in our Bayesian network. So, for instance, the edge ('Smokes', 'Lung Disease') which is on line 1 of this code cell indicates that smoking causes Lung Disease.

Of course, this causation is probabilistic. So it's not like if somebody smokes, there is a probability of 100% that that person has Lung Disease. But there is a causal relationship between smoking and Lung Disease. On line 2, we have a causal relationship between 'Lung Disease' and 'Chest Pain'. So if you smoke, you may have Lung Disease.

And if you have Lung Disease that might cause Chest Pain. On line 3, we have another causal relationship from 'Lung Disease' to 'Shortness of Breath', and on line 4 from 'Lung Disease' to 'Cough'. Then on lines 5 and 6, we introduce yet another illness, which is 'Cold'. So if an individual has a 'Cold', that might cause a 'Cough', that's on line 5. And if an individual has a 'Cold', that might cause a 'Fever', that's on line 6.

Please also do keep in mind that the basic building block of a Bayesian network is a directed-acyclic graph, which means that every one of these edges is directed. Smoking causes Lung Disease, Lung Disease does not cause smoking. In any case, let's quickly move on and visualize the network diagram for this model. For this, we make use of yet another Python library NetworkX. NetworkX builds on top of Matplotlib.

So we instantiate a figure using `plt.figure`, and then we invoke `nx.draw`. The first input argument is the model and then we pass in additional input arguments such as `nx.circular_layout`. We pass in the model into that. We also specify that we would like labels. We have `node_sizes`, `node_colors`, as well as the `width` and the `font_size`. [Video description begins] *The node_size = 5000, the node_color = 'cyan', the width = 3, and the font_size = 10.* [Video description ends]

And when we hit Shift-Enter, we see that we have a nice directed-acyclic graph encapsulating all of the relationships that we just discussed. We can see that the rightmost node is the Smokes node, and this has just one edge leading out of it. This is a directed edge from Smokes to Lung Disease. There are no edges leading into the Smokes node. So smoking is independent of all of the other factors here.

Smoking causes Lung Disease, that's why there is an edge from Smokes to Lung Disease. Then Lung Disease has edges heading outwards to Chest Pain, Shortness of Breath and Cough. All of those edges are directed and they all point from Lung Disease to the corresponding symptoms. Then down below, we have another independent node, which is Cold. Cold also has no incoming edges into the node.

It has two outgoing edges, one to Cough, and the other to Fever. So we can see that Fever is only caused by Cold. Cough on the other hand, could be caused either by Lung Disease or by Cold. So at this point, we've set up our directed-acyclic graph successfully. The next thing that we've got to do is to create individual tables.

These are the Tabular CPDs, and associate those Tabular CPDs with the nodes in our graph. Let's start with the TabularCPD for the variable 'Smokes'. Remember that this is a variable which is not influenced by any of the other variables in our graph. You can see the syntax for this on-screen now. We've called this `cpd_smokes`, and we instantiate an object of type `TabularCPD`.

The variable is named 'Smokes'. Its cardinality is 2, which means that 'Smokes' is a binary variable. And by the way, all of the variables that we'll be using in this demo are binary. So there are two possible values for the variable 'Smokes', true and false. Then we specify the probabilities of those two values, and those probabilities are specified using the `name` parameter values.

These take the values 0.2 and 0.8. The corresponding `state_names` are 'Smokes', 'Yes' and 'No'. So we can see from this that the probability that someone smokes is 0.2, and the probability that someone does not smoke is 0.8. We hit Shift-Enter, and we see a nice tabular representation.

These give us the absolute probabilities that someone smokes. 0.2 is a probability that someone smokes, 0.8 is a probability that they don't.

And these two probabilities must sum to 1. In similar fashion we create another TabularCPD for the variable 'Cold'. This one is also binary and that's why this one also has variable cardinality of 2. The probabilities are 0.02 and 0.98 for the states 'Cold', 'Yes', and 'No'. So the a priori probability that someone has a Cold is only 2%, and the probability that that person does not have a Cold is 98%.

Smokes and Cold were independent of all other variables in our graph, but that's not true for the remaining nodes. On-screen now is the TabularCPD for the variable 'Lung Disease'. This is also binary. So someone either has Lung Disease or she doesn't. However, this time around there are four possible probability values. And that's because the probability of Lung Disease depends on whether that person smokes or not.

The easiest way to interpret this is just by hitting Shift-Enter, and reading the table. You can see that we have two rows for whether the person has Lung Disease, Yes or No. And then we have two columns for whether the person Smokes, Yes or No. Do notice how this is a table in which every column sums to 1. So in terms of our `pd.crosstab`, this has been normalized column-wise.

For instance, $0.1009 + 0.8991 = 1$. And likewise, $0.001 + 0.999 = 1$ as well. And that key insight tells us how we can read this table. Let's focus on the column Smokes(Yes). This column is telling us that of all individuals who do smoke, there is a 10.09% probability that they have Lung Disease and an 89.91% probability that they don't have Lung Disease. And those two numbers sum up to 100%.

Similarly, amongst those individuals who do not smoke, that is a 0.1% probability that they have Lung Disease, and a 99.9% probability that they don't have Lung Disease. Once again, the two values in this column sum up to 1. Or put differently, these are four conditional probabilities. What is the probability that someone has Lung Disease, given that that person smokes? The answer is 0.1009.

What is the probability that someone does not have Lung Disease, given that they do not smoke? The answer is 99.9%. Now that we understand how the CPD for such a variable is set up, let's create a similar conditional probability density table for 'Shortness of Breath'. You can see this table on-screen now.

Once again, we have two rows for Shortness of Breath, Yes and No, and two columns for Lung Disease, Yes and No. What is the conditional probability that someone has Shortness of Breath given that that person has Lung Disease? The answer is 0.208. That's about 20.8%. What's the probability that someone does not have Shortness of Breath, even though that person has Lung Disease?

And the answer is 0.792 or about 79.2%. So we can see from this table that amongst those who do have Lung Disease, about 20.8% suffer from Shortness of Breath. Amongst those who do not have Lung Disease, only 0.01, that is 1%, suffer from Shortness of Breath. In similar fashion, we move ahead and define yet another TabularCPD. This time for the variable 'Chest Pain'.

By this point, we know pretty well how to interpret the CPD. So let's instead focus our attention on the code that we used to generate it. Notice that the variable that we are interested in here is called 'Chest Pain' and then the causal variable is called 'Lung Disease'. And that is defined on line 5. It's called the evidence.

So on line 5, we have 'Lung Disease', which is the name of another node in our network, and that's the node which has a directed edge leading into this 'Chest Pain' node. We've specified the variable's cardinality, which is 2, that's on line number 2. And then on line 6, we've specified the cardinality of the evidence, which is also 2 because all of these variables are binary.

Then we have `state_names` which correspond to 'Chest Pain', 'Yes' and 'No', and 'Lung Disease', 'Yes' and 'No'. Notice how `state_names` is a dictionary. The keys in this dictionary are the names of our variables, 'Chest Pain' and 'Lung Disease' and the corresponding values take the form of a list with the acceptable values which are 'Yes' and 'No'.

And we can see in the CPD down below that again, we have a conditional probability table. So again, we interpret these values column-wise. What's the probability that somebody has Chest Pain given that that person suffers from Lung Disease? The answer is 0.178. What's the probability that someone does not have Chest Pain given that they do not have Lung Disease?

That probability is 0.95. [Video description begins] *The probability that someone doesn't have Chest Pain given that that person has Lung Disease is 0.822. The probability that someone has Chest Pain given that that person does not have Lung Disease is 0.05.* [Video description ends] So we are getting pretty good at defining our tabular CPDs and interpreting the conditional probabilities that they contain. Let's pick up and continue with this process in the demo coming up ahead.

6. Video: Associating Probabilities with Bayesian Networks (it_daprrthdj_03_enus_06)

```

In [ ]: 1 cpd_cough = TabularCPD(variable = 'Cough',
2                               variable_card = 2,
3                               values = [[0.178, 0.505, 0.505, 0.87],
4                                         [0.178, 0.495, 0.495, 0.93]],
5                               evidence = ['Lung Disease', 'Cold'],
6                               evidence_card = [2, 2],
7                               state_names = {'Cough': ['Yes', 'No'],
8                                              'Lung Disease': ['Yes', 'No'],
9                                              'Cold': ['Yes', 'No']})
10
11 print(cpd_cough)

```

Objectives

- explore the probability tables of nodes in a Bayesian network

[Video description begins] *Topic title: Associating Probabilities with Bayesian Networks. Your host for this session is Vitthal Srinivasan.* [Video description ends]

Bayesian networks are really useful tools for attributing causality when we have multiple different causes which might be responsible for an effect that we observe. In the process of working with Bayesian networks, it's really important to get the conditional probability tables right. And that's why it's worth making sure that we understand this process really thoroughly. So let's keep going and let's set up yet another TabularCPD, this one for the variable 'Fever'.

The cardinality of the variable is 2, the evidence here has just one node, which is 'Cold'. That cardinality is also 2. And then we have the various `state_names` as well as the associated probabilities. Let's hit Shift-Enter and examine these probabilities. We have two rows corresponding to whether this individual has Fever or not, and two columns for whether this person suffers from a Cold or not.

Remember that we've got to interpret these conditional probabilities column-wise. So, for instance, what's the probability that a person has Fever, given that a person has a Cold? The answer is 0.307 or about 30%. What's the probability that a person has no Fever if a person does not have a Cold? Well, the answer is 0.97. [Video description begins] *The probability that a person has a cold, but no fever is 0.693. The probability that a person has fever but not a cold is 0.03.* [Video description ends]

There's one last TabularCPD that we need to set up, and this one is the most interesting. And this is for the variable 'Cough'. The reason that the 'Cough' node is so interesting is because there are two possible causes for 'Cough'. And you can see these reflected in the input argument termed evidence. And that's why on line 5 in the input argument evidence, we have 'Lung Disease' as well as 'Cold'.

Now the evidence cardinality is also in the form of a list [2, 2]. That's because 'Lung Disease' has two possible values and 'Cold' also has two possible values. Likewise, the state_names now is a dictionary with not two but three keys; these are 'Cough', 'Lung Disease', and 'Cold'. Each of these has a corresponding value, which is a list with 'Yes' and 'No'.

Let's go ahead and hit Shift-Enter, and we now see that we have a table with not four but eight different values. Every one of these values is a conditional probability. Let's try and understand some of them. So let's focus, for instance, on the number 0.7525. This is the conditional probability that an individual has a Cough if that individual also has a Cold and suffers from Lung Disease.

Let's try another one. Let's focus on the number 0.93, which is at the bottom right of this table. To find what this is the probability of, let's look at the row, and that row says Cough(No). So this number 0.93 is the conditional probability that an individual does not have a Cough given, given what? Well, now let's look up to the column names. So the column names here are Cold(No) and Lung Disease(No).

So the way to interpret 0.93 is, this is the conditional probability that an individual does not have a Cough, given that that individual does not have a Cold and does not suffer from Lung Disease. We are now done defining all of our CPDs. This was quite time-consuming, but it led us to really understand our data. Next, we've got to associate all of these CPDs with our model.

And the way to do this is to invoke the `add_cpds` method on the model object. And we simply pass in all of our CPD objects one by one. [Video description begins] *The objects are `cpd_smokes`, `cpd_cold`, `cpd_lung_dis`, `cpd_short_breath`, `cpd_chest_pain`, `cpd_cough`, and `cpd_fever`.* [Video description ends] Next, let's invoke the handy `check_model` method. This is going to check the model for the network structure and the CPDs. Ensure that all of the CPDs are correctly defined, all of the required probabilities sum to 1, and that there are no cycles in our directed-acyclic graph.

So when we run this `check_model` method, the returned value is `True`, indicating that our model is well-formed. You should also be aware that we could get the CPDs back, that is, we could query the model object for the CPDs using this getter method, `get_cpds`. As you can see, a list is returned, and every element in that list is a TabularCPD.

In fact, these are the same CPD objects which we instantiated and passed into the model. Next, let's run yet another method called `get_independencies`. This is going to give us all relationships where one variable is independent of other variables. This is a really long and exhaustive list. And that's why some of these don't make a whole lot of sense. But even so, we can perform a few quick sanity checks.

For instance, the first line tells us that Smokes is independent of Fever and Cold. That makes sense because, of course, we do not have any nodes which lead from Fever or Cold into Smokes. And in similar fashion, we can scroll down and try and make sense of every one of these relationships. In any case, the basic idea is that the structure of our graph defines causal relationships.

If node A has a directed edge leading to node B, then node B is not independent of node A. That's the basic idea behind this output. This exhaustive list is probably pretty overwhelming. So we can also make use of the `local_independencies` method in order to specify the name of a variable and get all of the local_independencies for that particular node.

So here, for instance, we've queried `local_independencies` passing in 'Smokes', and we get Smokes is independent of Fever, Cold. And in similar fashion, we can also explore the `local_independencies`

of the other nodes, such as 'Lung Disease', 'Shortness of Breath', and so on. In this context, however, it's also worth paying attention to the output of code cells [17] and [18].

In the output of code cell [17] for instance, you can see that Lung Disease is independent of Fever, Cold. But then is a pipe symbol followed by Smokes. And that pipe symbol tells us that Lung Disease is caused by Smokes. So that pipe symbol is used to indicate that there is a dependency between a pair of variables. The perpendicular symbol is used to indicate that there is no dependency.

At this point, we've finished setting up all of our dependencies and we can now start computing some marginal probabilities. For this, let's first import the VariableElimination class from pgmpy.inference. We then instantiate an object of type VariableElimination and pass in our model. And under the hood, pgmpy will go ahead and perform the calculation of all of those marginal probabilities by working through the directed-acyclic graph.

Once this process is complete, we can now go ahead and query this inference object with the names of the variables that we are interested in. On-screen now, for instance, you can see that we have invoked the method infer.query, and passed in the name of just the one variable 'Cold'. The returned object is then printed out to screen. And when we do, we see that first, there are some calculations.

These are printed out to standard error, that's why they are highlighted in red. And then down below is a table with the marginal probabilities. The probability table which is displayed shows the probabilities of a person having a Cold. Now, if we look at the messages about that, we can see that one by one, pgmpy has gone through and eliminated all of the other variables in our graph.

It has found that none of those variables has a causal relationship with Cold, and that's why the marginal probabilities for Cold are equal to the absolute probabilities. Cold is independent of all of those other variables. These marginal probabilities are described using the word phi. And we can see that the probability of a person having a Cold is 2%, and of not having a Cold is 98%.

In our original graph, we had two disorders, Cold was one of them, Lung Disease was the other. So we now run infer.query passing in 'Lung Disease', and the probability table here has the marginal probabilities for this disorder as well. It's about 97.9% for Lung Disease(No), and it's about 2.1% for Lung Disease(Yes).

Once again, please note that this variable Lung Disease is functionally independent of all other variables except Smokes. In case you're wondering why this table doesn't have any mention of the variable Smokes, do keep in mind that these are the marginal probabilities. So we are getting probabilities for Lung Disease across all values of the other variables.

If we look really closely at the output of the elimination process up above, you can see that the output for the Smokes variable is a little different. It says 100% and 6/6. That's probably some internal working of this algorithm. But even so, smoking clearly does impact Lung Disease. In similar fashion, let's very quickly run through some of the other marginal probabilities as well.

Here is the marginal probability for 'Chest Pain'. You can see 94.73% of our subjects do not have Chest Pain. Let's repeat this experiment for 'Cough'. And we can see that 91.23% of our study subjects do not suffer from Cough either. Where things get really interesting is when we switch from marginal probabilities to joint probabilities.

And you can see on-screen now that we've just run an infer.query with a request for the joint probabilities of 'Cold', 'Cough', and 'Lung Disease'. We've done this by specifying joint = True. We'll examine the output of this command in much more detail in the demo coming up ahead.

7. Video: Computing Probabilities from Bayesian Networks (it_dapnthdj_03_enus_07)

Lung Disease	Cold	Cough	phi(Lung Disease,Cold,Cough)
Lung Disease(Yes)	Cold(Yes)	Cough(Yes)	0.0003
Lung Disease(Yes)	Cold(Yes)	Cough(No)	0.0001
Lung Disease(Yes)	Cold(No)	Cough(Yes)	0.0104
Lung Disease(Yes)	Cold(No)	Cough(No)	0.0102
Lung Disease(No)	Cold(Yes)	Cough(Yes)	0.0099
Lung Disease(No)	Cold(Yes)	Cough(No)	0.0097
Lung Disease(No)	Cold(No)	Cough(Yes)	0.0672
Lung Disease(No)	Cold(No)	Cough(No)	0.8923

Objectives

- *query Bayesian networks to measure probabilities*

[Video description begins] *Topic title: Computing Probabilities from Bayesian Networks. Your host for this session is Vitthal Srinivasan.* [Video description ends]

We ended the previous demo by typing out the command to compute the joint probability of 'Cold', 'Cough', and 'Lung Disease'. We left that command tantalizingly poised. We had not yet hit Shift-Enter. We now go ahead and do that and scroll down to analyze the output. This is an output which has four columns. The first three columns correspond to the names of the variables that we passed in; Lung Disease, Cold and Cough.

The fourth column gives the joint probability for a specific combination of the other three variables. Lung Disease, Cold, and Cough are each binary variables, which means that each of them can take two values, Yes and No. If you look closely, you'll observe that this table has 8 rows, not counting the header. And that's because 2 raised to 3 is equal to 8. So this table is, in a sense, a true table for the three binary variables, Cough, Lung Disease, and Cold.

And then the phi, that is a joint probability, gives us the probability of that specific combination. For instance, let's hone in on the very last row. Cough is equal to No, Lung Disease is equal to No, and Cold is equal to No. For this particular combination of values, the joint probability is 0.8923. So it's pretty clear that the most common case in our data is for individuals who do not suffer from any of these afflictions.

However, if we start looking at some of the other seven cases, some more interesting patterns emerge. Other than that last row of individuals who are completely disease-free, the most common other combination is for Cough equal to Yes, Lung Disease equal to No, and Cold equal to No. [Video description begins] *The probability corresponds to 0.0672.* [Video description ends] There's virtually no one who has Lung Disease as well as a Cold, but does not have a Cough. And we can see that from the row where Cough is equal to No, Lung Disease is equal to Yes, and Cold is equal to Yes.

The probability there is the lowest in this table, 0.0001. Among those who do have a Cough, and exactly one of the other two disorders, the odds are almost equal that you have Lung Disease or a Cold. Why do we say this? Because if we look at the second and third rows in the table where Cough is equal to Yes, we find that for those where Lung Disease is Yes but Cold is No, the probability is 0.0104.

If Cough is equal to Yes, Lung Disease is No, and Cold is Yes, the probability is very similar, 0.0099. This table demonstrates how we can compute joint probabilities using pgmpy. Let's now move to something even more interesting and that is conditional probabilities. On-screen now, we've asked for the conditional probability of 'Cough' given that this person has a 'Cold'.

And the way we do this is by invoking infer.query, with 'Cough' as the only element in the first input argument, and then with a second input document called evidence. This has a key-value pair. So the key is the variable named 'Cold' and the value is 'Yes'. So this is equivalent to requesting the

probability that a person has 'Cough' given that the person has a 'Cold'. And when we hit Shift-Enter, pgmpy goes off and does its calculations.

It works through all of the different paths in the directed-acyclic graph and comes back with a table. And that table gives us two conditional probabilities. The probability that a person does have a Cough given that this person does have a Cold is 0.5102. And the probability that this person does not have a Cough, given that this person does have a Cold, is 0.4898.

This is a very interesting table. It tells us that amongst those people who do have Colds, it's pretty much a toss-up. It's almost 50/50, about half of the folks, or slightly more than half of the folks with a Cold do have a Cough and slightly less than half do not have a Cough. Next, let's go ahead and request another conditional probability. Now, we'd like to get the conditional probability of 'Cough', given that the individual does not have a 'Cold'.

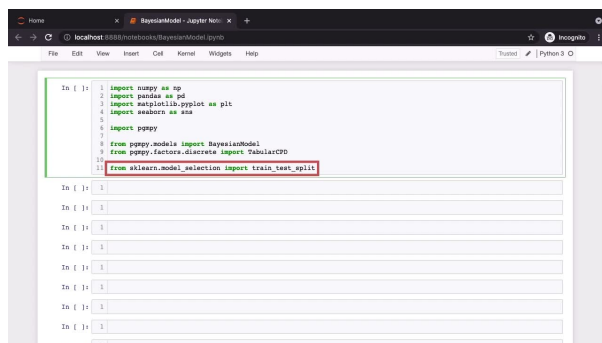
We hit Shift-Enter, and once pgmpy does its calculations, we can examine the output table. That table tells us that the probability of a person having a Cough, even in the absence of a Cold, is 0.0791. And the probability of not having a Cough in the absence of a Cold is 0.9209. So putting together this table and the previous one, if you do have a Cold, there's a slightly better than 50% chance that you'll also have a Cough.

But if you don't have a Cold, there is still a roughly 7.9% chance that you will have a Cough anyway. And it's possible that the driver of that Cough then is not a Cold and rather it's Lung Disease. Let's try and take that logic one step further and let's find the conditional probability of 'Lung Disease', given that an individual has a 'Cough'. And here on-screen are those probabilities.

We can see that amongst those individuals who do have a Cough, 12.19% have Lung Disease, and 87.81% do not have Lung Disease. So this is telling us that even if you do have a Cough, there's still a relatively low chance, only about 12.7% that the cause of that Cough is Lung Disease. And to wrap things up, let's compute the probability that you experience 'Chest Pain' if you are a smoker. And that's what we've now calculated.

The table tells us that if you are a smoker, the probability of experiencing Chest Pain is about 6.29% and of not experiencing Chest Pain, even if you are a smoker, is 93.71%. So this last table tells us that in our dataset, even amongst those individuals who did smoke, only a very small minority did actually experience the symptom of Chest Pain. We've now worked through a fair number of examples using our Bayesian network model. In the demos coming up ahead, we will continue to experiment with other Bayesian techniques.

8. Video: Creating Bayesian Machine Learning Models (it_daprthdj_03_enus_08)



```

In [ ]: 1 import numpy as np
        2 import pandas as pd
        3 import matplotlib.pyplot as plt
        4 import seaborn as sns
        5
        6 import pgmpy
        7
        8 from pgmpy.models import BayesianModel
        9 from pgmpy.factors.noiseless import TabularCPD
        10 from sklearn.model_selection import train_test_split
  
```

Objectives

- *define a Bayesian model in Python*

[Video description begins] *Topic title: Creating Bayesian Machine Learning Models. Your host for this session is Vitthal Srinivasan.* [Video description ends]

Now that we have a good sense of how Bayesian networks are constructed and used, we can take this idea one step further. Once we have a directed-acyclic graph and all of the conditional probability distribution tables, we can easily extend a Bayesian network, and use it for prediction.

That's exactly what we are going to do in this demo where we will be working with a Kaggle dataset, which seeks to predict insurance claims. Let's go ahead and get started. This is a brand new Jupyter notebook. So all of the import statements that we need are on-screen now. To begin with, we have all of the usual suspects, numpy which is imported as np, pandas as pd, matplotlib.pyplot, and seaborn.

Then we import pgmpy, and from within pgmpy.models, we import a new class, that's the BayesianModel. Of course, we also need the TabularCPD class. And then finally from scikit-learn, we import a function called train_test_split. In this demo we'll actually be building a machine learning Bayes model, and train_test_split will help us to split the data available to us into groups which we will use for training the model and then for testing the model to evaluate its accuracy, precision and recall.

With these imports out of the way, let's get the data loaded in. The URL of the Kaggle dataset that we'll be using is visible on-screen now. [Video description begins] *The URL reads <https://www.kaggle.com/easonlai/sample-insurance-claim-prediction-dataset?select=insurance2.csv>.* [Video description ends] And the name of this dataset is Sample Insurance Claim Prediction dataset. The idea of the dataset is that we are given various bits of demographic information, such as the age group, gender, BMI, the number of children, whether an individual smokes or not.

And then using all of these features, we seek to try and predict whether this individual will file an insurance claim. In this particular demo, we've downloaded the dataset from Kaggle, and performed just a little bit of simple preprocessing. Let's see what the preprocessing is. We first read the data in using pd.read_csv. We can see that we've downloaded the data into the datasets folder. [Video description begins] *The folder reads ./datasets/insurance_claim_cat.csv.* [Video description ends]

We've invoked the head command on this data frame, and we can see that we have columns for age, age_group, sex, bmi_group and so on. The little bit of preprocessing that we have performed consisted of adding label-encoded columns for age_group, bmi_group and charges_group. The way we've done this is we've grouped age into three categories. 18 to 35 is classified as Young_Adult, 35 to 60 as Middle_Aged, and above 60 as Elderly.

BMI has also been grouped into four categories. Values below 18.5 are classified as Underweight, from 18.5 to 24.9 as Normal, from 25 to 29.9 as Overweight, and above 30 as Obese. A similar process has been followed with the charges. These have been grouped into four categories. From 0 to 2000 is Low, 2000 to 8000 is Below_Average, 8000 to 16000 as Above_Average, and above 16000 as Very_High.

And finally, we have one-hot encoded, the region variable. That's why we have multiple columns there for northeast, northwest, southeast and southwest. [Video description begins] *The other columns read children, smoker, charges_group, charges, insuranceclaim.* [Video description ends] Now that we've performed a little bit of preprocessing by converting age, charges, and BMI from continuous to categorical, we will drop the continuous columns because we don't really need them in this particular dataset.

So you can see that we've invoked the drop method on our data frame, and specified the columns 'age', 'charges', and 'bmi'. axis = 1 tells pandas that it's columns that we'd like to drop rather than rows. Now, at this point, we still have string representations of the categorical values for age_group, bmi_group, and charges_group. Let's change that for ease of use in our model.

We'll do this by using mappings. So on-screen now we've set up mappings for the age_group. This takes the form of a dictionary with the string values being the keys, 'Young_Adult', 'Middle_Aged',

and 'Elderly'. And the values are the numeric values 0, 1, and 2. In similar fashion, we also set up mappings for the `bmi_group` and the `charges_group`.

These dictionaries will allow us to replace these string representations with the ordinal numeric representations instead. [Video description begins] *In code cell [5], `mapping_bmi_group = {'Underweight': 0, 'Normal': 1, 'Overweight': 2, 'Obese': 3}`. In code cell [6], `mapping_charges_group = {'Low': 0, 'Below_Average': 1, 'Average': 2, 'Very_High': 3}`.* [Video description ends] Once we've set up these dictionaries, we will invoke the `replace` method on the specific column of the pandas data frame that we are interested in.

Here, for instance, we have on the left-hand side of the equal to sign the `'age_group'` column from our `insurance_claim_data`. And then on the right-hand side, we again have that same column, but this time with the `.replace` method invoked on it. And the `replace` method takes in a dictionary. So pandas will look for values in that specific column, match them with keys in the dictionary, and replace those values with the associated value from the dictionary.

In similar fashion, we also replace the string values from the columns `'bmi_group'`, and then from the `'charges_group'`. Once we are done with this, let's reinvoke the `head` command, and examine the contents of our pandas data frame. And we can confirm from this that our replacement operation has gone through successfully.

Before we go a lot further with the model construction, let's quickly get a sense of the shape of our data. We invoke the `shape` property. And as you can see, we have 1338 rows and 11 columns. Let's quickly also confirm what those columns are. And here are all of the column names. [Video description begins] *In code cell [12], the columns are `'age_group'`, `'sex'`, `'bmi_group'`, `'children'`, `'smoker'`, `'charges_group'`, `'insuranceclaim'`, `'northeast'`, `'northwest'`, `'southeast'`, `'southwest'`.* [Video description ends] We are now done with all of the preprocessing and we can instantiate our `BayesianModel` object.

And this line of code that you see on-screen now is pretty important. As you can see, we've specified all of the edges that we would like in our `BayesianModel`. But what's important is that those edges also encode dependencies. So each edge is enclosed between a pair of parentheses, and each edge has a source node and a destination node. For instance, the first test that we've specified here on line 1 leads from `'age_group'` to `'insuranceclaim'`.

So this is our way of telling this `BayesianModel` that the column `'insuranceclaim'` depends on the column `'age_group'`. Or in other words, there is a causal relationship between `'age_group'` and `'insuranceclaim'`. A couple of points worth keeping in mind. Remember that these causal relationships are probabilistic. And secondly, at this point, we don't know how strong any one of these causal relationships is.

That's something that the model is going to have to determine. So when we specify a relationship between `'age_group'` and `'insuranceclaim'`, we are merely indicating to the model that a causal relationship might exist. The strength of that relationship will be inferred when the model is trained. Moving on, you can see that we also have various other causal relationships which we have posited.

These include from `'sex'` to `'insuranceclaim'`, from `'bmi_group'` to `'insuranceclaim'`, and so on. There are, however, three exceptions, and those are on lines 6, 7, and 8, where we have edges which lead from `'smoker'` to `'charges_group'`, from `'bmi_group'` to `'charges_group'` and from `'sex'` to `'charges_group'`. So we are positing in our model that the charges that an individual is charged depend on these three variables as well.

Now, at this point, we've merely instantiated the `BayesianModel` object. We've not actually performed any training or calibration of the CPDs. We can confirm that the CPDs at this point are blank. Let's invoke the `get_cpds` method on our object and the returned value is an empty list. In the demo coming up ahead, we will build on this, we will make use of training data in order to train our model, and then we will see that the CPDs will actually be populated.

9. Video: Predicting Values Using a Bayesian Model (it_daprthdj_03_enus_09)

```

In [13]: model = BayesianModel([
    1: ('age_group', 'insuranceclaim'),
    2: ('sex', 'insuranceclaim'),
    3: ('bmi_group', 'insuranceclaim'),
    4: ('children', 'insuranceclaim'),
    5: ('smoker', 'insuranceclaim'),
    6: ('smoker', 'charges_group'),
    7: ('bmi_group', 'charges_group'),
    8: ('sex', 'charges_group'),
    9: ('charges_group', 'insuranceclaim'),
    10: ('northwest', 'insuranceclaim'),
    11: ('northwest', 'insuranceclaim'),
    12: ('northwest', 'insuranceclaim'),
    13: ('southwest', 'insuranceclaim'),
    14: ])

In [14]: model.get_cpds()
Out[14]: []

In [15]: 
In [16]: 
In [17]: 
In [18]: 
In [19]: 
In [20]: 
In [21]: 

```

Objectives

- *predict values with Bayesian models*

[Video description begins] *Topic title: Predicting Values Using a Bayesian Model. Your host for this session is Vitthal Srinivasan.* [Video description ends]

We ended the last demo, having instantiated our BayesianModel object and specified all of the edges in the Bayesian network. We are going to be using this BayesianModel in order to try and predict whether a particular individual will file an insurance claim or not. And so in this problem, the Y variable, that is what we are attempting to predict, is insurance claim.

And you can see that this 'insuranceclaim' variable is on the right-hand side of most of the edges in our BayesianModel. The word most is important in this context because there are some edges in our BayesianModel which do not include the 'insuranceclaim'. You can see these on lines 6, 7, and 8. These edges go from 'smoker', 'bmi_group' and 'sex' to 'charges_group'.

And these edges are important because they make this a BayesianModel and not a Naive Bayes model. In a BayesianModel, it's permissible for some of the X variables, that is the features which we will be using to predict the Y variable to have dependencies with each other. In a Naive Bayes model on the other hand, it's strictly required that the X variables be independent of each other.

The term Naive in Naive Bayes comes from this assumption. We'll be moving on to a Naive Bayes model in just a bit and there, this difference will become more real. But again, for now, it's important to realize that in this model on-screen now, we do not meet the criteria of a Naive Bayes model because some of the X variables depend on others.

You can see that at this point, the CPDs are blank in our model object. You can see that from the output of model.gets_cpds which has returned an empty list. In order to calculate those CPDs, we are going to perform a training process on our model. And for this we employ the train_test_split function, contained in scikit-learn.

So we pass in the data, that's insurance_claim_data and that's the entire dataset. Then we specify a value for the test_size, this is 0.2, implying that 20% of our data should be reserved for testing, and the remaining 80% can be used for training. We also specify a value of random_state. This ensures that the train_test_split is performed randomly.

However, by specifying this initial value for random_state, we are ensuring that every time we run this code, it's going to be the same 20% that's going to be picked for the testing. That 20% will be picked at random, but using this random_state as a seed. [Video description begins]

The random_state = 123. [Video description ends] train_test_split returns a tuple with two elements, and those two elements are both data frames.

We've called them X_train and X_test. Let's go ahead and invoke the head method on X_train and X_test. And we can see that each of these has the same set of columns. [Video description begins]

The columns are age_group, sex, bmi_group, children, smoker, charges_group, insuranceclaim,

northeast, northwest, southeast, southwest. [Video description ends] Now, one little point worth noting. In these two data frames `X_train` and `X_test`, we retained the `insuranceclaim` column. That's actually the `Y` column, which is what we'd like to predict.

When we use our model for prediction, we are going to have to strip out the `insuranceclaim` column from `X_test`. But for now, while training our model, we pass in the `X` as well as the `Y` values. And one quirk of this `BayesianModel` object, is that when we fit the data, we pass in `X_train` without specifying an explicit value for the response.

In other words, the model does not treat the column to be predicted any differently during training compared to the feature columns. That's why on-screen now you can see that we've invoked the command `model.fit(X_train)`. We did not have to explicitly pass in `(X_train, Y_train)`. In any case, once we hit Shift-Enter, the `BayesianModel` will perform its own internal optimization and come up with the best fitting `BayesianModel`.

And after that, the CPDs will be populated as well. Let's verify that by running `model.get_cpds`. And you can see that this list, which was blank at the start of this demo, now contains many different `TabularCPD` objects. And if we look carefully, we can see that each one of them represents a different column. We can also invoke the `.edges` property on the model object.

We've done this now using a for loop, we've printed out each of the edges, and we can see that we do indeed have edges which lead from each of the `X` variables to the `'insuranceclaim'` column, which is what we'd like to predict. In addition, we also have the three dependencies between `X` variables such as from `'sex'` to `'charges_group'`, that's the third edge, then from `'bmi_group'` to `'charges_group'`, and finally from `'smoker'` to `'charges_group'`.

Again, these dependencies between `X` variables would not be allowed in a Naive Bayes model. So at this point we have a nicely trained `BayesianModel` object and it's time for us to perform some prediction. So the first thing that we are going to do is separate out the column that we are trying to predict. We call this `y_test`, and this is simply the `'insuranceclaim'` column from the `X_test`.

Next, let's also drop that `'insuranceclaim'` column from the `X_test`. So we've invoked the `drop` method on `X_test`, specifying the name of the column and `axis = 1`. We verify using the `head` command that it no longer shows up in `X_test`. And at this point we are ready to go. We can now invoke the `predict` method on our model. We pass in the `X_test`, and what we get back, is the predicted values of the `Y` column.

The model takes a while to run. And indeed, one of the advantages of Naive Bayes over a Bayesian model is the speed of execution. But even so, this is a pretty easy-to-use model. Now that we have the predicted values, we can start using various boilerplate bits of functionality from `scikit-learn` in order to measure accuracy, precision, and recall.

Let's start with the accuracy. On-screen now, we've imported `accuracy_score` from `sklearn.metrics`, and then we've printed out the model's accuracy score using an f-string. An f-string is a formatted string in Python. You can see that we have a nice format there. And this f-string invokes the `accuracy_score`, taking in two input arguments; `y_test` and `y_pred_bayes`.

`y_test` of course corresponds to the real values of the `Y` variables. `y_pred_bayes` are our model predictions. And we can see from the output that this model accuracy score is 77.24%. Now, that's a pretty decent number as far as it goes, but we can't really rely on this number in isolation. We also need to evaluate the model keeping in mind its precision and recall.

So from `sklearn.metrics`, we import two more functions, `precision_score` and `recall_score`. And we invoke those using f-strings, and print out the values of precision and recall to screen as well. The `Precision_score` is 87.6% and the `Recall_score` is 69.73%. Now, these three numbers, the accuracy, precision and recall are important and must be carefully understood.

If we think about the input and output into a classifier using a confusion matrix, then we have true positives, true negatives, false positives and false negatives. Accuracy is measured as the ratio of

true positives + true negatives to all test instances. And this is a measure of how many of the predictions are right. That is, how many predictions are accurate, that's 77.24.

In this context, any user who actually files for an insurance claim is a positive and if the model has correctly forecast this positive, then it's a true positive. Now, the issue with accuracy is that it doesn't tell us specifically how our model did with those who did and did not file insurance claims. That's where the precision and the recall come into play. You can think of precision as the accuracy for when our model has flagged that a claim is going to be filed.

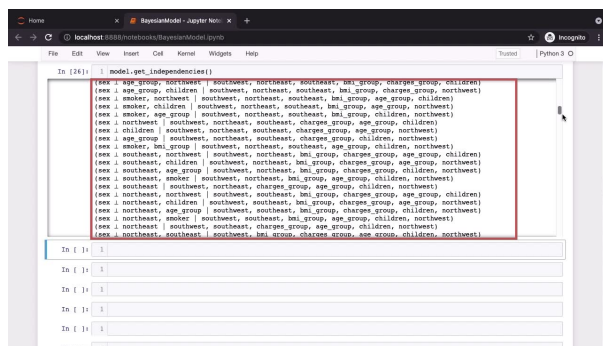
So precision is defined as true positives, divided by true positives plus false positives. The Precision_score of this model is 87.6%, which is higher than the overall accuracy. So that means that our model is pretty discerning. It's pretty careful in flagging that a particular person will file an insurance claim.

Now the Recall_score is telling us that maybe the model is being too careful. You can see that the Recall_score is quite a bit lower than both the accuracy and especially the Precision_score. And the recall can be thought of as the accuracy when an insurance claim is actually filed. So recall is defined as, true positives divided by true positives plus false negatives.

And this proportion is not very good. This is telling us that our model is creating too many false negatives. In other words, there are lots of individuals who do file insurance claims, but these individuals are not being flagged by this model. And that's why this model has a low recall and a high precision. It needs to be a little more aggressive in flagging positives.

In the demo coming up ahead, we will dig a little deeper by exploring the actual probabilities that our model throws up. And then we will go ahead and rebuild a model, a Naive Bayes model to fit the same data and compare the Naive Bayes model with this Bayesian model.

10. Video: Interpreting Probabilities Generated by Bayesian Models (it_dapnthdj_03_enus_10)



Objectives

- *explore probabilities associated with a Bayesian model*

[Video description begins] *Topic title: Interpreting Probabilities Generated by Bayesian Models. Your host for this session is Vitthal Srinivasan.* [Video description ends]

We ended the previous demo by observing that our model had an accuracy score of 77.24% and a higher Precision_score, but a much lower Recall_score. And we had hypothesized that this lower recall was happening because our model was too timid in predicting that an individual was going to file an insurance claim. Let's see whether the evidence bears this out.

Now, the Bayesian model that we are working with here is very similar to the Bayesian network that we had been making use of in the previous demos. And so a lot of this code is going to seem

pretty familiar. Let's start by running `model.get_independencies`. And you can see from the form of the output that we have a very long list of independencies, and indeed of dependencies.

This list is a little intimidating. So let's instead focus on the dependencies of specific variables. We'll do this by invoking the method `model.local_independencies` and specifying the name of a column which is 'smoker'. And the output confirms that smoker is independent of the geographic region, the `bmi_group`, `age_group`, `child` and `sex`. Let's perform a similar operation on the column 'age_group' and we get a similar response.

Please note the symbol which appears right after the column name, for instance, right after the word 'age_group' in the output of code cell [28]. That symbol, which is often used in geometry to represent lines that are perpendicular to each other, in the context of probability, refers to variables which are independent of each other. So `age_group` is conditionally independent of `southwest`, `smoker`, and so on.

There's no great surprise here. After all, while specifying the structure of our directed-acyclic graph, we had not linked up any dependency between these columns. In any case, let's now move on and start performing some inference. For this we import the `VariableElimination` class from `pgmpy.inference` and instantiate it passing in our model and then we query the inference object specifying the column name 'insuranceclaim'.

And then when we print all of this out, we get the absolute or the a priori probabilities that an insurance claim was filed. So this tells us that in our dataset, 46.07% of the rows did not lead to an insurance claim being filed, and 53.93% did lead to an insurance claim. So the marginal probability of an insurance claim being filed across all values of the other variables is 53.93%.

So these marginal probabilities tell us that our dataset is pretty balanced in terms of how many rows do or do not lead to insurance claims. Next, let's move on and compute some conditional probabilities. On-screen now we are trying to find the conditional probability of an 'insuranceclaim', given that this individual is not a 'smoker', and that this individual is overweight; 'smoker' = 0, and 'bmi_group' = 2.

When we hit Shift-Enter, we find that these conditional probabilities are pretty close to 50/50. So amongst individuals who are non-smokers, but are overweight, the probability of filing an insurance claim is only about 50.59%. Please remember again that this is a conditional probability. So this number, 50.59% tells us that given that an individual is a non-smoker, who is overweight, the probability of his or her filing an insurance claim is only 50.59%.

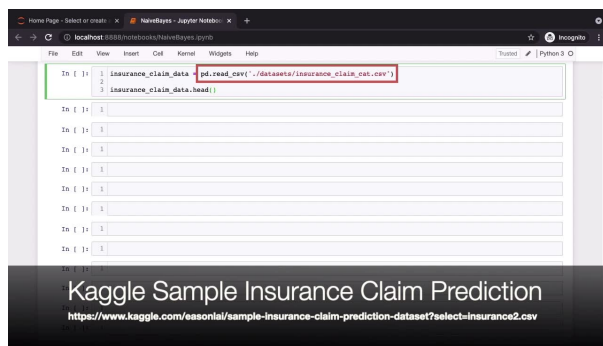
And in fact, this conditional probability is less than the marginal probability of an insurance claim, which a moment ago we saw was just a little north of 53%. [Video description begins] *The other probability reads 0.4941.* [Video description ends] Before we move any further, please also keep in mind that these probabilities have been inferred from the training data.

So there's nothing in the test data that's being taken into consideration for these numbers that you see on-screen now. Next, let's try with another combination. So we are now asking for the conditional probability that an 'insuranceclaim' is filed given that this person is a 'smoker', and given that this person is overweight as well. So 'bmi_group' equal to 2 indicates overweight.

The conditional probability of a claim being filed by an overweight smoker is only 58.55%. And the reason I use the word only is because if you remember the prior or marginal probability of an insurance claim being filed by everyone in the population as a whole stood north of 53%. [Video description begins] *The other probability reads 0.4145.* [Video description ends]

So this result on-screen now is beginning to tell us that our feeling may have been correct and our model might be a little too timid in predicting that someone is going to file an insurance claim. And in order to see how this output might change if we tweak some of the model assumptions, in the demo coming up ahead, we are going to construct a Naive Bayes model and rerun these probability calculations.

11. Video: Understanding and Creating Naive Bayes Models (it_daprthdj_03_enus_11)



Objectives

- *create naive Bayes models in Python*

[Video description begins] *Topic title: Understanding and Creating Naive Bayes Models. Your host for this session is Vitthal Srinivasan.* [Video description ends]

In this demo, we are going to continue working with Bayesian models, and the use of the pgmpy library. However we are going to turn our attention from the Bayesian model itself to a specific subclass, which is Naive Bayes. The key difference between Naive Bayes and the Bayesian model is that in Naive Bayes, we assume that all of the X variables are independent of each other.

In terms of the Bayesian network graph, this means that with Naive Bayes, the only edges in the model will be from the X variables directly to the Y variable. That is, all of the edges will go from independent variables to the one dependent variable. This is a little different than the structure we had with the Bayesian model. In any case, let's go ahead and get started.

We start with all of the import statements. Nothing particularly new here. And then we read in the data. [Video description begins] *Line 1 reads import numpy as np. Line 2 reads import pandas as pd. Line 3 reads import matplotlib.pyplot as plt. Line 4 reads import seaborn as sns. Line 6 reads import pgmpy.* [Video description ends]

This is the same dataset that we were making use of in the previous demo. It's a Kaggle Sample Insurance Claim dataset. The URL is visible on-screen now, but as in the previous demo, we've downloaded this data, and we've now read it in using pd.read_csv. [Video description begins] *The URL reads https://www.kaggle.com/easonlai/sample-insurance-claim-prediction-dataset?select=insurance2.csv.* [Video description ends]

We then invoke the head method in order to sample the first five rows. And here they are on screen now. We have a lot of the same preprocessing as in the previous demo, so we won't talk about that in detail again. Let's go ahead and drop some of the columns that we don't require, such as 'age', 'charges', and 'bmi'. We'll do this by invoking the drop method on our insurance_claim_data pandas data frame.

We don't need these columns because we already have the label-encoded versions. We can also verify that these drop operations have gone through successfully by examining the columns of the resulting data frame. [Video description begins] *In code cell [4], the columns are age_group, sex, bmi_group, children, smoker, charges_group, insuranceclaim, northeast, northwest, southeast, southwest.* [Video description ends] This looks good. Now let's move on and replace these string representations with ordinal values.

We'll do this in the same fashion that we did in the previous demo. We set up dictionaries in which the keys correspond to the strings, such as 'Young_Adult', 'Middle_Aged', and 'Elderly', and the values correspond to the ordinal values such as 0, 1, and 2, that we would like to use instead. On-screen is the dictionary for the mapping_age_group.

Let's also set up similar dictionaries for the other two columns that we'd like to map. These are the bmi and the charges columns. [Video description begins] *In code cell [6], mapping_bmi_group = {'Underweight': 0, 'Normal': 1, 'Overweight': 2, 'Obese': 3}. In code cell [7], mapping_charges_group = {'Low': 0, 'Below_Average': 1, 'Average': 2, 'Very_High': 3}.* [Video description ends]

Once we have the dictionary set up, let's go ahead and invoke the .replace method on our insurance_claim_data. We've got to invoke this method thrice, once for each column where we would like the mapping applied. And of course, the way the replace method works, it takes a dictionary, it finds values in the specified column, matches those values with keys in the dictionary, and replaces those values in the data frame with the corresponding values from the dictionary.

Let's rerun the head command to make sure that these mappings have been applied successfully. And when we do, we can see that these columns for bmi_group, age_group, and charges_group now contain ordinal values rather than strings. That does it for the data preprocessing.

We can move on now to the model building phase. To begin with, let's import train_test_split from sklearn.model_selection and invoke this function in order to split our data into two subsets, X_train and X_test. As in the previous demo, we have test_size = 0.2 and random_state = 123. So this is going to give us two pandas data frames, called X_train and X_test.

Next, let's instantiate our Naive Bayes model object. And here we import NaiveBayes from pgmpy.models, and then we instantiate it passing in two named parameters; feature_vars and dependent_var. It's interesting to note that when we instantiate a Naive Bayes model, we do not specify a list of edges. You might recall that in the case of the Bayesian model, as well as in the case of the Bayesian network, we had to specify edges.

These were tuples or pairs in which the first element was the source node and the second element was the target node. However, we don't do this in the case of the Naive Bayes case, and that's because Naive Bayes assumes that the only dependencies are from all of the feature variables to the single dependent variable.

Here there's just one dependent variable 'insuranceclaim', and feature variables is a list which has all of the features that we are going to try and use to predict whether a claim was filed or not. So we instantiate the NaiveBayes object on line 3, and then on line 7, we perform our training step. Once again, we just invoke model.fit, passing in X_train.

X_train is a pandas data frame and the Naive Bayes model is going to look for all of the columns that we've specified while instantiating the model. It will look up those columns by name. And it's also smart enough to figure out which of those are feature columns, and which is the dependent variable.

Now the Naive Bayes class inherits from the Bayesian model class, and that means that pretty much all of the functionality that we accessed in the Bayesian model example will also be available here. For instance, on-screen now we've invoked the model.get_cpds method. Because we are invoking this method after having performed our training step by invoking the fit method, the list of CPDs comes back fully populated.

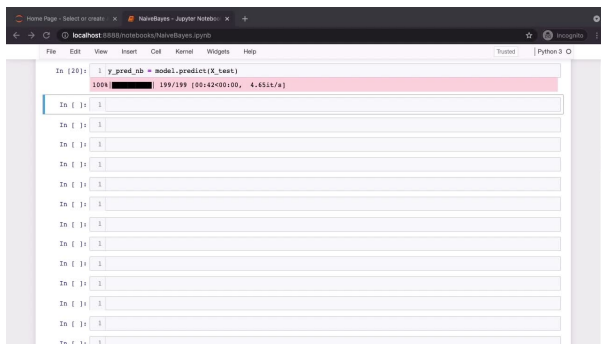
If we look closely at the output of code cell [14], we see that there are TabularCPDs for every one of the feature variables as well as for the dependent variable insuranceclaim. Next, let's print out all of the edges in our model. The code for this is the same. We iterate over the returned value from model.edges.

The loop variable is called edge, and we simply invoke print on each edge. And from the output we can see that this Naive Bayes model is also represented by a directed-acyclic graph. In this graph, every one of the edges leads to the dependent variable from the independent variables. We can also verify this property of the Naive Bayes model by checking the local_independencies of some of the X and the Y variables.

Here on screen, we've invoked the `.local_independencies` method on the model, specifying one of the X variables 'age_group'. And we can see from the returned value that age_group is functionally independent of all of the other X variables. However, it predicts or has a causal relationship with the insuranceclaim variable. Notice the use of the pipe symbol, that's the last character on the first row of the output.

So the only causal relationship involving age_group is between age_group and insuranceclaim. Let's now reinvoke the `.local_independencies` method on the dependent variable 'insuranceclaim'. And this time when we hit Shift-Enter, the output is blank. So there are no variables which depend on insuranceclaim. However, insuranceclaim depends on all of the other X variables. We are now ready to move from the training step to the inference step. We'll get to that in the demo coming up ahead.

12. Video: Testing Naive Bayes Machine Learning Models (it_daprthdj_03_enus_12)



Objectives

- *predict values with naive Bayes models*

[Video description begins] *Topic title: Testing Naive Bayes Machine Learning Models. Your host for this session is Vitthal Srinivasan.* [Video description ends]

In the previous demo, we instantiated our Naive Bayes model, performed the training step and got it ready for use in inference. In this demo, we'll actually use the Naive Bayes model and we'll contrast its performance characteristics with its close cousin, in fact, its ancestor, the Bayesian model. So let's get started with the inference part of this demo.

We start by carving out the 'insuranceclaim' column and placing it in a separate data frame called `y_test`. Remember that the insuranceclaim is what we are trying to predict. Once we set up our `y_test` variable, we also drop 'insuranceclaim' from the `X_test` just to make sure that there's no chance of it's being used inadvertently by the model. [Video description begins] *In code cell [19], the axis = 1.* [Video description ends] Then we'll actually invoke the predict method on our model.

Here, the model is a Naive Bayes model. And that's why we save the predictions in a variable called `y_pred_nb`. The prediction phase with a Naive Bayes model is noticeably faster than with a Bayesian model. And that, of course, is because the model itself is easier to evaluate.

In a Bayesian model there are dependencies between X variables which do not exist in the case of a Naive Bayes model. In any case, our model quickly runs through to completion and then we import the `accuracy_score` function from `sklearn.metrics`, and print out the model accuracy using a Python f-string. Note that we'll do this by invoking the `accuracy_score` method, passing in `y_test` as well as `y_pred_nb`.

We hit Shift-Enter, and this tells us that our Model accuracy score is 77.99% So this is slightly better than the Bayesian model, but not a whole lot better. You might recall that the accuracy of

the Bayesian model was 76%. Accuracy has to do with all correct predictions. You might recall that the definition of accuracy was true positives plus true negatives divided by all predictions.

So at least the Naive Bayes model hasn't done any worse than the Bayesian model. Let's now try and measure the precision and the recall. So we import the functions `precision_score`, and `recall_score` from `sklearn.metrics`, and then invoke those two functions once again, passing in `y_test` and `y_pred_nb`.

And when we hit Shift-Enter and we find that the precision of this model is around 80.7%. So this is noticeably lower than the precision of the Bayesian model. But we have a pleasant surprise in store for us when we examine the recall. The `Recall_score` of this Naive Bayes model is also at around 80.26%. So when we look at the accuracy, precision and recall, we see that the Naive Bayes model has roughly comparable accuracy, noticeably lower precision, but a lot higher recall.

And this mitigates a key weakness of the Bayesian model. Let's quickly recap how precision and recall are defined. Precision can be thought of as the accuracy in those cases where our model predicts that a claim is going to be filed. So precision is defined as true positives divided by true positives plus false positives. The precision score of the Naive Bayes model is lower than the precision score of the Bayesian model.

This means that the Naive Bayes model is flagging more false positives than the Bayesian model. And this in turn indicates that our Naive Bayes model is being more aggressive than the Bayesian model in predicting that a claim will be filed. But it's the same tendency which is hurting the precision that is boosting the recall. The recall is defined as true positives divided by true positives plus false negatives.

And you can think of the recall as the accuracy for those cases where a claim is actually filed in our test data. The `Recall_score` of the Naive Bayes model is 80.26%. It's significantly higher than the recall of the Bayesian model we had in the previous demo. And that means that this Naive Bayes model is flagging significantly fewer false negatives. This should come as no surprise because if the precision comes down, we will expect the recall to go up.

Let's dig deeper into the probabilities produced by our Naive Bayes model, and let's see how these compare to those we had with the Bayesian model. So let's import `VariableElimination` from `pgmpy.inference`, instantiate that `VariableElimination` object. This gives us an inference object which we can then query.

When we hit Shift-Enter, this gives us the marginal probabilities of the insuranceclaim being filed in this particular training dataset. And you can see here that those marginal probabilities are somewhat skewed in favor of an insuranceclaim being filed. The marginal probability of a claim being filed stands at 58.97%. The marginal probability of a claim not being filed stands at 41.03%.

These are the priors or the ex-ante probabilities without having any further information about any of the features. Next, let's move on to some of the conditional probabilities. So on-screen now, we are interested in the conditional probability that an 'insuranceclaim' will be filed, given the evidence that this individual is not a 'smoker' and that this individual is overweight. So we've specified 'smoker' equal to 0 and 'bmi_group' equal to 2.

We hit Shift-Enter, the Naive Bayes model does its calculations and comes back with probabilities that are pretty close to 50/50. We can see here that the conditional probability of a claim not being filed given that this individual is not a smoker and that this individual is overweight stands at 53.11. And the probability of a claim being filed by such an individual, given that this individual is not a smoker and that this individual is overweight, is 46.89%.

So far, there's nothing very different about these probabilities. However, we'll get a different picture, once we start investigating folks who are smokers. On-screen now we are running the `infer.query` command in order to get the probability of an 'insuranceclaim' being filed, given the evidence that this individual is a 'smoker' and that this individual is overweight. So just intuitively, this ought to be a pretty at-risk individual.

Let's hit Shift-Enter and find out what our model thinks. And indeed, there is a very marked difference in this output relative to the output of the Bayesian model. The Naive Bayes model thinks that for an individual who is a smoker and is overweight, the probability of filing a claim stands at a staggering 88.56%. And the probability of not filing a claim, given the evidence that this person is a smoker and that this person is overweight is just about 11.5%.

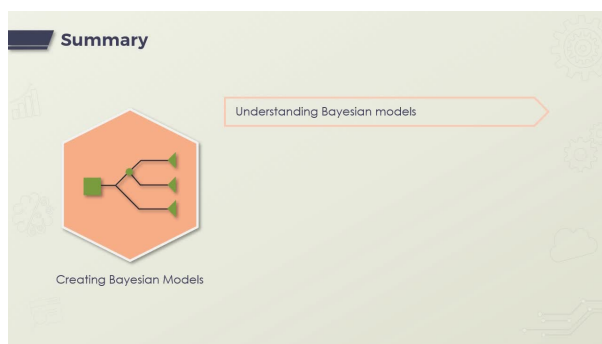
This is a really sharp distinction. And we can see that this is where the Naive Bayes model is significantly bolder than the Bayesian model in predicting that a claim will be filed. So this confirms our intuition that the Bayesian model was a little too timid, the Naive Bayes model is a lot bolder, and that shows up in the higher recall score of the Naive Bayes model.

Let's check out one last conditional probability. Here, we are querying for the conditional probability that a claim is filed given that this individual is a 'smoker' and is elderly. So the 'age_group' is equal to 2, and 'smoker' is equal to 1. We hit Shift-Enter, and in the light of this evidence, Naive Bayes seems to think that there is a very high probability that a claim will be filed.

The conditional probability that a claim is filed stands at about 94%. This is further confirmation of the fact that the Naive Bayes model is more willing to bet big when it sees a favorable combination of factors. [Video description begins] *The other probability reads 0.0571.* [Video description ends] Naive Bayes models are extremely powerful, and their name indeed is something of a misnomer. The term Naive and Naive Bayes merely refers to their assumption that all of the X variables are independent of each other.

This assumption sounds a lot worse than it is in practice. It turns out that Naive Bayes models perform really well in a variety of machine learning use cases, and indeed we've had some evidence of that in our own experiments in this demo and the previous one. That gets us to the end of our exploration of Naive Bayes models, and indeed of Bayesian models in general.

13. Video: Course Summary (it_daprthdj_03_enus_13)



Objectives

- *summarize the key concepts covered in this course*

[Video description begins] *Topic title: Course Summary.* [Video description ends]

We have now come to the end of this course, Creating Bayesian Models. We started by stating and interpreting Bayes' Theorem. This describes the probability of an event based on prior knowledge about conditions related to the event. We worked through an example in which we found the probability of an event with Bayes' Theorem. We then defined and used Bayesian networks.

These are networks which represent probabilistic relationships between multiple events. We also applied the chain rule to Bayesian networks. Next, we used Python to create a Bayesian network and specified the required probabilities. We explored the relationships and dependencies in our graph. We also viewed the probability of an event occurring given multiple combinations of other events.

After that, we calculated several complex conditional probabilities using a Bayesian machine learning model. We used this model to fill in the conditional probability tables for each relationship. We observed the accuracy, precision and recall scores of our model and noted that our model seemed to miss many true positives. We confirmed this intuition by viewing the probability tables.

Finally, we defined and used Naive Bayes models, which are a category of Bayesian models where we assume that the explanatory variables are all independent of each other. This assumption is the reason for the term Naive in the name. Despite the somewhat disparaging name, such models are extremely effective. For instance, we noted that the Naive Bayes model was rightly more aggressive in flagging positives than the regular Bayesian model.

This brings us to the end of this learning path. You now have a solid understanding of probability theory. You are ready to move on to Probability Distributions coming up ahead.

Course File-based Resources

- [Creating a Bayesian Network Model](#)
Topic Asset
- [Associating Probabilities with Bayesian Networks](#)
Topic Asset
- [Computing Probabilities from Bayesian Networks](#)
Topic Asset
- [Creating Bayesian Machine Learning Models](#)
Topic Asset
- [Predicting Values Using a Bayesian Model](#)
Topic Asset
- [Interpreting Probabilities Generated by Bayesian Models](#)
Topic Asset
- [Understanding and Creating Naive Bayes Models](#)
Topic Asset
- [Testing Naive Bayes Machine Learning Models](#)
Topic Asset

© 2022 Skillsoft Ireland Limited - All rights reserved.