16. Bayes Nets

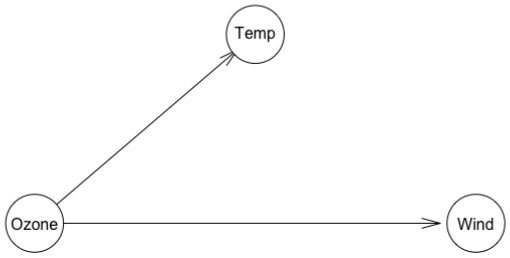

Figure 16.1: Airquality Bayesian Network

Bayesian networks, sometimes called belief networks, are unique in that they can be customized to encode not only Bayesian probabilities but also expert human knowledge. In turn, the network itself is highly interpretable. Figure 16.1 shows that Ozone is conditioned on Temp and Wind. Ozone is the parent node, Temp and Wind are child nodes. The absence of links can convey information as well. Notice there is no link between Temp and Wind. In a Bayes' network there are no target or predictor variables. Each node represents a random variable and arrows represent their probabilistic dependencies.

16.1 Bayes Nets in R

A Bayesian network is a directed acyclic graph as seen in Figure 16.1. The network was created with the code below. The bn.fit() function used below needs all the variables to be factors so first we subset the airquality data, removed rows with NAs, and converted them to binary factors with 1 meaning high and 0 meaning low. The text below the code example shows the conditional probabilities of the fit model.

Code 16.1.1 — Bayesian Network. Airquality data.

```
library(bnlearn)
f <- function(v){
  m <- mean(v)
  factor(ifelse(v>m,1,0))
}
df <- airquality[,c(1,3,4)]  # Ozone, Wind, Temp
df <- df[complete.cases(df),]  # omit rows with NAs
df$Ozone <- f(df$Ozone)
df$Wind <- f(df$Wind)
df$Temp <- f(df$Temp)
# build the net
bn1 <- hc(df)
plot(bn1)
# find the conditional probabilities
fit_air <- bn.fit(bn1, data=df)
fit_air
```

```
  Bayesian network parameters
  Parameters of node Ozone (multinomial distribution)
Conditional probability table:
         0         1
0.6206897 0.3793103

  Parameters of node Wind (multinomial distribution)
Conditional probability table:

    Ozone
Wind         0         1
   0 0.3611111 0.8181818
   1 0.6388889 0.1818182

  Parameters of node Temp (multinomial distribution)
Conditional probability table:

    Ozone
Temp         0         1
   0 0.7222222 0.0000000
   1 0.2777778 1.0000000
```

16.1.1 Querying the Network

Once we have the conditional probabilities determined by the bn.fit() function, we can query the net. Two queries run at the console and the results are shown below.

```
> cpquery(fit_air, event=(Ozone==1), evidence=(Temp==1))
[1] 0.6833689
> cpquery(fit_air, event=(Temp==1), evidence=(Ozone==1))
[1] 1
```

The cpquery() function performs conditional probability queries on the network. Specifically, it estimates the conditional probability of an event given evidence, and returns this probability. In the first query above we wanted to know P(Ozone=1|Temp=1), in other words, what is the probability that Ozone is high given that Temp is high. The value was 0.68. The second query asked for P(Temp=1|Ozone=1) which was 1.0.

16.2 Bayesian Net Semantics

A Bayesian network is a directed acyclic graph (DAG). In order to discuss properties of the network we first review some terminology from graph theory.

16.2.1 Review of Graph Terminology

A graph is defined by $G = (V,A)$ where V is the set of nodes or vertices and A is the set if arcs, links, or edges that connect the notes. An arc $a = (u,v)$ connects nodes u and v. If the connection is undirected, the order of the vertices does not matter. If the connection (u, v) is directed, the head will be u and the tail will be v. In Bayesian networks, all of the arcs will be directed. In a directed graph, if there is a path from v_i to v_j then v_i is an ancestor of v_j, which is a descendant. The direct ancestor is called a parent. The direct descendant is a child. Graphs can be either cyclic or acylic. Bayesian networks are acyclic, they can have to cycles.

In Figure 16.2 we have a directed, acyclic graph, that provides an overly simple model of diabetes. We have 5 random variables: (1) F - a Yes/No variable for family history of diabetes, (2) U - a Yes/No variable for an unhealthy lifestyle, (3) D - Yes/No for a diagnosis of diabetes, (4) S - Yes/No for elevated blood sugar, and (5) T - Yes/No for excessive thirst.

16.2.2 Graph Structure

If two variables are independent, there will be no arc connecting them. Conditional independence is specified by the directed separation criterion (d-separation). The intuition behind d-separation can be seen in Figure 16.2. There is no direct path from F to S that does not go through D. D is said to d-separate F and S. Therefore, F is independent of S|D. This is denoted by: $F \perp\!\!\!\perp S|D$.

Directly following from the idea of d-separation is the Markov property of Bayesian networks. The Markov property for a Bayes Net states that there are no direct dependencies in the graph that are not shown explicitly. For example, the unhealthy lifestyle cannot affect frequent thirst except through diabetes. Again, keep in mind this is an oversimplified example and not necessarily consistent with medical research.

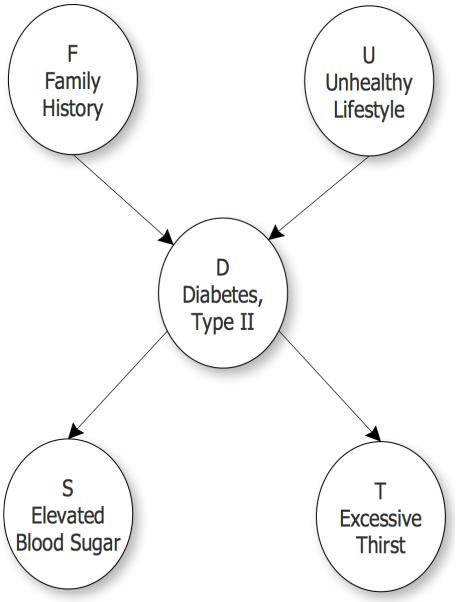

Figure 16.2: Bayesian Network

The structure of a Bayesian network implies that the value of a variable is conditioned only on its parent nodes:

$$P(x_1, x_2, ..., x_n) = \prod_i P((x_i|Parents(X_i)) \tag{16.1}$$

In the case of the sample diabetes graph, P(Thirst|Diabetes, Unhealthy) = P(Thirst|Diabetes). This indicates a conditional independence of variables.

16.2.3 Reasoning with Graphs

The DAG of the network factorizes the global probability distribution into a local probability distribution for each variable. The connections provide means of reasoning about the variables. Figure 16.2 illustrates four types of reasoning we can do with Bayesian networks:

- diagnostic - given the evidence of excessive thirst, the cause is likely diabetes; notice this traces the graph in a bottom-to-top direction
- predictive - given diabetes, it is likely that a person experiences excessive thirst; notice this traces the graph in a top-to-bottom direction
- intercausal - given a family history leading to diabetes, this may explain away an unhealthy lifestyle as a factor
- combined - given a family history leading to diabetes, this in turn predicts excessive thirst

16.3 The Algorithm

The algorithm used in the code sample above, hc(), is a hill climbing algorithm. Hill climbing has widespread application in AI. In simple hill climbing for Bayesian networks,

we start with an empty graph. Each variable in the data is evaluated by a score function that quantifies how well the network with the added node would fit the data. The search through the variables is greedy, adding the variables based on the highest score. Metrics vary, including a posteriori probability, or Bayes Information Criterion (BIC). There are many variations of hill climbing in AI and machine learning.

Another algorithm available in bnlearn() is TABU search, a constraint-based greedy search. The tabu search performs hill climbing until it finds a local optimum. It then searches through the next best variables that it has not visited recently, i.e., that are not on the tabu list.

16.4 Example: Coronary Data

As another example of a Bayes' net, we look at the coronary data set in R. The data specifies risk factors for coronary thrombosis for men. The data set has 1841 observations and 6 variables, all of which are binary factors:

- Smoking - yes or no
- M. Work - yes or no for strenuous mental work
- P. Work - yes or no for strenuous physical work
- Pressure - <140 or >140 systolic blood pressure
- Proteins - <3 or >3 ratio of beta and alpha lipoproteins
- Family - neg or pos for patient's indication of family history

Code 16.4.1 — Bayesian Network. Coronary Data.

```
library(bnlearn)
bn_coronary <- hc(coronary)
plot(bn_coronary)
```

Figure 16.3 shows the network for the coronary data. There are some links that make sense, such as Proteins being affected by smoking and mentally stressful work. There are other links that don't seem to make sense such as: Family being a child of mentally challenging work. We can list all links with "bn_coronary$arcs" at the console, and we see that the link we want to delete is the 7th one. We can remove this link as shown below.

```
bn_coronary$arcs <- bn_coronary$arcs[-c(7),]
```

Now if we replot, we see that the family node is still there but is not connected to any other node. Now we are ready to do some conditional probability queries.

```
> cpquery(fittedbn, event = (Pressure==">140"),
    evidence = ( Proteins=="<3" ) )
[1] 0.4310942
```

16.5 Summary

The term Bayesian Network was coined by Judea Pearl in 1985 to emphasize the Bayesian conditioning and the method for updating information. Judea Pearl was the 2011 winner of the ACM Turing Award.

Bayesian networks provide a graphical and highly interpretable representation of conditional probabilities in a data set. The examples in this chapter were for qualitative

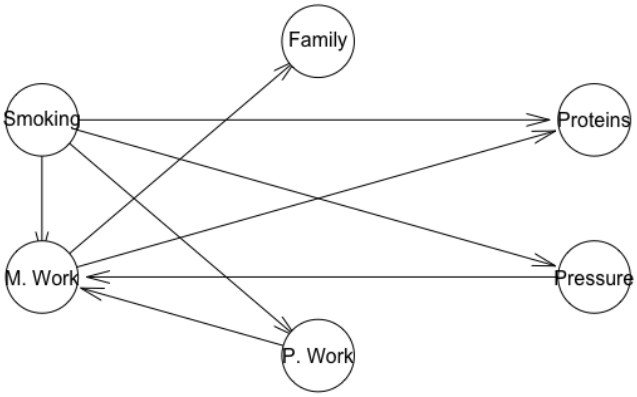

Figure 16.3: Bayesian Network for Coronary

data but the bnlearn package can also handle quantitative data with a custom.fit() function rather than the bn.fit() function. Another package for Bayesian networks in R is DEAL, which provides several methods for using discrete or continuous variables.

16.5.1 Going Further

- Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Judea Pearl. Morgan Kaufmann. 1988.
- Bayesian Networks with Examples in R. M. Scutari and J.B. Denis. Chapman & HallCRC. 2014.
- Bayesian Networks in R with Applications in System Biology. Nagarajan, M. Scutari and S. Lebre. Springer. 2013.
- Article by M. Scutari, the author of bnlearn: `https://arxiv.org/pdf/0908.3817.pdf`

murphy:

17. Markov Models

17.1 Overview

A hidden Markov Model, or HMM, is a probabilistic model often used to model sequential events such as temporal patterns, predictive text, part-of-speech tagging, and much more. First we discuss the simpler case of a Markov Model where we can see all the states, then move on to discuss the hidden Markov Model for unseen states which have observable results.

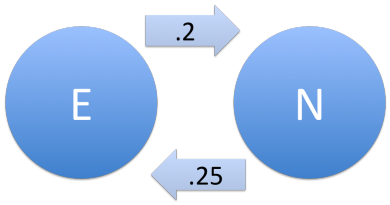

Figure 17.1: Markov Model for Two States: Exercise or Not

17.2 Markov Model

A Markov Model is a model for stochastic (random) processes. The model consists of a set of states and transition probabilities between states. The Markov assumption is that the probabilities for transition depend only on the previous state, not the entire string of preceeding states. Let's consider a simple system with 2 states modeling the likelihood that a person will exercise today, and is based on the observation a person made that if they exercised the day before they tend to exercise today, with .8 probability, but if something interferred with exercising the day before they tend to not exercise today either, with .75

probability. State E represents the exercising state and state N represents the not exercising state.

As we see in the Figure above, a person in state E stays in state E with .8 probability and moves to state N with .2 probability. Likewise a person in state N stays in state N with probability .75 and moves to state E with probability .25. Below we see some R code representing the initial state of having exercised 5 of 30 days, and the transition matrix.

Code 17.2.1 — Initial State. Representing Exercise or Not.

```
# build the transition matrix for the model
transMatrix <- matrix(c(.8, .25, .2, .75), nrow=2)
transMatrix  # output matrix
     [,1] [,2]
[1,] 0.80 0.20
[2,] 0.25 0.75
# represent the initial state in the exercise matrix
exercise <- matrix(c(5/30, 25/30), nrow=2)
exercise
          [,1]
[1,] 0.1666667
[2,] 0.8333333
```

Initially, the person had a poor exercise pattern, exercising 5 days out of 30, about 17% of the time. What happens after 6 iterations?

Code 17.2.2 — Markov Process. Six iterations.

```
for (i in 1:6){
  exercise <- transMatrix %*% exercise
  print(paste("exercise at i=", i, ":",
        format(round(exercise[1,], 2))))
}
exercise
[1] "exercise at i= 1 : 0.34"
[1] "exercise at i= 2 : 0.44"
[1] "exercise at i= 3 : 0.49"
[1] "exercise at i= 4 : 0.52"
[1] "exercise at i= 5 : 0.54"
[1] "exercise at i= 6 : 0.54"
          [,1]
[1,] 0.5447909
[2,] 0.4552091
```

We see above that E has higher and higher probability as the process iterates. This model stabilizes after only 6 iterations, so even if we iterate 5000 times we end up at about the same place: .555 and .444. It changes slightly at each iteration but stabilizes at these values.

17.3 Hidden Markov Model

The limitation of the Markov Model is that it only encodes what we know. There may be hidden, or latent, variables that influence our model. How can we discover these? Through hidden Markov models, HMMs.

Figure 17.2 shows the basics of a hidden Markov model. We only see the observed data X. We assume there is some latent variable Z that manifests X. Notice there are transitions from Z state to Z state horizontally. We do not know what the transition probabilities are, but a hidden Markov model can help us discover patterns that could have generated our data.

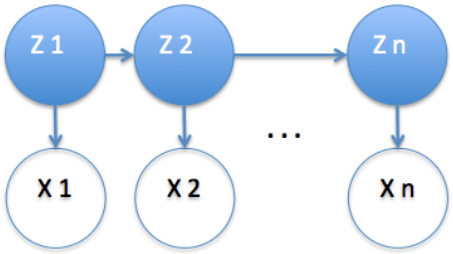

Figure 17.2: Hidden Markov Model

17.4 HMM in R

There are several packages that deal with HMM in R. We are going to look at `depmixS4`[1], a package by Visser and Speekenbrink. The package can generate mixture models including Markov models, hidden Markov models, and other mixture models. The documentation has several usage example. Below, we are going to continue our exercise model and see if we can detect a pattern in the data. First we load the depmixS4 package. Next we generate our observations. What we have are 7 transitions, one for each day of the week, and then we replicate that 10 times for 10 weeks. Interpreting the days in order, it seems that this person exercised (state 2) on Sunday and Saturday but not during the week.

Code 17.4.1 — HMM Fit the Model. Exercise Observations.

```
library(depmixS4)
#set of states
states <- c(2, 1) # E and NotE
n <- 140 # number of transitions (7 days, 10 weeks)
obs <-
  rep(c(c(2,2),c(2,1),c(1,1),c(1,1),c(1,1),c(1,1),c(1,2)),10)
```

Now that we have our observations, the Xs in Figure 17.2, we can let the algorithm find the transition probabilities. This is a 2-step process in this package. Step 1 creates the model and Step 2 fits the model. Notice we also set a seed.

[1] https://cran.r-project.org/web/packages/depmixS4/vignettes/depmixS4.pdf

Code 17.4.2 — HMM Fit the Model. Exercise Observations.

```
# Start the HMM
set.seed(1234)
# 1. create the model
mod <- depmix(response = obs ~ 1, data=data.frame(obs), nstates=2)
# 2. fit the model
f <- fit(mod)
summary(f)
Initial state probabilties model
pr1 pr2
  1   0
Transition matrix
        toS1  toS2
fromS1 0.744 0.256
fromS2 0.100 0.900
```

We output the summary() above of the fitted model and displayed a portion of the output in the code box. Notice that the transition matrix was learned from the data. Now let's plot our results. First we extract the estimates from the fitted model. Then plot the actual observations over the estimates. Notice that the spikes of exercising (weekends) in the observations was matched by the estimates.

Code 17.4.3 — HMM continued. Extract Estimates and Plot.

```
estimates <- posterior(f)  # get the estimated state for each day
par(mfrow=c(2,1))
plot(1:n, obs, type='l', main='Observations, X')
plot(1:n, estimates[,2], type='l', main='Estimate')
```

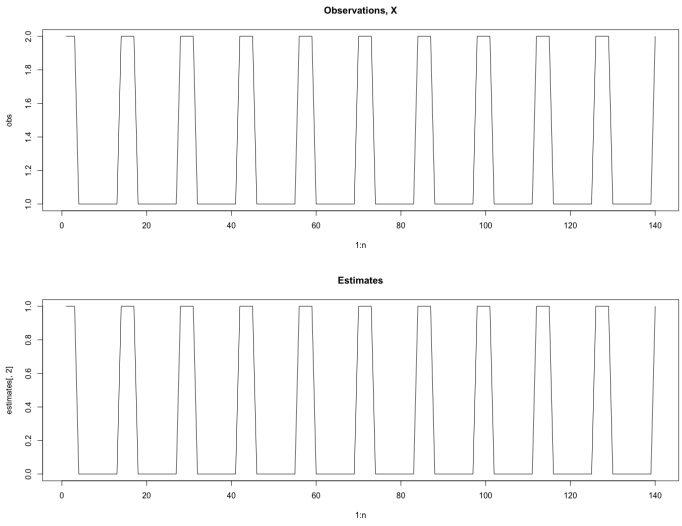

Figure 17.3: Observations and Estimates from HMM

17.5 Metrics

If we type the model name at the console we see the following:

```
> f
Convergence info: Log likelihood converged to within tol. (relative change)
'log Lik.' 4462.099 (df=7)
AIC:  -8910.198
BIC:  -8889.606
>
```

We are given information on the convergence, the log likelihood, AIC and BIC values. The AIC, Akaike information criterion, and BIC, Bayesian information criterion, are closely related. AIC and BIC are typically used to select among models, here we only have one model. If we had more than one model and compared either AIC or BIC metrics, we would prefer the model with the lowest score. In, the standard formulas for AIC and BIC below, n is the number of observations, k is the number of parameters.

$$AIC = 2k - 2ln(\hat{L}) \qquad BIC = ln(n)k - 2ln(\hat{L}) \tag{17.1}$$

17.6 The Algorithm

In the hidden Markov model, we have n observations, $X = (x_1, x_2, ..., x_n)$. We do not know the parameters, θ that generated our observations but we can estimate them. First, let us consider the likelihood of observing X given these unknown parameters.

$$p(X|\theta) = \sum_z p(X, Z|\theta) \tag{17.2}$$

We can't really perform the summation directly in the above equation because we have N variables over K states. Our example above was binomial with K=2 but HMMs are often used for multinomial scenarios. We would need to calculate K^n terms, so this will grow exponentially with n. For this and other reasons, a direct solution is not feasible. Instead the EM, Expectation-Maximization algorithm is used to estimate a solution. In the application to HMM, the EM algorithm iteratively maximizes the expected joint log-likelihood of the parameters given the observations and states. That is the M step. The E step calculates expected values for the latent states given the observations and a set of initial states.

17.7 Another HMM in R

For this example we use the sp500 data set included in package depmixS4. First, let's load the package and look at the data. The column of interest is the 6th column, logret, the log ratio of the closing indices. For example the second row is 0.004 which can be calculated as `log(17.29/17.22)` at the console. We took the ratio of the index of the previous month to the current month, then the log. The range of this column is [-0.245, 0.15] and the mean (not shown) is 0.0058.

Code 17.7.1 — HMM. S& P500 data

```
library(depmixS4)
data(sp500)
head(sp500)
range(sp500[,6])

> head(sp500)
             Open  High   Low Close  Volume       logret
1950-02-28 17.22 17.22 17.22 17.22 1310000  0.009921295
1950-03-31 17.29 17.29 17.29 17.29 1880000  0.004056801
1950-04-28 17.96 17.96 17.96 17.96 2190000  0.038018763
1950-05-31 18.78 18.78 18.78 18.78 1530000  0.044645411
1950-06-30 17.69 17.69 17.69 17.69 2660000 -0.059792966
1950-07-31 17.84 17.84 17.84 17.84 1590000  0.008443619
> range(sp500[,6])
[1] -0.2454280  0.1510432
```

Now that we have our data loaded and understand the data in column 6, we can run the HMM algorithm and plot our results.

Code 17.7.2 — HMM continued. S& P500 data

```
# create the model, then fit
mod <- depmix(logret~1, nstates=2, data=sp500)
set.seed(1)
fmod <- fit(mod)

# plot
par(mfrow=c(3,1))
plot(posterior(fmod)[,1], type='l')
plot(posterior(fmod)[,2], type='l')
plot(sp500[,6], type='l')
```

The bottom graph is the actual column 6 data. The top model predicts the volatility going down, the middle one going up. We can definitely detect volatile months in the sequence. The sp500 data starts in February 1950 and ends in January 2012.

17.8 Summary

In this chapter we explored hidden Markov models with package depmixS4. Another package worth investigating is seqHMM.[2] This package also outputs the transition probabilities in a graphical format.

The Markov of the Markov model is Andrey Markov, a Russian mathematician at the turn of the 20th Century. A Markov process assumes that future states depend only on the curent state, not states going back in time. A Markov chain describes a sequence

[2] https://github.com/helske/seqHMM

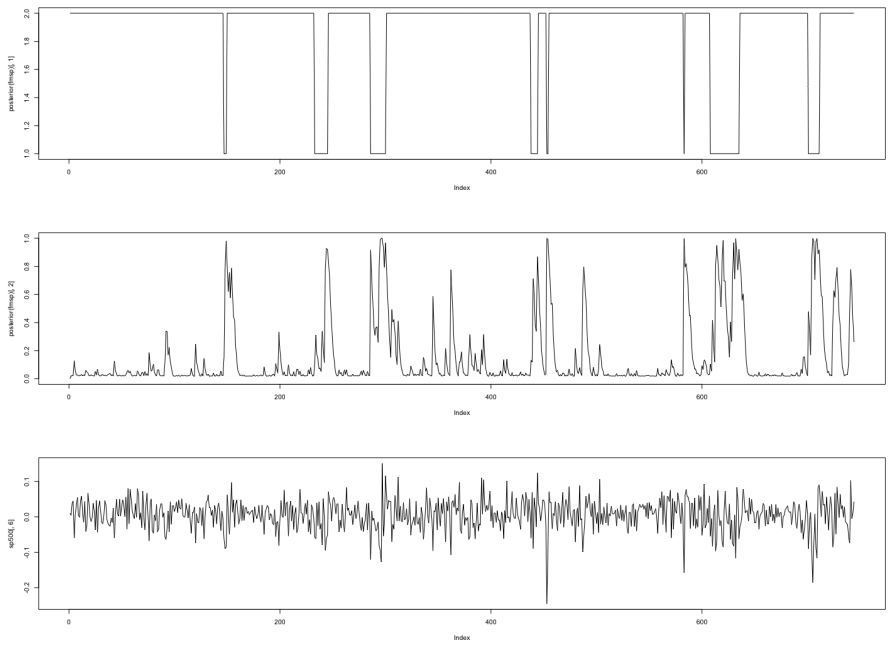

Figure 17.4: S&P 500 Data

of possible events in which the next state only depends on the current state. We saw a Markov chain in our first example of exercise or not exercise where we modeled these two states and the transition probabilities between them. HMMs are commonly used to model language, images, biological processes, virtually anything that is sequential in nature.

HMMs model hidden or latent variables that cause the observations. Variations of EM algorithms are often used to find optimal estimates for the latent variables and their transition probabilities. As we saw in the depmixS4 package, the algorithm discovers these probabilities and can output them for us.

A final thing to discuss is a Markov decision process. This is a Markov chain augmented with an action vector describing possible actions. This is often used in Reinforcement Learning, the subject of the next chapter.

17.8.1 Going Further

An excellent tutorial, *An Introduction to Hidden Markov Models and Bayesian Networks* is by Z. Ghahramani, and was published in the International Journal of Pattern Recognition and Artificial Intelligence[3].

[3]http://mlg.eng.cam.ac.uk/zoubin/papers/ijprai.pdf

18. Reinforcement Learning

Currently, Reinforcement Learning is a burgeoning field of AI, demonstrating success at improving self-driving cars, beating humans at the complex board game Go, optimizing data center energy usage, and much more. Reinforcement learning mimics how humans actually learn: a little trial and error, finding what works, and remembering. The reason that RL is taking off now is that it is being combined with deep learning. However in this chapter we discuss the traditional techniques of RL because most of us don't have access to data centers full of GPUs and data needed for deep learning RL approaches.

18.1 Overview

In previous chapters the focus was on learning from data for the purpose of either prediction or simply learning more about the data. In this chapter the focus shifts to learning how to act. We have a learner, an agent, who will learn to make decisions based on probability theory and a utility function that keeps track of rewards earned for actions in different situations. The agent must learn to act under uncertainty, taking the action that most probably leads to a reward.

The core components of reinforcement learning are:

- an agent that interacts with the environment
- the environment, which is a set of predefined states, S
- a predefined set of actions, A, which the agent can take
- a set of rewards, R, that serve as reinforcement signals

The agent learns over many iterations of interaction with the environment. At each iteration, i, the agent observes the available states and selects an action. Based on the action chosen, the agent receives a numerical reward. After an action, a new set of states is available.

Learning, sometimes called Q-learning, relies upon remembering what was learned. The state-action function, $Q(s,a)$ defines the expected reward of each possible action for

every state. A policy function $\pi^*(s,a)$, seeks to maximize the reward.

To jump-start learning, data in the form of sample state-action-reward sequences from which the agent learns can be fed into the learning algorithm. This *experience replay* can speed up convergence and help the agent learn faster.

18.2 The Markov Decision Process

In the last chapter we explored Markov Models which seem to have a lot in common with the reinforcement learning process described above. So what is the difference between RL and MDP? RL generally assumes there is some underlying Markov Decision Process, which it seeks to learn. In the MDP learning process, we know the states and possible actions at each time step, and the agent will receive a reward corresponding to the action chosen at the state. It is Markov in the sense that decisions consider only the current state, not previous states or actions. In Reinforcement Learning, the system may have to first learn the MDP. Another difference is that RL will try random actions in order to explore and learn new things beyond its current policy. So we can say that RL is an extension of the Markov decision process. To build up our understanding we will look at an example of MDP in R using package `MDPtoolbox`, available in CRAN.

The Markov decision processis a 4-tuple <S, A, P, R> where S is the set of states, A is the set of actions, P() is a set of transition probabilities and $R(s,s',a)$ is the immediate reward for moving from state s to s' via action a. All of these are predefined. What we want to learn is the policy function $\pi()$ that maximizes a cumulative reward. More specifically, P() is:

$P(s,s',a) = Pr(s_{t+1} = s'|s_t = s, a_t - a)$

the probability that action a in state s at time t leads to state s' at time $t+1$.

Learning the policy $\pi(s)$ means learning the optimal choice of action at that state in order to maximize the long-term reward. The reward is often discounted by a factor, gamma, that ranges from 0 to 1. This serves to disincentive immediate reward in favor of long term reward. The reward over the long term (possibly infinite) can be expressed as:

$$\sum_{t=0}^{\infty} \gamma\, R(s_t, s_{t+1}, \pi(s_t)) \tag{18.1}$$

How is the policy learned? It could be learned by linear or dynamic programming, more commonly the latter which we discuss here. One approach is to set up two arrays, V for rewards, and π for the policy. Both arrays are indexed by the state:

$$\pi(s) := argmax_a \sum_{s'} P_a(s,s')[R_a(s,s') + \gamma\, V(s')] \tag{18.2}$$

$$V(s) := \sum_{s'} P_{\pi(s)}(s,s')[R_{\pi(s)}(s,s') + \gamma\, V(s')] \tag{18.3}$$

These definitions above are recursively updated through algorithms such as dynamic programming.

18.3 MDPtoolbox

Next we look at a vignette from MDPtoolbox, using a sample forest problem included in the package. The problem models a tract of forest land that is managed with two objectives. Objective 1 is to maintain old forest for wildlife while Objective 2 is to cut wood to make a profit. Time intervals are annual for the two possible actions: 1, Wait, or 2, Cut. There are 3 states: in state 1 the trees are 20 years old or less, in state 2 the trees are between 21 and 40 years old, and in state 3 the trees are older than 40 years. There is a probability that a fire occurs, in which case the state reverts to the youngest state, state 1. The sample forest function has 4 arguments:

- S - the number of states; default is 3
- r1 - the real-valued reward when the forest is in the oldest state and action chosen is Wait; default is 4
- r2 - the real-valued reward when the forest is in the oldest state and action chosen is Cut; default is 2
- p - the probability of a wildfilre; default is 0.1

The algorithm generates an array P(s,s',a) for transition probabilities and an R(s',a) matrix. Below we show the problem with the default values used.

Code 18.3.1 — MDP. Forest Managment Example.

```
library(MDPtoolbox)

# Generates a MDP for a simple forest management problem
MDP <- mdp_example_forest()
# Find an optimal policy
results <- mdp_policy_iteration(MDP$P, MDP$R, 0.9)
# see the results
results
$V
[1] 26.244 29.484 33.484

$policy
[1] 1 1 1

$iter
[1] 2

$time
Time difference of 0.02498722 secs
```

So the policy chosen for the 3 states is wait, wait, wait. The Value, V, at each time period is given as well. Clearly the reward increased for not cutting. If you type the model name at the console you will see the transition probability array P, and the reward matrix R. Experiment with the parameters, changing them from the default settings, and see what happens.

18.4 Reinforcement Learning

Reinforcement learning is closer to how humans learn than the Markov decision process because when we learn, we have to learn everything: what actions we can take, what states we may end up in, and what rewards our actions bring. No one gives their baby walking lessons, they just figure it out, through trial and error and encouragement from those around him. RL gives a computational foundations to let an agent learn from experience. RL is automated, goal-directed learning.

As in MDP, we have a policy which specifies how the agent should act at a given state, a value function that focuses on cumulative reward over time, a reward function, and some means of encoding the environment.

18.5 The ReinforcementLearning Package

We are going to explore the `ReinforcementLearning` package in R, available in CRAN. The following example utilizes the tictactoe data provided in the package that consists of over 400K game states. The agent must learn the optimal actions for each state of the board. The reward is 0 for tie, +1 for win, and -1 for lose. The following code will take a few minutes to run.

Code 18.5.1 — Reinforcement Learning. TicTacToe

```
library(ReinforcementLearning)
# Load dataset
data("tictactoe")

# Define reinforcement learning parameters
control <- list(alpha = 0.2, gamma = 0.4, epsilon = 0.1)

# Perform reinforcement learning
model <- ReinforcementLearning(tictactoe, s="State", a="Action",
    r = "Reward", s_new = "NextState", iter = 1, control = control)

# Print optimal policy
head(policy(model))
```

You can dig further into the model at the console by typing `model$` and pausing while the options pop up. For example, `model$Reward` lists the reward.

The three control parameters are for the behavior of the agent and all three range between 0 and 1 and have default values of 0.1.

- alpha - controls the learning rate; the lower the value the slower the learning; alpha=0 means nothing is learned
- gamma - discount factor determines the importance of future rewards; gamma=0 means that the agent only considers immediate rewards; gamma closer to 1 makes the agent work toward longer term rewards
- epsilon - exploration parameter, the probability that the agent will explore the environment through a random action;

There is another built-in sample experience, shown in the code block below.

Code 18.5.2 — Reinforcement Learning. Grid World

```
print(gridworldEnvironment)

# define states and actions
states <- c("s1", "s2", "s3", "s4")
actions <- c("up", "down", "left", "right")

# Generate 1000 iterations
sequences <- sampleExperience(N = 1000, env = gridworldEnvironment,
    states = states, actions = actions)

#Solve the problem
solver_rl <- ReinforcementLearning(sequences,
    s = "State", a = "Action",
    r = "Reward", s_new = "NextState")

#Getting the policy; this may be different for each run
solver_rl$Policy

#Getting the Reward; this may be different for each run
solver_rl$Reward
```

Printing the gridworldEnvironment shows the definition of the world. There are four states in a 2x2 arrangement with s1 and s2 over s2 and s3, respectively. The maximum reward is to get to s4. If you try to make a move that is not defined, you get minus 1 reward. The optimal policy, starting at s1, is to move down to s2, then right to s3, then up to s4. The policy at s4 will be different every time you run it unless you set a seed.

18.6 Summary

Reinforcement Learning has deep roots in AI using Markov foundations and techniques such as dynamic or linear programming. The most exciting trend in RL is Deep Reinforcement Learning which is allowing systems to scale to problems unimaginable with traditional approaches. An excellent surved of deep RL from an IEEE Special issue can be found here: `https://arxiv.org/pdf/1708.05866.pdf`.

Full book on dynamic programming and RL is availble here: `https://orbi.uliege.be/bitstream/2268/27963/1/book-FA-RL-DP.pdf`

18.6.1 Going Further

A nice survey of the field, from historic roots to current trends is provided in this MIT Technology Review article: `www.technologyreview.com`.

DeepMind is a British AI company acquired by Google in 2014. Read about innovations going on at DeepMind here `https://deepmind.com/blog/deep-reinforcement-learning/`.