2
Conceitos Básicos

Neste capítulo, introduzimos alguns conceitos relevantes para o nosso trabalho. Inicialmente, na seção 2.1, detalhamos o funcionamento do mercado financeiro, definindo alguns termos usados no mercado de ações. Posteriormente, na seção 2.2, descrevemos um tipo de aprendizado de máquina (aprendizado por reforço) que será usado no nosso trabalho. Por último, na seção 2.3, apresentamos algumas técnicas de validação de algoritmos de aprendizado de máquina.

2.1
Microestrutura do Mercado

Nesta seção, descrevemos o funcionamento da maioria das bolsas de valores modernas, explicando como negociações são feitas e como as ofertas são organizadas. Um negociador pode fazer ofertas de compra ou de venda de ações. Dizemos que uma oferta foi executada quando as ações foram compradas (no caso de oferta de compra) ou vendidas (oferta de venda).

Mercados financeiros modernos como a BOVESPA & BMF, funcionam com *ofertas limitadas*, isto significa que compradores e vendedores além de informar o número de ações que desejam negociar (volume), especificam o preço desejado. Este tipo de oferta garante ao negociador o quanto ele vai pagar (ou receber) por cada ação, mas não garante que as ofertas serão executadas.

Todas as ofertas são organizadas em filas separadas: uma para todas as de compra (*book* de compra) e outra para as de venda (*book* de venda). Os *books* são ordenados pelos valores das ofertas, sendo o de compra na ordem decrescente e o de venda na crescente. Nas Tabelas 2.1 e 2.2, podemos ver um exemplo dos *books* de uma determinada ação. A primeira linha da Tabela 2.1 representa uma oferta de compra. Neste caso, o comprador deseja obter 15 ações pagando, no máximo, R$ 29, 32 por cada. De maneira semelhante, a linha do topo da Tabela 2.2 representa uma intenção de venda de 20 ações a R$ 29, 65 cada. O maior valor de compra é chamado de *bid* (no caso, R$ 29, 32) e o menor valor de venda é chamado de *ask* (no caso, R$ 29, 65). Estas filas (*books*) definem uma ordem de prioridade de execução das ofertas baseada

nos seus valores.

***Book* de Compra**	
Ações	**Preço**
15	29,32
35	29,30
250	29,25
120	28,32
200	28,20
1892	28,15
230	27,99
254	27,88
365	27,30
725	26,55

Tabela 2.1: *Book* de Compra

***Book* de Venda**	
Ações	**Preço**
20	29,65
325	29,65
252	29,66
520	29,67
8	29,70
280	29,81
1895	29,81
254	29,82
974	29,83
256	29,85

Tabela 2.2: *Book* de Venda

Os *books* são alterados a cada oferta feita. Suponha que uma determinada ação está sendo vendida a R$ 29, 65 (Tabela 2.2), mas nós queremos comprar 100 ações pagando no máximo R$ 29, 25 por cada. Ao invés de esperar o preço baixar até o que desejamos, podemos submeter uma oferta limitada que reflita nosso interesse. Assim, nossa oferta será colocada no *book* de compra imediatamente após a oferta por 250 ações a R$ 29, 25 cada, visto que oferecemos o mesmo preço e já existia uma com o mesmo valor. Sempre que uma nova oferta for feita, verificamos a possibilidade de sua execução. Se existirem ofertas que permitam sua execução, então ela ocorre. Por exemplo, uma oferta para comprar 700 ações pagando R$ 29, 66 por cada irá causar a execução de 20 ações a R$ 29, 65, 325 ações a R$ 29, 65 e 252 ações a R$ 29, 66 (as primeiras ofertas da Tabela 2.2), totalizando de 597 ações compradas. As 103 ações não executadas formarão uma oferta que será colocada no *book* de compra. Neste último exemplo, podemos ver que se quisermos comprar todas

as ações de uma só vez, iremos ter que nos submeter a preço cada vez piores do *book* de venda.

2.2 Aprendizado por Reforço

No aprendizado por reforço, o processo de aprendizagem é feito por um agente que toma decisões em um ambiente. Cada ação executada a fim de tentar resolver o problema, resulta em uma recompensa (ou punição). Em um processo de tentativa e erro, o agente deverá aprender a melhor política, que é a sequência de ações que maximiza a recompensa total (Alpaydin 2004).

2.2.1 Modelo de Aprendizado por Reforço

Podemos imaginar aprendizado por reforço como o problema de um robô que existe em um ambiente onde ele pode sentir e agir. O robô pode não ter ideia dos efeitos de suas ações sobre o ambiente, isto é, ele pode não saber como o ambiente irá ser alterado por suas ações. A cada ação feita, o robô recebe uma recompensa ou uma punição, que chamamos de sinal de reforço. Seu objetivo é definir uma política de execução de ações de maneira que a soma de todas as recompensas recebidas seja máxima. Para maximizar os sinais de reforço, será necessário prever como as ações alterarão as entradas e quais recompensas serão recebidas (Nilsson 2005).

Por exemplo, sabemos que um agente pode aprender a jogar xadrez por aprendizado supervisionado, usando exemplos de jogadas junto com os melhores movimentos para cada uma. Se o agente não dispuser de um conjunto de exemplos para aprender, podemos aplicar aprendizado por reforço. Tentando movimentos aleatórios, o agente pode eventualmente construir um modelo preditivo do ambiente (como o tabuleiro estará disposto depois de um dado movimento e qual o movimento mais provável do oponente, por exemplo). Mas sem alguma resposta do ambiente que defina o que é bom ou ruim, o agente não terá informações necessárias para decidir qual movimento fazer. No caso do xadrez, no fim do jogo o agente sabe quando ganhou ou perdeu. Este tipo de *feedback* é chamado de recompensa ou sinal de reforço. Em jogos como o xadrez e jogo da velha, o sinal de reforço só é recebido ao término da partida, mas em outros ambientes as recompensas são mais frequentes. No ping-pong, cada ponto marcado pode ser considerado uma recompensa. As recompensas também podem ser dadas por um professor que diz: "bela jogada", mas nunca diz qual é a melhor (Russell & Norvig 2003).

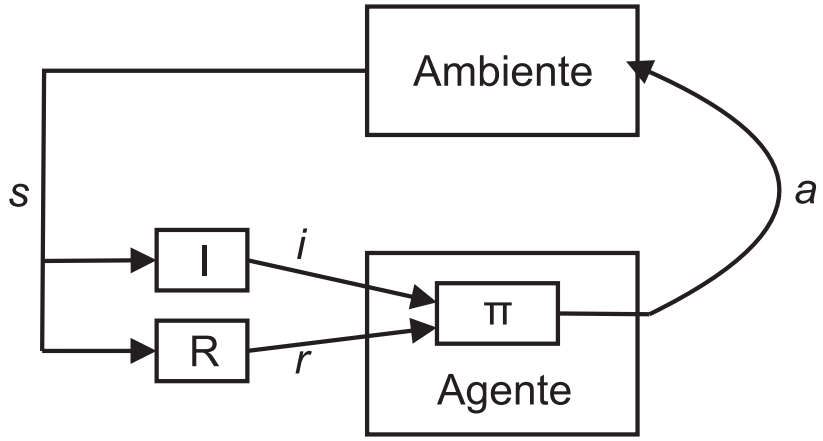

Figura 2.1: Modelo padrão de aprendizado por reforço

No modelo padrão de aprendizado por reforço, um agente é conectado a um ambiente via percepção e ação, como ilustrado na Figura 2.1. Em cada passo da interação o agente recebe uma entrada, i, que é uma indicação do estado atual, s, do ambiente. O agente então escolhe uma ação, a, que será feita. A ação altera o estado do ambiente e o valor dessa transição de estado é comunicado ao agente através de um sinal de reforço, r. A política do agente, π, deve escolher ações que maximizem a soma a longo prazo das recompensas. Isto pode ser aprendido através do tempo basicamente por um sistema de tentativa e erro (Kaelbling et al. 1996). Apresentamos um possível algoritmo de aprendizado na seção 2.2.5.

Formalmente, o modelo consiste de:

– um conjunto discreto de estados do ambiente (não necessariamente finito), S;

– um conjunto discreto de ações do agente (não necessariamente finito), A;

– um conjunto de sinais de reforço recebidos.

A figura também inclui uma função de entrada I, que determina como o agente deve ver o estado do ambiente. Assumimos que I é a função identidade, o que significa que o agente vê o estado do ambiente exatamente como ele é. Uma maneira intuitiva de entender a relação entre o agente e o ambiente pode ser vista no diálogo da Tabela 2.3.

O objetivo do agente é encontrar uma política, π, que mapeia as entradas, i, para ações de maneira que maximize as recompensas, r, recebidas ao longo do tempo. Este tipo de aprendizado é chamado de Aprendizado por Reforço (Nilsson 2005).

Ambiente:	Você está no estado 10 e tem 2 ações possíveis
Agente:	Farei a ação 1
Ambiente:	Você recebeu um sinal de reforço de 5.
	Você agora está no estado 2 e tem 4 ações possíveis
Agente:	Farei a ação 4
Ambiente:	Você recebeu um sinal de reforço de 3.
	Você agora está no estado 7 e tem 8 ações possíveis
Agente:	Farei a ação 7
Ambiente:	Você recebeu um sinal de reforço de 0.
	Você agora está no estado 1 e tem 2 ações possíveis
⋮	⋮

Tabela 2.3: Exemplo de diálogo entre o agente e o ambiente

2.2.2 Propriedade Markoviana dos Estados

No modelo de aprendizado por reforço, o agente toma a decisão de qual ação será feita baseado no estado atual do ambiente, ou seja, a ação do agente depende da sua percepção. Não esperamos que a percepção do agente informe tudo sobre o ambiente ou mesmo tudo que seria útil para se tomar decisões. Se o agente for um jogador de cartas, não esperamos que ele saiba qual a próxima carta a ser retirada do monte. Se o agente for um detetive, não esperamos que ele saiba assim que chegue à cena de um crime de todas as evidências deixadas pelo criminoso. Em todos os casos existem informações escondidas sobre o estado do ambiente que seriam importantes para o agente tomar uma decisão, mas o agente não pode saber da existência delas porque não recebeu nenhuma indicação. Não penalizamos o agente por não saber algo que importa, mas sim por ter sabido algo e depois esquecê-lo (Sutton & Barto 1998).

O importante é que um sinal que representa um estado indique as sensações passadas do agente e que as informações passadas relevantes sejam mantidas. Isto requer mais que as sensações imediatas, mas nunca mais que o histórico de todas as sensações passadas. Um sinal de estado que mantém todas as informações relevantes possui a *Propriedade Markoviana* (Sutton & Barto 1998). Por exemplo, no jogo de cartas Paciência, em um determinado instante, a configuração das cartas na mesa possui a *Propriedade Markoviana* porque sumariza tudo de importante sobre os movimentos feitos anteriormente. Olhando somente para uma determinada configuração das cartas, não sabemos qual a sequência de movimentos que foram feitos, mas o que realmente importa para o futuro do jogo está armazenado na atual configuração, ou seja, no atual estado.

De acordo com o apresentado por Sutton (Sutton & Barto 1998), iremos

formalmente definir a propriedade de Markov para o problema de aprendizado por reforço. Considere como o ambiente pode responder no instante $t+1$ à ação feita no instante t. No caso mais geral, a resposta do ambiente pode depender de tudo que aconteceu antes. Assim, podemos definir a seguinte distribuição de probabilidade:

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \cdots, s_0, a_0, r_0\} \quad (2\text{-}1)$$

Para todo s', r, e todos os possíveis valores dos eventos passados $s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \cdots, s_0, a_0, r_0$. Podemos interpretar a distribuição acima como sendo a probabilidade do estado no instante $t+1$ ser s' e do reforço recebido ser r, dado que fizemos a ação a_k no estado s_k resultando na recompensa r_k, $\forall k \in \{0, 1, \cdots, t\}$. Se o sinal de estado possui a *Propriedade Markoviana*, a resposta do ambiente no instante $t+1$ depende apenas de s_t e a_t. O que faz com que a distribuição de probabilidade seja definida como:

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\}, \quad (2\text{-}2)$$

para todo s', r, s_t e a_t. Em outras palavras, um sinal de estado possui a *Propriedade Markoviana* se e somente se a definição 2-1 for igual a 2-2 para todo s', r e $s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \cdots, s_0, a_0, r_0$. Neste caso, o ambiente e a tarefa do agente como um todo também possuem a *Propriedade Markoviana*. Se um ambiente possui essa propriedade, então pela definição 2-2 podemos predizer qual será o próximo estado e a próxima recompensa a ser recebida dado o estado atual e a ação que será feita (Sutton & Barto 1998).

2.2.3 Processos de Decisão de Markov (MDP)

Um problema de aprendizado por reforço que satisfaz à Propriedade de Markov é chamado de *Processo de Decisão de Markov* ou *MDP* (Bellman 1957). Se os estados e as ações forem finitos, então chamamos de *Processo de Decisão de Markov Finito* (*MDP* finito). *MDP's* finitos são particularmente importantes para a teoria de aprendizado por reforço (Sutton & Barto 1998). Um *MDP* finito é definido por seus estados, ações e por um passo na dinâmica do ambiente. Dado que estamos em um estado s e fizemos a ação a nele, a probabilidade do próximo estado ser s' é definida por:

$$P^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \quad (2\text{-}3)$$

Esses valores são chamados de *probabilidades de transição*. De maneira semelhante, dado que s' é o estado resultante da ação a sobre o estado s, o valor esperado da próxima recompensa é definido por:

$$R^a_{ss'} = E\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\} \quad (2\text{-}4)$$

Os valores de $P^a_{ss'}$ e $R^a_{ss'}$ especificam os aspectos mais importantes do comportamento de um *MDP* finito (Sutton & Barto 1998).

2.2.4 Funções Valor

Quase todos os algoritmos de aprendizado por reforço são baseados em estimar *funções valor*, que são funções de estados (ou pares de estado-ação) que estimam o quanto é bom para o agente estar em um determinado estado. A cada estado associamos um valor que representa as recompensas que deverão ser recebidas pelo agente em estados futuros. Os sinais de reforço esperados dependem de quais ações serão executadas em estados subsequentes, por isso, funções de valor são definidas para uma política, π, em particular (Sutton & Barto 1998).

Seja π uma política que mapeia cada par de estado $s \in S$ e ação $a \in A(s)$ em uma probabilidade $\pi(s, a)$ de fazer a ação a no estado s (onde $A(s)$ é o conjunto de todas as possíveis ações no estado s). O *valor* do estado s na política π, denotado por $V^\pi(s)$, é a recompensa total esperada quando iniciamos em s e seguimos π. Segundo Sutton (Sutton & Barto 1998), podemos definir formalmente $V^\pi(s)$ pela seguinte equação.

$$V^\pi(s) = E_\pi\{R_t|s_t = s\} = E_\pi\left\{\sum_{k=0}^{\infty} r_{t+k+1}\middle| s_t = s\right\} \quad (2\text{-}5)$$

Onde $E_\pi\{\}$ denota o valor das recompensas esperadas dado que o agente segue a política π, t é qualquer passo no tempo e r_t é o reforço recebido por fazer a melhor ação, de acordo com π, no estado s_t. Chamamos V^π de *função valor-estado*. Similarmente, definimos o valor de fazermos a ação a no estado s de acordo com a política π, denotado $Q^\pi(s, a)$, como sendo as recompensas esperadas se fizermos a ação a em s e depois seguirmos π. Chamamos Q^π de *função valor-ação para a política* π. Q^π pode ser definida formalmente pela equação abaixo (Sutton & Barto 1998).

$$Q^\pi(s, a) = E_\pi\{R_t|s_t = s, a_t = a\} = E_\pi\left\{\sum_{k=0}^{\infty} r_{t+k+1}\middle| s_t = s, a_t = a\right\} \quad (2\text{-}6)$$

Ainda segundo Sutton (Sutton & Barto 1998), as funções V^π e Q^π podem ser estimadas através de experiências. Por exemplo, se um agente segue uma política π e mantém uma média, para cada estado encontrado, dos sinais de reforço recebidos, então a média irá convergir para o valor do estado, $V^\pi(s)$, pois o número de vezes que o estado é encontrado tende ao infinito.

Se separarmos as médias dos valores recebidos para cada ação feita no estado, então essas médias irão convergir para o valor da ação, $Q^\pi(s,a)$. Um comportamento fundamental de V^π usado em aprendizado por reforço é que esta função satisfaz uma propriedade recursiva mostrada abaixo.

$$\begin{aligned} V^\pi(s) &= E_\pi\{R_t|s_t = s\} && (2\text{-}7)\\ &= E_\pi\left\{\sum_{k=0}^{\infty} r_{t+k+1}\middle| s_t = s\right\} && (2\text{-}8)\\ &= E_\pi\left\{r_{t+1} + \sum_{k=0}^{\infty} r_{t+k+2}\middle| s_t = s\right\} && (2\text{-}9)\end{aligned}$$

Pela equação 2-9, podemos ver que se somarmos a recompensa imediata recebida (r_{t+1}) com todas os sinais de reforço dos instantes seguintes, encontraremos $V^\pi(s)$. Na definição 2-4, especificamos formalmente a recompensa imediata esperada caso a ação a seja feita no estado s e o ambiente passe para o estado s'. Assim, $V^\pi_{s'}(s)$ pode ser definida da seguinte maneira.

$$V^\pi_{s'}(s) = R^a_{ss'} + E_\pi\left\{\sum_{k=0}^{\infty} r_{t+k+2}\middle| s_{t+1} = s'\right\} \quad (2\text{-}10)$$

Como não sabemos para qual estado s' o ambiente irá e nem qual ação provoca a transição de s para s', precisamos usar as probabilidades de transição de estados para que toda possível alteração no ambiente e todas as possíveis ações em s sejam levadas em consideração. Assim, $V^\pi(s)$ pode ser definida pela seguinte equação.

$$V^\pi(s) = \sum_{a\in A(s)} \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + E_\pi\left\{\sum_{k=0}^{\infty} r_{t+k+2}\middle| s_{t+1} = s'\right\}\right] \quad (2\text{-}11)$$

Pelas equações 2-7 e 2-8, temos:

$$V^\pi(s) = \sum_{a\in A(s)} \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + V^\pi(s')\right] \quad (2\text{-}12)$$

Essa propriedade recursiva nos permite definir o valor do estado atual baseado em seu sucessor, facilitando a aplicação de algoritmos de programação dinâmica no aprendizado.

2.2.5
Encontrando uma Política Ótima

Uma maneira de encontrar uma política ótima para um problema é determinar sua função valor ótima. Se soubermos o valor ótimo de cada

estado e a ação que implica nesse valor, então temos a política ótima, pois poderemos em todo estado fazer a ação que nos faz receber a recompensa ótima definida por $V^{\pi}(s)$. O valor ótimo de V^{π} pode ser determinado por um simples algoritmo chamado *value iteration*, cujo pseudocódigo é mostrado a seguir. Bellman (Bellman 1957) e Bertsekas (Bertsekas 1987) provam que este algoritmo faz os valores de V^{π} convergirem para os valores ótimos $V^{\pi *}$.

Algoritmo 1 *Value Iteration*

inicialize $V^{\pi}(s)$ arbitrariamente. $V^{\pi}(s) = 0, \forall s \in S$, por exemplo.
repeat
  for all $s \in S$ **do**
    for all $a \in A(s)$ **do**
      $Q^{\pi}(s,a) \leftarrow R(s,a) + \sum_{s' \in S} P^{a}_{ss'} V^{\pi}(s')$
    end for
    $V^{\pi}(s) \leftarrow \max_a Q^{\pi}(s,a)$
    $\pi(s) \leftarrow a$
  end for
until π ser boa o suficiente

No algoritmo, $R(s,a)$ é a média aritmética de todas as recompensas recebidas no estado s quando tentamos a ação a. Para todos os estados $s \in S$ e para todas as ações $a \in A(s)$, calculamos os valores de $Q^{\pi}(s,a)$. O maior valor $Q^{\pi}(s,a)$ é armazenado em $V^{\pi}(s)$, fazendo com que, no fim do algoritmo, V^{π} possua os melhores valores para cada estado. A política π guarda para cada estado a ação que faz com que $V^{\pi}(s)$ seja máximo. Não é obvio quando parar o *value iteration* (Kaelbling et al. 1996). Uma possível condição de parada do algoritmo seria impor um limite de variação dos valores de V^{π}, ou seja, se em uma iteração nenhum valor de V^{π} sofresse uma variação maior que θ, onde θ é um número positivo próximo de zero, o algoritmo se encerraria (Alpaydin 2004).

2.2.6 Exemplo: Encontrando a Saída de um Labirinto

Podemos aplicar Aprendizado por Reforço para encontrar caminhos que levem à saída de labirintos. A Figura 2.2 exibe o labirinto que iremos usar como exemplo. O nosso labirinto possui uma célula de entrada (marcada pela letra I), uma célula de saída (marcada pela letra F), células vazias (células em branco) e células consideradas obstáculos (células em cinza). O nosso agente se encontra na célula inicial e necessita encontrar um caminho até a célula de saída se movendo apenas nas células em branco.

Precisamos modelar o problema em estados, ações e recompensas para que seja viável a execução de um algoritmo de aprendizado por reforço.

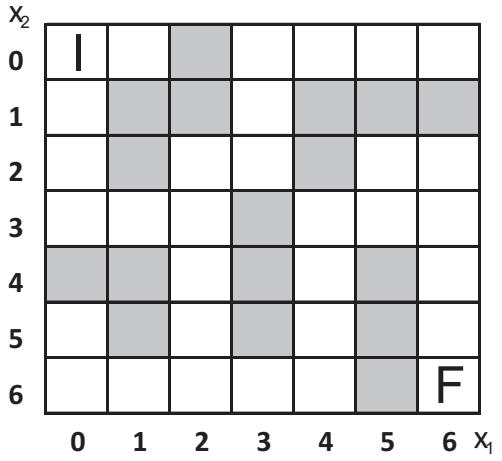

Figura 2.2: Labirinto

Representamos os estados como um vetor $s_i = (x_1, x_2)$, que identifica a célula em que o agente se encontra no labirinto. Quatro ações poderão ser feitas: c, b, d, e, que move o agente uma célula acima, abaixo, à direita e à esquerda respectivamente. O agente receberá -1 de recompensa quando uma ação o levar para uma célula inexistente ou obstruída, 0 quando atingir uma célula vazia e $+10$ quando atingir a célula final. Por exemplo, se o agente estiver na célula $(0, 1)$ e fizer a ação e, receberá -1 de recompensa por não existir célula à esquerda e permanecerá na mesma posição, por outro lado, se ele fizer a ação b, irá receber 0 de recompensa e passará para a posição $(0, 2)$.

Algoritmo 2 Aprender(s)

```
insereVisitado(s)
if valido(s) then
  for all a ∈ A do
    Q(s,a) = R(s,a) + V(s')
  end for
  V(s) = max_a Q(s,a)
end if
P = adjacentes(s)
for all s'' ∈ P do
  if não visitado(s'') then
    aprender(s'')
  end if
end for
```

O caminhamento em labirintos é de natureza Markoviana, isto é, a ação ótima em um determinado estado é independente de qualquer ação anteriormente feita, ou seja, a ação ótima em s só depende de s (Nevmyvaka et al. 2006). Seja $S = \{s_1, s_2, \cdots, s_n\}$ o conjunto de todos os estados ordenados pela distância euclidiana em relação ao estado final s_n, isto é,

s_1 é o estado que representa a célula mais distante da célula final s_n. Assim, se $i > j$, então s_i está mais próximo de s_n que s_j. Pela propriedade Markoviana do nosso problema, a ação ótima no estado s_k é independente de qualquer ação tomada nos estados $s_{k-1}, s_{k-2}, \cdots s_1$. Se estivermos no estado s_{n-1}, a melhor ação a se fazer é a que leve o nosso agente para o estado s_n, com isto, podemos resolver o problema por indução. Tendo feito a ação ótima no estado s_{n-1}, temos a informação necessária para fazer a ação ótima no estado s_{n-2}, que por sua vez nos permite fazer a ação ótima no estado s_{n-3}, assim, quando chegarmos no estado s_1, temos a nossa política ótima para encontrar um caminho entre s_1 e s_n. Esse tipo de iteração, onde iniciamos o processo de descobrimento da solução no final do problema (s_n) e encerramos no inicio (s_1), com o objetivo de encontrar a sequência ótima de ações , é chamado de *Backward Induction*.

Antes de explicarmos o funcionamento do algoritmo de aprendizado para esse problema, algumas considerações são necessárias. Para simplificar o problema, assumimos que $V(s) = 0 \ \forall s \in S$ no início da execução, onde S é o conjunto de todos os estados, e que os estado que representam as células I e F são conhecidos. O Algoritmo 2 é uma função recursiva que implementa a ideia exposta no parágrafo anterior. O algoritmo recebe como parâmetro um estado s que terá uma ação ótima associada no fim da execução da função. No início, o algoritmo chama a função $insereVisitado(s)$, que insere o estado em uma lista informando que a ação ótima já foi computada para ele. Antes de iniciar a seleção da ação ótima para o estado s, o algoritmo checa se o estado é válido através da função $valido(s)$, que verifica se o estado representa uma célula obstruída. Se o estado for válido, inicia-se o processo de seleção da melhor ação. O algoritmo itera sobre todas as possíveis ações e armazena a soma da recompensa recebida por fazer a ação a no estado s ($R(s, a)$) com a recompensa que será recebida se fizermos a ação ótima no estado subsequente s' ($V(s')$). Depois disto, armazenamos em $V(s)$ o maior $Q(s, a)$ calculado no laço anterior. Neste ponto do algoritmo, sabemos qual a melhor ação a ser feita no estado s e o quanto de recompensa iremos somar se continuarmos fazendo as ações ótimas nos estados seguintes até o estado final s_n. Antes de iniciarmos o último laço da função, armazenamos em P a lista de todos os estados adjacentes a s, ou seja, os estados que representam as células adjacentes à representada por s. Para cada estado adjacente, chamamos recursivamente a função de aprendizado se ele ainda não foi visitado, fazendo com que saibamos qual a ação ótima em cada um deles. Quando computarmos a ação ótima para o estado s_1, o algoritmo chega ao seu final. A execução do algoritmo se inicia chamando a função $aprender(s_n)$.

Estado	c	b	d	e	**Melhor Ação**
(6,5)	0	10	-1	-1	b
(6,4)	0	10	-1	-1	b
(6,3)	0	10	-1	0	b
(6,2)	-1	10	-1	0	b
(5,2)	-1	0	10	-1	d
(5,3)	10	-1	10	0	c
(4,3)	-1	0	10	-1	d
(4,4)	10	0	-1	-1	c
(4,5)	10	0	-1	-1	c
(4,6)	10	-1	-1	0	c

Tabela 2.4: Sequência de aprendizado da política ótima

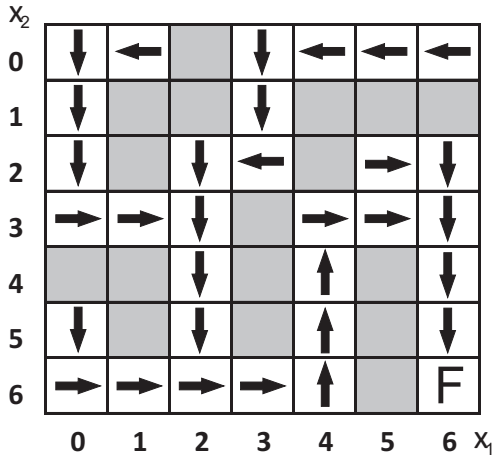

Figura 2.3: Política Ótima para Resolução do Labirinto

Na Tabela 2.4, podemos ver detalhadamente como foram feitos os cálculos da política para os dez primeiros estados. As colunas c, b, d e e mostram o quanto foi somado no cálculo de cada $Q(s, a)$, que é a soma da recompensa recebida pela ação com o V do estado para o qual a ação levaria o ambiente ($Q(s, a) = R(s, a) + V(s')$). A última coluna mostra qual a melhor ação a se tomar no estado representado pela linha. Por exemplo, na primeira linha podemos ver que o valor armazenado em $V((6, 5))$ foi 10 e que esse valor foi obtido fazendo a ação b. Na Figura 2.3, podemos visualizar qual a ação ótima em cada estado válido do nosso labirinto.

2.3 Validação de Algoritmos de Aprendizado de Máquina

Técnicas de validação de algoritmos de aprendizado de máquina são motivadas por dois problemas fundamentais: seleção de modelo e medida de desempenho (Kohavi 1995). Muitas técnicas de aprendizado possuem parâmetros livres que devem ser ajustados, como o número de vizinhos de um classificador

K-NN (Wang et al. 2006) ou o número de neurônios na camada escondida de uma rede neural (Setiono 2001). Podemos entender *seleção de modelo* como o ajuste correto desses parâmetros.

Para demonstrar como estimar a performance de um algoritmo de aprendizado de máquina por meio de técnicas de validação, vamos usar o exemplo de um classificador. Um classificador é uma função que mapeia dados não rotulados para um rótulo (Kohavi 1995). Seja $Y = \{y_i \mid y_i \in \mathbb{N}\}$ o conjunto de todos os possíveis rótulos, $D = \{x_1, x_2, \cdots x_k \mid x_i \in \mathbb{R}^n, 1 \leq i \leq k\}$ o conjunto de todos os vetores que precisam ser classificados e $C : D \rightarrow Y$ a nossa função classificadora. Definimos o erro de classificação de um elemento pela função $e(x_i) = (C(x_i) - y_i)^2$, onde $y_i \in Y$ é o rótulo correto de x_i. Assim, se o nosso classificador rotular corretamente x_i, $C(x_i)$ será igual a y_i e o erro será zero. Podemos comparar classificadores diferentes pelo valor global do erro, definido pela equação 2-13, onde $T \subseteq D$.

$$E(T) = \sum_{t_i \in T} e(t_i) \tag{2-13}$$

Se treinarmos o nosso classificador usando todos os dados (D) disponíveis, a função de erro global ($E(D)$) não refletirá o erro real do nosso classificador, seu resultado será muito otimista, já que o erro na fase de treinamento pode ser muito baixo (Kohavi 1995). Assim, nosso algoritmo de classificação pode ser muito bom para classificar os dados apresentados no conjunto de treino e, ao mesmo tempo, pode ter uma alta taxa de erro durante a classificação de dados não usados na fase de treino. Esse tipo de comportamento ocorre quando o algoritmo de aprendizado apenas "decora" os exemplos de treino e não consegue generalizar sua classificação para dados ainda não utilizados. Na literatura, esse problema é denotado como *overfitting*. Por esses motivos, para medir o erro real, é necessário dividir nosso conjunto de dados em conjuntos mutuamente exclusivos chamados de conjunto de treino e teste (Alpaydin 2004). Como o conjunto de testes não é usado na fase de aprendizado, o erro calculado para esse conjunto nos dá uma estimativa real do erro global do classificador. Existem algumas maneiras diferentes de dividir nosso conjunto de dados. Nas subseções seguintes iremos apresentar o método *holdout* e a validação cruzada.

2.3.1 Método Holdout

Neste método dividimos o conjunto de dados nos conjuntos de treino e teste (como o mostrado na Figura 2.4). Normalmente separamos 2/3 dos dados para treino e 1/3 para teste (Kohavi 1995). O erro global neste tipo de

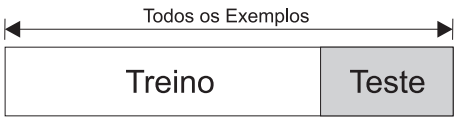

Figura 2.4: Método *holdout*

validação é calculado por $E(T)$, onde $T \subseteq D$ é a partição que foi selecionada para testes.

Existem alguns problemas com essa abordagem. Se o nosso conjunto de dados não for grande o suficiente, não podemos descartar uma boa parte dele para testes. Outro problema está no cálculo do erro. Como separamos apenas um conjunto para testes, a nossa estimativa de erro pode ser tendenciosa, se acontecer de escolhermos um conjunto de testes que produzam resultados muito bons ou muito ruins. As limitações dessa técnica de validação podem ser supridas pelas técnicas de validação cruzada.

2.3.2 Validação Cruzada

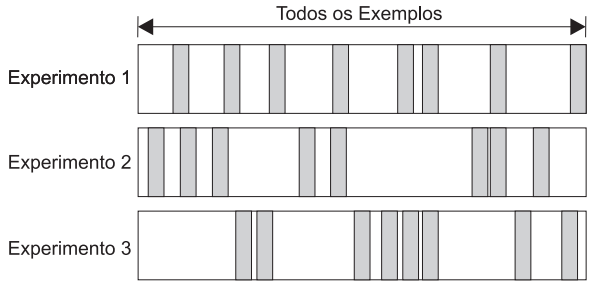

Figura 2.5: Seleção Aleatória de Subconjuntos (com $K = 8$)

Todas as técnicas de validação cruzada aqui apresentadas se baseiam em dividir o conjunto de dados em um ou mais subconjuntos de testes. Apresentamos três técnicas de validação cruzada: seleção aleatória de subconjuntos, *K-Fold Cross validation* e *Leave-one-out Cross Validation*. A principal diferença entre elas está na maneira de se subdividir o conjunto de dados.

Na seleção aleatória de subconjuntos, selecionamos aleatoriamente K subconjuntos de testes de maneiras diferentes, como mostrado na Figura 2.5. Cada configuração diferente dos subconjuntos de testes representa um experimento. Seja $H = \{h_1, h_2, \cdots h_K\}$ o conjunto de todos os experimentos e $T(h_i)$ o conjunto de dados usados para testes no experimento h_i. O erro global na seleção aleatória de subconjuntos pode ser calculado pela soma abaixo.

$$\frac{1}{K} \sum_{h \in H} E(T(h)) \tag{2-14}$$

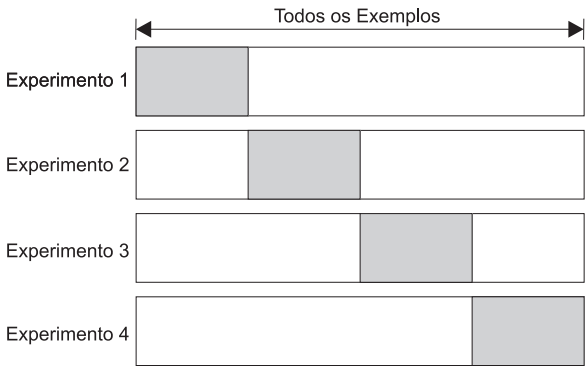

Figura 2.6: *K-Fold Cross validation* (com $K = 4$)

Na técnica *K-Fold Cross validation* dividimos o conjunto de dados em K partes de tamanhos aproximadamente iguais e em cada experimento, usamos $K - 1$ partes para treinar e uma parte para testar, como mostrado na Figura 2.6. A vantagem dessa abordagem é que todos os elementos do nosso conjunto de dados serão usados tanto para treino quanto para teste em algum momento. Normalmente K é escolhido como 10 ou 30 (Alpaydin 2004). O erro global pode ser calculado pela mesma função de erro da seleção aleatória de subconjuntos.

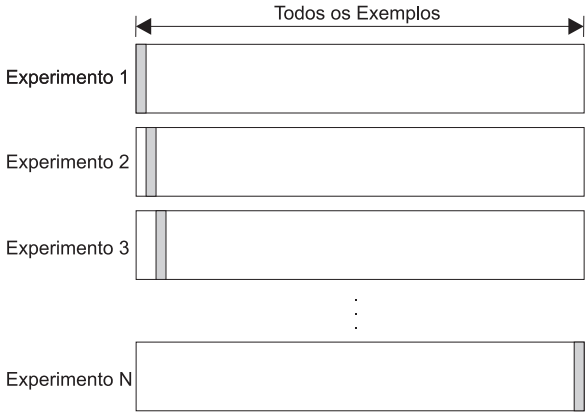

Figura 2.7: *Leave-one-out Cross Validation*

A técnica *Leave-one-out Cross Validation* é um caso particular da *K-Fold Cross validation*, quando K é escolhido como o número total de elementos do nosso conjunto de dados. Para um conjunto de dados com N elementos, fazemos N experimentos, usando $N - 1$ elementos para treino e apenas um para teste.