6
Aprendizado Online

Neste capítulo, algoritmos de aprendizado para redes Bayesianas são considerados no contexto de aprendizado *online*. Particularmente, é avaliado o desempenho desse tipo de algoritmo para estimar e caracterizar problemas de desempenho em sistemas distribuídos. Ambientes distribuídos possuem várias características para as quais um sistema de aprendizado *online* pode ser extremamente benéfico. Dentre tais características destacam-se o fluxo contínuo de dados e o dinamismo do ambiente de execução.

6.1
Motivação para o uso de aprendizado online

O aprendizado humano pode ser compreendido como um processo gradual de formação de conceitos. Nesse processo, o conhecimento proveniente de novas experiências é incorporado a estruturas de conceitos já aprendidas previamente. A observação desse fato deu origem aos sistemas de aprendizado *online*. Atualmente, existem diversas situações que podem tirar proveito desse tipo de aprendizado. Corporações armazenam milhares de novos registros diariamente, instituições financeiras registram milhares de novas transações e mecanismos de busca na Internet indexam milhares de novos conteúdos. Os sistemas de aprendizado tradicionais, ou seja, *offline* ou *batch*, apresentam dificuldades de incorporar esse fluxo contínuo de novos dados a bases de conhecimento já estabelecidas. Essa dificuldade é ainda maior quando tal incorporação deve ser feita de forma rápida e eficiente.

Tipicamente, os algoritmos de aprendizado *offline* examinam um grande volume de dados uma única vez e induzem um único modelo de predição, o qual não é mais revisado por novos dados. Nesses sistemas, a incorporação de novos dados requer a reconstrução do modelo de previsão a partir do processamento de toda a base de dados. Portanto, esse processo é lento e ineficiente.

Em resumo, muitas situações do mundo real apresentam fortes restrições que afetam o sistema de aprendizado. Em geral, podemos classificar essas restrições em três categorias principais, como descrito a seguir.

1. Restrição de recursos: muitas aplicações possuem fortes restrições de

tempo de processamento ou limitações de armazenamento, especialmente memória.

2. Fluxo contínuo de dados: em vários sistemas, dados provenientes de monitoramento são organizados como um fluxo contínuo de dados e não como um repositório estático de informações.

3. Ambientes que mudam com frequência: quando sistemas de aprendizado são implantados em ambientes dinâmicos, as mudanças no ambiente afetam os conceitos aprendidos. Nessas condições, os sistemas de aprendizado deveriam evoluir seus modelos ao longo do tempo.

Sistemas de aprendizado *online* podem ser a resposta a essas restrições. Uma das definições mais amplamente aceitas sobre sistemas de aprendizado dessa natureza foi estabelecida por Langley (Langley, 1995). Segundo essa definição, um algoritmo de aprendizado *online* é aquele que: *i*) processa dados à medida que os mesmos chegam no sistema; *ii*) não reprocessa experiências passadas e *iii*) retem apenas uma estrutura de conhecimento em memória.

É importante observar que essa definição impõe três restrições para que um sistema de aprendizado seja considerado *online*. A primeira restrição estabelece que os algoritmos de aprendizado *online* devem ser capazes de usar o conhecimento adquirido em qualquer instante durante o processo de aprendizado, e não apenas depois do processamento completo dos dados. A segunda restrição é colocada para manter baixo e constante o tempo requerido para processar cada novo dado sobre o restante dos dados. Finalmente, a terceira restrição visa assegurar o uso eficiente de memória.

Para o caso particular de determinação de problemas em sistemas distribuídos, dois fatores motivam o uso de algoritmos de aprendizado *online*. O primeiro fator refere-se aos serviços de monitoramento, os quais coletam dados sobre as aplicações constantemente. Isso permite atualizar os modelos de predição de problemas à luz desses novos dados. Adicionalmente, é importante que tal atualização seja feita de forma rápida e com pouco consumo de memória. O segundo fator relaciona-se com o ambiente de execução, cujas características podem variar frequentemente em sistemas distribuídos. Algoritmos de aprendizado *online* possuem maior capacidade de se ajustarem a mudanças, pois os modelos de predição são atualizados frequentemente.

Existem atualmente duas vertentes para a construção de algoritmos de aprendizado *online* (Schmitt et al., 2008). O primeiro grupo, denominado *algoritmos incrementais* ou *aprendizado incremental*, refere-se a um conjunto de soluções que adaptam técnicas de aprendizado *offline* para serem aplicadas

incrementalmente. Duas estruturas são fundamentais nesse processo: um algoritmo de aprendizado iterativo que permita a atualização do modelo existente à luz dos novos dados e um mecanismo de gerenciamento de instâncias que represente a experiência passada. Na literatura, algoritmos incrementais foram propostos para árvores de decisão (Utgoff, 1989), redes Bayesiana (Castillo e Gama, 2009) e SVM (Zhang, 2000).

A segunda vertente para a construção de sistemas de aprendizado *online* refere-se a uma área de pesquisa própria denominada *algoritmos online*. Nesse tipo de algoritmo, o aprendizado a partir de exemplos assume a forma de uma sequência consecutiva de tentativas. A cada tentativa, o módulo preditor faz uma estimação e, então, recebe um realimentação. Dessa forma, treinamento e teste acontecem simultaneamente. Dois algoritmos *online* amplamente referenciados na literatura são o STAGGER (Elfeky et al., 2006) e o Winnow (Littlestone, 1988).

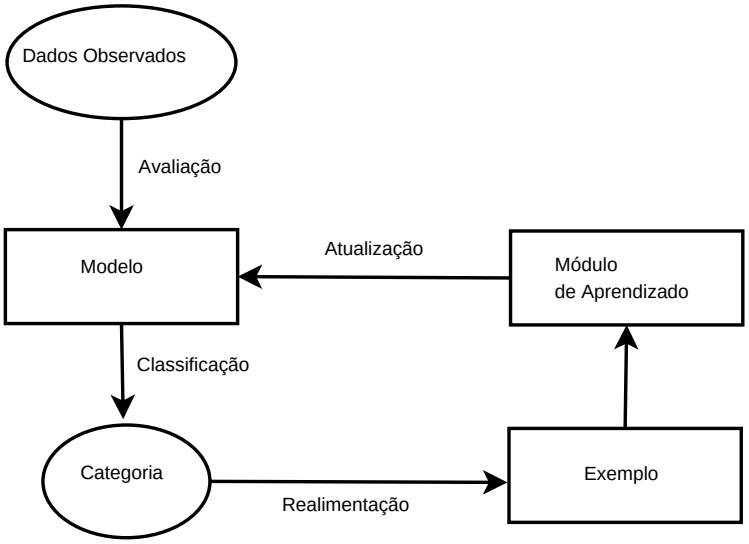
Uma característica essencial de algoritmos *online* puros é a ausência de gerenciamento e armazenamento de instâncias. Devido a esse comportamento, algoritmos *online* são tipicamente menos precisos que algoritmos incrementais (Schmitt et al., 2008). Por esse motivo, neste trabalho, optamos pela investigação de algoritmos incrementais. Apresentamos, na seção seguinte, um algoritmo de aprendizado incremental para um classificador Naive Bayes.

6.2 Aprendizado incremental

No cenário de aprendizado incremental, dados chegam no sistema em pequenos lotes de tamanhos iguais. Para ilustrar o comportamento de um sistema de aprendizado incremental, considere o ambiente da Figura 6.1. Nesse sistema, dados chegam em lotes B, cada um contendo m exemplos. O objetivo do sistema de aprendizado é estimar o rótulo de cada exemplo contido em um lote. Para isso, no tempo t_i, o lote B_i é avaliado usando a hipótese $h_C^{(i-1)}$, induzida usando os $i-1$ lotes anteriores. Quando os rótulos corretos de B_i se tornam disponíveis, os exemplos e os rótulos corretos desse lote são usados para atualizar a hipótese corrente. O objetivo do aprendizado é minimizar a quantidade de instâncias classificadas incorretamente em relação ao número de instâncias classificadas. O algoritmo básico para aprendizado supervisionado num cenário de aprendizado *online* é mostrado no Algoritmo 6.1 (Dawid, 1984).

Para construir a hipótese h_C do Algoritmo 6.1, diferentes algoritmos de aprendizado *offline* podem ser adaptados. Particularmente, este trabalho optou por adaptar o classificador Naive Bayes. Essa escolha foi motivada por dois fatores: *i*) o bom compromisso entre a acurácia e o custo computacional

Figura 6.1: Um sistema de aprendizado supervisionado incremental.



Algoritmo 6.1 Algoritmo básico para aprendizado supervisionado e incremental.

Require: um classificador h_C inicial, um *dataset* D de exemplos rotulados $< x, c >$, dividido em lotes B de m exemplos, a função $I(x, y)$ que retorna 0 se $x = y$ e 1 se $x \neq y$.

Ensure: o classificador h_C atualizado e a taxa de erro $errRate$

1: **for all** batch B in D **do**
2: **for all** example x in B **do**
3: $h_C(x) \leftarrow predict(x, h_C)$
4: $f(x) \leftarrow getActualClass(x)$
5: $incorrected \leftarrow incorrected + I(f(x), h_C(x))$
6: **end for**
7: $totalEvaluated \leftarrow totalEvaluated + m$
8: $errRate \leftarrow incorrected/totalEvaluated$
9: $update(h_C, B)$
10: **end for**
11: **return** h_C and $errRate$

apresentado pelo classificador ao estimar problemas de desempenho e *ii*) a natureza inerentemente iterativa do algoritmo.

O classificador Naive Bayes é simples e eficiente. Apesar da suposição de independência, o classificador Naive Bayes alcança desempenho comparável aos classificadores mais sofisticados, como foi relatado em diversos trabalhos na literatura e confirmado nos experimentos do Capítulo 4 deste trabalho. Além disso, a natureza desse classificador é inerentemente iterativa. Ou seja, é possível revisar o classificador Naive Bayes à luz de novos dados, sem descartar o modelo corrente e sem reprocessar dados anteriores.

Dada a suposição de independência, estimar o rótulo c, $c \in \Omega_C$, de um

exemplo $\mathbf{x} = \{x_1, x_2, ..., x_n\}$ consiste em calcular $c^* = \arg\max_j P(c_j|\mathbf{x})$, onde

$$P(c_j|\mathbf{x}) = \prod_{i=1}^{n} P(X_i = x_i|c_j)P(c_j). \quad (6\text{-}1)$$

Considerando o domínio $\mathbf{X} = \{X_1, X_2, ..., X_n\}$ de atributos discretos, cada termo da Equação 6-1 pode ser calculado eficientemente. Seja $\Omega_{Xi} = \{x_i^1, x_i^2, ..., x_i^{ri}\}$ o domínio de um atributo X_i, onde ri representa o número de valores possíveis de X_i. Para estimar $P(c_j)$ para cada classe $c_j \in \Omega_C$ e computar $P(X_i = x_i^k|c_j)$ para cada valor $x_i^k \in \Omega_{Xi}$, basta manter:

- um vetor contendo l contadores N_j, sendo um contador para cada rótulo c_j.
- Para cada atributo X_i, manter uma tabela de contingência $l \times r_i$ contendo os contadores N_{ijk}. Cada N_{ijk} é o número de exemplos em D, tal que $X_i = x_i^k$ e $C = c_j$.

Note que, devido à suposição de independência, a estrutura da rede é fixa e conhecida *a priori*. Dessa forma, aprender um classificador Naive Bayes incrementalmente se torna incrementar os contadores N_j e N_{ijk} à medida que novas instâncias de dados são processadas.

Apesar da natureza iterativa do classificador Naive Bayes, uma questão ainda persiste como um problema para seu uso em um cenário de aprendizado incremental. O algoritmo é projetado para operar sobre dados discretos. Contudo, como discutido no Capítulo 4, o domínio das variáveis ou atributos observados no problema proposto neste trabalho são, em sua maioria, contínuos.

No cenário de aprendizado *offline* apresentado no Capítulo 4, esse problema foi resolvido usando normalização e discretização. Para cada atributo, o valor do atributo em cada exemplo do *dataset* foi dividido pelo maior valor do atributo encontrado no *dataset*. Essa solução, no entanto, não pode ser aplicada em um cenário de aprendizado *online*, visto que, nessa situação, não temos todo o conjunto de dados previamente. Para contornar essa restrição, utilizamos uma técnica de discretização *online*, como descrito a seguir.

6.2.1 PID - Partition Incremental Discretization

Gama e Pinto propuseram um novo método para discretização incremental denominado *Partition Incremental Discretization* ou simplesmente PID (Gama e Pinto, 2006). O objetivo principal desse método é a atualização dos intervalos de uma discretização à medida que novos dados se tornam disponíveis, sem, contudo, armazenar todos os valores vistos até o instante em questão.

O PID é um algoritmo para manter e atualizar um histograma sobre um fluxo de dados contínuo. Um histograma, por sua vez, é um conjunto de intervalos, cada um dos quais definido por seus limites e um contador de frequência. Para construir o histograma, o PID define duas camadas. A primeira camada simplifica e sumariza os dados, enquanto a segunda constrói o histograma final.

Para ilustrar melhor essa idéia, considere a construção de um histograma para uma variável aleatória contínua que usa o algoritmo PID. A primeira camada é inicializada com o número de intervalos para o histograma e o intervalo de valores possíveis para a variável aleatória. Aqui são importantes duas ressalvas: *i*) o número de intervalos fornecido para a primeira camada deve ser maior que a quantidade de intervalos realmente desejada para o histograma final e *ii*) o intervalo de valores possíveis para a variável aleatória é apenas um indicativo inicial. Esse valor é usado para inicializar os limites dos intervalos do histograma, usando uma estratégia de intervalos de largura fixa.

O algoritmo 6.2 ilustra o funcionamento da primeira camada do PID. Sempre que um valor para a variável aleatória é observado, a primeira camada é atualizada. Isso significa que o intervalo correspondente ao valor observado é determinado e o contador associado a esse intervalo é incrementado. Quando o contador de um intervalo extrapola um limiar, como, por exemplo, uma porcentagem do número total de exemplos vistos até o momento, uma operação de *particionamento* é executada, gerando novos intervalos na primeira camada. Se o intervalo que acionou o particionamento for o primeiro ou o último, um novo intervalo de mesma largura é inserido. Nos demais casos, o intervalo acionador é particionado em dois. Observe que esse processo é *online* e que cada exemplo é processado em tempo e espaço constantes.

Na segunda camada, um histograma com um número fixo de intervalos é criado a partir da combinação dos intervalos do histograma da primeira camada. Para tanto, a segunda camada recebe o número de intervalos desejado para o histograma final e os intervalos e contadores do histograma da primeira camada. O algoritmo para a segunda camada é muito simples. Considerando uma estratégia de intervalos com larguras fixas, primeiramente os limites de cada intervalo no histograma final são computados usando o intervalo de valores possíveis para a variável aleatória. Esses intervalos são estimados usando os intervalos inicial e final do histograma da primeira camada. Em seguida, o algoritmo da segunda camada percorre o vetor de intervalos da primeira, ajustando os contadores desses intervalos para caberem dentro de seus próprios intervalos.

Uma questão importante no PID é determinar quando a segunda camada

Algoritmo 6.2 Funcionamento da primeira camada no PID.

Require: x: variável aleatória observada, $breaks$: vetor de intervalos da discretização, $counts$: vetor de frequência, NrB: número de intervalos, α: limiar para particionar um intervalo, Nr: número de valores observados, $step$: distância entre os intervalos.

Ensure: $split = true$: se ocorreu particionamento; $split = false$: caso contrário

$k \leftarrow find\_interval(breaks, x)$;
$counts[k] \leftarrow 1 + counts[k]$;
$Nr \leftarrow Nr + 1$;
$split \leftarrow false$;
if $(1 + counts[k])/(Nr + 2) > \alpha$ **then**
  $val \leftarrow counts[k]/2$;
  $counts[k] \leftarrow val$;
  if $k == 1$ **then**
    $breaks \leftarrow append(breaks[1] - step, breaks)$;
    $counts \leftarrow append(val, counts)$;
  else if $k == NrB$ **then**
    $breaks \leftarrow append(breaks, breaks[NrB] + step)$;
    $counts \leftarrow append(counts, val)$;
  else
    $breaks \leftarrow insert((breaks[k] + breaks[k + 1])/2, breaks, k)$;
    $counts \leftarrow insert(val, counts, k)$;
  end if
  $NrB \leftarrow NrB + 1$;
  $split \leftarrow true$;
end if
return *split*

é atualizada. Quando um novo exemplo chega ao sistema, a primeira camada é atualizada usando o Algoritmo 6.2. É pouco provável que uma única observação irá alterar a distribuição na segunda camada. Logo, nessa situação, apenas o contador do intervalo correspondente é atualizado na segunda camada. Por outro lado, quando o conjunto formado pelos limites dos intervalos do histograma da primeira camada é alterado, a segunda camada deve ser reconstruída. Isso ocorre sempre que a primeira camada executa uma operação de particionamento.

Em resumo, na arquitetura em camadas do PID, a primeira camada sumariza os dados vistos até o momento, enquanto a segunda camada provê efetivamente a discretização. A vantagem dessa arquitetura está no fato de, após gerar o histograma da primeira camada, o custo computacional para gerar o histograma final é baixo e depende apenas do número de intervalos na primeira camada.

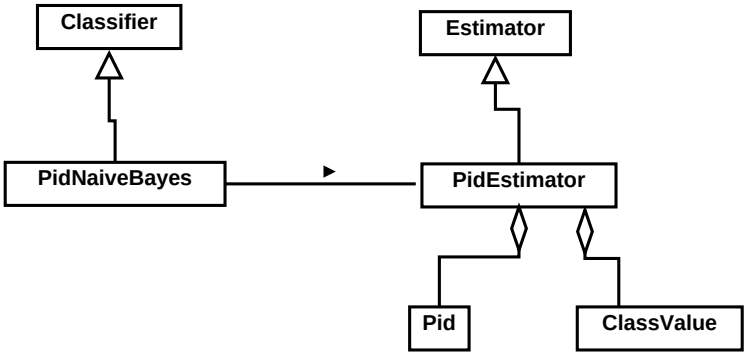
6.2.2 PID Naive Bayes

Aplicar o algoritmo PID para um cenário de aprendizado supervisionado requer poucas adaptações. Em cada camada do algoritmo, o vetor de frequência associado ao histograma é substituído por uma matriz que conterá a frequência dos intervalos por classe. Ou seja, em cada camada, substitui-se o vetor de frequência do histograma por uma matriz de distribuição. Cada linha nessa matriz corresponde a uma classe e cada coluna corresponde a um intervalo do histograma. Para atualizar a matriz de distribuição na primeira camada, deve-se fornecer, além do valor da variável aleatória, o valor da variável de classe. Se a atualização na primeira camada não gerar um particionamento, a matriz de distribuição na segunda camada é apenas atualizada. Nesse caso, a célula, determinada pela classe e o intervalo correspondente na segunda camada, é incrementada. Por outro lado, se ocorrer um particionamento na primeira camada, a matriz de distribuição da segunda camada é reconstruída para refletir a alteração dos novos intervalos.

Usando essa estrutura de dados, a implementação do algoritmo Naive Bayes da ferramenta Weka foi adaptada para operar em cenários que envolvam: *i*) aprendizado supervisionado e incremental, e *ii*) variáveis ou atributos de domínio contínuo. Esse algoritmo foi denominado PID Naive Bayes. A Figura 6.2 mostra o diagrama de classes que implementa esse algoritmo. A classe *PidNaiveBayes* representa o novo classificador. Como todo classificador no Weka, essa classe estende a classe *Classifier*, sobrescrevendo os métodos *buildClassifier* e *getProbabilityDistribution*. A classe *Pid* representa a estrutura de dados que implementa o algoritmo Pid, estendido para contemplar as matrizes de distribuição. É importante observar que a classe *Pid* representa a distribuição de valores de um único atributo.

Na implementação original do classificador Naive Bayes no Weka, a Equação 6-1 é calculada a partir da matriz de distribuição dos valores dos atributos e a matriz de distribuição dos valores da variável de classe. Cada célula na matriz de distribuição dos atributos é um objeto da classe *Estimator*. Um atributo numérico é associado a um estimador que assume uma distribuição normal para os valores do atributo (*NormalEstimator*). Por outro lado, um atributo nominal é vinculado a um estimador discreto (*DiscreteEstimator*). No classificador *PidNaiveBayes* o estimador *NormalEstimator* é substituído pela classe *PidEstimator*, a qual usa a classe *Pid* para estimar a distribuição dos valores de um atributo numérico.

Figura 6.2: Classes usadas na implementação do classificador Pid Naive Bayes.



6.2.3 Experimentos

Nesta parte do trabalho, é avaliado o desempenho do algoritmo PID Naive Bayes. Para todos os experimentos, supõe-se que os dados chegam no sistema em lotes contendo um único exemplo, ou seja, assume-se $m = 1$.

Avaliação do Nível de Discretização

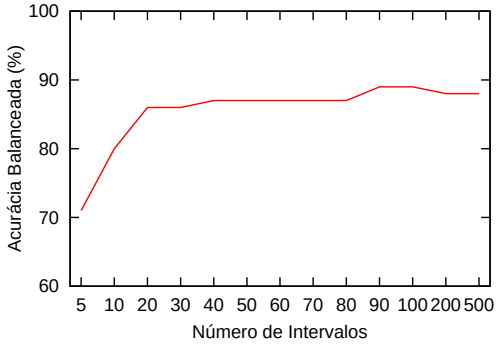
De maneira análoga aos experimentos do Capítulo 4, os experimentos com o algoritmo Pid Naive Bayes iniciam avaliando o impacto do nível de discretização na acurácia do algoritmo. Para tanto, o número de intervalos na segunda camada varia de 5 a 500. Na primeira camada, o número de intervalos foi configurado com o valor 200 e o limiar para particionamento, ou seja α, foi definido como 30%. Dessa forma, sempre que um contador, na primeira camada, extrapola 30% dos exemplos vistos até o momento, o intervalo é particionado. A Figura 6.3(a) mostra como a acurácia do algoritmo Pid Naive Bayes varia em função do número de intervalos. A acurácia foi avaliada usando o *dataset MapReduce*, em que aproximadamente 69000 exemplos foram usados para treinamento e 10000 para teste. O SLO foi definido como o 80º percentil de *avg_response_time*. Como acontece com os algoritmos de aprendizado *offline*, é possível notar que a acurácia do algoritmo Pid Naive Bayes aumenta à medida que o número de intervalos aumenta. A partir de 20 intervalos, a acurácia se estabiliza e um outro aumento só é observado em torno de 90 intervalos. A partir desse valor, um aumento no número de intervalos não implica em um aumento significativo na acurácia.

Ainda em relação ao nível de discretização, a Figura 6.3(b) compara o comportamento do algoritmo Pid Naive Bayes com o comportamento dos algoritmos *offline*. Para efeito de comparação, foram considerados os mesmos intervalos e os mesmos *datasets* de treinamento e teste para todos os algoritmos.

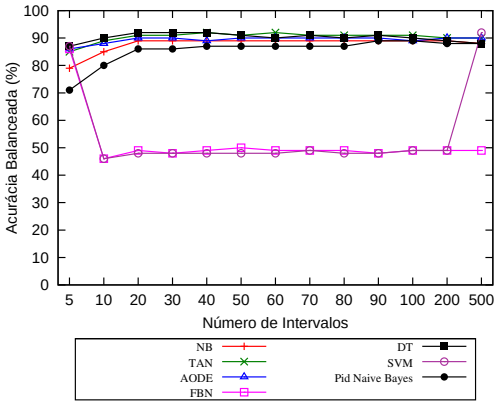
É possível notar que a acurácia do classificador Pid Naive Bayes mantém-se muito próxima à acurácia da maioria dos classificadores *offline*. É importante ressaltar, no entanto, que a técnica de treinamento utilizada para todos os algoritmos desse experimento, isto é, a divisão do *dataset* em arquivos de treinamento e teste, difere da técnica utilizada no experimento correspondente no Capítulo 4, onde foi utilizada a técnica *10-fold cross validation*. Esse fato explica porque a acurácia obtida pelos classificadores nesse experimento não é a mesma obtida no experimento correspondente no Capítulo 4.

Figura 6.3: Comportamento do algoritmo Pid Naive Bayes em relação ao número de intervalos da discretização e comparação desse comportamento com o comportamento dos algoritmos *offline*.

(a) Acurácia vs. número de intervalos.



(b) Pid Naive Bayes vs. algoritmos *offline*.



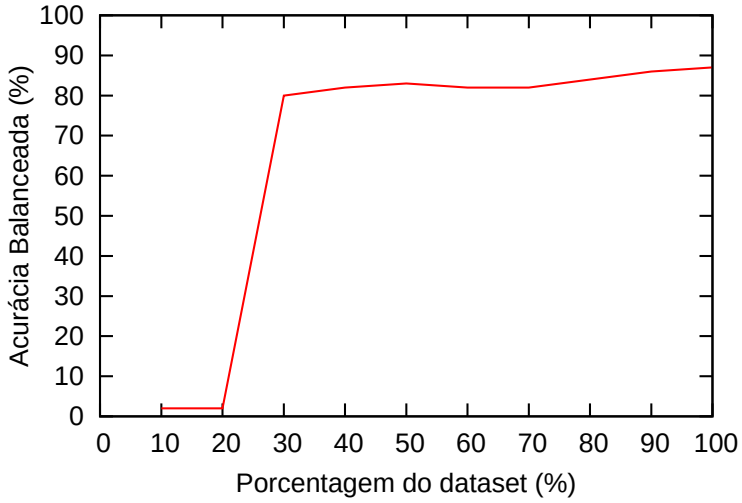
Avaliação do Desempenho

Na Figura 6.4, é exibida a acurácia do algoritmo Pid Naive Bayes ao longo de uma sucessão de testes e treinamentos sobre os *datasets* do experimento anterior. Para esse experimento, foram considerados um nível de discretização de 40 intervalos e um SLO definido como o 80º percentil de *avg_response_time*.

Nesse experimento, a acurácia é avaliada pela primeira vez quando 10% do *dataset* de treinamento foi apresentado ao algoritmo. A acurácia é medida novamente quando 20% do *dataset* foi processado. Esse processo se repete com um passo de 10% até que 100% do *dataset* de treinamento seja processado. Como esperado para um processo de aprendizado *online*, a acurácia do classificador evolui ao longo do tempo. No inicio do processo, o desempenho do algoritmo é muito baixo. Com 10% do *dataset*, a acurácia do classificador Pid Naive Bayes é de apenas 2%. Essa acurácia se mantém quando 20% do *dataset* é processado. Com 30% do *dataset* processado, o classificador atinge

uma acurácia de aproximadamente 80%. O classificador termina o processo com uma acurácia de 87%.

Figura 6.4: Comportamento do algoritmo Pid Naive Bayes ao longo do processo de aprendizado.



Uma avaliação mais detalhada do desempenho do algoritmo Pid Naive Bayes é apresentada na Tabela 6.1. As colunas dessa tabela refletem a taxa de detecção e a taxa de alarme falso para o estado violação, a acurácia balanceada e o tempo requerido para treinamento. Para esse experimento, foram utilizados cinco *datasets* de treinamento, cada um gerado a partir do *dataset* de treinamento do experimento anterior, e uma das definições de SLO (50º, 60º, 70º, 80ºe 90ºpercentis). Em todos os *datasets*, considerou-se um nível de discretização de 40 intervalos. Cada *dataset* de treinamento induziu um modelo o qual foi aplicado na classificação do *dataset* de teste, também do experimento anterior. As colunas da Tabela 6.1 representam a média dos valores obtidos pelo algoritmo Pid Naive Bayes nos cinco *dataset*. A tabela também mostra o desempenho do classificador Naive Bayes Updateable e dos algoritmos de aprendizado *offline*, sobre os mesmos *datasets* de treinamento e teste. O classificador Naive Bayes Updateable é a versão *online* do classificador Naive Bayes disponível no Weka. Ele difere do algoritmo Pid Naive Bayes por assumir uma distribuição normal para os atributos contínuos. A Tabela 6.1 mostra que o classificador Pid Naive Bayes atinge um desempenho melhor que o algoritmo Naive Bayes Updateable, tanto em relação à taxa de detecção quanto em relação à acurácia. O tempo requerido para treinamento do classificador Pid Naive Bayes é levemente maior, aproximadamente 0.3 segundos a mais. É possível notar também que o desempenho do classificador é praticamente o mesmo do classificar Naive Bayes em sua versão *offline*.

Tabela 6.1: Comparação do desempenho do algoritmo Pid Naive Bayes em relação ao desempenho dos algoritmos *offline* e o algoritmo Naive Bayes Updateable.

	Taxa de Detecção	**Alarme Falso**	**Acurácia**	**Tempo Treinamento**
Pid NB	76.6%	1.6%	85.4%	0.7s
NB Updateable	42.2%	1.8%	77.4%	0.4s
NB	80.0%	1.6%	85.8%	0.4s
TAN	75.4%	0.6%	89.8%	19s
AODE	79.6%	0.8%	87.8%	0.5s
FBN	75.0%	88.0%	46.2%	19s
DT	74.0%	0.6%	90.2%	19s
SVM	72.8%	71.2%	53.8%	2200s

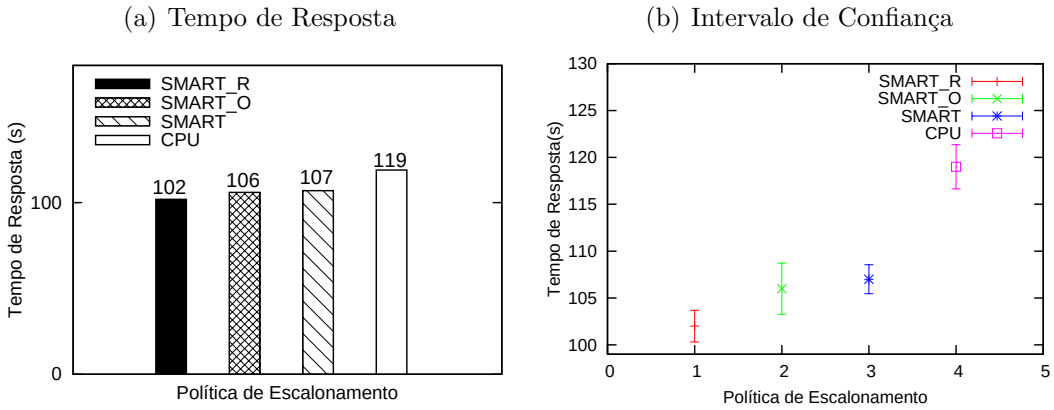
Avaliação do Desempenho do SMART

Como em outros algoritmos avaliados nos Capítulos 4 e 5, o algoritmo PID Naive Bayes foi implementado como a máquina de inferência do SMART. O intuito é verificar a efetividade do algoritmo na função de classificador do SMART. Para tanto, o experimento da Seção 4.6.6 foi repetido usando os mesmos nós do *cluster*, o mesmo cenário de carga e a mesma configuração da aplicação MapReduce daquele experimento. Como na Seção 4.6.6, o SLO foi definido como o 80ºpercentil de *avg_response_time*.

Nas condições descritas acima, o desempenho da aplicação MapReduce foi comparado em quatro situações: usando a política de escalonamento baseada em CPU (CPU); usando a política baseada em probabilidade de violação, sendo a probabilidade obtida pelo uso exclusivo do classificador TAN (SMART); usando a política baseada em probabilidade de violação, sendo a probabilidade obtida pelo algoritmo de alerta de problemas de desempenho (SMART_R); e usando a política baseada em probabilidade de violação, sendo a probabilidade obtida pelo algoritmo Pid Naive Bayes (SMART_O). Novamente, o desempenho da aplicação é medido considerando o tempo total de execução. Para fins de comparação, considera-se a política que proporciona o melhor desempenho, aquela que resulta em um menor tempo de execução da aplicação. A Figura 6.5(a) mostra a média do tempo total de execução da aplicação MapReduce, em 10 execuções, para cada política de escalonamento, usando um nível de 95% no intervalo de confiança. Como pode ser visto na Figura 6.5(b), não há uma diferença significativa entre os desempenhos proporcionados pelos algoritmos TAN e Pid Naive Bayes. Uma explicação para esse resultado pode ser a carga aplicada ao sistema durante o experimento, a qual é razoavelmente similar a uma das cargas percebidas durante o treinamento.

Isso torna o ambiente de execução mais previsível, favorecendo o algoritmo de aprendizado *offline*.

Figura 6.5: Desempenho da aplicação MapReduce usando o algoritmo Pid Naive Bayes.



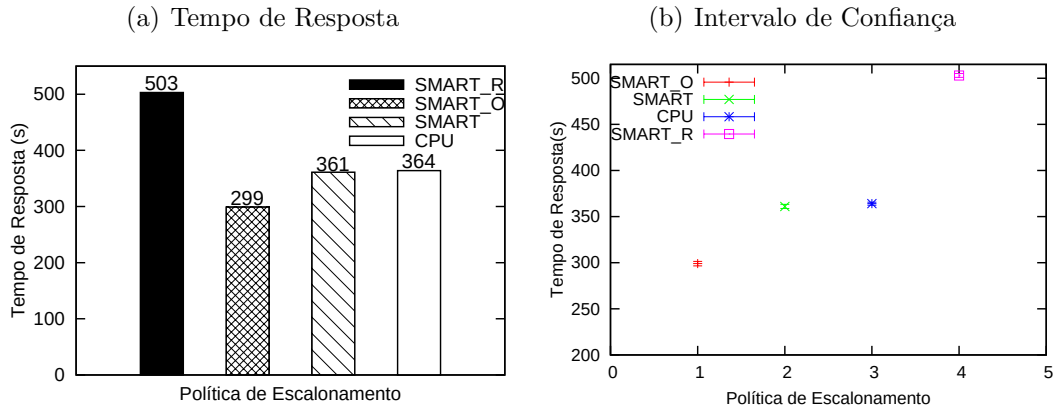
Para verificar se um cenário de execução diferente dos encontrados durante o treinamento dos algoritmos de aprendizado *offline* favoreceria o algoritmo Pid Naive Bayes, duas alterações foram propostas no experimento acima. Primeiramente, a configuração da aplicação MapReduce foi alterada para usar 40 componentes *Workers*, diferentemente dos 20 componentes usados nos experimentos anteriores. O cenário de carga aplicado ao sistema durante o experimento também foi alterado. A nova carga exercitou apenas a CPU e a intensidade aplicada ao recurso variou de forma senoidal, como descrito a seguir.

Até atingir o pico, a onda da senóide é dividida em 5 intervalos de intensidade de carga: 0-20%, 20-40%, 40-60%, 60-80% e de 80-100%. A intensidade de carga começa no intervalo de 0 a 20% de utilização de CPU e permanece neste estado por 30 segundos. Em seguida, a intensidade de carga é aumentada, fazendo a onda transitar para o intervalo de 20 a 40% de utilização de CPU. Novamente, a intensidade de carga permanece nesse estado por 30 segundos, transitando para o próximo intervalo ao final desse período. Esses passos são repetidos até que o último intervalo (80-100%) é atingido. Após permanecer no quinto intervalo por 30 segundos, a intensidade de carga começa a diminuir. Nesse momento, a onda transita para os intervalos de utilização de CPU seguindo o caminho inverso (decrescente), trocando de intervalo a cada 30 segundos. Esse padrão de carga foi aplicado aos nós d_6, d_7, d_8 e d_9. A Tabela 6.2 resume as configurações usadas neste novo experimento.

Tabela 6.2: Configuração usada no experimento para avaliar o desempenho da aplicação MapReduce usando o algoritmo Pid Naive Bayes, considerando o cenário de carga senoidal.

Configuração	**Descrição**
Nós utilizados	d_1 a d_{11}
Injeção de carga	d_6 a d_9
Número de *Workers*	40
SLO	80º percentil de *avg_response_time*

Figura 6.6: Desempenho da aplicação MapReduce usando o algoritmo Pid Naive Bayes, num cenário diferente dos percebidos no treinamento.



O resultado desse novo experimento é mostrado nas Figuras 6.6(a) e 6.6(b). Novamente, cada tempo de execução corresponde ao valor médio obtido em 10 execuções, usando um nível de 95% no intervalo de confiança. Como pode ser visto, houve uma diferenciação significativa dos desempenhos alcançados pelos algoritmos avaliados. Como esperado, a política de escalonamento baseada no algoritmo Pid Naive Bayes obteve o melhor desempenho, mostrando que algoritmos de aprendizado *online* são mais apropriados para lidar com situações não presenciadas anteriormente. O pior desempenho foi registrado pela política baseada no algoritmo de alerta. Esse resultado pode ser explicado pelo fato desse algoritmo possuir um comportamento mais conservador em relação a mudanças. Como o algoritmo de alerta busca atenuar alarmes falsos, o algoritmo leva mais tempo para perceber mudanças no ambiente. A política baseada no classificador TAN e a baseada em CPU obtiveram desempenho semelhantes. Esse resultado é explicado pelo fato da carga injetada no sistema se basear exclusivamente no uso de CPU. A Tabela 6.3 mostra a acurácia alcançada pelos algoritmos Pid Naive Bayes e TAN, durante a

execução da aplicação *WordCount* neste experimento.

Tabela 6.3: Acurácia de TAN e PID Naive Bayes durante o experimento

Classificador	**Acurácia**
TAN	69%
Pid Naive Bayes	80%

6.3 Gerenciamento de concept drift

Em sistemas distribuídos, um serviço de monitoramento pode coletar dados por períodos de tempo longos. Nessa situação, o ambiente de execução, o qual é o alvo do sistema de aprendizado, pode sofrer grandes modificações. Novos nós podem entrar ou deixar o sistema e a disponibilidade de recursos de hardware e software para um serviço pode variar ao longo do tempo. Mudanças como essas podem afetar os conceitos que se pretende aprender sobre um ambiente, tornando-os instáveis. Para sistemas de classificação que tentam aprender uma função a partir de amostras de entradas e saídas, esse problema se manifesta na forma de mudanças de função ao longo do tempo e é conhecido como *concept drift*. Cenários de *concept drift* requerem algoritmos de aprendizado capazes de detectar mudanças de conceito e se adaptarem rapidamente a elas.

Nos últimos anos, muitos trabalhos têm investigado problemas de aprendizado do mundo real que são afetados por *concept drift* (Schmitt et al., 2008; Castillo e Gama, 2009). Dentre esses, destacam-se trabalhos nas áreas de filtragem de informações, monitoramento de redes e modelagem de preferências de usuários. A ideia básica que permeia todos esses trabalhos é que em ambientes dinâmicos, dados recentes são mais importantes que dados antigos. Logo, hipóteses baseadas em exemplos antigos devem ser descartadas. Por esse motivo, algoritmos de aprendizado que consideram *concept drift* devem prover alguma estratégia para decidir quando uma hipótese deve de fato ser descartada.

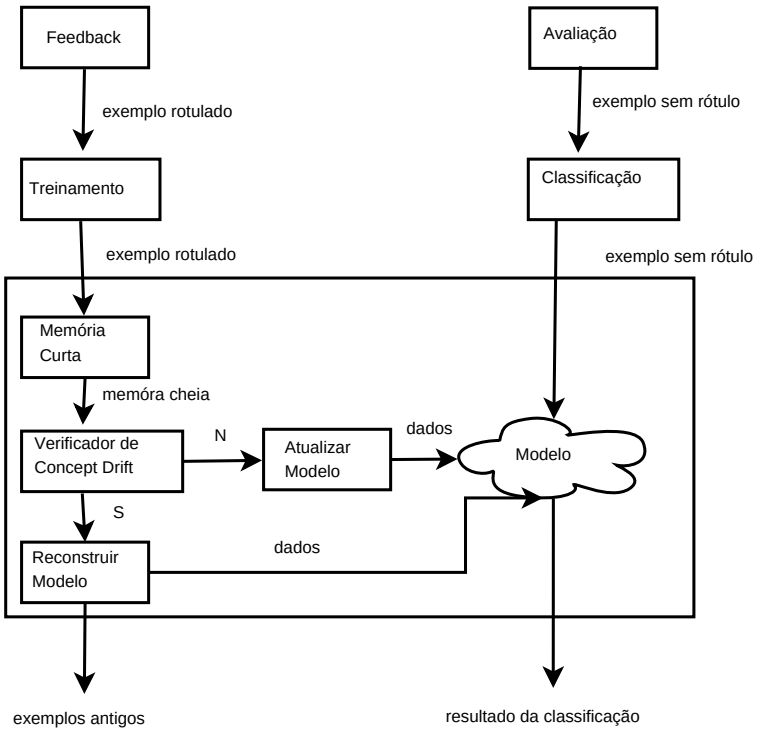
A seguir, é apresentada uma adaptação feita no algoritmo Pid Naive Bayes para operar em cenários que envolvam *concept drift*. Esse novo algoritmo foi denominado *Concept Drift Handler Pid Naive Bayes* ou simplesmente CDHPid Naive Bayes.

6.3.1 CDHPid Naive Bayes

No procedimento original do algoritmo Pid Naive Bayes, para cada instância que chega ao sistema de aprendizado, o classificador primeiramente infere o rótulo do exemplo. Quando o rótulo real se torna disponível, o exemplo é usado para atualizar o modelo corrente.

A Figura 6.7 ilustra o funcionamento do algoritmo CDHPid Naive Bayes. Diferentemente de sua versão original, o algoritmo CDHPid Naive Bayes não atualiza o modelo corrente imediatamente com os novos dados. Como a detecção de *concept drift* requer a diferenciação entre instâncias recentes e antigas, ambos tipos de exemplos são mantidos separadamente. Uma memória curta mantém as instâncias recentes, enquanto o modelo corrente representa os exemplos antigos. Quando a memória curta enche, os exemplos contidos nessa memória são verificados, ou seja, eles passam por um detector de *concept drift*. Se esse detector decidir que os exemplos da memória curta referem-se aos mesmos conceitos aprendidos até o momento, o modelo atual é atualizado com os dados recentes. No entanto, se for detectado que os exemplos da memória curta correspondem a conceitos diferentes dos aprendidos até então, o modelo corrente é descartado e um novo modelo é construído a partir dos exemplos da memória curta.

Figura 6.7: Comportamento do algoritmo CDHPid Naive Bayes.



Um módulo central no algoritmo CDHPid Naive Bayes é o verificador de *concept drift*. Para esse verificador foi implementada a solução proposta por Castillo e Gama em (Castillo e Gama, 2009). Essa proposta consiste em usar um gráfico de controle, *P-chart*, para monitorar o comportamento da taxa de erro[1] das instâncias presentes na memória curta. A construção do gráfico de controle é detalhada a seguir.

6.3.2 Gráfico de controle

Gráficos de Controle são ferramentas usadas para determinar se um processo, originalmente de manufatura, está sob controle. Um processo é considerado sob controle se observações sucessivas de uma característica apresentam uma variação estável em torno de um valor. Esse valor é normalmente denominado *valor-p* e o gráfico que o controla recebe o nome de *P-chart*.

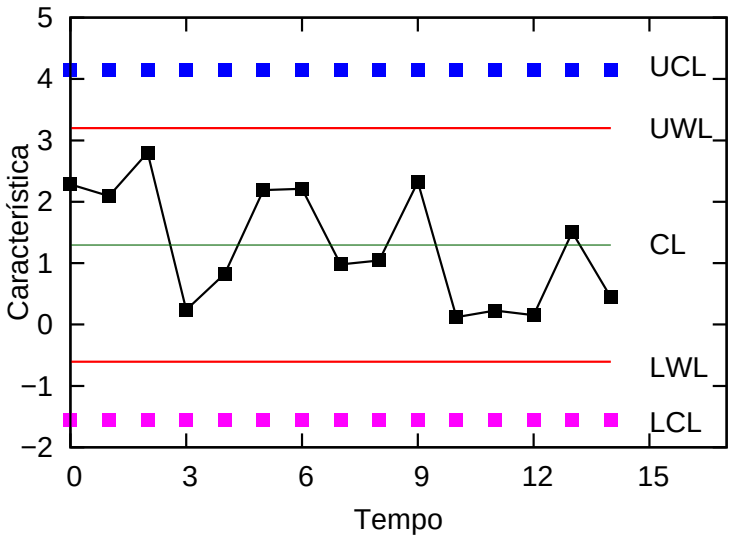
A Figura 6.8 mostra um gráfico de controle em que os valores observados para uma variável ou característica v são plotados em função do tempo. O gráfico possui uma linha central (CL) que representa o *valor-p*, um limite de controle inferior (LCL) e um limite superior (UCL). Os limites de controle são usados para indicar os limiares sob os quais o processo descrito é considerado estatisticamente sob controle. Pontos que caem fora desses limites representam mudanças estatisticamente significantes no processo. Se uma observação de v cai fora dos limites, o processo é considerado fora de controle.

Limiares de alerta também podem ser representados como linhas separadas. Esses limiares são definidos acima (UWL) e abaixo (LWL) da linha central e ajudam a aumentar a sensibilidade do mecanismo. Por exemplo, um certo número sucessivo de alertas pode ser uma indicação de que um processo está fora de controle.

Se o tamanho da amostra de observações é grande, ou seja, acima de 30 exemplos, a distribuição da característica v se aproxima de uma normal com média μ e variância σ^2. Nessas condições, aproximadamente 99,7% das observações cairão a uma distância de até 3σ da média. Assumindo μ e σ conhecidos, as linhas do gráfico de controle podem ser configuradas a partir desses valores como mostrado a seguir. Em geral, o valor 2 é usado para α na Equação 6-4.

[1] A taxa de erro é o complemento da acurácia, ou seja, $100 - Acuracia$

Figura 6.8: Um gráfico de controle típico.



$$\text{CL} = \mu \tag{6-2}$$

$$\text{LCL} = \text{CL} - 3\sigma; \quad \text{UCL} = \text{CL} + 3\sigma \tag{6-3}$$

$$\text{LWL} = \text{CL} - \alpha\sigma; \quad \text{UWL} = \text{CL} + \alpha\sigma; \; 0 < \alpha < 3. \tag{6-4}$$

Castillo e Gama propõem o uso de gráficos de controle para detectar *concept drift*. Conforme foi apresentado na Seção 6.2, um algoritmo básico de aprendizado supervisionado e incremental processa lotes de exemplos incrementalmente. A cada lote processado, uma taxa de erro Err_B é calculada. Essa taxa corresponde à proporção de instâncias classificadas incorretamente em um lote de m exemplos. Assumindo $m > 30$ e um cenário de aprendizado livre de *concept drift*, uma sequência de observações de Err_B, ou seja $p(t) = Err_B^{(t)}$, representa um processo aleatório discreto cuja variação oscila de forma estável em torno da média. Por outro lado, em cenários em que é possível a ocorrência de *concept drift*, um fluxo de dados pode ser considerado uma sequência de fases estáveis entre fases de instabilidade. Nesses cenários, a observação de um desvio significativo nos valores atual e anterior de Err_B é uma indicação de mudanças. Portanto, é possível detectar a ocorrência de *concept drift* monitorando a taxa de erro Err_B através de um gráfico de controle.

Apesar do *valor-p* ser normalmente associado à média (μ), Castillo e Gama propõem um calculo diferente para esse valor. Tendo como base o comportamento natural do próprio processo de aprendizado, o qual visa

minimizar a taxa de erro de um modelo (Err_S), os autores propõem configurar o *valor-p* com o menor valor de Err_S visto até o momento. Esse valor é representado por Err_{min}. Além disso, como uma taxa de erro menor é desejada, os autores não utilizam os limites inferiores (LCL e LWL) no gráfico de controle.

Os Algoritmos 6.3 e 6.4 descrevem o mecanismo para tratar *concept drift* proposto por Castillo e Gama. Sempre que um novo modelo é construído, Err_{min} é configurado para um valor grande. Para cada novo lote B, calcula-se $Err_B^{(t)}$ e $Err_S^{(t)}$. Se $Err_S^{(t)} + SErr_S^{(t)} < Err_{min}$, então Err_{min} é configurado com $Err_S^{(t)}$, onde $SErr_S^{(t)}$ é o desvio padrão da taxa de erro do modelo. Em seguida, CL é configurado com Err_{min} e as demais linhas do gráfico de controle são calculadas usando as Equações 6-3 e 6-4. Então, observa-se onde o valor $Err_B^{(t)}$ incide no gráfico de controle. Se esse valor cai acima de UCL, então o algoritmo sinaliza um *concept drift*. Se, pela primeira vez, $Err_B^{(t)}$ cai entre UCL e UWL, o algoritmo registra um alerta. Se essa situação ocorrer por duas ou mais vezes consecutivas, o algoritmo sinaliza um *concept drift*. Apenas se $Err_B^{(t)}$ cair abaixo de UWL assume-se que o processo está sob controle.

Algoritmo 6.3 Algoritmo CDHPid Naive Bayes para tratar concept drift

Require: h_c: modelo corrente, x_i: instância corrente, $short\_memory$: memória curta atual, m: número de instâncias em um lote, $P - chart$: gráfico de controle corrente.

Ensure: h_c atualizado

```
1: add x_i to short_memory;
2: if size(short_memory) == m then
3:    Err_B^(t) ← evaluate(h_c,short_memory);
4:    state ← getPchartState(Err_B^(t),P − chart); {Algoritmo 6.4}
5:    if state == CONCEPT_DRIFT then
6:       h_c ← build(short_memory);
7:    else
8:       h_c ← update(h_c, short_memory);
9:    end if
10: end if
11: return h_c
```

6.3.3 Experimentos

Esta seção apresenta uma avaliação da eficácia do algoritmo CDHPid Naive Bayes para tratar *concept drift*. Os experimentos usados na avaliação são divididos em três partes. Inicialmente, o algoritmo é avaliado em um *dataset* padrão para testes de algoritmos que tratam *concept drift*. Em seguida, o algoritmo é avaliado usando o *dataset MapReduce*. Finalmente, a eficácia do algoritmo proposto é analisada quando esse é implementado no SMART.

Algoritmo 6.4 Algoritmo para detectar concept_drift usando gráfico de controle

Require: $Err_B^{(t)}$: erro calculado para o último lote B, $P-chart$: gráfico de controle corrente, m: número de instâncias em um lote.
Ensure: estado corrente do sistema ($CONCEPT\_DRIFT$ ou $IN\_CONTROL$)

```
1: Add the new point Err_B^(t) to the P-Chart;
2: update Err_min;
3: CL ← Err_min;
4: σ ← sqrt(CL * (1 − CL)/m);
5: UCL ← CL + 3σ;
6: UWL ← CL + 2σ;
7: if Err_B^(t) > UCL then
8:    state ← CONCEPT_DRIFT;
9: else if Err_B^(t) > UWL then
10:   if lastAlert == t − 1 then
11:      state ← CONCEPT_DRIFT;
12:   else
13:      state ← IN_CONTROL
14:      lastAlert ← t
15:   end if
16: else
17:   state ← IN_CONTROL
18: end if
19: return state
```

Para os experimentos a seguir, o algoritmo CDHPid Naive Bayes foi implementado no Weka.

O dataset STAGGER

STAGGER é o *dataset* mais usado para avaliar o desempenho de classificadores que operam em ambientes dinâmicos. Proposto por Schlimmer e Granger (Schlimmer e Granger, 1986), esse dataset tem sido considerado um *benchmark* para problemas que envolvem *concept drift*.

No STAGGER, o espaço de característica é composto por três atributos: $size \in \{small, medium, large\}$, $color \in \{red, green, blue\}$ e $shape \in \{square, circular, triangular\}$. O *dataset* é composto por três conceitos:

- *Conceito 1*: $size = small$ e $color = red$
- *Conceito 2*: $color = green$ ou $shape = circular$
- *Conceito 3*: $size = medium$ ou $size = large$

O *dataset* STAGGER consiste em 120 exemplos. O *Conceito 1* está ativo do exemplo 1 ao 40; o *Conceito 2* está ativo do exemplo 41 ao 80 e o *Conceito 3* do exemplo 81 ao 120.

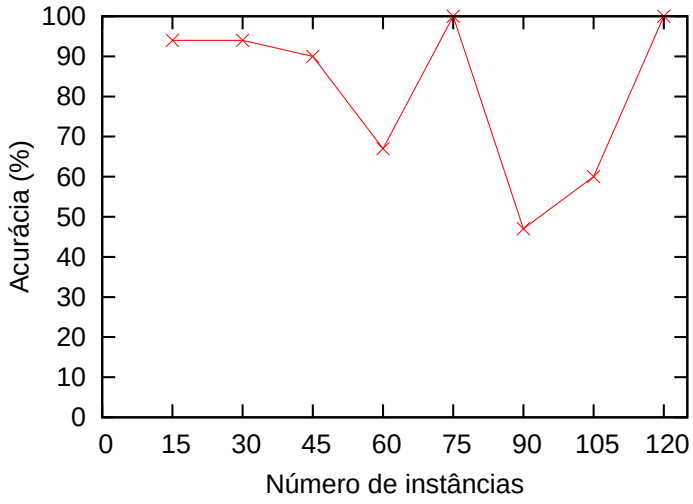
Um classificador CDHPid Naive Bayes foi treinado usando o dataset STAGGER e sua acurácia foi avaliada a cada 15 exemplos. A Figura 6.9(a) mostra o resultado desse experimento. A primeira acurácia é aferida com os 15 exemplos iniciais, os quais se referem ao *Conceito 1*. A segunda acurácia é aferida quando o tamanho do *datataset* atinge 30 exemplos. Como os 15 novos exemplos correspondem ao mesmo conceito aprendido no passo anterior, a acurácia do classificador é alta (maior que 90%). Entre 30 e 45 exemplos, a acurácia começa a cair. Dos 15 novos exemplos processados, 5 correspondem a um novo conceito (*Conceito 2*). Com 60 exemplos, a acurácia cai drasticamente (menos de 70%). Nesse momento, o algoritmo CDHPid Naive Bayes detecta um *concept drift* e um novo modelo é construído usando os exemplos referentes ao *Conceito 2*. Com 75 exemplos, a acurácia cresce novamente, pois os 15 novos exemplos processados correspondem ao *Conceito 2*. Entre 75 e 90 exemplos, a acurácia cai novamente (menos de 50%) porque 10 dos novos exemplos correspondem ao *Conceito 3*. O algoritmo detecta uma mudança de conceito e um novo modelo é construído com os 15 últimos exemplos. Com 105 exemplos, a acurácia atinge aproximadamente 60% e com 120 exemplos volta para o patamar próximo a 100%.

Na Figura 6.9(b) são mostradas as curvas do gráfico de controle durante o treinamento do classificador CDH PID Naive Bayes usando o *dataset* STAGGER. Como o gráfico é construído utilizando a taxa de erro, a figura mostra essa métrica em função do tamanho do *dataset* de treinamento. O gráfico mostra que durante o treinamento ocorrem duas mudanças de conceitos. Essas estão representadas no gráfico nos pontos em que o símbolo * está circulado, ou seja, quando o *dataset* atinge 60 exemplos e, posteriormente, com 90 exemplos.

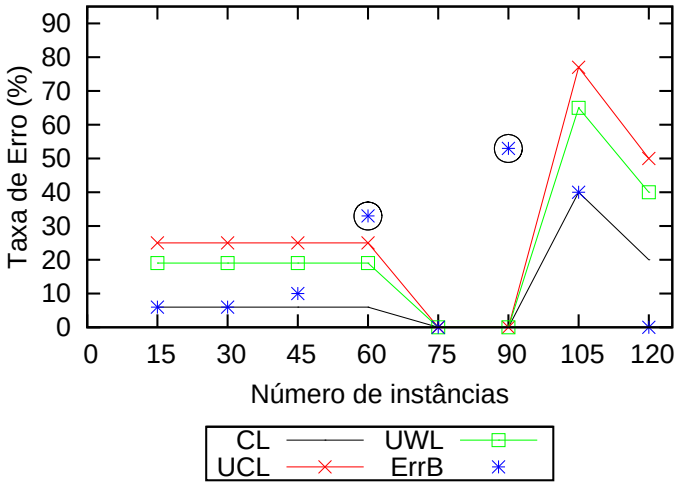
Para comparar o desempenho da versão Pid Naive Bayes com tratamento de *concept drift* e a versão sem tratamento, o desempenho da versão sem tratamento foi avaliada no *dataset* STAGGER. Como na versão com tratamento de *concept drift*, o classificador Pid Naive Bayes foi treinado incrementalmente e a acurácia do classificador foi aferida a cada 15 exemplos. A Figura 6.10 mostra a acurácia dos dois classificadores. Com 45 exemplos, as acurácias dos dois classificadores começam a se diferenciar. Entre 45 e 60 exemplos, a acurácia dos dois classificadores cai devido a uma mudança de conceito. Com 75 exemplos, todavia, a versão com tratamento de *concept drift* se adaptou à mudança e a acurácia desse classificador volta a crescer, enquanto a acurácia do classificador sem tratamento de de *concept drift* continua caindo. Com 90 exemplos, a acurácia do classificador CDHPid Naive Bayes cai novamente e o classificador detecta a segunda mudança de conceito. O classificador Pid Naive Bayes

Figura 6.9: Comportamento do algoritmo CDHPid Naive Bayes durante treinamento usando o *dataset* STAGGER.

(a) Comportamento da acurácia do Classificador CDHPid Naive Bayes.

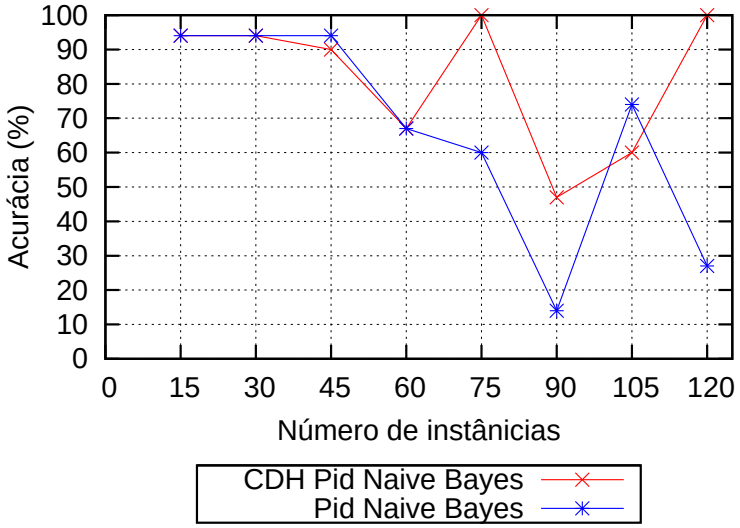


(b) Curvas do gráfico de controle.



também atinge o nível mais baixo de acurácia em toda a execução. Com 105 exemplos a acurácia dos dois classificadores volta a crescer. No entanto, a acurácia obtida pelo classificador Pid Naive Bayes não é consistente e com 120 exemplos a acurácia desse algoritmo cai novamente. A acurácia do classificador CDHPid Naive Bayes, ao contrário, aumenta de forma sustentável até atingir 100%. Fica claro, portanto, que o classificador CDHPid Naive Bayes tem um desempenho superior ao algoritmo Pid Naive Bayes em condições de mudanças de conceito.

Figura 6.10: Desempenho dos classificadores Pid Naive Bayes e CDHPid Naive Bayes no *dataset* STAGGER.



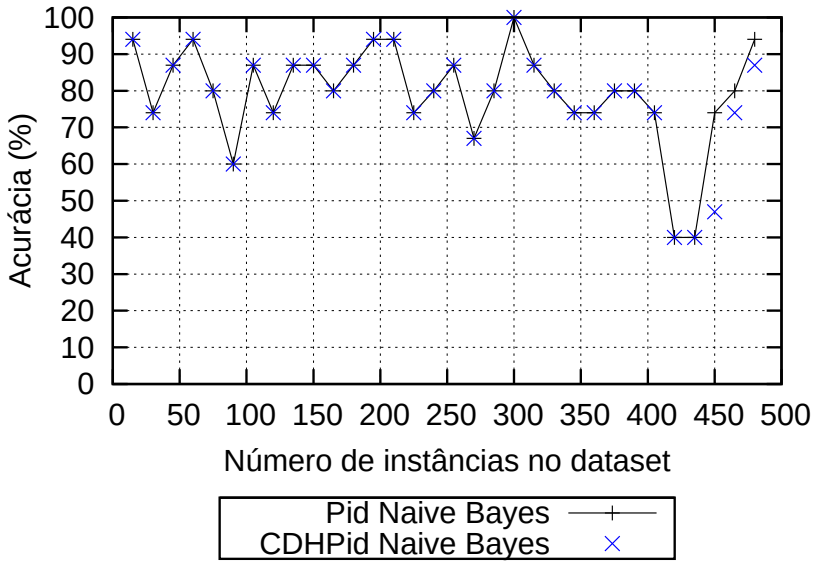
O dataset MapReduce

O desempenho do algoritmo CDHPid Naive Bayes foi avaliado também usando o *dataset MapReduce*. Para simular um *dataset* menos comportado, um subconjunto do *dataset* original foi criado, escolhendo aleatoriamente 500 exemplos. Esse *subdataset* foi denominado *dataset MapReduce CDH*. Para esse experimento, o SLO foi definido como o 80ºpercentil da variável *avg_response_time* e a memória curta foi configurada com o tamanho de 50 instâncias. Ou seja, para cada lote de 50 instâncias, verifica-se a ocorrência de *concept drift*. A Figura 6.11 mostra o desempenho do algoritmo nesse *dataset*. Para efeito de comparação, o desempenho do algoritmo Pid Naive Bayes sobre o mesmo *dataset* também é ilustrado. A acurácia de ambos os algoritmos é aferida a cada 15 exemplos. Como pode ser visto na figura, ambos os algoritmos apresentam um comportamento similar. Esse resultado permite concluir que o *dataset* criado não apresenta mudanças de conceitos drásticas. Esse fato pode ser provado analisando a estacionariedade[2] (Trivedi, 2002) do *dataset MapReduce CDH*.

Segundo (Yang e Shahabi, 2005), uma forma de determinar a estacionariedade de um *dataset* é determinar a estacionariedade dos processos representados por cada um de seus atributos. Se todos os atributos do *dataset* são

[2]De acordo com (Box et al., 1994) uma definição intuitiva de um processo estacionário é dada por um processo que permanece em equilíbrio em torno de um nível médio constante. Geralmente, se um processo (ou série de tempo) é estacionário, o mesmo pode ser descrito através de sua média ou variância.

Figura 6.11: Desempenho dos classificadores Pid Naive Bayes e CDHPid Naive Bayes no *dataset MapReduce CDH*.



representados por um processo estacionário, então o *dataset* é considerado estacionário. Por outro lado, a estacionariedade de um processo $\{X_i\}$ pode ser verificada de acordo com os seguintes passos:

1. inicialmente o processo $\{X_i\}$ é dividido em intervalos de tempo iguais, cada um contendo m amostras,

2. para cada intervalo, calcula-se a média das amostras, criando um nova seqüência, $\{X_k^{(m)}\}$, formada pelas médias,

3. testar a seqüência das médias, $\{X_k^{(m)}\}$, utilizando o teste de iteração (*run test*).

O teste de iteração é um testes de hipótese, ou seja, baseia-se em uma hipótese nula (h_0) e em uma hipótese alternativa (h_1). Nesse caso, h_0 diz que o processo sendo avaliado não possui tendência, ou seja, é estacionário e h_1 estabelece que há alguma tendência e, portanto, é não-estacionário. A Tabela 6.4 ilustra como o teste verifica as duas hipóteses, sendo que maiores detalhes sobre o mesmo podem ser obtidos em (Bendat e Piersol, 1986).

A Figura 6.12 mostra a aplicação do teste de estacionariedade para os nove atributos (Tabela 3.1) que compõem o *dataset MapReduce CDH*. A coluna *map_reduce_phase* foi excluída da análise por não fazer sentido o cálculo de tendência para esta coluna. A coluna *avg_response_time* também foi excluída da análise por não fazer parte do *dataset* analisado. Pelo gráfico, a

Tabela 6.4: Testes de estacionariedade.

Teste de iteração
1. As observações são divididas em dois tipos, por exemplo: acima ou abaixo da mediana 2. Define **iteração** como uma seqüência de observações do mesmo tipo 3. Toma como hipótese que não há tendência na seqüência de N observações 4. Dado um nível de significância α, verifica se o número de iterações observado está entre $r_{n;1-\alpha/2}$ e $r_{n;\alpha/2}$, onde $n = N/2$ e: – $\mu_r = \frac{2N_1N2}{N} + 1$ e se $N_1 = N_2$ então $\mu_r = \frac{N}{2} + 1$ – $r_{n;1-\frac{\alpha}{2}} = \mu_r + Z^{-1}(1-\alpha)\sigma_r + 0,5$ e $r_{n;\frac{\alpha}{2}} = \mu_r + Z^{-1}(\alpha)\sigma_r - 0,5$ – Se dentro do intervalo aceita hipótese, senão rejeita

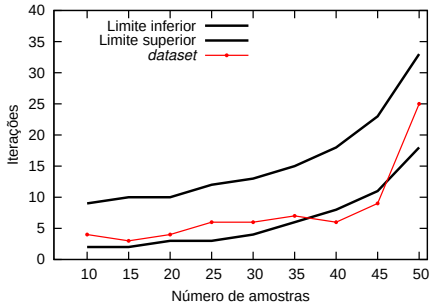
estacionariedade é verificada através da posição em relação aos limites inferior e superior. Esses limites são calculados pelo teste de iteração e são definidos pelo nível de significância (α) escolhido para o teste. Para este experimento foi usado o nível 0.05. Ser estacionário significa estar entre os limites inferior e superior, a maior parte do tempo. Como mostrado na Figura 6.12, exceto para o processo que representa o atributo *net_bytes_in*, todos os outros processos que representam os demais atributos analisados estão completamente contidos dentro dos limites superior e inferior do gráfico. Todavia, como pode ser notado no gráfico 6.12(a), o processo que representa a coluna *net_bytes_in* se encontra dentro dos limites inferior e superior do gráfico em 80% dos casos. Portanto, esse processo é considerado estacionário. Conclui-se, assim, que o *dataset MapReduce CDH* analisado é estacionário.

Avaliação do Desempenho do SMART

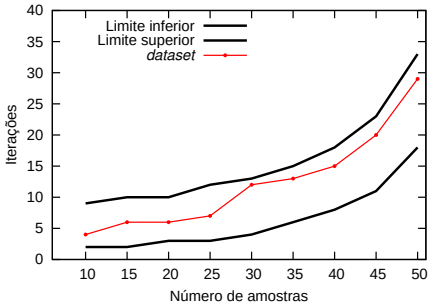
Para finalizar os experimentos deste capítulo, o desempenho do SMART foi avaliado quando sua máquina de inferência é implementada usando o algoritmo CDHPid Naive Bayes. Usando o mesmo cenário de carga senoidal e as mesmas configurações (Tabela 6.2) usadas na avaliação do algoritmo Pid Naive Bayes, a aplicação MapReduce foi configurada para usar 40 componentes *Wor-*

Figura 6.12: Teste de estacionariedade para o *dataset MapReduce CDH*

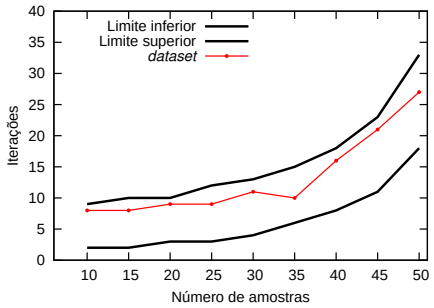
(a) Atributo net_bytes_in



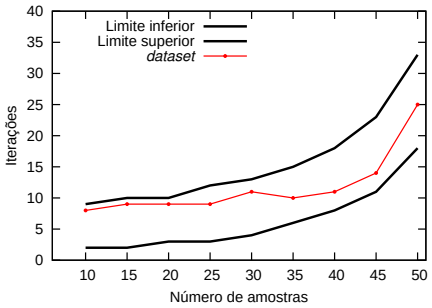
(b) Atributo net_bytes_out



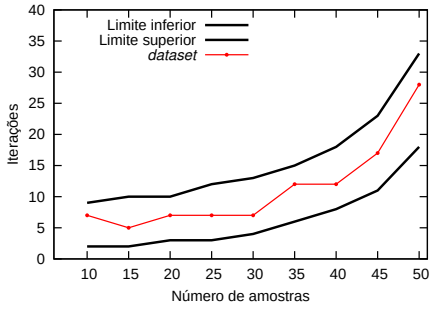
(c) Atributo disk_bytes_read



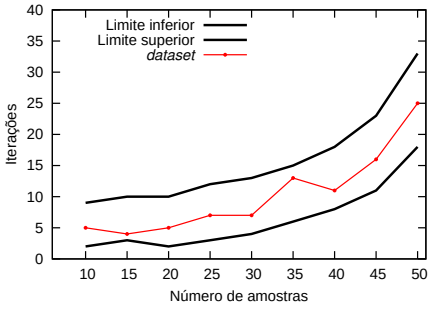
(d) Atributo disk_bytes_write



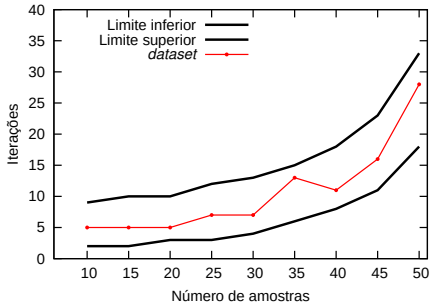
(e) Atributo nfs_bytes_read



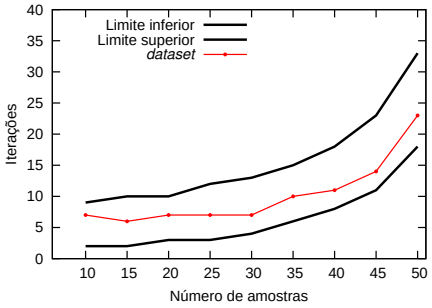
(f) Atributo nfs_bytes_write



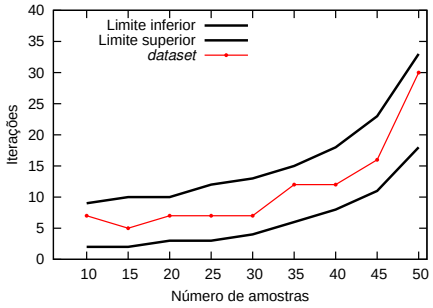
(g) Atributo cpu_usage



(h) Atributo memory_usage



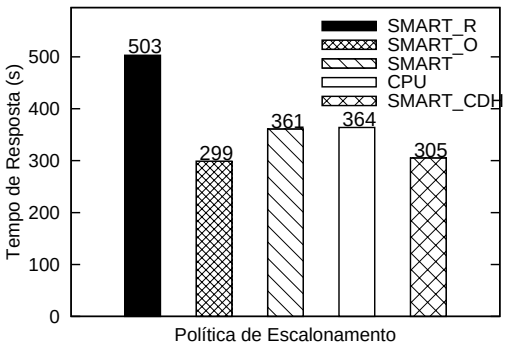
(i) Atributo containers_avg_cpu_usage



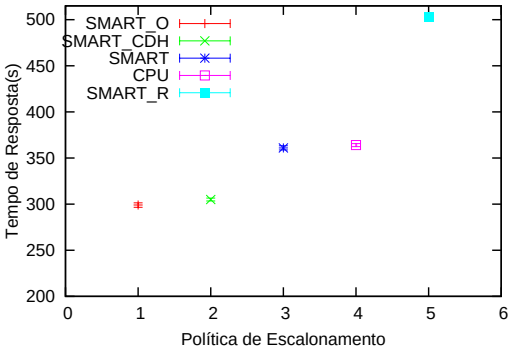
kers e o sistema foi submetido à carga senoidal. A memória curta do algoritmo CDHPid Naive Bayes foi configurada com capacidade para 15 exemplos. O desempenho do algoritmo CDHPid Naive Bayes (SMART_CDH) é mostrado nas Figuras 6.13(a) e 6.13(b), juntamente com o desempenho atingido pelos outros algoritmos, sob as mesmas condições de execução. Novamente, cada tempo de execução corresponde ao valor médio obtido em 10 execuções, usando um nível de 95% no intervalo de confiança. É possível notar que, mais uma vez, não houve uma diferenciação significativa dos desempenhos dos algoritmos Pid Naive Bayes e CDHPid Naive Bayes. Esse resultado demonstra que o ambiente de experimentação é relativamente estável em relação aos conceitos aprendidos e, portanto, os comportamentos dos dois algoritmos são semelhantes.

Figura 6.13: Desempenho da aplicação MapReduce usando o algoritmo CDHPid Naive Bayes.

(a) Tempo de Resposta



(b) Intervalo de Confiança



6.4 Considerações Finais

Neste capítulo, foi apresentado e avaliado um classificador baseado em uma rede Bayesiana, projetado para operar em cenários de aprendizado *online*. O intuito foi estabelecer o quanto um algoritmo de aprendizado *online* pode ser mais apropriado para a construção de modelos de desempenho de sistemas baseados em *middleware*. O classificador Pid Naive Bayes foi a proposta deste trabalho para tratar esse problema. O algoritmo proposto apresentou um bom desempenho em todos os cenários avaliados. Em especial, destaca-se o desempenho do algoritmo quando este foi usado na implementação da máquina de inferência do SMART. Sob um cenário de execução similar a algum cenário percebido em tempo de treinamento pelos algoritmos de aprendizado *offline*, o algoritmo Pid Naive Bayes alcançou desempenho semelhante aos classifica-

dores *offline*. Todavia, em um cenário de execução novo para todos os algoritmos, o algoritmo Pid Naive Bayes apresentou um desempenho superior aos classificadores *offline*. Considerando a complexidade dos sistemas baseados em middleware e a complexidade dos ambientes em que esses sistemas executam, a construção de *datasets* de treinamento abrangentes é um processo difícil e dispendioso. Portanto, soluções que possam aprender de forma *online* são de grande importância.

O Capítulo também discutiu um problema que pode ocorrer em ambientes dinâmicos, o *concept drift*. Nesse sentido, foi proposta uma alteração no algoritmo Pid Naive Bayes, a qual denominou-se CDHPid Naive Bayes. Embora o algoritmo CDHPid Naive Bayes tenha apresentado um desempenho superior ao algoritmo Pid Naive Bayes num *dataset* padrão para avaliação de *concept drift*, o desempenho do primeiro algoritmo foi muito parecido com o do segundo, considerando a máquina de inferência do SMART. Esse resultado mostrou que o cenário de referência considerado neste trabalho não apresenta mudanças drásticas de conceitos, tornando o comportamento de ambos algoritmos semelhantes. No futuro, será interessante a realização de experimentos que explorem o algoritmo CDHPid Naive Bayes no contexto de problemas de desempenho de outros tipos de aplicações.