Desenvolvimento de um arcabouço probabilístico para implementação de campos aleatórios condicionais

Ígor Bonadio

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação

Orientador: Prof. Dr. Alan Mitchell Durham

São Paulo, fevereiro de 2013

Desenvolvimento de um arcabouço probabilístico para implementação de campos aleatórios condicionais

Esta versão da dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa da versão original do trabalho, realizada em 19/03/2013. Uma cópia da versão original está disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Alan Mitchell Durham (orientador) - IME-USP
- Prof. Dr. André Fujita - IME-USP
- Prof. Dr. Ricardo Zorzetto Nicoliello Vêncio - FFCLRP-USP

Agradecimentos

Agradeço a minha família o apoio que recebi desde o início. Agradeço a minha *Pequena*, Jessica Eto, a paciência e carinho. Agradeço ao Prof. Dr. Alan Mitchell Durhan a orientação durante todo esse trabalho e a oportunidade de estudar problemas tão desafiadores. Agradeço ao Prof. Dr. André Yoshiaki Kashiwabara a ajuda no entendimento de vários conceitos importantes para a conclusão deste trabalho. Agradeço também a todos os amigos que fiz no IME-USP, em especial aos membros do nosso grupo de pesquisa: Alexandre Paschoal, Felipe Amado, Laecio Freitas, Liliane Santana, Pedro Ivo Gomes de Faria, Rafael Mathias, Renato Cordeiro Ferreira e Vitor Onuchic.

Resumo

Desenvolvimento de um arcabouço probabilístico para campos aleatórios condicionais

A segmentação de sequências é um problema desafiador que pode ser aplicado em diversas áreas de pesquisa. O uso de modelos geradores como Modelo Oculto de Markov (HMM, do inglês *Hidden Markov Model*) e Modelo Oculto de Markov Generalizado (GHMM, do inglês *Generalized Hidden Markov Model*) é uma das abordagens mais comuns, entretanto recentemente pesquisas apontam um novo modelo discriminativo promissor chamado Campo Aleatório Condicional. Nesse trabalho desenvolvemos um arcabouço probabilístico orientado a objetos para a implementação de Campos Aleatórios Condicionais de Cadeias Lineares, uma variante que é bastante utilizada na segmentação de sequências. Esse objetivo foi alcançado a partir da extensão do arcabouço probabilístico ToPS, originalmente projetado para HMMs, GHMMs e outros modelos geradores. Nossa implementação conta com algoritmos de inferência eficientes que paralelizam a computação e alcançam tempos de execução competitivos. Focando na facilidade de utilização, nosso arcabouço conta com um interpretador de modelos que permite a definição e treinamento de Campos Aleatórios Condicionais agilizando a prototipagem e investigação de problemas. Comparamos nossa abordagem com programas já existentes e observamos que nosso arcabouço permite ao usuário maior liberdade na definição de modelos além de obter melhor desempenho com relação ao tempo de execução.

Palavras-chave: campos aleatórios condicionais, segmentação de sequências, modelos probabilísticos.

Abstract

Development of a Probabilistic Framework for Conditional Random Fields Implementation

Sequence segmentation is a challenging problem that can be applied on several research areas. The use of generative models such as Hidden Markov Models and Generalized Hidden Markov Models is one of most common approaches, but recently researchers have introduced a new promising alternative, Conditional Random Fields, a discriminative probabilistic model. In this work we developed an object-oriented probabilistic framework for the implementation of Linear-Chain Conditional Rando Fields, a variant that is widely used in sequence segmentation. To do this we extended the probabilistic framework ToPS, originally designed for HMMs, GHMMs and other generative models. We developed a parallel, efficient variation of the inference algorithms that achieve competitive running times. Focusing on ease of use, our framework includes a model interpreter that allows users to define and train Conditional Random Fields with a notation closer to the mathematical model, which should accelerate prototyping and facilitate investigation of variant models. When compared to software with similar goals, our framework allows for the implementation of more general models, and achives superior run-time performance.

Keywords: conditional random fields, sequence segmentation, probabilistic models.

Sumário

Lista de Abreviaturas

AROW	*Adaptive Regularization Of Weight Vector*
BFGS	*Broyden-Fletcher-Goldfarb-Shanno*
CRF	Campo Aleatório Condicional (*Conditional Random Field*)
DNA	Ácido Desoxirribonucleico (*Deoxyribonucleic acid*)
EBNF	*Extended Backus-Naur Form*
GHMM	Modelo Oculto de Markov Generalizado (*Generalized Hidden Markov Model*)
HMM	Modelo Oculto de Markov (*Hidden Markov Model*)
LBFGS	*Limited-memory Broyden-Fletcher-Goldfarb-Shanno*
LCCRF	Campo Aleatório Condicional de Cadeias Lineares (*Linear-Chain Conditional Random Field*)
ToPS	*Toolkit for Probabilistic Models of Sequence*
MC	Cadeia de Markov (*Markov Chain*)
MEMM	Modelo de Markov de Entropia Máxima (*Maximum-entropy Markov model*)
OWL-QN	*Orthant-Wise Limited-memory Quasi-Newton*
SGD	*Stochastic Gradient Descent*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Automatizar a identificação e classificação de padrões não é um processo trivial e é foco em diversas áreas de pesquisa incluindo bioinformática (Kulp *et al.*, 1996), reconhecimento de fala (Rabiner e Juang, 1993) e biometria (Maltoni *et al.*, 2009). Os problemas de reconhecimento de padrões podem ser resolvidos com abordagens e técnicas diferentes, sendo que todos compartilham o mesmo objetivo: dado um modelo previamente determinado, atribuir uma classificação à um dado conjunto de observações cujo significado não é bem compreendido.

Vale ressaltar que o tipo de dado a ser analisado também influencia na escolha da técnica a ser utilizada. Neste trabalho focaremos em dados sequenciais discretos como, por exemplo, sequências biológicas (DNA e proteínas). Neste contexto, uma das abordagem mais utilizadas é a segmentação de sequências. Diferente do problema de classificação clássico, em que procuramos predizer somente uma classe para um dado vetor de observações, a segmentação de sequências lida com variáveis interdependentes e tem por objetivo, dada uma sequência $x = (x_1, x_2, ..., x_k)$ encontrar uma sequência $y = (y_1, y_2, ..., y_k)$ tal que cada elemento de y atribua um significado para cada elemento de x. Esta abordagem é normalmente seguida no clássico problema do cassino desonesto (Durbin *et al.*, 1998), em predição de genes (Burge, 1997) e em alinhamento de sequências (Eddy, 1998).

Dentre as técnicas mais comuns de segmentação de sequências, podemos destacar o Modelo Oculto de Markov (HMM, do inglês *Hidden Markov Model*) (Duda *et al.*, 2000) e sua generalização, o Modelo Oculto de Markov Generalizado (GHMM, do inglês *Generalized Hidden Markov Model*) (Kulp *et al.*, 1996). Ambos são conhecidos como modelos probabilísticos geradores por, dentre outras características, poderem ser usados tanto para a rotulação como para geração de sequências aleatórias.

Em 2001, um novo modelo chamado de Campo Aleatório Condicional (CRF, do inglês *Conditional Random Field*) foi introduzido (Lafferty *et al.*, 2001). Este modelo apresentou resultados promissores em diversas áreas como reconhecimento de entidades nomeadas (Finkel *et al.*, 2005), predição de genes (DeCaprio *et al.*, 2007) e análise de documentos (Shen *et al.*, 2007). Uma característica interessante é que, diferente dos modelos já citados, este modelo é classificado como um modelo discriminativo e modela a dependência entre as variáveis não observadas y e as variáveis observadas x a partir da distribuição de probabilidade condicional $P(y|x)$ que é suficiente para a classificação.

1.1 Motivação

Um ponto importante na elaboração de sistemas capazes de segmentar sequências é que estes são bastante sensíveis à técnica escolhida, bem como à modelagem do problemas e à escolha de seus parâmetros. Como exemplo, na elaboração de preditores de genes, uma pequena alteração na arquitetura de um GHMM pode trazer mudanças significativas aos resultados (Kashiwabara, 2012). Sendo assim, uma ferramenta capaz de auxiliar nessa etapa é bastante atraente.

Entretanto, a maioria das ferramentas existentes apresentam pouca ou nenhuma flexibilidade na modelagem. Por exemplo, Genezilla (Majoros *et al.*, 2004) e Twinscan (Korf *et al.*, 2001) são

preditores de genes que permitem a troca de submodelos, mas que mesmo assim possuem limitações, como não permitirem a troca da arquitetura do GHMM sem modificar seu código-fonte, além da dificuldade de serem treinados.

Vale ressaltar também que normalmente as implementações são voltadas a um problema específico, como predição de genes, como os já citados Genezilla e Twinscan, ou reconhecimento de entidades nomeadas, como o Stanford NLP (Finkel *et al.*, 2005) e o BANNER (Leaman e G., 2008).

Com isso em mente, Kashiwabara (2012) iniciou o desenvolvimento de um arcabouço probabilístico orientado a objetos de fácil extensão chamado ToPS, cujo objetivo é facilitar a criação de modelos probabilísticos geradores para a segmentação de sequências. O arcabouço também ajuda na execução de tarefas comuns como gerar sequências aleatórias, segmentar sequências e treinar modelos. Outra característica interessante é o fato de a definição de modelos probabilístico ser feita a partir de arquivos de configuração definidos de forma semelhante à linguagem matemática tradicional. Isso dispensa a necessidade de se escrever programas a fim de se segmentar sequências. Além disso, este arcabouço foi base para o desenvolvimento de preditores de genes bastante competitivos.

A introdução de CRFs (Lafferty *et al.*, 2001) e os bons resultados iniciais motivou a construção de alguns arcabouços para a modelagem de problemas a partir desta abordagem discriminativa. Exemplos são CRFSuite (Okazaki, 2007a) e CRF++ (Kudo, 2005). Porém estes arcabouços se limitam a modelar apenas funções de características simples e não permitem a adição de submodelos probabilísticos, nem mesmo a adição de informações provenientes de processos externos.

Nosso objetivo é estender o arcabouço ToPS, inicialmente desenvolvido para modelar modelos geradores, para que possamos utilizar também modelos discriminativos e assim aumentar as possibilidades no desenvolvimento de soluções para a segmentação de sequências. Estamos interessados em investigar o uso de CRFs, em particular uma variante específica chamada de Campo Aleatório Condicional de Cadeias Lineares (LCCRF, do inglês *Linear-Chain Conditional Random Field*) (Sutton e McCallum, 2006) que é relacionada na literatura aos HMMs. Desenvolvemos implementações paralelas dos algoritmos mais eficientes de inferência e treinamento para LCCRFs, ao mesmo tempo que mantivemos a modularidade do arcabouço ToPS. Estes pontos são importantes pois desta forma é possível utilizar CRFs como modelos agregadores e assim combinar a abordagem discriminativa com a geradora na segmentação de sequências. Desenvolvemos também uma nova arquitetura de classes para permitir a modelagem tanto de características probabilísticas como não probabilísticas.

Capítulo 2

Modelos Geradores

Modelos geradores tais como HMM e GMM são bastante utilizados em segmentação de sequências (Burge, 1997; Eddy, 1998; Hughey e Krogh, 1996; Nabende *et al.*, 2008; Rabiner, 1989). Esses modelos representam a distribuição de probabilidades conjunta do tipo $p(x, y)$, sendo x a sequência de dados que observamos e y a sequência de rótulos atribuídos à cada elemento de x que desejamos predizer.

Nesse capítulo abordaremos três modelos probabilísticos geradores Markovianos: Cadeias de Markov (MC, do inglês *Markov Chains*) (2.2), Modelo Oculto de Markov (HMM, do inglês *Hidden Markov Model*) (2.3) e Modelo Oculto Generalizado de Markov (GHMM, do inglês *Generalized Hidden Markov Model*) (2.6) bem como aplicações destes modelos.

2.1 Processos Markovianos

Dado um processo estocástico $X(t)$ definido para todo $t \in t_1 < t_2 < ... < t_n$, podemos obter as seguintes funções de densidade de probabilidade condicional de primeira ordem:

$$\begin{aligned} &f_X(x_1) \\ &f_X(x_2|x_1) \\ &f_X(x_3|x_2, x_1) \\ &f_X(x_{n-1}|x_{n-2}, ..., x_2, x_1) \\ &f_X(x_n|x_{n-1}, ..., x_2, x_1) \end{aligned} \tag{2.1}$$

Entretanto, podemos simplificar a definição do processo estocástico se aceitarmos as funções de densidade de probabilidade condicional como:

$$f_X(x_n|x_{n-1}, ..., x_2, x_1;) = f_X(x_n|x_{n-1}) \tag{2.2}$$

Essa restrição significa que o comportamento probabilístico do presente depende somente de seu passado imediato, caracterizando o dado processo estocástico como um processo Markoviano (Grimmett e Stirzaker, 2001).

2.2 Cadeias de Markov

Se um processo Markoviano possui estados discretos, este é chamado de Cadeia de Markov, e pode ser definido para tempos discretos ou contínuos, e para conjuntos de estados finitos ou infinitos (Krishnan, 2006). Durante esse capítulo abordaremos somente as Cadeias de Markov de tempo discreto para um conjunto de estados finito, já que o objetivo deste trabalho é a segmentação de sequências de símbolos pertencentes à um alfabeto discreto.

Uma Cadeia de Markov pode ser representada graficamente por um grafo direcionado cujos vértices representam os *estados* que são associados à um símbolo, sendo que cada um desse estados pode ser conectado à um outro qualquer. Como exemplo temos a figura 2.1, que representa um Cadeia de Markov simples que possui apenas dois estados A e B.

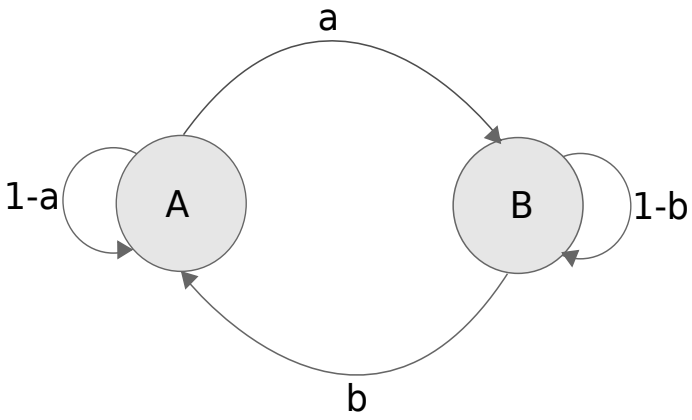

Figura 2.1: *Grafo direcionado representando uma Cadeia de Markov*

Cada aresta desse grafo tem uma probabilidade de transição associada, que também pode ser descrito pela equação abaixo:

$$a_{ij} = P(x_t = i | x_{t-1} = j) \tag{2.3}$$

onde x é a sequência de símbolos que estamos analisando, x_t representa o t-ésimo símbolo dessa sequência e i e j são dois estados da cadeia de Markov.

De modo geral, para qualquer modelo probabilístico podemos escrever a probabilidade de uma sequência x de comprimento L como:

$$\begin{aligned} P(x) &= P(x_L, x_{L-1}, ...x_1) \\ &= P(x_L | x_{L-1}, ...x_1) P(x_{L-1} | x_{L-2}, ...x_1) ... P(x_1) \end{aligned} \tag{2.4}$$

Sendo uma cadeia de Markov um processo markoviano, podemos reescrever (2.4) como

$$\begin{aligned} P(x) &= P(x_L | x_{L-1}) P(x_{L-1} | x_{L-2}) ... P(x_2 | x_1) P(x_1) \\ &= P(x_1) \prod_{i=2}^{L} a_{x_{i-1} x_i} \end{aligned} \tag{2.5}$$

Note que comportamento probabilístico do presente depende somente de seu passado imediato.

2.3 Modelo Oculto de Markov

Modelo Oculto de Markov é um tipo de modelo probabilístico gerador bastante popular em segmentação de sequências. Seguindo a notação de Rabiner (1989), podemos definir um HMM como uma quíntupla (S, V, A, B, D), onde:

* $S = \{s_1, s_2, ..., s_N\}$, o conjunto de estados ocultos
* $V = \{v_1, v_2, ..., v_M\}$, o conjunto de símbolos visíveis

* $A = \{a_{ij}\}$, o conjunto que define as probabilidades de transição de um estado s_i para um estado s_j para todo $1 \leq i, j \leq N$. A equação (2.6) define essas probabilidades, sendo $y = (y_1, y_2, ..., y_L)$ a sequência de rótulos cujo comprimento é L e t o instante atual.

$$a_{ij} = P(y_t = s_j | y_{t-1} = s_i), 1 \leq i, j \leq N \tag{2.6}$$

* $B = \{b_j(k)\}$, o conjunto que define as probabilidades de emissão do símbolo $v_k \in V$ pelo estado $s_j \in S$. A equação (2.7) define essas probabilidades, sendo $y = (y_1, y_2, ..., y_L)$ a sequência de rótulos, $x = (x_1, x_2, ..., x_L)$ a sequência de símbolos observados, ambas de comprimento L, e t o instante atual.

$$b_j(k) = P(x_t = v_k | x_t = s_j), 1 \leq j \leq N, 1 \leq k \leq M \tag{2.7}$$

* Um conjunto de probabilidades iniciais $D = \{d_i\}$. A equação (2.8) define essas probabilidade, sendo y_1 o primeiro rótulo.

$$d_i = P(y_1 = s_i), 1 \leq i, j \leq N \tag{2.8}$$

Note que as equações (2.9) e (2.10) devem ser satisfeitas.

$$\sum_{j=1}^{N} a_{ij} = 1, 1 \leq i \leq N \tag{2.9}$$

$$\sum_{k=1}^{M} b_j(k) = 1, 1 \leq j \leq N \tag{2.10}$$

Alem disso podemos definir a probabilidade conjunta de uma sequência de rótulos y e de uma sequência de símbolos visíveis x ambas de tamanho L como:

$$P(y, x) = d_{y_1} P(x_1 | y_1) \prod_{t=2}^{T} P(y_t | y_{t-1}) P(x_t | y_t) \tag{2.11}$$

Uma representação possível deste modelo é a apresentada na figura 2.2, que modela o problema do cassino desonesto. Este problema consiste de um cassino que alterna, de acordo com as probabilidades definidas pelas arestas na figura 2.2, a utilização dois tipos de dados: Um honesto, onde todas as faces têm a mesma probabilidade de ocorrer em um lançamento, e um desonesto, cuja face número 1 tem maior probabilidade de acontecer do que as outras.

2.4 Algoritmos de Inferência para Modelos Ocultos de Markov

Existem três problemas básicos a serem resolvidos na utilização de HMMs (Rabiner, 1989):

* Dada uma sequência de símbolos observados $x = (x_1, x_2, ..., x_T)$, calcular eficientemente qual a probabilidade da sequência x ocorrer dado o modelo.

  Este problema é conhecido como *avaliação*, e é importante para, por exemplo, dado uma série de modelos, escolher qual modelo melhor descreve uma sequência de símbolos.

* Dada uma sequência de símbolos observados $x = (x_1, x_2, ..., x_T)$, encontrar uma sequência de estados $y = (y_1, y_2, ..., y_T)$ de modo que y explique x.

  Este problema é conhecido como *rotulação*, e a determinação de y depende do critério escolhido.

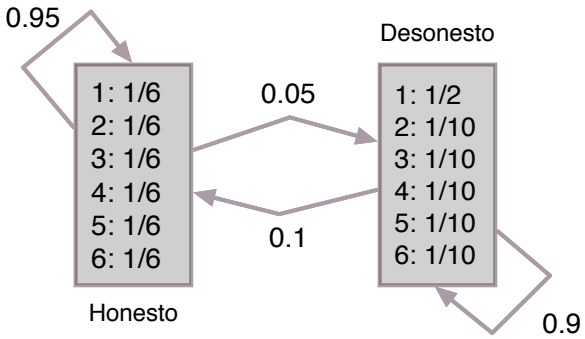

Figura 2.2: *Exemplo de HMM para o problema do Cassino Desonesto (Durbin et al., 1998)*

* Dado um conjunto de treinamento, ou seja um conjunto de sequências de símbolos observáveis, encontrar os parâmetros que maximizam a probabilidade dessas sequências ocorrerem dado o modelo.

Este problema é conhecido como *treinamento*.

2.4.1 Forward e Backward

O primeiro problema que apresentamos consiste de, dada uma sequência $x = (x_1, x_2, ..., x_T)$, encontrar $P(x)$ dado o modelo. Neste caso temos que:

$$P(x) = \sum_{\pi} P(x, \pi) \tag{2.12}$$

onde π é uma sequência de rótulos.

Entretanto, o número de caminhos possíveis π na equação (2.12) crescem exponencialmente e o cálculo de $P(x)$ pode não ser praticável para sequências de comprimento muito grandes. Uma alternativa é o algoritmo *forward* que utiliza uma técnica chamada programação dinâmica para calcular $P(x)$ eficientemente em tempo $O(TN^2)$ (Fraser, 2009).

Seja α a *matriz de programação dinâmica*, onde $\alpha_k(i)$ é a soma da probabilidade de todas as rotulações até o i-ésimo símbolo observado tal que o i-ésimo estado é k.

Temos então que $\alpha_k(i)$ pode ser facilmente calculada utilizando a seguinte recorrência:

$$\alpha_l(i) = b_l(x_{i+1}) \sum_{k} \alpha_k(i) a_{kl} \tag{2.13}$$

Por fim temos que

$$P(x) = \sum_{k} \alpha_k(T) a_{k0} \tag{2.14}$$

Uma alternativa ao algoritmo *forward* no cálculo de $P(x)$ é o algoritmo *backward* (Fraser, 2009), que também pode ser eficientemente calculado utilizando a técnica de programação dinâmica.

Seja β a *matriz de programação dinâmica*, onde $\beta_k(i)$ é a soma da probabilidade de todas as rotulações dos últimos i símbolos observados tal que o i-ésimo estado é k.

Temos então que $\beta_k(i)$ pode ser calculada eficientemente em tempo $O(TN^2)$ utilizando a seguinte recorrência:

$$\beta_k(i) = \sum_{l} a_{kl} b_l(x_{i+1}) \beta_l(i+1) \tag{2.15}$$

E também, assim como o algoritmo *forward*, nos permite calcular a probabilidade da sequência.

$$P(x) = \sum_{l} a_{0l} b_l(x_1) \beta_l(1) \tag{2.16}$$

2.4.2 Viterbi e Posterior Decoding

O segundo problema, rotulação, depende da abordagem escolhida. A primeira que apresentaremos é conhecida como algoritmo de *Viterbi* (Durbin *et al.*, 1998), que tem por objetivo encontrar uma sequência de estados ocultos π^* que maximize a probabilidade descrita em (2.11).

$$\pi^* = \arg\max_{\pi} P(x, \pi) \tag{2.17}$$

A sequência π^* também é conhecida por caminho mais provável, mas não pode ser obtida eficientemente utilizando a equação (2.17), que executa em tempo exponencial e torna a rotulação de sequência grandes um problema inviável. Felizmente π^* pode ser calculado eficientemente em tempo $O(TN^2)$ utilizando programação dinâmica.

Definimos π_i como o i-ésimo elemento da sequência π e $\pi_{1..j}$ como a sequência composta pelos símbolos $\pi_1, \pi_2, ..., \pi_j$.

Para calcular o caminho mais provável utilizaremos a *matriz de programação dinâmica* v, onde definimos, para cada estado k de nossa HMM, e cada posição i da sequência, o valor $v_k(i)$ dado pela fórmula:

$$v_l(i+1) = b_l(x_{i+1}) \max_k (v_k(i) a_{kl}) \tag{2.18}$$

que corresponde à probabilidade do caminho mais provável $\pi_{1..i+1}$, onde $\pi_{i+1} = l$.

Uma alternativa ao algoritmo de *viterbi* é encontrar qual o rótulo mais provável para uma dada posição da sequência de símbolos observados. Para isso utilizamos o algoritmo conhecido como *Posterior Decoding* (Durbin *et al.*, 1998):

$$\hat{\pi}_i = \arg\max_k P(\pi_i = k|x) \tag{2.19}$$

sendo que a probabilidade $P(\pi_i = k|x)$ pode ser calculada eficientemente utilizando as *matrizes de programação dinâmica* dos algoritmos *forward* e *backward*:

$$P(\pi_i = k|x) = \frac{\alpha_k(i)\beta_k(i)}{P(x)} \tag{2.20}$$

Lembrando que $P(x)$ pode ser facilmente calculado utilizando o algoritmo *forward* ou *backward*.

2.5 Aplicações de Modelos Ocultos de Markov

HMMs vêm sendo bastante utilizados em reconhecimento de fala (Rabiner, 1989), onde têm apresentado bons resultados (Dunham *et al.*, 1987) (Loof *et al.*, 2007) (Rybach *et al.*, 2009)

Além de reconhecimento de fala, HMMs vêm sendo bastante utilizados em bioinformática, como por exemplo, na predição de genes (Krogh, 1997), (Lukashin e Borodovsky, 1998), (Munch e Krogh, 2006), e para a caracterização de famílias de proteínas (Eddy, 1998). Neste último caso é utilizada uma variante de HMM chamada *profile HMM*, que representa o perfil de um alinhamento múltiplo e é capaz de auxiliar na busca em banco de dados por sequências homologas.

2.6 Modelo Oculto Generalizado de Markov

Os HMMs nos oferecem a possibilidade de associarmos uma distribuição de probabilidade a cada estado fazendo com que este possa emitir símbolos visíveis. Porém, se generalizarmos essa definição

fazendo com que cada estado possa ser um modelo probabilístico responsável por emitir partes da sequência de símbolos, temos um modelo bem mais flexível (Kulp *et al.*, 1996). Essa é a proposta do Modelo Oculto Generalizado de Markov (GHMM, em inglês *Generalized Hidden Markov Model*) que nos permite modelar as emissões e as durações de cada estado usando submodelos.

A definição formal de um GHMM é dada pela quíntupla (X, Y, a, d, b), onde

* X é o conjunto de estados/rótulos do GHMM;
* Y é o conjunto de símbolos observáveis;
* a é a função de probabilidade de transição entre os estados, sendo que $\sum_{i \in Y} a_{i,j} = 1$, para todo $j \in Y$;
* d é a função de probabilidade de duração, sendo que $\sum_{l \in \mathbb{N}^*} d(l, j) = 1$, para todo $j \in Y$;
* b é a função de probabilidade de emissão de símbolos observáveis, sendo que $\sum_{x \in X^l} b(x, j) = 1$, para todo $j \in Y$, $l \in \mathbb{N}^*$ e X^l é o conjunto de todas as palavras de comprimento l sobre o alfabeto X.

Como podemos notar o GHMM nos proporciona uma maior flexibilidade, entretanto dificulta a inferência estatística quando compararmos com HMM, que é um modelo mais simples (Axelson-Fisk, 2010).

2.7 Algoritmos de Inferência para Modelos Ocultos Generalizados de Markov

Os algoritmos *viterbi*, *forward* e *backward* foram definidos em 2.4 com o objetivo de realizar inferências em HMMs, e com algumas pequenas modificações podemos utilizá-los também em GHMMs (Axelson-Fisk, 2010).

Entretanto, sendo GHMM um modelo mais complexo e que pode emitir mais de um símbolo a cada estado, esses algoritmos de inferência passam a ser menos eficientes. Felizmente algumas técnicas podem ser empregadas para melhorar o desempenho. Burge (1997) e Majoros (2007) assumem algumas restrições com relação à arquitetura do modelo obtendo algoritmos de complexidade computacional $O(|X|^2 L)$, sendo $|X|$ o tamanho conjunto de estados e L o comprimento da sequência analisada.

2.8 Aplicações de Modelos Ocultos Generalizados de Markov

Após a introdução de GHMM em bioinformática por Kulp *et al.* (1996), este modelo vem sendo bastante utilizado na área, em particular para predição de genes (Burge, 1997) (Stanke e Waack , 2003) (Majoros *et al.*, 2004) (Stanke *et al.*, 2006). Kulp justifica sua escolha pelo fato de GHMM ser bastante flexível e modular, o que facilita a adição de novos sensores e estados.

Porém este tipo de modelo não é utilizado somente em bioinformática, existindo também aplicações no desenvolvimento de sistemas de síntese de voz a partir de texto (Zen *et al.*, 2004) e sistemas de manutenção preditivo baseado em condições (Dong e David, 2007).

Capítulo 3

Campos Aleatórios Condicionais

No capítulo anterior apresentamos alguns modelos geradores e destacamos algumas aplicações de sucesso. Entretanto, essa abordagem nos leva a necessidade de modelar a distribuição $p(x)$, que de modo geral é muito complexa e de difícil determinação.

Um exemplo da dificuldade de se modelar $p(x)$ foi apresentado por Charles Sutton e Andrew McCallum em seu trabalho (Sutton e McCallum, 2006), que consiste em utilizar um HMM para o reconhecimento de Entidades Nomeadas. Esse HMM é limitado a uma única característica que é a identidade entre palavras. Porém muitas palavras, especialmente os nomes próprios, não serão bem classificados, pois suas variações são grandes e podem nunca ter ocorrido no conjunto de treinamento. Essa é uma característica intrínseca aos modelos geradores, ou seja, eles podem classificar apenas as sequências que podem ser geradas pelo próprio modelo. Podemos resolver esse problema representando melhor a interdependência entre os dados de entrada, como por exemplo caracterizando seus prefixos, sufixos e se a palavra faz uso de letras maiúsculas. Porém essa abordagem nos leva à um modelo mais complexo e muitas vezes inviável.

Uma alternativa é utilizar modelos discriminativos, que são baseados na distribuição de probabilidade condicional:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} \tag{3.1}$$

que é suficiente para a classificação, como podemos notar na função $seg(x)$ que devolve uma sequência y de rótulos que melhor segmenta x.

$$seg(x) = \arg\max_y p(y|x) \tag{3.2}$$

E a partir de (3.1) e (3.2) temos

$$seg(x) = \arg\max_y \frac{p(x|y)p(y)}{p(x)} = \arg\max_y p(x|y)p(y) \tag{3.3}$$

Sendo assim, podemos destacar que uma das principais vantagens dos modelos discriminativos é a capacidade de adicionarmos funções de características complexas que melhorem a representação do problema sem recair na dificuldade de se modelar a distribuição das variáveis observáveis.

Esta é a abordagem adotada pelos Campo Aleatório Condicionais (CRF, do inglês *Conditional Random Fields*) (Lafferty *et al.*, 2001), que são o foco deste trabalho.

Por serem modelos discriminativos modelam a probabilidade condicional $p(y|x)$. Essa natureza condicional é sua principal vantagem sobre modelos geradores como HMM ou GHMM. Além disso, CRF evitam o conhecido *label bias problem* (Sutton e McCallum, 2006), que normalmente é observado em Modelos Markovianos Condicionais tal como o Modelo de Markov de Entropia Máxima (MEMM, do inglês *Maximum-entropy Markov model*).

3.1 Campos Aleatórios Condicionais de Cadeias Lineares

Campos Aleatórios Condicionais podem ser utilizados para variados tipo de dados de entrada, entretanto um modelo mais específico, chamado de Campo Aleatório Condicional de Cadeias Lineares (LCCRF, do inglês Linear-Chain Conditional Random Field), tem como objetivo segmentar sequências lineares.

Sejam x e y vetores aleatórios de tamanho N, $\Lambda = \{\lambda_k\} \in \mathbb{R}^K$ um vetor de parâmetros e $\{f_k(y, y', x)\}_{k=1}^K$ um conjunto de funções chamadas de *funções de características*. Então, um LCCRF é definido pela distribuição $p(y|x)$ que tem a forma de

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} exp\{\sum_{t=1}^{N}\sum_{k=1}^{K}\lambda_k f_k(y_t, y_{t-1}, \mathbf{x}_t)\} \tag{3.4}$$

onde

$$Z(\mathbf{x}) = \sum_{y} exp\{\sum_{t=1}^{N}\sum_{k=1}^{K}\lambda_k f_k(y_t, y_{t-1}, \mathbf{x}_t)\} \tag{3.5}$$

Podemos mover o somatório em N para fora da função exponencial na equação 3.4 e transformá-la em

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \prod_{t=1}^{N} \exp\{\sum_{k=1}^{K}\lambda_k f_k(y_t, y_{t-1}, \mathbf{x}_t)\} \tag{3.6}$$

E é simples notar que de 3.6, a probabilidade condicional (3.4) pode ser fatorada por funções do tipo:

$$\Psi_t(y_t, y_{t-1}, \mathbf{x}_t) = \exp\{\sum_{k=1}^{K}\lambda_k f_k(y_t, y_{t-1}, \mathbf{x}_t)\} \tag{3.7}$$

3.2 Representação Gráfica

Os modelos apresentados neste trabalho podem ser representados de modo gráfico. Tradicionalmente alguns modelos geradores são representados como grafos direcionados. Como exemplo, temos a representação na figura 3.1 de um HMM para o problema da moeda desonesta, que consiste em um jogo onde se alterna entre moedas honestas e desonestas.

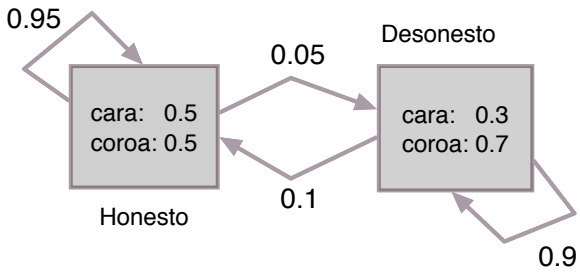

Figura 3.1: *Exemplo de HMM para o problema da moeda desonesta. Nesta representação os vértices representam os dois tipos de moedas e as arestas representam as probabilidades de alternar ou manter a utilização de uma certa moeda.*

Podemos também representar uma sequência de lançamentos como a figura 3.2. Nesta figura apresentamos um grafo direcionado que representa o lançamento de uma moeda honesta seguido de

2 lançamentos de uma moeda desonesta e novamente mais um lançamento de uma moeda honesta. Cada estado têm arestas que representam as possíveis emissões.

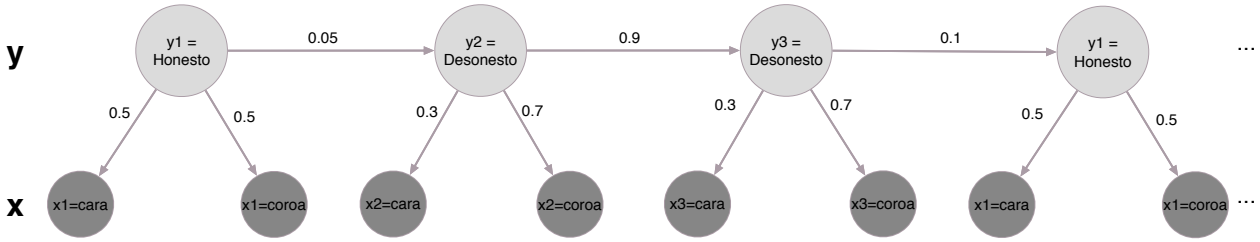

Figura 3.2: *Representação alternativa a apresentada na figura 3.1*

Já para a representação de modelos discriminativos, a utilização de grafos de fatores é mais comum (Sutton e McCallum, 2006).

Podemos utilizar grafos de fatores para modelar uma família de distribuição de probabilidade. Para isso, representamos esta distribuição de probabilidade como o produto de funções locais Ψ em que cada uma depende de um pequeno conjunto de variáveis aleatórias.

Seja V um conjunto de variáveis aleatórias, $A_i \subset V$ uma coleção de subconjuntos de V, e $\Psi_{A_i} : V^{n_i} \to \mathbb{R}^+$ um conjunto de funções, onde n_i é o número de elementos de A_i e v_{A_i} os elementos de A_i.

Sendo assim, para uma sequência v temos que

$$P(v) = \frac{1}{Z} \prod_i \Psi_{A_i}(v_{A_i}) \tag{3.8}$$

onde Z é um fator de normalização definidor por

$$Z = \sum_v \prod_i \Psi_{A_i}(v_{A_i}) \tag{3.9}$$

Podemos representar um conjunto de fatores como na figura 3.3, onde temos um grafo bipartido $G = (V, F, E)$ em que cada variável aleatória é representada como um vértice $v_s \in V$ que está conectado com à um vértice $\Psi_{A_i} \in F$, sendo $F = \{\Psi_{A_i}\}$ chamado de fator somente se v_s é um argumento de Ψ_{A_i}.

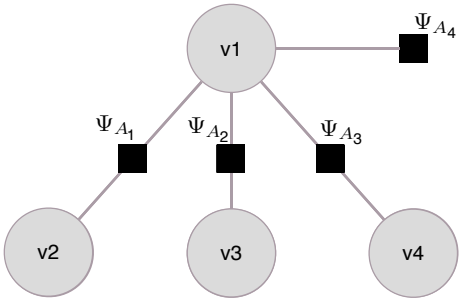

Figura 3.3: *Grafo de fatores*

Voltando ao exemplo do lançamento de moedas desonestas, podemos representar o lançamento descrito na figura 3.2 na forma de um grafo de fatores como o da figura 3.4, onde

* $f1(y_i, x_i) = 0.5$ se $x_i = cara$ e $y_i = honesto$

* $f2(y_i, x_i) = 0.5$ se $x_i = coroa$ e $y_i = honesto$
* $f3(y_i, y_{i-1}) = 0.05$ se $y_i = desonesto$ e $y_{i-1} = honesto$
* $f4(y_i, x_i) = 0.3$ se $x_i = cara$ e $y_i = desonesto$
* $f5(y_i, x_i) = 0.7$ se $x_i = coroa$ e $y_i = desonesto$
* $f6(y_i, y_{i-1}) = 0.9$ se $y_i = desonesto$ e $y_{i-1} = desonesto$
* $f7(y_i, y_{i-1}) = 0.1$ se $y_i = honesto$ e $y_{i-1} = desonesto$

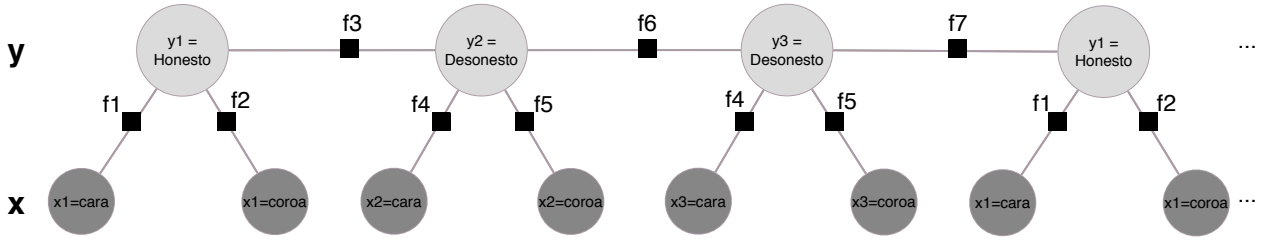

Figura 3.4: *Representação alternativa a apresentada na figura 3.2*

De maneira mais geral, observe que o CRF apresentado na figura 3.6 se assemelha ao HMM da figura 3.5, porém utilizando um grafo de fatores em sua representação. Este CRF possui funções de características do tipo $f_{ij}(y, y', x) = 1_{\{y=i\}}1_{\{y'=j\}}$ para cada transição entre estados não-observáveis (i, j) e $f_{io}(y, y', x) = 1_{\{y=i\}}1_{\{x=o\}}$ para cada par estado não-observado e seu símbolo emitido (i, o), sendo que $1_{\{a=b\}}$ representa a função cujo valor é 1 se a é igual à b e 0 caso contrário, e seus respectivos pesos $\lambda_{ij} = \log p(y' = i|y = j)$ e $\lambda_{oi} = \log p(x = o|y = i)$.

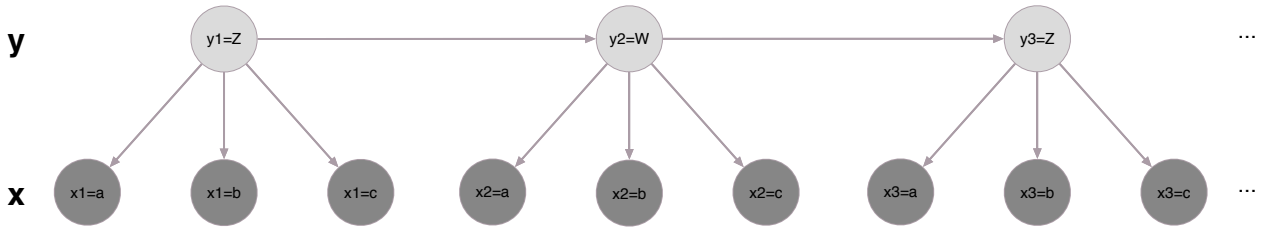

Figura 3.5: *HMM semelhante ao LCRF representado na figura 3.6, onde y_t são os estados ocultos e x_t os símbolos. As transições de um estado oculto y_t para um estado oculto y_{t-1} estão representadas pelas setas que interligam estes estados e ocorrem com probabilidade $p(y_t = i|y_{t+1} = j)$. Já as emissões de símbolos estão representadas pelas setas que interligam um estado oculto y_t com um símbolo x_t e ocorrem com probabilidade $p(x_t = o|y_t = i)$. Note também que esta é uma representação alternativa, mas também válida, ao HMM que apresentamos na figura 2.2.*

Assim, adicionar novas características ao modelo é bastante simples. Por exemplo, se desejamos que a transição também dependa da observação atual podemos adicionar a função de característica $1_{\{y_t=j\}}1_{\{y_{t-1}=i\}}1_{\{x_t=o\}}$ como observado na Figura 3.7.

É importante ressaltar que na definição de um LCCRF cada função de característica f_k pode depender de um observação à qualquer deslocamento de tempo t. Sendo assim, x_t deve ser definido com um vetor que contenha todos os símbolos necessários para o cálculo de f_k no dado tempo t.

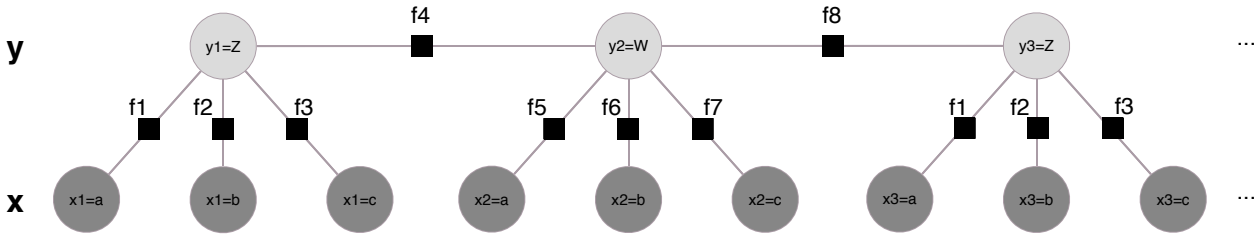

Figura 3.6: *LCCRF semelhante ao HMM representado na figura 3.5, onde y_t são os estados que representam os rótulos e x_t os estados que representam os símbolos que desejamos segmentar. Os fatores estão representadas como quadrados pretos seguindo a notação de grafos de fatores definida anteriormente neste trabalho. Os fatores $f4$ e $f8$ possuem funções de características do tipo $1_{\{y_t=i\}}1_{\{y_{t-1}=j\}}$, já o restante seguem o formato $1_{\{y_t=i\}}1_{\{x_t=o\}}$.*

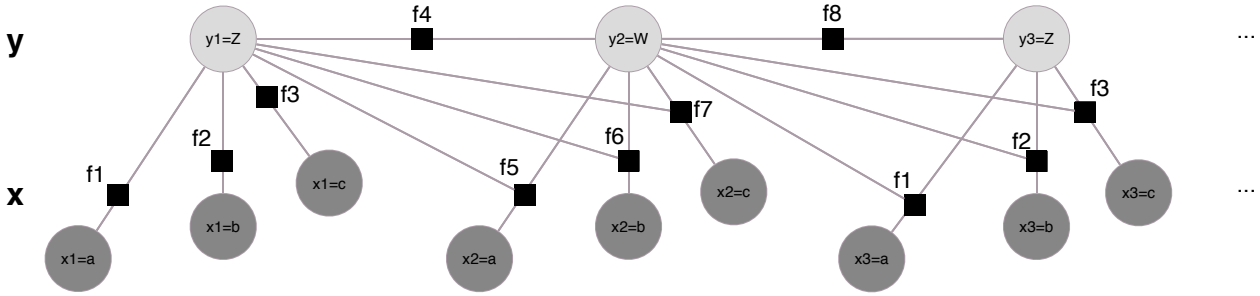

Figura 3.7: *LCCRF semelhante ao da figura 3.6 onde que a transição também depende da observação atual. Novamente os fatores $f4$ e $f8$ possuem funções de características do tipo $1_{\{y_t=i\}}1_{\{y_{t-1}=j\}}$, e o restante seguem o formato $1_{\{y_t=i\}}1_{\{y_{t-1}=j\}}1_{\{x_t=o\}}$.*

3.3 Algoritmos de Inferência para Campos Aleatórios Condicionais de Cadeias Lineares

Devido ao fato do treinamento de LCCRFs utilizar valores calculados pelos algoritmos de inferências *forward* e *backward*, nesta seção apresentaremos estes algoritmos seguindo a notação que Sutton e McCallum (2006) utilizaram em seu trabalho. Cabe destacar que todos os algoritmos apresentados utilizam a técnica de programação dinâmica para que possam ser calculados em tempo eficiente, e são bastante relacionados com suas variantes para HMMs e GHMMs.

Antes de apresentar os algoritmos, vamos definir novamente um LCCRF, já que esta definição é importante para o entendimento dos mesmos. Assim, seja um LCCRF definido como:

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \prod_{t=1}^{N} \Psi_t(y_t, y_{t-1}, \mathbf{x}_t) \tag{3.10}$$

onde

$$Z(\mathbf{x}) = \sum_{y} \prod_{t=1}^{N} \Psi_t(y_t, y_{t-1}, \mathbf{x}_t) \tag{3.11}$$

e onde cada fator é definido como

$$\Psi_t(y_t, y_{t-1}, \mathbf{x}_t) = \exp\{\sum_{k=1}^{K} \lambda_k f_k(y_t, y_{t-1}, \mathbf{x}_t)\} \tag{3.12}$$

Uma das tarefas mais comuns de inferência em LCCRFs é a de se encontrar qual sequência de estados ocultos y melhor descreve a sequência de estados observados x de tamanho T. Essa tarefa pode ser executada a partir do algoritmo de *Viterbi*, que encontra $y^* = \arg\max_y p(y|x)$, também conhecido como caminho de *Viterbi*, e pode ser calculado a partir da seguinte recursão:

$$\delta_t(j) = \max_{i \in S} \Psi_t(j, i, x_t)\delta_{t-1}(i) \tag{3.13}$$

Outros dois algoritmos bastante comuns em HMM são o *Forward* e o *Backward*. O algoritmo *Forward* consiste no cálculo de $\alpha_t(j) = p(x_{1\ldots t}, y_t = j)$, onde $x_{1\ldots t}$ é a sequência dos t primeiros símbolos de x e pode ser obtido eficientemente a partir da seguinte recursão:

$$\alpha_t(j) = \sum_{i \in S} \Psi_t(j, i, x_t)\alpha_{t-1}(i) \tag{3.14}$$

onde

$$Z(x) = \sum_{y_T} \alpha_T(y_T). \tag{3.15}$$

Já o algoritmo *Backward* é definido como $\beta_t(i) = p(x_{t+1\ldots T}, y_t = i)$, onde $x_{t+1\ldots T}$ é a sequência dos últimos t símbolos de x e pode ser eficientemente calculado pela seguinte recursão:

$$\beta_t(i) = \sum_{j \in S} \Psi_{t+1}(j, i, x_{t+1})\beta_{t+1}(j) \tag{3.16}$$

E de forma semelhante, temos que

$$Z(x) = \beta_0(y_0) \tag{3.17}$$

Combinando os resultados obtidos pela computação dos algoritmos *Forward* e *Backward* podemos calcular a distribuição marginal, que é utilizada durante a etapa de treinamento de LCCRFs, como definido na seguinte equação:

$$p(y_{t-1}, y_t|x) = \alpha_{t-1}(y_{t-1})\Psi_t(y_t, t_{t-1}, x_t)\beta_t(y_t) \tag{3.18}$$

Outra forma de inferência em campos aleatórios condicionais é a conhecida *posterior decoding* descrita em 3.19.

$$\hat{\pi}_i = \arg\max_k p(y_i = k|x) \tag{3.19}$$

onde

$$p(y_i = k|x) = \frac{\alpha_k(i)\beta_k(i)}{P(x)} \tag{3.20}$$

3.4 Treinamento de Campos Aleatórios Condicionais de Cadeias Lineares

A etapa de treinamento de um LCCRF consiste em se encontrar os parâmetros $\theta = \{\lambda_k\}$. Seja, então, um conjunto de treinamento $\mathcal{D} = \{x^{(i)}, y^{(i)}\}_{i=1}^{N}$, onde $x^{(i)} = \{x_1^{(i)}, x_2^{(i)}, ...x_T^{(i)}\}$ é a sequência de estados observados de entrada e $y^{(i)} = \{y_1^{(i)}, y_2^{(i)}, ...y_T^{(i)}\}$ é a sequência de símbolos que rotulam

x. Assim, podemos obter os parâmetros θ ótimos a partir da maximização do log da probabilidade condicional definida por $l(\theta)$.

$$l(\theta) = \sum_{i=1}^{N} \log p(y^{(i)}|x^{(i)}) \tag{3.21}$$

Substituindo 3.4 em 3.21 temos

$$l(\theta) = \sum_{i=1}^{N} \sum_{t=1}^{T} \sum_{k=1}^{K} \lambda_k f_k(y_t^{(i)}, y_{t-1}^{(i)}, x_t^{(i)}) - \sum_{i=1}^{N} \log Z(x^{(i)}) \tag{3.22}$$

Podemos melhorar $l(\theta)$ adicionando uma penalidade baseada na norma Euclidiana de θ e no parâmetro de regularização $1/2\sigma^2$, onde σ^2 é somente um parâmetro que indica a intensidade dessa penalidade. Essa adição é conhecida como regularização e é necessária para que evitemos o *overfitting*, ou seja, para que o nosso modelo tenha a capacidade de generalização e não fique preso somente ao exemplos contidos no conjunto de treinamento (Sutton e McCallum, 2006). Sendo assim temos o log da probabilidade condicional regularizada

$$l(\theta) = \sum_{i=1}^{N} \sum_{t=1}^{T} \sum_{k=1}^{K} \lambda_k f_k(y_t^{(i)}, y_{t-1}^{(i)}, x_t^{(i)}) - \sum_{i=1}^{N} \log Z(x^{(i)}) - \sum_{k=1}^{K} \frac{\lambda_k^2}{2\sigma^2} \tag{3.23}$$

e suas derivadas parciais definidas como

$$\frac{\partial l}{\partial \lambda_k} = \sum_{i=1}^{N} \sum_{t=1}^{T} f_k(y_t^{(i)}, y_{t-1}^{(i)}, x_t^{(i)}) - \sum_{i=1}^{N} \sum_{t=1}^{T} \sum_{y,y'} f_k(y, y', x^{(i)}) p(y, y'|x^{(i)}) - \sum_{k=1}^{K} \frac{\lambda_k}{\sigma^2} \tag{3.24}$$

Uma característica interessante é que $l(\theta)$ é estritamente concava, ou seja possui um único ótimo global.

De modo geral $l(\theta)$ não pode ser maximizada em sua forma fechada, sendo necessário um método numérico como o BFGS (Broyden-Fletcher-Goldfarb-Shanno) (Sutton e McCallum, 2006) que é uma aproximação do método de Newton. Vale notar que o treinamento é uma etapa bastante custosa computacionalmente, isso porque o cálculo de $Z(x)$ e da distribuição marginal $p(y_t, y_{t-1}|x)$ são obtidos através dos algoritmos *forward backward* que têm no caso geral complexidade $O(TKM^2)$, sendo K o número de funções de características e M o número de estados não observáveis possíveis. Esse algoritmo é computado para cada instância do conjunto de treinamento e para cada cálculo do gradiente num total de complexidade $O(TKM^2NG)$, sendo G o número de gradientes calculados.

3.5 Aplicações de Campos Aleatórios Condicionais de Cadeias Lineares

Após a introdução dos Campos Aleatórios Condicionais por John Lafferty e colegas em 2001 (Lafferty *et al.*, 2001), diversas aplicações foram propostas.

Sato e Sakakibara utilizaram CRFs para o alinhamento estrutural de RNA (Sato e Sakakibara , 2005). Um ponto de destaque neste trabalho é que mesmo com um pequeno número de amostras para o treinamento do modelo, o método desenvolvido mostrou-se mais eficiente do que outras abordagens existentes.

Wang e Xu (2011) investigaram o uso de LCCRFs para a modelagem da relação entre a estrutura terciária de um RNA e sua sequência. As funções de características foram extraídas de informações sobre cada nucleotídeo tais como seu tipo, o tipo dos nucleotídeos vizinhos e pareamento. Os autores obtiveram resultados bastante promissores se comparados com outras abordagens mais tradicionais.

Na área de segmentação de palavras em chinês temos o trabalho de Peng *et al.* (2004). Diferente de línguas ocidentais como português e inglês, palavras em chinês não são separadas por espaços em branco e portanto a segmentação é considerada uma tarefa desafiadora. O sistema desenvolvido obteve desempenho competitivo, além de permitir que os autores estudassem novas formas de modelagem para a detecção de palavras em chinês.

Finalmente podemos mencionar o trabalho de Feng e McCallum que utilizaram LCCRFs para o reconhecimento de palavras manuscritas como alternativa à HMMs (Feng *et al.*, 2006). Neste trabalho além de apresentam as vantagens do uso de LCCRFs também apresentam uma forma alternativa de treinamento chamada de *Beam Search* visando melhoria do tempo de treinamento.

Capítulo 4

Estendendo o ToPS

ToPS (do inglês *Toolkit of Probabilistc Model of Sequence*) é um arcabouço orientado a objetos para manipular modelos probabilísticos que utilizaremos como base para a implementação de LCCRFs. Foi inicialmente desenvolvido por Kashiwabara (2012) com o objetivo facilitar a caracterização de sequências heterogêneas onde variados modelos probabilísticos são necessários. Em particular, Kashiwabara estava interessado no desenvolvimento de preditores de genes, área em que é comum a utilização de um modelo agregador e diversos submodelos. Porém ToPS é um arcabouço de uso genérico capaz de manipular qualquer sequência de símbolos de um alfabeto discreto.

Outro ponto positivo do ToPS é que, além dos modelos probabilísticos disponíveis temos também um conjunto de ferramentas que facilitam a experimentação. Todas essas ferramentas recebem como entrada um arquivo de texto que descreve os modelos probabilísticos. Este arquivo é definido utilizando um linguagem bem simples e bastante próxima da linguagem matemática usual. Todas essas características aceleram o desenvolvimento de modelos e facilitam uma investigação mais profunda do problema, já que alterar parâmetros ou até mesmo arquiteturas é necessário apenas uma simples edição de um arquivo de texto.

Este arcabouço foi desenvolvido utilizando os padrões de projeto descritos por Gamma E. *et al.* (1994) o que facilita o entendimento e extensão do código existente. Neste capítulo descrevemos as alterações e adições necessárias para que o arcabouço seja compatível com o modelo LCCRF.

Como já apresentamos neste trabalho, as implementações de CRF disponíveis são normalmente direcionadas à um problema específico e não tem como objetivo servir como base para a criação de modelos genéricos. Mesmo arcabouços mais genéricos são limitados e não permitem a combinação de outros modelos probabilísticos.

Nossa implementação de CRF como uma extensão do ToPS tem por objetivo resolver esses problemas.

4.1 Ferramentas

Como já mencionado, ToPS possui um série de ferramentas que auxiliam no desenvolvimento de modelos probabilísticos. A partir destas, o ToPS pode ser utilizado para gerar sequências aleatórias, segmentar sequências e treinar modelos. A figura 4.1 apresenta de forma resumida todas as ferramentas que o arcabouço fornece, sendo que as caixas retangulares representam arquivos ou processos manuais e as caixas arredondadas representam os programas que executam operações específicas.

Os arquivos de entrada apresentados na figura 4.1 são definidos utilizando uma linguagem própria do ToPS. Como exemplo, abaixo temos a definição de um HMM para o problema do cassino desonesto. Neste hipotético problema, um cassino utiliza dois tipos de dados revezando-os: um honesto, cuja probabilidade de sair qualquer face em um lançamento de dado é a mesma, e outro desonesto, cuja probabilidade de sair uma das faces é maior do que a das outras. Em nosso exemplo, um lançamento do dado desonesto tem probabilidade de 50% de sair a face número 1 e 10% para o restante.

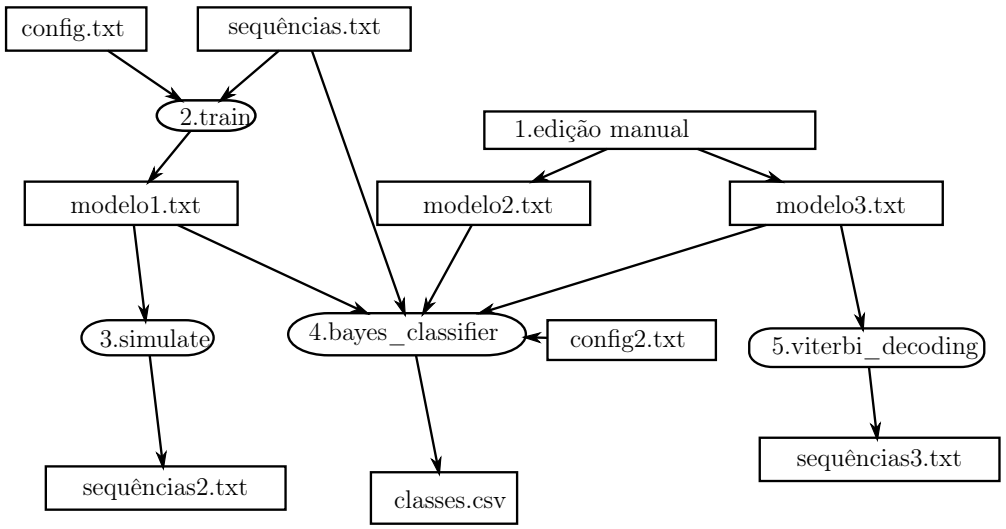

Figura 4.1: *Diagrama de uso do ToPS. Caixas retangulares representam arquivos com os dados ou processos manuais, caixas arredondadas representam programas. (Kashiwabara, 2012)*

```
model_name="HiddenMarkovModel"

state_names= ("Fair", "Loaded" )

observation_symbols= ("1", "2", "3", "4", "5", "6" )

transitions = ("Loaded" | "Fair": 0.1;
                          "Fair" | "Fair": 0.9;
                          "Fair" | "Loaded": 0.1;
                          "Loaded"| "Loaded": 0.9 )

emission_probabilities = ("1" | "Fair" : 0.166666666666;
                          "2" | "Fair" : 0.166666666666;
                          "3" | "Fair" : 0.166666666666;
                          "4" | "Fair" : 0.166666666666;
                          "5" | "Fair" : 0.166666666666;
                          "6" | "Fair" : 0.166666666666;
                          "1" | "Loaded" : 0.5;
                          "2" | "Loaded" : 0.1;
                          "3" | "Loaded" : 0.1;
                          "4" | "Loaded" : 0.1;
                          "5" | "Loaded" : 0.1;
                          "6" | "Loaded" : 0.1)

initial_probabilities= ("Fair": 0.5; "Loaded": 0.5)
```

Observe que a definição das transições é bastante próxima da definição formal ($p(\text{"}Loaded\text{"} \mid \text{"}Fair\text{"}) = 0.1$), e o mesmo acontece com as probabilidades de emissão e as probabilidades iniciais. Essa característica ajuda pessoas sem conhecimento prévio em programação a definir modelos probabilísticos. Uma vantagem importante desta abordagem é que ajustes nos modelos podem ser especificados e testados rapidamente.

4.2 Arquitetura

A implementação original do ToPS tem como núcleo uma arquitetura de classes cuja raiz é *ProbabilisticModel*, como pode ser observado na figura 4.2. Vale ressaltar, no segundo nível, a classe abstrata *DecodableModel* que é responsável por caracterizar todos os modelos que implementam os

algoritmos *forward*, *backward* e *viterbi*. Portanto temos que *HiddenMarkovModel* e *GeneralizedHiddenMarkovModel* são subclasses de *DecodableModel*.

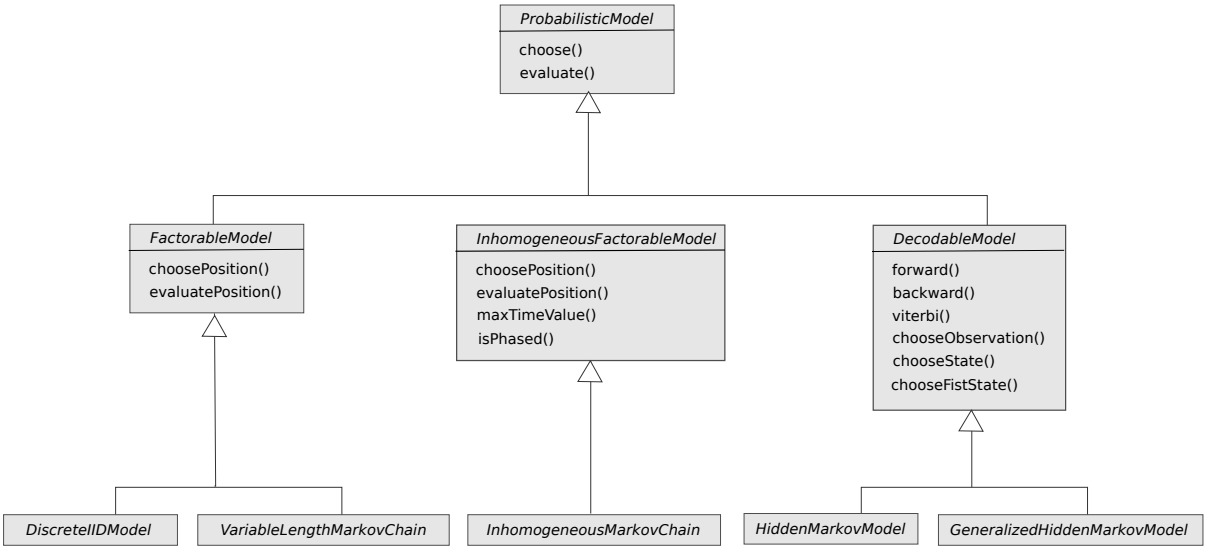

Figura 4.2: *Hierarquia de classes de* ProbabilisticModel. *(Kashiwabara, 2012)*

Essas observações são importantes, pois como veremos no seção 4.3, é a partir de *DecodableModel* que iniciaremos a extensão do arcabouço.

Outro ponto importante a se destacar aqui é que a modelagem dessa arquitetura central é de extrema importância para que o ToPS atinja seus objetivos. Lembrando que o problema atacado por Kashiwabara é o desenvolvimento de modelos para caracterizar sequências heterogêneas, e que para isso precisamos combinar diversos submodelos distintos. Sendo assim, temos, por exemplo, que o modelos agregador GHMM deve ser definido como um conjunto de modelos probabilísticos descendentes de *ProbabilisticModel*.

4.3 Adicionando Modelos Discriminativos

Anteriormente ToPS permitia apenas a definição de modelos geradores como HMM e GHMM, que dentro do arcabouço eram subclasses de *DecodableModel*. A principal característica de *DecodableModel* é definir uma interface para os algoritmos de decodificação *forward*, *backward* e *viterbi*, sendo assim uma classe que descreve LCCRFs deveria ser subclasse de *DecodableModel*. Entretanto, LCCRFs são bastante diferentes dos modelos HMM e GHMM como já foi discutido nos capítulos anteriores.

ToPS definia os métodos *choosePath*, *chooseObservation* e *chooseState* para *DecodableModel*. Este métodos são importantes para modelos geradores, pois são capazes de sortear e gerar sequências aleatórias. Mas sendo LCCRF um modelo discriminativo essas definições não fazem sentido. Assim, nossa solução foi modificar a arquitetura de modelos probabilísticos do ToPS adicionando duas novas subclasses de *DecodableModel*: *DiscriminativeModel*, que representa os modelos discriminativos, e *GenerativeModel*, que representa os modelos geradores e define os métodos *choosePath*, *chooseObservation* e *chooseState*. Uma representação gráfica desta soluções é apresentada na figura 4.3.

Essa modificação na hierarquia dos modelos probabilísticos requereu algumas alterações nos modelos geradores já existentes. As classes *GeneralizedHiddenMarkovModel* e *HiddenMarkovModel* que representavam respectivamente os modelos GHMM e HMM eram subclasses de *DecodableModel*, como podemos observar na figura 4.2, entretanto, nesta nova proposta são subclasses de *GenerativeModel*.

Essa arquitetura é importante pois a adição de novos modelos discriminativos é facilitada.

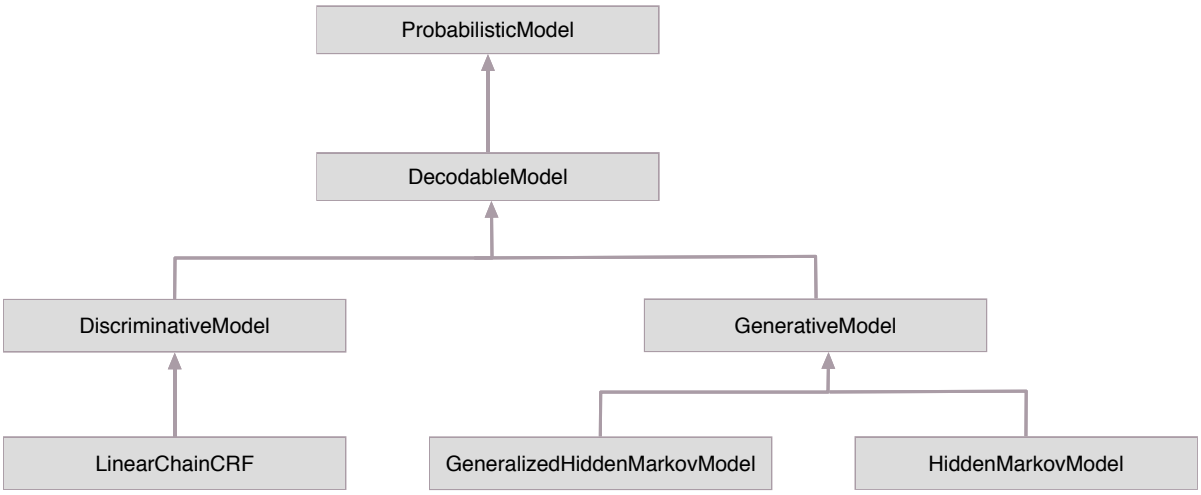

Figura 4.3: *Diagrama de classes dos modelos probabilísticos proposto para a adição de modelos discriminativos*

Novas variantes de CRF, ou outros modelos discriminativos podem ser facilmente adicionadas na arquitetura como subclasses de *DiscriminativeModel*, como foi feito com *LinearChainCRF*.

4.4 Funções de Características

Outro ponto importante necessário para essa extensão do ToPS é a possibilidade de se modelar funções de características. Essas funções são de extrema importância na definição de CRFs.

Comumente, essas funções são compostas por subfunções simples do tipo $1_{\{a=b\}}$ que representam as funções cujo valor é 1 se a é igual a b, e 0 caso contrário. Podemos observar que essa nova hierarquia segue o padrão de projeto *Composite* descrito por Gamma E. *et al.* (1994) como apresentado na figura 4.4 e nos permite tratar de maneira uniforme funções de características e conjuntos de funções de características. Vale ressaltar que as funções de características não se limitam as funções do tipo $1_{\{a=b\}}$ e podem fazer uso de todos os modelos probabilísticos disponíveis no ToPS.

Essa hierarquia, apresentada na figura 4.4, é encabeçada pela classe abstrata *AbstractFeatureFunction*, que define o método *eval* que tem por objetivo avaliar a função em um dado instante i para uma certa transição de rótulos e um vetor de observações.

Fazem parte desta hierarquias algumas classes pré-definidas que representam as funções de características mais comuns:

* *FeatureFunctionCurrentLabel*: Representa as funções $1_{\{y_t=a\}}$, ou seja, verifica se em um dado instante t o rótulo atual é a.
* *FeatureFunctionPreviousLabel*: Representa as funções $1_{\{y_{t-1}=a\}}$, ou seja, verifica se em um dado instante t o rótulo anterior é a.
* *FeatureFunctionObservation*: Representa as funções $1_{\{o_{t-i}=a\}}$, ou seja, verifica se em um dado instante t a observação $t+1$ é a, para $i \in \mathbb{Z}$.
* *FeatureFunctionProbabilistModel*: Representa as funções de características que avaliam a sequência para um dado modelo probabilístico.
* *Feature*: Agrega funções de características para a criação de funções mais complexas.
* *Factor*: Agrega objetos do tipo *Feature* para a criação de fatores.

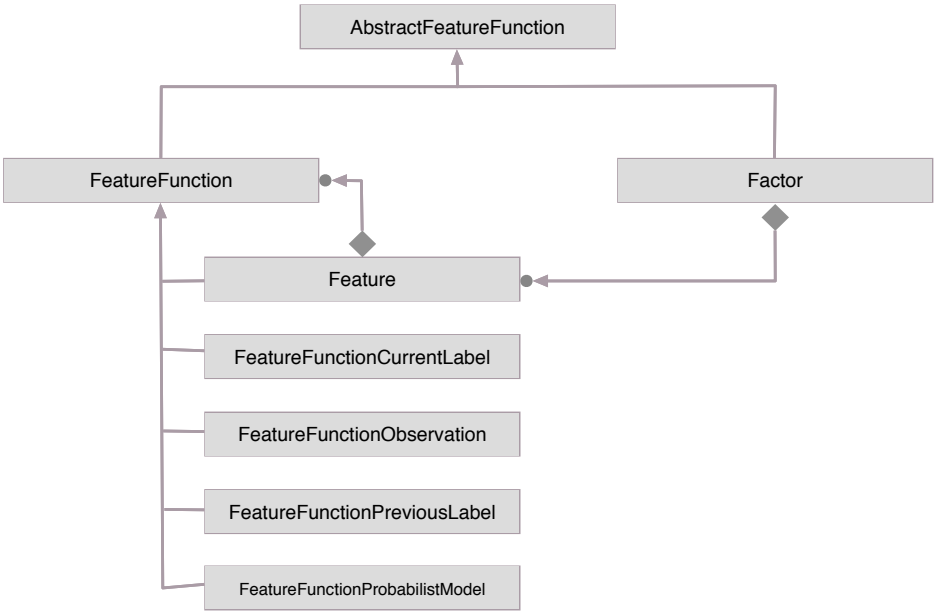

Figura 4.4: *Hierarquia de classes proposta para a definição de funções de características*

Além dessas, podem ser definidas novas classes para operações mais específicas. Em bioinformática, o uso de informações extrínsecas, ou seja, informações além das obtidas da análise da sequência a ser segmentada em si, podem melhorar alguns processos (DeCaprio *et al.*, 2007). Sendo assim a definição de uma função de característica que acesse um banco de dados ou obtenha informações de um processo externo pode ser modelada como uma subclasse de *FeatureFunction*.

4.5 Linguagem

Um dos pontos principais no desenvolvimento do ToPS foi a atenção dada em facilitar o uso e a criação de modelos. Parte deste objetivo foi alcançado através da definição de uma linguagem bastante próxima da linguagem matemática, como foi descrito na seção 4.1.

Para que nossa extensão ficasse completa, adicionamos também uma extensão à linguagem de definição de modelos do ToPS. Para isso, refatoramos a linguagem e adicionamos novas regras para que fosse possível criar LCCRFs tão facilmente quanto qualquer outro modelo disponível no ToPS.

A listagem completa da gramática original e de sua extensão podem ser observadas no apêndice A.

4.5.1 Redefinição da Linguagem Original

Anteriormente a linguagem do ToPS foi definida utilizando um pacote C++ chamado Spirit (http://boost-spirit.com/home/). Além disso, com a evolução do arcabouço probabilístico, a linguagem cresceu e seu projeto inicial passou a ser pouco flexível. Como a adição de LCCRFs ao ToPS, decidimos por refatorar a definição da linguagem utilizando a forma EBNF (do inglês *Extended Backus-Naur Form*) e implementá-la utilizando os pacotes *Flex* (http://flex.sourceforge.net/) e *Bison* (http://www.gnu.org/software/bison/).

Esta tarefa proporcionou uma simplificação na implementação da gramática, o que facilita o entendimento e possibilita uma maior evolução do arcabouço. Por exemplo, implementamos um módulo acoplado ao interpretador de modelos que exibe mensagens detalhadas quando existe algum erro de sintaxe, indicando, entre outros, a posição do erro e qual o símbolo esperado naquela ocasião.

4.5.2 Extensão da Linguagem

Após a redefinição da linguagem original, desenvolvemos uma extensão para suportar a definição de LCCRFs. Nossa extensão permite a definição de funções de características do tipo $1_{\{t_t=Loaded\}}1_{\{t_{t-1}=Fair\}}$:

```
1 t_fair_to_loaded = ({"current" : "Loaded"} , {"previous" : "Fair"}  )
```

E também permite a definição de funções de características que agregam submodelos. Como exemplo, temos abaixo a definição de um função de característica que utiliza um *DiscreteIIDModel* para modelar a observação de um único símbolo dado que o estado atual é *Fair*. Note que como queremos analisar um único símbolo, definimos *upper_ endpoint* e *lower_ endpoint* como zero, pois estes parâmetros especificam a largura da janela centralizada na posição atual.

```
1 fair_model = [
2   model_name = "DiscreteIIDModel"
3   alphabet = ("1","2","3","4","5","6")
4   probabilities = (0.333333, 0.333333, 0.333333, 0.333333, 0.333333, 0.333333)
5 ]
6
7 fair_feature_function = [
8   model = fair_model
9   upper_endpoint = 0
10  lower_endpoint = 0
11 ]
12
13 feature_7 = ({"current" : "Fair"} , {"model" : fair_feature_function})
```

Lembrando que um LCCRF possui uma coleção de funções de características, devemos especificar quais são as funções de características pertencentes ao modelo. Isso é realizado definindo o parâmetro *factors*. Note que cada função de característica tem um peso associado:

```
1 factors = (
2   t_fair_to_loaded      : -1.11095,
3   t_fair_to_fair        : 1.15319,
4   t_loaded_to_loaded    : 1.06853,
5   t_loaded_to_fair      : -1.11077,
6   e_fair_1              : -0.391067,
7   e_fair_2              : -0.116853,
8   e_fair_3              : -0.120932,
9   e_fair_4              : 0.230136,
10  e_fair_5              : 0.216345,
11  e_fair_6              : 0.224797,
12  e_loaded_1            : 0.391067,
13  e_loaded_2            : 0.116854,
14  e_loaded_3            : 0.120932,
15  e_loaded_4            : -0.230136,
16  e_loaded_5            : -0.216345,
17  e_loaded_6            : -0.224797
18 )
```

Por fim, abaixo temos a modelagem do problema do cassino desonesto utilizando a linguagem de descrição de modelos do ToPS:

```
1 model_name = "LinearChainConditionalRandomField"
2 state_names = ("Fair", "Loaded")
3 observation_symbols = ("1", "2", "3", "4", "5", "6")
4
5 ## funções de características
```

```
t_fair_to_loaded     = ({"current" : "Loaded"} , {"previous" : "Fair"}   )
t_fair_to_fair       = ({"current" : "Fair"}   , {"previous" : "Fair"}   )
t_loaded_to_loaded   = ({"current" : "Loaded"} , {"previous" : "Loaded"})
t_loaded_to_fair     = ({"current" : "Fair"}   , {"previous" : "Loaded"})
e_fair_1             = ({"current"  : "Fair"}   , {"observation" : "1"}  )
e_fair_2             = ({"current"  : "Fair"}   , {"observation" : "2"}  )
e_fair_3             = ({"current"  : "Fair"}   , {"observation" : "3"}  )
e_fair_4             = ({"current"  : "Fair"}   , {"observation" : "4"}  )
e_fair_5             = ({"current"  : "Fair"}   , {"observation" : "5"}  )
e_fair_6             = ({"current"  : "Fair"}   , {"observation" : "6"}  )
e_loaded_1           = ({"current"  : "Loaded"}, {"observation" : "1"}  )
e_loaded_2           = ({"current"  : "Loaded"}, {"observation" : "2"}  )
e_loaded_3           = ({"current"  : "Loaded"}, {"observation" : "3"}  )
e_loaded_4           = ({"current"  : "Loaded"}, {"observation" : "4"}  )
e_loaded_5           = ({"current"  : "Loaded"}, {"observation" : "5"}  )
e_loaded_6           = ({"current"  : "Loaded"}, {"observation" : "6"}  )

## fatores
factors = (
  t_fair_to_loaded     : -1.11095,
  t_fair_to_fair       : 1.15319,
  t_loaded_to_loaded   : 1.06853,
  t_loaded_to_fair     : -1.11077,
  e_fair_1             : -0.391067,
  e_fair_2             : -0.116853,
  e_fair_3             : -0.120932,
  e_fair_4             : 0.230136,
  e_fair_5             : 0.216345,
  e_fair_6             : 0.224797,
  e_loaded_1           : 0.391067,
  e_loaded_2           : 0.116854,
  e_loaded_3           : 0.120932,
  e_loaded_4           : -0.230136,
  e_loaded_5           : -0.216345,
  e_loaded_6           : -0.224797
)
```

Este arquivo de configuração descreve funções de características que correspondem à transições entre estados em um HMM (*t_fair_to_loaded*, *t_fair_to_fair*, *t_loaded_to_loaded* e *t_loaded_to_fair*) e outras que correspondem à emissão de símbolos dado um estado atual (*e_fair_1*, *e_fair_2*, ..., *e_loaded_1*, *e_loaded_2*, ...).

Além da definição de modelos, a linguagem também permitem definirmos arquivos que configuram treinamentos de LCCRFS. Abaixo temos um exemplo:

```
training_algorithm="LCCRF-LBFGS"
initial_model="cassino.txt"
observation_training_set="train.seq"
label_training_set="train.hid"
```

Onde definimos o algoritmo a ser usado, o modelo inicial e o conjunto de treinamento. Lembrando que o treinamento é supervisionado e que o conjunto de treinamento é definido com dois arquivos diferentes: um contendo as sequências de observações e outro contendo os rótulos dessas observações.

4.6 Algoritmos de Inferência

Os algoritmos *forward*, *backward* e *viterbi* foram descritos no capítulo 3. Entretanto, um implementação direta das definições destes algoritmos pode não ser viável para modelos mais complexos ou para um grande volume de dados devido a sua complexidade computacional. No ToPS, desenvolvemos uma versão que paraleliza a computação destes algoritmos e a descreveremos nesta seção.

4.6.1 Cálculo de Probabilidade com Computação de Matrizes

Antes de discutirmos a paralelização dos algoritmos, iremos definir como calcular as probabilidades e algoritmos utilizando matrizes de fatores. Lembrando que LCCRF são representados como os grafos de fatores que foram apresentados no capítulo 3.

Lembrando que a definição de LCCRF é:

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} exp\{\sum_{t=1}^{N}\sum_{k=1}^{K}\lambda_k f_k(y_t, y_{t-1}, \mathbf{x}_t)\} \tag{4.1}$$

onde $Z(x)$ é definido como

$$Z(\mathbf{x}) = \sum_{y} exp\{\sum_{t=1}^{N}\sum_{k=1}^{K}\lambda_k f_k(y_t, y_{t-1}, \mathbf{x}_t)\} \tag{4.2}$$

Utilizaremos a notação de Wallach (2004) para o calculo de $p(y|x)$. Seja $\mathbb{L}$ o conjunto de rótulos e sejam y e y' pertencentes a $\mathbb{L}$. Podemos definir um conjunto de matrizes de fatores $\{M_i(y', y|x)|i = 1, ..., n+1\}$ onde $M_i(y', y|x)$ é uma matriz de tamanho $|\mathbb{L} \times \mathbb{L}|$ cujos elementos são:

$$M_i(y', y|x) = \exp(\sum_{j} \lambda_j f_j(y', y, x, i)) \tag{4.3}$$

Sendo assim, a partir de 4.3 e 4.1 temos

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \prod_{i=1}^{n+1} M_i(y_{i-1}, y_i|x) \tag{4.4}$$

Com isso podemos redefinir a recursão do algoritmo de *viterbi*:

$$\delta_t(j) = \max_{i \in S} M_t(i, j|x_t)\delta_{t-1}(i) \tag{4.5}$$

bem como o cálculo do algoritmo *forward*:

$$\alpha_t(j) = \sum_{i \in S} M_t(j, i|x_t)\alpha_{t-1}(i) \tag{4.6}$$

e do algoritmo *backward*:

$$\beta_t(i) = \sum_{j \in S} M_{t+1}(j, i|x_{t+1})\beta_{t+1}(j) \tag{4.7}$$

4.6.2 Paralelização

Originalmente a descrição do algoritmos utilizando matrizes de fatores não define como o conjunto de matrizes deve ser calculado. Note que o cálculo de $M_i(x)$ depende somente da posição i na sequência de estados observáveis, e portanto pode ser facilmente paralelizado já que não existem variáveis compartilhadas.

Sendo assim desenvolvemos uma implementação em que dividimos a computação do conjunto de matrizes de fatores em um número configurável de linhas de execução (ou em inglês, *threads*). Uma vez calculado, este conjunto pode ser reaproveitado no cálculo dos algoritmos *viterbi*, *forward*, *backward* e *posterior decoding*.

Vale ressaltar que essa abordagem tem maior impacto quando mais de um algoritmo é executado dado um modelo e uma sequência. Isso porque depois de calculado o conjunto de matrizes de fatores, os algoritmos *viterbi*, *forward*, *backward* e *posterior decoding* executam em tempo $O(TL^2)$, onde T o tamanho da sequência e L o número de rótulos possíveis. Entretanto, o cálculo de todo o conjunto de matrizes de fatores tem complexidade $O(TL^2K)$, que é a parte paralelizada do algoritmo para

que o impacto no tempo total seja amenizado, onde T o tamanho da sequência e L o número de rótulos possíveis e K o número de funções de características definidas no modelo. Discutiremos mais sobre o tempo de execução no próximo capítulo.

4.7 Treinamento

Como descrito no capítulo 3, o treinamento é realizado a partir da maximização do log da probabilidade condicional:

$$l(\theta) = \sum_{i=1}^{N} \log p(y^{(i)}|x^{(i)}) \tag{4.8}$$

onde $l(\theta)$ não pode ser maximizada em sua forma fechada, sendo necessário algum método numérico para isso. Nossa escolha foi pelo LBFGS (do inglês, Limited-Memory Broyden-Fletcher-Goldfarb-Shanno) que é uma variação do algoritmo original BFGS (Broyden-Fletcher-Goldfarb-Shanno) que utiliza menos memória e é citado no artigo de Sutton e McCallum (Sutton e McCallum, 2006) como uma boa solução para o problema.

Para tanto utilizamos uma biblioteca de código aberto chamada libLBFGS (Okazaki, 2007b). Vale ressaltar que utilizamos regularização, como apontado no capítulo 3, e portanto implementamos o cálculo da função objetivo e das derivadas parciais:

$$l(\theta) = \sum_{i=1}^{N} \sum_{t=1}^{T} \sum_{k=1}^{K} \lambda_k f_k(y_t^{(i)}, y_{t-1}^{(i)}, x_t^{(i)}) - \sum_{i=1}^{N} \log Z(x^{(i)}) - \sum_{k=1}^{K} \frac{\lambda_k^2}{2\sigma^2} \tag{4.9}$$

$$\frac{\partial l}{\partial \lambda_k} = \sum_{i=1}^{N} \sum_{t=1}^{T} f_k(y_t^{(i)}, y_{t-1}^{(i)}, x_t^{(i)}) - \sum_{i=1}^{N} \sum_{t=1}^{T} \sum_{y,y'} f_k(y, y', x^{(i)}) p(y, y'|x^{(i)}) - \sum_{k=1}^{K} \frac{\lambda_k}{\sigma^2} \tag{4.10}$$

Note que o cálculo da distribuição marginal $p(y_{t-1}, y_t|x)$ depende da computação dos algoritmos *forward* e *backward*, e sendo assim aqui também aproveitamos das otimizações realizadas na seção 4.6.

$$p(y_{t-1}, y_t|x) = \alpha_{t-1}(y_{t-1}) \Psi_t(y_t, t_{t-1}, x_t) \beta_t(y_t) \tag{4.11}$$

Capítulo 5

Testes e Validações

Neste capítulo apresentaremos as validações e testes que executamos a fim de se verificar o funcionamento da implementação de LCCRF. Todos os experimentos foram realizados em um *MacBook Air* com processador *Intel Core i5* de 1,7 GHz e 4 GB de memória RAM.

Para tanto, escolhemos abordar o problema do cassino desonesto, que consiste de um cassino que utiliza dois tipos de dados: Um honesto, onde todas as faces têm a mesma probabilidade de ocorrer em um lançamento, e um desonesto, cuja face número 1 tem maior probabilidade de acontecer do que as outras. Esse casino alterna o uso do dado desonesto com o honesto para dificultar que o jogador perceba tal fraude. Este problema pode ser modelado como o HMM da figura 5.1.

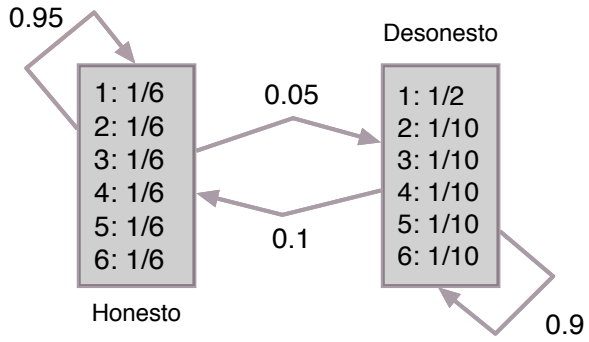

Figura 5.1: *HMM que modela o problema do cassino desonesto*

Este problema pode ser abordado utilizando um LCCRF, que de fato pode ser facilmente modelado para ser semelhante à um HMM. Essa característica nos ajuda a fazer comparações quanto ao tempo de execução dos algoritmos de treinamento e inferência além de ajudar na validação da implementação. Isso porque o ToPS nos fornece uma implementação já validada de HMM. Além disso a comparação entre HMM e LCCRF foi motivada por Sutton e McCallum (2006), que define um LCCRF incrementalmente a partir da definição de um HMM. O LCCRF de validação é similar ao apresentado na figura 3.6.

É importante notar que as funções de características deste LCCRF devem descrever as transições entre estados e as emissões de símbolos do HMM da figura 5.1, e que os seus respectivos parâmetros devem indicar a probabilidade de suas ocorrências. Desta forma, temos a tabela 5.1 que apresenta de forma resumida essas funções e os parâmetros necessários para a criação desse LCCRF que modela o problema do cassino desonesto.

Este mesmo LCCRF pode ser definido de outra maneira. Como as funções de características não se limitam a retornarem apenas 0's e 1's, podemos agregar submodelos probabilísticos à um

#	*Descrição*	*função de característica* $f_n(y, y', x)$	*Parâmetro* λ_n
1	Transição de Honesto para Desonesto	$1_{\{y=Honesto\}}1_{\{y'=Desonesto\}}$	$\log 0.05$
2	Transição de Honesto para Honesto	$1_{\{y=Honesto\}}1_{\{y'=Honesto\}}$	$\log 0.95$
3	Transição de Desonesto para Desonesto	$1_{\{y=Desonesto\}}1_{\{y'=Desonesto\}}$	$\log 0.90$
4	Transição de Desonesto para Honesto	$1_{\{y=Desonesto\}}1_{\{y'=Honesto\}}$	$\log 0.10$
5	Emissão do número 1 pelo estado Honesto	$1_{\{y=Honesto\}}1_{\{x=1\}}$	$\log 0.1667$
6	Emissão do número 2 pelo estado Honesto	$1_{\{y=Honesto\}}1_{\{x=2\}}$	$\log 0.1667$
7	Emissão do número 3 pelo estado Honesto	$1_{\{y=Honesto\}}1_{\{x=3\}}$	$\log 0.1667$
8	Emissão do número 4 pelo estado Honesto	$1_{\{y=Honesto\}}1_{\{x=4\}}$	$\log 0.1667$
9	Emissão do número 5 pelo estado Honesto	$1_{\{y=Honesto\}}1_{\{x=5\}}$	$\log 0.1667$
10	Emissão do número 6 pelo estado Honesto	$1_{\{y=Honesto\}}1_{\{x=6\}}$	$\log 0.1667$
11	Emissão do número 1 pelo estado Desonesto	$1_{\{y=Desonesto\}}1_{\{x=1\}}$	$\log 0.50$
12	Emissão do número 2 pelo estado Desonesto	$1_{\{y=Desonesto\}}1_{\{x=2\}}$	$\log 0.10$
13	Emissão do número 3 pelo estado Desonesto	$1_{\{y=Desonesto\}}1_{\{x=3\}}$	$\log 0.10$
14	Emissão do número 4 pelo estado Desonesto	$1_{\{y=Desonesto\}}1_{\{x=4\}}$	$\log 0.10$
15	Emissão do número 5 pelo estado Desonesto	$1_{\{y=Desonesto\}}1_{\{x=5\}}$	$\log 0.10$
16	Emissão do número 6 pelo estado Desonesto	$1_{\{y=Desonesto\}}1_{\{x=6\}}$	$\log 0.10$

Tabela 5.1: *Features e parâmetros do LCCRF similar ao HMM da figura 5.1*

#	*Descrição*	*função de característica* $f_n(y, y', x)$	*Parâmetro* λ_n
1	Transição de Honesto para Desonesto	$1_{\{y=Honesto\}}1_{\{y'=Desonesto\}}$	$\log 0.05$
2	Transição de Honesto para Honesto	$1_{\{y=Honesto\}}1_{\{y'=Honesto\}}$	$\log 0.95$
3	Transição de Desonesto para Desonesto	$1_{\{y=Desonesto\}}1_{\{y'=Desonesto\}}$	$\log 0.90$
4	Transição de Desonesto para Honesto	$1_{\{y=Desonesto\}}1_{\{y'=Honesto\}}$	$\log 0.10$
5	Emissão de símbolos pelo estado Honesto	$1_{\{y=Honesto\}}DiscreteIIDModel_H$	1
6	Emissão de símbolos pelo estado Desonesto	$1_{\{y=Desonesto\}}DiscreteIIDModel_D$	1

Tabela 5.2: *função de características e parâmetros do LCCRF similar ao HMM da figura 5.1 e ao LCCRF definido pelas funções de características da tabela 5.1*

LCCRF na forma de funções de características. Para o problema do cassino desonesto, podemos agregar dois *DiscreteIIDModels*, modelos já disponíveis no ToPS:

* $DiscreteIIDModel_H$: Representa a probabilidade de cada face sair em um dado honesto, ou seja, todas iguais à $\frac{1}{6}$
* $DiscreteIIDModel_D$: Representa a probabilidade de cada face sair em um dado desonesto, ou seja, $\frac{1}{2}$ para a face número 1 e $\frac{1}{10}$ para as outras.

Além disso, este LCCRF agregador têm mais 4 funções de características que indicam as transições entre os dados honesto e desonesto. Podemos visualizar mais facilmente esta definição na tabela 5.2. É simples notar que este LCCRF é semelhante ao HMM da figura 5.1 e LCCRF definido pelas funções de características da tabela 5.1.

5.1 Modelagem no ToPS

Durante este capítulo utilizaremos os modelos definidos acima em diversos testes e validações. Como referência, apresentaremos nesta seção como estes modelos são definidos.

O primeiro modelo apresentado neste capítulo é o HMM que modela o cassino desonesto da figura 5.1. Este modelo será utilizado como base para comparações e é definido pelo seguinte conjunto de instruções:

```
1 model_name="HiddenMarkovModel"
```

```
state_names= ("Fair", "Loaded" )
observation_symbols= ("1", "2", "3", "4", "5", "6" )
# transition probabilities
transitions =
("Loaded"    | "Fair": 0.05;
 "Fair"      | "Fair": 0.95;
 "Fair"      | "Loaded": 0.1;
 "Loaded"    | "Loaded": 0.9 )
# emission probabilities
emission_probabilities = ("1" | "Fair" : 0.166666666666;
                          "2" | "Fair" : 0.166666666666;
                          "3" | "Fair" : 0.166666666666;
                          "4" | "Fair" : 0.166666666666;
                          "5" | "Fair" : 0.166666666666;
                          "6" | "Fair" : 0.166666666666;
                          "1" | "Loaded" : 0.5;
                          "2" | "Loaded" : 0.1;
                          "3" | "Loaded" : 0.1;
                          "4" | "Loaded" : 0.1;
                          "5" | "Loaded" : 0.1;
                          "6" | "Loaded" : 0.1)
initial_probabilities= ("Fair": 0.5; "Loaded": 0.5)
```

Já o LCCRF que utiliza somente funções de características simples para modelar o problema do cassino desonesto e que é semelhante ao HMM da figura 5.1 é definido pelas funções de características da tabela 5.1, e pode ser modelado no ToPS com a seguinte configuração:

```
model_name = "LinearChainConditionalRandomField"
state_names = ("Fair","Loaded")
observation_symbols = ("1","2","3","4","5","6")

# features
feature_0 = ({"current" : "Loaded"} , {"previous" : "Fair"})
feature_1 = ({"current" : "Fair"} , {"previous" : "Fair"})
feature_2 = ({"current" : "Loaded"} , {"previous" : "Loaded"})
feature_3 = ({"current" : "Fair"} , {"previous" : "Loaded"})
feature_4 = ({"current" : "Fair"} , {"observation" : "1"})
feature_5 = ({"current" : "Fair"} , {"observation" : "2"})
feature_6 = ({"current" : "Fair"} , {"observation" : "3"})
feature_7 = ({"current" : "Fair"} , {"observation" : "4"})
feature_8 = ({"current" : "Fair"} , {"observation" : "5"})
feature_9 = ({"current" : "Fair"} , {"observation" : "6"})
feature_10 = ({"current" : "Loaded"} , {"observation" : "1"})
feature_11 = ({"current" : "Loaded"} , {"observation" : "2"})
feature_12 = ({"current" : "Loaded"} , {"observation" : "3"})
feature_13 = ({"current" : "Loaded"} , {"observation" : "4"})
feature_14 = ({"current" : "Loaded"} , {"observation" : "5"})
feature_15 = ({"current" : "Loaded"} , {"observation" : "6"})

# factors
factors = (
  feature_0 : -2.995732273553991,
  feature_1 : -0.05129329438755058,
  feature_2 : -0.10536051565782628,
  feature_3 : -2.3025850929940455,
  feature_4 : -1.791759269228075,
  feature_5 : -1.791759269228075,
  feature_6 : -1.791759269228075,
  feature_7 : -1.791759269228075,
  feature_8 : -1.791759269228075,
  feature_9 : -1.791759269228075,
  feature_10 : -0.6931471805599453,
  feature_11 : -2.3025850929940455,
```

```
37   feature_12 : -2.3025850929940455,
38   feature_13 : -2.3025850929940455,
39   feature_14 : -2.3025850929940455,
40   feature_15 : -2.3025850929940455
41 )
```

E por fim, o LCCRF que simplifica a modelagem criando funções de características que agregam submodelos e que é definido pelas funções descritas na tabela 5.1, é modelado no ToPS pela seguinte definição:

```
model_name = "LinearChainConditionalRandomField"
state_names = ("Fair","Loaded")
observation_symbols = ("1","2","3","4","5","6")

# submodel
loaded_model = [
  model_name = "DiscreteIIDModel"
  alphabet = ("1","2","3","4","5","6")
  probabilities = (0.5, 0.1, 0.1, 0.1, 0.1, 0.1)
]

fair_model = [
  model_name = "DiscreteIIDModel"
  alphabet = ("1","2","3","4","5","6")
  probabilities = (0.333333, 0.333333, 0.333333, 0.333333, 0.333333, 0.333333)
]

fair_feature_function = [
  model = fair_model
  upper_endpoint = 0
  lower_endpoint = 0
]

loaded_feature_function = [
  model = loaded_model
  upper_endpoint = 0
  lower_endpoint = 0
]

# features
feature_0 = ({"current" : "Loaded"} , {"previous" : "Fair"})
feature_1 = ({"current" : "Fair"} , {"previous" : "Fair"})
feature_2 = ({"current" : "Loaded"} , {"previous" : "Loaded"})
feature_3 = ({"current" : "Fair"} , {"previous" : "Loaded"})
feature_4 = ({"current" : "Fair"} , {"model" : fair_feature_function})
feature_5 = ({"current" : "Loaded"} , {"model" : loaded_feature_function})

# factors
factors = (
  feature_0 : -2.995732273553991,
  feature_1 : -0.05129329438755058,
  feature_2 : -0.10536051565782628,
  feature_3 : -2.3025850929940455,
  feature_4 : 1,
  feature_5 : 1
)
```

5.2 Paralelização

Nossa primeira investigação foi quanto a paralelização dos algoritmos de inferência *viterbi*, *forward* e *backward*. O objetivo foi desenvolver implementações corretas dos algoritmos que pu-

dessem explorar a capacidade de processamento paralelo dos computadores modernos.

Assim, utilizamos o ToPS para modelar o HMM da figura 5.1 e experimentamos 3 implementações diferentes de LCCRF no ToPS: Sem a computação das matrizes de fatores, com a computação das matrizes de fatores em uma única linha de execução, e com a computação das matrizes de fatores em 4 linhas de execução. Estes LCCRFs possuem as funções de características descritas na tabela 5.1. Utilizando o HMM foram geradas 470 sequências agrupadas em 47 conjuntos, sendo que cada conjunto é formado por 10 sequências de mesmo comprimento, variando de 100.000 a 4.700.000 símbolos. Para cada sequência, utilizando os modelos LCCRF (com suas 3 variações) e HMM, executamos os algoritmos *viterbi* e *posterior decoding*, obtendo seus rótulos e comparamos esses resultados.

Como esperado, as sequências segmentadas obtidas pelos HMM e LCCRF foram idênticas já que estes foram construídos para serem equivalentes.

Vale ressaltar algumas observações quanto a quantidade de tempo necessária para a execução dos algoritmos. Nas figuras 5.2 e 5.3 podemos ver a evolução na implementação dos algoritmos *viterbi* e *posterior decoding*. Estas implementações foram comparadas com a implementação padrão de HMM disponível no ToPS, representada pela linha vermelha nas figuras 5.2 e 5.3:

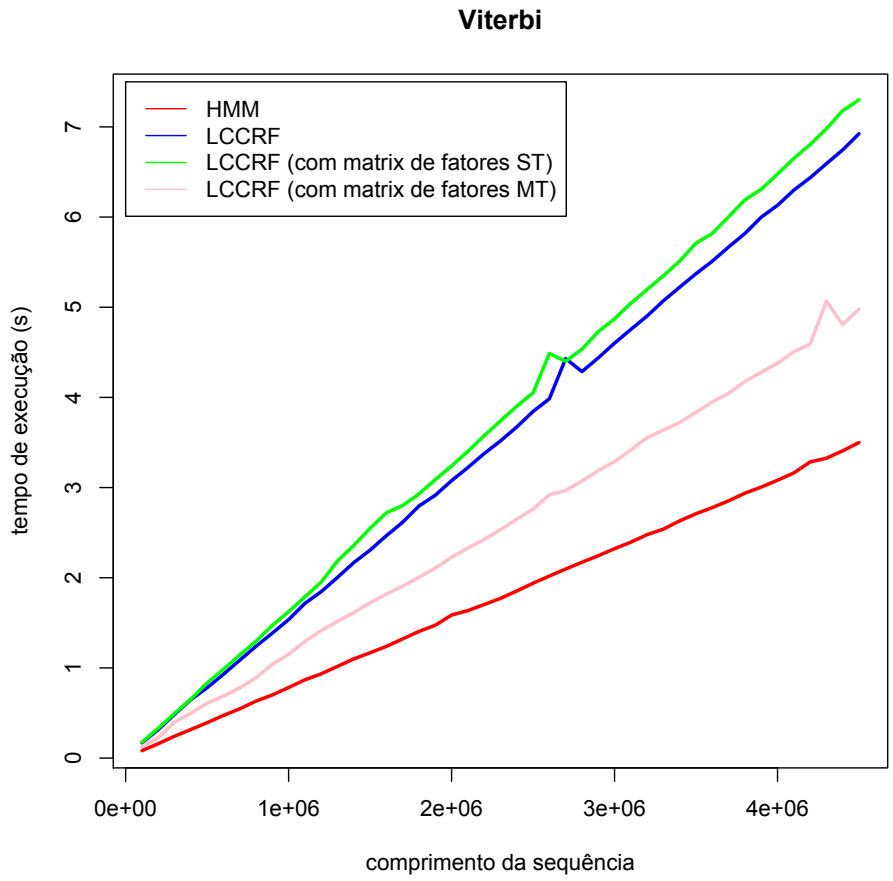

Figura 5.2: *Em vermelho, o tempo médio de execução do algoritmo* viterbi *em um HMM para o cassino desonesto. Em azul, verde e rosa, o tempo médio de execução de 3 variantes do algoritmo* viterbi *em um LCCRF para o cassino desonesto: implementação padrão, com pré-computação das matrizes de fatores, e com paralelismo na computação das matrizes de fatores, respectivamente.*

* Em azul temos o tempo de execução da implementação sem usar o pré-cálculo das matrizes de fatores. Com esperado, essa versão é bastante ineficiente se comparada com HMM.

* Em verde temos a implementação que faz uso das matrizes de fatores mas que não paraleliza sua computação. Note que como *posterior decoding* calcula *forward* e *backward* o desempenho

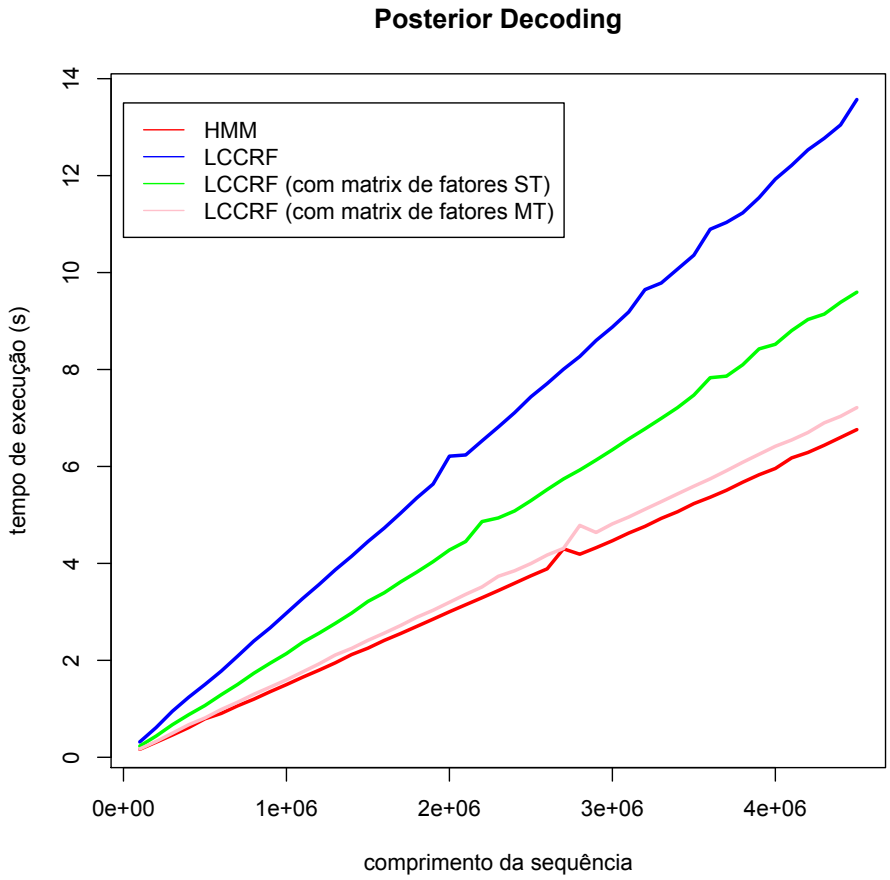

Figura 5.3: *Em vermelho, o tempo médio de execução do algoritmo* posterior decoding *em um HMM para o cassino desonesto. Em azul, verde e rosa, o tempo médio de execução de 3 variantes do algoritmo* posterior decoding *em um LCCRF para o cassino desonesto: implementação padrão, com pré-computação das matrizes de fatores, e com paralelismo na computação das matrizes de fatores, respectivamente*

melhorou com relação a implementação anterior já que os fatores foram calculados apenas uma única vez. Entretanto o algoritmo *viterbi* apresentou resultados piores. Isso pode ser explicado pelo fato de a pré-computação das matrizes de fatores calcula todos os fatores possíveis, e não somente aqueles que serão utilizados pela sequência analisada. Essa abordagem se justifica quando reaproveitamos essa pré-computação, como o que ocorreu com *posterior decoding*.

* Por fim, em rosa temos a versão paralelizada em 4 linhas de execução da computação das matrizes de fatores. Essa implementação aproxima-se bastante do tempo de execução dos respectivos algoritmos em HMM e é a implementação utilizada atualmente pelo ToPS.

5.3 Modelo Agregador

Além das validações de LCCRFs com funções de características simples, validamos o uso de LCCRF como modelo agregador.

Portanto, modelamos no ToPS o HMM da figura 5.1 e o LCCRF que agrega dois submodelos cujas funções de características são apresentadas na tabela 5.2.

Novamente utilizamos o HMM modelado no ToPS para gerar 470 sequências de lançamento de dados. Estas 470 sequências foram agrupadas em 47 conjuntos, sendo que cada conjunto contêm apenas 10 sequências de mesmo comprimento, variando de 100.000 a 4.700.000 símbolos. Rotulamos cada sequência utilizando os algoritmos *viterbi* e *posterior decoding* do HMM e do LCCRF agregador.

Cada rotulação foi comparada, e como esperado a rotulação foi idêntica utilizando cada um dos modelos. Além disso, vale ressaltar o desempenho obtido nesta última variação de LCCRF para o cassino desonesto, como podemos observar nas figuras 5.4 e 5.5. Observe que adicionamos o tempo de execução do LCCRF que utiliza apenas funções de características simples como o que é definido pelas funções da tabela 5.1 como referência.

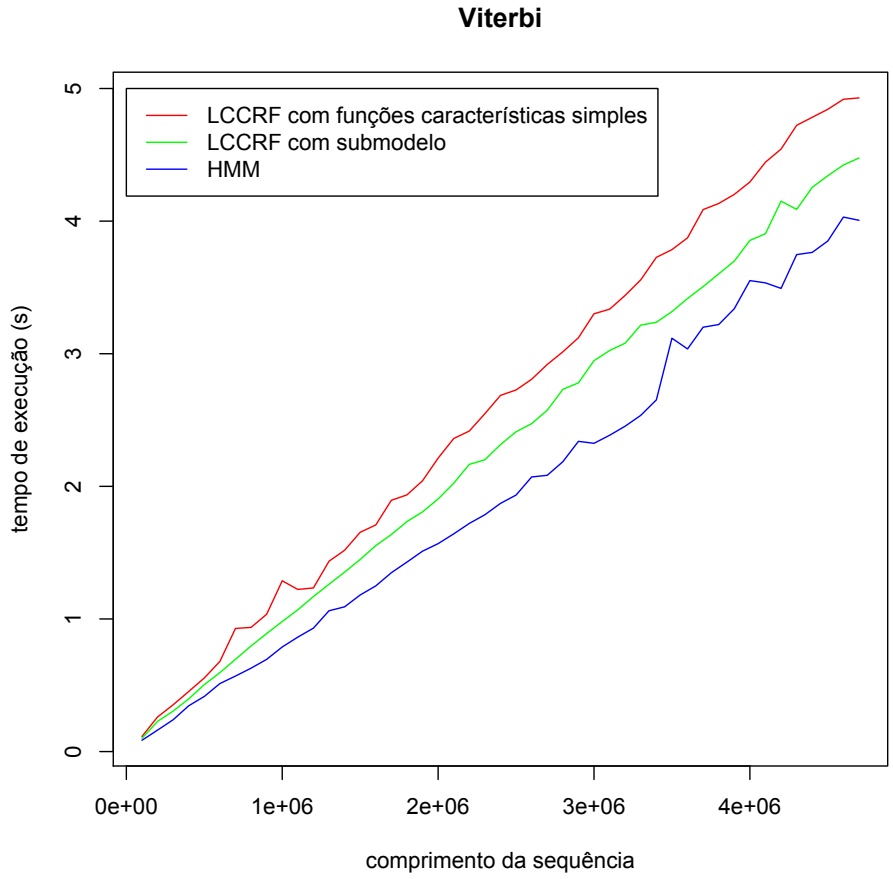

Figura 5.4: *Em azul, vermelho e verde, o tempo de execução do algoritmo* viterbi *em um HMM, LCCRF com funções de características simples e um LCCRF agregando dois* DiscreteIIDModel *respectivamente*

Com a diminuição do número de funções de características, e como o modelo *DiscreteIIDModel* é rapidamente avaliado pelo ToPS, esta modelagem de LCCRF que agrega submodelos mostrou-se bastante eficiente.

5.4 Treinamento

Sabemos que o treinamento de LCCRFs é bastante custoso. Investigamos o tempo necessário para o treinamento de LCCRFs comparando-o com o tempo necessário para treinar um HMM.

Utilizamos o HMM representado na figura 5.1 e o LCCRF de funções de características simples definidas na tabela 5.1.

Ambos os modelos foram treinados com os mesmos conjuntos de treinamento. Geramos aleatoriamente, utilizando o HMM já pré-definido da figura 5.1, 10 conjuntos de treinamento contendo cada um 10 sequências de lançamentos de dados. Cada conjunto têm sequências de mesmo comprimento variando de 10000 até 100000 lançamentos cada.

Os algoritmos de treinamento utilizados foram, LBFGS para o LCCRF e a implementação do algoritmo Baum-Welch disponível no ToPS para HMM. O desempenho com relação ao tempo de execução pode ser observado na figura 5.6.

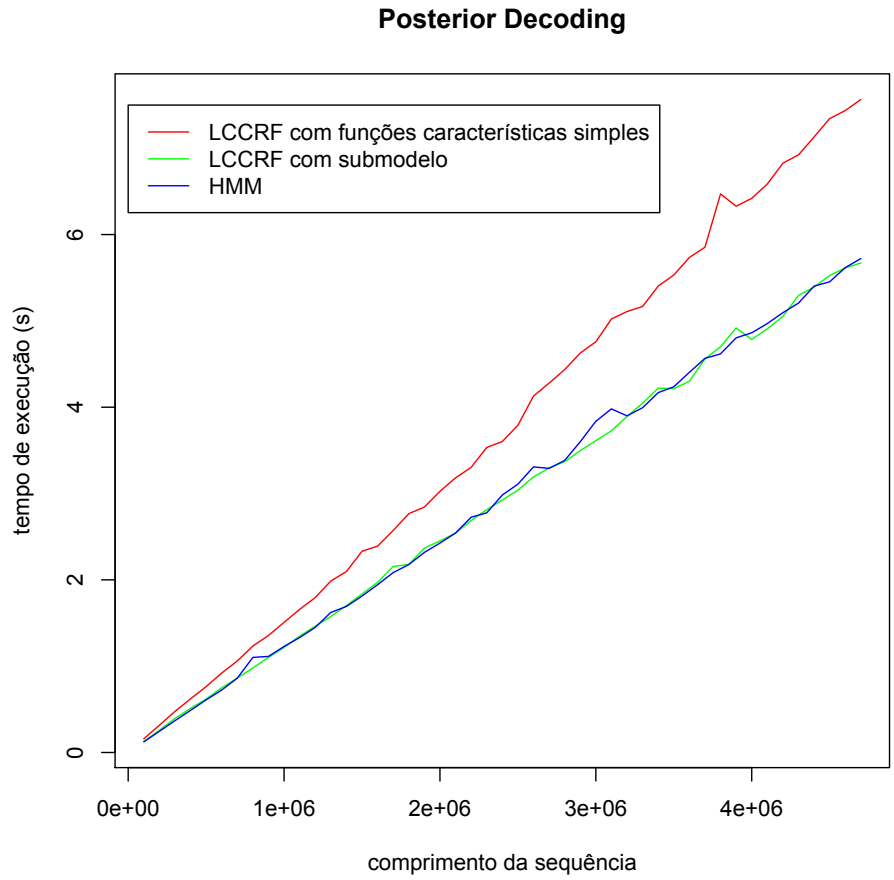

Figura 5.5: *Em azul, vermelho e verde, o tempo de execução do algoritmo* posterior decoding *em um HMM, LCCRF com funções de características simples e um LCCRF agregando dois* DiscreteIIDModel *respectivamente*

Note que mesmo com as otimizações nos algoritmos *forward* e *backward*, necessários durante o treinamento, o tempo de execução do treinamento de LCCRF é ainda bastante custoso. Além disso, podemos observar que, variando o tamanho do conjunto de treinamento, o crescimento do tempo de execução é mais acentuado em LCCRF. Esta é a justificativa de algumas aplicações que utilizam CRF em utilizar conjunto de dados reduzidos, como o preditor de gene CONRAD (DeCaprio *et al.* , 2007) que é aplicado à pequenos organismos, em contraste à preditores baseados em GHMM que são aplicados em genomas mais complexos.

Além dos testes de desempenho, validamos também se o treinamento de LCCRF conseguia generalizar o problema do cassino desonesto. Desta forma, utilizamos os mesmos dois modelos destacados no início desta seção para realizar algumas comparações.

Novamente utilizamos o HMM da figura 5.1 para gerar 10 sequências aleatórias de comprimento 100000 que foram utilizadas como conjunto de treinamento e teste.

Os dois modelos, HMM e LCCRF, foram treinados com uma das sequências e testado com o restante, totalizando 10 experimentos. Em cada experimento foram analisados os algoritmos *viterbi* e *posterior decoding*. Os resultados podem ser observados na tabela 5.3 e nas figuras 5.7 e 5.8, e como esperado, os dois modelos tiveram resultados semelhantes já que o LCCRF foi modelado para ser semelhante ao HMM do cassino desonesto.

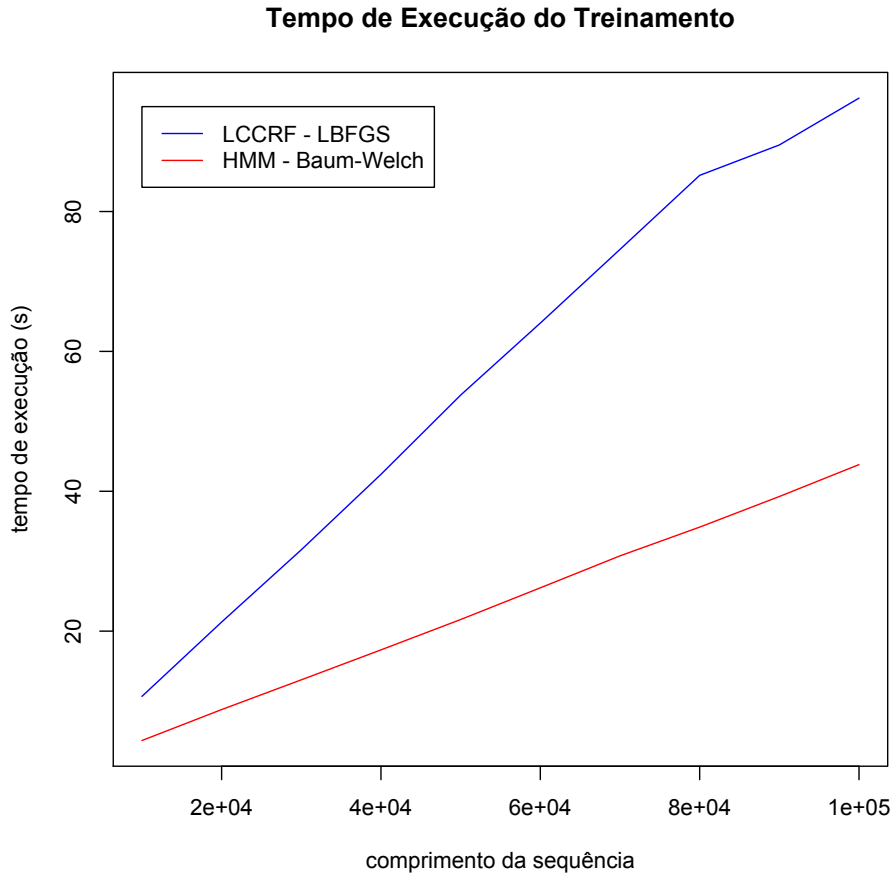

Figura 5.6: *Tempo de execução dos algoritmos LBFGS para LCCRF, em azul, e Baum-Welch para HMM, em vermelho*

modelo	*algoritmo*	*VP*	*FP*	*FN*	*VN*	*Sensibilidade*	*Especificidade*
HMM	Viterbi	3247194	1174660	1260294	3317852	72,04%	73,85%
LCCRF	Viterbi	3305419	1184319	1202069	3308193	73,33%	73,63%
HMM	Posterior Decoding	3366266	963267	1141222	3529245	74,68%	78,56%
LCCRF	Posterior Decoding	3389028	979434	1118460	3513078	75,18%	78,19%

Tabela 5.3: *Resultado dos algoritmos* viterbi *e* posterior decoding *para os HMM e LCCRF treinados. VP são os verdadeiros positivos, FP são os falsos positivos, FN são os falsos negativos e VN são os verdadeiros positivos.*

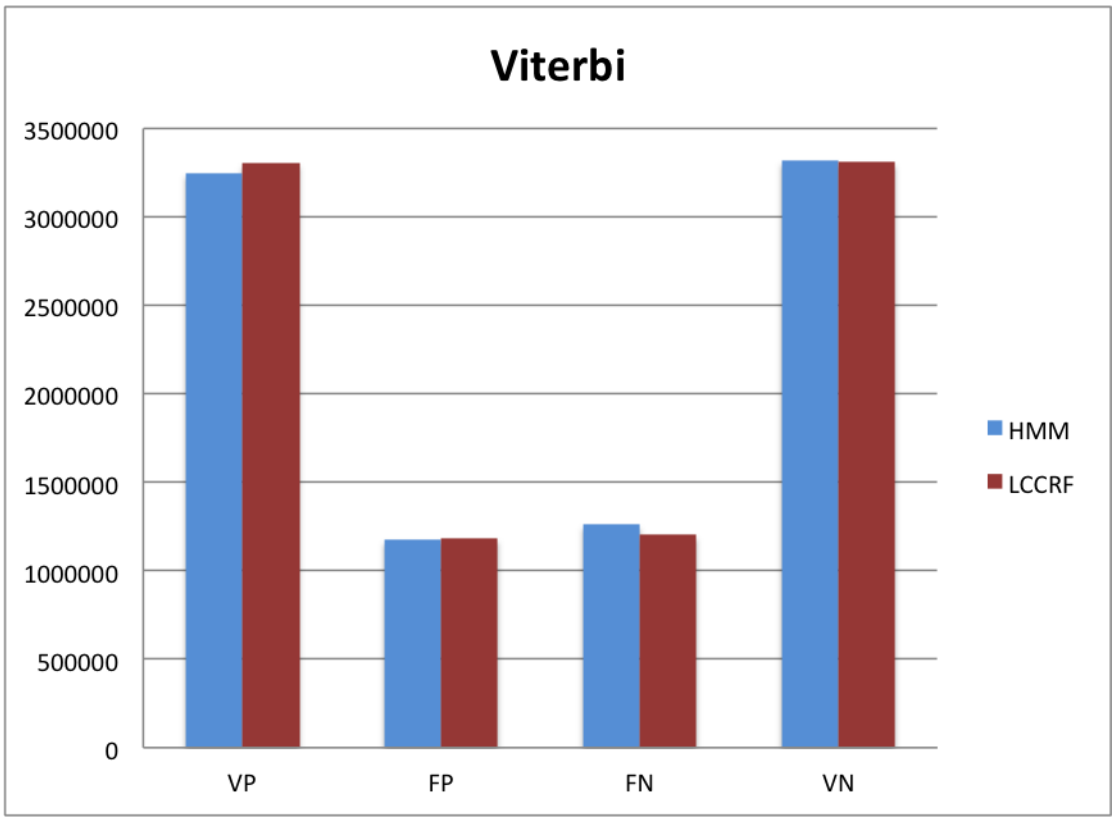

Figura 5.7: *Resultado do algoritmo* viterbi *para o HMM e LCCRF treinados. VP são os verdadeiros positivos, FP são os falsos positivos, FN são os falsos negativos e VN são os verdadeiros positivos.*

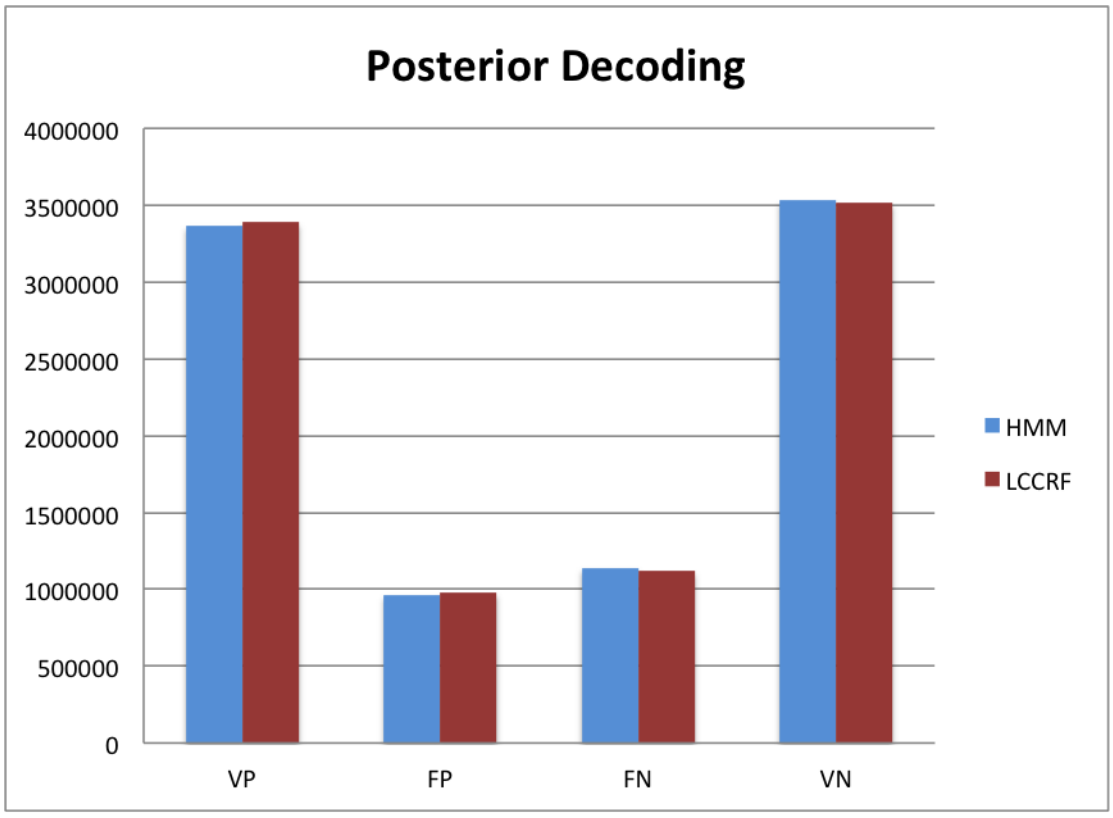

Figura 5.8: *Resultado do algoritmo* posterior decoding *para o HMM e LCCRF treinados. VP são os verdadeiros positivos, FP são os falsos positivos, FN são os falsos negativos e VN são os verdadeiros positivos.*

Capítulo 6

Comparações

Neste capítulo, apresentaremos comparações entre nossa implementação de LCCRF e dois arcabouços de uso genérico que também implementam este modelo: CRF++ (Kudo, 2005) e CRFSuite (Okazaki, 2007a).

6.1 CRF++

O arcabouço CRF++ traz uma implementação de LCCRF de código aberto cujo objetivo é segmentar dados sequenciais eficientemente. De fato, o tempo de convergência do treinamento de modelos utilizando CRF++ é reduzido, além de oferecer uma implementação do algoritmo de *viterbi* que executa em tempo competitivo. Outra característica importante é o fato deste arcabouço permitir a modelagem de problemas com mais de uma sequência de símbolos observados.

6.1.1 Modelagem

A modelagem de problemas é feita através da definição de *templates* de características. A partir destes *templates*, CRF++ extrai as funções de características necessárias a partir de um conjunto de treinamento e treina um novo modelo.

O conjunto de treinamento é definido como uma sequência de linhas, sendo que cada linha tem o mesmo número de colunas. Cada coluna, com exceção da última, contêm símbolos observáveis. Já a última coluna contêm rótulos atribuídos a posição em questão. Abaixo temos um exemplo desta estrutura (Kudo, 2005):

```
He          PRP  B-NP
reckons     VBZ  B-VP
the         DT   B-NP << Posição Atual
current     JJ   I-NP
account     NN   I-NP
```

A definição desses *templates* de características é composta por *macros* do tipo $x[a, b]$, sendo a o deslocamento da linha e b o deslocamento da coluna. Por exemplo, para a sequência anterior temos:

```
x[0,0]           = the
x[0,1]           = DT
x[-1,0]          = reckons
x[-2,1]          = PRP
x[0,0],x[0,1]    = the,DT
```

Existem dois tipos de *templates* de características:

* *Unigram*: Esse tipo de *template* segue o formato "U01:%x[a,b]", sendo que U indica seu tipo e *x[a,b]* é a macro utilizada. Esse *template* define automaticamente um conjunto de funções de características de tamanho $(L \cdot N)$, onde L é o número de rótulos e N é o número de combinações possíveis geradas pelas macros deste template. São comumente utilizadas para adicionar dependências entre observações e o rótulo da posição atual.

* *Bigram*: Esse tipo de *template* é representado pelo formato "B01:%x[0,1]", sendo que B indica seu tipo e *x[a,b]* é a macro utilizada. Esse *template* define $(L \cdot L \cdot N)$, onde L é o número de rótulos e N é o número de combinações possíveis geradas pelas macros deste template. É responsável por gerar funções de características têm dependência na transição do rótulo atual e anterior. Esse tipo de template tem um uso especial, sendo que pode-se defini-lo como "B", e assim gerar todas as $(N \cdot N)$ transições entre rótulos possíveis.

Essa abordagem reduz o número de funções de características que o usuário tem que definir, porém exige um maior conhecimento do funcionamento dos dois tipos de *templates* que o arcabouço disponibiliza.

Como exemplo, temos abaixo parte da definição do conjunto de treinamento do já mencionado problema do Cassino Desonesto, sendo que na primeira coluna temos as observações e na segunda coluna temos os rótulos.

```
6   Fair
5   Fair
2   Loaded
1   Loaded
1   Loaded
```

E a definição dos *templates* necessários para que o LCCRF seja similar à um HMM para o mesmo problema:

```
U00:%x[0,0]
B
```

Essa notação mostra-se bastante enxuta, já que, como podemos observar, utilizamos apenas dois *templates* para a definição de 16 funções de características.

6.1.2 Treinamento

Assim como o ToPS, o CRF++ utiliza o método LBFGS para treinar LCCRFs. O arcabouço fornece um programa chamado *crf_ learn* que recebe um arquivo de definição de *templates* e um conjunto de treinamento, e produz um arquivo binário que representa o modelo LCCRF já treinado, utilizando a linha de comando abaixo:

```
% crf_learn template_file train_file model_file
```

Nos testes que realizamos, o treinamento é otimizado e executa em tempo eficiente. Comparamos o tempo de execução com nossa implementação de LCCRF (representado na tabela 5.1) e de HMM (representado na figura 5.1), como pode ser observado na figura 6.1. Assim como fizemos no capítulo 5, utilizamos o problema do cassino desonesto e geramos, utilizando um HMM, 10 conjuntos de treinamento contendo cada um 10 sequências de lançamentos de dados. Cada conjunto têm sequências de mesmo comprimento variando de 10000 até 100000 lançamentos cada.

Vale ressaltar que o tempo de treinamento de um LCCRF no CRF++ não é constante, mas sim aumenta em uma taxa muito menor do que o tempo de treinamento de LCCRF e HMM no ToPS, e que devido a escala do gráfico essa pequena inclinação na reta é quase imperceptível. Os valores utilizado na construção do gráfico podem ser observados no apêndice B.

Outro ponto importante a se destacar é que o arcabouço CRF++ limita a definição de funções de características o que permite a ele utilizar estruturas de dados menos genéricas e mais eficientes do que as utilizadas no ToPS, o que justifica o melhor desempenho do algoritmo de treinamento.

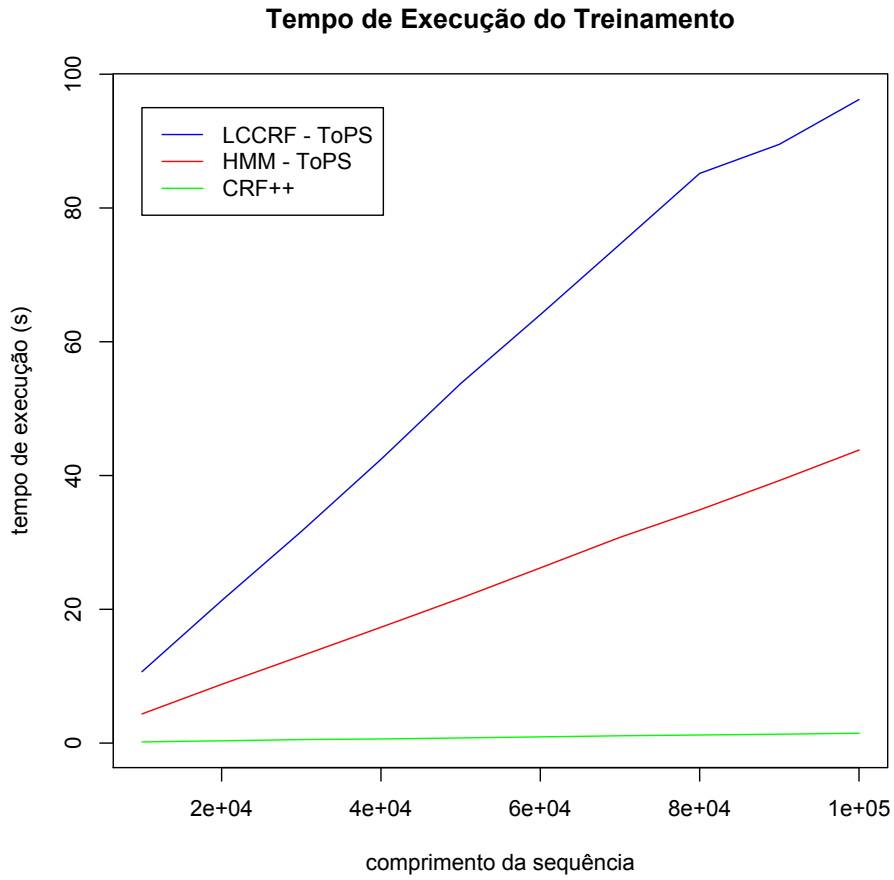

Figura 6.1: *Tempo de execução para treinar HMM no ToPS (vermelho), LCCRF no ToPS (azul) e CRF++ (verde)*

6.1.3 Inferência

CRF++ fornece apenas o algoritmo de *viterbi* para inferência. Este algoritmo pode ser utilizado através do programa *crf_test* que recebe como parâmetro um modelo previamente treinado e um arquivo de teste, que segue o mesmo formato do arquivo de treinamento. Após executado, o programa retorna o mesmo arquivo com uma nova coluna contendo os rótulos previstos.

```
% crf_test -m model_file test_files ...
```

O tempo de execução deste algoritmo é bastante competitivo, entretanto nossa implementação apresenta melhor desempenho, como pode ser notado na figura 6.2. A abordagem seguida foi a mesma do capítulo 5. Novamente o problema utilizado foi o do cassino desonesto, que foi modelado utilizando a implementação de HMM do ToPS para gerar 47 conjuntos. Cada conjunto possuía 10 sequências de mesmo tamanho variando de 100.000 a 4.700.000 símbolos. No CRF++ definimos um LCCRF semelhante ao definido pela tabela 5.1, e no ToPS utilizamos as duas variantes de LCCRF definidas pelas tabelas 5.1 e 5.2.

Vale destacar que mesmo o ToPS sendo uma implementação mais genérica de LCCRF, devido a paralelização dos algoritmos pudemos obter um resultado superior.

6.2 CRFSuite

CRFSuite é outra alternativa de código aberto que implementa o modelo LCCRF. Os algoritmos implementados são bastante eficientes, entretanto esta ferramenta exige que os dados sejam pre-

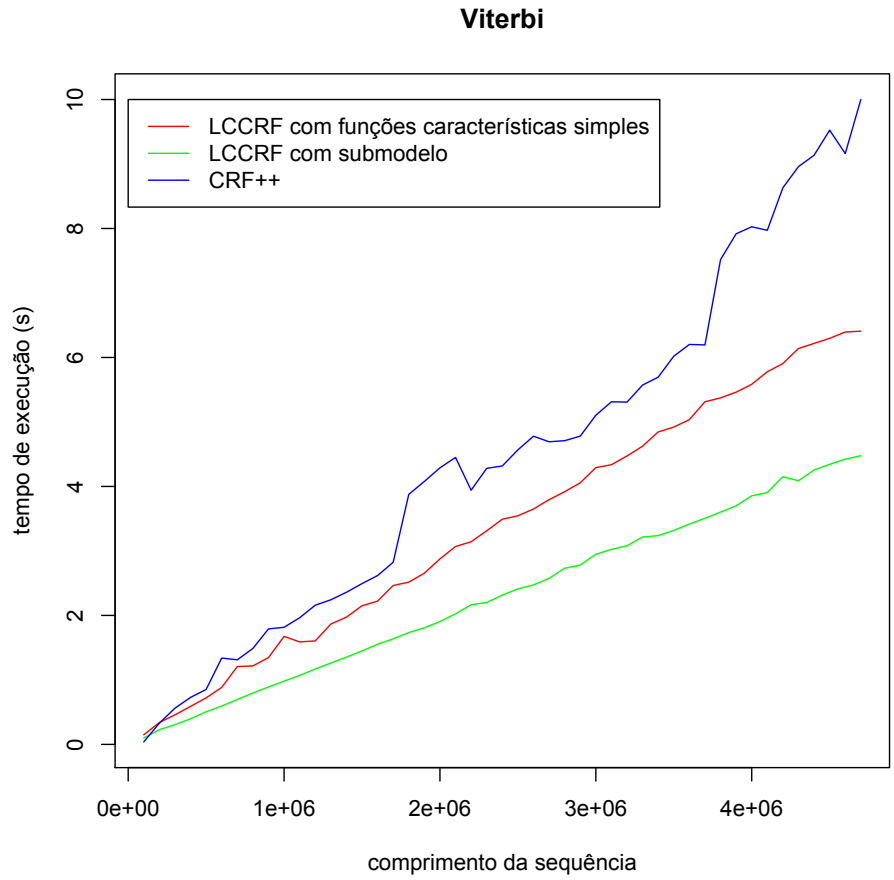

Figura 6.2: *Tempo de execução para segmentar sequências utilizando LCCRF com funções de características do tipo* $1_{\{a=b\}}$ *no ToPS (vermelho) e LCCRF com submodelos no ToPS (verde) e CRF++ (azul)*

processados antes do treinamento e rotulação. Esse preprocessamento mostrou-se bastante custoso o que pode ser uma problema quando os dados a serem analisados estão disponíveis em grande volume.

6.2.1 Modelagem

CRFSuite segue a mesma abordagem do CRF++ e o usuário deve definir uma série de *templates* que extrairão funções de características automaticamente. Mas diferente do CRF++, a definição desses *templates* é realizada através de um *script* escrito em linguagem *Python*. Essa abordagem dificulta a utilização da ferramenta por pessoas com nenhum ou pouco conhecimento em programação. Abaixo exibimos um *script* que extrai funções de características de uma série de lançamentos de dados de acordo com o problema do cassino desonesto:

```
1 #!/usr/bin/env python
2
3 separator = "\t"
4 fields = 'x y'
5
6 templates = (
7     (('x', 0), ),
8     )
9
10 import crfutils
11
12 def feature_extractor(X):
```

```
13      crfutils.apply_templates(X, templates)
14      if X:
15          X[0]['F'].append('__BOS__')      # BOS feature
16          X[-1]['F'].append('__EOS__')     # EOS feature
17
18  if __name__ == '__main__':
19      crfutils.main(feature_extractor, fields=fields, sep=separator)
```

O *script* acima gera automaticamente funções de características que definem as transições entre os rótulos, sendo que definimos explicitamente apenas as relações entre as observações x de deslocamento zero com o rótulo atual. É importante destacar que CRFSuite não consegue lidar com funções de características que condicionem observações com dois rótulos consecutivos.

6.2.2 Treinamento

CRFSuite tem disponível uma série de algoritmos de treinamento, dentre eles LBFGS que também é usado no ToPS. Os algoritmos executam em tempo bastante satisfatório. Entretanto, como já destacamos, o treinamento exige que os dados sejam preprocessados fazendo que o tempo de treinamento total seja elevado.

Modelamos dois LCCRFs idênticos para o problema do cassino desonesto, sendo um no ToPS e outro no CRFSuite. Utilizando um HMM modelado no ToPS, geramos 10 conjuntos de treinamento contendo cada um 10 sequências de lançamentos de dados. Cada conjunto têm sequências de mesmo comprimento variando de 10000 até 100000. O tempo necessário para treinar utilizando cada conjunto de treinamento pode ser observado na figura 6.3. Adicionamos o tempo de treinamento de um HMM modelado no ToPS apenas como referência.

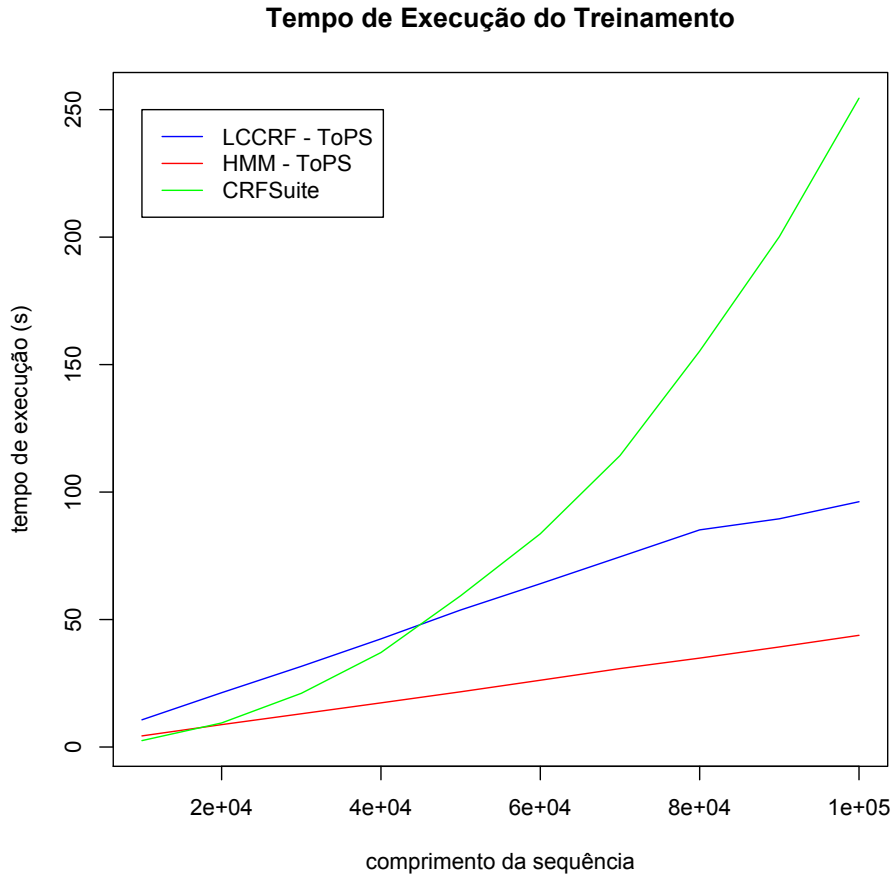

Figura 6.3: *Tempo de execução para treinar HMM no ToPS (vermelho), LCCRF no ToPS (azul) e LCCRF no CRFSuite (verde)*

6.2.3 Inferência

Assim como o CRF++, CRFSuite fornece apenas o algoritmo de *viterbi* para a segmentação de sequências. A segmentação também necessita que as sequências sejam preprocessadas, e desta forma a o tempo de execução desta operação também mostra-se lento.

A abordagem seguida foi a mesma do capítulo 5 e novamente o problema do cassino desonesto foi utilizado. Modelamos este problema utilizando a implementação de HMM do ToPS para gerar 47 conjuntos de sequências. Cada conjunto possuía 10 sequências de mesmo tamanho variando de 100.000 a 4.700.000 símbolos.

Modelamos dois LCCRFs idênticos, sendo um no ToPS e outro no CRFSuite e executamos o algoritmo de *viterbi* para cada conjunto de sequências. O resultado pode ser comparado na figura 6.4.

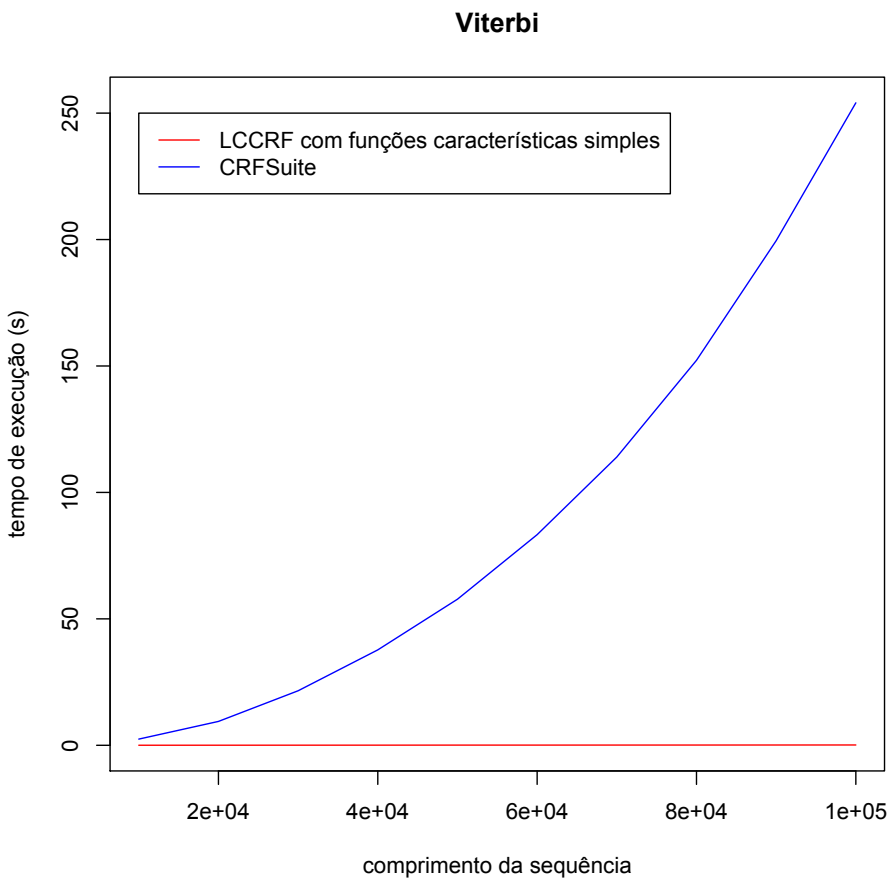

Figura 6.4: *Tempo de execução para segmentar sequências utilizando LCCRF no ToPS (vermelho) e no CRFSuite (azul). Vale ressaltar que a curva vermelha não é constante e sim linear, e que este fato é devido a escala utilizada para comportar o crescimento da curva azul.*

Observe que o tempo de execução da implementação do algoritmo de *viterbi* do ToPS para LCCRFs não é constante, mas sim cresce em uma taxa bastante inferior se comparado com o tempo de execução da implementação do arcabouço CRFSuite. Sendo assim, devido a escala do gráfico a inclinação da reta correspondente a implementação do ToPS é quase imperceptível. Os valores utilizado na construção do gráfico podem ser observados no apêndice B.

6.3 Conclusão

Ambos os programas apresentados neste capítulo seguem abordagens parecidas para a definição de funções de características. Essa abordagem tem a vantagem de se escrever um menor conjunto

de definições para se atingir o mesmo objetivo. Entretanto, nossa abordagem permite a definição de funções de características mais complexas além de serem escritas de forma mais próxima da definição formal de LCCRFs:

* funções de características que relacionam qualquer símbolo observável em qualquer deslocamento com o rótulo atual e/ou com o anterior.

* funções de características que utilizem submodelos probabilísticos.

Porém LCCRFs definidos no ToPS são limitados à uma única sequência de símbolos observáveis, o que não é uma limitação no CRF++ nem no CRFSuite.

Ainda sobre a modelagem, vale destacar que CRFSuite exige a edição de um programa escrito em linguagem *Python*, o que pode ser um complicador para usuários com nenhum ou pouco conhecimento em programação.

Outro ponto importante é que ambos fornecem apenas um único algoritmo para inferência. ToPS possui para segmentação os algoritmos *viterbi* e *posterior decoding* implementados, além de outros algoritmos acessados através da API (do inglês *application programming interface*) como forward, backward e o cálculo das probabilidades à posteriori.

Quanto ao desempenho, o CRF++ se destaca no treinamento que é mais rápido que o treinamento de HMM no ToPS, um modelo mais simples. Entretanto nossa implementação dos algoritmos de inferência mostra-se superior, aumentando em uma taxa inferior quando o comprimento das sequências aumentam.

E, por fim, as características dos 3 arcabouços podem ser resumidas na tabela 6.1.

	ToPS	*CRF++*	*CRFSuite*
Modelos	LCCRF, HMM, GHMM, IID, etc	LCCRF	LCCRF
Rotulação	*Viterbi* e *Posterior Decoding*	*Viterbi*	*Viterbi*
Treinamento	LBFGS	LBFGS	LBFGS
Sequências de Entrada	1	Múltiplas	Múltiplas
Tipo de Funções de Características	Simples, Composta, Submodelos	Simples	Simples
Definição de Modelos	Linguagem Própria	Template	*Script* em *Python*

Tabela 6.1: *Resumo das características do arcabouços ToPS, CRF++ e CRFSuite.*

Capítulo 7

Conclusão e Considerações Finais

Desenvolvemos um arcabouço probabilístico orientado à objetos para a modelagem de problemas utilizando Campos Aleatórios Condicionais de Cadeias Lineares. Essa implementação foi feita como uma extensão do arcabouço probabilístico ToPS. A maior vantagem de nossa abordagem é que, sendo o ToPS um arcabouço com alguns modelos geradores já implementados, podemos utilizar esses modelos na definição de funções de características, já que em nossa implementação essas função não se limitam à funções do tipo $1_{\{a=b\}}$ como é frequente em outros arcabouços.

Alem disso, esta extensão adiciona novas possibilidade de uso do arcabouço ToPS, que agora, além da abordagem geradora, temos também disponível um modelo probabilístico discriminativo bastante promissor. Podemos também fazer o uso de funções de características não probabilísticas, o que pode ajudar na melhoria de problemas bastante difíceis como a predição de genes.

Vale ressaltar que durante esta extensão foi mantida a modularidade do arcabouço, e que quase todos os aplicativos já existentes continuam compatíveis com a nova versão do arcabouço. Isso porque os modelos discriminativos não são capazes de gerar sequências a partir de uma simulação, assim o aplicativo *simulate* não pode ser utilizado por LCCRFs.

Melhoramos a definição da linguagem original e com isso todos os aplicativos ganharam uma apresentação melhor dos erros ocorridos durante a definição de modelos probabilísticos. Também estendemos essa linguagem para que o ToPS fosse capaz de receber definições de LCCRFs e arquivos de configuração para o treinamento deste modelo.

Implementamos os algoritmos de inferência *viterbi*, *forward*, *backward* e *posterior decoding*, bem como investigamos como melhorar o tempo de execução destes algoritmos. Como pudemos constatar, nossa implementação é bastante competitiva e eficiente comparando com modelos mais simples como HMM ou com outras implementação de LCCRFs como CRF++ (Kudo, 2005) e CRFSuite (Okazaki, 2007a).

Comparamos nossa abordagem com outras implementações de LCCRF e notamos que nossa implementação permite a definição de modelos mais elaborados. Pode-se definir funções de características complexas compondo funções de características mais simples. E além disso nossa implementação é a única que permite a adição de submodelos probabilísticos.

Como trabalho futuro, vale destacar a importância de se estudar formas de treinar LCCRFs mais eficientemente no arcabouço ToPS. Uma abordagem é investigar novos métodos de treinamento, como OWL-QN (do inglês *Orthant-Wise Limited-memory Quasi-Newton*) (Nocedal, 1980), SGD (do inglês, *Stochastic Gradient Descent*) (Shalev-Shwartz *et al.*, 2007), *Averaged Perceptron* (Collins, 2002) and AROW (do inglês *Adaptive Regularization Of Weight Vector*) (Mejer e Crammer, 2010).

Outro ponto importante para o futuro do arcabouço é a adição da possibilidade de permitir a entrada de mais de uma sequência de símbolos observáveis. Essa característica pode auxiliar na construção de modelos melhores, como a adição de informação extrínseca em predição de genes.

Por fim, é importante ressaltar que nosso grupo vem aplicando o ToPS com sucesso em algumas áreas da bioinformática como predição de genes (Kashiwabara, 2012) e alinhamentos múltiplos (Onuchic, 2012), áreas que devem obter um ganho significativo ao usar ou combinar as técnicas estudadas neste trabalho.

Apêndice A

Gramática

Apresentamos as gramáticas implementadas no ToPS utilizando a notação EBNF (do inglês *Extended Backus-Naur Form*)

A.1 Gramática Original

Antes da adição de LCCRFs ao arcabouço ToPS, refatoramos a gramática original para que esta pudesse ser mais facilmente estendida. Sua definição pode ser observada na listagem abaixo:

```
1  model                          : properties
2                                 ;
3
4  properties                     : property
5                                 | properties property
6                                 ;
7
8  property                       : IDENTIFIER '=' value
9                                 ;
10
11 value                          : STRING
12                                | INTEGER_NUMBER
13                                | FLOAT_POINT_NUMBER
14                                | list
15                                | probability_map
16                                | conditional_probability_map
17                                | sub_model
18                                | IDENTIFIER
19                                ;
20
21 list                           : '(' list_elements ')'
22                                ;
23
24 list_elements                  : list_element
25                                | list_elements ',' list_element
26                                ;
27
28 list_element                   : STRING
29                                | INTEGER_NUMBER
30                                | FLOAT_POINT_NUMBER
31                                ;
32
33 probability_map                : '(' probabilities_list ')'
34                                ;
35
36 probabilities_list             : probabilities
37                                | probabilities ';'
38                                ;
39
```

```
probabilities                  : probability
                               | probabilities ';' probability
                               ;

probability                    : TSTRING ':' list_element
                               ;

conditional_probability_map    : '(' conditional_probabilities_list ')'
                               ;

conditional_probabilities_list : conditional_probabilities
                               | conditional_probabilities ';'
                               ;

conditional_probabilities      : conditional_probability
                               | conditional_probabilities ';'
                                   conditional_probability
                               ;

conditional_probability        : condition ':' probability_number
                               ;

condition                      : STRING '|' STRING
                               ;

probability_number             : INTEGER_NUMBER
                               | FLOAT_POINT_NUMBER
                               ;

sub_model                      : '[' properties ']'
                               ;
```

Sendo que os *tokens* *IDENTIFIER*, *STRING*, *COMMENTS*, *FLOAT_POINT_NUMBER* e *INTEGER_NUMBER* são definidos pelas seguintes expressões regulares:

```
IDENTIFIER           : [a-zA-Z_][a-zA-Z0-9_]*
STRING               : L?\"(\\.|[^\\"])*\"
COMMENTS             : "#"[^\r\n]*
FLOAT_POINT_NUMBER   : \-?[0-9]+\.[0-9]+([Ee][+-]?[0-9]+)?
INTEGER_NUMBER       : \-?[0-9]+
```

A.2 Extensão

Dada a definição anterior, adicionamos algumas regras a linguagem e modificamos a regra *value*, como pode ser notada na listagem abaixo:

```
value                : STRING
                     | INTEGER_NUMBER
                     | FLOAT_POINT_NUMBER
                     | list
                     | probability_map
                     | conditional_probability_map
                     | sub_model
                     | IDENTIFIER
                     | feature
                     | factors
                     ;

feature              : '(' feature_functions ')'
                     ;
```

```
15
16 feature_functions       : feature_function
17                         ;
18
19 feature_function        : '{' STRING ':' feature_function_value '}'
20                         ;
21
22 feature_function_value : STRING
23                         | IDENTIFIER
24                         | feature
25                         ;
26
27 factors                 : '(' factor_list ')'
28                         ;
29
30 factor_list             : factor
31                         | factor_list ',' factor
32                         ;
33
34 factor                  : IDENTIFIER ':' probability_number
35                         ;
```

Apêndice B

Comparações

Apresentamos neste apêndice os valores dos resultados obtido e apresentados no capítulo 6.

B.1 Tempo de Execução para o Treinamento

Na tabela B.1 temos o tempo de execução dos algoritmos de treinamento nos arcabouços ToPS, CRF++ e CRFSuite. Estes valores foram utilizados na construção das figuras 6.1 e 6.3.

Tamanho da Sequência	*LCCRF - ToPS*	*HMM - ToPS*	*CRF++*	*CRFSuite*
10000	10.669	4.360	0.173	2.511
20000	21.299	8.784	0.339	9.442
30000	31.632	13.038	0.522	21.060
40000	42.419	17.319	0.615	37.061
50000	53.749	21.661	0.768	59.332
60000	64.049	26.193	0.929	83.642
70000	74.585	30.763	1.090	114.306
80000	85.171	34.874	1.209	155.294
90000	89.517	39.249	1.327	200.174
10000	96.216	43.813	1.475	254.488

Tabela B.1: *Tempo necessário para treinar HMM no ToPS e LCCRF no ToPs, CRF++ e CRFSuite.*

B.2 Tempo de Execução para a Inferência

B.2.1 CRFSuite

Como CRFSuite necessita de muito tempo para realizar inferências, as sequências utilizadas nos testes foram de menor comprimento. E vale ressaltar que este arcabouço implementa apenas o algoritmo de *viterbi*.

A comparação entre o tempo de execução do algoritmo de *viterbi* implementado no ToPS e CRFSuite pode ser observada na tabela B.2. Esses valores foram utilizados na construção da figura 6.4.

B.2.2 CRF++

Assim como CRFSuite, CRF++ implementa apenas o algoritmo de *viterbi*. Na tabela B.3 temos os valores obtidos nas comparações entre as implementações deste algoritmo nos arcabouços ToPS e CRF++. Esses valores foram utilizados na construção da figura 6.2.

Tamanho da Sequência	*ToPS*	*CRFSuite*
10000	0.021	2.447
20000	0.032	9.441
30000	0.044	21.561
40000	0.055	37.732
50000	0.074	57.773
60000	0.077	83.298
70000	0.090	113.999
80000	0.099	152.245
90000	0.114	199.546
10000	0.127	254.066

Tabela B.2: *Tempo necessário para executar o algoritmo de viterbi em LCCRF no ToPs e CRFSuite.*

Tamanho da Sequência	*ToPS - tabela 5.1*	*ToPS - tabela 5.2*	*CRF++*
100k	0.1478	0.1584	0.0382
200k	0.3387	0.3156	0.3288
300k	0.4583	0.4760	0.5615
400k	0.5890	0.6213	0.7288
500k	0.7213	0.7629	0.8520
600k	0.8838	0.9203	1.3385
700k	1.2071	1.0603	1.3118
800k	1.2172	1.2334	1.4878
900k	1.3460	1.3560	1.7907
1.0M	1.6740	1.5070	1.8158
1.1M	1.5895	1.6583	1.9637
1.2M	1.6041	1.7949	2.1580
1.3M	1.8660	1.9865	2.2423
1.4M	1.9732	2.0965	2.3588
1.5M	2.1497	2.3309	2.4931
1.6M	2.2214	2.3908	2.6185
1.7M	2.4634	2.5715	2.8244
1.8M	2.5164	2.7664	3.8772
1.9M	2.6547	2.8433	4.0774
2.0M	2.8767	3.0254	4.2891
2.1M	3.0686	3.1817	4.4495
2.2M	3.1424	3.3026	3.9415
2.3M	3.3123	3.5336	4.2810
2.4M	3.4915	3.6025	4.3173
2.5M	3.5448	3.7904	4.5672
2.6M	3.6497	4.1293	4.7789
2.7M	3.7948	4.2789	4.6926
2.8M	3.9184	4.4351	4.7095
2.9M	4.0566	4.6272	4.7804
3.0M	4.2917	4.7583	5.1040
3.1M	4.3367	5.0219	5.3128
3.2M	4.4720	5.1085	5.3076
3.3M	4.6244	5.1670	5.5723
3.4M	4.8451	5.4030	5.6947
3.5M	4.9204	5.5310	6.0198
3.6M	5.0360	5.7344	6.2016
3.7M	5.3131	5.8533	6.1945
3.8M	5.3729	6.4709	7.5172
3.9M	5.4620	6.3286	7.9181
4.0M	5.5835	6.4195	8.0264
4.1M	5.7781	6.5858	7.9718
4.2M	5.9062	6.8264	8.6319
4.3M	6.1393	6.9232	8.9597
4.4M	6.2178	7.1306	9.1357
4.5M	6.2964	7.3438	9.5242
4.6M	6.3944	7.4364	9.1622
4.7M	6.4080	7.5644	9.9989

Tabela B.3: *Tempo necessário para executar o algoritmo de viterbi em LCCRF no ToPs e CRF++.*

Referências Bibliográficas

Axelson-Fisk (2010) Marina Axelson-Fisk. *Comparative Gene Finding: Models, Algorithms and Implementation*. Springer, 1º ed. Citado na pág. 8

Burge (1997) C. Burge. *Identification of genes in human genomic DNA*. Tese de Doutorado, Stanford University. Citado na pág. 1, 3, 8

Collins (2002) Michael Collins. Discriminative training methods for hidden markov models: theory and experiments with perceptron algorithms. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, páginas 1–8. Citado na pág. 45

DeCaprio *et al.* **(2007)** David DeCaprio, Jade Vinson, Matthew Pearson, Philip Montgomery, Matthew Doherty e James Galagan. Conrad: Gene prediction using conditional random fields. *Genome Research*, 17(9):1389–1398. Citado na pág. 1, 21, 34

Dong e David (2007) Ming Dong e He David. A segmental hidden semi-markov model (hsmm)-based diagnostics and prognostics framework and methodology. *Mechanical systems and signal processing*, 21(5):2248–2266. Citado na pág. 8

Duda *et al.* **(2000)** Richard O. Duda, Peter E. Hart e David G. Stork. *Pattern Classification*. Wiley-Interscience, 2º ed. Citado na pág. 1

Dunham *et al.* **(1987)** M. Dunham, O. Kimball, M. Krasner, G. Kubala, J. Makhoul, P. Price, S. Roucos e R.Schwartz. Byblos: The bbn continuous speech recognition system. *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '87*, páginas 89–92. Citado na pág. 7

Durbin *et al.* **(1998)** Richard Durbin, Sean R. Eddy, Anders Krogh e Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1º ed. Citado na pág. xi, 1, 6, 7

Eddy (1998) S. R. Eddy. Profile hidden markov models. *Bioinformatics*, 14:755–763. Citado na pág. 1, 3, 7

Feng *et al.* **(2006)** Shaolei L. Feng, R. Manmatha e Andrew McCallum. Exploring the use of conditional random field models and hmms for historical handwritten document recognition. *IEEE International Conference on Document Image Analysis for Libraries (DIAL 06)*, páginas 30–37. Citado na pág. 16

Finkel *et al.* **(2005)** Jenny Rose Finkel, Trond Grenager e Christopher Manning. Incorporating non-local information into information extraction systems by gibbs sampling. *Proceedings of the 43nd Annual Meeting of the Association for Computational Linguistics*, páginas 363–370. Citado na pág. 1, 2

Fraser (2009) Andrew M. Fraser. *Hidden Markov Models and Dynamical Systems*. Society for Industrial Mathematics, 1º ed. Citado na pág. 6

Gamma E. *et al.* **(1994)** Helm R. Gamma E., Johnson R. e Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. Citado na pág. 17, 20

Grimmett e Stirzaker (2001) Geoffrey R. Grimmett e David R. Stirzaker. *Probability and Random Processes*. Oxford University Press, USA, 3º ed. Citado na pág. 3

Hughey e Krogh (1996) Richard Hughey e Anders Krogh. Hidden markov models for sequence analysis: extension and analysis of the basic method. *Comput Appl Biosci*, 12:95–107. Citado na pág. 3

Kashiwabara (2012) A. Y. Kashiwabara. *MYOP/ToPS/SGEval: Um ambiente computacional para estudo sistemático de predição de genes*. Tese de Doutorado, Universidade de São Paulo. Citado na pág. xi, 1, 2, 17, 18, 19, 45

Korf *et al.* **(2001)** I. Korf, P. Flicek e M.R. Brent. ntegrating genomic homology into gene structure prediction. *Bioinformatics*, 17:140–148. Citado na pág. 1

Krishnan (2006) Venkatarama Krishnan. *Probability and Random Processes*. Wiley-Interscience, 1º ed. Citado na pág. 3

Krogh (1997) A. Krogh. Two methods for improving performance of a hmm and their application for gene finding. *Proceedings of the Fifth International Conference on Intelligent Systems for Molecular Biology*, páginas 179–186. Citado na pág. 7

Kudo (2005) Taku Kudo. Crf++: Yet another crf toolkit, 2005. URL http://crfpp.googlecode.com/svn/trunk/doc/index.html. Citado na pág. 2, 37, 45

Kulp *et al.* **(1996)** D. Kulp, D. Haussler, M.G. Reese e F.H. Eeckman. A generalized hidden markov model for the recognition of human genes in dna. *International Conference on Intelligent Systems for Molecular Biology*, 4:134–42. Citado na pág. 1, 8

Lafferty *et al.* **(2001)** J. Lafferty, A. McCallum e F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. Em *Proc. 18th International Conf. on Machine Learning*. Citado na pág. 1, 2, 9, 15

Leaman e G. (2008) R. Leaman e Gonzalez G. Banner: An executable survey of advances in biomedical named entity recognition. *Pacific Symposium on Biocomputing*, páginas 652–663. Citado na pág. 2

Loof *et al.* **(2007)** J. Loof, C. Gollan, S. Hahn, G. Heigold, B. Hoffmeister, C. Plahl, D. Rybach, R. Schluter e H. Ney. The rwth 2007 tc-star evaluation system for european english and spanish. *In Interspeech*, páginas 2145–2148. Citado na pág. 7

Lukashin e Borodovsky (1998) A. Lukashin e M. Borodovsky. Genemark.hmm: new solutions for gene finding. *Nucleic Acids Research*, 26:1107–1115. Citado na pág. 7

Majoros (2007) W.H. Majoros. Availability. methods for computational gene prediction. *Cambridge University Press*. Citado na pág. 8

Majoros *et al.* **(2004)** W.H. Majoros, M. Pertea e S.L. Salzberg. Tgrscan and glimmerhmm: tow open-source ab initio eukaryotic gene-finders. *Bioinformatics*, 20:2878–2879. Citado na pág. 1, 8

Maltoni *et al.* **(2009)** Davide Maltoni, Dario Maio, Anil K. Jain e Salil Prabhakar. *Handbook of Fingerprint Recognition*. Springer, 2º ed. Citado na pág. 1

Mejer e Crammer (2010) Avihai Mejer e Koby Crammer. Confidence in structured-prediction using confidence-weighted models. *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, páginas 971–981. Citado na pág. 45

Munch e Krogh (2006) K. Munch e A. Krogh. Automatic generation of gene finders for eukaryotic species. *BMC Bioinformatics*. Citado na pág. 7

Nabende *et al.* (2008) Peter Nabende, Jorg Tiedemann e John Nerbonne. Pair hidden markov model for named entity matching. *In Proceedings of the fourth International Joint Conference on Computer, Information, and System Sciences, and Engineering*. Citado na pág. 3

Nocedal (1980) Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35:773–782. Citado na pág. 45

Okazaki (2007a) Naoaki Okazaki. Crfsuite: a fast implementation of conditional random fields (crfs), 2007a. URL http://www.chokkan.org/software/crfsuite/. Citado na pág. 2, 37, 45

Okazaki (2007b) Naoaki Okazaki. liblbfgs: a library of limited-memory broyden-fletcher-goldfarb-shanno (l-bfgs), 2007b. URL http://www.chokkan.org/software/liblbfgs/. Citado na pág. 25

Onuchic (2012) Vitor Onuchic. Inovações em técnicas de alinhamentos múltiplos e predições de genes., 2012. Citado na pág. 45

Peng *et al.* (2004) Fuchun Peng, Fangfang Feng e Andrew McCallum. Chinese segmentation and new word detection. *Proceedings of The 20th International Conference on Computational Linguistics (COLING 2004)*. Citado na pág. 16

Rabiner e Juang (1993) Lawrence Rabiner e Biing-Hwang Juang. *Fundamentals of Speech Recognition*. Prentice Hall, 1º ed. Citado na pág. 1

Rabiner (1989) Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, páginas 257–286. Citado na pág. 3, 4, 5, 7

Rybach *et al.* (2009) D. Rybach, C. Gollan, G. Heigold, B. Hoffmeister, J. Loof, R. Schluter e H. Ney. The rwth aachen university open source speech recognition system. *In Interspeech*, páginas 2111–2114. Citado na pág. 7

Sato e Sakakibara (2005) Kengo Sato e Yasubumi Sakakibara. Rna secondary structural alignment with conditional random fields. *Bioinformatics*, 21:ii237–ii242. Citado na pág. 15

Shalev-Shwartz *et al.* (2007) Shai Shalev-Shwartz, Yoram Singer e Nathan Srebro. Pegasos: Primal estimated sub-gradient solver for svm. *Proceedings of the 24th International Conference on Machine Learning*, páginas 807–814. Citado na pág. 45

Shen *et al.* (2007) Dou Shen, Jian-Tao Sun, Hua Li, Qiang Yang e Zheng Chen. Document summarization using conditional random fields. Em *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, páginas 2862–2867, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. URL http://dl.acm.org/citation.cfm?id=1625275.1625736. Citado na pág. 1

Stanke e Waack (2003) M. Stanke e S. Waack. Gene prediction with a hidden markov model and a new intron submodel. *Bioinformatics*, 19(suppl 2):ii215–ii225. Citado na pág. 8

Stanke *et al.* (2006) Mario Stanke, Oliver Schoffmann, Burkhard Morgenstern e Stephan Waack. Gene prediction in eukaryotes with a generalized hidden markov model that uses hints from external sources. *Bioinformatics*, 7(1):62. Citado na pág. 8

Sutton e McCallum (2006) Charles Sutton e Andrew McCallum. An introduction to conditional random fields for relational learning. Em Lise Getoor e Ben Taskar, editors, *Introduction to Statistical Relational Learning*, páginas 93–122. MIT Press, 1º ed. Citado na pág. 2, 9, 11, 13, 15, 25, 27

Wallach (2004) Hanna M. Wallach. Conditional random fileds: An introduction. Relatório Técnico MS-CIS-04-21, University of Pennsylvania. Citado na pág. 24

Wang e Xu (2011) Zhiyong Wang e Jinbo Xu. A conditional random fields method for rna sequence-structure relationship modeling and conformation sampling. *Bioinformatics*, 27:i102–i110. Citado na pág. 15

Zen *et al.* **(2004)** Heiga Zen, Keiichi Tokuda, Takashi Masuko, Takao Kobayashi e Tadashi Kitamura. Hidden semi-markov model based speech synthesis. Em *in Proc. of ICSLP, 2004*. Citado na pág. 8