Universidade Federal da Paraíba

Centro de Informática

Programa de Pós-Graduação em Informática

# Classificação Supervisionada com Programação Probabilística

Danilo Carlos Gouveia de Lucena

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal da Paraíba como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Computação Distribuída

Andrei de Araújo Formiga

(Orientador)

João Pessoa, Paraíba, Brasil

L935c Lucena, Danilo Carlos Gouveia de.
Classificação supervisionada com programação probabilística / Danilo Carlos Gouveia de Lucena.-- João Pessoa, 2014.
98f. : il.
Orientador: Andrei de Araújo Formiga
Dissertação (Mestrado) - UFPB/CI
1. Informática. 2. Computação distribuída. 3. Mecanismos de inferência. 4. Linguagens probabilísticas. 5. Classificado de textos.

UFPB/BC CDU: 004(043)

Ata da Sessão Pública de Defesa de Dissertação de Mestrado de **Danilo Carlos Gouveia de Lucena**, candidato ao Título de Mestre em Informática na Área de Sistemas de Computação, realizada em 10 de Fevereiro de 2014.

Ao décimo dia do mês de fevereiro do ano dois mil e quatorze, às quatorze horas, no auditório do CCEN - Universidade Federal da Paraíba, reuniram-se os membros da Banca Examinadora constituída para examinar o candidato ao grau de Mestre em Informática, na área de "*Sistemas de Computação*", na linha de pesquisa "*Computação Distribuída*", o Sr. **Danilo Carlos Gouveia de Lucena**. A comissão examinadora foi composta pelos professores doutores: ANDREI DE ARAÚJO FORMIGA (PPGI-UFPB), Orientador e Presidente da Banca, LUCÍDIO DOS ANJOS FORMIGA CABRAL (PPGI-UFPB), examinador interno, ROBERTO QUIRINO DO NASCIMENTO (PPGI-UFPB), examinador interno e RICARDO BASTOS CAVALCANTE PRUDENCIO (UFPE) como examinador externo. Dando início aos trabalhos, o professor ANDREI DE ARAÚJO FORMIGA cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou a palavra ao candidato para que o mesmo fizesse, oralmente, a exposição do trabalho de dissertação intitulado "*Classificação Supervisionada com Programação Probabilística*". Concluída a exposição, o candidato foi arguido pela Banca Examinadora que emitiu o seguinte parecer: "A*provado*". Assim sendo, deve a Universidade Federal da Paraíba expedir o respectivo diploma de Mestre em Informática na forma da lei e, para constar, eu, Alisson Vasconcelos de Brito, Coordenador deste Programa, servindo de secretário, lavrei a presente ata que vai assinada por mim e pelos membros da Banca Examinadora. João Pessoa, 10 de Fevereiro de 2014.

Alisson Vasconcelos de Brito

Prof. Dr. Andrei de Araújo Formiga _____
Orientador (PPGI-UFPB)

Prof. Dr. Lucídio dos Anjos Formiga Cabral _____
Examinador Interno (PPGI-UFPB)

Prof. Dr. Roberto Quirino do Nascimento _____
Examinador Interno (PPGI-UFPB)

Prof. Dr. Ricardo Bastos Cavalcante Prudencio _____
Examinador Externo (UFPE)

# Resumo

Mecanismos de inferência probabilísticos estão na intersecção de três áreas: estatística, linguagens de programação e sistemas de probabilidade. Esses mecanismos são utilizados para criar modelos probabilísticos e auxiliam no tratamento de incertezas. As linguagens de programação probabilísticas auxiliam na descrição de alto nível desses tipos de modelos. Essas linguagens facilitam o desenvolvimento abstraindo os mecanismos de inferência de mais baixo nível, favorecem o reuso de código e auxiliam na análise dos resultados. Este estudo propõe a análise dos mecanismos de inferência implementados pelas linguagens de programação probabilísticas e apresenta um estudo de caso com a implementação de um classificador supervisionado de textos com programação probabilística.

**Palavras-chave:** mecanismos de inferência, linguagens probabilísticas, classificador de textos.

# Abstract

Probabilistic inference mechanisms are at the intersection of three main areas: statistics, programming languages and probability. These mechanisms are used to create probabilistic models and assist in treating uncertainties. Probabilistic programming languages assist in high-level description of these models. These languages facilitate the development of the models because they abstract the inference mechanisms at the lower levels, allow reuse of code, and assist in results analysis. This study proposes the analysis of inference engines implemented by probabilistic programming languages and presents a case study of a supervised text classifier using probabilistic programming.



**Keywords:** inference mechanisms, probabilistic languages, text classification.

# Agradecimentos

Agradeço ao apoio do Departamento de Pós-graduação em Informática da Universidade Federal da Paraíba que recebi enquanto aluno do curso de mestrado. Dedico a dissertação à família, aos amigos e professores que me auxiliaram nesses dois anos.

# Conteúdo

# Lista de Símbolos

**JAGS** : *Just another Gibbs sampler*

**NLTK** : *Natural Language Toolkit*

**BUGS** : *Bayesian inference Using Gibbs Sampling*

**DAG** : *Direct Acyclic Graph*

**NLP** : *Natural Language Processing*

**KR** : *Knowledge Representation*

**MCMC** : *Markov Chain Monte Carlo*

**OMC** : *Ordinary Markov Chain*

**CSV** : *Comma Separated Values*

**GPL** : *General Public License*

**BSD** : *Berkeley Software Distribution*

# Lista de Figuras

# Lista de Tabelas

# Lista de Códigos Fonte

# Capítulo 1

# Introdução

## 1.1 Motivação

Alguns dos avanços na área de inteligência artificial, ao longo do histórico de pesquisas na área da ciência da computação, estão relacionados com o estudo e a criação de novas técnicas para aprendizado de máquina. Um dos objetivos da área de aprendizado de máquina é o desenvolvimento de técnicas que permitam desenvolver sistemas capazes de realizar inferência, como sistemas de predição e análise de padrões, em conjuntos de dados [Bishop et al. 2006; Sebastiani 2002; Witten e Frank 2005; Roy].

Uma outra área, a da estatística computacional, estuda técnicas necessárias para a criação de modelos probabilísticos. Esses modelos são utilizados para representar distribuições de probabilidades e também auxiliam na resolução de problemas que precisam lidar com incertezas relacionadas às situações modeladas. Uma das tarefas dos mecanismos de inferência probabilísticos é o de auxiliar a implementação e execução de sistemas que são baseados nas especificações de modelos probabilísticos.

Modelos podem ser descritos com o uso de linguagens de programação desde que especifiquem todos os elementos necessários para a composição em alto nível do modelo probabilístico desejado. Para essa tarefa, linguagens de programação *determinísticas* como *Java* e *Python* podem ser utilizadas para a implementação desse tipo de modelo mas nesses casos é necessário implementar todos os mecanismos de inferência probabilística. Além disso, embora algumas técnicas possuam grande semelhança quanto à sua modelagem matemática, os mecanismos de implementação de cada uma delas normalmente compartilha muito pouco, o

que ocasiona que cada nova técnica que deve ser implementada demanda um grande esforço de desenvolvimento.

Para esta dissertação, uma nova categoria de linguagens de programação é analisada: as linguagens de programação *probabilísticas*. Essas categoria de linguagem facilita a especificação e a implementação de modelos probabilísticos, permitindo a descrição em alto nível dos modelos e disponibilizando mecanismos genéricos de inferência. Exemplos de linguagens de programação desse tipo são o *Church* [Church MIT 2013], *Infer.net* [Minka et al.], *BUSG/WinBUGS* [WinBUGS 2013] e *JAGS* [JAGS 2013].

Esta dissertação tem duas propostas principais. A primeira é realizar um estudo sobre as linguagens de programação probabilísticas e seus mecanismos de inferência. Os mecanismos de inferência compreendem as técnicas necessárias para obter informações dos modelos analisados. A segunda proposta é fazer um estudo sobre a implementação de um classificador supervisionado de texto com programação probabilística comparada com implementações utilizando programação determinística. Como base para as implementações é utilizada a técnica de classificação com *Naive Bayes*, estudada na literatura relacionada ao *processamento de linguagem natural*, (*natural language processing*, *NLP*).

O objetivo da área de processamento de linguagem natural é o desenvolvimento de sistemas computacionais que utilizam a linguagem natural tanto em sua forma escrita quanto falada. A classificação supervisionada de textos [Segaran 2007; Barber 2012] é um dos problemas dessa área e é referente ao tópico de aprendizado supervisionado.

Demonstra-se que é possível utilizar linguagens probabilísticas para tarefas de classificação supervisionada. Os resultados comparativos com as outras implementações são analisados em conjunto com as vantagens e desvantagens de cada tipo de implementação.

## 1.2 Objetivos

O estudo proposto possui dois tipos de objetivos, um geral e um conjunto de objetivos específicos, que são apresentados nos capítulos que se seguem.

### 1.2.1 Geral

- Analisar métodos de inferência probabilística e avaliar o uso desses métodos para resolução do problema da classificação supervisionada.

### 1.2.2 Específicos

- Analisar implementações de linguagens probabilísticas modernas.
- Destacar as principais limitações encontradas nas linguagens probabilísticas analisadas.
- Analisar soluções para as limitações encontradas nas linguagens estudadas.
- Demonstrar a aplicação de linguagens probabilísticas para implementação de classificadores supervisionados de texto.

## 1.3 Justificativa

As linguagens de programação *probabilísticas* têm como objetivo auxiliar na implementação de modelos probabilísticos a partir de descrições de alto nível, automatizando os processos de inferência necessários para utilizar tais modelos em situações reais. Essa categoria de linguagem facilita a especificação e o desenvolvimento em alto nível de sistemas que usam modelos probabilísticos como base. As linguagens probabilísticas seguem muitos dos princípios de facilidade de desenvolvimento de código encontrados nas linguagens de programação determinísticas tradicionais, tais como modularidade, reuso de código, *debug* e testes; porém com foco nas técnicas probabilísticas.

Um dos objetivos das pesquisas iniciais na área da *inteligência artificial* [Russell et al. 1995; Crevier 1993] era o estudo para o desenvolvimento de mecanismos para tentar simular o pensamento humano. Os métodos de inferência para esse propósito eram desenvolvidos com o uso de sistemas derivados da matemática e da lógica, e permitiam lidar com certos problemas propostos ainda de forma limitada. Metodologias recentes, junto com a melhoria da capacidade computacional, permitem criar modelos de inferência mais precisos e que conseguem resolver um conjunto maior de problemas. Também é observado que sistemas

que utilizam modelos em mecanismos probabilísticos podem modelar uma quantidade maior de situações do que os modelos tradicionais determinísticos [Goodman 2013].

Muitas das contribuições das técnicas para construção de modelos probabilísticos em ambientes computacionais são derivadas do estudo das *redes Bayesianas*, fundamentadas na regra de *Bayes* [Joyce 2008]. Unidas com as características já mencionadas das linguagens probabilísticas, as redes *Bayesianas* oferecem uma metodologia favorável para o desenvolvimento de sistemas de inferência em modelos que usam conjunto de dados. A partir da especificação de um modelo probabilístico pode-se construir uma rede *Bayesiana* como uma representação estruturada equivalente. Dois componentes fundamentais em uma rede *Bayesiana* são os componentes quantitativos e os qualitativos [Darwiche 2009]. A parte quantitativa corresponde a um grafo direto acíclico e representa a estrutura da rede *Bayesiana*. Para tarefas de inferência nessa estrutura considera-se a relação entre as *variáveis* e as suas *conexões*. As linguagens de programação probabilísticas apresentadas fazem uso, em maior ou menor grau, dos conceitos das redes *Bayesianas*. Demonstra-se, no estudo apresentado nesta dissertação, a eficiência do uso de modelos probabilísticos implementados com as linguagens propostas para a classificação supervisionada de textos. Para verificar a eficiência das implementações, são comparadas as implementações determinísticas tradicionais conhecidas na área com as implementações utilizando linguagens probabilísticas.

## 1.4 Metodologia

A metodologia apresentada envolve dois pontos principais: a pesquisa exploratória e avaliação teórica e experimental.

Na pesquisa exploratória serão analisadas linguagens de programação probabilísticas e seus mecanismos. Os tópicos abordados são:

- Como as linguagens probabilísticas funcionam e como podem ser utilizadas para tarefas na área de aprendizado de máquina.
- A forma de descrição e implementação de modelos probabilísticos.
- O funcionamento dos mecanismos probabilísticos: uso de famílias de distribuições, geradores de amostras aleatórias, quais os tipos de inferência disponíveis.

- Quais as fundamentações teóricas necessárias para a compreensão da área de linguagens probabilísticas: revisão bibliográfica.

A segunda parte da metodologia é referente à descrição, avaliação e implementação de um classificador supervisionado de textos utilizando programação probabilística. Serão analisados os seguintes tópicos:

- Especificação de métricas para avaliação de desempenho das implementações.
- Facilidade de desenvolvimento do classificador: expressividade em alto nível, facilidade na especificação do problema, facilidade no uso de famílias de distribuições.
- Vantagens e desvantagens da implementação probabilística quando comparados com as abordagens tradicionais determinísticas.

## 1.5 Divisão do Conteúdo

A dissertação apresentada está dividida da seguinte forma: o Capítulo 2 apresenta uma revisão das áreas de probabilidade e estatística com a descrição dos conceitos principais que serão utilizados nos capítulos posteriores. Os Capítulos 3 e 4 apresentam, respectivamente, as análises das áreas de Redes Bayesianas e de Mecanismos de Amostragem. Essas duas áreas são fundamentais para a compreensão da área da programação probabilística e abrangem o conteúdo teórico necessário para a implementação do sistema probabilístico definido nos capítulos seguintes.

O Capítulo 5 apresenta um estudo de caso comparativo para a tarefa de criação de classificadores supervisionados de texto. Apresenta-se a implementação de três versões de classificadores utilizando a linguagem determinística *Python* e uma versão utilizando a linguagem probabilística *JAGS*. O Capítulo 6 apresenta as conclusões finais da dissertação e as considerações sobre o estudo proposto na dissertação.

# Capítulo 2

# Probabilidade e Estatística

Este capítulo apresenta uma introdução aos conceitos da teoria de probabilidades que serão utilizados nos demais capítulos. São revisados os tópicos que envolvem as definições e propriedades para o cálculo de probabilidades, valores de expectância e variância, o processo de amostragem aleatória e métodos para estimadores de parâmetros. O conteúdo apresentado no capítulo é baseado no conteúdo apresentado em [Casella e Berger 1990].

## 2.1 Teoria da Probabilidade

Um *espaço amostral* é um conjunto $S$, contável ou incontável, contendo todos os resultados possíveis de um determinado experimento. Um *evento* é um subconjunto de $S$, e representa um conjunto de resultados do experimento que são do nosso interesse. Seja $\mathcal{B}(S)$ o conjunto de todos os eventos que podem ser definidos no espaço amostral $S$[1] . Uma função de probabilidade sobre um espaço amostral é uma função $P : \mathcal{B}(S) \rightarrow \mathbb{R}$ que possui as seguintes características para todo evento $A \in \mathcal{B}$:

- $P(A) \geq 0$.
- $P(A) \leq 1$.
- $P(S) = 1$.

[1]No caso geral não podemos simplesmente fazer $\mathcal{B}(S) = \mathcal{P}(S)$, ou seja, nem todo subconjunto de $S$ pode ser um evento. Os detalhes para a definição correta do conjunto $\mathcal{B}(S)$ não são apresentados neste trabalho, sendo objeto de estudo da *Teoria da Medida*.

- Se $A_1, A_2, \cdots \in \mathcal{B}$ são disjuntos dois a dois, então $P(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} P(A_i)$ (*princípio da aditividade contável*).

Como caso especial do princípio da aditividade contável, temos para dois eventos disjuntos $A$ e $B$ que $P(A \cup B) = P(A) + P(B)$. Se $A^c$ é o complemento do evento $A$, então $P(S) = P(A) + P(A^c)$ pois $A$ e $A^c$ são disjuntos e $A \cup A^c = S$. Ou seja,

$$1 = P(A) + P(A^c) \Leftrightarrow P(A^c) = 1 - P(A).$$

Com base nas relações acima, podemos derivar outras propriedades da função de probabilidade $P$ (para dois eventos $A$ e $B$):

- $P(\emptyset) = 0$, onde $\emptyset$ é o conjunto vazio.
- $P(A \cap B)^c = 1 - P(A \cap B)$.
- $P(B \cap A^c) = P(B) - P(A \cap B)$.
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$.
- $P(A) \leq P(B)$, quando $A \subset B$.
- $P(A \cap B) \geq P(A) + P(B) - 1$.

### 2.1.1 Independência e Probabilidade Condicional

Dois eventos são considerados *independentes* quando $P(A \cap B) = P(A)P(B)$. A probabilidade condicional $P(A|B)$ é entendida como a probabilidade do evento $A$ dado que $B$ ocorra.

O valor da probabilidade condicional para dois eventos $A$ e $B$, com $P(B) > 0$, em um espaço amostral $S$ pode ser calculado da seguinte forma:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}.$$

Para lidar com probabilidades condicionais é muitas vezes necessário usar a Equação 2.1, denominada de *Regra de Bayes*. A simplificação da equação da *Regra de Bayes* é possível

ao aplicar as propriedades $P(A \cap B) = P(A|B)P(B)$ e $P(A \cap B) = P(B|A)P(A)$.

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)} \tag{2.1}$$

Para eventos $A$ e $B$ independentes, $P(A|B) = P(A)$ e $P(B|A) = P(B)$, como pode ser facilmente verificado pelas propriedades anteriores.

### 2.1.2 Variáveis Aleatórias

Uma variável aleatória $X$ é definida como uma função $X : S \to \mathbb{R}$ de um espaço amostral $S$ para os números reais. Dado um espaço amostral $S = \{s_1, \ldots, s_n\}$ então

$$P_X(X = x_i) = P(\{s_j \in S : X(s_j) = x_i\}) \tag{2.2}$$

A função de distribuição acumulada, *fda*, para uma variável aleatória $X$, denotada por $F_X(x)$, é definida por

$$F_X(x) = P_X(X \geq x) \text{ para todo } x. \tag{2.3}$$

Uma variável aleatória $X$ será contínua ou discreta de acordo com $F_X(x)$. As variáveis aleatórias $X$ e $Y$ serão identicamente distribuídas se, para cada evento $A$, $P(X \in A) = P(Y \in A)$. Uma forma conveniente de calcular probabilidades para valores de uma variável aleatória é baseada na função densidade de probabilidade (*fdp*) ou na função de probabilidade (*fp*), onde a primeira é referente ao casos contínuos e a segunda aos casos discretos:

- Função de probabilidade, *fp*, de uma variável aleatória discreta $X$ é dada por: $f_X(x) = P(X = x)$ para todo x.

- Função densidade de probabilidade, *fdp*, de uma variável aleatória contínua $X$ é a função que satisfaz: $F_X(x) = \int_{-\infty}^{x} f_X(t)dt$ para todo $t$.

## 2.2 Expectância e Variância

Seja $X$ uma variável aleatória e $x_1, \ldots, x_n$ é o conjunto dos valores de $X$. Se $X$ é uma variável aleatória com função densidade $F_X(x)$ então qualquer função aplicada a $X$, tal qual $g(X)$, também é uma variável aleatória.

A *expectância* de uma variável aleatória é o valor médio dos valores para a sua distribuição de probabilidades. Seja a variável aleatória $g(X)$, sua expectância $E(g(X))$ é definida por:

$$E(g(X)) = \begin{cases} \int_{-\infty}^{\infty} g(x) f_X(x) dx & \text{se X for contínuo} \\ \sum_{x \in \mathcal{X}} g(x) f_X(x) = \sum_{x \in \mathcal{X}} g(x) P(X = x) & \text{se X for discreto} \end{cases} \tag{2.4}$$

A *variância* de uma variável aleatória $X$ é definida por

$$Var(X) = E(X - E(X))^2$$

e o seu valores de desvio padrão é a raiz quadrada positiva de $Var(X)$. A variância é uma medida para grau de dispersão de uma distribuição ao redor de seu valor médio.

## 2.3 Amostras Aleatórias

**Definição 2.1.** O conjunto $X_1, \ldots, X_n$ de variáveis aleatórias será uma *amostra aleatória*, de tamanho $n$ e definidas por $f(x)$, se as variáveis aleatórias do conjunto forem mutuamente independentes e se a $fdp$ ou $fp$ marginal de cada $X_i$ também tiver como base a função $f(x)$.

Em uma amostra aleatória, cada $X_i$ é uma observação de uma variável $X$ e sua distribuição marginal é dada por $f(x)$. Cada observação feita não tem relação com qualquer uma das outras observações, dada a propriedade de independência mútua. A $fdp$ ou $fp$ conjunta para as variáveis $X_1, \ldots, X_n$ é definida por

$$f(x_1) \times f(x_2) \times \cdots \times f(x_n) = \prod_{i=1}^{n} f(x_i). \tag{2.5}$$

Quando a $fdp$ ou $fp$ de uma população é relacionada a uma forma paramétrica $f(x|\theta)$,

estas podem ser representadas com a Equação 2.6.

$$f(x_1, \ldots, x_n|\theta) = \prod_{i=1}^{n} f(x_i|\theta) \tag{2.6}$$

### 2.3.1 Estatísticas de Amostras Aleatórias

Seja $T(x_1, \ldots, x_n)$ uma função cujo domínio é o espaço amostral de um conjunto aleatório $X_1, \ldots, X_n$, o resultado $Y = T(X_1, \ldots, X_n)$ é um vetor, unitário ou não, de valores.

**Definição 2.2.** A variável ou o vetor aleatório $Y = T(X_1, \ldots, X_n)$ é denominado uma estatística. Exemplos de estatísticas são a média, variância e desvio padrão.

**Definição 2.3.** A *média amostral* é a média aritmética dos valores em uma amostra aleatória

$$\overline{X} = \frac{X_1 + \cdots + X_n}{n} = \frac{1}{N} \sum_{i=1}^{n} X_i.$$

**Definição 2.4.** A *variância amostral* é a estatística definida por

$$S^2 = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \overline{X})^2.$$

O desvio *padrão amostral* é a estatística definida por $S = \sqrt{S^2}$.

## 2.4 Estimação Pontual

Quando a amostragem aleatória é feita em populações cuja $fdp$ ou $fp$ é parametrizada na forma $f(x|\theta)$, ao aplicar diferentes valores de $\theta$ na função é possível analisar o comportamento da população inteira. O uso de estimadores é uma forma utilizada para determinar valores para os parâmetros das funções.

**Definição 2.5.** Um *estimador* é uma função aplicada às variáveis aleatórias $X_1, \ldots, X_n$ e uma *estimativa* é o valor resultado da aplicação de um estimador. Um *estimador pontual* é uma função $W(X_1, \ldots, X_n)$ aplicada a uma amostra.

Uma estatística é um estimador pontual, e.g., a média da distribuição a posteriori pode ser utilizada como uma estimativa pontual de $\theta$.

Calcular uma estimativa para um parâmetro depende da complexidade do modelo aplicado. Alguns métodos podem ser utilizados para o cálculo das estimativas como o *método dos momentos*, *estimadores de máxima verossimilhança* [Leroux 1992], *maximização da expectância* [Moon 1996] e os *estimadores de Bayes* [O'HAGAN e Leonard 1976].

### 2.4.1 Estimadores de Bayes

Em distribuições paramétricas, o valor de $\theta$ é desconhecido e deve ser estimado da forma mais eficiente possível. Se $X_1, \ldots, X_n$ é uma amostra aleatória de uma população com parâmetro $\theta$, os valores observados em uma amostragem ajudam a determinar os valores mais adequados para o parâmetro. Na abordagem *Bayesiana* o valor de $\theta$ pode ser assumido como uma distribuição a *priori* e aplicada ao processo de amostragem. Os valores das amostras obtidas atualizam a distribuição *a priori* criando então uma distribuição *a posteriori* em um processo que utiliza a regra de *Bayes*.

Seja $\pi(\theta)$ a distribuição *a priori* e $f(x|\theta)$ a distribuição amostral, a distribuição *a posteriori*, $\pi(\theta|x)$, segue a Equação 2.7 com

$$f(x|\theta)\pi(\theta) = f(x, \theta).$$

A função $m(x)$ é a distribuição marginal de $X$,

$$m(x) = \int f(x|\theta)\pi(\theta)d\theta.$$

$$\pi(\theta|x) = \frac{f(x|\theta)\pi(\theta)}{m(x)} \tag{2.7}$$

A distribuição *a posteriori* é condicionada à observação da amostra e é utilizada para analisar os valores de $\theta$. Para qualquer distribuição amostral existe uma família natural de distribuições *a priori*, denominada de família conjugada.

# Capítulo 3

# Redes Bayesianas

Este capítulo apresenta a definição formal de uma rede *Bayesiana* e descreve as suas propriedades. O Capítulo 5 apresenta um estudo comparativo entre classificadores de texto e uma implementação utilizando a linguagem probabilística *JAGS* que utiliza a estrutura de redes *Bayesianas* para o mecanismo de amostragem aleatória com *Cadeias de Markov com Monte Carlo*.

## 3.1 Propriedades de uma Rede Bayesiana

**Definição 3.1.** Seja $Z$ um conjunto de variáveis, uma *rede Bayesiana* é um par $(G, \theta)$. O grafo $G$ é a *estrutura* da rede e $\theta$ é o conjunto de tabelas de probabilidades condicionais de cada variável $Z$, que corresponde à *parametrização* da rede.

Uma rede *Bayesiana* $(G, \theta)$ permite representar, de forma compacta, um modelo probabilístico. Ao especificar uma rede *Bayesiana* é preciso determinar as tabelas de probabilidade condicionais para cada variável do grafo. Em certos casos não é possível especificar todos os valores das tabelas de forma *a priori*, então o objetivo dos mecanismos de inferência é o de definir os valores de forma aproximada tomando como base as outras variáveis do grafo.

**Definição 3.2.** O grafo $G$ de uma rede *Bayesiana* é do tipo direto e acíclico e possui dois componentes principais: as variáveis e as arestas. A relação entre as variáveis do grafo é determinada pelas tabelas de probabilidades condicionais.

O grafo na Figura 3.1 apresenta uma estrutura de rede *Bayesiana* e a Tabela 3.1 apresenta as tabelas de probabilidade condicionais para as variáveis.

**Definição 3.3.** Quando o estado de uma variável do grafo é conhecido então esta variável está *instanciada*. Uma *instanciação* de uma rede *Bayesiana* $(G, \theta)$ é a atribuição específica de estados para todas as variáveis $Z \in G$.

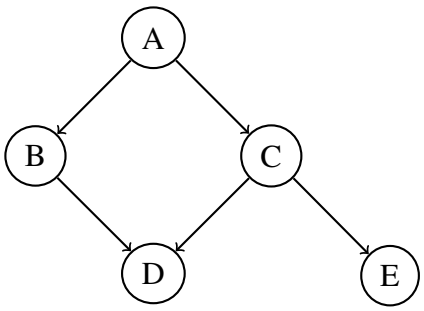



Figura 3.1: Exemplo de rede Bayesiana.

| B | C | D | $\Theta_{D\|BC}$ |
|---|---|---|---|
| V | V | V | .95 |
| V | V | F | .05 |
| V | F | V | .9 |
| V | F | F | .1 |
| F | V | V | .8 |
| F | V | F | .2 |
| F | F | V | 0 |
| F | F | F | 1 |

| C | E | $\Theta_{E\|C}$ |
|---|---|---|
| V | V | .7 |
| V | F | .3 |
| F | V | 0 |
| F | F | 1 |

| A | C | $\Theta_{C\|A}$ |
|---|---|---|
| V | V | .8 |
| V | F | .2 |
| F | V | .1 |
| F | F | .9 |

| A | B | $\Theta_{B\|A}$ |
|---|---|---|
| V | V | .2 |
| V | F | .8 |
| F | V | .75 |
| F | F | .25 |

| A | $\Theta_A$ |
|---|---|
| V | .6 |
| F | .4 |

Tabela 3.1: Tabelas de probabilidades condicionais para a Figura 3.1.

O total de combinações possíveis entre tabelas de probabilidade condicionais das variáveis de um grafo é exponencial em relação à quantidade de variáveis e expressa na forma $k^n$ com $k$ representando o número de estados possíveis para cada variável $n$ o número total de variáveis do grafo.

**Definição 3.4.** Seja $V$ uma variável do grafo. Denota-se por *Pa(V)* os nós parentes de $V$, i.e., todas as variáveis que possuem uma aresta direcionada para a variável $V$). O conjunto

*Desc(V)* representa todas as variáveis descendentes de $V$, i.e., todos os nós em que $V$ possui uma aresta direcionada. E *NonDesc(V)* serão todas as variáveis que não estão no conjunto *Pa(V)* e *Desc(V)*.

Uma propriedade importante de uma rede *Bayesiana* é a possibilidade de poder determinar a independência entre as variáveis da estrutura, simplificando o cálculo relacionado às tabelas de distribuições.

**Definição 3.5.** A independência de variáveis em um grafo direto acíclico é denotado pela função *Ind(V, Pa(V), NonDesc(V))* estabelecendo que uma variável $V$ é independente do conjunto de suas variáveis não-descendentes, *NonDesc(V)*, dado o conjunto de variáveis parentes *Pa(V)*.

Exemplificando de acordo com as propriedades definidas na rede *Bayesiana* da Figura 3.1, é possível determinar que:

- $Pa(B) = \{A\}$.
- $Desc(A) = \{B, C\}$.
- $Ind(B, A, C)$.
- $Ind(C, A, B)$.
- $Ind(D, C, E)$.

Na literatura relacionada, outras metodologias fazem uso do conceito básico de redes *Bayesianas* como base para teorias de redes causais [Pearl 2000], redes de crença [Darwiche 2009] e redes de representação de conhecimento [Barber 2012]. Essas metodologias ampliam o uso das redes *Bayesianas* adicionando novas formas de interpretação da estrutura da rede e dos modelos probabilísticos.

## 3.2 Representação Gráfica

Exemplos de redes *Bayesianas* como na Figura 3.1 apresentam uma estrutura simples, geralmente utilizada para apresentar os conceitos principais da forma de representar o grafo. Em

linguagens probabilísticas a estrutura do grafo criado é mais complexo e a sua representação precisa ser melhor descrita.

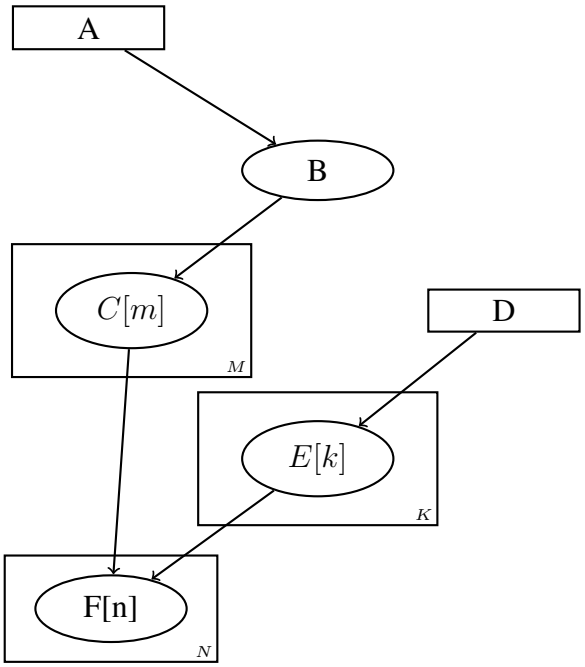



Figura 3.2: Exemplo de rede Bayesiana com estutura de grafo

A Figura 3.2 apresenta uma rede *Bayesiana* com algumas propriedades adicionais em sua estrutura. Os nós do grafo, que representam as variáveis, podem ser descritos de duas formas: em formato de elipse ou de retângulo. As representações com elipses determinam que a *variável é estocástica* (i.e. os nós $B, C, E, F$) e as representações com retângulos são as *variáveis determinísticas* e os seus valores são fixos (i.e. os nós $A, D$).

Também é utilizada a *plate notation* [Buntine 1994], representado por retângulos maiores que contêm variáveis, tal como no exemplo da variável $C$:

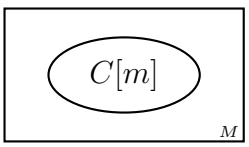



Figura 3.3: Exemplo de variável com plate notation

Essa notação indica que a variável $C$ é repetida $M$ vezes na estrutura do grafo. Com a *plate notation* a visualização do grafo é compacta. Por exemplo, para cada iteração do processo de amostragem serão geradas $M$ da variável $C[m], m \in M$.

## 3.3 Independência e d-separação

A análise da estrutura da rede *Bayesiana* permite verificar a propriedade de independência de acordo com as conexões entre variáveis.

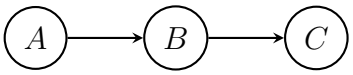



Figura 3.4: Exemplo de conexão sequencial

Uma conexão é *sequencial* quando apresenta uma configuração similar à Figura 3.4. Supondo uma variável $B$ instanciada (i.e. seu estado é conhecido) então não será possível manter a dependência condicional entre as variáveis $A$ e $C$ e nesse caso as variáveis tornam-se independentes. Para esse caso, as variáveis $A$ e $C$ são independentes e *d-separadas* por $B$.

**Definição 3.6.** A *d-separação* é um teste gráfico que permite verificar a propriedade de independência entre variáveis em um grafo. Seja $X, Y, Z$ um conjunto disjunto de variáveis em um grafo direto acíclico $G$. As variáveis $X$ e $Y$ são *d-separadas* por $Z$, descrito por $dsepG(X, Y, Z)$, se, e somente se, todo caminho possível entre as variáveis $X$ e $Y$ for *bloqueada* pela variável $Z$. Um caminho é considerado *bloqueado* por $Z$ se, e somente se, ao menos um ponto do caminho entre as variáveis $X$ e $Y$ estiver fechado dado a variável $Z$.

A Figura 3.5 exemplifica o caso de uma conexão *divergente*. A variável $A$ pode *influenciar* suas variáveis descendentes desde que não esteja instanciada; caso contrário, quando $A$ está instanciada, as variáveis $\{B, C, D\}$ serão *d-separadas* por $A$.

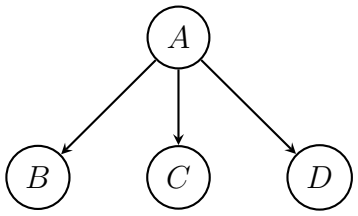



Figura 3.5: Conexão divergente

O terceiro caso, apresentado na Figura 3.6, é o de conexões *convergentes*. Se a variável $A$ não está instanciada, então seu estado pode ser determinado através de suas variáveis parentes $\{B, C, D\}$. Nesse caso os ancestrais de $A$, $Pa(A)$ são *independentes*.

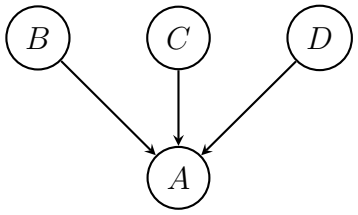



Figura 3.6: Conexão convergente

Para que duas variáveis distintas $A$ e $B$ em uma rede *Bayesiana* sejam *d-separadas*, é necessário que as condições sejam satisfeitas:

1. Para quaisquer variáveis $A$ e $B$ que possam ser conectadas através de um caminho qualquer no grafo, é preciso que exista uma variável intermediária $C$ que seja distinta de $A$ e $B$.

2. Caso divergente: o caminho que conecta as variáveis $A$ e $B$ deve possuir uma conexão sequencial ou divergente. A variável intermediária $C$ deve ter seu estado conhecido, ou seja, deve estar instanciada.

3. Caso convergente: caso a conexão entre $A$ e $B$ seja convergente, a variável $C$, ou os seus descendentes no grafo, devem estar instanciados.

As variáveis $A$ e $B$ são estruturalmente independentes se, e somente se, elas são *d-separadas*. E quando as variáveis $A$ e $B$ não são *d-separadas* então elas estão *d-conectadas*. Essas condições determinam uma propriedade importante para a análise de estrutura em redes *Bayesianas*, a *cobertura de Markov*.

**Definição 3.7.** A *cobertura de Markov* de uma variável $A$ é o conjunto dos seus parentes, $Pa(A)$, dos seus descendentes, $Desc(A)$, e de todas as demais variáveis que compartilham um descendente com $A$. Uma vez instanciada, $A$ é *d-separada* do resto da rede.

É importante destacar que a *d-separação* entre duas variáveis indica que elas são independentes mas o inverso não é necessariamente verdadeiro. O uso do conceito da *d-separação* auxilia a determinar quais variáveis são independentes no grafo simplificando os cálculos de probabilidades.

## 3.4 Modelos Naives Bayes

Modelos *Naive Bayes* são modelos Bayesianos mais simples onde um conjunto de variáveis são independentes dado uma variável parente, como demonstrado na Figura 3.7. Para os casos em que uma *Naive Bayes* é utilizada para tarefas de classificação, a variável $C$ pode representar uma categoria e as variáveis $O_1 \dots O_n$ os objetos que devem ser classificados.

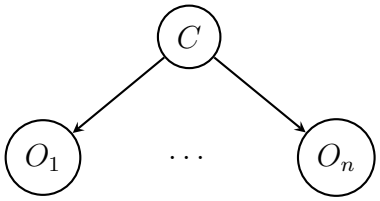



Figura 3.7: Independência entre variáveis Naive Bayes

A distribuição de probabilidade condicional para a *Naive Bayes* segue as seguintes propriedades:

- A variável $C$ possui probabilidade $P(C)$.
- Para todas as variáveis $O$, é preciso determinar a probabilidade condicional distribuída $P(O|H)$.
- A probabilidade posterior para $C$ é calculada por $P(H|o_1, \dots, o_n) = \mu P(C) \prod_{i=1}^{n} P(o_i|H)$ com $\mu = 1/P(o_1, \dots, o_n)$ como uma constante de normalização.

A complexidade para os cálculos em uma *Naive Bayes* é linear em relação ao número de variáveis. O modelo de *Naive Bayes* é utilizado de forma prática no caso de uso de classificação supervisionada, descrito no Capítulo 5.

## 3.5 Inferência em Redes Bayesianas

Inferência em redes *Bayesianas* é um processo em que é estipulado quais os melhores valores de parâmetros para as distribuições de probabilidades modeladas na rede. Podemos categorizar os tipos de inferência em *exata* e *aproximativa*.

Na categoria de inferência exata destacam-se dois conjuntos de técnicas: as de *eliminação de variáveis* e a de *condicionamento de variáveis*. A complexidade para a inferência exata é exponencial ao número de variáveis no grafo analisado e para casos em que o modelo probabilístico possui um grande número de variáveis, utilizar algoritmos exatos para a inferência na rede *Bayesiana* associada pode ser inviável. Em razão da complexidade associada, os mecanismos de inferência exatos não são abordados nesse estudo. Para referências de algoritmos desse tipo, existem referências em [Darwiche 2009].

A inferência do tipo *aproximativa* é utilizada em conjunto com o processo de amostragem aleatória realizando simulação de estados para as variáveis do grafo considerando os valores de probabilidades disponíveis. O resultado de uma amostragem aleatória é baseada nas frequências de eventos ocorridos no processo de simulação.

A Figura 3.1 apresenta um exemplo de rede *Bayesiana* genérica cuja distribuição é $P(X)$ e $X = \{A, B, C, D, E\}$ representa o conjunto de variáveis. A inferência aproximada utiliza valores de expectância e de desvio padrão para determinar a qualidade das amostras geradas.

O processo de simular a rede *Bayesiana* envolve criar uma instanciação que atribui um valor de estado para cada variável. A ordem de atribuição dos estados define primeiro valores para as variáveis paternas e depois as variáveis descendentes. A Tabela 3.2 apresenta uma amostra aleatória com sete iterações. Para cada iteração são escolhidos, aleatoriamente, valores para cada uma das variáveis da rede *Bayesiana* de forma que o algoritmo que gera uma amostra aleatória opera em tempo linear, de acordo com o número de variáveis.

Tabela 3.2: Exemplo de uma amostra aleatória

| Iteração | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | V | V | V | V | V |
| 2 | V | F | V | V | F |
| 3 | F | F | F | F | F |
| 4 | F | V | F | F | T |
| 5 | V | V | V | F | V |
| 6 | V | V | V | V | V |
| 7 | F | V | F | F | T |

# Capítulo 4

# Mecanismos de Amostragem

Este capítulo apresenta o processo de amostragem utilizada para inferência aproximada. O principal método analisado é o de *Cadeias de Markov com Monte Carlo*, *MCMC*, e suas principais implementações. Também é apresentado uma análise comparativa entre três sistemas para gerar amostras aleatórias e que implementam algoritmos baseados no *MCMC*: *BUGS/WinBUGS*, *JAGS* e *Stan*.

## 4.1 Processo de Amostragem

Retomando o exemplo de rede *Bayesiana* descrito no Capítulo 3. A Figura 4.1 apresenta a estrutura de grafo da rede *Bayesiana* e o conjunto de tabelas de probabilidades descrevendo o relacionamento entre as variáveis do grafo.

A complexidade do processo de inferência para a rede *Bayesiana* descrita está relacionada a dois fatores: (i) a quantidade de variáveis do grafo e (ii) a quantidade de estados possíveis que podem ser determinados em cada variável. Esse crescimento de complexidade é *exponencial* de acordo com esses dois fatores. Quando o processo de inferência exata é computacionalmente complexo, uma alternativa são os processos de inferência aproximados que utilizam amostragem aleatória.

Gerar uma amostra aleatória de uma rede *Bayesiana* significa gerar valores para cada uma das variáveis, esse resultado é denominado de *distribuição amostral*. O processo de amostragem pode ser aplicado em casos univariados ou multivariados, discretos ou contínuos.

Supondo que a rede *Bayesiana* apresentada possui uma distribuição de probabilidade, denotada por $p(X)$ com $X = \{A, B, C, D, E\}$. Para gerar uma amostra aleatória de tamanho $n$ a partir da distribuição $p(X)$ é preciso gerar uma sequência de instanciações $x^1, x^2, \ldots, x^n$ da rede *Bayesiana* onde o processo utilizado para gerar cada instância $x^i$ é determinar o valor de $p(x^i)$. A Tabela 4.2 apresenta um conjunto de seis amostras geradas para a rede *Bayesiana*.

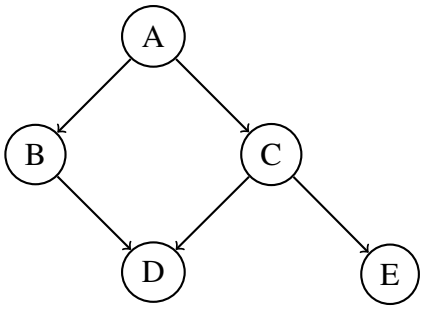



Figura 4.1: Exemplo de rede Bayesiana.

| B | C | D | $\Theta_{D\|BC}$ |
|---|---|---|---|
| V | V | V | .95 |
| V | V | F | .05 |
| V | F | V | .9 |
| V | F | F | .1 |
| F | V | V | .8 |
| F | V | F | .2 |
| F | F | V | 0 |
| F | F | F | 1 |

| C | E | $\Theta_{E\|C}$ |
|---|---|---|
| V | V | .7 |
| V | F | .3 |
| F | V | 0 |
| F | F | 1 |

| A | C | $\Theta_{C\|A}$ |
|---|---|---|
| V | V | .8 |
| V | F | .2 |
| F | V | .1 |
| F | F | .9 |

| A | B | $\Theta_{B\|A}$ |
|---|---|---|
| V | V | .2 |
| V | F | .8 |
| F | V | .75 |
| F | F | .25 |

| A | $\Theta_A$ |
|---|---|
| V | .6 |
| F | .4 |

Tabela 4.1: Tabelas de probabilidades condicionais para a Figura 4.1.

Tabela 4.2: Exemplo de amostras geradas para a distribuição $p(X)$ com $X = \{A, B, C, D, E\}$

| | A | B | C | D | E |
|---|---|---|---|---|---|
| $x^1$ | V | V | V | V | V |
| $x^2$ | V | F | V | F | V |
| $x^3$ | F | F | F | F | F |
| $x^4$ | F | V | F | F | V |
| $x^5$ | V | V | V | F | V |
| $x^6$ | V | V | V | V | V |

**Definição 4.1.** Se $\mathcal{X} = \{x^1, \ldots, x^N\}$ é um conjunto de amostras geradas para uma determinada distribuição $p(X)$ e $X$ é o conjunto de variáveis, o processo de amostragem estabelece que, dada uma quantidade suficiente de amostras, $N$, os valores gerados são compatíveis com a distribuição original, $p(X)$.

A probabilidade conjunta de cada variável em $X$ corresponde ao produto de suas funções de probabilidade e será equivalente à distribuição amostral $\mathcal{X}$ que é gerada. Dessa forma, $\mathcal{X}$ é um processo amostral compatível com a distribuição de probabilidade original $X$.

Uma observação importante é que para os casos em que a distribuição é univariada, como é o caso de $p(X)$, o processo de geração de amostras é menos complexa, tanto para os casos discretos quanto para os casos contínuos [1].

Para os casos multivariados, como o exemplo utilizado na Figura 4.1, uma abordagem utilizada é a conversão para uma distribuição unidimensional e o processo de amostragem segue da mesma forma que os casos univariados [Barber 2012]. A abordagem não é eficiente para casos em que o número de variáveis é grande porque número de estados aumenta exponencialmente e o processo de conversão é ineficiente. Uma segunda alternativa é a de dividir a distribuição inicial em um produto de distribuições com dimensões menores.

Aplicando um formalismo para casos em geral, tomemos o conjunto de variáveis $X = \{X_1, \ldots, X_n\}$. A distribuição conjunta das variáveis $p(X_1, X_2)$ pode ser decomposta na relação da Equação 4.1. Essa operação permite criar amostras primeiro para a variável $p(X_1)$

[1] Nos casos contínuos utilizam-se cálculos com integrais para determinar valores de amostras que ocupem o espaço de valores permitidos pelas variáveis.

e depois criar as amostras condicionadas para $p(X_2|X_1)$.

$$p(X_1, X_2) = p(X_2|X_1)p(X_1) \tag{4.1}$$

A Equação 4.1 pode ser generalizada quando $p(X_1, \ldots, X_n)$ pode ser decomposta em

$$p(X_1, \ldots, X_n) = p(X_n|X_{n-1}, \ldots, X_1)p(X_{n-1}|X_{n-2}, \ldots, X_1) \ldots p(X_2|X_1)p(X_1)$$

## 4.2 Cadeias de Markov com Monte Carlo

O *Monte Carlo*, *MC*, é uma classe de métodos estatísticos utilizados para simulação de processos aleatórios [Robert, Casella e Robert 1999; Doucet et al. 2001]. Uma variação do Monte Carlo, as Cadeias de Markov com Monte Carlo (*Markov Chain Monte Carlo*, *MCMC*), é um mecanismo importante que auxilia no processo de geração de amostras para distribuições de probabilidades.

Iniciado com o algoritmo *Metropolis*, publicado em $1953$ [Metropolis et al. 1953], os mecanismos que implementavam *MCMC* começaram a ter relevância em pesquisas acadêmicas, principalmente nas áreas da física e da química. Com o aumento da capacidade computacional, na década de 1990 o *MCMC* começou a se tornar conhecido em outras áreas, com um destaque para a área da estatística computacional. Diversas variantes e implementações de MCMC foram desenvolvidas tais como os algoritmos *Metropolis*, *Metropolis-Hastings*, *Metropolis-Hastings-Green* e o *Amostrador de Gibbs*.

O *Amostrador de Gibbs*, descrito na Seção 4.2.3, implementa um processo de simulação que resulta na distribuição *a posteriori* da distribuição original utilizada. Essa característica permite usar os *Amostradores de Gibbs* para inferência Bayesiana [Geman e Geman 1984].

Uma descrição geral do *MCMC*, incluindo a demonstração e implementação do mecanismo pode ser analisado em [Brooks et al. 2011].

O *MCMC* é utilizado para os casos em que a inferência exata do modelo probabilístico é complexo. Dessa forma, o algoritmo *MCMC* não realiza amostragem na distribuição original $p(X)$ mas procura uma nova distribuição que, dado um número suficiente de amostras geradas, seja possível determinar que esta nova distribuição gera amostras que podem ser

consideradas compatíveis com $p(X)$. Seja a distribuição multivariada

$$p(X) = \frac{1}{Z}p'(X)$$

com $p'(X)$ representando uma nova distribuição (o $Z$ é uma constante de normalização). Nesse processo estabelece-se que $p'(X)$ é uma distribuição de probabilidade computável e menos complexa de calcular do que a distribuição de probabilidade $p(X)$.

### 4.2.1 Cadeias de Markov

Para uma Cadeia de Markov é preciso definir uma distribuição inicial, $X_1$, e uma distribuição de probabilidade de transição para qualquer $X_{n+1}$ dado $X_n$.

**Definição 4.2.** Uma sequência $X_1, X_2, \ldots, X_n$ de variáveis aleatórias é uma Cadeia de Markov se, e somente se, a distribuição condicional de um elemento $X_{n+1}$ dado $X_1, \ldots, X_n$ depender apenas de $X_n$, ou seja, o processo depende apenas do estágio atual, $P(X_{n+1}|X_n)$.

Para os casos em que a distribuição condicional de $X_{n+1}$ dado $X_n$ não depende do índice $n$, então diz-se que a Cadeia de Markov possui uma *distribuição de probabilidades de transição estacionárias*. Uma Cadeia de Markov é estacionária se, e somente se, possui um processo estocástico estacionário, logo conclui-se que a Cadeia de Markov é estacionária se a distribuição marginal de $X_n$ não depende de $n$. Para esses casos diz-se que a simulação *atingiu o seu estado de equilíbrio*.

**Definição 4.3.** Uma distribuição de probabilidade de transição preserva a sua distribuição inicial quando ambas a distribuição inicial e a distribuição de transição são estacionárias.

Uma propriedade importante de uma Cadeia de *Markov* é a *reversibilidade* que ocorre quando um conjunto de variáveis aleatórias $p(X_{i+1}, \ldots, X_{i+k})$ é reversível, ou seja, $p(X_{i+1}, \ldots, X_{i+k}) = p(X_{i+k}, \ldots, X_{i+1})$ ($i$ e $k$ são índices no conjunto $X$). Os valores de probabilidades de transição são reversíveis de acordo com a distribuição inicial [Kipnis e Varadhan 1986]. É importante destacar que a propriedade de reversibilidade implica em estacionariedade mas não vice-versa [Brooks et al. 2011].

### 4.2.2 Cadeias de Markov com Monte Carlo

O *MCMC* segue as teorias estabelecidas no Monte Carlo Comum [Freedman, Pisani e Purves 2007] e no Teorema Central do Limite. O *Monte Carlo Comum* (*Ordinary Monte Carlo*, *OMC*) estabelece que o conjunto $X_1, X_2, \ldots$ é independente e identicamente distribuída e a Cadeia de Markov formada é estacionária e reversível. O *Teorema Central do Limite*[2] (*Central Limit Theorem*, *CLT*) aplicado na Cadeia de Markov auxilia na estimação dos valores de variância.

Com o desenvolvimento da estatística em sistemas computacionais, e com a maior capacidade de processamento dos computadores, tornou-se possível utilizar o *MCMC* para fazer simulações a um custo computacional aceitável. Existem implementações do *MCMC* que trabalham na forma de *black boxes* (i.e. implementações do mecanismo do *MCMC* de forma especializada sem que haja necessidade de conhecimento avançado em cálculo estatístico ou da teoria associada aos mecanismos). Na seção 4.3 são exemplificados casos de implementações do *MCMC* como sistemas especialistas.

Um dos objetivos na simulação utilizando *MCMC* é assegurar que o processo atinja o estado estacionário, ou estado de equilíbrio. Em determinadas ocasiões pode acontecer uma pseudo-convergência, quando a cadeia aparenta estar em um estado estacionário quando na verdade não está [Brooks et al. 2011]. É comum a utilização da técnica de *burn-in* que elimina da cadeia gerada parte dos valores nas primeiras iterações. A justificativa para a eliminação é de que nas primeiras iterações o estado de equilíbrio pode não ser alcançado facilmente, então essas informações podem atrapalhar, ou não acrescentar nada de relevante, o resultado final para análise. Algumas técnicas optam por não utilizar essa técnica de *burn-in* para não descartar informações do início das iterações na cadeia que poderiam ser utilizadas para obter mais informações sobre o sistema de amostragem em geral.

### 4.2.3 O Algoritmo MCMC

Os métodos baseados nas Cadeias de Markov com Monte Carlo, utilizados para inferência *Bayesiana*, geram amostras aleatórias em simulações estocásticas. Exemplos de variantes que implementam o *MCMC* são o algoritmo *Metropolis* [Chib e Greenberg 1995] e o *Amos-*

[2] O Teorema Central do Limite determina que a média de um conjunto de variáveis aleatórias independentes, com um tamanho grande o suficiente, pode ser representado como uma distribuição Normal

*trador de Gibbs* [Casella e George 1992].

Técnicas desenvolvidas com o *MCMC* geram resultados mais precisos para estimar distribuições *a posteriori* e a base do conceito do *MCMC* utiliza cadeias de Markov para convergir para as distribuições desejadas quando atinge o momento de equilíbrio, obtendo a distribuição posterior desejada. Essa característica diferencia o *MCMC* dos métodos de simulação direta. As Cadeias de Markov possuem a característica de que cada etapa de execução depende apenas da etapa anterior, não necessitando ter informações de todas as etapas já realizadas, caracterizando o método como um procedimento iterativo. Duas propriedades são importantes para a construção de uma cadeia de Markov: (i) é preciso ser possível gerar a função de probabilidade condicional[3] $P(X_{t+1}|X_t)$, e (ii) a distribuição de equilíbrio deve ser equivalente à distribuição posterior. Os passos para construir e executar uma cadeia de Markov são:

1. Selecionar um valor inicial $X_i$ .
2. Gerar amostras até que a distribuição de equilíbrio seja alcançada.
3. Verificar a convergência do algoritmo durante a geração das amostras.
4. Analisar o conjunto de observações iniciais e, se for o caso, eliminar amostras da etapa de *burn-in*.
5. Fazer análise estatística com os dados das amostras coletadas calculando informações como média, mediana, desvio padrão ou qualquer outra métrica necessária.

Nem todas as implementações *MCMC* seguem as etapas definidas acima mas algumas dessas etapas são importantes para a implementação do método *MCMC* tais como a distribuição de equilíbrio, convergência do algoritmo, iteração para gerar amostras, valores iniciais e o período de *burn-in*.

Algumas considerações sobre as etapas de execução da Cadeia de Markov precisam ser observadas. A técnica de *burn-in* elimina uma parte inicial do conjunto de valores amostrais porque na fase inicial o algoritmo *MCMC* está em fase de adaptação e ainda não está gerando amostras em estado de convergência. Também é preciso atentar para a possibilidade de uma

[3] A notação $X_t$ é utilizada para indicar que a variável $X$ é analisada em um dado momento $t$.

pseudo-convergência da cadeia gerada e nesse caso a cadeia aparenta estar em estado de convergência quando na verdade não está. Utilizar trechos da cadeia em um estado não convergido resultará em uma análise final errônea sobre o processo simulado. Uma discussão sobre esses dois problemas é feita em [Barber 2012].

A distribuição de equilíbrio (i.e. distribuição estacionária ou distribuição alvo) é o momento em que a cadeia estabiliza e apresentará valores de $X$ e $X_{t+1}$ idênticos. A partir desse momento a cadeia gerada pode ser utilizada para análise estatística. A cadeia gerada pelo algoritmo converge quando esta atinge o seu equilíbrio e gera valores compatíveis com a distribuição original para a qual se faz a simulação.

O processo de amostragem é feito de forma iterativa e cada iteração corresponde a um ciclo de execução do algoritmo que gera valores para os parâmetros da distribuição a partir da distribuição posterior. Não existe um número específico de iterações para que as amostras sejam obtidas corretamente porém quanto maior for o número de iterações maiores são as chances de obter valores amostrais precisos. O desafio é conciliar um número mínimo, mas suficiente, para que as amostras obtidas estejam no período de convergência.

O processo definido para gerar amostras é uma parte importante para o desenvolvimento de algoritmos *MCMC*. Quando a distribuição original é preservada, o mecanismo é dito *elementar* [Brooks et al. 2011]. Uma forma de provar a corretude de algoritmos MCMC é demonstrar que o algoritmo é um caso especial do algoritmo *Metropolis-Hastings-Green* [Green 1995].

### Metropolis-Hastings

O mecanismo de atualização implementado no *Metropolis-Hastings* inicialmente determina uma função chamada de *proporção de Hastings*, definida na Equação 4.2.

$$r(X, Y) = \frac{h(Y)q(Y, X)}{h(X)q(X, Y)} \tag{4.2}$$

O $X$ é o estado atual na cadeia, $Y$ é o estado para o qual é proposta a transição, a função $q$ determina a densidade de probabilidade condicional de $X \rightarrow Y$ e a função $h$ representa a densidade não-normalizada da distribuição que se deseja obter com o amostrador *MCMC* [Hastings 1970]. A probabilidade de aceitar ou não a transição é determinada pelo

valor de probabilidade da função $a(X, Y) = 1 - r(X, Y)$. O mecanismo de atualização do *Metropolis-Hastings* garante reversibilidade com relação a função $h$.

Uma alternativa para o *Metropolis-Hastings* é o mecanismo de atualização *Metropolis*, que apresenta uma versão simplificada da Equação 4.2 resultando na Equação 4.3.

$$r(X, Y) = \frac{h(Y)}{h(X)} \tag{4.3}$$

**Amostrador de Gibbs**

A técnica de *Cadeia de Markov com Monte Carlo*, *MCMC*, permite estimar a expectância sem precisar realizar amostragem diretamente da distribuição original $P(X)$. O *MCMC* faz uma simulação usando *Cadeias de Markov* criando uma cadeia que gera amostras a partir de uma distribuição simulada $\mathbb{P}(X)$.

O algoritmo *MCMC* opera de forma linear e não precisa fazer amostragem diretamente da distribuição original $P(X)$ e sua maior vantagem é poder computar as expectâncias para distribuições que seriam difíceis de realizar amostra usando outras formas de inferência.

Seja uma distribuição $P(X)$, com $m$ variáveis, a matriz de transição de *Gibbs*, que determina a transição de uma variável para outra, segue as regras da Equação 4.4. Uma cadeia de *Markov* com essas propriedades de transição é uma cadeia de *Gibbs*.

O processo de simulação utiliza uma distribuição original $P(X)$, cria uma distribuição simulada $\mathbb{P}(X)$ e verifica se $\mathbb{P}(X_t) = P(X)$ em um determinando momento $t > 1$. Caso essa condição seja satisfeita, então a distribuição simulada é estacionária e pode ser considerada como uma distribuição válida para $P(X)$.

$$\mathbb{P}(X_i = x' | X_{i-1} = x) \begin{cases} 0 \text{ , caso (i)} \\ 1/m P(s' | X - S) \text{ , caso (ii)} \\ 1/m \sum_{S \in X} P(s_x | x - S) \text{ , caso (iii)} \end{cases} \tag{4.4}$$

Os seguintes casos são definidos para a transição:

- (i) Se $x$ e $x'$ diferem em mais de uma variável.
- (ii) Se $x$ e $x'$ diferem em uma variável $S$, que possui valores $s'$ em $x'$.

- (iii) Se $x = x'$ e $x_s$ é o valor da variável $S$ em $x$.

Seja $X = \{A, B, C\}$ o conjunto de variáveis e $x = a, b, \bar{c}$ uma instanciação da rede, a tabela 4.3 apresenta os casos para a matriz de transição de *Gibbs* para a distribuição[4] $P(X)$. Para determinar quais casos são válidos, o mecanismo cria instanciações $x'$ rede e compara com a instanciação original $x$ calculando a probabilidade de que $x'$ ocorra dado $x$. A intenção é determinar se a instanciação $x'$ pode ser considerada como uma amostra válida para $x$.

A Tabela 4.3 exemplifica o conceito de uma cadeia de *Gibbs*.

Tabela 4.3: Exemplo de uma matriz de transição de Gibbs

| x' | $\mathbb{P}(x'\|x = a, b, \bar{c})$ | x' e x diferem em |
|---|---|---|
| a, b, c | $P(c\|a, b)/3$ | C |
| a, b, $\bar{c}$ | $P(a\|b, \bar{c}) + P(b\|a, \bar{c}) + P(\bar{c}\|a, b))$ | - |
| a, $\bar{b}$, c | 0 | B,C |
| a, $\bar{b}$, $\bar{c}$ | $P(\bar{b}\|a, \bar{c})/3$ | B |
| $\bar{a}$, b, c | 0 | A,C |
| $\bar{a}$, b, $\bar{c}$ | $P(\bar{a}\|b, \bar{c})/3$ | A |
| $\bar{a}$, $\bar{b}$, c | 0 | A,B,C |
| $\bar{a}$, $\bar{b}$, $\bar{c}$ | 0 | A,B |

## 4.3 Implementações do MCMC

Nesta seção são apresentadas implementações do processo de amostragem aleatória utilizando *MCMC*. Estes mecanismos servem como referência para o Capítulo 5, principalmente para a discussão apresentada na Seção 5.4.

### 4.3.1 BUGS e WinBUGS

O projeto *BUGS* (*Bayesian Inference Using Gibbs Sampling*) propõe um ambiente para análise Bayesiana de modelos estatísticos complexos utilizando cadeias de Markov com Monte Carlo. Iniciado em 1989 por pesquisadores da área de estatística em Cambridge, teve como

[4]Detalhe de nomenclatura: $P(a)$ indica a situação em que $a = V$ e $P(\bar{a})$ indica a situação em que $a = F$

principal foco o desenvolvimento do *WinBUGS*, um software que disponibiliza uma linguagem de programação para auxiliar a criação de amostras aleatórias em modelos Bayesianos. Por ser um software com interface gráfica, diversas funcionalidades facilitam o desenvolvimento dos modelos e a análise estatística das amostras geradas. Possui três componentes principais: especificação do modelo, organização dos dados e configuração dos valores iniciais (valores de parâmetros utilizados nos modelos).

A linguagem utilizada pelo *WinBUGS* é similar ao *S*, também utilizada pela linguagem *R*, facilitando o desenvolvimento de aplicações estatísticas quando já se tem conhecimento prévio na área ou conhecimentos básicos de linguagens de programação. Os passos para desenvolver aplicações utilizando o *WinBUGS* são:

1. Definir o modelo e os valores iniciais dos parâmetros.

2. Inicializar, compilar e executar o modelo, gerando as amostras aleatórias.

3. Analisar e validar os dados obtidos. O *WinBUGS* fornece ferramentas para análise estatística.

O *OpenBUGS*[5] [Open BUGS] é uma vertente *open-source* do projeto *BUGS/WinBUGS* [WinBUGS 2013]. Embora o *WinBUGS* seja uma ferramenta antiga e não esteja mais em desenvolvimento, ainda continua sendo uma referência para pesquisadores que estudam *MCMC* e inferência Bayesiana. Uma boa literatura de referência com exemplos e casos de uso utilizando o *WinBUGS* pode ser encontrada em [BUGS 2013; Open BUGS; WinBUGS 2013].

### 4.3.2 JAGS

Outro projeto para a criação de modelos e geração de amostras aleatórias é o *JAGS* (*Just Another Gibbs Sampler*), um sistema para análise de modelos hierárquicos Bayesianos utilizando simulação de cadeias de Markov com Monte Carlo. O objetivo do projeto é oferecer as mesmas funcionalidades encontradas no *BUGS/WinBUGS* com a possibilidade de executar o sistema em vários sistemas operacionais (o *WinBUGS* funciona apenas em versões do

[5]Informações sobre o funcionamento do *OpenBUGS* e detalhes sobre os mecanismos *MCMC* implementados podem ser acessados em http://www.openbugs.info/Manuals/Introduction.html

sistema operacional Windows). O *JAGS* não possui uma interface gráfica como o *WinBUGS* e seu uso é feito através de execuções por linha de comando.

O *JAGS* permite adicionar extensões ao sistema permitindo que os usuários desenvolvam pacotes de funções e amostradores personalizados [JAGS 2013]. Na página de exemplos do projeto é possível encontrar uma coletânea de casos de uso de aplicações do *JAGS*[6]. Apesar de não possuir uma interface gráfica, o JAGS possui uma integração com a linguagem *R*, através do módulo *rjags*, permitindo que aplicações desenvolvidas na plataforma *R* possam fazer chamadas para as funções do *JAGS* de forma transparente ao usuário e também permite que toda a análise das amostras geradas possa ser feita pelos pacotes especializados do *R*, tais como o *coda* [coda: Output analysis and diagnostics for MCMC 2012].

### 4.3.3 Stan

O *Stan* [Stan Development Team 2013], assim como o *JAGS*, também gera amostras aleatórias a partir de modelos probabilísticos, não possui uma interface gráfica e seu uso é feito por chamadas em linha de comando. Utiliza uma variação das cadeias de Markov com Monte Carlo conhecido como amostragem Monte Carlo Hamiltoniano (*Hamiltonian Monte Carlo*, HMC) [Ntzoufras 2011].

Para o mecanismo de amostragem com o *Stan* é preciso definir um modelo probabilístico utilizando de funções de probabilidade condicionais no estilo $p(\theta|y;x)$ onde $\theta$ representa os valores desconhecidos (i.e. parâmetros), $y$ representa a sequência de valores modelados já conhecidos e $x$ representa uma sequência de constantes não modeladas. O *Stan* é estruturado de forma semelhante ao *WinBUGS/BUGS*, a construção e execução de modelos é feita de forma semelhante ao definido no *BUGS/WinBUGS* e *JAGS*.

### 4.3.4 Escolha do Sistema

O *Stan* é um projeto recente, iniciado em $2010$ e sua proposta é ser uma alternativa ao *JAGS*. Para modelos simples o *Stan* funciona relativamente bem e com um bom desempenho (i.e. tempo para execução, facilidade de descrição do modelo, análise dos resultados gerados) mas para modelos mais complexos, com um grande número de parâmetros no modelo, o seu uso

[6]Disponível em http://sourceforge.net/projects/mcmc-jags/files/Examples/3.x/

tende a ser mais lento que a execução feita no *JAGS*. Em testes realizados utilizando o modelo do estudo de caso definido no Capítulo 5 que realiza amostragem em centenas de milhares de variáveis, o *Stan* se mostrou incapaz de realizar a geração das amostras aleatórias enquanto que o *JAGS* conseguiu executar o mesmo modelo de forma satisfatória. Por esse motivo, o *JAGS* foi escolhido para as implementações probabilísticas descritas nesta dissertação.

A Tabela 4.4 apresenta as principais características de cada sistema analisado. Não existe um padrão que determine o uso de um sistema específico que seja mais eficiente. A complexidade das tarefas e dos modelos analisados determinam qual sistema é melhor adequado para o processo de amostragem.

Tabela 4.4: Resumo das principais características de cada sistema.

| | Mecanismo | Características |
|---|---|---|
| R | Diversos via pacotes extras | Open Source (GNU GPL). Multiplataforma. Interface gráfica. Integração com JAGS |
| WinBUGS | Amostrador de Gibbs | Proprietário. Apenas Windows. Interface gráfica. Projeto descontinuado |
| OpenBUGS | Amostrador de Gibbs, Metropolis Hastings | Open Source (GNU GPL). Multiplataforma. Interface gráfica. Integração com R |
| JAGS | Amostrador de Gibbs | Open Source (GNU GPL2). Multiplataforma. Linha de comando |
| Stan | Hamiltonian Monte Carlo | Open Source (BSD). Multiplataforma. Linha de comando |

# Capítulo 5

# Estudo Comparativo para Classificação Supervisionada de Textos

Este capítulo utiliza as análises realizadas nos capítulos anteriores e propõe uma aplicação prática para o uso de linguagens de programação probabilísticas [Lucena, Brito e Formiga 2013].

Uma das técnicas estudadas na área de aprendizado de máquina é o da classificação supervisionada. É uma técnica que utiliza um conjunto finito de objetos previamente classificados e então analisa formas de construção de classificadores que possam classificar, de forma eficiente, novos objetos.

Para o estudo comparativo apresentado neste capítulo são analisadas implementações de classificadores *Naive Bayes* e uma implementação puramente probabilística utilizando o mecanismo *JAGS*, descrito no Capítulo 4. Para uma análise completa, primeiro é descrito a teoria referente ao uso de *Naive Bayes* na Seção 5.1, seguido pela descrição da técnica de classificação supervisionada de documentos em 5.2. O estudo comparativo é apresentado na Seção 5.3 com as quatro implementações de classificadores de texto que serão analisadas. O Capítulo 5.4 amplia a análise do estudo comparativo descrevendo, de forma detalhada, o funcionamento dos mecanismos de amostragem implementados pelo *JAGS*.

## 5.1 Teoria do Naive Bayes

Dado um conjunto de dados (ou instância, vetor) $d$ e um conjunto de categorias $C = \{c_i, \ldots, c_n\}$, a tarefa de atribuir uma categoria $c_i$ ao vetor $d$ representa uma função de classificação. O processo de classificação pode ser entendida como uma função de categorização, como definido em 5.1, e é um dos métodos desenvolvidos na área de aprendizado de máquina.

$$f_{NB}(d) \rightarrow c_i \tag{5.1}$$

O *Naive Bayes* é um mecanismo simples utilizado para classificação de vetores de dados e utiliza o princípio básico da *Regra de Bayes*. É um processo de contagem de frequências e de atribuição de valores de probabilidades entre os conjuntos de dados e as categorias possíveis. Dois requisitos são necessários para a construção de uma *Naive Bayes*, (i) é preciso apresentar um conjunto de vetores de dados iniciais, $D$, previamente classificados, (ii) um mecanismo que crie uma distribuição de frequências a partir dos vetores de dados iniciais e construa a função de classificação que será a base para a classificação de novos vetores de dados. As etapas para a implementação de um Naive Bayes são:

1. Analisar o conjunto de dados inicial, $D = \{d_1, \ldots, d_n\}$, já categorizados.
2. Criar uma distribuição de frequências dos vetores de dados em $D$.
3. Construir a função de classificação, $f_{NB}$.
4. Aplicar a função para classificação de novas instâncias.

Determinar o valor de probabilidade de uma instância $d^*$ pertencer a uma classe $c_i$ significa calcular $p(c_i|d^*)$. Utilizando a *Regra de Bayes* é possível calcular o valor de probabilidade necessário.

$$p(c_i|d^*) = \frac{p(d^*|c_i)\, p(c_i)}{p(d^*)} = \frac{p(d^*|c_i)\, p(c_i)}{\sum_{c_i \in C} p(d^*|c_i)\, p(c_i)} \tag{5.2}$$

Em uma *Naive Bayes*, assume-se que o valor de probabilidade associado a cada elemento de um vetor é independente, como demonstrado na Figura 5.1 com $n$ igual ao tamanho do

conjunto $d$. Essa característica simplifica a *Naive Bayes* para a Equação 5.3. A eficiência da Naive Bayes depende da quantidade de elementos utilizados para treinamento e quanto maior o número de elementos maior será o grau de precisão para o valor de probabilidade de $p(d_i|c)$.

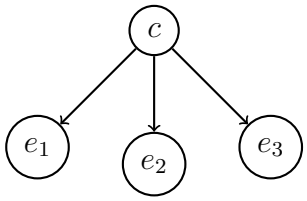



Figura 5.1: Elementos de um vetor são independentes dada uma categoria.

$$p(d, c) = p(c) \prod_{i=1}^{n} p(d_i|c) \tag{5.3}$$

Ao classificar um novo vetor $d^*$ o valor do denominador de 5.2, $p(d^*)$, será o mesmo, independente da categoria analisada e por ser uma divisão constante o denominador divide o numerador de maneira proporcional. A Equação 5.4 apresenta uma versão simplificada para a classificação de novos vetores.

$$p(c_i|d^*) = p(d^*|c_i)\, p(c_i) \tag{5.4}$$

O processo de construção de um classificador Naive Bayes, resume-se, então ao cálculo de dois valores, (i) o valor de verossimilhança $p(d^*|c_i)$ e (ii) o valor da probabilidade a priori $p(c_i)$. O modelo do classificador pode ter uma quantidade variável de categorias em $C$ e os casos mais simples envolvem duas classes de categorias, denominados de classificadores binários[1].

[1]Para versões de classificadores com mais de duas categorias, referências em [Barber 2012] demonstram técnicas que podem ser utilizadas.

## 5.2 Classificação Supervisionada de Documentos

Mecanismos para classificação de documentos estão na intersecção entre as áreas de aprendizado de máquina e processamento de linguagem natural e ambas estudam técnicas para a tarefa de classificação. Uma aplicação prática é a tarefa de classificação de documentos de texto.

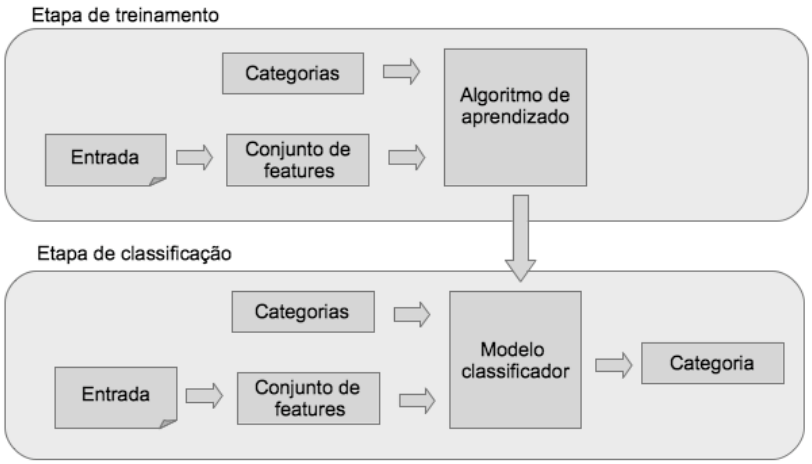
A classificação de documentos é uma tarefa que envolve a escolha de uma categoria para um determinado documento de texto: a função $C_{NB}$, definida na Equação 5.5, recebe um documento $d$ e determina a categoria $c_i \in C$ mais provável para o documento. Para este caso específico de classificador, deve-se determinar qual é a categoria com maior valor de probabilidade a *posteriori* retornado pela função *Naive Bayes*.

$$C_{NB}(D) \rightarrow c_i \in C \tag{5.5}$$

Existem diversas técnicas para a criação da função de classificação [Barber 2012], dentre elas estão a classificação manual, supervisionada e semi-supervisionado. O método escolhido para a implementação da *Naive Bayes* neste capítulo é o da classificação supervisionada. Nesse tipo de classificador é preciso fornecer um conjunto de documentos de texto previamente categorizados, o *corpora*, para ser utilizado na fase de treinamento do classificador. A classificação de novos documentos será dependente da quantidade de informações fornecidas previamente.

O processo de classificação é feito em três etapas: (i) organização dos documentos categorizados que serão usados pelo classificador, (ii) treinamento do classificador e (iii) classificação de novos documentos. A Figura 5.2 apresenta as etapas para a construção do classificador *Naive Bayes*.

Figura 5.2: Etapas para a construção de um classificador Naive Bayes




### 5.2.1 Organizando os Corpora

Os *corpora* são os conjuntos de documentos de texto que serão utilizados pelo classificador nas fases de treinamento e de teste. Para garantir que as duas etapas não estejam enviesadas é preciso que os *corpora* originais sejam divididos em subconjuntos distintos.

Os subconjuntos serão criados em uma proporção de $70\%/30\%$: $70\%$ dos documentos de texto serão utilizados para a tarefa de treinamento do classificador e os $30\%$ restantes serão utilizados para teste.

### 5.2.2 Classificação com Naive Bayes

O componente principal de um classificador de documentos são as palavras que compõem os documentos e cada palavra contribui no cálculo de probabilidades utilizado na *Naive Bayes*. O objetivo do processo é determinar qual a probabilidade de que um dado documento pertença à uma determinada categoria. Utilizando a Equação 5.3 da seção anterior, a *Regra de Bayes* aplicada ao classificador de documentos de texto é reescrita na Equação 5.6.

$$p(d, c_i) = p(c_i) \prod_{w \in d} p(w|c_i) \tag{5.6}$$

O objeto $d$ representa um documento, $w$ o conjunto de palavras de $d$ e $c$ uma categoria. De acordo com os passos da Seção 5.1, o classificador *Naive Bayes* é feito em duas etapas:

- Calcular o valor *a priori* de $p(c_i)$ para cada $c_i \in C$.
- Determinar o valor de verossimilhança $p(d|c_i)$ do documento dada uma categoria $c_i$.

O termo *Naive* refere-se ao fato de que em um documento a probabilidade de ocorrência de cada palavra é considerada independente das demais palavras no texto, como demonstrado na Figura 5.1. O classificador *Naive Bayes* está relacionado ao processo de contagem de frequências das palavras e uma palavra possui um peso maior para o valor da verossimilhança quando ela tem uma frequência maior em uma dada categoria. Para alguns casos analisados, quanto maior for o conjunto de treinamento, o corpora, mais preciso será o classificador.

Um problema a ser observado são os casos em que um documento a ser classificado possui palavras que não estão presentes no conjunto de treinamento. Ao encontrar uma palavra que não pertence ao conjunto de treinamento, os valores de probabilidade de $p(w|c)$ será zero e, por consequência, o valor de $p(c_i|d) = 0$ e o processo de classificação será falho. Para essas situações são apresentadas alternativas tais como *smoothing* ou *laplace weight* [Bird, Klein e Loper 2009; Segaran 2007]. Essas soluções, de forma geral, adicionam um valor padrão para o valor de probabilidade $p(w|c)$, evitando valores zero nos cálculos.

### Cálculos da Priori e Máxima Verossimilhança

Dada a Equação 5.6, o objetivo do classificador é determinar o valor que satisfaça o cálculo do máximo *a posteriori*, definido na Equação 5.7.

$$C_{MAP} = argMAX_{c_i \in C} \quad p(c_i|d) \tag{5.7}$$

$$= argMAX_{c_i \in C} \quad p(d|c_i)\, p(c_i) \tag{5.8}$$

$$= argMAX_{c_i \in C} \quad p(w_1, \ldots, w_n|c_i)\, p(c_i) \tag{5.9}$$

O conjunto $\{w_1, \ldots, w_n\} \in d$ representa as palavras de um vocabulário. Se as palavras são consideradas independentes então

$$p(w_1, \ldots, w_n|c) = p(w_1|c)\, p(w_2|c)\, \ldots p(w_n|c) \tag{5.10}$$

A formulação final do classificador *Naive Bayes*, $C_{NB}$, é

$$C_{NB} = C_{MAP} = argMAX_{c_i \in C} \quad p(c_i) \prod_{w \in d} p(w|c_i) \tag{5.11}$$

O cálculo da *priori* $p(c_i)$ é feito com base no conjunto de documentos disponibilizados para o treinamento. Utilizando todos os documentos de uma categoria as palavras são extraídas e colocadas em uma *bag of words* junto com a frequência de cada palavra. O processo envolve contar a quantidade de documentos em cada categoria, $qtde_{doc}(c = c_i)$ e dividir pela quantidade total de documentos, $total_{doc}$. O processo é definido na Equação 5.12.

$$p(c_i) = \frac{qtde_{doc}(c = c_i)}{total_{doc}} \tag{5.12}$$

Para o cálculo da verossimilhança $p(w|c_i)$, com $w \in d$ e $c_i \in c$, é preciso determinar o valor de probabilidade da Equação 5.13. O numerador determina quantas vezes uma palavra $w$ ocorre na categoria $c_i$ e o denominador conta a quantidade total de palavras de todos os documentos na categoria $c_i$.

$$p(w|c_i) = \frac{qtde(w, c_i)}{\sum_{d \in D} qtde(w, c_i)} \tag{5.13}$$

Com as etapas da *piori* e verossimilhança calculadas, o processo de classificação é calculado de forma direta. No estudo de caso apresentado na Seção 5.3 são apresentadas as implementações de três classificadores e é proposta uma implementação de classificação utilizando mecanismos de amostragem aleatória utilizando Cadeias de Markov com Monte Carlo.

### 5.2.3 Métricas de Avaliação

Para determinar a eficiência de um classificador é preciso estabelecer métricas para avaliar o processo de acordo com os resultados obtidos. As métricas utilizadas [Bird, Klein e Loper 2009] são as estabelecidas na Tabela 5.1.

Como discutido na Seção 5.2.1, os dados para treinamento e teste devem ser separados sem que haja sobreposição de informações nos dois conjuntos. Definido o mecanismo de

classificação os dados de teste serão utilizados e os resultados da classificação serão analisados. Os dados no conjunto de testes já estão previamente categorizados então é possível determinar se o classificador está realizando a operação de forma correta.

Seja $d$ um documento do conjunto de testes cuja categoria associada seja $c_i$, o documento $d$ é submetido ao classificador $C_{NB}$ e o resultado pode ser avaliado de acordo com a Tabela 5.1.

Tabela 5.1: Descrição das saídas possíveis para o classificador

| | Saída | |
|---|---|---|
| Classes | Positivo | Negativo |
| Positivo | VP | FN |
| Negativo | FP | VN |

Os valores de saída para a tarefa de classificação podem ser de quatro tipos:

- $VP$ (verdadeiro positivo): são itens relevantes que foram corretamente classificados como relevantes.
- $VN$ (verdadeiro negativo): são itens irrelevantes que foram corretamente identificados como irrelevantes.
- $FP$ (falso positivo ou *Erro de Tipo I*): itens irrelevantes que foram incorretamente classificados como relevantes.
- $FN$ (falso negativo ou *Erro de Tipo II*): itens relevantes que foram incorretamente classificados como irrelevantes.

Três tipos de métricas são utilizadas. O primeiro, precisão, indica o total de itens que foram classificados de forma correta, *recall* indica quantos dos itens classificados como relevantes foram identificados. O cálculo do *F-Score* valor médio da combinação dos valores de *recall* e de precisão e será utilizado como indicador principal da eficiência do classificador.

- Precisão: $\frac{VP}{VP + FP}$
- Recall: $\frac{VP}{VP + FN}$

- F-Score: $\frac{2 \times \text{precisão} \times recall}{\text{precisão} + recall}$

## 5.3 Estudo Comparativo

Classificadores *Naive Bayes* normalmente são implementados com o uso de linguagens de programação determinísticas. O estudo comparativo proposto apresenta três implementações distintas de classificadores de texto *Naive Bayes* utilizando a linguagem *Python* e a quarta implementação de classificador utiliza a linguagem probabilística *JAGS*.

Diferente das outras implementações, na versão em *JAGS* são utilizadas funções de distribuição de probabilidades e estimação com geradores de amostras aleatórias. Demonstra-se as vantagens de uso desse tipo de classificador e o ganho de desempenho e precisão no resultado das métricas comparativas.

Tabela 5.2: Implementações de classificadores Naive Bayes

| | |
|---|---|
| Versão 1 | Classificador com Python |
| Versão 2 | Classificador com Python e NLTK |
| Versão 3 | Classificador otimizado com Python e NLTK |
| Versão 4 | Classificador com JAGS |

As implementações analisadas estão apresentadas na Tabela 5.2 e todas realizam a tarefa de classificação em uma mesma *corpora* cujos documentos de texto representam análises de filmes previamente classificados em duas categorias, positiva e negativa, com um total de $1000$ documentos de texto para cada categoria[2]. Para a etapa de treinamento foram separados, de forma aleatória e sem sobreposição, um subconjunto para treinamento e outro para teste do classificador.

[2]Os documentos do corpora utilizado está disponível em `http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz`.

Tabela 5.3: Organização dos corpora para os classificadores.

| | Quantidade de documentos | | |
|---|---|---|---|
| Categoria | Treinamento | Teste | Total |
| Positivo | 700 | 300 | 1000 |
| Negativo | 700 | 300 | 1000 |

### 5.3.1 Implementação em Python

A primeira implementação apresentada[3] utiliza a linguagem *Python* sem o uso de bibliotecas ou *frameworks* específicos. A primeira etapa da implementação, que extrai as palavras de um documento, é definida no Código Fonte 5.1. A função *tokenize()* recebe um documento de texto e retorna as palavras do documento em formato de uma lista.

Código Fonte 5.1: Função para extração de palavras de um arquivo de texto

```
1 def tokenize(str):
2     """ Tokenizer. Recebe uma String e retorna array de palavras
3     """
4     return re.findall('\w+', str.lower())
```

A etapa de treinamento consiste em analisar todos os documentos separados para a etapa e criar um dicionário para cada categoria utilizando a palavra como chave e a sua frequência como valor. O Código Fonte 5.2 demonstra a etapa de treinamento para os documentos da categoria positiva. A função *train_positivive()* inicia o treinamento com a chamada da função *getFrequency()* que recebe como argumento o caminho para o diretório onde os documentos da categoria estão armazenados. A função processa todos os documentos de texto do diretório e, para cada documento, extrai todas as suas palavras e em seguida cria um dicionário geral de palavras da categoria, que será a *bag of words*. O mesmo processo é repetido para a categoria de documentos de texto negativos.

Com a *bag of words* definida, a função *calculate_probabilities()* cria um novo dicionário similar à *bag of words* mas apenas com os valores de probabilidade da palavra pertencer às categorias positiva e negativa.

[3]Todas as implementações feitas neste capítulo estão disponíveis no endereço `https://github.com/labml/textclass`

Código Fonte 5.2: Processo de treinamento para uma categoria de documentos de texto

```
def train_positive (self):
    # freq de palavras nos textos positivos
    self.pos_freq = self.getFrequency(self.positive_corpus)
    # atualiza o dicionario da categoria
    self.update_counts(self.pos_freq, "pos")
def getFrequency(self, dirc):
    # objeto de frequencia inicial (vazio)
    freq = collections.Counter()
    for filename in os.listdir(dirc):
        # analisa conteudo do arquivo e atualiza
        # frequencia de ocorrencia de cada palavra
        with open(os.path.join(dirc, filename)) as f:
            words = tokenize(f.read())
            freq.update(words)
```

**Exemplo 5.1.** Suponha que se deseja calcular o valor de probabilidade para a palavra *"terrible"*, cujos valores na *bag of words* são $<' pos' : 30,' neg' : 700 >$, indicando que a palavra é mais frequente nos documentos categorizados como negativos do que nos positivos. Para os casos de palavras que serão classificadas mas não estão presentes na etapa de treinamento utiliza-se o *Laplace smoothing* que adiciona um valor de probabilidade padrão[4] evitando cálculos com valor zero.

Após o treinamento do classificador *Naive Bayes* define-se o mecanismo de classificação de documentos. A função *classify_item* definida no Código Fonte 5.3 demonstra o processo de classificação implementado. Para cada documento a ser classificado cria-se um dicionário de palavras únicas e para determinar a categoria mais provável do documento compara-se cada palavra com a *bag of words* do classificador para obter os valores de probabilidades correspondentes.

Código Fonte 5.3: Processo de classificação de um documento de texto

```
def classify_item(self, item):
    """ Classifica um item do conjunto de teste
    """
    words = tokenize(item)
    pos_probs = [self.__word_probability(w, 'pos') for w in words]
    neg_probs = [self.__word_probability(w, 'neg') for w in words]
    total_pos_logprob = sum([math.log(p) for p in pos_probs if p > 0.0])
    total_neg_logprob = sum([math.log(p) for p in neg_probs if p > 0.0])
    item_class = 'pos'
    if total_neg_logprob > total_pos_logprob:
        item_class = 'neg'
    return item_class
```

[4]O valor especificado para o *Laplace smoothing* é variável à cada implementção e geralmente é um valor pequeno, e.g. $0.1$.

### 5.3.2 Implementação em Python com NLTK

O *NLTK* é um *framework* desenvolvido em Python especializado na área de processamento de linguagem natural. Dentre as funcionalidades do *framework*, um conjunto de classes implementadas podem ser utilizadas para a tarefa de classificação de documentos de texto. O Código Fonte 5.4 demonstra um exemplo de uso da classe *NaiveBayesClassifier* [Bird, Klein e Loper 2009].

Código Fonte 5.4: Exemplo de uso da classe NaiveBayesClassifier

```
1  classifier = nltk.NaiveBayesClassifier.train(train_set)
2  print nltk.classify.accuracy(classifier, test_set)
```

A construção do classificador *Naive Bayes* com o *NLTK* é similar à implementação com *Python*. As funcionalidades do *NLTL* foram utilizadas em tarefas tais como a extração das palavras de um texto, a criação do dicionário de palavras treinadas e a forma de classificação. O Código Fonte 5.5 mostra a forma como as funções para extrair palavras e como criar o dicionário no formato especificado pelo *NLTK*.

Código Fonte 5.5: Implementação das funções para extração de palavras e criação da bag of words

```
1  def get_feature(word):
2      """ Transforma uma palavra em um item de dicionário para
3          o nltk
4      """
5      return dict([(word, True)])
6  def bag_of_words(words):
7      """ Método auxiliar, transforma uma lista de palavras em uma lista de
8          dicionários para o nltk
9      """
10     return dict([(word, True) for word in words])
```

A etapa de treinamento é feita com a função *nltk.NaiveBayesClassifier.train()*, demonstrada em 5.6. Após treinar cada categoria possível, o conjunto total de palavras é utilizado como parâmetro para a função de treinamento.

Código Fonte 5.6: Implementação da etapa de treinamento do classificador

```
1  def train(self):
2      self.train_category(self.positive_corpus, "pos")
3      self.train_category(self.negative_corpus, "neg")
4      self.trained_classifier = nltk.NaiveBayesClassifier.train(
5              self.training_set)
```

Na etapa de classificação também utiliza uma função especializada definida pelo *NLTK*, *classifier()*, que recebe um documento, extrai suas palavras e as utiliza no classificador já treinado para determinar a categoria mais provável para o documento. O processo é demonstrado no Código Fonte 5.7.

Código Fonte 5.7: Função de classificação utilizando NLTK

```
1 def classifier (self, doc):
2     # Lendo documento
3     words = []
4     with open(doc) as f:
5         tokens = tokenize(f.read())
6         words = words + tokens
7     test_set = bag_of_words([w.lower() for w in words])
8     return self. trained_classifier . classify ( test_set )
```

Essa primeira versão implementada com o *NLTK* utiliza o conjunto total de palavras fornecido pelos *corpora* de treinamento, dessa forma o classificador, ao receber novos documentos para categorizar irá utilizar todo esse conjunto de palavras para realizar a tarefa. É observado no dicionário de palavras para treinamento que algumas possuem uma frequência bem maior do que outras então é proposto uma segunda implementação, otimizada, do classificador que utiliza apenas um subconjunto com das palavras mais frequentes (ordenando o vocabulário geral pela frequência de ocorrência em ordem decrescente, foi utilizada a primeira metade mais frequente de palavras) encontradas nos documentos de treinamento. Para essa versão otimizada, a precisão final calculada foi maior do que na versão mais simples com *NLTK* porém esse resultado será dependente das palavras no documentos classificados.

O Código Fonte 5.8 apresenta o mecanismo utilizado para obter as palavras mais frequentes de um conjunto de palavras e o resultado é uma lista ordenada com a primeira metade de palavras mais frequentes. A *bag of words* para esse caso será reduzido mas as palavras armazenadas são as com maiores chances de ocorrer em novos documentos. O Código Fonte 5.9 apresenta a função *classifier()* otimizada.

Código Fonte 5.8: Mecanismo para obter o conjunto de palavras mais frequentes

```
1 def get_most_common(self, words):
2     common = nltk.FreqDist(w.lower() for w in words)
3     index = int(len(common)*0.50)
4     common = common.keys()
5     return common[:index]
```

Código Fonte 5.9: Versão otimizada do classificador NLTK

```
1  def classifier (self, doc, cat):
2      # Lendo documento
3      words = []
4      with open(doc) as f:
5          tokens = tokenize(f.read())
6          words = words + tokens
7      if cat == "pos":
8          most_common = self.common_words_pos
9      else:
10         most_common = self.common_words_neg
11     bag =[]
12     for w in words:
13         if w in most_common:
14             bag.append(w)
15     test_set = bag_of_words([w.lower() for w in bag])
16     return self. trained_classifier . classify ( test_set )
```

### 5.3.3 Implementação com Programação Probabilística

As versões implementadas nas seções anteriores são implementações tradicionais de classificadores *Naive Bayes* utilizando linguagens determinísticas. Segue-se a demonstração de uma implementação utilizando uma linguagem de programação probabilística, com o *JAGS* [JAGS 2013]. Nas avaliações realizados essa implementação apresenta um desempenho superior às implementações determinísticas e serão discutidos no fim deste capítulo.

Linguagens probabilísticas como o *JAGS* modelam distribuições de probabilidades e disponibilizam mecanismos para o cálculo de valores da *posteriori* utilizando amostragem aleatória com *MCMC*. Como o objetivo da construção da *Naive Bayes* é obter o valor para a distribuição posterior, a probabilidade condicional $p(c = c_i|d)$, então o mecanismo implementado pelo *JAGS* pode ser utilizado para a tarefa de classificação de documentos de texto.

No modelo *JAGS* devem ser especificadas todas as variáveis relacionadas ao problema e como elas interagem entre si. O *JAGS* utiliza dois arquivos para seus modelos, um arquivo de dados e o arquivo com a especificação modelo e das funções de distribuição. O Código Fonte 5.10 apresenta o arquivo de dados (com alguns valores omitidos, para simplificação). A variável $K$ representa o número de categorias, $N$ representa a quantidade de palavras nos documentos de treinamento, $V$ é a quantidade de palavras únicas e $M$ a quantidade de

documentos.

Algumas das variáveis definidas são vetores com dados que serão utilizados pelo modelo. A variável $z$ é um vetor de tamanho $M$ e seus valores representam a categoria de cada documento (e.g. $z_i = 1$ representa o documento na posição $i$ cuja categoria é 1). A variável $w$, de tamanho $N$, é um vetor com todas as palavras dos documentos. O vetor $doc$ representa todos os documentos e suas palavras, e seu tamanho é $\sum m \in M|d|$ (i.e. cada documento adiciona a quantidade de palavras contidas no seu texto ao tamanho total do vetor). As variáveis $alpha$ e $beta$ são definidas como *hiperparâmetros* do modelo probabilístico. O valor padrão de $alpha$ é $(1, 1)$; a variável $beta$ é o vetor que armazena os valores de probabilidade para a ocorrência de cada palavra.

A operação de classificação deve selecionar uma categoria dentre as opções de categorias disponíveis e então utiliza a distribuição *Categorical* (ou distribuição Categórica). A primeira tarefa para a construção do modelo classificador é determinar como serão calculados os valores de verossimilhança e da *priori*. Para estes dois cálculos é utilizada a distribuição conjugada da *Categorical*, a distribuição de *Dirichlet*.

Distribuição de *Dirichlet*, utilizada para inferência Bayesiana, é a conjugada *a priori* das distribuições *Categorical* e *Multinomial*. Por ser utilizado em distribuições de probabilidade que são contínuas e multivariadas, a distribuição de *Dirichlet* é uma simplificação da distribuição *Beta*. A distribuição *Categorical* representa uma distribuição de probabilidade discreta sobre um dado espaço amostral finito.

Para tarefas de processamento de linguagem natural como a classificação supervisionada de textos, a distribuição *Categorical* é utilizada para descrever a forma de seleção entre conjunto possível de resultados. No estudo de caso apresentado, trata-se da definição para um documento estar associado a uma determinada categoria. Por utilizar a distribuição *Categorical* como resultado a *posteriori*, a distribuição de *Dirichlet* pode ser utilizada para cálculos de *priori*.

O modelo probabilístico do *JAGS* é definido no Código Fonte 5.12. As funções $dcat()$ e $ddirch()$ representa, respectivamente, as funções probabilísticas *Categorical* e *Dirichlet*. No início do modelo são declaradas as variáveis que serão utilizadas para as funções de distribuição de probabilidade. Funções de distribuições de *Dirichlet* são utilizadas para determinar os valores de $theta$ e $phi[k, ]$. A variável $\theta_k(c_i)$ representa a probabilidade *a priori* que um

Código Fonte 5.10: Definição das variáveis no arquivo de dados do modelo JAGS

```
1 K <- 2
2 V <- 34434
3 M <- 1400
4 N <- 932648
5 z <- [...]
6 w <- [...]
7 doc <- [...]
8 alpha <- (1,1)
9 beta <- [...]
```

dado documento na posição $k$ pertence à categoria $c_i$. O $\phi(c_i, w_i)$ representa a probabilidade de uma palavra $w_i$ estar presente em um documento de classe $c_i$. A Equação 5.14 apresenta a especificação da variável $\theta$ no modelo probabilístico.

Código Fonte 5.11: Distribuições de Dirichlet utilizadas pelo modelo

```
1 theta ~ ddirch(alpha)
2 for (k in 1:K) {
3     phi[k,] ~ ddirch(beta)
4 }
```

$$\theta \sim Dirichlet(\alpha) \tag{5.14}$$

$$\phi(k) \sim Dirichlet(\beta) \tag{5.15}$$

A probabilidade de uma palavra na *i*-ésima posição ocorrer no documento $d$ é dado pela distribuição *Categorical* da Equação 5.16. A variável $z_d$ tem como valor a categoria para o documento $d$. A categoria para o documento $d$ é do tipo *Categorical* e está definida na Equação 5.17.

$$w_{i,d} \sim Categorical(\phi(z_d)) \tag{5.16}$$

$$z_d \sim Categorical(\theta) \tag{5.17}$$

O tamanho do vetor $\theta$ é $K$ (i.e. a quantidade de categorias possíveis) e a matriz $\phi$ tem dimensão $K \times V$ ($V$ é a quantidade de palavras únicas). No total, a quantidade de parâmetros que devem ser estimados é $V(K+1)$.

Código Fonte 5.12: Modelo JAGS para o classificador Naive Bayes

```
1  var theta[K], alpha[K], phi[K, V], beta[V], label[M], w[N];
2  model {
3    theta ~ ddirch(alpha)
4    for (k in 1:K) {
5      phi[k,] ~ ddirch(beta)
6    }
7    for (m in 1:M) {
8      z[m] ~ dcat( theta )
9    }
10   for (n in 1:N) {
11     w[n] ~ dcat(phi[ label[doc[n ]],])
12   }
13 }
```

Calculadas as distribuições para $theta$ e $phi$, o modelo realiza distribuições categóricas para determinar valores de $w$ e $z$. A finalidade do vetor $w$ é armazenar os valores de probabilidades de que uma determinada palavra $i$ no documento $d$, $w_{i,d}$, pertença à categoria 1 ou 2.

No Código Fonte 5.12, as linhas que definem as variáveis $theta$, $phi$, $w$ e $z$ são uma transcrição direta das equações definidas em 5.14, 5.15, 5.16 e 5.17, respectivamente. Essa é uma das principais vantagens de linguagens de programação probabilísticas como o *JAGS*: a facilidade de implementação de código na linguagem baseado nos modelos probabilísticos.

A Figura 5.3 apresenta o grafo direto acíclico equivalente ao modelo descrito no Código Fonte 5.12. O *JAGS* cria a representação do modelo em uma estrutura de grafo para aplicar o algoritmo *MCMC* e então fazer o processo de amostragem aleatória. É possível observar no grafo a descrição de todas as variáveis do modelo e a forma como estes se relacionam.

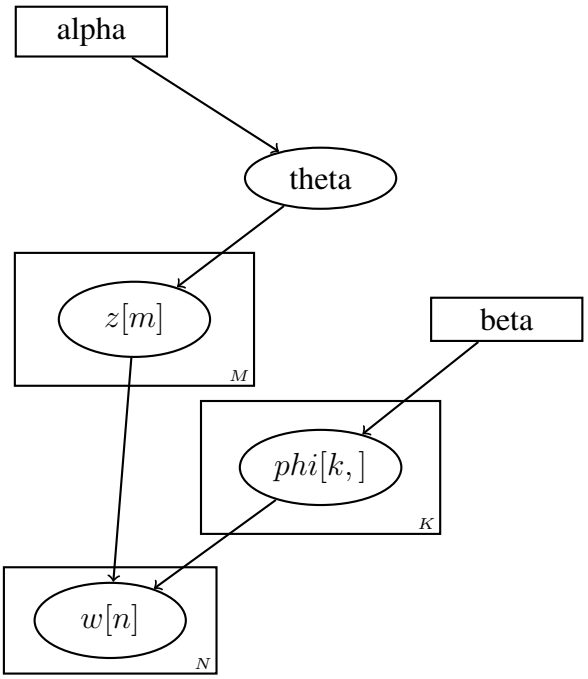



Figura 5.3: Rede Bayesiana para o modelo classificador supervisionado de documentos

A execução do processo de amostragem é feita de forma iterativa (no caso do *JAGS*, deve ser especificado no *script* de execução a quantidade de iterações que o algoritmo *MCMC* deve realizar). Concluída a especificação do modelo no *JAGS*, o algoritmo *MCMC* executa as instruções do modelo até que sejam gerados valores de amostras em estado de convergência para a variável $w$. O valor armazenado em cada variável $w_i$ representa a categoria mais provável para cada palavra $w_i$ do vocabulário. Os resultados armazenados em $w$ serão utilizados para a classificação de novos documentos.

## 5.4 Amostragem Aleatória no JAGS

Uma breve introdução ao *JAGS* foi feita no Capítulo 4 em conjunto com a descrição de outros mecanismos de amostragem aleatória. O objetivo específico deste capítulo é o de descrever o funcionamento, em baixo nível, do modelo apresentado no estudo de caso do Capítulo 5. Inicialmente é feita uma introdução detalhada ao mecanismo implementado pelo *JAGS* e uma revisão de suas principais funcionalidades. Em seguida analisa-se como o modelo descrito para classificação supervisionada é construído no *JAGS* e como é feito o processo de amostragem aleatória.

O *JAGS* (*Just Another Gibbs Sampling*) é um sistema utilizado para simular modelos probabilísticos e implementa mecanismos similares ao disponibilizados pelo *WinBUGS/OpenBUGS* (ambos baseados na linguagem BUGS). A principal vantagem do *JAGS* é ser uma alternativa multi-plataforma, de código aberto, e permitir a implementação de novos mecanismos de amostragem.

Assim como os outros projetos relacionados, o *JAGS* disponibiliza um conjunto de funções derivados do *BUGS*, facilitando a migração entre os sistemas. Também implementa famílias de distribuições de probabilidade e algoritmos de amostradores aleatórios para uso em redes Bayesianas. Para tarefas de inferência, o *JAGS* simula modelos Bayesianos usando Cadeias de Markov com Monte Carlo *MCMC*.

O funcionamento do *JAGS* é divido em etapas sequenciais: (i) definição do modelo e dos dados auxiliares, (ii) compilação, (iii) inicialização, (iv) fase de adaptação do algoritmo, (v) monitoramento dos valores de amostras gerados.

### 5.4.1 Modelo e Dados Auxiliares

Os modelos e as definições dos dados utilizados são definidos em arquivos separados. A linguagem utilizada no modelo segue um padrão similar à linguagem do *BUGS*, que é derivada da linguagem *S*.

O Código Fonte 5.12 apresenta o modelo *JAGS* criado para o estudo de caso do Capítulo 5. O *JAGS* converte o modelo em uma estrutura de grafo e cada variável definida representa um nó do grafo. Na definição

$$z[m] \sim dcat(theta)$$

o lado esquerdo define o nó $z[m]$ e o lado direito define como o nó terá valores gerados (nesse caso, o nó possui um relacionamento direto com o nó $theta$). O conjunto de nós define o grafo direto acíclico que será utilizado pelo mecanismo de amostragem. Para os casos que o nó é estocástico as definições usam o operador $\sim$ e para os casos de nós determinísticos, o operador utilizado será o <-. No *JAGS* os nós estocásticos representam as variáveis aleatórias do modelo e os nós determinísticos são utilizados apenas como suporte para relações entre os nós estocásticos.

É possível especificar um conjunto de dados que serão utilizados pelas variáveis do mo-

delo na sua fase inicial e devem ser especificados em um arquivo separado e formatados de acordo com a especificação da linguagem *R*.

### 5.4.2 Compilação e Inicialização

O arquivo do modelo deve definir todas as variáveis necessárias para o processo de amostragem e o grafo gerado deve ser direto e acíclico. Na fase de compilação é criado um grafo equivalente ao modelo e é disponibilizado para uso interno do sistema. O grafo gerado deve direto e acíclico.

Na etapa de inicialização, o mecanismo interno do *JAGS* ajusta os valores iniciais para os nós estocásticos do grafo (estes nós são os parâmetros do modelo) de acordo com as informações declaradas no modelo (i.e. a forma como a variável é declarada e o relacionamento do nó com seus nós parentes).

Também é preciso atribuir um gerador de número aleatórios (*random number generator*, RGN) que será utilizado pelo mecanismo de amostragem. O *JAGS* implementa quatro algoritmos geradores de números aleatórios e cada um será responsável por gerar números aleatórios baseados nos valores iniciais de cada nó.

O *JAGS* utiliza um mecanismo de amostragem para cada nó do grafo. A escolha do mecanismo é definida de acordo com a definição da variável correspondente no modelo.

### 5.4.3 Mecanismos de Amostragem

O módulo base do *JAGS* disponibiliza um conjunto de algoritmos geradores de amostras. O estado inicial de um gerador é baseado no *timestamp* do sistema e a cada repetição do processo de amostragem os valores gerados serão independentes das execuções anteriores. O mecanismo interno do *JAGS* define o algoritmo adequado para cada um dos nós na fase de inicialização do modelo com a atribuição de um objeto do tipo *Sampler* que é responsável por atualizar os valores do nó a cada execução do processo de amostragem. O controle de todos os tipos de objetos *Sampler* é feito por uma *Sampler Factory* e a atribuição do tipo adequado é feita de forma automática pelo sistema e não permite controle externo.

Além dos algoritmos de amostragem derivados da implementação do *BUGS*, o *JAGS* disponibiliza algoritmos para amostragem que usam mecanismos de atualização genéricos

baseados apenas nas definições dos nós. A Tabela 5.4 apresenta os mecanismos genéricos.

Tabela 5.4: Algoritmos amostradores aleatórios genéricos disponíveis no JAGS

| Amostrador | Tipo de Nó |
|---|---|
| Finite Sampler | Nós com valores discretos com no máximo 20 valores possíveis |
| Real Slice Sampler | Nós estocásticos com valores reais |
| Discrete Slice Sampler | Nós estocásticos escalares com valores discretos |

### 5.4.4 Período de Adaptação e Monitores

O período de adaptação compreende o início do processo de amostragem aleatória. Nessa fase as amostras geradas não são consideradas no resultado final gerado pelo *JAGS*. É no período de adaptação que o sistema faz testes com relação à geração de valores para os nós e verifica as melhores configurações para a execução do processo de amostragem. O período de descarte (*burn-in*) ocorre no intervalo entre a inicialização do modelo e a criação dos monitores.

No *JAGS* os monitores são objetos que armazenam os valores das amostras gerados por cada nó do grafo em cada iteração do processo de amostragem. É preciso especificar quais nós devem ser monitoradas e também é possível utilizar monitores especiais disponibilizados pelo sistema tais como o monitor de valores de desvio padrão (i.e. a soma dos desvios de todas os nós estocásticas observados na iteração), e o monitor que estima o número efetivo de parâmetros do modelo.

## 5.5 Análise do Modelo Implementado

Continuando a análise do funcionamento do mecanismo implementado pelo *JAGS*, esta seção analisa o modelo criado para o sistema de classificação supervisionada de textos desenvolvido para o estudo de caso do Capítulo 5.

Para fazer amostragem aleatória a partir de um modelo, o *JAGS* executa as etapas definidas na seção anterior através de comandos que podem ser especificados em um arquivo de *script*, como demonstrado no Código Fonte 5.13.

Código Fonte 5.13: Arquivo de script para execução do modelo *JAGS*

```
1  model in "modelo_definicao.jags"
2  data in "modelo_dados.R"
3  compile, nchains(2)
4   initialize
5  update 1000
6  monitor theta
7  monitor phi
8  monitor label
9  samplers to "modelo_samplers.txt"
10 update 10000
11 coda *
```

### 5.5.1 Fase 1: Arquivos de Modelo e Dados

O *JAGS* implementa um módulo denominado *Console* responsável por interpretar os comandos definidos no arquivo de *script* e iniciar a execução da tarefas internas implementadas pelo *JAGS*. O comando de *script MODEL IN* inicia o processo de leitura do arquivo com o modelo que descreve as distribuições probabilísticas e é definido utilizando a linguagem *BUGS*.

O primeiro passo é o de verificar o arquivo do modelo e se suas especificações estão corretas. Para essa tarefa o arquivo do modelo é enviado para um objeto *Parser* que verifica sua corretude e em seguida é feita a leitura das variáveis e das definições presentes no modelo. Todas as informações serão disponibilizadas internamente para os demais módulos do sistema. O pseudocódigo no Algoritmo 5.1 apresenta as etapas dessa fase.

**Algoritmo 5.1:** Etapa: MODEL IN.

```
1 procedure VerificarModelo(arquivo)
2 |   Analisa o modelo ;
3 |   ParserBugs(arquivo) ;
4 |   Adiciona(variáveis, dados, relacionamentos);
5 fim
```

O comando de *script DATA IN* determina o local onde estão definidos os dados que serão utilizados em conjunto com o modelo. Nessa etapa o *JAGS* apenas faz a leitura do arquivo, que deve estar formatado no padrão da linguagem *R*, e disponibiliza os dados para os demais módulos.

### 5.5.2 Fase 2: Compilando o Modelo

Na fase de compilação do modelo, o comando de *script COMPILE* inicia o processo de criação dos objetos que serão utilizados internamente pelo mecanismo de amostragem.

Inicialmente é criado um objeto do tipo *BugsModel*, representando um modelo *BUGS*, com a quantidade de cadeias que serão executadas. Cada cadeia será uma execução independente, podendo ser feita em paralelo, do mecanismo de amostragem de valores para os nós do grafo. Para os casos em que mais de uma cadeia for especificada no modelo os valores gerados em cada cadeia serão disponibilizados na etapa de pós-processamento do *JAGS*.

Em seguida o *JAGS* cria um objeto do tipo *Compiler* que recebe o objeto criado para o modelo, *BugsModel*, junto com a referência para os dados que serão utilizados. Todas as variáveis e os seus relacionamentos são adicionados no objeto do compilador. O pseudocódigo no Algoritmo 5.2 apresenta as etapas gerais para a etapa de compilação.

**Algoritmo 5.2:** Etapa: COMPILE.

```
1  procedure CompilarModelo(modelo)
2      VerificarModelo(modelo) ;
3      bugs ← BugsModel(quantidade) ;
4      compilador ← Compiler(bugs, tabela de dados)
5      para todo variável no modelo faça
6          Declarar variáveis para o compilador;
7      fim
8      para todo relacionamento no modelo faça
9          Adicionar os relacionamentos entre os nós;
10     fim
11 fim
```

Ao término do processo de compilação, uma estrutura de grafo acíclico direto está definida e é disponibilizada para o uso interno do sistema.

### 5.5.3 Fase 3: Inicialização

A fase de inicialização do *JAGS* compreende os processos de definição dos valores iniciais para o grafo e de configuração da forma como deve ser realizado o processo de amostragem de valores para os nós.

A primeira tarefa é a escolha do mecanismo de geração de números aleatórios, *RNG*, para cada cadeia definida. O *JAGS* implementa quatro tipos de algoritmos de *RNG* que podem

ser atribuídos para a cadeia e o controle da criação de objetos do tipo *RNG* é feito por uma *RNGFactory*. O *JAGS* deve inicializar os nós do grafo atribuindo valores para cada um deles. Para cada nó do grafo, a escolha do valor feita em função dos nós parentes.

Na última etapa da inicialização são escolhidos os objetos de amostragem para cada nó que será utilizado pelo mecanismo de amostragem. Esses objetos são do tipo $Sampler$ e são controlados por um objeto do tipo *SamplerFactory*. O *JAGS* implementa algoritmos genéricos de amostragem e também disponibiliza um conjunto de algoritmos de amostragem derivados das definições do *BUGS*. Para cada nó do grafo, é escolhido o objeto adequado ao tipo de amostragem que será realizado. Por exemplo, na inicialização do grafo definido na Figura 5.3 os nós $phi$ e $theta$ serão associados a objetos *Sampler* do tipo *ConjugateDirichlet*, implementados pela biblioteca *BUGS*.

O pseudocódigo no Algoritmo 5.3 mostra as etapas para o processo de inicialização do

grafo.

**Algoritmo 5.3:** Etapa: INITIALIZE.

```
1  procedure Inicializa()
2  |  EscolherRNG();
3  |  InicializaNós();
4  |  EscolheAmostradores();
5  fim
6  procedure EscolherRNG()
7  |  Cria uma lista de objetos RNGFactory;
8  |  rng ← RNGFactories();
9  |  para todo cadeias no modelo faça
10 |  |  Atribui um objeto RNG para a cadeia;
11 |  fim
12 fim
13 procedure InicializaNós()
14 |  listaNos ← NósDoGrafo();
15 |  para todo nó em listaNos faça
16 |  |  Verifica os valores dos nós parentes;
17 |  |  Inicializa o Nó;
18 |  fim
19 fim
20 procedure EscolheAmostradores()
21 |  para todo nó estocástico do grafo faça
22 |  |  Verifica a lista de objetos retornada por SamplerFactory();
23 |  |  Escolhe o melhor amostrados para o nó;
24 |  fim
25 |  Verifica se todos os nós possuem amostradores definidos
26 fim
```

### 5.5.4 Fase 4: Atribuição de Monitores

O *JAGS* disponibiliza uma forma para armazenar os valores gerados para cada nó do grafo no processo de amostragem aleatória. Os objetos são do tipo *Monitor* e é preciso especificar no arquivo de *script* quais variáveis/nós devem ser monitorados. No final da execução de todas as etapas do script o *JAGS* disponibiliza os monitores e seus valores em arquivos.

Uma $MonitorFactory$ é responsável pelo gerenciamento de objetos do tipo $Monitor$. Ao receber um comando do tipo $monitor$ o *JAGS* primeiramente verifica se o sistema está com o modo adaptativo desligado e em seguida adiciona a variável especificado para um monitor. O pseudocódigo no Código Fonte em 5.4 demonstra o processo de criação e adição

de monitores.

**Algoritmo 5.4:** Etapa: MONITORS.

```
1 procedure DefinirMonitores()
2     se algoritmo em fase de adaptação então
3         Desliga modo de adaptação;
4     fim
5     Recebe lista de monitores da MonitorFactory();
6     Adiciona o monitor na lista de monitores;
7 fim
```

### 5.5.5 Fase 6: Atualizando Valores dos Nós

O processo de atualização dos nós do grafo inicia com o comando de *script update* e deve ser informado a quantidade de iterações que amostrador deve executar na estrutura do grafo gerando amostras para cada nó estocástico. No início do processo de atualização o sistema verifica se o objeto do modelo está definido e inicializado de forma correta e também verifica se os monitores para as variáveis estão definidos. Em seguida, seleciona quais os objetos $Sampler$ serão utilizados e então inicia a atualização dos valores de cada nó. Esse processo é repetido diversas vezes de acordo com o número de atualizações definidas no *script*.

O pseudocódigo no Algoritmo 5.5 exemplifica as etapas realizadas na etapa de atualização.

**Algoritmo 5.5:** Etapa: UPDATE.

```
1 procedure Atualiza()
2     Verifica se o modelo está inicializado;
3     Verifica os monitores definidos para as variáveis;
4     Ajusta os objetos Sampler que serão utilizados;
5     para todo nó do grafo faça
6         Atualiza o valor do nó de acordo com o objeto Sampler;
7     fim
8     Atualiza a lista de monitores com os valores de amostra gerados;
9 fim
```

## 5.6 Conclusões

Para comparar as implementações demonstradas é preciso analisar os valores das métricas de avaliação. As métricas definidas serão utilizadas em todos os classificadores com os

documentos de teste previamente separados dos *corpora* originais. A Tabela 5.5 apresenta os resultados das métricas de avaliação para todas as implementações construídas.

Tabela 5.5: Resultados para as implementações de classificadores Naive Bayes

| Implementação | Precisão | Recall | F-1 Score |
|---|---|---|---|
| Python | 0.8409 | 0.74 | 0.7872 |
| NLTK | 0.8858 | 0.75 | 0.8122 |
| NLTK aperfeiçoado | 0.9666 | 0.87 | 0.9157 |
| JAGS | 0.9416 | 0.92 | 0.9326 |

Dentre as implementações, a que utiliza apenas o *Python* para a construção do classificador apresenta o menor resultado para o *F1-Score* e também tem a menor taxa de acertos e de precisão. A implementação mais simples com *NLTK* apresenta um desempenho superior à implementação com *Python* em todas as métricas analisadas e o ganho com relação ao *F1-Score* é razoável.

A versão aperfeiçoada com o *NLTK*, que utiliza as palavras mais frequentes, apresenta um ganho considerável de desempenho, atingindo o segundo maior *F1-Score* e com uma taxa de acertos superior à versão simples com *NLTK*. É preciso observar que apesar de apresentar uma melhora considerável, este tipo de implementação que utiliza apenas um conjunto de palavras de acordo com sua frequência.

A implementação utilizando modelos probabilísticos com o *JAGS* apresenta o maior *F1-Score* de todas as implementações. A vantagem da abordagem está na característica de que linguagens probabilísticas como o *JAGS* fazem inferência *Bayesiana* enquanto as demais implementações realizam uma versão simplificada da proposta *Bayesiana*, a *Naive Bayes*. Outra vantagem desse tipo de implementação é que o *JAGS* realiza amostragem *MCMC* de distribuições probabilísticas e os dados das amostras geradas podem ser analisadas com métricas estatísticas tais como o cálculo de intervalos de confiança das amostras, grau de confiabilidade das amostras e análise do estado de convergência das cadeias *MCMC* geradas.

A Tabela 5.6 apresenta uma segunda comparação entre os classificadores implementados. Verifica-se que a quantidade de linhas de código necessárias para escrever um classificador típico em *Python* é grande. A vantagem do uso do *NLTK* é que as chamadas para os métodos do *framework* dispensam a implementação manual de mecanismos para tratamento e

diminuem a complexidade relacionadas à construção de classificador *Naive Bayes* e, como consequência, a sua implementação é mais compacta e direta do que na versão utilizando apenas *Python*.

Tabela 5.6: Características das implementações de classificadores Naive Bayes

| | Linhas de código | F-1 Score |
|---|---|---|
| Python | 480 | 0.7872 |
| NLTK | 92 | 0.8122 |
| NLTK aperfeiçoado | 92 | 0.9157 |
| JAGS | 13 | 0.9326 |

O código do modelo especificado no *JAGS* é compacto e é uma transcrição direta das famílias de distribuições probabilísticas. Por esse motivo, a implementação com o *JAGS* tem a menor quantidade de código escrito.

Também é importante explicitar que o modelo criado com o *JAGS* é um modelo de classificador genérico, que pode ser aplicado a uma grande variedade de categorias. As implementações utilizando linguagens determinísticas necessitam refletir as necessidades específicas do problema proposto e tornar as suas implementações mais genéricas implica em reescrita de código e adaptação de funções.

A questão da complexidade computacional requerida pelas implementações também deve ser levada em consideração. Como ponto negativo, a implementação utilizando programação probabilística exige uma quantidade maior de processamento. Para a tarefa de classificação testada, o mecanismo *JAGS* cria uma estrutura de grafo com $1.901.141$ nós para o processo de amostragem e devido ao tamanho do grafo o custo de processamento é alto e o tempo necessário para a tarefa é grande. As implementações utilizando *Python* e *NLTK* executam em um tempo menor e com custo computacional menor. Observa-se que a adoção de uma determinada implementação deve levar em consideração a necessidade de precisão do classificador e o poder computacional disponível para a execução das implementações.

Apesar de ser possível implementar mecanismos probabilísticos em linguagens determinísticas, verifica-se que para aumentar a eficiência, o desempenho, ter menor reuso de código e mais facilidade de implementação, é recomendado o uso de linguagens especificamente probabilísticas. Com os resultados apresentados conclui-se que, para tarefas de classificação

de texto, o uso de técnicas de inferência probabilísticas (auxiliadas pelo uso de linguagens probabilísticas como o *JAGS*) é uma opção viável.

# Capítulo 6

# Conclusão

A dissertação objetiva duas metas principais, (i) uma pesquisa exploratória complementada de (ii) um estudo de caso. A primeira meta, a pesquisa exploratória, apresenta a área de programação probabilística e fundamenta todos os tópicos necessários para a compreensão do estudo de caso. O estudo de caso apresenta uma implementação de classificador supervisionado de textos utilizando programação probabilística e compara com implementações tradicionais que utilizam programação determinística.

As análises apresentadas demonstram como é possível a construção de um classificador supervisionado para documentos de textos implementado utilizando o *JAGS*. Como fundamentação, os capítulos foram divididos de modo a apresentar inicialmente a análise teórica e então iniciar a pesquisa prática.

O Capítulo 1, de *Introdução*, sumariza a proposta e os objetivos almejados e também inclui a motivação, justificativa e metodologias utilizadas. Como prelúdio para o estudo de caso, o Capítulo 2 revisa os fundamentos de probabilidade e estatística que serão utilizadas nas áreas de programação probabilística e de mecanismos de amostragem. O Apêndice A complementa essa revisão com descrições das famílias de distribuições probabilísticas mais utilizadas.

Um tópico importante e frequentemente citado na literatura de programação probabilística são as redes *Bayesianas*. O Capítulo 3 apresenta as definições e propriedades da estrutura das redes *Bayesianas* como fundamento para a implementação do classificador supervisionado de textos utilizando o *JAGS*. Justificada a importância das redes *Bayesianas* então é possível analisar mecanismos para amostragem aleatória, parte importante do *JAGS*.

O Capítulo 4 apresenta as definições e propriedades dos amostradores aleatórios e seu uso na programação probabilística. Esse tipo de mecanismo de amostragem é utilizado para inferência aproximada, diminuindo a complexidade de execução dos modelos probabilísticos. Um tópico importante neste capítulo é a análise dos dos algoritmos que usam *Cadeias de Markov com Monte Carlo*.

O Capítulo 5 descreve a técnica de classificação supervisionada que é vinculada à área de aprendizado de máquina. São apresentadas quatro implementações de classificadores, uma primeira implementação que utiliza a linguagem *Python* de forma pura (sem o uso de bibliotecas externas) e duas implementações que utilizam o *NLTK*, um *framework Python* utilizado para processamento de linguagem natural. A última implementação utiliza programação probabilística com o *JAGS*.

Com os resultados obtidos, observou-se que, dentre as implementações, a versão com *JAGS* apresenta o melhor *F1-Score*, medida principal escolhida para avaliar o desempenho dos classificadores. A conclusão do estudo comparativo é que a versão utilizando programação probabilística tem uma melhor qualidade de classificação quando comparado com as implementações com programação determinística.

É preciso enfatizar que o desempenho superior do *JAGS* para a tarefa de classificação não está relacionado apenas ao seu uso; é decorrente do uso de um modelo de classificação mais sofisticado do que as demais implementações. Enquanto as implementações feitas com *Python* e *NLTK* são baseadas no conceito de *Naive Bayes*, a versão com *JAGS* utiliza conceitos e modelos mais sofisticados (tais como a amostragem aleatória *MCMC*). Dessa forma, o melhor desempenho do *JAGS* não está diretamente relacionado ao uso da linguagem em si, mas está relacionado à forma como o modelo de classificação é definido. O *JAGS* facilita essa implementação mais sofisticada e o resultado superior pode ser obtido com o uso de outras linguagens probabilísticas que adotem os mesmos conceitos para tarefas de classificação. Também é observado que a implementação com o *JAGS* é mais compacta do que as demais implementações, facilitando o desenvolvimento de modelos probabilísticos.

As principais vantagens observadas na implementação com o *JAGS* são as que seguem.

- Facilidade para descrever modelos probabilísticos: os modelos são compactados e a quantidade de código escrito é menor.

- Além de compacto, o código é escrito em alto nível, sem a necessidade de implementação de algoritmos relacionados às funções probabilísticas. O *JAGS*, e as demais linguagens probabilísticas citadas, já possuem a implementação de mecanismos necessários para tarefas que envolvem o uso de probabilidades.

- Uma característica importante nas linguagens probabilísticas é que o código para o modelo é uma transcrição quase literal das funções probabilísticas e facilita a etapa de implementação de código quando se tem um conhecimento prévio em probabilidade e estatística.

- Os resultados obtidos no classificador foram mais precisos em todas as métricas de avaliação utilizadas.

O *JAGS*, em específico, apresentou algumas desvantagens quando comparadas com as implementações em *Python* e *NLTK*. Os principais pontos são apresentados a seguir.

- Custo computacional elevado: o modelo probabilístico para o classificador de texto exige um tempo maior de processamento do que as outras implementações.

- Muitas linguagens probabilísticas, incluindo o *JAGS*, estão em constante evolução em suas implementações. Por ser uma área recente, o uso de linguagens probabilísticas ainda demanda cuidados na escolha da melhor opção de linguagem. Em certos casos (como verificado com a linguagem *Stan*) algumas linguagens probabilísticas podem não ser capazes de executar de forma correta o modelo definido para a tarefa de classificação supervisionada.

- Uma das metas das linguagens probabilísticas é a de oferecer as facilidades encontradas nas linguagens de programação determinísticas tradicionais. O estado atual das linguagens probabilísticas estudadas ainda não apresentam tantas dessas facilidades requeridas, então existe uma clara diferença no grau de dificuldade de uso entre opções de linguagens probabilísticas disponíveis.

Concluindo, é esperado que as discussões propostas nesta dissertação tenham um nível de relevância suficiente para contribuir positivamente para a área de programação probabilística. Ainda em estado inicial de pesquisa na comunidade acadêmica a área apresenta um potencial

para novas pesquisas e é uma alternativa viável para novas abordagens área de aprendizado de máquina.

## 6.1 Trabalhos Futuros

Alguns pontos ficam em aberto após a conclusão do estudo apresentado. Os seguintes tópicos são apresentados como trabalhos futuros que podem ser desenvolvidos tomando como ponto de partida esta dissertação.

Pontos relacionados à tarefa de classificação supervisionada:

- Testar o mecanismo de classificação utilizando *JAGS* em outros tipos de documentos de texto.
- Implementar o modelo de classificação supervisionado em outras linguagens probabilísticas.
- As implementações em *Python* e *NLTK* utilizam o conceito de *Naive Bayes*. Uma pesquisa futura relevante pode ser a implementação de uma versão *bayesiana* em *Python*.

Dois pontos principais relacionados à área de programação probabilística em geral:

- Analisar a implementação de outros problemas na área de aprendizado de máquina que possam ser beneficiados com o uso de programação probabilística.
- Verificar novas funcionalidades que podem ser adicionadas nas implementações das linguagens probabilísticas. Um dos objetivos das pesquisas na área é a de tentar implementar o maior número de funcionalidades encontradas em linguagens determinísticas para facilitar a adoção das linguagens probabilísticas.

# Bibliografia

[Banko e Brill 2001]BANKO, M.; BRILL, E. *Scaling to Very Very Large Corpora for Natural Language Disambiguation*. 2001.

[Barber 2012]BARBER, D. *Bayesian reasoning and machine learning*. [S.l.]: Cambridge University Press, 2012.

[Barber 2012]BARBER, D. *Bayesian Reasoning and Machine Learning*. [S.l.]: Cambridge University Press, 2012.

[Bird, Klein e Loper 2009]BIRD, S.; KLEIN, E.; LOPER, E. *Natural language processing with Python*. [S.l.]: O'reilly, 2009.

[Bird, Klein e Loper 2009]BIRD, S.; KLEIN, E.; LOPER, E. *Natural Language Processing with Python*. [S.l.]: O'Reilly Media, 2009.

[Bishop 2007]BISHOP, C. M. *Pattern Recognition and Machine Learning*. [S.l.]: Springer, 2007.

[Bishop et al. 2006]BISHOP, C. M. et al. *Pattern recognition and machine learning*. [S.l.]: springer New York, 2006.

[Brooks et al. 2011]BROOKS, S. et al. *Handbook of Markov Chain Monte Carlo*. [S.l.]: Taylor & Francis US, 2011.

[Brooks et al. 2011]BROOKS, S. et al. (Ed.). *Handbook of Markov Chain Monte Carlo*. [S.l.]: Chapman and Hall/CRC, 2011.

[BUGS 2013]BUGS. 2013. Disponível em: <http://www.mrc-bsu.cam.ac.uk/bugs/welcome.shtml>.

[Buntine 1994]BUNTINE, W. L. Operations for learning with graphical models. *arXiv preprint cs/9412102*, 1994.

[Casella e Berger 1990]CASELLA, G.; BERGER, R. L. *Statistical inference*. [S.l.]: Duxbury Press Belmont, CA, 1990.

[Casella e George 1992]CASELLA, G.; GEORGE, E. Explaining the gibbs sampler. *The American Statistician*, Taylor & Francis, v. 46, n. 3, p. 167–174, 1992.

[Chib e Greenberg 1995]CHIB, S.; GREENBERG, E. Understanding the metropolis-hastings algorithm. *The American Statistician*, Taylor & Francis Group, v. 49, n. 4, p. 327–335, 1995.

[Church MIT 2013]CHURCH MIT. 2013. Disponível em: <http://projects.csail.mit.edu/church/wiki/Church>.

[coda: Output analysis and diagnostics for MCMC 2012]CODA: Output analysis and diagnostics for MCMC. 2012. Disponível em: <http://cran.r-project.org/web/packages/coda/index.html>.

[Crevier 1993]CREVIER, D. *AI: The tumultuous history of the search for artificial intelligence*. [S.l.]: Basic Books, Inc., 1993.

[DARPA Information Innovation Office 2013]DARPA Information Innovation Office. *Probabilistic Programming for Advancing Machine Learning*. 2013. Disponível em: <http://www.darpa.mil>.

[Darwiche 2009]DARWICHE, A. *Modeling and reasoning with Bayesian networks*. [S.l.]: Cambridge University Press, 2009.

[Domingos e Pazzani 1997]DOMINGOS, P.; PAZZANI, M. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, v. 29, n. 2-3, p. 103–130, 1997.

[Doucet et al. 2001]DOUCET, A. et al. *Sequential Monte Carlo methods in practice*. [S.l.]: Springer New York, 2001.

[Freedman, Pisani e Purves 2007]FREEDMAN, D.; PISANI, R.; PURVES, R. *Statistics*. 4th. ed. [S.l.]: WW Norton & Co, 2007.

[Gelman et al. 2003]GELMAN, A. et al. *Bayesian Data Analysis*. 2nd. ed. [S.l.]: Chapman and Hall/CRC, 2003.

[Geman e Geman 1984]GEMAN, S.; GEMAN, D. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, IEEE, n. 6, p. 721–741, 1984.

[Goodman 2013]GOODMAN, N. *A grand unified theory of AI*. 2013. Disponível em: <http://web.mit.edu/newsoffice/2010/ai-unification.html>.

[Goodman et al. 2008]GOODMAN, N. D. et al. Church: a language for generative models. In: *Proc. of Uncertainty in Artificial Intelligence*. [s.n.], 2008. Disponível em: <http://danroy.org/papers/church_GooManRoyBonTen-UAI-2008.pdf>.

[Green 1995]GREEN, P. J. Reversible jump markov chain monte carlo computation and bayesian model determination. *Biometrika*, Biometrika Trust, v. 82, n. 4, p. 711–732, 1995.

[Hastings 1970]HASTINGS, W. K. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, Biometrika Trust, v. 57, n. 1, p. 97–109, 1970.

[JAGS 2013]JAGS. 2013. Disponível em: <http://sourceforge.net/projects/mcmc-jags>.

[Joyce 2008]JOYCE, J. Bayes' theorem. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Fall 2008. [S.l.]: Stanford, 2008.

[Jurafsky e Martin 2009]JURAFSKY, D.; MARTIN, J. H. *Speech and Language Processing: An Introduction to Natural Language Processing , Computational Linguistics, and Speech Recognition*. 2nd. ed. [S.l.]: Prentice Hall, 2009.

[Kipnis e Varadhan 1986]KIPNIS, C.; VARADHAN, S. Central limit theorem for additive functionals of reversible markov processes and applications to simple exclusions. *Communications in Mathematical Physics*, Springer, v. 104, n. 1, p. 1–19, 1986.

[Leroux 1992]LEROUX, B. G. Maximum-likelihood estimation for hidden markov models. *Stochastic processes and their applications*, Elsevier, v. 40, n. 1, p. 127–143, 1992.

[Lucena, Brito e Formiga 2013]LUCENA, D.; BRITO, G.; FORMIGA, A. A probabilistic programming approach to naive bayes text classification. In: . [S.l.]: X Encontro Nacional de Inteligência Computacional e Artificial, 2013. p. 8.

[Lunn et al. 2012]LUNN, D. et al. *The BUGS Book: A Practical Introduction to Bayesian Analysis*. [S.l.]: CRC Press, 2012.

[Manning e Schütze 1999]MANNING, C.; SCHüTZE, H. *Foundations of Statistical Natural Language Processing*. [S.l.]: MIT Press, 1999.

[Mansinghka 2009]MANSINGHKA, V. *Natively Probabilistic Computation*. Tese (Doutorado) — Massachusetts Institute of Technology, 2009. Disponível em: <http://web.mit.edu/vkm/www/vkm-dissertation.pdf>.

[Metropolis et al. 1953]METROPOLIS, N. et al. Equation of state calculations by fast computing machines. *The journal of chemical physics*, v. 21, p. 1087, 1953.

[Minka et al.]MINKA, T. et al. *Infer .NET 2.4, 2010. Microsoft Research Cambridge*.

[Minka et al. 2010]MINKA, T. et al. Infer .net 2.4. microsoft research cambridge. *See http://research. microsoft. com/infernet*, 2010.

[Moon 1996]MOON, T. K. The expectation-maximization algorithm. *Signal processing magazine, IEEE*, IEEE, v. 13, n. 6, p. 47–60, 1996.

[Ntzoufras 2011]NTZOUFRAS, I. *Bayesian modeling using WinBUGS*. [S.l.]: Wiley, 2011.

[O'HAGAN e Leonard 1976]O'HAGAN, A.; LEONARD, T. Bayes estimation subject to uncertainty about parameter constraints. *Biometrika*, Biometrika Trust, v. 63, n. 1, p. 201–203, 1976.

[Open BUGS]OPEN BUGS. Disponível em: <http://mathstat.helsinki.fi/openbugs/>.

[Pang e Lee 2004]PANG, B.; LEE, L. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In: *Proceedings of the ACL*. [S.l.: s.n.], 2004.

[Pearl 2000]PEARL, J. *Causality: models, reasoning and inference*. [S.l.]: Cambridge Univ Press, 2000.

[Pfeffer 2001]PFEFFER, A. IBAL: A probabilistic rational programming language. In: *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence*. [S.l.]: Morgan Kaufmann Publ., 2001.

[Plummer 2012]PLUMMER, M. *JAGS Version 3.3.0 User Manual*. [S.l.: s.n.], 2012.

[Robert, Casella e Robert 1999]ROBERT, C. P.; CASELLA, G.; ROBERT, C. P. *Monte Carlo statistical methods*. [S.l.]: Springer New York, 1999.

[Roy]ROY, D. *Machines that learn better*. Disponível em: <http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml>.

[Russell et al. 1995]RUSSELL, S. J. et al. *Artificial intelligence: a modern approach*. [S.l.]: Prentice hall Englewood Cliffs, 1995.

[Sebastiani 2002]SEBASTIANI, F. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, ACM, v. 34, n. 1, p. 1–47, 2002.

[Segaran 2007]SEGARAN, T. *Programming collective intelligence: building smart web 2.0 applications*. [S.l.]: O'Reilly Media, 2007.

[Stan Development Team 2013]Stan Development Team. *Stan Modeling Language: User's Guide and Reference Manual, Version 1.3.0*. [S.l.: s.n.], 2013.

[WinBUGS 2013]WINBUGS. 2013. Disponível em: <http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml>.

[Witten e Frank 2005]WITTEN, I. H.; FRANK, E. *Data Mining: Practical machine learning tools and techniques*. [S.l.]: Morgan Kaufmann, 2005.

[Zhou 2012]ZHOU, Z.-H. *Ensemble Methods: Foundations and Algorithms*. [S.l.]: CRC Press, 2012.

# Apêndice A

# Famílias de Distribuições

## A.1 Famílias de Distribuições Discretas

Uma variável possui uma distribuição discreta se o conjunto de valores de X (seu espaço amostral) for contável. Para a maioria dos casos, os resultados dessas distribuições, para as variáveis variáveis aleatórias, são números inteiros.

### A.1.1 Distribuição Uniforme Discreta

Uma variável aleatória X tem *distribuição uniforme discreta (1,N)* se

$$P(X = x|N) = \frac{1}{N}, \quad x = 1, 2, \ldots, N \tag{A.1}$$

Essa distribuição atribui massa igual em cada um dos resultados $1, 2, \ldots, N$. Para o caso das *distribuições paramétricas*, a distribuição é dependente dos valores dos parâmetros. Distribuições estatísticas são utilizadas para modelar populações, normalmente é utilizada uma família de distribuições, indexada por um ou mais parâmetros.

$$EX = \sum_{x=1}^{N} x\, P(X = x|N) = \sum_{x=1}^{N} x\, \frac{1}{N} \tag{A.2}$$

$$VarX = EX^2 - (EX)^2 \tag{A.3}$$

### A.1.2 Distribuição Binomial

É baseada na ideia de uma *Prova de Bernoulli*, onde um experimente permite apenas dois, e somente dois, resultados possíveis. Uma variável tem uma distribuição de *Bernoulli(p)* se

$$X = \begin{cases} 1, \text{ com probabilidade p} \\ 0, \text{ com probabilidade 1-p} \end{cases} \qquad 0 \leq p \geq 1 \tag{A.4}$$

O valor $X = 1$ geralmente é identificado como sucesso, $p$ é o seu valor de probabilidade. O inverso se aplica para $X = 0$, e $p - 1$ é o valor da probabilidade de fracasso.

$$EX = 1p + 0(1-p) = p \tag{A.5}$$

$$VarX = (1-p)^2 p + (0+p)^2(1-p) = p(1-p) \tag{A.6}$$

Uma sequência de $n$ provas com exatamente $y$ sucessos tem probabilidade $p^y(1-p)^{n-y}$ de ocorrer. Uma vez que existem $\binom{n}{y}$ dessas sequências então

$$P(Y = y|n,p) = \binom{n}{y} p^y (1-p)^{n-y}, \quad y = 0, 1, 2, \ldots, n \tag{A.7}$$

A variável $Y$ é chamada de uma *variável aleatória binomial(n,p)*. O Teorema Binomial estabelece que para quaisquer números reais $x$ e $y$ inteiros e $n \geq 0$

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i} \tag{A.8}$$

Se a variável $X \sim$ é $binomial(n, p)$, então

$$EX = n\,p \tag{A.9}$$

$$VarX = n\,p\,(1-p) \tag{A.10}$$

### A.1.3 Distribuição de *Poisson*

É uma distribuição discreta que serve para modelagem de situações tais como quando é desejado obter o número de ocorrências em um determinado intervalo de tempo (ex., deseja-se saber o número de clientes que um atendente pode esperar em um determinado período de tempo).

Uma das suposições para a distribuição de *Poisson* é a de que para pequenos intervalos de tempo, a probabilidade de uma chegada é proporcional ao tempo de sua espera. A distribuição utiliza apenas um parâmetro, $\lambda$, denominado de parâmetro de intensidade. Uma variável aleatória $X$ tem uma distribuição de *Poisson*, se:

$$P(X = x|\lambda) = \frac{e^{-\lambda}\lambda^x}{x!} \quad \text{para x = 0,1,...} \tag{A.11}$$

Os valores para média e variância são:

$$EX = VarX = \lambda$$

O cálculo de probabilidades de *Poisson* é feita utilizando a relação de recursão:

$$P(X = x|\lambda) = \frac{\lambda}{x}P(X = x-1) \quad \text{para x =1,2,...} \tag{A.12}$$

## A.2 Distribuições Contínuas

Toda função contínua integrável, não negativa, pode ser transformada em uma função densidade de probabilidade (f.d.p.).

### A.2.1 Distribuição Uniforme

A distribuição uniforme é definida pela massa uniformemente espalhada sob um intervalo $[a, b]$. Sua *f.d.p.* é representada por

$$f(x|a,b) = \begin{cases} \dfrac{1}{b-a} & \text{se } x \in [a,b] \\ 0 & \text{caso contrário} \end{cases} \tag{A.13}$$

Dessa forma, $\int_a^b f(x)dx = 1$, e

$$EX = \int_b^a \frac{x}{b-a}dx = \frac{b+a}{2} \tag{A.14}$$

$$VarX = \int_a^b \frac{(x - \frac{b+a}{2})^2}{b-a}dx = \frac{(b-a)^2}{12} \tag{A.15}$$

### A.2.2 Distribuição Gama

A distribuição Gama é um tipo de distribuição cujos valores pertencem ao intervalo em $[0, \infty)$. A função gama, é representada por

$$\Gamma(\alpha) = \int_0^\infty t^{\alpha-1}e^{-t}dt \tag{A.16}$$

com $\Gamma(\alpha+1) = \alpha\Gamma(\alpha)$, a > 0. A média de $\Gamma(\alpha, \beta)$ é:

$$EX = \frac{1}{\Gamma(\alpha)\beta^\alpha}\int_0^\infty x \quad x^{a-1}e^{-x/\beta}dx \tag{A.17}$$

E o valor para sua variância, $VarX = \alpha\beta^2$.

### A.2.3 Distribuição Normal

Também chamada de *distribuição Gaussiana*, sua distribuição possui o formato de um sino, cuja simetria a transforma em uma escolha razoável para diversos modelos de populações de dados. É uma distribuição que pode ser utilizada para aproximar uma grande variedade de distribuições em grandes amostras. A distribuição Normal possui dois parâmetros, $\mu$ e $\sigma^2$ que são, respectivamente, sua média a sua variância. A *f.d.p.* da distribuição normal com média $\mu$ e variância $\sigma^2$ ($n(\mu, \sigma^2)$) é

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/(2\sigma^2)} \quad , -\infty < x < \infty \tag{A.18}$$

Se $X \sim N(\mu, \sigma^2)$ então a variável aleatória $Z = (X - \mu)/\sigma$ tem uma distribuição $N(0, 1)$, que é a definição da *normal padrão*:

$$P(Z \leq z) = P(\frac{X - \mu}{\sigma}) \leq z \tag{A.19}$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt \tag{A.20}$$

O valor da expectância é $EX = E(\mu + \sigma Z) = \mu + \sigma EZ = \mu$. Os dois parâmetros da distribuição normal fornecem informações sobre o formato e a localização exatos da distribuição. A *f.d.p.* tem o seu máximo em $x = \mu$ e pontos de inflexão (onde a curva se modifica de côncava para convexa) em $\mu + -\sigma$. O conteúdo de probabilidade dentro de 1, 2 ou 3 desvios padrão da média é

$$P(|X - \mu| \leq \sigma) = P(|Z| \leq 1) = 0,6826 \tag{A.21}$$

$$P(|X - \mu| \leq 2\sigma) = P(|Z| \leq 2) = 0,9544 \tag{A.22}$$

$$P(|X - \mu| \leq 3\sigma) = P(|Z| \leq 3) = 0,9974 \tag{A.23}$$

Percebe-se que 95% dos casos estão entre $\mu - 2\sigma$ e $\mu + 2\sigma$.

### A.2.4 Distribuição Beta

É uma família de distribuições contínua com valores no intervalo $(0, 1)$ e é indexada por dois parâmetros. A *f.d.p.* $beta(\alpha, \beta)$ é

$$f(x|\alpha, \beta) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1}(1-x)^{\beta-1}, \quad 0 < x < 1, a > 0, \beta > 0 \tag{A.24}$$

onde $B(\alpha, \beta)$ denota a função beta,

$$B(\alpha, \beta) = \int_0^1 x^{\alpha-1}(1-x)^{\beta-1} dx \tag{A.25}$$

A média e a variância de $B(\alpha, \beta)$ são

$$EX = \frac{\alpha}{\alpha + \beta} \tag{A.26}$$

$$VarX = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)} \tag{A.27}$$

# Apêndice B

# Artigo: A Probabilistic Programming Approach to Naive Bayes Text Classification

O seguinte artigo foi submetido e aprovado para publicação no *ENIAC 2013 – X National Meeting on Artificial and Computational Intelligence* 2013[1].

- Title: "A Probabilistic Programming Approach to Naive Bayes Text Classification"
- Authors: Danilo Lucena, Gustavo Brito and Andrei Formiga (rygaweb@gmail.com, gustavobs.katel@gmail.com, andrei@ci.ufpb.br)

## B.1 Abstract

Naive Bayes classification is a simple but effective technique often used for text classification. Although simple, the structure of the probabilistic model for Naive Bayes classification is obscured when implemented in a deterministic programming language. Probabilistic programming is proposed as a way to define more general tools that allow the expression of probabilistic models at a high level, independent of the implementation details necessary for inference. In this paper we show a formulation for Naive Bayes classification that is appropriate for implementation in probabilistic programming languages, along with an implemen-

[1]Site do evento: `http://www2.unifor.br/bracis2013/`

tation in the probabilistic programming JAGS for a text classification task. The results from our experiments with this implementation show that the probabilistic program is much shorter and concisely expresses the underlying probabilistic model, while also obtaining good classification performance.

## B.2 Introduction

Naive Bayes classification is a technique based on the idea that the values of all the features of a given object are independent given its class [Barber 2012]. The simplifying assumption of conditional independence of features is often wrong in practice, but classifiers based on it are simple to build and often perform well, especially when there is enough data [Banko e Brill 2001]. Naive Bayes models are optimal when the features are indeed conditionally independent given the class, but Naive Bayes classifiers can be optimal even when there is no such independence [Domingos e Pazzani 1997]. Naive Bayes classification is often used for text classification [Jurafsky e Martin 2009], for example *spam filtering*, i.e. the classification of email messages as spam or not spam.

*Probabilistic programming* is the idea of using programming languages and tools that can express probabilistic models at a high level, and can provide automated inference on these models [Mansinghka 2009]. Users of probabilistic programming languages can concentrate on the specification of the high-level probabilistic model desired, and get inference on this model "for free". While the use of probabilistic models in artificial intelligence and related fields has greatly increased in the last decades, the tools for implementing such systems have not improved considerably. When users want to implement probabilistic systems using standard deterministic programming tools, they have to write highly-intricate code for probabilistic inference in each case, often repeating work already done by other users. Probabilistic programming languages are intended as a *lingua franca* for working with probabilistic models, which would allow researchers to concentrate on the high-level ideas instead of the implementation details [Goodman et al. 2008]. A recent project proposed by DARPA highlights the strategic importance of probabilistic programming for innovation in machine learning and data analysis [DARPA Information Innovation Office 2013].

In this paper we present a formulation and an implementation of Naive Bayes text classi-

fication using a probabilistic programming language. Although Naive Bayes is not difficult to implement in deterministic programming languages, a probabilistic program allows for the direct expression of the underlying model of classification, resulting in a clearer and smaller program. The probabilistic program is also able to employ the capabilities of more sophisticated inference engines than what is usually possible when writing the program in deterministic languages; for example, it is easy to get not only direct classification results but also confidence intervals and other statistical analysis measurements that can be used to assess the quality of the trained classifier and the adequacy of the training data. Also, while Naive Bayes is a simple model to understand, when used for classification of text it involves a great number of parameters, which means it is an easy case to test the limits of current probabilistic programming tools.

This paper is organized as follows: Section B.3 describes the Naive Bayes model for classification, as it is generally presented. Section B.4 presents the main ideas behind probabilistic programming, as well as the more important probabilistic programming languages in current use. However, the Naive Bayes model as presented in Section B.3 is not appropriate for direct implementation in current probabilistic programming languages, so in Section B.5 we present an alternative formulation of Naive Bayes. This formulation was implemented in the probabilistic programming system JAGS, as shown in Section B.6, which details the experiments performed with this implementation and their results. Finally, Section B.7 presents the conclusions of this work and the avenues for further investigation that it opens up.

## B.3 Naive Bayes Classification

Naive Bayes is based on using Bayes' rule for classification, assuming features in an object are independent given its class. To classify an object, we want to know what is the probability that the class $C$ of the object is $c_i$, given that its feature values $F$ are $f$:

$$p(C = c_i|F = f) = \frac{p(F = f|C = c_i)p(C = c_i)}{p(F = f)}.$$

The problem is that if features interact, the quantity $p(F = f|C = c_i)$ may be difficult to calculate because the joint probability density $p(F|C)$ will be a complex function of the feature values. The simplifying assumption that the features are conditionally independent given the class means that the conditional density $p(F|C)$ can be factored as a product of the conditional distributions of individual features given the class:

$$p(F|C) = \prod_{i=1}^{N} p(f_i|C).$$

For text classification, assuming the "bag of words" representation for text documents [Jurafsky e Martin 2009], each probability $p(f_i|C = c_i)$ is the probability that the word which corresponds to feature $f_i$ is present in a document of the class $c_i$. This can be easily estimated by maximum likelihood just by counting the occurrences of the word in documents of class $c_i$ from a training corpus. Due the deleterious effect of words with count zero in some classes, it is customary to use Laplace smoothing for obtaining the probability values instead of maximum likelihood. This is an instance of the tendence for overfitting that occurs with maximum likelihood estimation [Bishop 2007]. An alternative to using Laplace smoothing to avoid overfitting is estimating the parameters using a Bayesian formulation [Barber 2012]; this is a useful route for the probabilistic programming formulation and will be detailed in Section B.5. More details about Naive Bayes classification for text can be found in many recent Natural Language Processing textbooks [Jurafsky e Martin 2009; Manning e Schütze 1999].

## B.4 Probabilistic Programming Languages

Probabilistic programming languages have been developed by statistics and computer science researchers as a way to solve in a more general way the problems faced by probabilistic systems.

The statistical languages tend to follow the ideas pioneered in the BUGS language [Lunn et al. 2012]. BUGS stands for Bayesian inference Using Gibbs Sampling, and this makes it clear that is a tool used in bayesian statistics. BUGS generates samples drawn from a posterior probability density derived from a probabilistic model and a set of known data; using the

samples it is possible to estimate parameters of interest and assess confidence measures for the estimation. Besides BUGS, other languages in this tradition are JAGS [Plummer 2012] and Stan [Stan Development Team 2013]. In all three cases, inference is done by Markov Chain Monte Carlo techniques [Brooks et al. 2011]. Both BUGS and JAGS have been in use by the statistical community for a number of years and are quite reliable, while Stan is a more recent project.

The languages defined by computer scientists are in many cases derived from existing programming languages by the addition of probabilistic extensions. This is the case of the Church language [Goodman et al. 2008] and its predecessors like IBAL [Pfeffer 2001]. Other systems, like Infer.NET [Minka et al. 2010], are defined over an existing development platform (in this case, the .NET platform). Some of the languages use Markov Chain Monte Carlo for inference, but some others use Variational Bayes techniques, or exact inference procedures based on the literature on Probabilistic Graphical Models.

## B.5 Naive Bayes for Probabilistic Languages

As mentioned in Section B.4, most probabilistic programming languages are based on bayesian statistics. Thus, to express a probabilistic model in a way that is amenable to automated inference in a probabilistic programming language, such model must be expressed in bayesian terms. Naive Bayes classification – as in Section B.3 – can be characterized as a class of categorical probability densities defined over the features; these densities define the probability of each word appearing in each document class. It is also necessary to calculate the class priors, the *a priori* probability that a given document belongs to a certain class. The class priors are also categorical probability densities.

A bayesian model may be formulated for the Naive Bayes classifier using conjugate prior densities for both the class prior and the likelihood. The conjugate prior of the categorical probability density is the Dirichlet distribution [Gelman et al. 2003]. The parameters for the model are a vector $\theta(c_i)$, the prior probabilities of a document being from class $c_i$, and a matrix $\mathbf{\Phi}(c_i, w_i)$ in which each element is the probability of word $w_i$ appearing in a document of class $c_i$. As mentioned, these parameters have Dirichlet priors, where the Dirichlet density

is given by

$$\text{Dirichlet}(\alpha) \equiv p(\alpha) = \Gamma\left(\sum_i \alpha_i\right) \prod_j \frac{p_j^{\alpha_j - 1}}{\Gamma(\alpha_j)}$$

for a parameter $\alpha$. Considering each line of $\mathbf{\Phi}$ as a separate vector we can thus state the prior distributions for the parameters as:

$$\theta \sim \text{Dirichlet}(\alpha) \tag{B.1}$$

$$\mathbf{\Phi}(k) \sim \text{Dirichlet}(\beta), \quad k = 1, \ldots, K \tag{B.2}$$

where $\alpha$ and $\beta$ are the hyperparameters for these priors and $K$ is the number of classes.

Having parameter values for $\mathbf{\Phi}$, the probability of word $i$ appearing in document $d$ is given by a categorical distribution:

$$w_{i,d} \sim \text{Categorical}(\mathbf{\Phi}(z_d)) \tag{B.3}$$

where $z_d$ is the class label for document $d$. Similarly, given $\theta$, the probability distribution for the label of document $d$ is categorical:

$$z_d \sim \text{Categorical}(\theta) \tag{B.4}$$

The size of the vector $\theta$ is $K$, where $K$ is the number of classes, while the matrix $\mathbf{\Phi}$ has dimensions $K \times V$, where $V$ is the vocabulary size of the classifier. The total number of parameters to estimate is thus $V(K + 1)$.

## B.6 Experiments and Results

The formulation of Naive Bayes classification presented in Section B.5 was implemented in the probabilistic language JAGS [Plummer 2012]. The resulting code for classifier training is shown in Figure B.1. It can be seen immediately that the probabilistic program is quite concise and translates directly the ideas in the formulation presented in Section B.5. Note how the lines that define the distributions for `theta`, `phi`, `w` and `label` are direct representations of equations (B.1), (B.2), (B.3) and (B.4), respectively. The JAGS program

is almost a direct transcription of the probabilistic model developed in Section B.5, and this is an important upside of using probabilistic programming languages.

```
var theta[K], alpha[K], phi[K, V], beta[V], label[M], w[N];

model {
  theta ~ ddirch(alpha)

  for (k in 1:K) {
    phi[k,] ~ ddirch(beta)
  }

  for (m in 1:M) {
    label[m] ~ dcat(theta)
  }

  for (n in 1:N) {
    w[n] ~ dcat(phi[label[doc[n]],])
  }
}
```

Figura B.1: Naive Bayes classifier training in JAGS.

For comparison, we implemented Naive Bayes classification using Laplace smoothing for parameter estimation in the deterministic programming language Python, in two versions: one using just the base language and another one using NLTK, the Natural Language Toolkit [Bird, Klein e Loper 2009][2]. All three implementations were tested in a text classification task comprising sentiment analysis on a collection of movie reviews taken from a real corpus [Pang e Lee 2004]. We also tested two variations of the NLTK version, one using the same feature extraction as in the pure Python version, and another one using an optimization for feature representation that is included in NLTK.

The main points of comparison between the three versions are shown in Table B.1. The Python program (without NLTK) is the longest of the three, and it's harder to understand, even though Naive Bayes is considered a simple machine learning technique. The NLTK version is shorter because it uses the Naive Bayes implementation in the Toolkit. This implementation is more sophisticated and more complex than the one from the first program, but it resulted in better classification performance. The probabilistic program took the lon-

[2]All implementations mentioned in the experimental part of this paper are publically available at the address `https://github.com/labml/textclass`

gest time for training (not shown in Table B.1), but performed better when classifying items in the test set, as measured by the F1-score.

Tabela B.1: Comparison between probabilistic and deterministic implementations of Naive Bayes classification

| Implementation | F1-score | Lines of code |
|---|---|---|
| Python | 0.7887 | 480 |
| NLTK | 0.8123 | 92 |
| NLTK optimized | 0.9158 | 92 |
| JAGS | 0.9326 | 17 |

In summary, the probabilistic programming version in JAGS is the simpler to write and it directly expresses the underlying probabilistic model, making it easier to understand and to adjust at the modeling level. The NLTK version performed well and it was easy to write; although toolkits and libraries like NLTK provide obvious code reuse, it is important to note that it is of a different nature than the one provided by JAGS and other probabilistic programming tools; while the Naive Bayes implementation in NLTK can be reused in similar situations, there is nothing especifically about Naive Bayes in the JAGS tool. JAGS provides probabilistic inference for a great variety of probabilistic models, in many application domains, while NLTK provides specific tools for text and natural language processing. In a way this is another instance of the classic tradeoff between generality and efficiency; JAGS is much more general and declarative than NLTK, but inference using JAGS tends to take more time. It is also important to note that the classification performance in the JAGS version is not due to the fact that a probabilistic programming language was used, but to a more sophisticated probabilistic model. The Python versions used Naive Bayes with Laplace smoothing, which is similar to maximum likelihood estimation with a correction for zero counts. The use of a bayesian model in the JAGS program means the predictive posterior used for classification is averaged over the possible values for the parameters, effectively achieving an effect similar to an ensemble method [Zhou 2012]. The same idea could be implemented in Python, but this would result in a much larger and more complex program, reinforcing the advantages of using a probabilistic programming system like JAGS.

We also implemented the same probabilistic model (detailed in Section B.5) in the the probabilistic programming language Stan [Stan Development Team 2013], version 1.3.0, but

due to the high number of parameters, the Stan tool was not able to compile the probabilistic program. Recall from Section B.5 that the number of parameters is at least $V(K+1)$, where $K$ is the number of classes and $V$ is the size of the vocabulary (number of unique words in the corpus); the corpus used for sentiment analysis has two classes (positive or negative review) and more than 30 thousand unique words (out of close to a million total words). This means the probabilistic model has almost a hundred thousand parameters to estimate, an amount of parameters that can cause problems for many inference engines. Even though Naive Bayes is considered a simple classification technique, its expression as a probabilistic program creates difficulties for many of the current tools.

## B.7 Conclusion

In this paper we presented a formulation of the Naive Bayes classification procedure that makes it easy to implement in probabilistic programming systems. An implementation in the probabilistic language JAGS was shown and compared to two implementations of Naive Bayes classification in Python, one of them using a specific toolkit for NLP tasks called NLKT. The probabilistic program is concise and expresses clearly the underlying probabilistic model used in the classification procedure. The JAGS program also performed well in a realistic classification task for a corpus of movie reviews, achieving a better F1-score than the versions programmed in a deterministic language. The downsides of the probabilistic program are that the computational effort necessary to train the classifier on a realistic corpus is much larger, and it requires more statistical training, especially in bayesian statistical techniques, to be understood. However, once the user has the training to use the tool, they can formulate other models, even for widely different situations and techniques, with great ease, in contrast with the situation with standard deterministic programming systems, where chaging from one machine learning technique to another may require a completely different new program. The conclusion that can be drawn from these results is that it is very much worth it to learn and use probabilistic programming tools for machine learning related tasks in natural language processing, even if the dimensionality of the datasets may create time efficiency problems for current systems.

As future work in this line of research we intend to test other classification techniques for

text data in a probabilistic programming setting, and see how they compare with the Naive Bayes formulation shown here. Another line of inquiry is to determine ways to influence the automated inference in probabilistic programming tools such that they perform better for natural language processing tasks, and the development of new inference procedures that can overcome the difficulties of dealing with the large vector representations that are common in this domain.