Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Programação Probabilística em Python, uma API em Python para o UnBBayes

Felipe Gomes Paradas

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Marcelo Ladeira

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Programação Probabilística em Python, uma API em Python para o UnBBayes

Felipe Gomes Paradas

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Marcelo Ladeira (Orientador)
Universidade de Brasília

Prof. Dr. Marcos Fagundes Caetano
Universidade de Brasília

Dr. Flavio Barros Vidal
Universidade de Brasília

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 5 de janeiro de 2023

Dedicatória

Eu dedico esse trabalho a quem me fez ser quem sou. Obrigado por tudo, mãe!

Agradecimentos

No dia que eu descobri que havia entrado na Universidade já havia encontrado meus maiores parceiros, aqueles que estiveram do meu lado em momentos de muito estudo, risadas, estresse e suporte durante toda a minha graduação. Muito obrigado Lucas Pietra e Rafael Braz, sem vocês eu até teria conseguido, mas com certeza teria sido muito mais doloroso.

É impossível não mencionar também os amigos que fiz no processo, principalmente aqueles que estavam do meu lado durante o terror que foi a pandemia da COVID19 e a bagunça de continuar uma graduação naquele momento. Aqui vou citar apenas alguns nomes pois seria impossível listar todos, mas Gabriel Pinheiro, João Luiz, Arthur Lorenzo e Sofia Alves sou muito grato por todo o suporte e momentos que vivemos juntos.

Quero agradecer também aos meus amigos que me impediram de surtar nessa muito dolorida reta final. Giovanna Tati, Isabella Bianchi, Emanuel Santos e Luís Chaves muito obrigado por partilharem tudo isso comigo.

Agradeço também aos meus professores por manterem vivo em mim o amor pelos estudos e por terem me dado a habilidade de sonhar, sem vocês eu não saberia nem do que eu mais gosto. Ao meu professor orientador Marcelo Ladeira, por toda compreensão e por toda sabedoria que me foi passada, agradeço profundamente. E também, a todos os servidores da UnB, sou extremamente grato por terem me permitido viver toda essa maravilha que é a universidade pública de qualidade.

Por fim, agradeço aqueles que me fizeram ser quem eu sou, que me deram amor durante toda minha vida e quem eu amo tão profundamente, a minha família. A minha madrinha Ana Flávia, meu pai Claudio, meu primo-irmão João, e todos aqueles que tanto amo, muito obrigado por estarem comigo provendo suporte todo esse tempo. Riquito muito obrigado por cuidar de mim tão bem. A minha vó Graça e minha mãe Ana Carla, muito obrigado por tudo, eu não saberia colocar em palavras o quão grato sou pelo privilégio que tive de ter sido criado por duas professoras, mães e amigas tão cuidadosas, amorosas e compreensivas.

Essa conquista é de todos nós, muito obrigado pessoal, espero comemorar ao lado de todos vocês!

Resumo

A área de raciocínio automatizado pode ser dividir em 2 categorias de modelagem: modelos matemáticos baseados em gradiente e equações diferenciais e os modelos baseados em inferência bayesiana em redes probabilísticas. A Programação Probabilística [1] se propõe a ser, para a segunda categoria, o que os frameworks de diferenciação automatizada são para a primeira categoria. Dessa forma, é importante construir ferramentas de inferência automatizadas para darem suporte a aprendizado de máquina probabilístico.

O objetivo deste projeto é prover uma API em Python de programação probabilística, ou seja, prover um ferramental que simplifique a definição de modelos probabilísticos, com enfoque em redes bayesianas e diagramas de influência. A API é integrada ao UnBBayes [2] de forma a tornar mais acessível a utilização das funcionalidades nele existentes, sem necessidade de reimplementá-las, e utilizá-las para construir em Python modelos que possam ser utilizados em inferências probabilísticas.

Palavras-chave: Programação Probabilística, UnBBayes, API Python

Abstract

The subject of automated reasoning can be divided into 2 modeling categories: mathematical models based on gradient and differential equations and models based on bayesian inference in probabilistic networks. Probabilistic Programming [1] proposes to be, for the second category, what automated differentiation frameworks are for the first category. Therefore it is important to build automated inference tools to support probabilistic machine learning.

The objective of this project is to provide a probabilistic programming API in Python, that is, provide a tool that simplifies the definition of probabilistic models, focusing on bayesian networks and influence diagrams. The API is integrated into UnBBayes [2] in order to make it more accessible to use its existing features, without the need to re-implement them and to use them to build models in Python that can be used in probabilistic inferences.

Keywords: Probabilistic Programming, UnBBayes, API Python

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Um dos objetivos da área de raciocínio automatizado e inteligência artificial é propor e criar sistemas que agem racionalmente [3]. Contudo, quando aplicado sobre a realidade, limitações e incerteza das informações não permitem que o melhor resultado seja atingido, então se mostra necessário o desenvolvimento de métodos matemáticos para modelar a incerteza e a realidade.

Consequentemente, duas principais formas de se modelar a realidade foram criadas: aquelas com modelos matemáticos baseados em gradientes e equações diferenciais e aquelas com modelos baseados em inferência bayesiana e redes probabilísticas. Quando se lida com incerteza, a segunda categoria de modelos se mostra extremamente útil, dado que a inferência bayesiana permite atrelar a uma variável um grau de crença e atualizá-lo com base em novas evidências observadas.

Programação probabilística [1] se propõe a ser, para modelos baseados em inferência bayesiana e redes probabilísticas, aquilo que frameworks de diferenciação automatizada são para modelos baseados em gradientes. Assim, ela busca construir ferramentas para dar suporte para o aprendizado de máquina probabilístico.

O objetivo desse projeto é prover uma API de programação probabilística em Python, ou seja uma ferramenta que simplifique a definição de modelos probabilísticos em Python, baseados em redes bayesianas ou diagramas de influência. Para isso, alguns elementos são utilizados.

O primeiro elemento é o UnBBayes [2], um framework para raciocínio probabilístico escrito em Java, construído na Universidade de Brasília, disponibilizado sob a licença LGPL, utilizado por pesquisadores em 137 países. A API Python é integrada ao UnBBayes de forma a tornar acessível a utilização das funcionalidades nele existentes, sem necessidade de programar em Java, e utilizá-las para construir em Python modelos que possam ser utilizados em inferências probabilísticas.

Como Java e Python não possuem, de forma nativa, métodos para utilizar algoritmos implementados na outra linguagem, se faz necessário o uso do pacote Py4J com essa funcionalidade. O uso do pacote Py4J permite acessar as funcionalidades já implementadas em Java no framework UnBBayes e criar uma API de programação probabilística em Python.

O Capítulo 2 conceitua programação probabilística, redes bayesianas e diagramas de influência. No Capítulo 3 são descritos os ferramentais utilizados durante esse projeto e as funcionalidades do Py4J que permitem a extensão da API Java.

No Capítulo 4 estão descritos os métodos disponibilizados na API Python e três exemplos implementados. O Capítulo 5 apresenta conclusões e trabalhos futuros.

Capítulo 2

Programação Probabilística

Quando se tenta raciocinar em problemas reais é necessário lidar com a incerteza. Por vezes fatos são incertos e a informação disponível não é capaz de modelar o mundo real em sua totalidade [2]. A abordagem mais difundida hoje em dia é a intencional, na qual a incerteza é associada a "estados do assunto" ou subconjuntos de "mundos possíveis", com significado semântico claro. A sintaxe consiste de declarações sobre estados do assunto que refletem o conhecimento corrente sobre o mundo. A teoria de probabilidades é o representante típico da abordagem intencional.

O raciocínio probabilístico é baseado em inferências probabilísticas, ou seja, em calcular a probabilidade de um evento dado as evidências já conhecidas. O calculo é baseado na probabilidade condicional, uma medida de crença no evento dado as evidências, e no teorema de Bayes [4]. Com o intuito de facilitar o uso de modelos de inferência bayesiana em inteligência artificial, nasceu a programação probabilística visando tornar simples fazer estatística com o ferramental provido pela Ciência da Computação [1].

Ao longo do tempo, a Ciência da Computação tem buscado formas eficientes de executar e interpretar programas, dado parâmetros de entrada, para produzir alguma forma de saída. Existe um fluxograma claro para programação em Ciência da Computação e ele pode ser visto na Figura 2.1.

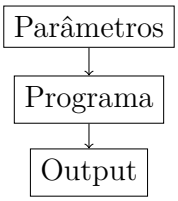

Figura 2.1: Fluxograma de programação em Ciência da Computação

Entretanto, quando se fala de estatística e inferência, o fluxograma é um completamente diferente. O primeiro passo se dá com as observações (Y), e após isso se específica um modelo generativo abstrato $p(X, Y)$, normalmente feito com notação matemática e

usando álgebra e técnicas de inferência é categorizada a distribuição $p(X|Y)$ das quantidades desconhecidas no modelo dado as quantidades observadas [1].

Essa mudança faz com que todo o fluxograma inverta, o início se dá na saída do programa. Assim, a programação probabilística é sobre fazer inferência bayesiana usando o ferramental já desenvolvido pela Ciência da Computação: sintaxe para descrição de modelos e algoritmos de inferência estatística para cálculo da probabilidade condicional das entradas do programa que podem gerar as observações da saída desse programa. O fluxograma utilizado para estatística e programação probabilística pode ser visto na Figura 2.2.

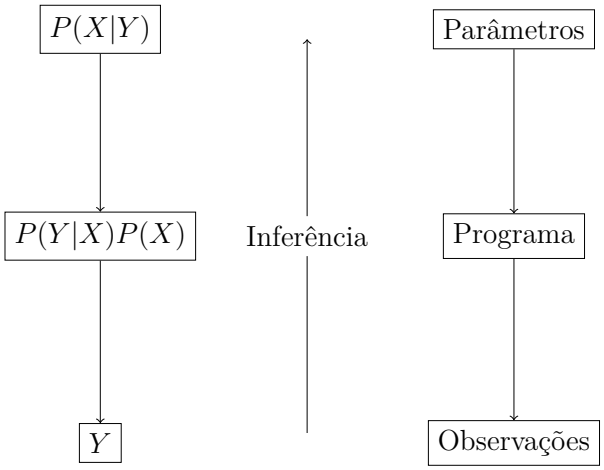

Figura 2.2: Fluxograma de programação probabilística

Programação probabilística requer o desenvolvimento de sintaxe e semântica para linguagens que denotem problemas de inferência condicional, bem como o desenvolvimento de interpretadores e "resolvedores"que resolvam computacionalmente esses modelos. Ela se encontra na intersecção de campos como aprendizado de máquina, estatística, linguagens de programação e outras ferramentas computacionais que constroem algoritmos de inferência eficientes e de fácil uso ao usuário final.

2.1 Modelos Gráficos de Probabilidade

Para modelar problemas que podem ser implementados usando programação probabilística, uma abstração bastante utilizada é a de modelos gráficos de probabilidade. O uso da inferência bayesiana permite classificar o grau de crença de uma proposição e ainda provê um meio matemático de atualizar essa crença, ou seja, do sistema aprender.

Alinhando isso com o uso de grafos para expressar as relações de dependência condicional entre as variáveis, é possível representar o conhecimento como elementos modulares

interrelacionados, [5]. Dentre os diversos tipos de modelos gráficos probabilísticos, será dado enfoque a redes bayesianas e a diagramas de influência.

2.1.1 Redes Bayesianas

Redes bayesianas (BNs) são modelos gráficos para raciocínio baseado em incerteza. Nessas redes, a representação é feita por meio de grafos acíclicos direcionados em que os nós representam variáveis probabilísticas, podendo essas serem discretas ou contínuas, e as arestas representam uma relação de probabilidade condicional entre as variáveis. Cada variável aleatória possui um conjunto de valores possíveis e em dado momento do tempo essa variável deve assumir um desses valores [5].

A distribuição de probabilidade local de uma rede bayesiana é uma função, geralmente representada por uma tabela de probabilidade condicional. Juntando o grafo e a distribuição de probabilidade local, é possível utilizar-se da inferência bayesiana para atualização da crença, ou seja, atualizar a probabilidade de uma variável assumir um possível valor.

Existem três formas com que a evidência se propaga entre as variáveis em um grafo direcionado aciclíco, e estas foram analisadas por Pearl [5]. Na primeira forma, presente na Figura 2.3, uma evidência em A influencia uma crença em B, que por sua vez influencia C, o mesmo acontece no sentido contrário, uma evidência em C também é propagada para A. Porém, caso B seja instanciada, não há propagação de evidência.

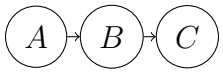

Figura 2.3: Conexão Serial

A segunda forma exemplificada na Figura 2.4 é a conexão divergente, ou seja, o grafo possui o nó A divergente, influenciando tanto B quanto C. Uma evidência em um ascendente de A influencia a crença sobre os filhos de A, porém, se A é instanciado seus filhos se tornam condicionalmente independentes.

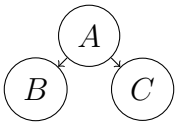

Figura 2.4: Conexão Divergente

Por fim, na Figura 2.5 chamada de conexão convergente, a evidência em A ou em um de seus descendentes influencia a crença nos pais de A. Contudo, se nada é conhecido sobre A, a evidência em seus pais não se propaga, ou seja, são independentes.

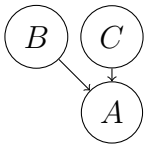

Figura 2.5: Conexão Convergente

Redes bayesianas usam desse conceito de rede, alinhado com uma tabela de probabilidade condicional e suas variáveis probabilísticas para poder raciocinar em meio a incerteza. Para além disso, é muito comum encontrar aplicações de redes bayesianas para auxílio de diagnósticos médicos em função dos sintomas esperados de um diagnóstico positivo.

2.1.2 Diagramas de Influência

Diagramas de influência (IDs) são modelos gráficos probabilísticos que podem ser vistos como uma extensão de redes bayesianas [4]. Neles, os nós não representam apenas variáveis probabilísticas, como também nós de decisão e nós de utilidade. É possível defini-los da seguinte maneira:

- Nós de probabilidade: os mesmos presentes em redes bayesianas, variáveis aleatórias associadas a uma tabela de probabilidade condicional;
- Nós de decisão: pontos de escolha de ação, seus nós pais podem ser outros nós de decisão ou nós de probabilidade;
- Nós de utilidade: representam funções de utilidade cujos argumentos são os valores de seus pais, que podem ser nós de decisão ou nós de probabilidade.

A partir desses nós, os diagramas de influência se propõe a auxiliar na tomada de decisão de acordo com resultados obtidos em seus nós de decisão e utilidade. Nós de decisão representam as possíveis ações a serem tomadas, enquanto funções de utilidade permitem raciocinar com preferências, pois cada estado que possa ser gerado ao se tomar uma decisão possui uma utilidade associada e, quanto maior a utilidade, mais preferido é esse estado.

Dessa forma, os diagramas de influência usam as informações disponíveis sobre suas variáveis aleatórias para deduzir qual das decisões é a melhor decisão lógica possível, dado a preferência do agente decisor. Suas aplicações podem ser encontradas em diversos locais, principalmente em assuntos correlatos a economia, dado que uma forma simples de modelar uma função utilidade é ao utilizar o retorno monetário esperado das ações passíveis de serem tomadas.

2.2 UnBBayes

Com isso em mente, já existe um *framework* que disponibiliza essas funcionalidades para o usuário final, o UnBBayes. O UnBBayes é um *framework* para modelos probabilísticos em inteligência artificial desenvolvido dentro da UnB [2]. Ele oferece tanto uma interface gráfica, quanto um sistema de plugins além de uma interface de programação de aplicativos (API) Java para integração com outros programas desenvolvidos. Nesse trabalho, serão exploradas as funcionalidades disponíveis na API do UnBBayes com o intuito de integrar outros programas.

Descrição do modelo de classes para representação de BNs e IDs

Conforme [2], segue uma breve descrição das classes presentes no *framework* UnBBayes.

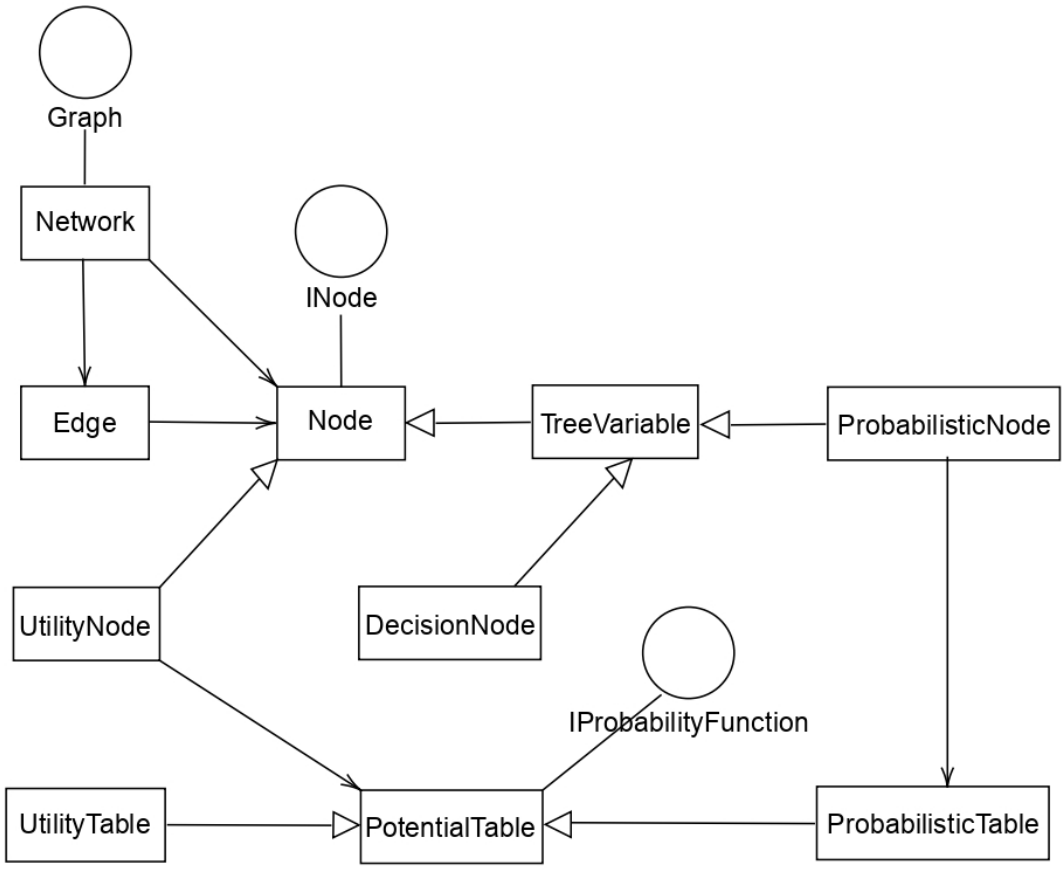

Figura 2.6: Classes presentes no UnBBayes (core)

- *Graph*: interface para grafos construídos sob um conjunto de nós e arestas.

- *Network*: implementação concreta de uma rede genérica. Se uma rede é composta por nós probabilísticos, usar ProbabilisticNetork (uma extensão de Network) seria útil.
- *Edge*: é a classe que representa uma aresta entre dois nós.Ao modelar a relação como uma classe separada se torna possível usar atributos, o que permite tratamento diferente por outras classes.
- *INode*: interface para representar um nó genérico.
- *ProbabilisticNode*: representa um nó probabilístico.
- *UtilityNode*: um nó de utilidade para IDs.
- *UtilityTable*: representa a função de utilidade para IDs (que é representado como uma tabela no UnBBayes)
- *DecisionNode*: um nó de decisão para IDs.
- *IProbabilityFunction*: uma interface para objetos que especificam a distribuição de probabilidade de um nó.
- *PotentialTable*: classe abstrata que representa *IProbabilityFunciton* em formato de tabela.
- *ProbabilisticTabel*: tabelas de probabilidade condicional para BNs.

Descrição dos métodos utilizados

A API do *framework* UnBBayes disponibiliza diversos métodos para modelos probabilísticos gráficos e é possível dividi-los em duas principais categorias:

- PRS: nesse pacote encontram-se os métodos referentes a computação de modelos probabilísticos gráficos.
- IO: nessa categoria estão os métodos referentes a leitura e escrita de arquivos externos que usam a extensão *.net*

Dessa forma, para carregar uma rede em formato *.net* na API Java do UnBBayes, basta utilizar as seguintes funções:

Código Fonte 2.1: Carregando uma rede em formato *.net*

```
1 // Abre um arquivo .net e o lê como rede probabilística
2 ProbabilisticNetwork net
3     = ( ProbabilisticNetwork ) newNetIO().load( new File( "
      ↪  ./examples/bn/net/asia.net" ) );
```

Nesse exemplo, é carregado uma rede do famoso modelo Ásia, que será descrito mais a frente na Seção 4.2 e disponibilizado em [2]. Caso o usuário queira inserir um novo nó K, com estados "sim"e "não"e sua CPT associada, basta utilizar as funções:

Código Fonte 2.2: Inserindo um novo nó na rede bayesiana

```
// Inserindo um novo nó manualmente
ProbabilisticNode newNode = new ProbabilisticNode () ;
newNode.setName("K"); // Define um nome
newNode.setDescription("A test node"); // Define uma descrição
newNode.appendState("State 0"); // Adiciona um novo estado possível
newNode.appendState("State 1");

// Buscando a função de probabilidade do novo nó
PotentialTable auxCPT = newNode.getProbabilityFunction();
auxCPT.addVariable( newNode ); // Adiciona o novo nó como uma variável
net.addNode( newNode ); // Adiciona o novo no a rede


// Preenchendo a tabela de probabilidade condicional do novo nó
auxCPT.addValueAt(0 , 0.99f); auxCPT.addValueAt(1 , 0.01f);
auxCPT.addValueAt(2 , 0.1f); auxCPT.addValueAt(3 , 0.9f);
```

E, para compilar a rede, utilizando o algoritmo de arvore de junção, usa-se:

Código Fonte 2.3: Compilando a rede

```
// Prepara o algoritimo para compilar a rede
JunctionTreeAlgorithm alg = new JunctionTreeAlgorithm () ;
alg.setNetwork( net );
alg.run(); // Roda o algoritmo
```

Com isso, o programador já é capaz de representar uma rede e compilá-la. Porém, caso ele queira atualizar evidências observadas sobre seus nós e propagá-las sobre a rede, o código é:

Código Fonte 2.4: Atualizando e propagando a evidência

```
// Insere evidência encontrada no primeiro nó da rede
ProbabilisticNode findingNode = ( ProbabilisticNode )
↪  net.getNodes().get(0);
```

```
3  findingNode.addFinding(0);

4  // Insere probabilidades
5  float likelihood[] = new float[ newNode.getStatesSize() ];
6  likelihood[0] = 1f;
7  likelihood[1] = 0.8 f;
8  newNode.addLikeliHood(likelihood);

9  // Propagando a evidência
10 net.UpdateEvidences();
```

É possível ver no código que o *framework* provê um extensivo conjunto de métodos para modelagem e execução de modelos probabilístico gráficos. Desde ferramentas para representar uma rede até algoritmos para compilar, atualizar evidências e propagar evidências.

Com isso, já se tem definida a sintaxe e semântica necessária para criação e manipulação de modelos gráficos de probabilidade em uma API Java. Dessa forma, disponibilizar a API do UnBBayes em Python tornará possível fazer programação probabilística, utilizando um software já validade e mundialmente utilizado.

Capítulo 3

Ferramental

Já é possível fazer estatística de forma simples e intuitiva com o ferramental disponibilizado pelo *framework* UnBBayes, contudo, o mesmo é implementado em Java. Java é uma linguagem de programação genérica e plataforma computacional lançada pela Sun Microsystems em 1995 que utiliza os paradigmas de programação imperativo e orientado a objeto [6]. A linguagem Java possui diversas vantagens sendo boa parte delas atribuídas ao uso da Máquina Virtual Java (JVM).

A JVM é um programa que carrega e executa os aplicativos Java, interpretando os bytecodes. Munida de diversas funcionalidades e responsabilidades como o gerenciamento de aplicativos, ela permite que o mesmo código escrito em Java rode em qualquer sistema operacional independente de hardware e software, desde que exista uma versão da JVM nele instalada.

Apesar das diversas vantagens disponibilizadas pela linguagem Java, principalmente dentro da comunidade de inteligência artificial e ciência de dados, outra linguagem vem ganhando destaque, a linguagem Python.

3.1 Python vs Java

Python é uma linguagem de programação genérica, interpretada, interativa e de simples curva de aprendizado, o que permite que estudantes e pesquisadores de outras áreas do conhecimento que não informática tenham um fácil acesso às ferramentas computacionais. Assim como Java, Python também possui suporte aos paradigmas de programação imperativo e orientado a objeto [7].

Uma das principais características que impulsionam o uso de Python, principalmente dentro da comunidade de ciência de dados, é sua interatividade, que permite executar o código "linha por linha", tornando muito simples realizar análises de dados. Apesar de Python ser mais antigo que Java, sua popularidade cresceu nos anos recentes, como é

visto nas pesquisas realizadas ao longo dos anos ([8], [9], [10], [11], [12], [13], [14]) de um dos maiores fóruns de programação do mundo, Stack Overflow.

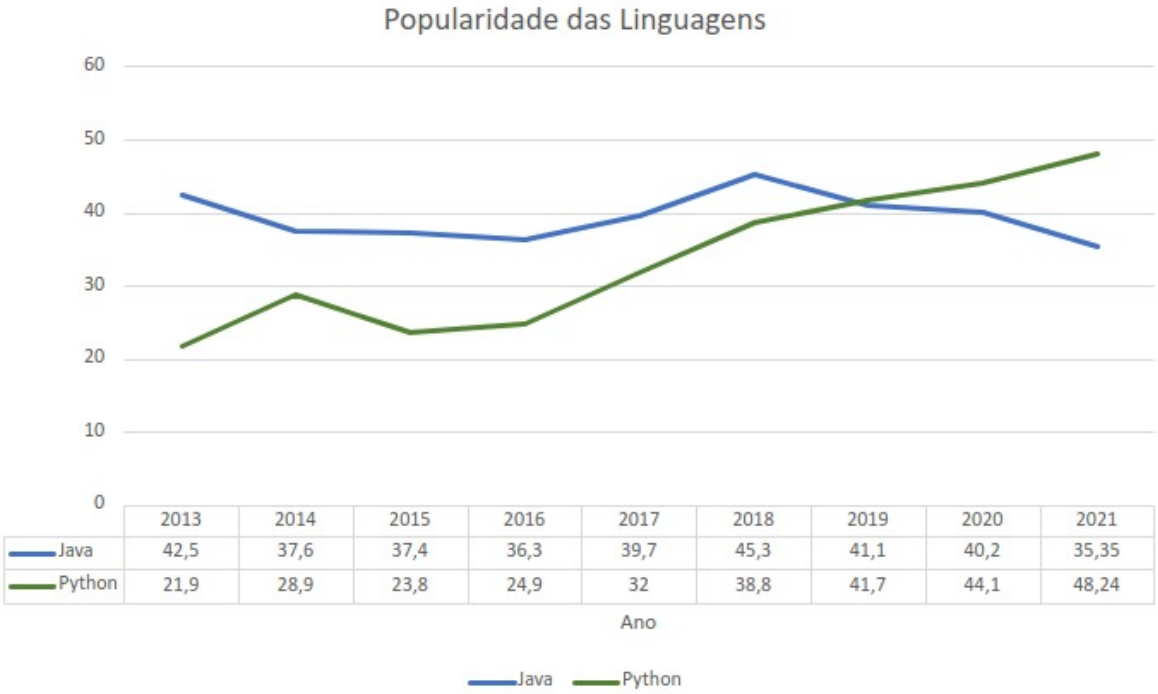

Figura 3.1: Popularidade das linguagens Java e Python ao longo do tempo

Dentre os inúmeros resultados obtidos desses questionário, a popularidade das linguagens é mensurada baseado na quantidade do total, ou porcentagem, de participantes que sabem programar nela entre todos os respondentes. Então, compilando os dados das pesquisas realizadas pelo fórum desde 2013, é possível montar o gráfico da Figura 3.1.

É possível ver na Figura 3.1 que, desde 2018, os números de Java vem decrescendo enquanto Python se mantém cada ano mais popular, superando Java em 2019. Sabendo desse fato, um dos principais objetivos desse trabalho será, por meio de uma biblioteca Python, estender as funcionalidades da API do UnBBayes em Java para a nova linguagem.

3.2 Py4J

Para a criação de uma API Python do UnBBayes será necessário transformar métodos e objetos executados na JVM em computação em Python. Contudo, a conversão de programas executados dentro da JVM para aqueles executados no interpretador Python não é direta. Não é permitido por nenhuma das duas linguagens, pelo menos de forma nativa, a interoperabilidade e integração de uma com a outra, ou seja, não é simples dentro do código Python realizar uma chamada a um algoritmo implementado na linguagem Java.

Assim, o *framework* Py4J, disponível sob a licença BSD, implementa métodos para disponibilizar essa interoperabilidade [15]. Py4J é uma ferramenta que possibilita que programas em Python sendo executados no interpretador Python acessem, de forma dinâmica, objetos Java em uma Máquina Virtual Java. A ferramenta também permite o acesso inverso, programas Java fazerem chamadas para objetos Python.

Esse pacote pode ser visto como um híbrido entre chamadas de procedimento remoto (RPC) e usar a JVM para executar um código Python. Isso é possível pois o Py4J usa conexões em *sockets* para comunicar a JVM com o interpretador Python e, para isso, é necessário expor uma porta de entrada em Java para o interpretador Python.

O Py4J é muito útil no sentido de que ele não apenas permite que Python utilize as funções nativas da linguagem Java, ele também possibilita que o programa consuma qualquer API desenvolvida em Java, desde que essa seja importada.

Código Fonte 3.1: JavaGateway

```
1 from py4j.java_gateway import JavaGateway

2 JavaGateway.launch_gateway( classpath=unbbayes_path, die_on_exit=True )
```

No código 3.1, o Py4J disponibiliza um ponto de entrada a todas as chamadas da JVM dentro de Python, o *JavaGateway*. A partir desse objeto é possível utilizar diversas funções, entre elas *launch_gateway*, uma função que permite iniciar um processo da JVM, importar um pacote Java disponível em um arquivo *.jar* no seu argumento *classpath* e especificar que esse processo deve ser destruído ao final da execução do programa Python no argumento *die_on_exit*.

A partir do uso dessa bibliotéca, é possível chamar as funções já existentes em Java como se fossem nativas Python. Entretanto, a intereção com o processo da JVM pode ser confusa para o usuário final.

Capítulo 4

API

A principal responsabilidade da API Python do UnBBayes é encapsular as interações com a JVM de forma com que o usuário não precise saber quais conexões e chamadas estão sendo feitas. Porém é necessário garantir que será mantido apenas um processo da JVM rodando para evitar problemas com performance e inconsistência dos resultados.

Para cumprir essa garantia, é criado uma classe seguindo o padrão de design *singleton*. Esse padrão especifica que apenas uma instância da classe pode existir [16], o que permite que o UnBBayes inicialize uma JVM logo na criação do objeto.

Código Fonte 4.1: Implementação da metaclasse Singleton

```
1 class Singleton(type):
2     _instances = {}
3     def __call__(cls, *args, **kwargs):
4         if cls not in cls._instances:
5             cls._instances[cls] = super(Singleton, cls).__call__(*args,
              ↪  **kwargs)
6         return cls._instances[cls]

7 class UnBBayes(metaclass=Singleton):
```

Com intuito de implementar o padrão **singleton** em Python, o recurso de metaclasses foi utilizado. Metaclasses permitem customizar o processo de criação de instâncias uma classe na linguagem [7], esse recurso permite separar a responsabilidade de manter apenas uma instância criada das outras responsabilidades presentes na API.

Para a definição da metaclasse *Singleton*, é apenas necessário um atributo, *_instances*, um dicionário que mantém a referência das instâncias criadas. Além disso, também se define o método *___call___*, que é chamado toda vez que uma nova instância das suas

subclasses são criadas. Dentro de *__call__*, se a subclasse ainda não foi instanciada, cria-se uma nova, caso contrário apenas retorna a instância que já existe.

Dessa forma, garante-se que apenas uma instância da classe UnBBayes será criada. Esse fato permite que a classe seja responsável apenas pelas interações com a API em Java do UnBBayes.

Para a definição da classe *UnBBayes* em Python, alguns requisitos precisam ser atendidos. O primeiro é, logo na criação do objeto, iniciar o processo da JVM e garantir que o mesmo seja destruído assim que o programa Python parar. Também é necessário manter a conexão com a JVM como atributo da própria classe, além dos pacotes que serão utilizados para seus métodos.

Código Fonte 4.2: Implementação do método *__init__* classe UnBBayes

```
1  class UnBBayes(metaclass=Singleton):

2      def __init__(self):

3          cwd = os.getcwd()

4          unbbayes_path = None
5          for filename in os.listdir( os.path.join(cwd, 'unbbayes', 'lib',
           ↪  'unbbayes') ):
6              if re.match( 'unbbayes.*\.jar', filename ):
7                  unbbayes_path = os.path.join( cwd, 'unbbayes', 'lib',
                   ↪  'unbbayes', filename )

8          self._gateway = JavaGateway.launch_gateway(
           ↪  classpath=unbbayes_path, die_on_exit=True )
9          self._prs = self._gateway.jvm.unbbayes.prs
10         self._io = self._gateway.jvm.unbbayes.io
```

Em Python, o método *__init__* executa após um objeto ser instanciado, porém antes de ser retornado ao usuário [7]. Dessa forma, nesse método a classe, utilizando a função disponibilizada pelo Py4J *launch_gateway*, inicia um processo da JVM com os pacotes da API Java do *UnBBayes* em background especificando no argumento *classpath* o caminho dos pacotes Java que serão utilizados, e no argumento $die_on_exit = True$, garante que ao final do programa o processo também será terminado [15].

A classe *UnBBayes* então possui três atributos:

_gateway: Objeto que guarda a conexão com a JVM;

_prs: Referência para o pacote PRS da API Java do UnBBayes;

_io: Referência para o pacote IO da API Java do UnBBayes.

Também são implementadas duas outras classes em Python para facilitar as abstrações dos usuários: *Node*, que representa nós da rede, e *Network*, que representa a rede em si. Sua implementação pode ser vista no código fonte 4.3

Código Fonte 4.3: Classes Network e Node

```
class Network:
    def __init__(self, jNet):
        self.net = jNet
        self.compiled = False

class Node:
    def __init__(self, name: str, parents: List[str], states: List[str],
    ↪ cpt: List[float]):
        self.name = name
        self.parents = parents
        self.states = states
        self.probs = cpt
```

Na classe *Node* são guardadas informações referentes a seu nome, o nome dos nós pais, os estados que essa variável pode assumir e sua tabela de probabilidade condicional nos atributos *name*, *parents*, *states* e *probs*, respectivamente. Enquanto na classe *Network*, é guardado apenas a referência a rede na JVM e um booleano indicando se a rede já foi compilada ou não nos atributos *net* e *compiled*, respectivamente.

4.1 Métodos disponíveis na API

Utilizando as classes já definidas e seguindo as ideias de programação probabilística, a API disponibiliza diversos métodos ao usuário para permitir a criação e manipulação de redes bayesianas e diagramas de influência. Esses métodos são estáticos e implementados diretamente na classe *UnBBayes* e podem ser acessados pelo seu objeto instanciado.

4.1.1 *create_java_node*

Código Fonte 4.4: *create_java_node*

```
def create_java_node(self, node: Node)
```

O primeiro método disponibilizado, ele aceita como argumento um objeto da classe *Node*, cria seu equivalente na JVM e retorna sua referência.

4.1.2 *add_node*

Código Fonte 4.5: *add_node*

```
1 def add_node(self, network: Network, node: Node)
```

Essa função aceita como argumento um nó da classe *Node* e uma rede da classe *Network*, adiciona o nó a esta rede e a retorna.

4.1.3 *create_network*

Código Fonte 4.6: *create_network*

```
1 def create_network(self, name: str, nodeList: List[Node])
```

Aceita como entrada o nome da rede e uma lista de nós, cria uma rede baseada baseada na lista de nós e a retorna.

4.1.4 *create_network_from_file*

Código Fonte 4.7: *create_network_from_file*

```
1 def create_network_from_file(self, path: str)
```

Aceita como entrada o caminho para um arquivo *.net*, cria uma rede a partir dele e a retorna.

4.1.5 *save_network*

Código Fonte 4.8: *save_network*

```
1 def save_network(self, path: str, network: Network)
```

Aceita como entrada o caminho onde a rede será salva e a rede em si. Salva a rede no formato *.net* no caminho especificado.

4.1.6 *print_network*

Código Fonte 4.9: *print_network*

```
1 def print_network(self, network: Network)
```

Imprime no terminal a rede passada como argumento.

4.1.7 *compile_network*

Código Fonte 4.10: *compile_network*

```
1 def compile_network(self, network: Network)
```

Compila a rede passada como entrada utilizando o algoritmo de *JunctionTrees*.

4.1.8 *set_evidence*

Código Fonte 4.11: *set_evidence*

```
1 def set_evidence(self, pyNet: Network, evidences)
```

A partir das novas evidências passadas como entrada, atualiza as probabilidades presentes na rede passada como entrada.

4.1.9 *propagate_evidence*

Código Fonte 4.12: *propagate_evidence*

```
1 def propagate_evidence(self, pyNet: Network)
```

Propaga as evidências que foram atualizadas na rede passada como entrada.
Utilizando esses métodos, três exemplos foram implementados.

4.2 Exemplo 1: Ásia

Asia é um famoso exemplo de aplicação de redes bayesianas. O mesmo é descrito em [2] e nele estão presentes 8 nós probabilísticos e suas relações. Esses nós são:

A: Visitou a Ásia?

S: É fumante?

T: Tem tuberculose?

L: Tem câncer de pulmão?

B: Tem bronquite?

E: Tem câncer ou tuberculose?

X: Tem raio-x positivo?

D: Tem dispneia?

Cada uma dessas variáveis podem assumir os valores "sim"e "não"e se relacionam da seguinte forma:

A -> T: visita a Ásia aumenta a chance de ter tuberculose;

S -> L e B: fumar aumenta a chance de câncer de pulmão e bronquite

T e L -> E: tabela verdade da relação "tuberculose ou câncer de pulmão"

E e B -> D: tuberculose, câncer de pulmão e bronquite aumentam a chance de dispneia

E -> X: tuberculose e câncer de pulmão aumentam a chance de raio X positivo

E possui as tabelas de probabilidade condicional da Figura 4.1.

A	sim	não
P(A)	0,01	0,99

S	sim	não
P(S)	0,5	0,5

$P(B|S)$

$B\|S$	sim	não
sim	0,6	0,3
não	0,4	0,7

$P(X|E)$

$X\|E$	sim	não
sim	0,98	0,05
não	0,02	0,95

$P(T|A)$

$T\|A$	sim	não
sim	0,05	0,01
não	0,95	0,99

$P(L|S)$

$L\|S$	sim	não
sim	0,1	0,9
não	0,01	0,99

$P(D|B,E)$

E	sim		não	
$D\|B$	sim	não	sim	não
sim	0,9	0,8	0,7	0,1
não	0,1	0,2	0,2	0,9

$P(E|T,L)$

L	sim		não	
$E\|T$	sim	não	sim	não
sim	1	0	0	0
não	0	1	1	1

Tabela 4.1: Tabelas de probabilidade

Ao combinar as relações descritas e a tabela de probabilidade condicional 4.1, dentro da interface gráfica do UnBBayes é possível montar a rede apresentada na Figura 4.1.

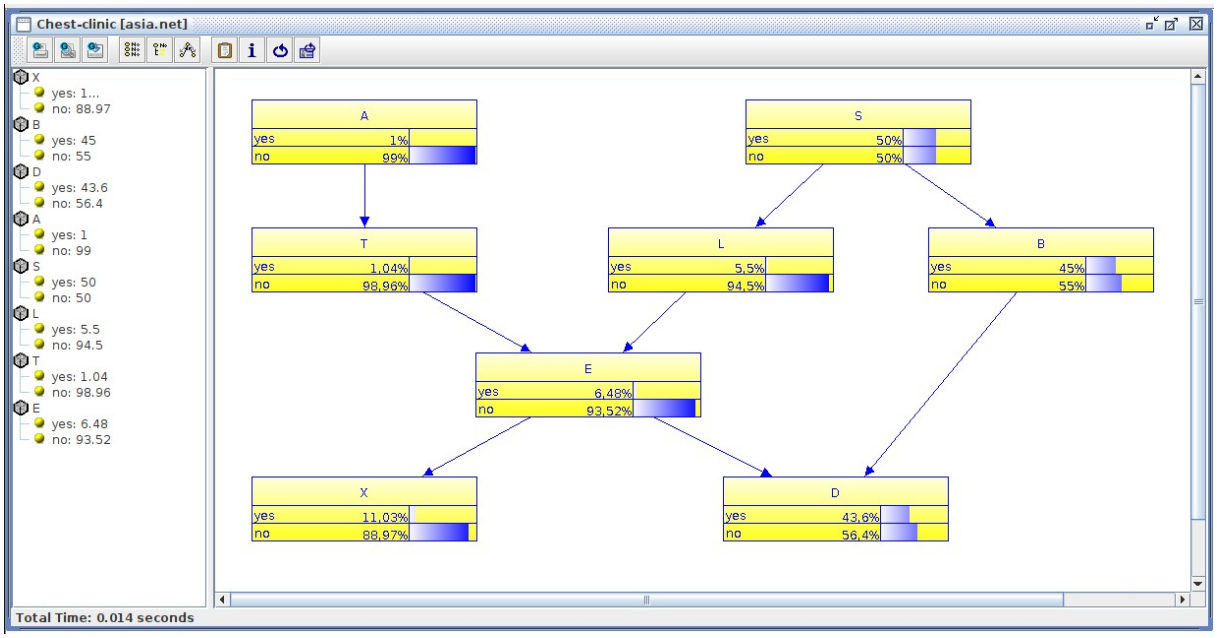

Figura 4.1: Rede bayesiana do modelo Ásia

Para criar uma rede bayesiana, utilizando a API em Python desenvolvida, é necessário especificar os nós e seus estados possíveis e, para cada nó, o nome dos nós que exercem influência direta sobre ele (seus pais) e a tabela de probabilidades condicionais do nó, dado os seus pais. O código fonte 4.13 ilustra a criação dos nós para a rede Ásia.

Código Fonte 4.13: Definição dos nós no modelo Ásia

```
1 nodeList = [
2     Node(name="asia", parents=[], states=["yes", "no"], cpt=[0.01,
      ↪ 0.99]),
3     Node(name="tub", parents=["asia"], states=["yes", "no"], cpt=[0.05,
      ↪ 0.95, 0.01, 0.99]),
4     Node(name="smoke", parents=[], states=["yes", "no"], cpt=[0.5,0.5]),
5     Node(name="lung", parents=["smoke"], states=["yes", "no"], cpt=[0.1,
      ↪ 0.9, 0.01, 0.99]),
6     Node(name="bronc", parents=["smoke"], states=["yes", "no"], cpt=[0.6,
      ↪ 0.4, 0.3, 0.7]),
7     Node(name="either", parents=["lung", "tub"], states=["yes", "no"],
      ↪ cpt=[1,0,1,0,1,0,0,1]),
```

```
8       Node(name="xray", parents=["either"], states=["yes", "no"],
        ↪  cpt=[0.98, 0.02, 0.05, 0.95]),
9       Node(name="dysp", parents=["bronc", "either"], states=["yes", "no"],
        ↪  cpt=[0.9, 0.1, 0.7, 0.3, 0.8, 0.2, 0.1, 0.9]),
10  ]
```

Após a criação os nós no código fonte 4.13, basta criar a rede e compilá-la. Para imprimir na tela o resultado, basta utilizar a função *print_network*.

Código Fonte 4.14: Compilando a rede Ásia

```
1  net = unb.create_network("Asia.net", nodeList)
2  net = unb.compile_network(net)
3  unb.print_network(net)
```

Os resultados obtidos ao se compilar a rede, com seu código fonte presente em 4.14 são:

X	sim	não
P(X)	0,1102	0,8897

S	sim	não
P(S)	0,5	0,5

B	sim	não
P(B)	0,4500	0,55

L	sim	não
P(L)	0,0550	0,945

D	sim	não
P(D)	0,4359	0,5640

T	sim	não
P(T)	0,0103	0,9896

A	sim	não
P(A)	0,0099	0,9899

E	sim	não
P(E)	0,0648	0,9351

Tabela 4.2: Probabilidade dos nós obtidos pelo PyUnBBayes

É possível notar que os valores obtidos são os mesmos valores presentes na rede criada na interface gráfica do UnBBayes vista na Figura 4.1, porém arredondados. Isso se dá devido a característica da API de apenas estender as funcionalidades já presentes no UnBBayes, sem precisar reimplementá-las.

Também é possível atualizar as evidências e propagá-las sobre a rede. No caso em que foi observado que o paciente não é fumante e tem dispneia a rede pode ser atualizada seguindo os passos demonstrados no código fonte 4.15

Código Fonte 4.15: Propagando evidências na rede Ásia

```
1  net = unb.propagate_evidence(unb.set_evidence(net, [("dysp", "yes"),
   ↪  ("smoke", "no")]))
```

E ao imprimir a rede atualizada na tela, obtém-se os seguintes resultados:

X	sim	não
P(X)	0,0949	0,9050

S	sim	não
P(S)	0,0	0,9999

B	sim	não
P(B)	0,7539	0,2460

L	sim	não
P(L)	0,0238	0,9761

D	sim	não
P(D)	1,0	0,0

T	sim	não
P(T)	0,0247	0,9752

A	sim	não
P(A)	0,0105	0,9894

E	sim	não
P(E)	0,0483	0,9516

Tabela 4.3: Probabilidade dos nós obtidos pelo PyUnBBayes após atualização de evidências

Realizando o mesmo procedimento de atualização de crença na interface gráfica do UnBBayes, consegue-se a seguinte rede:

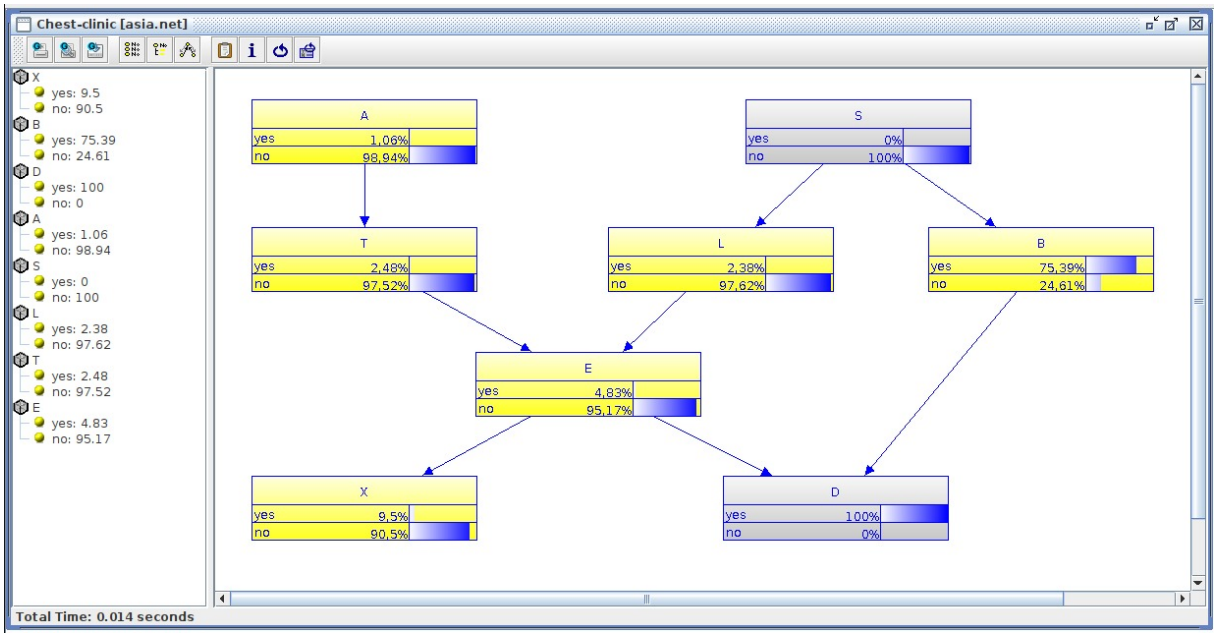

Figura 4.2: Modelo após atualização de evidências

Com isso, utilizando a API Python, é possível:

- Definir nós probabilísticos;
- Criar e compilar uma rede bayesiana;

- Atualizar as evidências baseadas em observações; e,
- Propagar as evidências pela rede.

4.3 Exemplo 2: Mildew cenário 2

Mildew é um exemplo famoso de aplicação de diagramas de influência (IDs), descrito em [4]. Dois meses antes da colheita de trigo o agricultor observa o estado da espiga para ver se está atacado pelo fungo Míldio e deve decidir sobre o tratamento do plantio. Para esse problema, as seguintes variáveis probabilísticas são mapeadas:

- *Estado da espiga*, com estados razoável, médio, bom e muito bom;
- *Observação do estado da espiga*, com estados razoável, médio, bom e muito bom;
- *Ataque atual*, com estados não, leve, moderado e severo;
- *Observação do ataque atual*, com estados não, leve, moderado e severo;
- *Danos após tratamento*, com estados não, leve, moderado e severo;
- *Estado da espiga na época de colheita*, com estados muito ruim, ruim, inferior, razoável, médio, bom e muito bom.

E as seguintes variáveis de decisão foram utilizadas:

- *Tratamento*, com as ações não, leve, moderado e forte;
- *Época de colheita*, com as ações agora, espere uma semana e espere duas semanas.

As funções de utilidade, com a mesma escala de valores, consideradas foram:

- *Custo do tratamento com fungicida*
- *Valor total da safra*

Para facilitar a notação, as seguintes letras serão atribuídas as variáveis:

Q = Estado da espiga

OQ = Observação do estado da espiga

M = Ataque atual

OM = Observação do ataque atual

M* = Danos após tratamento

H = Estado da espiga na época de colheita

A = Tratamento

T = Época de colheita

$C(A)$ = Custo do tratamento com fungicida

$U(H, T)$ = Valor total da safra

As seguintes letras serão utilizadas para representar os estados:

f = razoável

a = médio

g = bom

v = muito bom

r = muito ruim

b = ruim

p = inferior

As seguintes letras foram utilizadas para representar as ações:

no = não

l = leve

m = moderado

h = forte

now = agora

w1w = espere uma semana

w2w = espere duas semanas

As seguintes relações de influência foram mapeadas:

Q influencia tanto OQ quanto H;

OQ influencia A;

M influencia tanto M* quanto OM;

OM influencia A;

M* influencia H;

H influencia tanto T quanto $U(H, T)$;

A influencia tanto M* quanto $C(A)$;

T influencia $U(H, T)$.

A Tabela 4.4 apresenta as distribuições de probabilidade associadas.

Q	f	a	g	v
P(Q)	0,2	0,4	0,3	0,1

M	no	l	m	s
P(M)	0,4	0,3	0,2	0,1

A	no	l	m	s
C(A)	0	-2	-3	-4

$P(OQ|Q)$

$OQ\|Q$	f	a	g	v
f	0,8	0,3	0,1	0
a	0,15	0,6	0,2	0,1
g	0,05	0,1	0,6	0,4
v	0	0	0,1	0,5

$P(OM|M)$

$OM\|M$	no	l	m	s
no	0,9	0,2	0,1	0
l	0,1	0,5	0,2	0,1
m	0	0,2	0,5	0,3
s	0	0,1	0,2	0,6

Tabela 4.4: Tabelas de probabilidade Mildew

$P(M*|A,M)$

A	no				l			
$M*\|M$	no	l	m	s	no	l	m	s
no	1	0	0	0	1	0,8	0	0
l	0	1	0	0	0	0,2	0,8	0
m	0	0	1	0	0	0	0,2	0,8
s	0	0	0	1	0	0	0	0,2

$P(M*|A,M)$

A	m				h			
$M*\|M$	no	l	m	s	no	l	m	s
no	1	1	0,8	0	1	1	1	0,8
l	0	0	0,2	0,8	0	0	0	0,2
m	0	0	0	0,2	0	0	0	0
s	0	0	0	0	0	0	0	0

Tabela 4.5: Tabelas de probabilidades Mildew $P(M*|AM)$

$U(T,H)$	H	r	b	p	f
T	now	-1	1	5	8
	w1w	-1	1	5	8
	w2w	-1	1	5	8

$U(T,H)$	H	a	g	v
T	now	10	12	13
	w1w	10	12	12
	w2w	10	12	11

Tabela 4.6: Tabelas de probabilidade Mildew $U(T,H)$

$P(H|M*, Q)$

M*	no				l			
$H\|Q$	f	a	g	v	f	a	g	v
r	0	0	0	0	0,05	0	0	0
b	0,05	0	0	0	0,1	0	0	0
p	0,1	0,05	0	0	0,7	0,05	0,05	0
f	0,7	0,1	0,05	0	0,1	0,1	0,1	0,05
a	0,1	0,7	0,1	0,1	0,05	0,7	0,7	0,15
g	0,05	0,1	0,7	0,2	0	0,1	0,15	0,7
v	0	0,05	0,15	0,7	0	0,05	0	0,1

$P(H|M*, Q)$

M*	m				h			
$H\|Q$	f	a	g	v	f	a	g	v
r	0,15	0,05	0	0	0,9	0,15	0,05	0
b	0,7	0,1	0,05	0	0,1	0,7	0,1	0,05
p	0,1	0,7	0,1	0,05	0	0,1	0,7	0,1
f	0,05	0,1	0,7	0,1	0	0,05	0,1	0,7
a	0	0,05	0,1	0,15	0	0	0,05	0,1
g	0	0	0,05	0,15	0	0	0	0,05
v	0	0	0	0	0	0	0	0

Tabela 4.7: Tabelas de probabilidade Mildew $P(H|M*, Q)$

A Figura 4.3, ilustra o modelo Mildew no UnBBayes. Pela interface gráfica do UnBBayes é possível salvar essa rede como um arquivo *.net*. Esse arquivo pode ser carregado na API Python e a rede compilada, conforme ilustrado no código fonte 4.16.

Código Fonte 4.16: Carregando e compilando a rede Mildew

```
1 net = unb.create_network_from_file ("examples/mildew3-2.net")
2 net = unb.compile_network(net)
```

A Figura 4.3 apresenta a rede obtida no UnBBayes após combinar as relações descritas e as tabelas de probabilidades 4.4, 4.5, 4.6 e 4.7. A Tabela 4.8 apresenta as probabilidades após a compilação da rede Mildew.

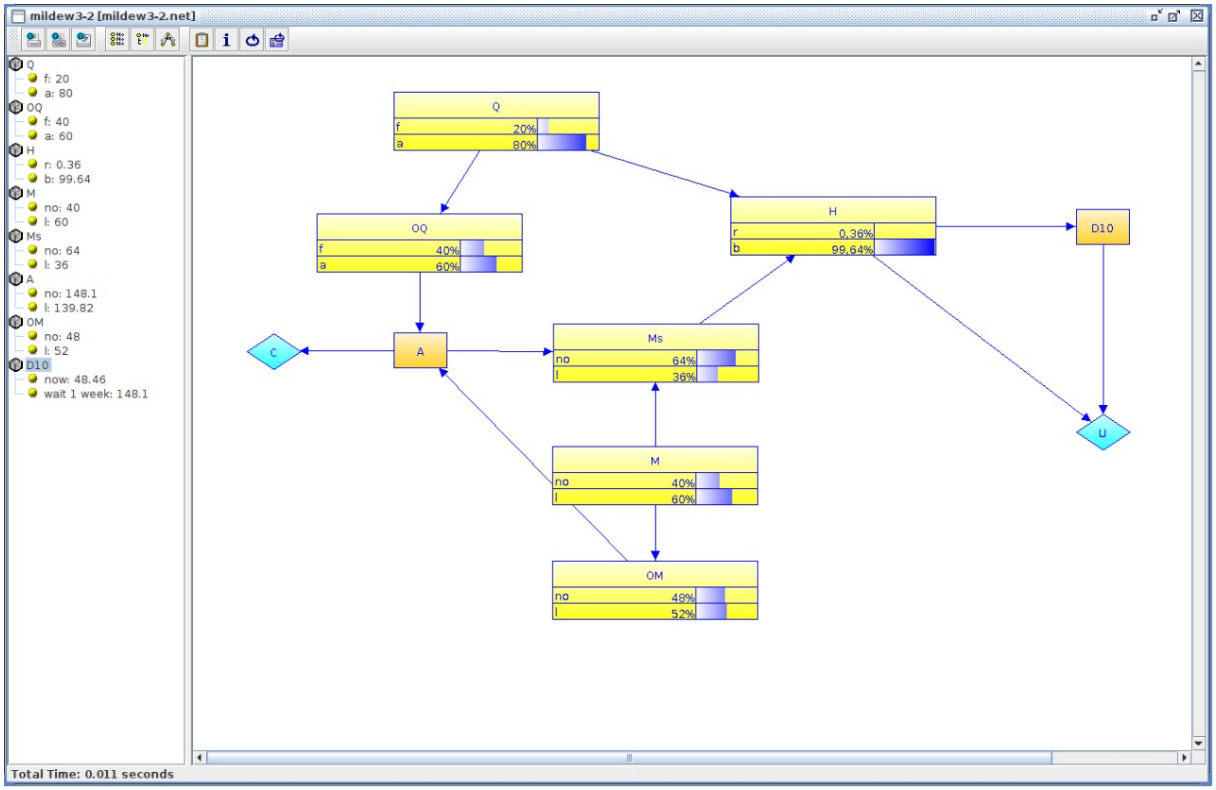

Figura 4.3: Diagrama de Influência do modelo Mildew

Q	f	a
P(Q)	0,2000	0,8

Ms	no	l
P(Ms)	0,64	0,36

OQ	f	a
P(OQ)	0,4000	0,6

A	no	l
A	148.25	139.85

H	r	b
P(H)	0,0036	0,9964

T	now	wait 1 week
T	48.4600	148.1

M	no	l
P(M)	0,3999	0,6

OM	no	l
P(OM)	0,48	0,52

Tabela 4.8: Probabilidade dos nós Mildew obtidos pelo PyUnBBayes

Com a propagação da evidência *observação da espiga como razoável* e *estado do ataque do Mildew como não ocorrido* no UnBBayes, as probabilidades são atualizadas, conforme apresentado na Figura 4.4.

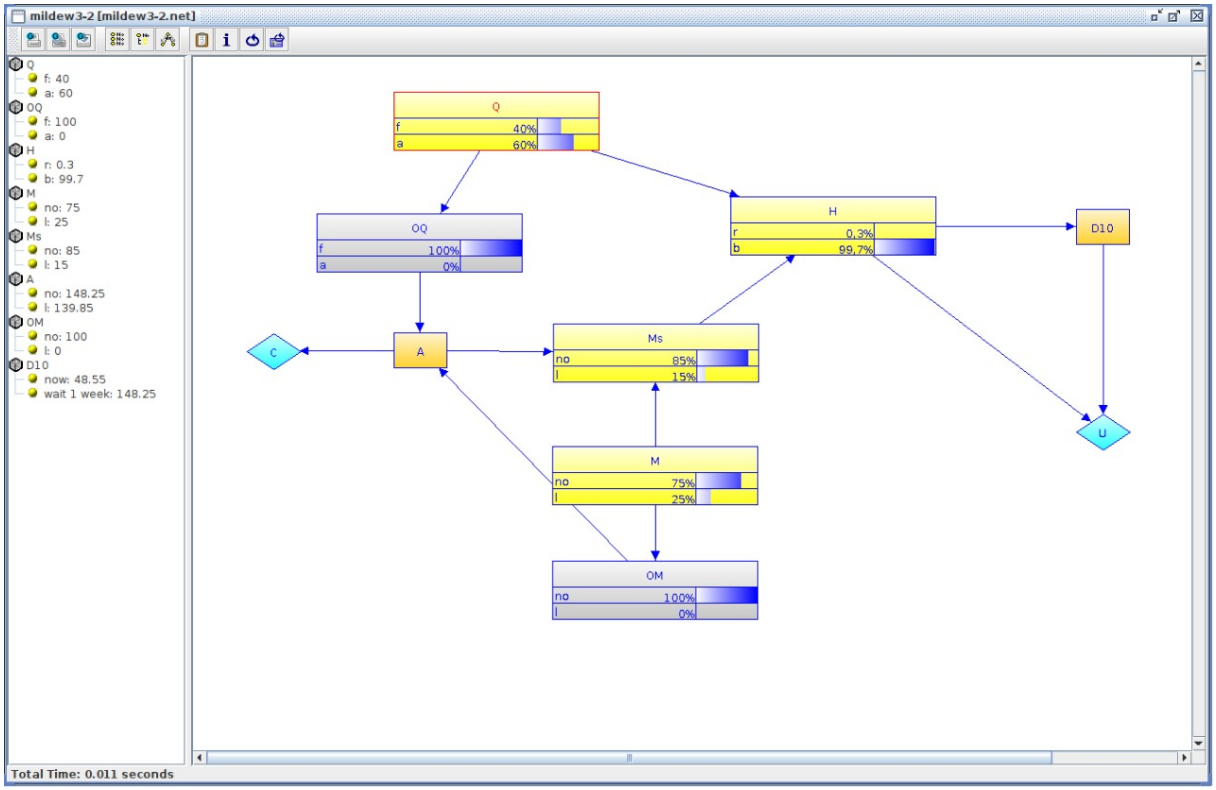

Figura 4.4: Diagrama de Influência do modelo Mildew após atualização de evidências

Os mesmos valores podem ser obtidos na API Python ao se atualizar as observações, utilizando o código fonte 4.17.

Código Fonte 4.17: Propaga evidência na rede Mildew

```
1 net = unb.propagate_evidence( unb.set_evidence(net, [("OQ", "f"), ("OM",
  ↪  "no")]) )
```

Q	f	a
P(Q)	0,4	0,6

Ms	no	l
P(Ms)	0,85	0,15

OQ	f	a
P(OQ)	1,0	0,0

A	no	l
A	148.25	139.85

H	r	b
P(H)	0,003	0,997

T	now	wait 1 week
T	48.55	148.25

M	no	l
P(M)	0,75	0,25

OM	no	l
P(OM)	1,0	0,0

Tabela 4.9: Probabilidade dos nós obtidos pelo PyUnBBayes após atualizar evidências

4.4 Exemplo 3: Diagnóstico de COVID-19

Pesquisadores da *Umm Al Qura University* desenvolveram uma rede bayesiana com 14 variáveis probabilísticas que podem assumir os valores "sim"ou "não"[17] que avalia a probabilidade de ter COVID-19, bem como a probabilidade de desenvolver os sintomas.

- Inf: Infectado?
- LW: Vivendo/Trabalhando em local com presença do coronavirus?
- COM: Se comunicou nos últimos 14 dias com algum infectado?
- Fe: Sente febre?
- Co: Sente tosse?
- SB: Sente perda de ar?
- He: Sente dor de cabeça?
- ST: Sente dor de garganta?
- Na: Sente náusea?
- Vo: Vomitou?
- Di: Teve diarreia?
- CRF: Ocorreu falha crônica renal?

- IP: Possui alguma imunossupressão?
- HF: Ocorreu falha cardíaca?

Estar infectado ou não por covid (Inf) exerce influência sobre as outras variáveis. A CPT está apresentada na Tabela 4.10.

Inf	sim	não
P(Inf)	0,01	0,99

$P(LW|Inf)$

$LW\|Inf$	sim	não
sim	0,05	0,01
não	0,95	0,99

$P(Fe|Inf)$

$Fe\|Inf$	sim	não
sim	0,01	0,01
não	0,99	0,99

$P(Co|Inf)$

$Co\|Inf$	sim	não
sim	0,01	0,01
não	0,99	0,99

$P(SB|Inf)$

$SB\|Inf$	sim	não
sim	0,01	0,05
não	0,99	0,95

$P(He|Inf)$

$He\|Inf$	sim	não
sim	0,02	0,01
não	0,98	0,99

$P(ST|Inf)$

$ST\|Inf$	sim	não
sim	0,02	0,01
não	0,98	0,99

$P(Na|Inf)$

$Na\|Inf$	sim	não
sim	0,02	0,01
não	0,98	0,99

$P(Vo|Inf)$

$Vo\|Inf$	sim	não
sim	0,02	0,01
não	0,98	0,99

$P(Di|Inf)$

$Di\|Inf$	sim	não
sim	0,02	0,01
não	0,98	0,99

$P(CRF|Inf)$

$CRF\|Inf$	sim	não
sim	0,02	0,01
não	0,98	0,99

$P(IP|Inf)$

$IP\|Inf$	sim	não
sim	0,02	0,01
não	0,98	0,99

$P(HF|Inf)$

HF | Inf	sim	não
sim	0,02	0,01
não	0,98	0,99

$P(COM|Inf)$

$COM\|Inf$	sim	não
sim	0,05	0,01
não	0,95	0,99

Tabela 4.10: Tabelas de probabilidades - COVID-19

Para definir essa rede na API Python, primeiro é necessário se definir os nós assim como feito no modelo Asia, disponível no código 4.18.

Código Fonte 4.18: Define nós da rede COVID

```
nodeList = [
    Node(name="Inf", parents=[], states=["yes", "no"], probs=[0.01,
    ↪ 0.99]),
    Node(name="LW", parents=["Inf"], states=["yes", "no"], probs=[0.05,
    ↪ 0.95, 0.01, 0.99]),
    Node(name="Fe", parents=["Inf"], states=["yes", "no"], probs=[0.01,
    ↪ 0.99, 0.01, 0.99]),
    Node(name="Co", parents=["Inf"], states=["yes", "no"], probs=[0.01,
    ↪ 0.99, 0.01, 0.99]),
    Node(name="SB", parents=["Inf"], states=["yes", "no"], probs=[0.01,
    ↪ 0.99, 0.05, 0.95]),
    Node(name="He", parents=["Inf"], states=["yes", "no"], probs=[0.02,
    ↪ 0.98, 0.01, 0.99]),
    Node(name="ST", parents=["Inf"], states=["yes", "no"], probs=[0.02,
    ↪ 0.98, 0.01, 0.99]),
    Node(name="Na", parents=["Inf"], states=["yes", "no"], probs=[0.02,
    ↪ 0.98, 0.01, 0.99]),
    Node(name="Vo", parents=["Inf"], states=["yes", "no"], probs=[0.02,
    ↪ 0.98, 0.01, 0.99]),
    Node(name="Di", parents=["Inf"], states=["yes", "no"], probs=[0.02,
    ↪ 0.98, 0.01, 0.99]),
    Node(name="CRF", parents=["Inf"], states=["yes", "no"], probs=[0.02,
    ↪ 0.98, 0.01, 0.99]),
    Node(name="IP", parents=["Inf"], states=["yes", "no"], probs=[0.02,
    ↪ 0.98, 0.01, 0.99]),
    Node(name="HF", parents=["Inf"], states=["yes", "no"], probs=[0.02,
    ↪ 0.98, 0.01, 0.99]),
    Node(name="COM", parents=["Inf"], states=["yes", "no"], probs=[0.05,
    ↪ 0.95, 0.01, 0.99]),
]
```

Após isso, basta criar a rede e compilá-la para obter os primeiros resultados.

Código Fonte 4.19: Cria e compila a rede COVID

```
net = unb.create_network("COVID19.net", nodeList)

net = unb.compile_network(net)
```

E quando tem seus valores impresso em tela, se obtém os resultados:

Inf	sim	não
P(Inf)	0,01	0,99

Na	sim	não
P(Na)	0,0101	0,9899

LW	sim	não
P(LW)	0,0104	0,9896

Vo	sim	não
P(Vo)	0,0101	0,9899

Fe	sim	não
P(Fe)	0,01	0,99

Di	sim	não
P(Di)	0,0101	0,9899

Co	sim	não
P(Co)	0,01	0,99

CRF	sim	não
P(CRF)	0,0101	0,9899

SB	sim	não
P(SB)	0,0496	0,9504

IP	sim	não
P(IP)	0,0101	0,9899

He	sim	não
P(He)	0,0101	0,9899

HF	sim	não
P(HF)	0,0101	0,9899

ST	sim	não
P(ST)	0,0101	0,9899

COM	sim	não
P(COM)	0,0104	0,9896

Tabela 4.11: Probabilidade dos nós COVID obtidos pelo PyUnBBayes

Para atualizar na API as evidências marcando que o paciente não se comunicou e nem frequentou um ambiente com um infectado porém teve febre, falta de ar e naúsea, mas não apresentou vômito é preciso executar o código 4.20.

Código Fonte 4.20: Atualiza e propaga evidência na rede COVID

```
net = unb.propagate_evidence( unb.set_evidence(net, [
    ("COM", "no"),
    ("Fe", "yes"),
    ("SB", "yes"),
```

Ao usar a função *save_network* é possível salvar a rede criada em um arquivo *.net* e esse arquivo pode ser carregado no UnBBayes. Ao fim do processo, se obtém a seguinte rede:

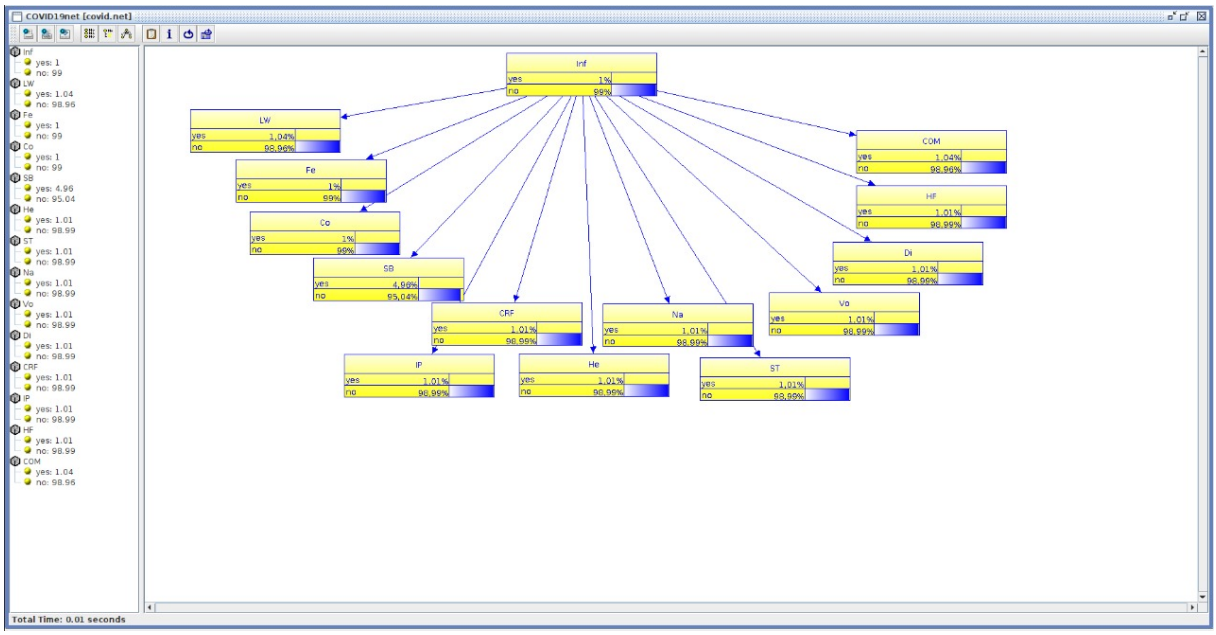

Figura 4.5: Rede Bayesiana do modelo COVID

```
5       ("LW", "no"),
6       ("Na", "yes"),
7       ("Vo", "no")
8       ]) )
```

O resultado obtido em tela ao chamar a função *print_network* é:

Inf	sim	não
P(Inf)	0,0036	0,9963

Na	sim	não
P(Na)	0,0100	0,9899

LW	sim	não
P(LW)	0,0	1,0

Vo	sim	não
P(Vo)	0,0100	0,9899

Fe	sim	não
P(Fe)	1,0	0,0

Di	sim	não
P(Di)	0,0100	0,9899

Co	sim	não
P(Co)	0,01	0,99

CRF	sim	não
P(CRF)	0,0100	0,9899

SB	sim	não
P(SB)	1,0	0,0

IP	sim	não
P(IP)	0,0100	0,9899

He	sim	não
P(He)	0,0100	0,9899

HF	sim	não
P(HF)	0,0100	0,9899

ST	sim	não
P(ST)	0,0100	0,9899

COM	sim	não
P(COM)	0,0	1,0

Tabela 4.12: Probabilidade dos nós COVID obtidos pelo PyUnBBayes após atualização de evidências

Fazendo o mesmo processo na interface gráfica do UnBBayes se obtém os mesmos resultados aproximados. Esse evento pode ser visto na seguinte rede:

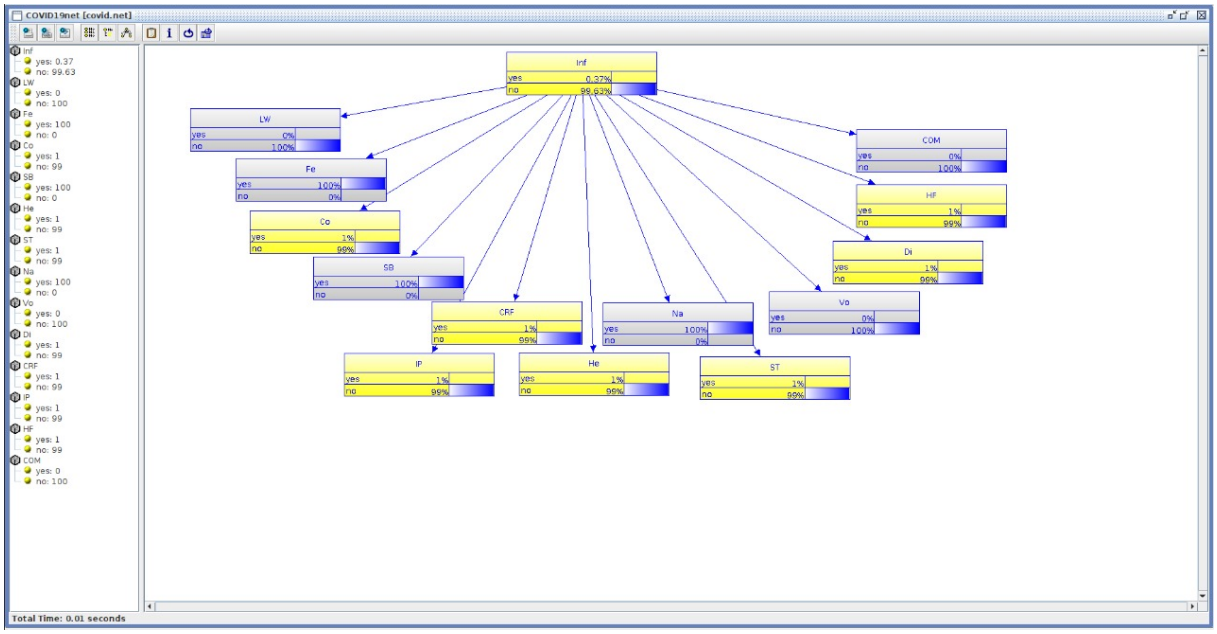

Figura 4.6: Rede Bayesiana do modelo COVID após atualização de evidências

Capítulo 5

Conclusão e Trabalhos Futuros

O estudo da inteligência artificial é um dos campos relevante dentro da Ciência da Computação e, se utilizando de modelos probabilísticos gráficos, consegue representar o conhecimento incerto e tenta modelar a realidade. Programação probabilística nasce com o objetivo de prover o ferramental, seja uma própria sintaxe ou algoritmos, para suportar fazer estatística (mais especificamente inferência bayesiana) usufruindo de métodos computacionais já existentes.

Dentro da Universidade de Brasília, o framework UnBBayes foi desenvolvido na linguagem Java para auxiliar programadores a modelar, compilar e analisar modelos gráficos de probabilidade, provendo diversos algoritmos e estruturas de dados para facilitar em todos os passos do processo. Esse framework é utilizado em todo o mundo e disponibiliza suporte para redes probabilísticas, redes probabilística multi-entidade (MEBN), dentre outros formalismos.

Entretanto, a linguagem Python vem ganhando destaque dentro da comunidade de inteligência artificial em detrimento de Java. Assim, o pacote Py4J disponibiliza diversas maneiras de estender métodos implementados na JVM para o interpretador Python. Dessa forma, esse trabalho traz as funcionalidades do UnBBayes para Python, no formato de uma API e disponibiliza diversas funções e abstrações para programação probabilística.

Para demonstrar as funcionalidades, esse trabalho implementa três exemplos de modelos gráficos probabilísticos:

- o modelo Ásia, uma rede bayesiana;
- Mildew cenário 3.2, um diagrama de influência; e
- uma rede bayesiana para diagnóstico de sintomas de COVID-19.

Eles são implementados tanto dentro da API Python, quanto na interface gráfica do UnBBayes, escrita em Java. Analisando os três exemplos implementados de forma comparativa, vê-se que os resultados obtidos para as funcionalidades estendidas de raciocínio

probabilístico, mais especificamente redes bayesianas e diagramas de influências, atingem os mesmos resultados vistos dentro da JVM em Python.

Por fim, é disponibilizado uma API com funções que podem ser chamadas em Python para construção e realização de inferências probabilísticas em redes bayesianas e em diagramas de influência. Chamando funções do framework UnBBayes em programas em Python, por meio do pacote Py4J, é possível construir e utilizar, de forma simples e em poucas linhas, modelos probabilísticos. O código e a documentação para uso estão disponíveis sob a licença GPL 2.

A API disponibilizada tem uma sintaxe familiar para o usuário que já programa em Python, o que facilita o acesso ao framework UnBBayes para usuários Python que não conheçam a linguagem Java. E, por meio do Py4J o interpretador Python chama funções do framework do UnBBayes. Devido a essa característica, a API construída pode ser facilmente estendida para permitir acesso as outras funcionalidades presentes no UnBBayes como, por exemplo, Redes Bayesianas Múltiplas Seccionadas (MSBN) e Redes Bayesianas Orientadas a objetos (OOBN).

Referências

[1] Meent, Jan Willem van de, Brooks Paige, Hongseok Yang e Frank Wood: *An introduction to probabilistic programming*. http://arxiv.org/abs/1809.10756, acesso em 2022-08-30. v, vi, 1, 3, 4

[2] Shou Matsumoto, Rommel Carvalho, Marcelo Ladeira, Paulo Costa, Laecio Santos, Danilo Silva, Michael Onishi, Emerson Machado e Ke Cai: *UnBBayes: a java framework for probabilistic models in AI*. ISSN 9781461098706. v, vi, 1, 3, 7, 9, 18

[3] Russell, Stuart J. (Stuart Jonathan) e Peter Norvig: *Inteligência artificial*. Elsevier, ISBN 9788535237016. http://bibliotecadigital.tse.jus.br/xmlui/handle/bdtse/7408, acesso em 2022-08-30. 1

[4] Ladeira, Marcelo: *Diagrama de influências múltiplo secionado*. 3, 6, 23

[5] Pearl, Judea: *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, ISBN 9780080514895. 5

[6] Gosling, James, Bill Joy, Guy L. Steele, Gilad Bracha e Alex Buckley: *The Java® language specification*. Addison-Wesley, java SE 8 edition edição, ISBN 9780133900699. OCLC: ocn873760233. 11

[7] *The python language reference — python 3.10.7 documentation*. https://docs.python.org/3/reference/, acesso em 2022-09-12. 11, 14, 15

[8] *Stack overflow developer survey 2015*. 12

[9] *Stack overflow developer survey 2016 results*. 12

[10] *Stack overflow developer survey 2017*. 12

[11] *Stack overflow developer survey 2018*. 12

[12] *Stack overflow developer survey 2019*. 12

[13] *Stack overflow developer survey 2020*. 12

[14] *Stack overflow developer survey 2021*. 12

[15] *Py4j documentation — py4j*. https://www.py4j.org/contents.html, acesso em 2022-08-29. 13, 15

[16] Gamma, Erich (editor): *Design patterns: elements of reusable object-oriented software*. Addison-Wesley. 14

[17] Emad Alsuwat, Sabah Alzahran e Hatim Alsuwat: *Detecting COVID-19 utilizing probabilistic graphical models*. 12(6). 30