PONTÍFICIA UNIVERSIDADE CATÓLICA DE SÃO PAULO
COGEAE

PÓS-GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

ADAPTAÇÃO DE ALGORÍTMO DE LÓGICA FUZZY EM UMA LINGUAGEM FUNCIONAL PARA TOMADA DE DECISÕES EM PROCESSOS DE NEGÓCIOS

MARCELO WAGNER DE SOUZA HABLOCHER

São Paulo - SP
2015

MARCELO WAGNER DE SOUZA HABLOCHER

ADAPTAÇÃO DE ALGORITMO DE LÓGICA FUZZY EM UMA LINGUAGEM FUNCIONAL PARA TOMADA DE DECISÕES EM PROCESSOS DE NEGÓCIOS

Monografia apresentada à Pontifícia Universidade Católica de São Paulo, como exigência parcial para obtenção do titulo de **Especialista em Engenharia de Software**, sob orientação do professor Daniel Gatti.

São Paulo - SP
2015

Á minha esposa e filhos, que com
infinita paciência, me fizeram chegar até aqui.

RESUMO

Este trabalho de pesquisa tem por objetivo adaptar algoritmos de tomada de decisão de negócios, criados originalmente no ano de 1994, usando técnicas de lógica Fuzzy, desenvolvidos em um ambiente de software hoje obsoleto. O foco é adaptar os mesmos modelos e algoritmos em um ambiente de software mais atual, usando o paradigma de programação funcional, através da linguagem de programação R que é ao mesmo tempo uma ferramenta de programação e um ambiente de análise estatística. O foco da pesquisa é a adaptação dos modelos originais e identificação das facilidades e/ou dificuldades encontradas durante a adaptação.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

INTRODUÇÃO

Atualmente, as linguagens de programação mais utilizadas estão centradas em duas plataformas/ambientes de programação:

- .NET
- Java

Não por acaso, as duas plataformas foram desenvolvidas em cima do conceito de máquina virtual. O código feito pelo desenvolvedor é compilado de acordo com a especificação de uma máquina virtual. Na plataforma .NET o código compilado chama-se CIL (Common Intermediate Language) e em Java o código compilado chama-se Bytecode. Em qualquer caso, o código gerado é executado pela máquina virtual, que o traduz em código de máquina nativo.

As linguagens disponíveis para estes ambientes praticamente são linguagens que seguem o paradigma de **Programação Orientada à Objetos**. Atualmente, é um paradigma bem documentado, bem conhecido pelos desenvolvedores de software e faz parte do currículo das universidades. Desde o início da década de 90, com o suporte de várias linguagens, esse paradigma se disseminou nas empresas e nas escolas de tal forma que é quase impossível um desenvolvedor de software não conhecê-lo hoje em dia.

No entanto, existem outros paradigmas de programação. São disseminados no meio acadêmico e muito pouco utilizados no meio empresarial. Mas um deles, em especial, começou a entrar em evidência: **Programação Funcional**.

Programação funcional enfatiza o uso de *funções*, ao contrario da Orientação à Objetos, que foca nos *objetos*. Como será explicado mais a frente, o uso de funções significa um alinhamento forte com o conceito de **imutabilidade** (significa que a informação, uma vez definida, não muda mais).

Algumas linguagens de programação estão incorporando alguns conceitos de programação funcional, tais como:

- Lambda Expressions em Java 8
- Lambda Expressions em C#

Até mesmo algumas linguagens foram criadas para as plataformas citadas:

- Clojure – Java
- Scala – Java
- F# – .NET

Por que Programação Funcional está em evidência? Por que um paradigma, inventado em 1930, com a criação do cálculo lambda (MICHAELSON, 2012), está se tornando importante?

Um dos principais motivos é a **mutabilidade**. Muitos erros em tempo de execução ocorrem por causa de alterações indevidas no estado de objetos/variáveis. Esse tipo de comportamento também dificulta a programação paralela, onde alterações indevidas podem causar erros quase impossíveis de serem rastreados.

A programação funcional, com seus conceitos de **imutabilidade** e **funções**, elimina alterações indevidas em objetos/variáveis e facilita muito a programação paralela.

Outro motivo é a complexidade dos algoritmos de análise de dados, que cada vez mais entram no mundo de expressões matemáticas, exigindo que o desenvolvedor crie/altere algoritmos usando modelos matemáticos. Em linguagens funcionais, é natural pensar em modelos matemáticos, porque o próprio paradigma exige esse tipo de pensamento. A vantagem é que o código criado torna-se menor evitando-se uso extensivo de classes e evita-se um efeito chamado de "código

criptográfico" (BACKFIELD, 2014), que é a complexidade que linguagens imperativas acabam criando quando é necessário programar um algoritmo mais robusto.

Esse cenário de algoritmos mais complexos leva a outras necessidades, não só de paradigmas e de linguagens de programação, mas também de métodos de solução de problemas.

Nos processos de negócios atuais, a complexidade de análise de dados sai da esfera da certeza booleana e entra no mundo da incerteza, descrita pelo nome de "**fuziness**", que o dicionário classifica como "**a qualidade de ser indistinto ou sem linhas bem definidas**".

Processos de negócios precisam analisar dados dentro de espectros que vão além da lógica booleana, e um dos melhores métodos conhecidos para se utilizar nesses cenários é a **Lógica Fuzzy**.

A **Lógica Fuzzy** sai do cenário **Sim/Não** e entra no cenário de **Não, Ligeiramente, Um pouco, Mais ou menos, Alguns, Sobretudo, Sim**. Essas opções não são bem codificadas em linguagens imperativas usando lógica booleana.

Este trabalho tem como objetivo analisar as ferramentas, linguagens e métodos existentes e demonstrar como é possível montar uma ferramenta de análise de regras de negócios utilizando uma linguagem de programação funcional e a Lógica Fuzzy para programar as regras de negócios, executar a análise e fornecer o resultado esperado. No caso, utilizarei a Linguagem de programação R, que está em evidência e que fornece um ambiente de análise estatística ideal para reproduzir modelos de regras de negócios.

O Capítulo 2 apresenta o paradigma de programação funcional, suas origens, como e o que é e apresenta os principais conceitos que uma linguagem de programação funcional deve ter.

O Capítulo 3 apresenta a Lógica Fuzzy com exemplos e fundamenta a base do que será usado no restante do trabalho.

O Capítulo 4 apresenta a Linguagem R, sua história, seus recursos e como ela se encaixa dentro do paradigma funcional.

O Capítulo 5 apresenta um estudo de caso, codificado em Linguagem R, que exemplifica um modelo de negócios usando Lógica Fuzzy.

1. PROGRAMAÇÃO FUNCIONAL

Programação Funcional tem suas raízes na lógica matemática. Sistemas de lógica matemática existem há séculos, mas só recentemente dois importantes sistemas foram formalizados matematicamente, de acordo com (MICHAELSON, 2012, p. 28).

Cálculo proposicional que tem como valores básicos **true** e **false** e como operações básicas **and**, **not** e **or**. Segundo (Lógica proposicional, 2014):

> A lógica proposicional estuda como raciocinar com afirmações que podem ser verdadeiras ou falsas, ou ainda como construir a partir de um certo conjunto de hipóteses (proposições verdadeiras num determinado contexto) uma demonstração de que uma determinada conclusão é verdadeira no mesmo contexto.

Cálculo de Predicados é uma extensão do *Cálculo proposicional* que adiciona novas ferramentas para uso em valores não lógicos. De acordo com (MICHAELSON, 2012, p. 28):

> This is achieved through the introduction of predicates, which generalize logical expressions to describe properties of non-logical values, and functions to generalize operations on non-logical values. It also introduces the idea of quantifiers to describe properties of sequences of values, for example, universal quantification, for all of a sequence having a property, or

> existential quantification, for one of a sequence having a property. Additional axioms and rules of inference are provided for quantified expressions.

O paradigma da programação funcional está baseado no *Cálculo de Predicados* e suas idéias existem há décadas, mas só recentemente começou a ser incorporado em algumas linguagens de programação. De acordo com (FORD, 2014, p. 2):

> Functional programming follows the same conceptual trajectory as object orientation: developed in academia over the last few decades, it has slowly crept into all modern programming languages.

O cálculo proposicional e de predicados faz com que as associações entre nomes e valores sejam imutáveis e a ordem de execução de expressões não seja fixa.

Segundo (FORD, 2014, p. 4), as maiores mudanças que vêm ocorrendo nas linguagens de programação são adições de recursos funcionais.

As linguagens de programação tradicionais, sejam orientadas ou não a objetos, são chamadas de **imperativas**, como em (MICHAELSON, 2012, p. 19). Linguagens imperativas executam os comandos na ordem em que foram definidos, de forma sequencial. Em linguagens funcionais, não existe uma ordem fixa de execução, sendo esta uma das grandes diferenças entre os paradigmas. Basicamente, o que define uma linguagem como funcional é:

- **Variáveis com um único valor**

  Um nome em linguagens funcionais só possui um único valor ao contrario das linguagens imperativas, onde um nome pode possuir diferentes valores.

- **Ordem de execução**

  Não existe uma ordem fixa de execução das funções em uma linguagem de programação funcional, muito útil em programação paralela e em algoritmos que exigem execução de forma paralela.

- **Chamadas recursivas**

  Recursividade é a possibilidade de uma função chamar a si mesma. É uma característica de praticamente todas as linguagens de programação, mas que é essencial em linguagens funcionais.

- **Funções como tipos**

  Funções são tratadas como valores e podem ser passadas como parâmetros ou retornadas para outras funções. Um recurso que possibilita a criação de soluções mais abstratas e com maior clareza, segundo (FORD, 2014, p. 4).

De acordo com (FORD, 2014, p. 5):

> Object Oriented makes code understandable by encapsulating moving parts. Functional Programming makes code understandable by minimizing moving parts.

Ou seja, o paradigma de orientação a objeto tenta controlar o acesso a valores mutáveis enquanto o paradigma de programação funcional tenta **remover** a mutabilidade.

Imutabilidade é um conceito muito forte dentro do paradigma funcional que garante que os valores utilizados nunca mudem. Elimina código propenso a erro (error-prone) e facilita muita a codificação paralela.

Uma característica muita evidente em linguagens funcionais é o uso intensivo de listas como estrutura de dados principal. As operações nesse tipo de estrutura de dados são bem otimizadas.

Programas funcionais descrevem expressões e transformações, modelando fórmulas matemáticas e evitando a mutabilidade de valores e alguns blocos de construção são definidos pelo paradigma funcional, na forma do uso e do resultado. Esses blocos de construção são ubíquos nas linguagens de programação funcionais (FORD, 2014, p. 24). Ou seja, o conceito é único, mas cada linguagem funcional tem sua sintaxe. Segue um resumo de quais e do que são esses blocos de construção:

- **Filter**

Uma operação que filtra uma lista segundo um critério definido pelo usuário. O resultado é quase sempre uma lista menor, filtrada (FORD, 2014, p. 24).

- **Map**

Uma operação que transforma uma coleção de valores em uma nova coleção aplicando uma função de critério para cada item da coleção original (FORD, 2014, p. 25).

- **Fold/Reduce**

É o conceito mais enraizado no paradigma funcional. É uma generalização do conceito de Catamorfismo (Catamorphism, 2014). Segundo (FIGUEIREDO, 2004):

> Resumida e bastante informalmente, um catamorfismo sobre uma estrutura de dados (operação comum de teoria das categorias) aplica uma operação (binária) sobre cada um dos elementos dessa estrutura de dados, compondo resultados da operação anterior com o valor do próximo elemento. Por exemplo, podemos definir a soma dos elementos de uma lista como fold (+) 0. Essa definição pode ser feita em linguagens como ML e Haskell, mas apenas para uma única estrutura de dados (tipicamente uma lista), e não para somar os elementos de

> qualquer estrutura de dados para a qual exista uma definição de fold apropriada.

Fold/Reduce são chamadas de operações de redução, que processam uma lista e a "reduzem" a um único resultado ou a um sumário. Esse tipo de operação pode ser construído como loops em linguagens tradicionais, mas a grande vantagem em linguagens funcionais é a possibilidade de paralelização, especialmente quando se usa operadores com a propriedade comutativa.

Funções que recebem funções como parâmetros e/ou retornam funções são chamadas de funções de alta ordem. Se for possível expressar uma operação em uma função de alta ordem, o paradigma funcional se adéqua perfeitamente. Sendo assim, é possível definir a operação em termos de uma linguagem funcional, com o ganho adicional da paralelização (FORD, 2014, p. 39).

Um dos recursos que praticamente toda linguagem funcional possui é o conceito de **Closures**. Um Closure é uma função que encapsula o contexto (variáveis) do momento em que foi chamada. Um Closure mantém uma cópia encapsulada de tudo que existia no escopo quando o Closure foi criado. Praticamente, um mecanismo de execução portátil (FORD, 2014, p. 41).

Linguagens imperativas usam o **estado** para modelar a programação, enquanto linguagens funcionais usam o **comportamento**, encapsulado dentro de um Closure, que pode ser passado dentro de estruturas de dados tradicionais e executados no momento exato (FORD, 2014, p. 42).

As funções em linguagens funcionais podem ser utilizadas utilizando técnicas derivadas da Matemática (FORD, 2014, p. 44):

- **Função curried**: é uma função que possui vários argumentos, mas cuja chamada é convertida em várias chamadas de funções de apenas um argumento.

- **Função parcialmente aplicada**: quando uma função com vários parâmetros é chamada, mas alguns desses parâmetros são omitidos e uma função com os parâmetros faltantes é criada. Esse comportamento chama-se Aplicação Parcial e a função criada é chamada de função parcial. É uma forma de criar em tempo real uma nova função.

Exemplos de **Curry** e **Aplicação Parcial** (em Groovy):

```
def volume = {h, w, l -> h * w * l}          // Bloco de código que define o cálculo do volume de um objeto retangular

def area = volume.curry(1)                   // Aqui faz-se o curry da função, fixando o valor de h em 1 retornando uma
                                             // função de 2 parâmetros: w e l. A função retornada é uma função parcial.

def lengthPA = volume.curry(1, 1)            // Novamente faz-se um curry, fixando os valores de h = 1 e w = 1, retornando
                                             // uma função de apenas um parâmetro: l. A função retornada é uma função
                                             // parcial.

def lengthC = volume.curry(1).curry(1)       // Aqui faz-se o curry duas vezes. Na primeira retorna uma função parcial de 2
                                             // parâmetros e na segunda retorna uma função de apenas um parâmetro, com o
                                             // mesmo resultado de lengthPA. A função retornada é uma função parcial.
```

2. LÓGICA FUZZY

2.1 O que é lógica?

Lógica é uma maneira específica de raciocínio. De acordo com Aristóteles a lógica tem como objeto de estudo o pensamento e tem como elementos constituintes o **conceito**, **juízo** e **raciocínio**. As ligações entre esses elementos formam as leis da lógica.

Em programação, a lógica é usada para definir o fluxo de execução de um programa e é baseada na lógica proposicional, que define como unidades mínimas os valores **verdadeiro** e **falso**.

No entanto, nem tudo pode ser representado apenas pelos valores **verdadeiro** ou **falso**. Alguns tipos de situação são complexos e dinâmicos (MCNEIL e THRO, 1994, p. 2), como por exemplo:

- Como definir seu uma pessoa é ***saudável***?
- Como classificar o nível de ***depressão*** de um paciente?
- Como classificar objetos como ***largos***?
- Como classificar pessoas como ***velhas***?
- Como decidir frear quando um obstáculo está ***perto***?

Os termos em negrito não podem ser definidos de forma que a resposta seja **sim** ou **não**.

2.2 O que é Fuzzy?

Difuso é incerto. No mundo, os eventos ocorrem de forma constante, caótica, dentro de variações muito longe de apenas **verdadeiro** ou **falso**. Em outras palavras, existem mais situações Fuzzy's do que exatas. É aí que a lógica Fuzzy (fuzzy logic)

entra. Para analisar essas situações e permitir que sistemas possam ser programados e tomar decisões diante de situações aparentemente caóticas.

2.3 Exemplo de Lógica Fuzzy

Um exemplo encontrado em (MCNEIL e THRO, 1994) (que é o livro motivador deste trabalho) demonstra bem como funciona a lógica Fuzzy.

Imagine um cesto de frutas que pode conter laranjas ou maçãs. Segundo a figura 1 abaixo, o cesto só contém maçãs.

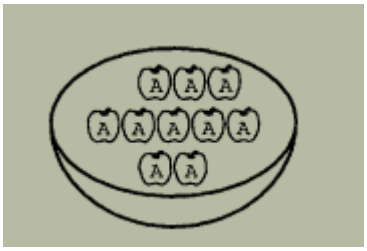

Figura 1 – Cesto de Maçãs

Na figura 2 o cesto só contém laranjas

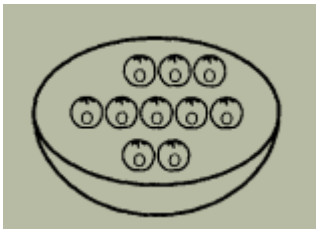

Figura 2 - Cesto de Laranjas

Pela lógica, podemos afirmar que o cesto da Figura 1 possui somente maçãs. Ou seja, podemos através da lógica proposicional afirmar por **verdadeiro** ou **falso** que o cesto da figura 1 contém ou não maçãs.

O mesmo ocorre com o cesto da figura 2. Podemos afirmar por **verdadeiro** ou **falso** que o cesto possui ou não laranjas.

Esse é um exemplo de uso da lógica proposicional, onde podemos afirmar que um cesto **OU** possui só maçãs **OU** possui só laranjas.

Mas o mundo real não funciona assim.

Alguém pode ter misturado o conteúdo dos cestos dessa forma:

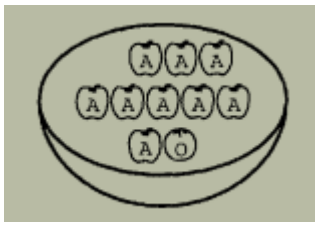

Figura 3 - Uma laranja no meio das maçãs

Ou dessa forma:

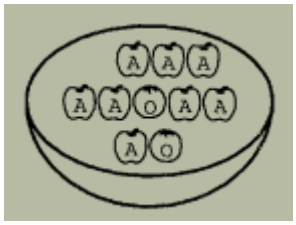

Figura 4 - Ainda tem mais maçãs!

Ou dessa forma:

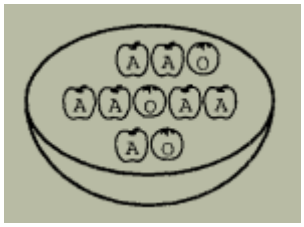

Figura 5 - Maçãs ou Laranjas?

E podemos piorar:

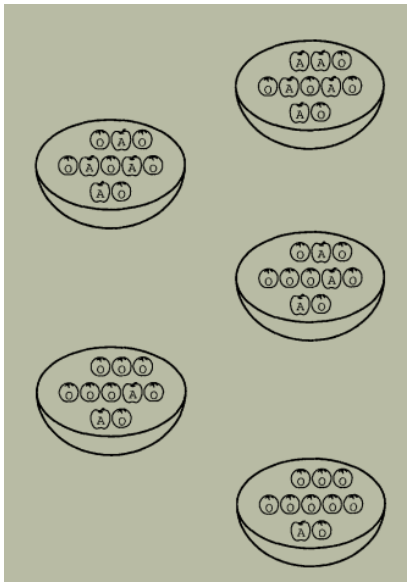

Figura 6 - Como decidir se é um cesto de maçãs ou de laranjas?

Como decidir agora se é um cesto de maçãs ou de laranjas? Note que poderíamos contar as maçãs ou laranjas. Mas o que queremos saber aqui é se o cesto é de maçãs (com algumas laranjas) ou de laranjas (com algumas maçãs). Não dá pra simplesmente usar a lógica proposicional para decidir isso.

Por que pensar em lógica Fuzzy usando laranjas e maçãs? Porque são mais parecidas do que você pensa: são esferas, têm quase o mesmo tamanho, ambas crescem em árvores, pode-se fazer suco de qualquer uma, ocupam o mesmo espaço no cesto, etc.

Podemos dizer que valores lógicos proposicionais, tais como **verdadeiro** e **falso** podem assumir os seguintes valores:

```
falso          = 1
verdadeiro     = 0
```

Sendo assim, soluções usando lógica proposicional só pode ter o valor 0 **OU** o valor 1.

No exemplo do cesto de frutas, podemos assumir valores **ENTRE** 0 e 1. Por exemplo:

```
Não             = 1

Ligeiramente    = 0.9

Um pouco        = 0.7

Mais ou Menos   = 0.5

Alguns          = 0.3

Sobretudo       = 0.1

Sim             = 0
```

Usando estes valores, agora podemos classificar o cesto:

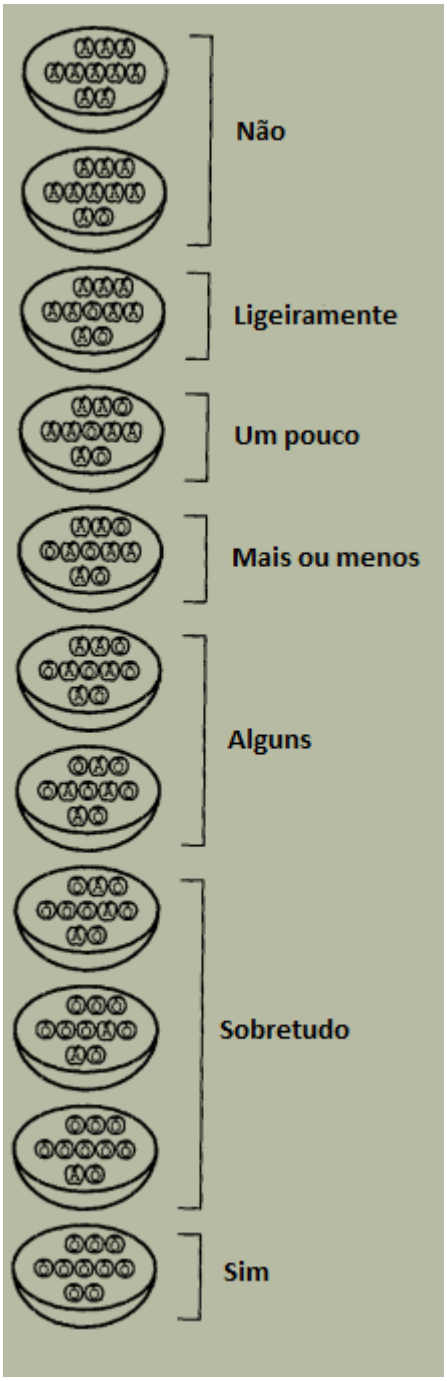

Figura 7 - Classificação do cesto de Laranjas ou Maçãs

No mundo real, quando falamos “estou com pouca fome” ou “o tempo está parcialmente nublado”, estamos definindo situações que são simples para as pessoas, mas que são complexas para um computador decidir. São situações de lógica Fuzzy,

que nosso cérebro consegue assimilar e resolver com facilidade, mas que quando transportadas para um computador se tornam muito difíceis de serem representadas.

Como no exemplo das maçãs e laranjas, é fácil para as pessoas decidirem se um cesto é de maçãs ou de laranjas, analisando a quantidade de maçãs e laranjas no cesto, e decidindo pela maioridade. Mas representar esse modelo de pensamento em computador, em linguagem de programação não é trivial.

Segundo (MCNEIL e THRO, 1994), o nome "fuzzy" é recente, mas o conceito já tem mais de 2500 anos. Aristóteles já considerava a existência de graus de **verdadeiro/falso**. David Hume acreditava na lógica do senso comum, que é baseada no conhecimento que pessoas normais adquiriam vivendo no mundo.

Bertrand Russell estudou a imprecisão da língua falada, demonstrada nas frases:

- Quantos grãos de areia você precisa remover de uma pilha de areia até ela não se tornar mais uma pilha de areia?
- Quantos fios de cabelo precisam cair da cabeça de homem para considerar ele careca? (Paradoxo de Bertrand Russel)

A Lógica Fuzzy tem suas origens nos trabalhos de Jan Łukasiewicz, e foi aprimorada por Lotfi Zadeh em 1960, que uniu os conceitos de lógica proposicional com os conjuntos difusos de Łukasiewicz.

3. R

3.1 O que é R?

Para uma pergunta direta, uma resposta mais direta. **R** é um dialeto de **S**.

3.2 O que é S?

S é uma linguagem desenvolvida em 1976, nos laboratórios da AT&T Corp. Foi criada para ser um ambiente de análise estatística. A ideia era fornecer aos usuários um ambiente interativo, inclusive para usuários que não fossem desenvolvedores. Durante muitos anos **S** foi um misto de linguagem e ambiente utilizado em empresas para análise de dados, sendo vendido como um produto completo.

Pelo fato de **S** ser um produto pago, logo surgiu a necessidade de ter um produto similar, mas gratuito. **R** foi criado em 1991 na Universidade de Auckland, tendo sido anunciado em 1993 publicamente, através da licença GNU, tornando-o gratuito.

R é um dialeto de **S**, possui a maioria de seus recursos, mas evoluiu independentemente de **S**. Atualmente, possui um grupo de desenvolvimento bem ativo, que libera versões constantemente e agrega novos recursos regularmente. **R** ainda segue a filosofia de **S** (PENG, 2015, p. 6), mas possui recursos mais avançados, como visualizações gráficas 3D complexas.

3.3 Ferramentas

O projeto que mantém as ferramentas de desenvolvimento reside no site http://www.r-project.org/ , onde pode ser encontrado endereços para download do software, manuais, notícias e diversas outras informações sobre a linguagem R.

O ambiente de desenvolvimento R, que engloba o interpretador da linguagem R, um console para execução de comandos e funções, documentação e bibliotecas, pode ser baixado para os sistemas operacionais Windows 32/64, OSX e Linux no endereço http://cran.r-project.org/mirrors.html, que fornece diversos mirrors (inclusive no Brasil).

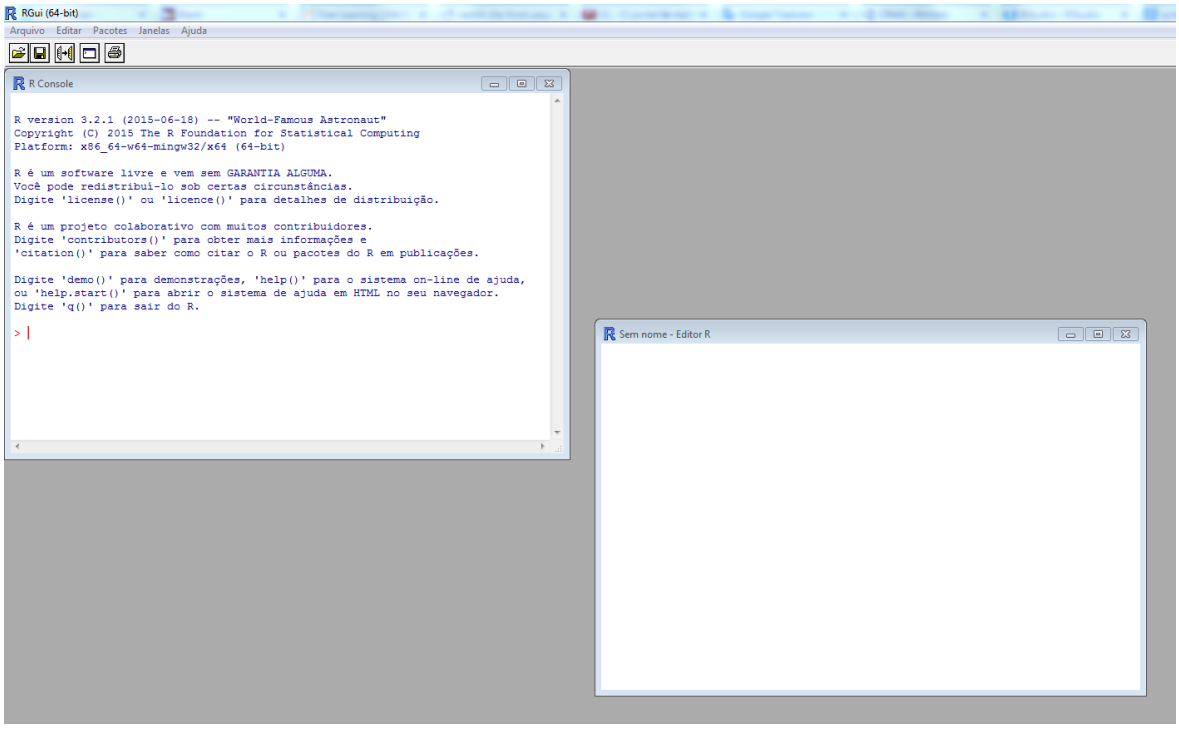

Figura 8 - Tela do Ambiente de Desenvolvimento Padrão de R

Outra ferramenta muito interessante é o **RStudio**, um IDE para a Linguagem R, que pode ser baixado no endereço http://www.rstudio.com/. A versão Open Source oferece mais recursos do que o ambiente padrão de desenvolvimento da Linguagem R, tais como, Ajuda Integrada, Execução de Scripts direto do código fonte, um debugger melhor, realce de sintaxe, etc.

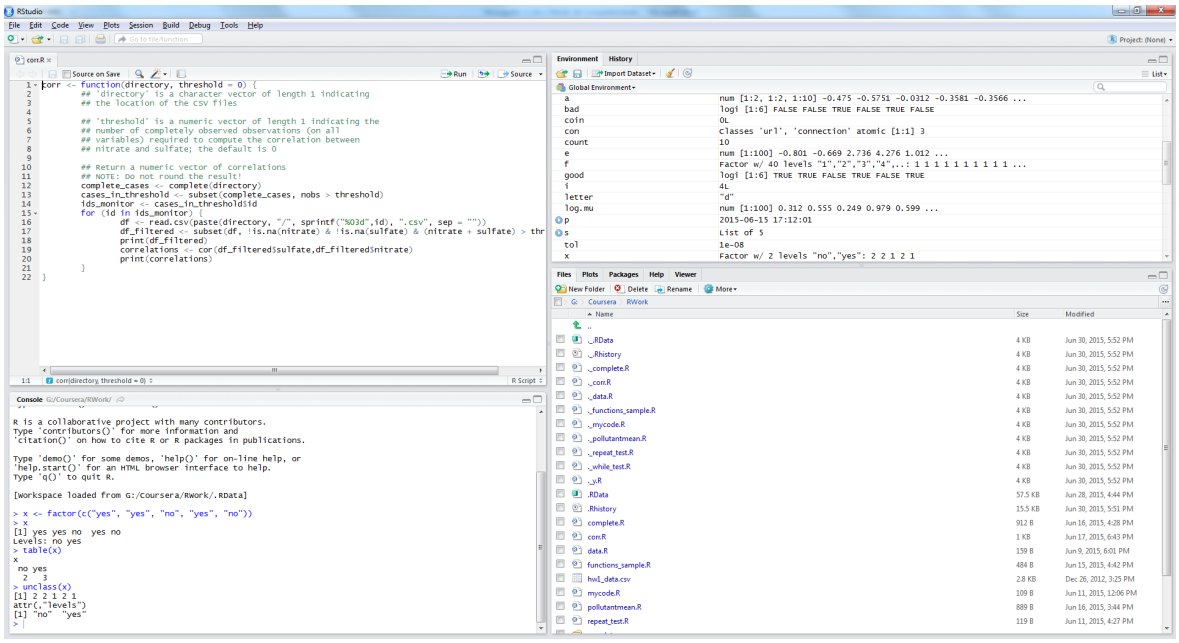

Figura 9 - IDE do RStudio

Recomendo que seja utilizado o **RStudio** para testar todos os exemplos desse trabalho.

O **RStudio** permite gerenciar as bibliotecas instaladas em seu ambiente de desenvolvimento. Isso facilita o uso das bibliotecas necessárias para demonstrar os exemplos desse trabalho.

3.4 R: Estruturas de Controle

R possui estruturas de controle semelhantes às outras linguagens de programação. As estruturas de controle que R possui são:

- if-else
- for Loops
- for Loops Aninhados
- while Loops
- repeat Loops

- next, break

Estas estruturas de controle existem porque durante a programação em R podem existir cenários em que elas sejam necessárias. Como esperado por desenvolvedores experientes, elas agem como esperado e não é novidade, além de serem bem vindas à caixa de ferramentas. Mas vale lembrar que estamos falando de uma linguagem funcional. O conceito de programação funcional nos faz pensar de forma diferente. Ao longo deste trabalho será demonstrado como realizar as mesmas tarefas que são realizadas com essas estruturas de controle, mas de forma "funcional".

3.5 R: Tipos de Dados

Os tipos de dados mais básicos de R são:

- character
- numeric (números reais)
- integer
- complex
- logical (True/False)

Tudo em R é tratado como objeto, até mesmo os tipos de dados básicos. Objetos em R podem conter informações (metadata) associadas ao objeto que ajudam a descrever o objeto. A função ***attributes()***, quando usada com um objeto, retorna os atributos do objeto ou **NULL** no caso do objeto não possuir atributos.

Os tipos de dados mais característicos de R não são os tipos básicos, e sim os que trabalham com conjuntos de dados. São eles:

Vetores (Vectors) – Contém um conjunto de objetos que **devem** ser do mesmo tipo de classe. Ou seja, se o **Vector** for de **integer**, todos os elementos de um **Vector** devem ser do mesmo tipo, **integer**.

Matrizes (Matrices) – São **Vectors** com um atributo que define uma dimensão (quantidade de linhas e colunas da matriz). Exatamente como os **Vectors**, devem possuir objetos do mesmo tipo.

Listas (Lists) – Listas são parecidas com **Vectors**, mas podem conter elementos de qualquer tipo de classe.

Fatores (Factors) – Fatores são usados para representar dados categorizados, ordenados ou não. São muito importantes em modelagem estatística. De forma simples, cada valor de **Factor** é um rótulo (label) que recebe um valor e pode ser contado. No exemplo a seguir é demonstrada a criação de Fator e o seu uso:

```
>## Criando um Fator de nome "x"
> x <- factor(c( " yes" , " yes" , " no" , " yes" , " no" ))

>## Exibindo o conteúdo do Fator
> x
[ 1 ] yes yes no yes no
Levels: no yes

## Tabulando o conteúdo do Fator e as respectivas quantidades para cada Fator
> table(x)
x
no yes
2   3

> ## Visualizando a representação interna do Fator
> unclass(x)
[ 1 ] 2 2 1 2 1
attr(, " levels" )
[ 1 ] " no" " yes"
```

Data Frames – Armazena dados em formato tabular, que podem ser obtidos de arquivos. Podem ser criados manualmente, mas o uso mais comum é ler um arquivo CSV, por exemplo, e armazenar o conteúdo em um Data

Frame, para posterior análise. Desse tipo de dado vem a principal deficiência da Linguagem R: o uso de memória. Data Frames são armazenadas integralmente na memória. Isso pode causar problemas para conjuntos de dados muito grandes.

`Funções (Functions)` – Funções em R são tratadas como objetos, como qualquer outro tipo. Podem ser passadas como parâmetros para outras funções e pode-se definir uma função dentro de outra função.

3.6 R: Funções

Funções são tipos de dados em R e podem ser passadas como parâmetros para outras funções. São o tipo de dado mais importante e a base da programação funcional existente na linguagem R (PENG, 2015, p. 69) e podem ser definidas em qualquer lugar, mesmo dentro de outra função.

Funções em R possuem o recurso de **parâmetro correspondente** (parameter matching), que permite especificar parâmetros pelo nome, pela posição do parâmetro ou pelo nome *parcial* de um parâmetro.

Funções têm seus parâmetros avaliados de forma preguiçosa (lazy). Somente quando os parâmetros são necessários é que eles são avaliados e recuperados para serem usados no corpo da função. Existe um tipo de parâmetro especial "**...**", que, quando usado, indica que a função possui um número variável de parâmetros. É muito utilizado quando existe a necessidade de **estender** outra função (o mesmo conceito de estender uma classe em linguagens orientadas à objeto). Normalmente o parâmetro "**...**" é utilizado no final da lista de parâmetros, mas pode ser utilizado no meio. Nesse caso, todos os parâmetros subsequentes devem ser nomeados, de forma que não exista uma necessidade de usar parâmetros posicionais.

A seguir alguns exemplos de funções e parâmetros:

```
> f <- function() {
+ ## Essa é função vazia. Note que a variável f "contêm" a função.
+ }
> ## Funções possuem sua própria classe
> class(f)
[ 1 ] " function"
> ## Executando a função
> f()
NULL
```

3.7 R: Pacotes de Lógica Fuzzy

Uma das grandes vantagens da linguagem R são os pacotes de terceiros, que podem ser adicionados ao ambiente da linguagem R. No caso, existem diferentes pacotes disponíveis para programação em lógica Fuzzy. O pacote que será utilizado como base no desenvolvimento do caso de uso é o "sets".

Para instalar o pacote "sets", deve-se executar o seguinte comando na linha de comando interativa do ambiente R:

```
install.packages("sets")
```

Uma janela de seleção de servidor de pacotes pode aparecer. Nesse caso, não importa o local de onde será baixado o pacote. Apenas certifique-se de que o servidor esteja no ar.

Para utilizar o pacote deve-se executar o seguinte comando:

```
library(sets)
```

Se algum erro surgir da execução desse comando, o pacote "sets" não foi instalado corretamente. Caso nenhuma mensagem apareça, o pacote "sets" foi carregado corretamente.

3.8 R: Exemplo de uso do pacote “sets”

Apenas como demonstração do uso do pacote “sets”, vamos criar um sistema classificatório simples, utilizado para definir a classificação de um potencial segurado que está sendo analisado por uma companhia de seguros. Criaremos as variáveis linguísticas e as regras sobre essa variáveis, além de realizar um **Fuzzificação** e uma **Defuzzificação**.

Alguns termos precisam ser definidos antes:

- **Fuzzificar**: significa pegar os valores de entrada e converter para conjuntos fuzzy, que são quantidades fuzzy geradas em função das variáveis.
- **Defuzzificar**: significa pegar os conjuntos fuzzy e as regras definidas para obter uma valor específico. No exemplo, qual a classificação do segurado.

Primeiro definimos o Universo. Vamos definir o alcance e a granularidade. No exemplo que vamos criar, o alcance vai e 0 à 40 e granularidade é de 0.1. Todos as entradas devem estar dentro da faixa do alcance e a granularidade garante a precisão. Basicamente, o alcance define os valores do eixo X quando da plotagem em um gráfico. Definimos isso com o seguinte comando:

```
sets_options("universe", seq(from = 0, to = 40, by = 0.1))
```

Variáveis linguísticas são criadas para descrever os valores numéricos usados na análise fuzzy de forma que o resultado seja bem legível. Vamos criar alguma variáveis para o nosso exemplo:

1. **imc** – Índice de massa corporal
    a. abaixo
    b. ideal
    c. acima

    d. obeso
2. **a1c** – Teste de Hemoglobina Glicada
    a. b = baixo
    b. n = normal
    c. a = alto
3. **pressao** – Pressão sanguínea
    a. normal
    b. prehipertenso
    c. hipertenso
    d. fatal
4. **classificacao** – Classificação do segurado
    a. NAU = Não aceito
    b. N = Normal
    c. P = Preferido

O seguinte comando deverá ser executado na linha de comando do IDE do R:

```
variaveis <- set(
imc = fuzzy_partition(varnames = c(abaixo = 9.25, ideal = 21.75, acima = 27.5, obeso = 35),  sd = 3.0),
a1c = fuzzy_partition(varnames = c(b = 4, n = 5.25, a = 7), FUN = fuzzy_cone, radius = 5),
classificacao = fuzzy_partition(varnames = c(NAU = 10, N = 5, P = 1), FUN = fuzzy_cone, radius = 5),
pressao = fuzzy_partition(varnames = c(normal = 0, prehipertenso = 10, hipertenso = 20, fatal = 30), sd
= 2.5))
```

OBS.:

- O parâmetro **sd** indica o valor do **Desvio padrão**.
- **FUN** indica qual tipo de função será utilizada na fuzzificação, ou seja, a função que irá gerar os conjuntos fuzzy.
- **radius** é parâmetro da função fuzzificadora.

Após executar o comando, digite "variaveis" na linha comando do IDE do R e o seguinte resultado deverá aparecer:

```
{a1c = (b = <<gset(91)>>, n = <<gset(100)>>, a = <<gset(101)>>), classificacao = (NAU =
<<gset(101)>>, N = <<gset(101)>>, P = <<gset(61)>>), imc = (abaixo = <<gset(401)>>, ideal =
<<gset(401)>>, acima = <<gset(401)>>, obeso = <<gset(401)>>), pressao = (normal =
<<gset(401)>>, prehipertenso = <<gset(401)>>, hipertenso = <<gset(401)>>, fatal =
<<gset(401)>>)}
```

Isso significa que as variáveis forma configuradas corretamente dentro do pacote "sets". Com os valores das variáveis definidas, podemos agora criar as regras.

As regras no exemplo vão classificar um segurado (variável **classificacao**) de acordo com os valores do Índice de Massa Corporal (variável **imc**), do Teste de Hemoglobina Glicada (variável **a1c**) e da pressão sanguínea (variável **pressão**).

Digite o seguinte comando na linha de comando do IDE do R:

```
regras <- set(
    fuzzy_rule(imc %is% abaixo || imc %is% obeso || a1c %is% b, classificacao %is% NAU),
    fuzzy_rule(imc %is% acima  || a1c %is% n || pressao %is% prehipertenso, classificacao %is% N),
    fuzzy_rule(imc %is% ideal && a1c %is% n && pressao %is% normal, classificacao %is% P)
)
```

Esse comando cria regras dentro do pacote "sets":

1. **Regra 1:** Classifica como não autorizado se o IMC estiver baixo **OU** obeso **OU** o Teste de Hemoglobina Glicada for baixo
2. **Regra 2:** Classifica como normal se o IMC for acima **OU** o Teste de Hemoglobina Glicada for normal **OU** a pressão sanguínea for pré-hipertensa.
3. **Regra 3:** Classifica como preferido se o IMC for ideal **E** o Teste de Hemoglobina Glicada for normal **E** a pressão sanguínea for normal.

Agora precisamos definir um sistema, que no caso do pacote "sets" é o conjunto de variáveis e regras. Para definir um sistema, digite o seguinte comando na linha de comando do IDE do R:

```
sistema <- fuzzy_system(variaveis, regras)
```

Podemos ver como foi criado o sistema digitando o seguinte comando na linha de comando do IDE do R:

```
print(sistema)
```

Que deverá apresentar o seguinte resultado:

```
A fuzzy system consisting of 4 variables and 3 rules.

Variables:

a1c(b, n, a)
classificacao(NAU, N, P)
imc(abaixo, ideal, acima, obeso)
pressao(normal, prehipertenso, hipertenso, fatal)

Rules:

imc %is% ideal && a1c %is% n && pressao %is% normal => classificacao %is% P
imc %is% acima || a1c %is% n || pressao %is% prehipertenso => classificacao %is% N
imc %is% abaixo || imc %is% obeso || a1c %is% b => classificacao %is% NAU
```

Também podemos demonstrar o sistema criado através de gráficos. Digite o seguinte comando na linha de comando do IDE do R:

```
plot(sistema)
```

Obtemos o seguintes gráficos como resultado:

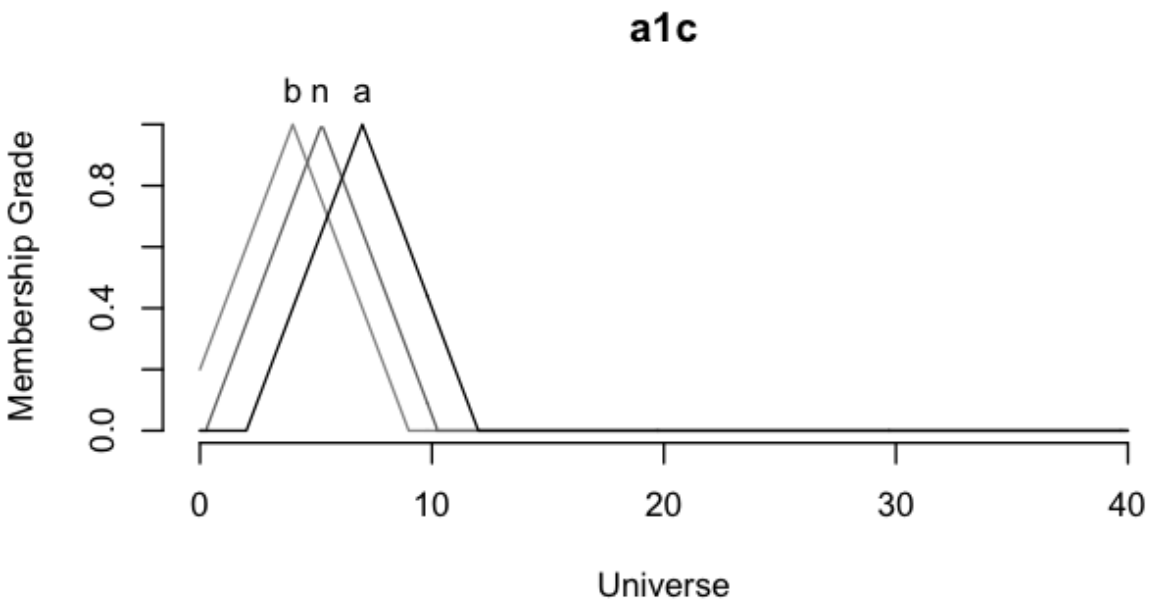

Figura 10 – a1c - Teste de Hemoglobina Glicada

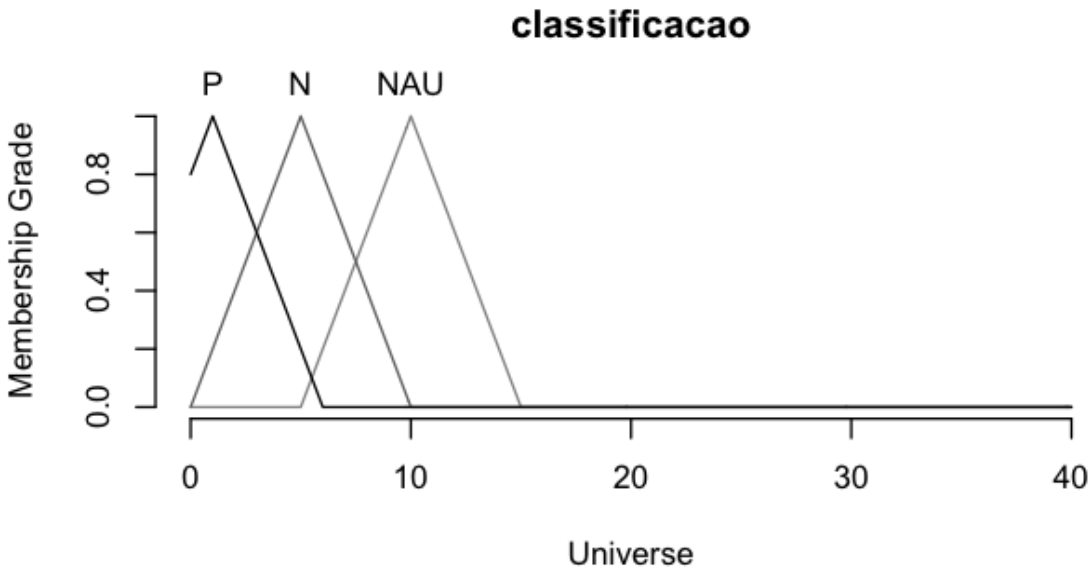

Figura 11 – classificacao – Classificação do segurado.

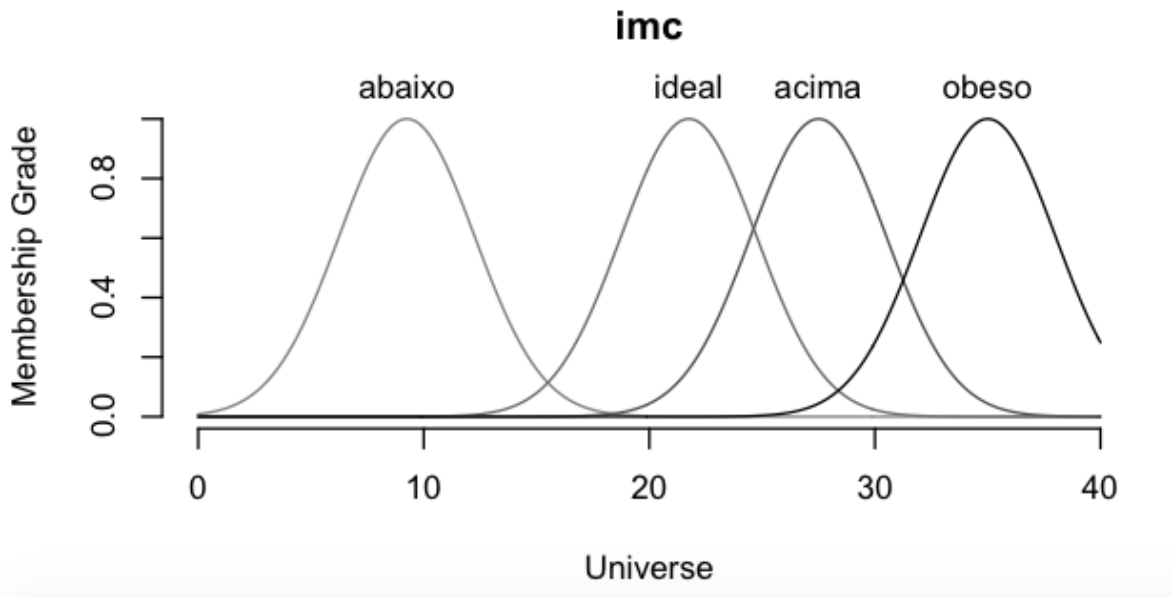

Figura 12 – imc – Índice de Massa Corporal.

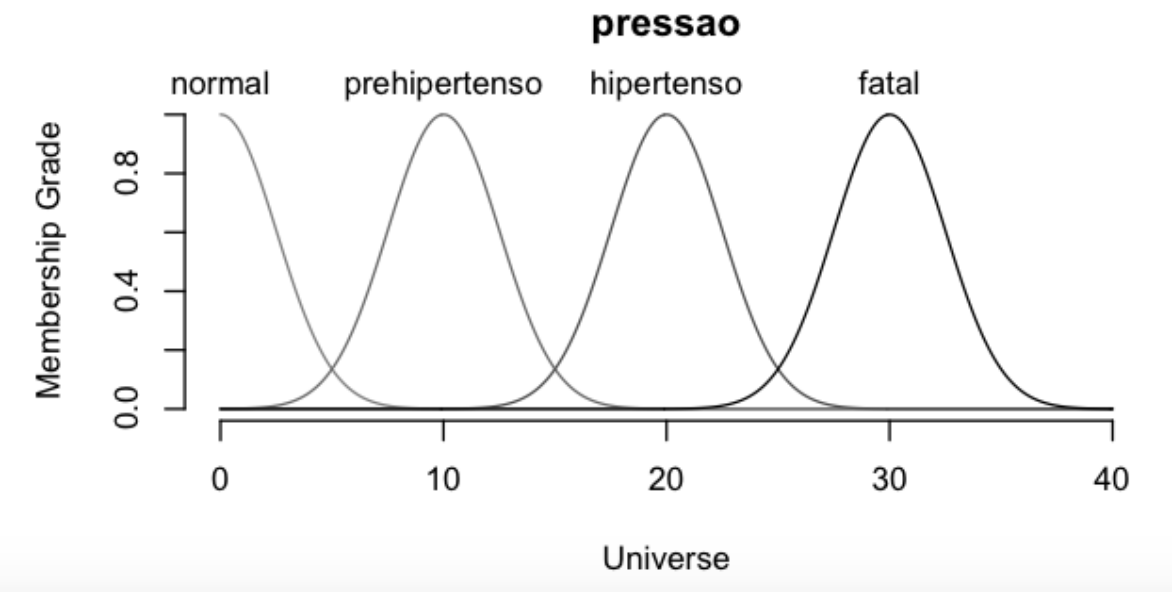

Figura 13 – pressao – Pressão Sanguínea.

Os gráficos demonstram o que foi definido dentro do exemplo apresentado. De forma clara, ficou representado que o "universo" do modelo escolhido ficou entre os valores 0 e 40. Todas as variáveis são representadas nos gráficos dentro desse universo.

A definição de bons nomes para as variáveis ajuda no entendimento do modelo e na interpretação dos resultados. Os gráficos acima dão uma idéia de como a interpretação fica mais fácil.

Com o sistema definido com suas variáveis e suas regras, agora podemos utilizar o sistema de inferência fuzzy implementado pelo pacote “sets”. Na linha de comando do IDE do R, digite o seguinte comando:

```
fi <- fuzzy_inference(sistema, list(imc = 29, a1c=5, pressao=20))
```

Este exemplo tem como Índice de Massa Corporal (imc) o valor de 29, o valor do Teste de Hemoglobina Glicada como 5 e o valor da Pressão Sanguínea como 20.

Podemos visualizar o gráfico dessa inferência digitando o seguinte comando na linha de comando do IDE do R:

```
plot(fi)
```

E obtemos o seguinte gráfico:

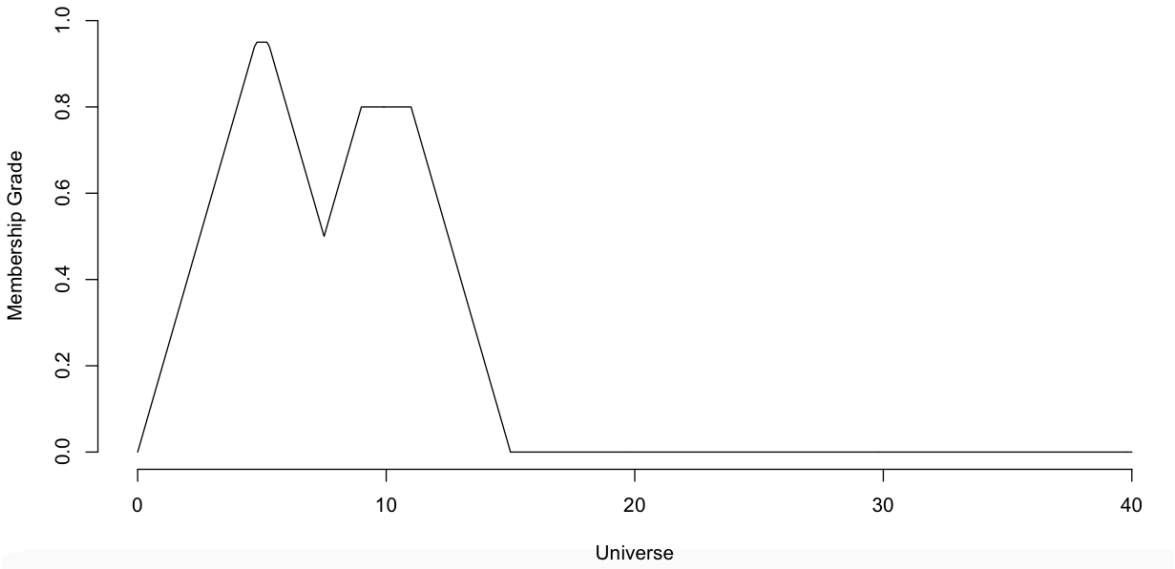

Figura 14 – Resultado da Inferência do sistema fuzzy

O que acabamos de fazer foi a “fuzzificação” dos dados do exemplo, baseado nas variáveis e nas regras que criamos anteriormente. “Fuzzificar” significa realizar a análise e gerar valores que representem um resultado.

Os valores “fuzzificados” não são de fácil interpretação. Por exemplo, se digitarmos o seguinte comando na linha de comando do IDE do R:

```
print(fi)
```

Obtemos o seguinte resultado:

```
{0 [1.266417e-14], 0.1 [0.02], 0.2 [0.04], 0.3 [0.06], 0.4 [0.08], 0.5 [0.1], 0.6 [0.12], 0.7 [0.14],
0.8 [0.16], 0.9 [0.18], 1 [0.2], 1.1 [0.22], 1.2 [0.24], 1.3
 [0.26], 1.4 [0.28], 1.5 [0.3], 1.6 [0.32], 1.7 [0.34], 1.8 [0.36], 1.9 [0.38], 2 [0.4], 2.1 [0.42], 2.2
[0.44], 2.3 [0.46], 2.4 [0.48], 2.5 [0.5], 2.6 [0.52], 2.7
 [0.54], 2.8 [0.56], 2.9 [0.58], 3 [0.6], 3.1 [0.62], 3.2 [0.64], 3.3 [0.66], 3.4 [0.68], 3.5 [0.7], 3.6
[0.72], 3.7 [0.74], 3.8 [0.76], 3.9 [0.78], 4 [0.8], 4.1 [0.82],
 4.2 [0.84], 4.3 [0.86], 4.4 [0.88], 4.5 [0.9], 4.6 [0.92], 4.7 [0.94], 4.8 [0.95], 4.9 [0.95], 5 [0.95],
5.1 [0.95], 5.2 [0.95], 5.3 [0.94], 5.4 [0.92], 5.5 [0.9], 5.6
 [0.88], 5.7 [0.86], 5.8 [0.84], 5.9 [0.82], 6 [0.8], 6.1 [0.78], 6.2 [0.76], 6.3 [0.74], 6.4 [0.72], 6.5
[0.7], 6.6 [0.68], 6.7 [0.66], 6.8 [0.64], 6.9 [0.62], 7 [0.6],
 7.1 [0.58], 7.2 [0.56], 7.3 [0.54], 7.4 [0.52], 7.5 [0.5], 7.6 [0.52], 7.7 [0.54], 7.8 [0.56], 7.9
[0.58], 8 [0.6], 8.1 [0.62], 8.2 [0.64], 8.3 [0.66], 8.4 [0.68], 8.5
 [0.7], 8.6 [0.72], 8.7 [0.74], 8.8 [0.76], 8.9 [0.78], 9 [0.8], 9.1 [0.8], 9.2 [0.8], 9.3 [0.8], 9.4 [0.8],
9.5 [0.8], 9.6 [0.8], 9.7 [0.8], 9.8 [0.8], 9.9 [0.8], 10
 [0.8], 10.1 [0.8], 10.2 [0.8], 10.3 [0.8], 10.4 [0.8], 10.5 [0.8], 10.6 [0.8], 10.7 [0.8], 10.8 [0.8],
10.9 [0.8], 11 [0.8], 11.1 [0.78], 11.2 [0.76], 11.3 [0.74], 11.4
 [0.72], 11.5 [0.7], 11.6 [0.68], 11.7 [0.66], 11.8 [0.64], 11.9 [0.62], 12 [0.6], 12.1 [0.58], 12.2
[0.56], 12.3 [0.54], 12.4 [0.52], 12.5 [0.5], 12.6 [0.48], 12.7
 [0.46], 12.8 [0.44], 12.9 [0.42], 13 [0.4], 13.1 [0.38], 13.2 [0.36], 13.3 [0.34], 13.4 [0.32], 13.5
[0.3], 13.6 [0.28], 13.7 [0.26], 13.8 [0.24], 13.9 [0.22], 14
 [0.2], 14.1 [0.18], 14.2 [0.16], 14.3 [0.14], 14.4 [0.12], 14.5 [0.1], 14.6 [0.08], 14.7 [0.06], 14.8
[0.04], 14.9 [0.02], 15 [2.220446e-16]}
```

que não é de fácil interpretação.

Mas ainda não é o resultado final de nosso exemplo. Precisamos saber qual a classificação desse exemplo, de acordo com nosso sistema (lembre-se, o sistema é composto de variáveis e regras). Para fazer isso, precisamos “defuzzificar” nossa inferência.

Para “defuzzificar” uma inferência, digite o seguinte comando na linha de comando do IDE do R:

```
gset_defuzzify(fi,  “centroid “)
```

Obtemos o seguinte valor

```
[1] 7.445238
```

Que é o valor da variável **classificacao**.

Finalmente temos, para o nosso exemplo, uma valor que classifica o segurado. Nesse caso, o segurado de exemplo está muito perto de não ser aceito.

Toda vez que encerramos as inferências, é considerado uma boa prática limpar todo a configuração do conjunto fuzzy com o seguinte comando:

```
sets_options( "universe ", NULL)
```

Basicamente, para trabalhar com lógica fuzzy, precisamos seguir os seguintes passos:

1. Definir as variáveis lingüísticas e seus valores
2. Definir as regras, baseadas nas variáveis lingüísticas
3. "Fuzzificar" os valores de análise.
4. "Defuzzificar" os valores fuzzy resultante, de forma a obter uma classificação.

A linguagem R, junto com o pacote "sets"torna fácil implementar soluções baseadas em lógica fuzzy. Em conjunto com a linguagem funcional R, definir regras de negócios mais elaboradas torna-se uma tarefa mais fácil e efetiva.

4. ADAPTAÇÃO DE ALGORÍTMO

A adaptação de algoritmo descrita nesse capítulo refere-se à (MCNEIL e THRO, 1994, p. 89), que é um algoritmo usado para pousar uma nave na lua. O algoritmo controla dois movimentos distintos da nave, um no eixo vertical e outro no eixo horizontal. Trata-se de um simulação que fornece uma valor de impulso para a nave de forma que ela possa pousar na superfície da lua suavemente.

O exemplo encontrado em (MCNEIL e THRO, 1994, p. 89) foi orginalmente construído com ferramentas desenvolvidas pelo próprio autor do livro. Esta adaptação irá utilizar uma linguagem de programação moderna (**R**), utilizando um paradigma de programação mais recente (**funcional**) e bibliotecas padrões de mercado (**sets**). O objetivo é obter o mesmo resultado do exemplo do livro, mas com ferramentas mais recentes.

A lógica fuzzy criada aqui calcula o valor de impulso do **eixo vertical** e do **eixo horizontal**. O valor de impulso de cada eixo é calculado com base nos valores de **distância** e **velocidade** de cada eixo.

O cálculo será feito por eixo, por questões de simplificação. Cada eixo terá seu conjunto de variáveis e seu respectivo valor de impulso calculado, juntos com suas regras.

A seguinte terminologia será utilizada na nomenclatura nos exemplos de códigos:

Table 1 Nomenclatura da variáveis

Abreviatura	Descrição
Vd	Distância vertical
VV	Velocidade vertical
vi	Impulso vertical
hd	Distância horizontal
hv	Velocidade horizontal
hi	Impulso horizontal

A definição da nomenclatura de valores seguirá a seguinte tabela (os nomes estão de acordo com o exemplo original):

Table 2 Range de valores das variáveis fuzzy

Abreviatura	Descrição
LN	Large negative
N	Negative
SN	Small Negative
Z	Zero
SP	Small Positive
P	Positive
LP	Large Positive

4.1 Definindo as variáveis

Primeiro vamos definir dois **sets**: um para o eixo vertical (chamado de **variaveisV**) e outro para o eixo horizontal (chamado de **variaveisH**). Cada conjunto possuirá 3 variáveis fuzzy:

1. `distancia_?` – os valores fuzzy de distância, de acordo com o eixo (substituir o ? por **v** ou **h**)
2. `velocidade_?` – os valores fuzzy de velocidade, de acordo com o eixo (substituir o ? por **v** ou **h**)

3. **`impulso_?`** – os valores fuzzy do impulso no eixo (substituir o ? por **v** ou **h**)

OBS.:

- As variáveis fuzzy são definidas por um vetor de R ((PACE, 2012, p. 8)), que contém valores nomeados (um recurso de R). Cada valor nomeado torna-se uma variável fuzzy.
- O valor **sd** significa **Desvio Padrão** e foi definido como 1.5 em função de testes feitos na adaptação do algoritmo.
- **FUN** é a função que será usada na **fuzzificação** dos valores. Estamos usando um dos recursos do paradigma funcional, que são as funções de primeira ordem (FORD, 2014, p. 114), que podem ser passadas como parâmetros de outras funções. A biblioteca **sets** possui várias funções de fuzzificação. Nesse exemplo adaptado, utilizamos a função **fuzzy_cone** porque é a correta para este tipo de cálculo, segundo os testes de adaptação. Os algoritmos envolvidos nas funções de fuzzificação são extremamente importantes, mas a explicação de cada um está fora do escopo deste trabalho.
- **radius** é parâmetro da função definida por **FUN**.

A definição completa das variáveis é a seguinte:

Table 3 Definindo das varáveis fuzzy

```
variaveisV <-
        set(

        distancia_v =
                fuzzy_partition(varnames=c(vdLN=-300,vdN=-200,vdSN=-
        100,vdZ=0,vdSP=100,vdP=200,vdLP=300)),

        velocidade_v =
                fuzzy_partition(varnames=c(vvLN=-20,vvN=-14,vvSN=-
        8,vvZ=0,vvSP=8,vvP=14,vvLP=20),sd=1.5),

        impulso_v =
                fuzzy_partition(varnames=c(viLN=-20,viN=-14,viSN=-
        8,viZ=0,viSP=8,viP=14,viLP=20), FUN = fuzzy_cone, radius = 5)

)

variaveisH <-
        set(

        distancia_h=
                fuzzy_partition(varnames=c(hdLN=-300,hdN=-200,hdSN=-
        100,hdZ=0,hdSP=100,hdP=200,hdLP=300)),

        velocidade_h=
                fuzzy_partition(varnames=c(hvLN=-20,hvN=-14,hvSN=-
        8,hvZ=0,hvSP=8,hvP=14,hvLP=20), sd = 1.5),

        impulso_h=
                fuzzy_partition(varnames=c(hiLN=-20,hiN=-14,hiSN=-
        8,hiZ=0,hiSP=8,hiP=14,hiLP=20), FUN = fuzzy_cone, radius = 5)

)
```

O range de valores para cada variável está de acordo com a tabela Table 2 Range de valores das variáveis fuzzy.

4.2 Definindo as regras

As regras fuzzy também serão definidas por eixo. Cada conjunto de regras define qual será o valor do **impulso** no eixo, de acordo com os valores de **distância** e **velocidade** informados.

Novamente é criado um **set** (objeto da biblioteca **sets**) com uma lista de regras. Cada regra analise o valor de **distância** e **velocidade** comparando (mais ou menos, porque a "comparação" nesse caso segue modelos de fuzzificação de valores de aproximação e a explicação da teoria envolvida está fora do escopo deste trabalho) com as variáveis fuzzy, definida de acordo com Table 3 Definindo das varáveis fuzzy.

As regras do eixo vertical são as seguintes:

```
regrasV <- set(
  # Regras do eixo Vertical
  fuzzy_rule(distancia_v %is% vdLN && velocidade_v %is% vvLN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdLN && velocidade_v %is% vvN , impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdLN && velocidade_v %is% vvSN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdLN && velocidade_v %is% vvZ , impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdLN && velocidade_v %is% vvSP, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdLN && velocidade_v %is% vvP,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdLN && velocidade_v %is% vvLP, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdN && velocidade_v %is% vvLN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdN && velocidade_v %is% vvN,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdN && velocidade_v %is% vvSN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdN && velocidade_v %is% vvZ,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdN && velocidade_v %is% vvSP, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdN && velocidade_v %is% vvP,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdN && velocidade_v %is% vvLP, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdSN && velocidade_v %is% vvLN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdSN && velocidade_v %is% vvN,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdSN && velocidade_v %is% vvSN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdSN && velocidade_v %is% vvZ,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdSN && velocidade_v %is% vvSP, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdSN && velocidade_v %is% vvP,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdSN && velocidade_v %is% vvLP, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdZ && velocidade_v %is% vvLN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdZ && velocidade_v %is% vvN,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdZ && velocidade_v %is% vvSN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdZ && velocidade_v %is% vvZ,  impulso_v %is% viSP),
  fuzzy_rule(distancia_v %is%  vdZ && velocidade_v %is% vvSP, impulso_v %is% viSP),
  fuzzy_rule(distancia_v %is%  vdZ && velocidade_v %is% vvP,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdZ && velocidade_v %is% vvLP, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdSP && velocidade_v %is% vvLN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdSP && velocidade_v %is% vvN,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdSP && velocidade_v %is% vvSN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdSP && velocidade_v %is% vvZ,  impulso_v %is% viSP),
  fuzzy_rule(distancia_v %is% vdSP && velocidade_v %is% vvSP, impulso_v %is% viSP),
  fuzzy_rule(distancia_v %is% vdSP && velocidade_v %is% vvP,  impulso_v %is% viSP),
  fuzzy_rule(distancia_v %is% vdSP && velocidade_v %is% vvLP, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdP && velocidade_v %is% vvLN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdP && velocidade_v %is% vvN,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdP && velocidade_v %is% vvSN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is%  vdP && velocidade_v %is% vvZ,  impulso_v %is% viP),
  fuzzy_rule(distancia_v %is%  vdP && velocidade_v %is% vvSP, impulso_v %is% viP),
  fuzzy_rule(distancia_v %is%  vdP && velocidade_v %is% vvP,  impulso_v %is% viP),
  fuzzy_rule(distancia_v %is%  vdP && velocidade_v %is% vvLP, impulso_v %is% viSP),
  fuzzy_rule(distancia_v %is% vdLP && velocidade_v %is% vvLN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdLP && velocidade_v %is% vvN,  impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdLP && velocidade_v %is% vvSN, impulso_v %is% viZ),
  fuzzy_rule(distancia_v %is% vdLP && velocidade_v %is% vvZ,  impulso_v %is% viLP),
  fuzzy_rule(distancia_v %is% vdLP && velocidade_v %is% vvSP, impulso_v %is% viP),
  fuzzy_rule(distancia_v %is% vdLP && velocidade_v %is% vvP,  impulso_v %is% viP),
  fuzzy_rule(distancia_v %is% vdLP && velocidade_v %is% vvLP, impulso_v %is% viP)
)
```

Um total de 49 regras são criadas com a função **fuzzy_rule**. As regras criadas com essa função têm uma sintaxe própria e utilizam as variáveis fuzzy definidas em Table 3 Definindo das varáveis fuzzy***.***

Essa função cria máquinas de inferência fuzzy, baseadas na linguagem fuzzy, nas variáveis e nas regras.

Basicamente, são 5 passos:

1. Fuzzificação das variáveis de entrada
2. Aplicação dos operadores (AND, OR, NOT) em antecedentes de alguns valores
3. Implicação de um antecedente para seu consequente
4. Agregação de consequentes através das regras
5. Defuzificação de resultado

A implicação é baseada ou no mínimo ou no produto. A avaliação das expressões lógicas em antecedentes, bem como a agregação da avaliação de cada regra, depende da lógica do set fuzzy.

O detalhamento desses passos está fora do escopo deste trabalho.

As regras do eixo horizontal são as seguintes:

```
regrasH <- set(
  # Regras do eixo Horizontal
  fuzzy_rule(distancia_h %is% hdLN && velocidade_h %is% hvLN, impulso_h %is% hiZ),
  fuzzy_rule(distancia_h %is% hdLN && velocidade_h %is% hvN,  impulso_h %is% hiSN),
  fuzzy_rule(distancia_h %is% hdLN && velocidade_h %is% hvSN, impulso_h %is% hiN),
  fuzzy_rule(distancia_h %is% hdLN && velocidade_h %is% hvZ,  impulso_h %is% hiN),
  fuzzy_rule(distancia_h %is% hdLN && velocidade_h %is% hvSP, impulso_h %is% hiN),
  fuzzy_rule(distancia_h %is% hdLN && velocidade_h %is% hvP,  impulso_h %is% hiLN),
  fuzzy_rule(distancia_h %is% hdLN && velocidade_h %is% hvLP, impulso_h %is% hiLN),
  fuzzy_rule(distancia_h %is% hdN  && velocidade_h %is% hvLN, impulso_h %is% hiSP),
  fuzzy_rule(distancia_h %is% hdN  && velocidade_h %is% hvN,  impulso_h %is% hiZ),
  fuzzy_rule(distancia_h %is% hdN  && velocidade_h %is% hvSN, impulso_h %is% hiSN),
  fuzzy_rule(distancia_h %is% hdN  && velocidade_h %is% hvZ,  impulso_h %is% hiSN),
  fuzzy_rule(distancia_h %is% hdN  && velocidade_h %is% hvSP, impulso_h %is% hiN),
  fuzzy_rule(distancia_h %is% hdN  && velocidade_h %is% hvP,  impulso_h %is% hiN),
  fuzzy_rule(distancia_h %is% hdN  && velocidade_h %is% hvLP, impulso_h %is% hiLN),
  fuzzy_rule(distancia_h %is% hdSN && velocidade_h %is% hvLN, impulso_h %is% hiSP),
  fuzzy_rule(distancia_h %is% hdSN && velocidade_h %is% hvN,  impulso_h %is% hiSP),
  fuzzy_rule(distancia_h %is% hdSN && velocidade_h %is% hvSN, impulso_h %is% hiSP),
  fuzzy_rule(distancia_h %is% hdSN && velocidade_h %is% hvZ,  impulso_h %is% hiSN),
  fuzzy_rule(distancia_h %is% hdSN && velocidade_h %is% hvSP, impulso_h %is% hiSN),
  fuzzy_rule(distancia_h %is% hdSN && velocidade_h %is% hvP,  impulso_h %is% hiN),
  fuzzy_rule(distancia_h %is% hdSN && velocidade_h %is% hvLP, impulso_h %is% hiN),
  fuzzy_rule(distancia_h %is% hdZ  && velocidade_h %is% hvLN, impulso_h %is% hiP),
  fuzzy_rule(distancia_h %is% hdZ  && velocidade_h %is% hvN,  impulso_h %is% hiSP),
  fuzzy_rule(distancia_h %is% hdZ  && velocidade_h %is% hvSN, impulso_h %is% hiSP),
  fuzzy_rule(distancia_h %is% hdZ  && velocidade_h %is% hvZ,  impulso_h %is% hiZ),
  fuzzy_rule(distancia_h %is% hdZ  && velocidade_h %is% hvSP, impulso_h %is% hiSN),
  fuzzy_rule(distancia_h %is% hdZ  && velocidade_h %is% hvP,  impulso_h %is% hiSN),
  fuzzy_rule(distancia_h %is% hdZ  && velocidade_h %is% hvLP, impulso_h %is% hiN),
  fuzzy_rule(distancia_h %is% hdSP && velocidade_h %is% hvLN, impulso_h %is% hiP),
  fuzzy_rule(distancia_h %is% hdSP && velocidade_h %is% hvN,  impulso_h %is% hiP),
  fuzzy_rule(distancia_h %is% hdSP && velocidade_h %is% hvSN, impulso_h %is% hiSP),
  fuzzy_rule(distancia_h %is% hdSP && velocidade_h %is% hvZ,  impulso_h %is% hiSP),
  fuzzy_rule(distancia_h %is% hdSP && velocidade_h %is% hvSP, impulso_h %is% hiZ),
  fuzzy_rule(distancia_h %is% hdSP && velocidade_h %is% hvP,  impulso_h %is% hiSN),
  fuzzy_rule(distancia_h %is% hdSP && velocidade_h %is% hvLP, impulso_h %is% hiSN),
  fuzzy_rule(distancia_h %is% hdP  && velocidade_h %is% hvLN, impulso_h %is% hiLP),
  fuzzy_rule(distancia_h %is% hdP  && velocidade_h %is% hvN,  impulso_h %is% hiP),
  fuzzy_rule(distancia_h %is% hdP  && velocidade_h %is% hvSN, impulso_h %is% hiP),
  fuzzy_rule(distancia_h %is% hdP  && velocidade_h %is% hvZ,  impulso_h %is% hiSP),
  fuzzy_rule(distancia_h %is% hdP  && velocidade_h %is% hvSP, impulso_h %is% hiSP),
  fuzzy_rule(distancia_h %is% hdP  && velocidade_h %is% hvP,  impulso_h %is% hiZ),
  fuzzy_rule(distancia_h %is% hdP  && velocidade_h %is% hvLP, impulso_h %is% hiSN),
  fuzzy_rule(distancia_h %is% hdLP && velocidade_h %is% hvLN, impulso_h %is% hiLP),
  fuzzy_rule(distancia_h %is% hdLP && velocidade_h %is% hvN,  impulso_h %is% hiLP),
  fuzzy_rule(distancia_h %is% hdLP && velocidade_h %is% hvSN, impulso_h %is% hiP),
  fuzzy_rule(distancia_h %is% hdLP && velocidade_h %is% hvZ,  impulso_h %is% hiP),
  fuzzy_rule(distancia_h %is% hdLP && velocidade_h %is% hvSP, impulso_h %is% hiP),
  fuzzy_rule(distancia_h %is% hdLP && velocidade_h %is% hvP,  impulso_h %is% hiSP),
  fuzzy_rule(distancia_h %is% hdLP && velocidade_h %is% hvLP, impulso_h %is% hiZ)
)
```

Um total de 49 regras são criadas com a função **fuzzy_rule**. As regras do eixo horizontal possuem uma complexidade maior, porque o problema em si demanda uma análise maior no eixo horizontal, onde o impulso varia muito. Vale ressalta que este exemplo incorpora as características físicas do veículo e as características da gravidade e atmosfera da lua.

4.3 Fuzzificando!

Após definirmos as variáveis fuzzy e suas regras, precisamos combinar essas informações em sistemas.

Um sistema fuzzy é a combinação de variáveis e regras.

Devemos definir dois sistemas. Um para o eixo vertical e outro para o eixo horizontal:

```
sistemaV <- fuzzy_system(variaveisV, regrasV)
sistemaH <- fuzzy_system(variaveisH, regrasH)
```

Cada variável armazena um sistema com suas variáveis fuzzy e suas regras.

Após a criação dos sistemas, já podemos utilizá-los para calcular o valor do impulso de cada eixo, da seguinte forma:

```
      fiV<-
fuzzy_inference(sistemaV,values=list(distancia_v=0,velocidade_v=7))

      fiH<-
fuzzy_inference(sistemaH,values=list(distancia_h=0,velocidade_h=7))
```

As variáveis **fiV** e **fiH** são os resultados fuzzificados, respectivamente do eixo vertical e do eixo horizontal, de acordo com o sistema usado na função **fuzzy_inference**.

No exemplo acima foram passados como valor de distância **0** e como valor de velocidade **7**. Os mesmo valores foram passados para os dois sistemas (vertical e horizontal), mas devem produzir um resultado diferente porque as regras de cada sistema são completamente diferentes.

Para **defuzzificar** os valores de **fiV** e **fiH** devemos utilizar a função **gset_defuzzify**, que "transforma" o resultado fuzzy em um valor esperado de **impulso**, de acordo com o sistema usado.

```
print(gset_defuzzify(fiV, "centroid"))
print(gset_defuzzify(fiH, "centroid"))
```

O comando **print** deverá exibir os seguintes valores:

```
[1] 7.999722
[1] -7.999722
```

O primeiro valor refere-se ao impulso do eixo vertical e o segundo valor ao impulso do eixo horizontal, de acordo com os níveis definidos nas variáveis fuzzy **impulso_v** e **impulso_h**

OBS.:

centroid – calcula a media aritmética dos elementos do set, usando os valores dos membros como pesos.

5. CONCLUSÃO

As regras que regem os processos de negócios estão se tornando mais complexas, dificultando uma análise usando a simples lógica booleana. Sistemas que precisam analisar grandes conjuntos de dados em cenários muito complexos estão se tornando comuns.

Utilizar a lógica fuzzy, juntamente com a linguagem R e a filosofia da programação funcional pode ser uma solução na criação de motores de avaliação de regras.

Pode-se definir as regras em R, dentro de scripts que podem ser executados dentro dos ambientes de máquinas virtuais mais famosas, tais como .NET e Java. Ao invés de codificar em linguagens tradicionais as regras, elas podem ser codificadas em R e somente os dados necessários para a análise precisam ser fornecidos ao ambiente de execução de R.

Por exemplo, já existem uma interface de R com Java (Java/R Interface), que permite executar scripts R dentro da máquina virtual Java. Esses scripts podem ser alterados com facilidade, e se, a arquitetura for bem planejada, o motor de avaliação de regras pode ficar completamente independente de qualquer aplicação ou ambiente.

Tomadas de decisões podem ser mais facilmente implementadas com as ferramentas apresentadas nesse trabalho.

BIBLIOGRAFIA

BACKFIELD, J. **Becoming Functional**. 1ª. ed. [S.l.]: O'Reilly, 2014.

CATAMORPHISM. **Wikipédia A Enciclopédia Livre**, 2014. Disponivel em: <http://en.wikipedia.org/wiki/Catamorphism>. Acesso em: 8 maio 2015.

FIGUEIREDO, C. C. D. Porque polimosfismo de Sobrecarga. **Porque polimosfismo de Sobrecarga**, 2004. Disponivel em: <http://homepages.dcc.ufmg.br/~camarao/Derivar/porque-polimorfismo-de-sobrecarga.html>. Acesso em: 8 maio 2015.

FORD, N. **Functional Thinking:** Paradigm over syntax. 1. ed. [S.l.]: O'Reilly Media, 2014. 180 p. ISBN 9781449365516.

HEATON, J. **Introduction to the Math of Neural Networks**. [S.l.]: Heaton Research, Inc., 2012. 119 p. ISBN B00845UQL6.

LÓGICA proposicional. **Wikipédia A Enciclopédia Livre**, 2014. Disponivel em: <http://pt.wikipedia.org/wiki/L%C3%B3gica_proposicional>. Acesso em: 01 maio 2015.

LANDER, J. P. **R for Everyone:** Advanced Analytics and Graphics. [S.l.]: Addison-Wesley Professional, 2013. 464 p. ISBN B00HFULELW.

LIMA, I.; PINHEIRO, C.; OLIVEIRA, F. S. **Inteligência Artificial**. Primeira. ed. [S.l.]: Elsevier, 2014. 257 p.

MATLOFF, N. **The Art of R Programming:** A Tour of Stattistical Software Design. First. ed. [S.l.]: No Starch Press, 2011. 400 p. ISBN B00683GXUO.

MCNEIL, F. M.; THRO, E. **Fuzzy Logic:** A Practical Approach. First. ed. [S.l.]: AP professional, 1994. 309 p. ISBN 978-0124859654.

MICHAELSON, G. **An Introduction to Functional Programming Through Lambda Calculus**. First. ed. [S.l.]: Dover Publications, 2012. 336 p. ISBN B00CWR4USM.

NGUYEN, H. T.; WALKER, E. A. **A First Course in Fuzzy Logic**. Third. ed. [S.l.]: Chapman & Hall/CRC, 2006. 463 p.

NORVIG, P.; RUSSEL, S. **Inteligência Artificial**. Terceira. ed. [S.l.]: Elsevier Acadêmico, 2013. 1016 p. ISBN 978-8535237016.

PACE, L. **Beginning R:** An Introduction to Statistical Programming. First. ed. [S.l.]: Apress, 2012. 336 p. ISBN 9781430245544.

PENG, R. D. **R Programming for Data Science**. 1ª. ed. [S.l.]: Leanpub, 2015.

TEETOR, P. **R Cookbook**. First. ed. [S.l.]: O'Reilly, 2011. 438 p. ISBN B004VB3UYW.