ISEL

Biblioteca Python para seleção de características em grandes dados

DIOGO FILIPE CARVALHOSA AMORIM

(Licenciado)

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientadores: Doutora Matilde Pós-de-Mina Pato
Doutor Nuno Soares Datia

Júri:

Presidente: Doutor José Manuel de Campos Lages Garcia Simão
Vogais: Doutor Rui Manuel Feliciano de Jesus
Doutor Nuno Soares Datia

Outubro 2024

Biblioteca Python para seleção de características em grandes dados

DIOGO FILIPE CARVALHOSA AMORIM

(Licenciado)

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientadores: Doutora Matilde Pós-de-Mina Pato, ISEL
Doutor Nuno Soares Datia, ISEL

Júri:

Presidente: Doutor José Manuel de Campos Lages Garcia Simão, ISEL
Vogais: Doutor Rui Manuel Feliciano de Jesus, ISEL
Doutor Nuno Soares Datia, ISEL

Outubro 2024

Agradecimentos

Este trabalho é dedicado a todos aqueles que, de uma forma ou de outra, contribuíram para a sua concretização.

Aos meus orientadores, doutor Nuno Datia e doutora Matilde Pós-de-Mina Pato, pela orientação, paciência e sabedoria com que me guiaram ao longo deste percurso. O vosso apoio e incentivo foram fundamentais para a conclusão desta dissertação.

Aos professores, pelo conhecimento transmitido e pelo exemplo de dedicação ao ensino e à investigação. Cada uma das vossas aulas foi um passo importante na minha formação e crescimento. Aos meus amigos, pela companhia, pelo apoio nos momentos mais difíceis e pelas conversas que tornaram este caminho mais leve e agradável. A vossa amizade foi um pilar essencial para ultrapassar os desafios deste percurso.

À minha família, pelo amor incondicional, pelo apoio constante. Sem vocês, nada disto teria sido possível.

A todos, o meu mais sincero obrigado.

Declaração de integridade

Declaro que esta(e) dissertação é o resultado da minha investigação pessoal e independente. O seu conteúdo é original e todas as fontes listadas nas referências bibliográficas foram consultadas e estão devidamente mencionadas no texto. Mais declaro que todas as referências científicas e técnicas relevantes para o desenvolvimento do trabalho estão devidamente citadas e constam das referências bibliográficas.

O autor

Lisboa, 28 de Outubro de 2024

Biblioteca Python para seleção de características em grandes dados

Copyright© DIOGO FILIPE CARVALHOSA AMORIM, Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa.
O Instituto Superior de Engenharia de Lisboa e o Instituto Politécnico de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "iselthesis" [46] desenvolvido no DEETC do ISEL-IPL

Resumo

Na Mineração de Dados e Aprendizagem Automática, a Seleção de Atributos tem como objetivo otimizar modelos preditivos, eliminando características irrelevantes e redundantes do conjunto de dados original. Automatizar esse processo é desafiador devido à diversidade dos dados e domínios, tornando a generalização e automação da seleção de características uma tarefa complexa.

O algoritmo Ensemble Feature Ranking (EFR), publicado em 2014 no artigo "*Ensemble feature ranking applied to medical data*", destaca-se pela sua capacidade de integrar vários métodos de filtragem e realizar múltiplas execuções em partições de dados reduzidas. Em 2021 foi criado um *package* na linguagem R que implementa uma versão melhorada do algoritmo EFR, sendo esta versão chamada de Enhanced Ensemble Feature Ranking (EEFR), generalização do algoritmo EFR, sendo capaz de ser aplicada a diversos domínios.

Python foi uma das linguagens de programação mais usadas em 2023, sendo uma linguagem bastante usada em Mineração de Dados devido às bibliotecas que esta tem disponíveis, enquanto a linguagem R é muito menos usada. Tendo como base o *package* em R, fizemos uma reimplementação em Python deste algoritmo. Além disso, a versão em Python foi projetada com funcionalidades adicionais, como a seleção de classes e uma lista negra de características, permitindo que os usuários manipulem os dados de forma mais intuitiva.

Outro destaque do trabalho é o desenvolvimento de um *dashboard* interativo, que visa aumentar a explicabilidade do algoritmo, essencial em áreas críticas como a medicina. O *dashboard* oferece uma visualização clara das métricas internas e do comportamento do algoritmo, promovendo transparência e compreensão nas decisões tomadas pelo modelo. Também se transformou esta versão em Python numa biblioteca.

O trabalho inclui a análise de desempenho das implementações em ambas as linguagens, utilizando conjuntos de dados reais, que possuem características distintas em termos de dimensões e instâncias. Os testes foram realizados com diferentes métricas, além da implementação de conjuntos de dados sintéticos para explorar o impacto do número de instâncias e características.

Palavras-chave: Seleção de Caraterísticas, Aprendizagem Automática, Filtros, Conjunto de Filtros, Aprendizagem em Conjunto, Explicabilidade

Abstract

In Data Mining (DM) and Machine Learning (ML), Feature Selection (FS) aims to optimize predictive models by eliminating irrelevant and redundant features from the original data set. Automating this process is challenging due to the diversity of data and domains, making generalization and automation of feature selection a complex task.

The Ensemble Feature Ranking (EFR) algorithm, published in 2014 in the paper "Ensemble feature ranking applied to medical data,"stands out for its ability to integrate various filtering methods and perform multiple executions on reduced data partitions. In 2021, a package was created in the R language that implements an improved version of the EFR algorithm, called Enhanced Ensemble Feature Ranking (EEFR), which generalizes the EFR algorithm and can be applied to various domains.

Python was one of the most used programming languages in 2023, being a widely used language in Data Mining due to the available libraries, while the R language is much less utilized. Based on the R package, we reimplemented this algorithm in Python. Additionally, the Python version was designed with additional features such as class selection and a blacklist of features, allowing users to manipulate data more intuitively.

Another highlight of this work is the development of an interactive dashboard, which aims to increase the explainability of the algorithm, essential in critical areas such as medicine. The dashboard provides a clear visualization of internal metrics and the algorithm's behavior, promoting transparency and understanding in the decisions made by the model. This version has also been transformed into a library in Python, making both the algorithm and the dashboard available.

The work includes a performance analysis of the implementations in both languages, using real datasets that have distinct characteristics in terms of dimensions and instances. The tests were conducted with different metrics, along with the implementation of synthetic datasets to explore the impact of the number of instances and features.

Keywords: Feature Selection, Machine Learning, Filters, Filter Ensemble, Ensemble Learning Explainability

Índice

Índice de Figuras xv

Índice de Tabelas xvii

Índice de Listagens xix

Siglas xxi

1 Introdução 1
- 1.1 Contributos . . . 3
- 1.2 Organização do documento . . . 3

2 Conceitos fundamentais e trabalho relacionado 5
- 2.1 Trabalhos relacionados . . . 6

3 Reimplementação da biblioteca em Python 11
- 3.1 Estudo e compreensão do *package* em R . . . 11
- 3.2 Reimplementação da biblioteca em Python . . . 15

4 Novas funcionalidades e “explicabilidade” do algoritmo 23
- 4.1 Funcionalidades adicionadas à biblioteca . . . 23
- 4.2 Programa auxiliar de visualização . . . 29

5 Transformar em biblioteca 37

6 Avaliação 43

7 Conclusões e trabalho futuro 49

Bibliografia 51

Apêndices

A Diagramas de classes da biblioteca em Python 57

Índice de Figuras

1.1 Conjunto ótimo de caraterísticas 2

3.1 Diagrama de classes do *package* EnsembleFeaturesRanking feito em R 12
3.2 Diagrama de classes das funções usadas no package `FSelector` 14
3.3 Diagrama de classe da biblioteca 16

4.1 *Log* gerado durante o teste da primeira versão da biblioteca 25
4.2 Exemplo de diagrama de blocos usando as classes desenvolvidas para a aplicação auxiliar de visualização da evolução do algoritmo 31
4.3 Páginas ilustrativas da aplicação. A interface mostra os blocos do diagrama do algoritmo, permitindo um raciocínio sequencial (2). Ao clicar num dos blocos, tem-se acesso a um painel com informações mais detalhadas e contextuais, neste caso, para os diagramas gerados para as métricas (6). 33

6.1 Tempo de execução para o *dataset* KDD Cup 2008 em Python 44
6.2 Tempo de execução para o *dataset* KDD Cup 2008 em R 44
6.3 Tempo de execução para o *dataset* Arcene em Python 45
6.4 Tempo de execução para o *dataset* Arcene em R 45
6.5 Comparação do resultado gerado em Python com a *baseline* em R para o *dataset* KDD Cup 2008 47
6.6 Comparação do resultado gerado em R com a *baseline* em R para o *dataset* KDD Cup 2008 47
6.7 Comparação do resultado gerado em Python com a *baseline* em R para o*dataset* Arcene 48
6.8 Comparação do resultado gerado em R com a *baseline* em R para o *dataset* Arcene 48

A.1 Diagrama de classe do *package* metrics 58
A.2 Diagrama de classe do *package* discretizer 58
A.3 Diagrama de classe do *package* knowledge viewer 59
A.4 Diagrama de classe do *package* block diagram 59
A.5 Diagrama de classe do *package* page 60

Índice de Tabelas

4.1 Informação de *log* guardada em cada parte do algoritmo 28
4.2 Caraterísticas das “páginas” da aplicação auxiliar para visualizar a evolução do algoritmo com descrição de caraterísticas especiais dos gráficos 34

6.1 Dimensões dos *datasets* sintéticos 46

Índice de Listagens

3.1 Código comum às métricas 14
5.1 Adição ao ficheiro de configuração para usar autoapi 38
5.2 Estrutura dos ficheiro de índice da documentação 38
5.3 Bloco de código usado para obter o *path* para os *icons* 39
5.4 Alterações a realizar ao ficheiro pyproject.toml para conseguir usar dependências geradas de forma dinâmica 40
5.5 Alterações a realizar ao ficheiro pyproject.toml para adicionar recursos ao *package* 40

Siglas

AI	Artificial Inteligence 7, 8
ANN	Artificial Neural Network 7
DL	Deep Learning 6, 7
DM	Data Mining xi
DNN	Deep Neural Networks 8
EEFR	Enhanced Ensemble Feature Ranking ix, xi, 3, 49
EFR	Ensemble Feature Ranking ix, xi, 2, 7
FE	Feature Extraction 5, 6
FR	Feature Reduction 1, 2, 3, 5
FROC	Free-response Receiver Operating Characteristic 6
FS	Feature Selection xi, 5, 6, 7, 9
GNN	Graph Neural Network 7
JVM	Java Virtual Machine 17
MDL	Minimum Description Length 14, 17
ML	Machine Learning xi, 1, 2, 7, 8
PCA	Principal Component Analysis 6
PyPI	Python Package Index 3, 37, 41
SVD	Singular Value Decomposition 6
XAI	eXplainable Artificial Intelligence 2, 3, 7, 8

Introdução

Nos últimos anos, houve um crescimento dos dados em vários domínios que gerou grandes desafios na análise de dados eficiente e eficaz [8]. Estes dados de grande volume (dimensões e instâncias) são atualmente conhecidos por *big data*, podendo os dados serem estruturados, semi-estruturados e não estruturados. Este termo pode estar associado não só à dimensão dos dados em si, mas também com a sua variância, velocidade de produção e veracidade dos mesmos [23].

Este tipo de dados traz consigo grandes desafios, tais como a ineficiência computacional, o aumento do risco de sobre-parametrização (*overfitting*) e a diminuição da interpretabilidade, uma vez que estes dados podem vir de diferentes fontes, tais como redes sociais, redes de sensores, transações do próprio negócio entre outras, sendo bastante difícil para um ser humano conseguir interpretar toda esta informação ou até mesmo conseguir entender o que é relevante para o problema. Com todos estes problemas gerados pelo *big data*, aumentou a necessidade de usar técnicas de Feature Reduction (FR) em *pipelines* de pré-processamento de dados para melhorar o desempenho de algoritmos de Machine Learning (ML).

As técnicas de Feature Reduction, também conhecida como *Dimensionality Reduction*, têm como objetivo fazer uma seleção ou transformação de um subconjunto das caraterísticas relevantes, enquanto preserva a informação relevante necessária para criar uma modelação precisa. Ao descartar caraterísticas irrelevantes e/ou redundantes, estas técnicas diminuem o esforço computacional necessário para calcular o modelo e reduzem a maldição da dimensionalidade[1], aumentando a capacidade de generalização [68]. Ao descartar características redundantes ou irrelevantes, essas técnicas não apenas aliviam a carga computacional, mas também atenuam a maldição da dimensionalidade, facilitando assim uma melhor generalização e interpretabilidade do modelo. Entre a miríade de abordagens de redução de recursos, os métodos de conjunto têm atraído atenção significativa pela sua capacidade de aproveitar a inteligência coletiva de diversos modelos. A maldição da dimensionalidade, indica que para um dado número finito de amostras, existe um número de caraterísticas ótimo para o qual o desempenho do classificador é máximo, a partir deste limite o classificador apresenta *overfitting* [15]. Deste modo tem-se como objetivo diminuir a dimensionalidade do conjunto de dados para diminuir este

[1] Também conhecida como fenómeno de Hughes [29].

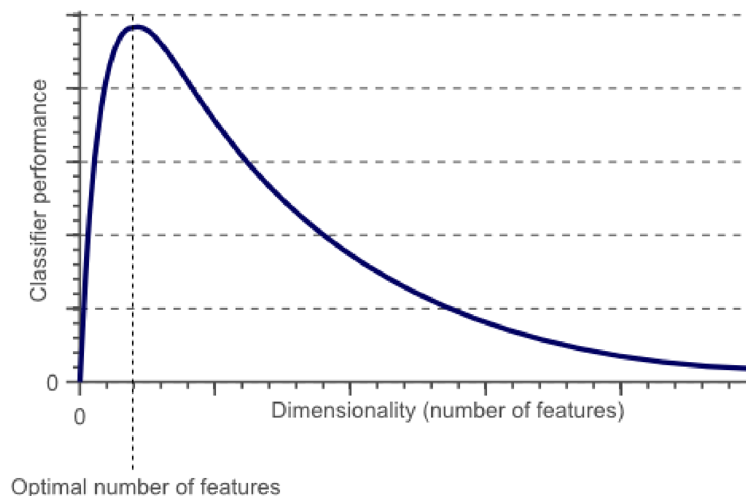


Figura 1.1: *Gráfico do classificador em função da dimensionalidade do conjunto de caraterísticas [54]*

enviesamento. Na Figura1.1 pode-se observar um gráfico que ilustra o fenómeno de Hughes. As técnicas de FR são bastante eficazes, mas por vezes apresentam resultados enviesados [57]. Para evitar que este tipo de problema ocorra, recorre-se ao uso de *Ensemble Learning*. *Ensemble Learning* é uma técnica de ML que melhora a precisão e resiliência, misturando o resultado de diversas técnicas. O objetivo desta técnica é minimizar os erros ou viés que possam existir em modelos individuais, aproveitando a capacidade coletiva do conjunto. O conceito por de trás do *Ensemble Learning* é combinar o resultado das diferentes técnicas e gerar um resultado mais preciso. Ao utilizar os pontos fortes de diferentes modelos, o *Ensemble learning* melhora o desempenho geral do sistema [56]. Um exemplo bastante conhecido deste tipo de técnicas que gera bons resultados é a algoritmo *Random Forest*, que internamente gera varias árvores aleatoriamente e que através da combinação dos seus resultados é capaz de apresentar resultados muito superiores aos da capacidade das árvores individuais.

Esta dissertação tem como objetivo reimplementar o algoritmo Ensemble Feature Ranking (EFR) criado por Vítor Santos [52] e o qual teve uma primeira implementação, generalização e melhoria feita em R pelo Henrique Gomes [25]. Este algoritmo tem como objetivo efetuar o pré-processamento de dados de elevada dimensionalidade de modo a que se possa gerar um conjunto de dados com menos ruído e de menor dimensão. Esta reimplementação pretende levar a biblioteca em R para a linguagem Python, que é uma das linguagens de programação mais usadas na atualidade, que obtenha resultados equivalentes e cujo desempenho seja igual ou melhor do que o da biblioteca original.

Adicionamos novas funcionalidades face à implementação em R, tais como a seleção da classe (atributo dependente), lista negra de caraterísticas e criação de um *dashboard* interativo para consultar a evolução do algoritmo (*offline*). Este *dashboard* interativo tem como objetivo dotar o algoritmo explicável (eXplainable Artificial Intelligence (XAI)) através da utilização de diagramas e gráficos, dado este ser um dos desafios atuais aprendizagem automática [34]. Pretende-se que o utilizador consiga efetuar uma análise visual e perceber o funcionamento do

algoritmo e os tornar mais inteligíveis os resultados obtidos na execução.

Ao explorar as bibliotecas e algoritmos disponíveis em Python, encontramos várias opções para implementar técnicas de FR. Exemplos incluem a `feature-engine` e o `scikit-learn`. Além disso, há bibliotecas que suportam técnicas de *Ensemble Learning*, como o `pycobra`. No entanto, o *ensemble* oferecido por essas bibliotecas é limitado, envolvendo apenas a combinação de modelos para geração de previsões, sem ser o mesmo tipo de *ensemble* proposto pelo algoritmo reimplementado nesta dissertação.

1.1 Contributos

Como resultado do trabalho desenvolvido nesta dissertação temos as seguintes contribuições:

- implementação do algoritmo EEFR em Python, resultando no código *open source* em [40];
- criação de XAI para explicar os resultados do algoritmo EEFR;
- criação da XEFRpy biblioteca com esta implementação disponível no repositório principal Python Package Index (PyPI) [41];
- escrita e apresentação do artigo intitulado "Explainable Feature Ranking using Interactive Dashboards"[5] na 28th International Conference Information Visualisation.

1.2 Organização do documento

Este documento está dividido em 7 capítulos. Este capítulo faz o enquadramento dos temas abordados, apresenta conceitos fundamentais relativos ao algoritmo, motivação e objetivos da dissertação.

No capítulo 2 aprofunda-se mais os conceitos por trás do funcionamento do algoritmo implementado. Apresenta um estudo do estado atual dos algoritmos e seus resultados face à deteção de cancro, que é o tipo dados usado durante os testes da biblioteca.

No capítulo 3 apresenta o estudo da implementação em R, a reimplementação em Python. No capítulo 4 explica-se as melhorias que foram efetuadas face a versão em R para que esta se torna-se mais rápida e útil. No capítulo 5 apresenta o processo de estruturar o código, alterações necessárias e escrita de documentação para publicação da biblioteca em Python no PyPI.

No capítulo 6 apresenta a comparação dos resultados em função do tempo e resultado da implementação Python e R (comparação de resultados Python-R e R-R, para se perceber a variância dos resultados).

Por fim, no capítulo 7 são apresentadas as conclusões que obtivemos ao longo da dissertação, bem como aquilo que achamos que ainda se deve adicionar/melhorar na biblioteca.

2 Conceitos fundamentais e trabalho relacionado

Neste capítulo pretende-se efetuar uma apresentação abrangente sobre o estado das técnicas Feature Reduction (FR), também conhecidas como *Dimensionality Reduction* existentes e trabalhos que demonstram a utilização de técnicas.

As técnicas de FR são usadas para diminuir o conjunto de dados, removendo caraterísticas irrelevantes, para melhorar o desempenho de algoritmos e reduzir o tempo de treino. Pode-se dividir as técnicas de FR em 2 conjuntos mais específicos, relativamente ao facto das caraterísticas resultantes pertencerem ou não ao conjunto de caraterísticas inicial, sendo estes o Feature Selection (FS) e Feature Extraction (FE) [10]. As técnicas de FS envolvem a seleção de um subconjunto de caraterísticas mais relevantes do conjunto de dados original. É feito sem transformar as caraterísticas e pode ser alcançado através dos métodos de:

- *Filter*: Estes métodos aplicam uma medida estatística para atribuir uma pontuação a cada característica. Os recursos são classificados pela pontuação e selecionados para serem mantidos ou removidos do conjunto de dados. Os métodos geralmente consideram a característica independentemente, ou em relação à variável dependente. Exemplos destes métodos incluem *Chi-squared* e *information gain*.
- *Wrapper*: Estes métodos consideram a seleção de um conjunto de características como um problema de pesquisa, onde diferentes combinações são preparadas, avaliadas e comparadas com outras combinações. Um modelo preditivo é usado para avaliar uma combinação de características e atribuir uma pontuação com base na precisão do modelo. Os métodos de *Wrapper* usam um subconjunto de recursos e treinam um modelo usando estes recursos. Com base nas inferências que podem ser extraídas do modelo anterior, eles decidem adicionar ou remover recursos do seu subconjunto. O problema é basicamente reduzido a um problema de pesquisa. Exemplos destes métodos incluem a eliminação recursiva de características, a *forward selection*, *backwards propagation* e os algoritmos genéticos.
- *Embedded*: Estes métodos combinam as qualidades dos métodos de *Filter* e *Wrapper*. É implementado por algoritmos que têm os próprios métodos de seleção de caraterísticas. Alguns algoritmos de aprendizagem realizam seleção de caraterísticas como parte da

operação geral. Exemplos destes métodos incluem regressão *lasso* e *ridge* que têm regularização incorporada, o que ajuda a eliminar características irrelevantes.

- *Ensamble*: Estes métodos combinam múltiplos métodos de FS diferentes para criar um resultado melhor. Geralmente reduz o risco do subconjunto resultante estar enviesado.

As técnicas de *Feature Extraction* envolvem a criação de novas caraterísticas combinando ou transformando as caraterísticas originais. O objetivo é criar um conjunto de caraterísticas que capturam a essência dos dados originais em um espaço de menores dimensões. Existem vários métodos para a extração de características, incluindo a Principal Component Analysis (PCA) e Singular Value Decomposition (SVD).

2.1 Trabalhos relacionados

Dos trabalhos com especial relevância, temos o trabalho desenvolvido por Perlich *et al.* [47] no âmbito do KDD Cup 2008 e o trabalho desenvolvido por Talukder *et al* [61]. O primeiro é o relatório apresentado pelos vencedores do concurso que descreve as etapas e os conceitos usados no trabalho. A competição do KDD Cup 2008 consistiu em duas tarefas distintas: (i) aumento da AUC da Free-response Receiver Operating Characteristic (FROC), entre 0,2 e 0,3 falsos positivos por imagem, e (ii) redução da carga de trabalho dos radiologistas, diminuindo o número de exames que devem ser revistos, garantindo que todos os pacientes doentes são reavaliados. A segunda tarefa foi avaliada segundo dois parâmetros, sensibilidade e especificidade. Os 3 primeiros lugares conseguiram todos alcançar a sensibilidade máxima, por outro lado, este trabalho destacou-se muito em especificidade, onde este conseguiu valores entre 0,624 e 0,681, enquanto o segundo classificado caiu para uma especificidade de 0,174.[1]

O segundo trabalho é um trabalho de pesquisa que tinha como objetivo acelerar a deteção de cancro. Esta pesquisa introduz um modelo híbrido de *ensemble feature extraction* e *ensemble learning* para identificar cancro do pulmão e do cólon. De acordo com os resultados divulgados, avaliando com base em conjuntos de dados de pulmão e cólon histopatológicos (LC25000), este modelo híbrido pode detetar cancro de pulmão, cólon e (pulmão e cólon) com precisão de 0,991, 1 e 0,993, respetivamente.

O que é realmente importante nestes trabalhos é que estes utilizam *ensemble* de modelos e conseguiram melhorar os resultados face aos modelos usados. Também é importante realçar que métodos de FE tendem a ser computacionalmente mais exigentes que métodos de FS, tendo também o problema de que estes não costumam poder ser aplicados a novos dados, devido ao facto de gerarem um novo conjunto de caraterísticas. Ao usar FS não se tem este tipo de limitação, uma vez que esta não altera as caraterísticas, podendo ser aplicado a novos dados e sendo possivelmente computacionalmente menos exigente.

Passando para o Deep Learning (DL), que é uma área que tem ganho popularidade em diversas áreas. O trabalho de Jabeen *et al.* [31] demonstra que DL também consegue obter resultados

[1]Resultados disponíveis em `https://kdd.org/kdd-cup/view/kdd-cup-2008`. Competição KDD CUP 2008

impressionantes, alcançando precisões de 0,991. O grande problema que se pode observar relativamente ao uso de DL é o tempo e a capacidade de processamento que este necesita.

O trabalho desenvolvido por Yu *et al.* [71] é um estudo de diversos algoritmos de *Feature Ranking* aplicados a conjuntos de dados de diagnóstico de cancro da mama. Os resultados deste estudo demonstram que existem 3 algoritmos que são relativamente estáveis, independentemente do conjunto de dados. Pode-se constatar que para alguns conjuntos de dados é possível atingir precisões superiores a 0,9. Com este trabalho conseguimos perceber que também é possível obter bons resultados através de Feature Selection. Por outro lado, não se encontrou nenhum trabalho que realizasse Ensemble Feature Ranking, além das dissertações antecedentes de Santos [52] e de Gomes [25].

Embora muitos algoritmos de aprendizagem automática e inteligência artificial apresentem resultados excelentes em várias situações, eles frequentemente enfrentam um desafio conhecido como o "problema da caixa-preta", em que os utilizadores têm dificuldade em compreender a lógica por trás das soluções apresentadas por estes algoritmos. Notavelmente, os modelos baseados em Artificial Neural Network (ANN) estão entre os mais difíceis de interpretar. Assim, a eXplainable Artificial Intelligence (XAI) é crucial para melhorar a transparência e a confiança nos sistemas de AI, ao permitir que os utilizadores compreendam as decisões por eles tomadas. Isto ajuda a identificar e a reduzir preconceitos, garantindo assim equidade e conformidade com as regulamentações [6]. A XAI capacita os utilizadores a verificar os resultados dos modelos, tomar decisões informadas e avançar a tecnologia de AI ao oferecer *insights* sobre o comportamento e desempenho dos modelos. Além disso, promove a sensibilização pública sobre AI e é vital para garantir responsabilidade, equidade e o uso responsável da IA em vários domínios [4].

Vários artigos de investigação exploram o conceito de explicabilidade em diferentes áreas da AI, como métodos aplicados a DL [36], aplicação em dados tabulares [51], uso em Graph Neural Network (GNN) [3], para dados categóricos e mistos [35], ou estudando como os profissionais avaliam artefactos de software, aplicando o conceito de seleção explicável de características [72].

No campo dos algoritmos de seleção e ordenação de características, a literatura disponível é relativamente limitada. Apenas alguns estudos recentes abordaram este tema, incluindo [28]. Estes estudos têm empregado principalmente métricas para quantificar a importância das características dentro de modelos de aprendizagem específicos que incorporam uma fase de ordenação de características [28], ou utilizam métodos visuais não interativos para transmitir a importância das características, como visto em [72]. Este artigo distingue-se pelo tratamento do algoritmo de ordenação de características como um componente central, independentemente das técnicas de ML empregues. Permite que os utilizadores acompanhem o processo do algoritmo passo a passo, oferecendo tanto *insights* visuais quanto interativos sobre como a ordenação final é calculada. Esta abordagem é particularmente benéfica para lidar com conjuntos de dados de alta dimensionalidade, melhorando a interpretabilidade e a compreensão da importância das características.

O trabalho de Adadi *et al.* [2] oferece uma exploração abrangente de XAI, enfatizando a sua natureza interdisciplinar e importância em vários domínios. Examina o contexto histórico

da explicabilidade, contrastando os sistemas especialistas com os modelos modernos de ML, como as Deep Neural Networks (DNN). O trabalho identifica desafios chave na XAI, incluindo complexidades técnicas, a falta de uma classificação sistemática na investigação existente e a necessidade de definições e estruturas mais claras. A pesquisa destaca tendências recentes, o papel da interação humana e as implicações económicas da XAI, concluindo com sugestões para direções futuras de investigação para melhorar a explicabilidade nos sistemas de AI.

Adicionalmente, Erico *et al.*[67] exploram a interpretabilidade e a explicabilidade dos algoritmos de ML na área médica. Categorizam vários métodos interpretativos, enfatizando os desafios impostos pela complexidade dos modelos e pela qualidade dos dados. As suas descobertas destacam os riscos associados a más interpretações, restrições incompletas e a necessidade de supervisão humana nas aplicações médicas de ML. Os autores defendem a realização de mais estudos comparativos sobre técnicas de interpretabilidade e sugerem uma abordagem cautelosa na implementação de algoritmos em contextos clínicos, priorizando métodos tradicionais até que estruturas interpretativas fiáveis sejam estabelecidas.

O regulamento de AI da União Europeia [16] representa a primeira regulamentação abrangente sobre AI, visando estabelecer um quadro para a implementação segura e responsável das tecnologias de AI nos estados membros. Classifica os sistemas de AI de acordo com os seus níveis de risco, impondo requisitos mais rigorosos nas aplicações de alto risco, enquanto proíbe sistemas inaceitáveis, como aqueles que utilizam práticas manipulativas ou reconhecimento facial em tempo real. O Regulamento também enfatiza a transparência para a AI generativa e procura promover a inovação, fornecendo apoio a *startups* através de ambientes de teste. A aplicabilidade total está prevista para 2026, com algumas disposições a entrarem em vigor mais cedo.

De acordo com o que Erico *et al.*[67] referiram na qualidade dos dados, área onde se acredita que este trabalho seja mais usado, a qualidade dos dados é importante. Este trabalho visa realizar uma filtragem dos dados de modo a melhorar a qualidade dos resultados, independentemente da área. Como foi referido pelo regulamento da União Europeia, para áreas de alto-risco como dispositivos médicos, as técnicas necessitam de ser provadas/testadas para garantir a segurança de quem as usa. Deste modo, é importante o nosso algoritmo ser "explicável" para que possa ser usado nas mais diversas áreas, uma vez que os dados usados têm de ser de boa qualidade.

A pesquisa de Dwivedi *et al.* [14] examina as técnicas de programação de ponta para XAI, delineando as suas fases no desenvolvimento de ML. Ao classificar e comparar diversas abordagens de XAI com exemplos concretos, o objetivo é orientar os intervenientes na seleção de métodos, estruturas e kits de ferramentas adequados para a implementação de XAI. De acordo com as classificações que Dwivedi *et al.* [14] o nosso *dashboard* é global, *white-box* e específico do modelo. Global porque apresenta uma análise de todo o modelo. *White-box* porque permite perceber claramente as caraterísticas que mais se destacam e é específico do modelo porque o desenvolvemos especificamente para este algoritmo. Utilizando a mesma técnica que apresentamos para a recolha de dados é possível criar *dashboards* para outros algoritmos, no entanto serão sempre específicos do modelo para o qual se está a desenvolver.

Do que é do nosso conhecimento, existem muito poucos mecanismos para aplicar XAI às

técnicas de *ensemble* FS, deste modo, este trabalho é um contributo importante devido à falta de estudo nesta área.

3 Reimplementação da biblioteca em Python

Este capítulo é focado no estudo da implementação da biblioteca existente em R, a sua reimplementação em Python e melhorias que efetuamos relativamente à versão original. O estudo da implementação em R serve para compreendermos melhor os conceitos subjacentes ao algoritmo, bem como as soluções que foram usadas para passar de um algoritmo especializado para um algoritmo mais genérico, e perceber como funciona a seleção automática de caraterísticas que este disponibiliza. Na reimplementação em Python focamo-nos em explicar as dificuldades que obtivemos ao longo do percurso devido à diferença no funcionamento das linguagens, bem como quais foram as soluções que alcançamos para ultrapassar essas diferenças.

Na secção 3.1 é efetuado o estudo da implementação em R, bem como uma compreensão do funcionamento interno das dependências que esta apresenta. O estudo é bastante importante devido ao facto das dependências calcularem as métricas, sendo o seu resultado importante para a geração dos resultados do algoritmo. Na secção 3.2 é explicada a implementação, as dificuldades e soluções para as mesmas, além da otimização da biblioteca para diminuir o tempo de execução.

3.1 Estudo e compreensão do *package* em R

Já existe uma implementação do algoritmo usado nesta tese no *package* EnsembleFeaturesRanking feito na linguagem R. Decidimos usar este como base, começando por analisar a sua estrutura e compreender o seu funcionamento. Na Figura 3.1 pode-se observar o diagrama de classes do *package*. O *package* tem como programa principal o EnsembleFeaturesRanking, tendo dependência do *package* `FSelector` [50] e os programas auxiliares. É importante realçar que este *package* já existia em R, não fazendo parte do código da implementação. Como se pode constatar no diagrama, a implementação apresenta uma grande dependência deste *package*, uma vez que as métricas usadas, ou são do mesmo ou dependem dele.

O programa principal recebe como argumentos o conjunto de dados para o qual se deseja obter o *ranking* das características, podendo também aceitar parâmetros para substituir valores padrão:

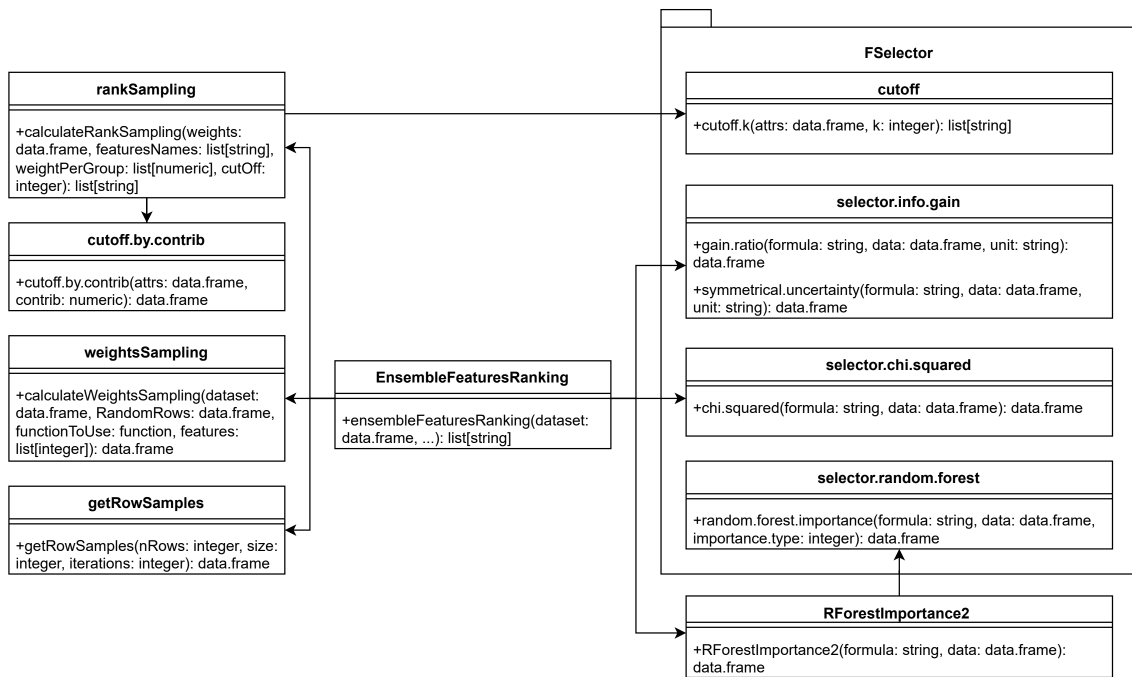


Figura 3.1: *Diagrama de classes do package EnsembleFeaturesRanking feito em R*

- *nRows*: número de instâncias em cada partição do conjunto de dados, sendo metade do número total de instâncias por padrão;
- *iterations*: número de iterações do algoritmo, estabelecido em 10 por padrão;
- *metrics*: lista de métricas a ser utilizada, incluindo por padrão *gain ratio*, *symmetrical uncertainty*, *chi squared* e *random forest importance*;
- *cutOff*: determina o corte para as características melhor classificadas, onde:
  - Se *cutOff* estiver definido como -1, a função inclui automaticamente características que contribuem para 99% do peso total.
  - Alternativamente, com um valor de *cutOff* especificado (0 para todas as características ou um inteiro positivo para as principais 'k' características).

As métricas estão dispostas no lado direito do diagrama, enquanto os programas auxiliares ocupam o lado esquerdo. Todas as métricas utilizadas neste programa têm os mesmos parâmetros fundamentais:

- *formula*: responsável por formatar o conjunto de dados para identificar a classe do mesmo;
- *data*: representando o conjunto de dados em si;
- Os demais parâmetros que essas métricas podem apresentar possuem valores padrão, não necessitando de uma passagem explícita para a métrica.

O processo tem início com a geração dos índices dos *subsets* por meio do programa auxiliar *getRowSamples*. Após a criação dos *subsets*, uma lista de pesos gerada através de uma função

não linear. O programa auxiliar *weightsSampling* é utilizado para aplicar as métricas aos *subsets*. Por fim, os resultados dessa operação são empregados para criar o *rank* por meio do programa auxiliar *rankSampling*.

Relativamente aos programas auxiliares, estes são partes do algoritmo que foram extraídas para este ser mais simples de compreender. O *getRowSamples* gera N subconjuntos de índices de linhas aleatórios para um conjunto de dados. Recebe o número total de linhas no conjunto de dados (*nRows*), o número desejado de linhas em cada subconjunto (*size*), e a quantidade de subconjuntos a serem gerados (*iterations*). O resultado é uma lista que contém N subconjuntos de índices de linhas aleatórios.

O *weightsSampling* calcula os pesos das características para cada amostra com base na métrica passada (*functionToUse*). Este recebe o conjunto de dados original (*dataset*), uma lista de índices de linhas de subconjunto (*RandomRows*), a função estatística e uma lista de nomes de características. O resultado é uma matriz N x M de pesos, onde N é o número de subconjuntos e M é o número de características.

O *rankSampling* determina a importância de cada característica para cada amostra num conjunto de dados. Recebe pesos de entrada para cada característica, obtidos a partir das várias métricas e subconjuntos de dados. Durante a execução, gera a classificação (*rank*) com os nomes das características como identificadores.

Os argumentos incluem:

- *weights*: os pesos calculados através das métricas;
- *featuresNames*: a lista de características a serem classificadas;
- *weightPerGroup*: um vetor que especifica o peso atribuído a cada classificação;
- *cutOff*: determina o corte para as características melhor classificadas, onde:
  - Se *cutOff* estiver definido como -1, a função inclui automaticamente características que contribuem para 99% do peso total, recorrendo ao uso do programa auxiliar *cutoff.by.contrib*.
  - Alternativamente, com um valor de *cutOff* especificado (0 para todas as características ou um inteiro positivo para as principais 'k' características).

Devido ao facto das métricas serem todas calculadas pelo *package* `FSelector` [50] , sentimos a necessidade de estudar os métodos que são usados para percebermos o que estes realmente fazem. O package `FSelector` [50], desenvolvido por Lars Kotthoff, é um conjunto de ferramentas em R projetadas para seleção de características em *machine learning*. Este *package* oferece ferramentas de normalização e discretização dos dados, métricas para calcular a relevância e corte de caraterísticas. Na Figura 3.2 pode-se observar o diagrama de classes deste *package* para as funções que usamos.

O *package* que queremos traduzir utiliza as métricas disponibilizadas por este *package* como possíveis métricas a usar, como tal, decidimos compreender o funcionamento destas métricas, para sabermos que caraterísticas é que precisamos que as métricas apresentem. As métricas

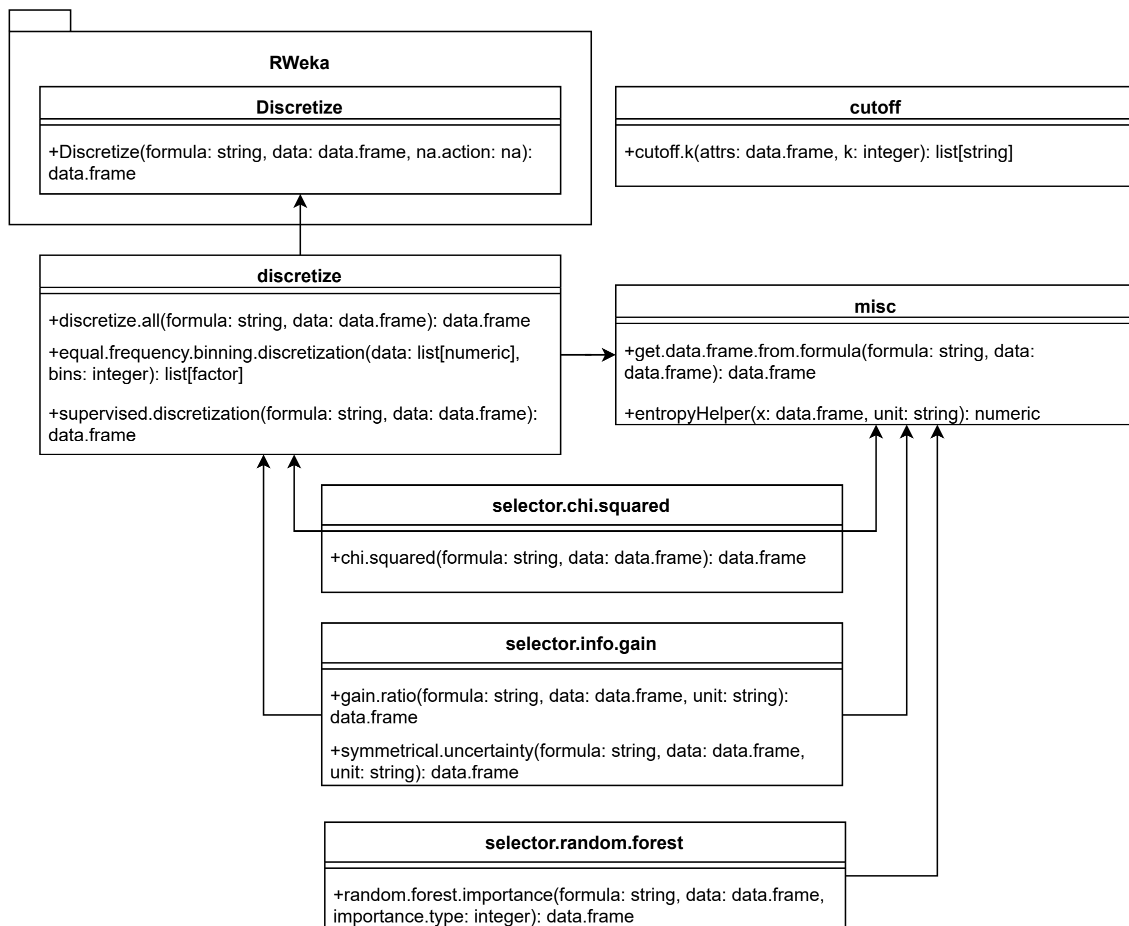


Figura 3.2: *Diagrama de classes das funções usadas no package* `FSelector`

que o algoritmo suporta são as apresentadas na Figura 3.2. Após observar o código fonte deste *package* [33], conseguimos encontrar um padrão entre as métricas que são usadas, como se pode ver no seguinte código:

```
1     new_data = get.data.frame.from.formula(formula, data)
2     new_data = discretize.all(formula, new_data)
```

Listagem 3.1: Código comum às métricas

Este código recebe um conjunto de dados passado no parâmetro data e uma formula que é usada para formatar o mesmo. A primeira linha é a que efetua a reformulação do conjunto de dados através da formula passada. No caso do *package* que queremos traduzir, este utiliza sempre a mesma formula, que resulta no seguinte formato "`classe | atributos`". Este formato é importante para a segunda linha, onde é efetuada a discretização do conjunto de dados. Excluindo este bloco apresentado, código restante é apenas o cálculo das métricas.

É importante saber que a função *get.data.frame.from.formula* remove as instâncias do conjunto de dados que apresentam valores omissos. Passando agora para a função *discretize.all*, verifica se é necessário discretizar a classe, caso seja, efetua discretização não supervisionada, dividindo a mesma em 5 classes com a mesma frequência. Depois desta operação, é efetuada uma discretização supervisionada através do algoritmo Minimum Description Length (MDL) de Fayyad e Irani [18]. Note-se que este discretizador recorre a um outro *package* chamado

`RWeka` [27], que é apenas um *wrapper* para a biblioteca `Weka` escrita em Java.

Através deste pré-processamento, as métricas são capazes de processar dados discretos e contínuos. Como se pode constatar a métrica *random.forest.importance* é a única que não necessita de discretizador devido à própria estrutura do algoritmo *random forest*.

Podemos considerar que, para o caso de uso no *package* de Emsamble Features Ranking, este tipo de métricas efetua processamento desnecessário, uma vez que apesar de se manter o conjunto de dados ao longo do cálculo das várias métricas, este efetua a discretização do mesmo cada vez que vai usar uma métrica, enquanto poderíamos simplesmente efetuar a discretização fora da função das métricas e diminuir o número de discretizações efetuadas.

A função *entropyHelper* simplifica o cálculo de entropia, construindo uma tabela de contingência e, em seguida, calcula a entropia. Esta garante que a função considera valores NA no cálculo da entropia. O resultado é o valor de entropia para a variável fornecida, conseguindo processar tanto listas como matrizes. Esta função é usada pelas métricas baseadas no ganho de informação (*gain.ratio* e *symmetrical.uncertainty*), uma vez que estas dependem da entropia conjunta.

3.2 Reimplementação da biblioteca em Python

Esta secção é focada em explicar as dificuldades e soluções para as mesmas com as quais nos deparamos durante a reimplementação. Também abordamos a reimplementação das métricas e do discretizador que estas usam, acabando esta secção com as otimizações que efetuamos para tornar a nossa implementação inicial mais eficiente, sabemos que a linguagem Python tem um desempenho pior que a linguagem R [44], caso não se recorra a nenhum tido de biblioteca para melhorar o processamento de dados numéricos, tal como o `NumPy` [64].

Na Figura 3.3 temos a estrutura da biblioteca, de modo a facilitar a compreensão das explicações e decisões que se seguem. Caso pretenda obter mais informações sobre a estrutura da biblioteca pode-se consultar os restantes diagramas de classes presentes no Apêndice A.

Começamos por procurar uma alternativa em Python para o *package* `FSelector` [50] , a qual não conseguimos encontrar, levando à implementação em Python das funções e algoritmos que foram usados no *package* EnsembleFeaturesRanking em R. O que não conseguíamos obter era a discretização embutida nas métricas, já que existem diversas bibliotecas estatísticas de disponibilizam estas métricas, mas sem a discretização.

Decidimos implementar estas métricas com discretização, para que a primeira versão da biblioteca em Python fosse fiel à implementação da mesma em R. Durante esta implementação, efetuamos uma conversão estrita do código destas métricas presente no repositório da biblioteca `FSelector` [33] para Python. Esta tarefa foi dificultada devido às diferenças da linguagem e mesmo operações que deveriam ser semelhantes entre as mesmas, tal como a entropia, funcionam de formas distintas. Este problema ainda é alargado para a falta de opções em Python para algumas operações em R.

Todas as métricas implementadas recebem como parâmetros um *DataFrame* data da biblioteca `Pandas` [45], uma *string* formula e geram uma *DataFrame* onde o índice das linhas é os atributos e a coluna é a importância do atributo.

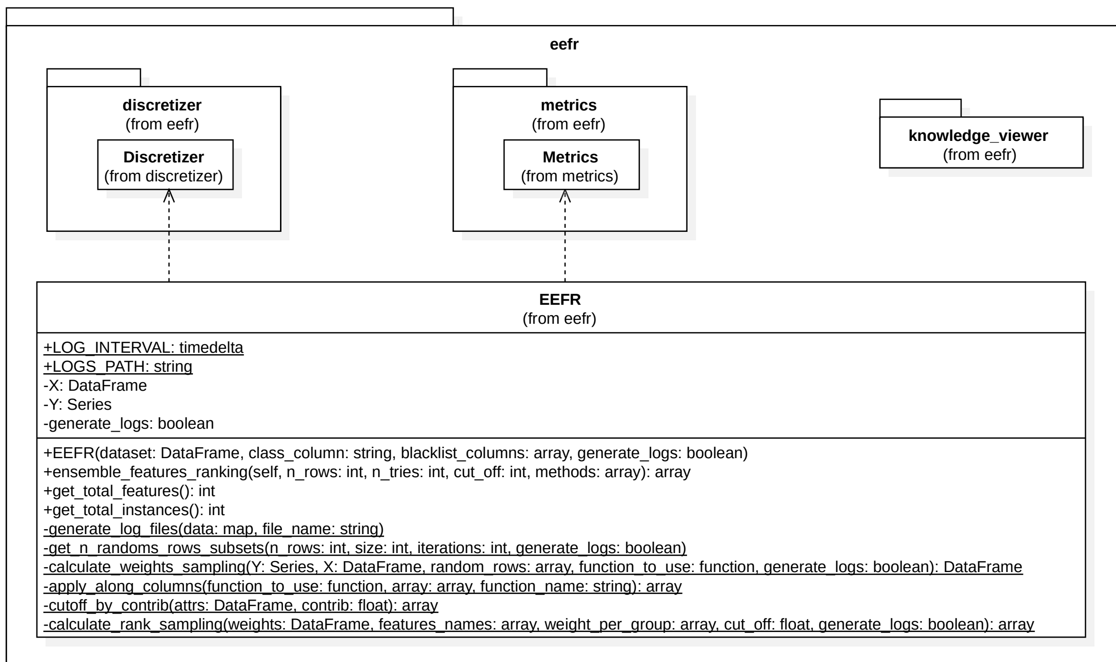


Figura 3.3: *Diagrama de classe da biblioteca*

Começamos pelas métricas baseadas no ganho de informação (*gain ratio* e *symmetrical uncertainty*). Um ponto importante na tradução foi termos de converter as listas do Python para *arrays* do `NumPy` sempre que pretendíamos usar cálculos matriciais, uma vez que as listas do Python, ao contrário das matrizes do R, não apresentam suporte nativo para cálculo matricial, sendo necessário recorrer ao uso do `NumPy` ou ao uso de ciclos, que torna mais difícil a leitura do código, para efetuar os cálculos. Para podermos nomear as colunas no resultado usamos o *DataFrame* da biblioteca `Pandas` devido à sua facilidade de utilização.

Relativamente ao cálculo da entropia, tal como em R, também criámos a função auxiliar *entropyHelper*. Em Python não existe suporte nativo para criar contagens de elementos, tal como existe o *table* em R, nem suporte nativo para o cálculo de entropia. Para não implementarmos este tipo de funções que estão presentes em bibliotecas amplamente usadas, decidimos usar a função *unique* do `NumPy`, que permite efetuar contagens e a função *entropy* do scipy stats [53], que têm implementações eficientes em linguagens de baixo nível. A função *entropyHelper* implementada efetua o cálculo da entropia sempre com logaritmo de base 10.

Depois das métricas de ganho de informação, implementamos a *chi squared*. Em R, esta apresenta um bloco de código dentro da função *sapply*, o que levou à criação de um método auxiliar em Python para refatorizar este bloco de código. Este volta a utilizar a função *table*, mas neste caso está a receber 2 listas, o que gera uma matriz de contingência.

Para gerar uma matriz de contingência a partir de 2 listas em Python começamos por concatenar estas a partir da função *column_stack* do `NumPy`, que gera uma lista de pares (tuplos). A esta lista aplicamos a função *unique* do `NumPy`, para obter a contagem dos pares, mas é necessário

transformar os valores únicos retornados em pares (tuplos). Geramos uma *Serie* do `Pandas` a partir de um dicionário criado a através destas listas e, por fim, aplicamos a função *unstack* à *Serie* para desdobrar a lista de *pivots* e contagens e gerar a matriz.

Para efetuar as contagens recorremos à função *sum* do `Pandas`, aplicando-a aos eixos respetivos. Por fim, a última grande mudança face ao código em R desta métrica é o cálculo matricial, que tal como nas outras métricas foi efetuado usando a biblioteca `NumPy`. Para esta métrica também acabamos por usar a função *transpose* do `NumPy` para transpor a matriz antes de efetuar o cálculo.

Para a métrica *random forest importance*, esta não tem a sua implementação completamente disponível no código fonte da biblioteca `FSelector`, uma vez que esta utiliza a função *randomForest* da biblioteca `RandomForest` [39]. Como tal, também acabamos por utilizar o *RandomForestRegressor* da biblioteca `Scikit-learn` [65], sendo possível que este algoritmo acabe por apresentar resultados ligeiramente diferentes comparado à implementação em R.

Por fim, criámos um enumerado com as várias métricas para estas serem passadas como um tipo concreto, neste caso um elemento do enumerado, em vez de poder ser qualquer tipo de função ou se ter de alterar o código do programa sempre que se pretenda acionar uma métrica nova. Deste modo, apenas se necessita de adicionar a métrica ao enumerado.

Tal como foi referido na secção 3.1, as métricas em R fazem uso de um discretizador, como tal, depois de criar as métricas, precisamos de resolver o problema de não termos o discretizador. A primeira maneira para discretizar que encontramos foi recorrer ao discretizador presente na biblioteca `Weka` (Python) [59], uma vez que, tal como a biblioteca `RWeka`, esta era apenas uma interface para a biblioteca escrita em Java.

Durante os testes do discretizador com *datasets* com algumas dezenas de milhar de linhas este começou a lançar erros. Relativamente a estes erros, apenas conseguimos ter a garantia que são gerados devido a problemas de memória na Java Virtual Machine (JVM), o mesmo se sucedeu com a biblioteca Python weka wrapper 3 [49]. Como consequência destes problemas, decidimos abandonar o uso de bibliotecas que fossem *wrappers* da biblioteca `Weka`, procurando por alternativas implementassem discretização supervisionada.

Testámos diversas bibliotecas que disponham desta funcionalidade, tal como a Discrust [30], que apresentava ganho de desempenho na discretização fase ao discretizador da biblioteca `RWeka`. O problema das bibliotecas usadas é que apresentavam resultados muito diferentes do esperado, sendo assim difícil de conseguir comparar os resultados do algoritmo. Para corrigir este problema decidamos procurar por bibliotecas que implementassem o MDL de Fayyad e Irani [18].

O Princípio do Minimum Description Length (MDL) é uma abordagem da teoria da informação aplicada em várias áreas. O MDL orienta a criação de representações discretas, em busca de otimizar o equilíbrio entre simplicidade e precisão. O trabalho de Fayyad e Irani aplicou o MDL de forma específica à discretização de variáveis contínuas. O objetivo foi encontrar uma representação ótima que minimizasse o comprimento da descrição necessário para capturar informações essenciais nos dados. Essa abordagem visa encontrar uma representação concisa que equilibre eficientemente a complexidade do modelo com a precisão da representação dos

dados.

Infelizmente, durante a nossa pesquisa não conseguimos encontrar nenhum algoritmo que indicasse explicitamente que aplicava este algoritmo. Decidimos realizar a implementação deste algoritmo baseando-nos no código fonte da biblioteca `Weka` (em Java) [69], uma vez que era este que efetuava a discretização na biblioteca em R.

A primeira versão acabou por ser uma versão simplificada do discretizador da biblioteca `Weka`, uma vez que este apresenta diversas opções de que não precisamos neste trabalho. Esta foi realizada sem recorrer a nenhuma biblioteca, sendo por isso completamente em Python, que apresenta um desempenho inferior à linguagem Java.

Durante os testes deste discretizador conseguimos verificar que este conseguia gerar os mesmos resultados que a biblioteca `RWeka`, 5 vezes mais lenta do que esta. Com este desempenho sabíamos que era necessário voltar a pensar no discretizador, mas decidimos continuar a implementar o resto do programa, já que uma versão funcional do programa foi considerada uma prioridade face ao seu desempenho (devido ao facto da linguagem Java ser mais rápida do que a linguagem Python sem recorrer a bibliotecas [62]).

Passando agora para a função *get_data_from_formula*, já existem várias bibliotecas que tentam trazer o sistema de formulas R para o Python, tais como o `Patsy` [58] e `formulaic` [70]. O problema destas bibliotecas é que não suportam o símbolo "`.`", que é utilizado para referir "todas as caraterísticas restantes" que é usado pela biblioteca Ensemble Features Ranking em R.

Para contornar este problema a única solução que encontramos foi efetuar processamento da lista de caraterísticas para gerar uma formula com o formato "`class ~ atr1 + ... + atrN`". Como a forma usada na biblioteca não é alterável, isto significa que era necessário a classe ser a coluna com nome "`class`", sendo que nem sempre se verifica. Deste modo acabamos por fazer uma ligeira alteração, em vez de assumirmos que a classe é a coluna "`class`", assumimos que esta é sempre a primeira coluna de modo a diminuir o processamento.

Tendo acabado de implementar as funções equivalentes às dependências que o *package* em R tinha, decidimos reimplementar a biblioteca em Python. A primeira versão da biblioteca foi a conversão mais direta que conseguimos da linguagem R para Python, usando uma abordagem *Object Oriented*. Para que isto fosse possível, a principal alteração foi transformar o EnsembleFeaturesRanking realmente numa classe, sendo necessário criar uma instância da mesma para se poder usar o método.

Já que adotamos esta abordagem, criámos o construtor da classe que recebe e guarda o conjunto de dados que se pretende usar. Deste modo não é necessário passar este como argumento ao método. Este dataset é armazenado num *DataFrame* da biblioteca `Pandas` e, tal como a versão original, este recebe o conjunto de dados diretamente, não sendo necessário que este esteja guardado em nenhum formato específico, mas tendo a limitação anteriormente referida de necessitar que a classe seja a primeira coluna do conjunto de dados.

Passando agora para o método, este passou a ser um método da classe em si, passando a ter acesso direto ao conjunto de dados armazenado na instância. Este continua a receber os restantes argumentos, tal como a versão em R, bem como a ter os mesmos valores por omissão.

Este retorna uma lista com o *ranking* dos nomes das caraterísticas devidamente ordenados.

Relativamente ao funcionamento interno, como o Python não apresenta suporte nativo para efetuar cálculos com as listas, necessitamos de recorrer a *list comprehension* para efetuar o cálculo do *weightPerGroup*, passando esta operação para um ciclo que aplica uma operação aos vários elementos. O mesmo é válido para a iteração pelas métricas que se pretende aplicar para calcular o *weightsEns*.

Enquanto na implementação em R se recorreu ao uso da função *rbind* para transformar a lista *weightsEns* em uma matriz, em Python recorremos à função *concat* da biblioteca `Pandas`, ignorando os, gerando um *DataFrame*. Por fim, retornamos o resultado da função *calculateRankSampling*, tendo-se alterado a lista com as caraterísticas. Enquanto na versão em R esta enviava todas as caraterísticas exceto a denominada "`class`", esta passou a ser todas as caraterísticas exceto a primeira.

As funções auxiliares foram criadas fora da classe, uma vez que estas não dependem do conjunto de dados para funcionar. Para implementar a geração de uma lista de números aleatórios da função *getRowSamples* recorremos ao uso da função *random.randomint* do `NumPy`, tendo de converter posteriormente o seu resultado para lista. Relativamente à concatenação das várias, decidimos simplesmente usar a função *append* e assumir que o retorno é uma lista de listas de números aleatórios.

Na função *calculateWeightsSampling* para extrair o subconjunto dos dados do *DataFrame* recorremos à função *iloc* que nos permite selecionar as caraterísticas e instâncias, tal como a indexação feita em R. Também recorremos a esta função para extrair os pesos resultantes da aplicação da métrica em lista. Para invocar a métrica continuamos a passar a formula, apesar de esta não ser usada. Para substituir a função *apply* do R neste caso em particular, recorremos ao uso da função *apply_along_axis* da bíblioteca `NumPy`, tendo selecionado para que esta fosse aplicada ao longo das caraterísticas. Por fim, usamos o resultado desta para criar o dataset com os pesos das caraterísticas.

Passando agora para a função *calculateRankSampling*, começamos por criar a matriz recorrendo ao *DataFrame* da biblioteca `Pandas`. Voltamos a usar a função *iloc* para efetuar a indexação e usamos o *T* para obter a sua transposta. Relativamente à função *cutoff.k*, esta ordena as amostras e retorna as 4 melhores, mas uma vez que esta é usada para extrair todas as amostras da lista, esta acaba apenas por ordenar as amostras. Sabendo disto, acabamos por usar a função *sort_values* da biblioteca `Pandas`. Para o outro *cutoff.k* usamos a mesma lógica, mas de seguida efetuamos indexação para retornar os primeiros `k` elementos.

Por fim, na função auxiliar *cutoff.by.contrib*, usamos o campo *shape* e campo *index* do *DataFrame* para substituir a função *dim* e *dimnames* do R, respetivamente e a função *sort_values* para ordenar. Para calcular a contribuição necessitamos de usar 2 vezes a função *sum* do *`Pandas`*, uma vez que esta só reduz uma dimensão. Por fim, voltamos a substituir a função *sapply* por *list comprehension*.

Para melhorar o desempenho da biblioteca tivemos de deixar de usar as estruturas nativas do Python, uma vez que estas apresentam um desempenho bastante reduzido quando comparado com o desempenho das estruturas disponibilizadas pelo `NumPy`, que é uma biblioteca feita em

c, uma linguagem de baixo nível que apresenta um desempenho, devido a elevada flexibilidade das mesmas, como tal, substituímos as listas do Python pelos *arrays* do NumPy e a maioria dos *DataFrames* do pandas também foram substituídos pelos *arrays* do NumPy, devido ao facto de o NumPy ter suporte nativo para operações matriciais e ser mais eficiente do que as estruturas do Pandas, devido à complexidade das mesmas.

Este procedimento foi relativamente simples, já que existem funções similares do Python e Pandas para o NumPy, tal como a função *sum*, que realiza o somatório dos valores da lista ou *sqrt* que calcula a raiz quadrada. As situações onde não se deixou de usar o Pandas foi nas métricas e para armazenar os próprios dados, uma vez que esta estrutura de dados do Pandas apresenta muito mais informação do que apenas os valores por linha e coluna. O maior desafio desta troca de estruturas de dados foi o cálculo das matrizes de contingência. No Pandas efetuava-se a contagem do conjunto de dados, seguido de 2 operações do Pandas. Por outro lado no NumPy esta operação é mais complexa. Primeiro é necessário calcular as contagens das classes e das categorias, seguida da criação de uma matriz com base no número de valores únicos retornados. Por fim, esta é preenchida com a informação das contagens.

Após estas melhorias, verificámos um aumento de desempenho de 500% na velocidade de execução em comparação com a primeira versão da biblioteca funcional (com o discretizador desenvolvido). Este ganho de desempenho é principalmente notável no discretizador, onde eram efetuadas imensas afetações de listas do Python, enquanto no NumPy estas funcionam por concatenação, o que diminui drasticamente o número de operações necessárias para conseguir juntar listas. A maioria das variáveis usadas no discretizador também eram suportadas através do Pandas, que apresenta uma indexação 100 vezes mais lenta [32] que a indexação dos *arrays* do NumPy, fazendo com que este também seja uma grande diferença quando se está a lidar com grandes volumes de dados.

Depois de melhorarmos o desempenho do algoritmo através da otimização do código, decidimos otimizar o algoritmo. Reorganizamos o código, de modo a que as operações não necessitassem de ser realizadas 2 vezes para os mesmos dados, para otimizar o tempo de execução. Algumas destas alterações necessitaram de estruturas de dados para suportar o resultado destas operações. Isto acontecia na discretização, que era efetuada para cada métrica, sendo que esta gerava sempre o mesmo resultado e como tal, poderia ser removida das métricas e passar a ser executada antes das métricas, passando as métricas a receber os dados já discretizados. Esta alteração gera uma melhoria de desempenho proporcional ao número de métricas, uma vez que as métricas recebem exatamente os mesmos dados.

Esta alteração não favorece o desempenho da métrica *Random Forest Importance*, pois é a única que consegue funcionar com dados discretos ou contínuos. No entanto, caso apenas se pretenda utilizar esta métrica, ocorre uma perda de desempenho. Isto deve-se ao facto de a discretização ser efetuada diretamente no construtor, armazenando os dados já discretizados. Consequentemente, realiza-se uma discretização desnecessária quando o algoritmo *Random Forest Importance* é utilizado.

Por outro lado, passar a discretização para o construtor aumenta o desempenho em cenários onde se pretende executar o algoritmo várias vezes, permitindo comparar os diferentes *rankings*

gerados por combinações das métricas.

4 Novas funcionalidades e "explicabilidade" do algoritmo

Neste capítulo explicamos a adição de caraterísticas que consideramos importantes para um algoritmo de filtragem deste tipo, tais como a lista negra de caraterísticas, que é extremamente útil para remover a necessidade do utilizador realizar 2 processos de filtragem, um para remover caraterísticas que são indesejadas, como identificadores que indiquem qual é a amostra, que não adicionam informação devido ao facto de serem únicos e outra para reduzir o número de caraterísticas antes de usar o conjunto de dados para treinar um classificador. A capacidade de selecionar a classe também é importante uma vez que a implementação em R força o conjunto de dados a ter uma determinada estrutura. E por fim, a adição de um programa auxiliar que nos permita ver a evolução do algoritmo, para que este não apresente os resultados como uma *black box*.

Na secção 4.1 são apresentadas as funcionalidades que adicionamos à biblioteca, tais como o *logging* para acompanhar o progresso do algoritmo, a seleção da classe em vez de esperar que o conjunto de dados tenha de cumprir determinados requisitos, o mesmo se passa com a lista negra de caraterísticas para que não seja necessário efetuar pré-processamento do conjunto de dados para remover atributos. Por fim, na secção 4.2 também é apresentado o programa auxiliar para visualizar a evolução do algoritmo.

4.1 Funcionalidades adicionadas à biblioteca

Após conseguirmos uma versão funcional da biblioteca implementamos *logging* para conseguirmos acompanhar o progresso do processamento. Este *logging* tem como objetivo mostrar a percentagem do processamento, bem como a etapa em que este está, uma vez que a versão da biblioteca em R não nos permitia acompanhar o progresso do algoritmo. Esta informação pode ser útil para impedir que alguém ponha o algoritmo a correr durante várias horas e quanto este estiver quase a terminar o aborte porque acha que este deixou de funcionar.

Para efetuar *logging* recorremos ao uso da biblioteca [21] do Python. A biblioteca Logging do Python é uma biblioteca que permite capturar e categorizar eventos durante a execução de um programa, sendo especialmente útil ao desenvolver software que exige monitorização ou relatórios de progresso abrangentes. As suas componentes principais são:

- *Logger*: permite categorizar mensagens de *log* em áreas distintas, facilitando a gestão e análise das informações. Cada instância representa uma secção do programa, com diferentes níveis de *log* para capturar mensagens conforme a gravidade. O *Logger* oferece flexibilidade ao configurar *Handlers* e *Formatters*;

- *Handler*: especifica para onde devem ser direcionadas as mensagens de *log*, como consola ou ficheiro;

- *Formatter*: determina o *layout* e conteúdo das mensagens de *log*. Permite a personalização da saída do *log*, como o formato da data/hora, nível de *log* e a própria mensagem de *log*;

- níveis de *log*: define a gravidade/importância das mensagens de *log*. Os níveis padrão de *log*, em ordem crescente de gravidade:

  - *DEBUG*: frequentemente utilizado para informações detalhadas de *debug*;
  - *INFO*: para mensagens de informação gerais;
  - *WARNING*: para possíveis problemas, como já existe no Python por omissão, como os avisos de *deprecated*;
  - *ERROR*: para mensagens de erro. Erros podem impactar uma operação, mas a aplicação pode continuar a correr;
  - *CRITICAL*: para falhas críticas que podem fazer a aplicação terminar.

  Ao definir um nível de *log* apropriado, podemos filtrar e controlar a quantidade de informações com base nas necessidades.

Efetuamos a configuração no construtor da classe para ele ficar com o aspeto presente na Figura 4.1. Decidimos adotar este formato por ser bastante compacto e mesmo assim apresentar toda a informação que consideramos importante. A data e hora são importantes caso se esteja a lidar com conjuntos de dados muito grandes, porque o algoritmo pode demorar vários dias a correr e através da data e hora é mais fácil de identificarmos a velocidade com que este está a processar. A mensagem contém o nível do *log*, uma vez que podemos querer procurar por algum tipo de evento específico, por exemplo se a aplicação apresentou algum erro durante o processamento. Depois vem a mensagem que se pretende mostrar. É importante realçar que também estamos a adicionar ao *log* erros que não são gerados pela aplicação, tais como erros de *Type mismatch*.

Para configurar este formato recorremos ao uso das funções *basicConfig*, passando o formato da mensagem e da data, e *captureWarnings* da biblioteca Logging, respetivamente. O nível de *log* que usamos por omissão é o *INFO*, necessário para apresentar as mensagens de progresso. Como se pode ver na Figura estamos a realizar *log* para indicar o processo de guardar o conjunto de dados ao iniciar a classe, bem como as informações básicas do mesmo, indicar em que parte do algoritmo é que este vai executar e mensagens de progresso.

Tirando as mensagens referentes à métricas, as mensagens de *log* não são complexas, sendo apenas necessário adicionar a mensagem na função *info* (que adiciona mensagens de nível *INFO*

```
2023-11-02 16:26:06 INFO     Dataset loaded
2023-11-02 16:26:06 INFO     The Dataset as 102294 lines and 128 columns
2023-11-02 16:26:06 INFO     Starting Weights calculation...
2023-11-02 16:26:06 INFO     Calculating weights for metric: gain ratio (1/2)
2023-11-02 16:26:37 INFO     gain ratio: Calculating... 65.0%
2023-11-02 16:26:51 INFO     Calculating weights for metric: chi squared (2/2)
2023-11-02 16:27:22 INFO     chi squared: Calculating... 15.0%
2023-11-02 16:27:54 INFO     chi squared: Calculating... 35.0%
```

Figura 4.1: *Log gerado durante o teste da primeira versão da biblioteca*

ao *log*) na linha de código antes de iniciar cada parte. Relativamente ao *log* com informação de progresso nas métricas foi necessário uma solução um pouco mais complexa, uma vez que a implementação recorria ao uso de uma função do `NumPy` para aplicar a função aos vários valores, não conseguindo assim guardar estado de forma simples. Para resolver este problemas criámos a função auxiliar *apply_along_columns*, que permite a iteração do conjunto de dados pelas colunas, tal como usávamos a função do `NumPy`, mas guardando informação de estado sobre quando deve de ocorrer a próxima mensagem de *log* para apresentar o progresso. Esta função recebe como argumento a função a aplicar (métrica), o conjunto de dados a que esta deve ser aplicada e o o nome da métrica que deve de aparecer na mensagem de *log*. Esta função começa por calcular quando deve escrever a próxima mensagem de *log*. De seguida criámos uma lista para guardar o resultado da métrica, que é preenchida, recorrendo a um ciclo, que aplica a métrica ao conjunto de dados. Dentro deste ciclo verificamos se o tempo atual é superior ao próximo tempo de *log*, se for, este escreve a mensagem no *log* e volta a calcular o próximo tempo de *log*. Para calcular o progresso da aplicação da métrica efetuamos um cálculo simples que simplesmente divide o índice atual pelo número de linhas do conjunto de dados. O mesmo raciocínio também foi aplicado ao cálculo do *rank* final, uma vez que este também é computacionalmente intensivo.

Já que efetuamos um pré-processamento (discretização) no construtor, decidimos adicionar mais algumas opções de pré-processamento que consideramos úteis num algoritmo deste tipo, sendo estes:

- Seleção da classe: tanto nesta versão quanto na versão em R o algoritmo apenas funciona se a classe estiver com o nome ou posição que o algoritmo considera como classe, não funcionando corretamente caso esta esteja noutra posição ou com outro nome;
- lista negra de caraterísticas: por vezes os conjuntos de dados têm uma coluna que indica o número da amostra (identificador). Este acaba por ser uma caraterística excelente para identificar a classe, já que todos os valores vão ter apenas um valor da classe associada. Para evitar que estas apareçam no *rank*, é necessário efetuar um pré-processamento para remover estas caraterísticas do conjunto de dados a processar.

A seleção da classe é efetuada passando ao construtor o nome da caraterística que deve de ser usada como classe, fornecendo deste modo uma maneira simples de conseguir indicar a classe. Caso esta não seja indicada, tem como valor por omissão a primeira caraterística. Como

a posição da classe passou a ser incerta no conjunto de dados, decidimos passar a dividir o conjunto de dados em `X` e `Y`, sendo `X` as caraterísticas e `Y` a classe. Para implementar esta separação recorremos ao uso da indexação através do nome da caraterística para obter o `Y` e usamos a função *drop* para obter o conjunto de dados sem as caraterísticas usadas. Ambos os recursos são disponibilizados no conjunto de dados no formato *DataFrame* da biblioteca `Pandas`.

A lista negra é uma lista passada ao construtor, caso esta não seja passada, tem como valor por omissão uma lista vazia. Para remover as caraterísticas da lista negra do conjunto de dados recorremos novamente ao uso da função *drop* e guardando o seu resultado no campo `X` da classe. Ao adicionar a lista negra ao construtor diminuímos o processamento desnecessário de discretizar caraterísticas que não são úteis e também permite remover caraterísticas que funcionem como identificadores sem precisar de recorrer a um pré processamento antes de executar o algoritmo.

Para que seja possível compreender os resultados que o algoritmo gera decidimos criar um programa auxiliar que permita visualizar a evolução do algoritmo. Este programa auxiliar permite-nos perceber de onde é que este obtém o conhecimento, permitindo que se perceba como é que este chegou ao resultado apresentado. este tipo de comportamento é conhecido como explicabilidade do algoritmo, que é um fator cada vez mais importante com o crescimento da inteligência artificial, uma vez que *chat bots* como o chatGPT são algo cada vez mais usados e que parte deles funciona como *black box*. Este tipo de comportamento é problemático, uma vez que não conseguimos perceber o motivo pelo qual o algoritmo nos apresenta aquele resultado, não permitindo que se identifique se este cometeu um erro *a priori*.

Para que este programa auxiliar consiga apresentar a evolução do algoritmo necessitamos de guardar mais informação durante a execução do algoritmo. A primeira coisa que consideramos importante guardar foi os tempos de execução, uma vez que isto ajuda-nos a perceber em que parte é que este algoritmo está a consumir mais tempo. Esta informação além de ser útil para apresentar os tempos de execução no programa auxiliar, também é útil para identificar possíveis partes do algoritmo que podemos otimizar. Além dos tempos de execução também consideramos importante guardar o resultado dos principais métodos que compõem o algoritmo, tal como a execução das várias métricas, onde esta informação pode ser usada para percebermos se uma dada caraterística é considerada relevante por várias métricas ou apenas por uma, estes resultados também nos permitem fazer um *rank* para cada métrica, que facilita a leitura dos resultados das métricas. Para determinados métodos também é relevante apresentar os parâmetros passados ao mesmo, tal como na geração dos subconjuntos, onde os parâmetros passados a este podem alterar drasticamente o resultado, o qual também não é algo interessante de apresentar ao utilizador (sequência de linhas que compõem cada subconjunto).

Conforme aprimorávamos a informação a apresentar ao utilizador surgiu outra questão importante: como é que vamos guardar esta informação de modo a ser fácil de processar para mostrar ao utilizador. Começamos por guardar a informação recorrendo à biblioteca Logging que permite a criação de vários *logs*, suportando escrita para a consola e para ficheiros, deste modo podíamos configurar o formato do *log* apresentar o tempo em que escreveu a linha, o que aparenta ser uma vantagem, diminuindo a quantidade de informação com a qual nos

tínhamos de preocupar. Esta mesma vantagem acabou por se tornar um problema, uma vez que era necessário processamento para generalizar a leitura do ficheiro, uma vez que tínhamos de considerar as linhas do mesmo para o cálculo do tempo de execução, tínhamos de ter em atenção se a linha realmente era gerada pelo *log* ou era da informação que tinha uma quebra de linha, tal como acontecia quando se guarda uma tabela. As próprias tabelas não eram guardadas num formato muito apropriado.

Para resolver estes problemas, começamos por guardar as tabelas diretamente a partir do `Pandas`, que nos permite configurar em que formato pretendemos guardar e ler as mesmas em ficheiros. Ao passar a guardar as tabelas de forma independente da restante informação que se pretendia guardar para aquele método conseguimos resolver o problema das linhas desformatadas, mas passamos a ter a informação resultante da execução do método espalhada por dois ficheiros, além do facto de que normalmente a tabela era o resultado do mesmo, ou seja, precisávamos de adicionar uma mensagem ao *log* à mesma para sabermos o tempo de execução do método, aumentando as escritas necessárias (o que considerando o tempo de execução dos métodos é irrelevante). Mesmo após a remoção das tabelas dos ficheiros de *log* estes continuavam a parecer complexos de interpretar, devido ao processamento necessário para calcularmos o tempo de execução, deste modo, acabamos por abandonar a ideia de guardar a informação usando a biblioteca Logging e passamos a guardar a informação em tabelas de modo a aproveitar o suporte do `Pandas`.

Com uma visão geral da informação e o formato em pretendíamos a guardar começamos a efetuar a implementação deste "*log*" que vai ser usado pelo programa auxiliar de visualização. Começamos por guardar a informação que conseguíamos recolher apenas dentro do método principal deste algoritmo e no construtor da classe. Como já não tínhamos os tempos do *log* necessitámos de outra forma de calcular o tempo de execução, para isso decidimos criar um dicionário onde o a chave era o nome da operação e o valor o tempo em que esta era chamada, como tal, podia-se aceder ao dicionário no fim da execução da operação, obtendo o tempo de início para calcular o tempo de execução. Após calcular o tempo de execução da operação, criámos uma tabela com as informações que pretendemos guardar e depois guardamos a mesma na diretoria de *logs* através da biblioteca pandas, guardando as tabelas no formato TSV. A nomenclatura que usamos para guardar as tabelas foi dar o nome do método ao qual estas são referentes. Caso o resultado do método também seja guardado numa tabela, então estas apresentam nomenclaturas especiais.

Passando agora para uma visão mais detalhada, começando pelo construtor, foi necessário efetuar mais algumas alterações, uma vez que não se estava a considerar a distribuição da classe, no entanto esta estatística é relevante, motivo pelo qual a passamos a calcular para guardar no ficheiro de *log*, juntamente com as classes existentes e o tempo de execução da discretização. Esta informação é guardada no *log* relativo à discretização. As outras informações importantes do construtor, tais como as dimensões do conjunto de dados, a lista negra de caraterísticas, a classe(nome da coluna) e tempo de execução são guardados no *log* do construtor.

No método que implementa o algoritmo guardamos o seu tempo de execução e *logs* para as 3 etapas principais do mesmo: gerar os subconjuntos, cálculos das métricas e o cálculo do *rank*. Em todos estes algoritmos, registámos o respetivo tempo de execução, apesar de cada

Tabela 4.1: Informação de *log* guardada em cada parte do algoritmo

Região	Tempo de Execução	Parâmetros de Entrada	Saída da Região	Outras Informações
Pré processamento	X	X		
Discretizer	X			Distribuição da classe
EnsembleFeature Ranking	X			
GetNRandomRows Subsets	X	X		
CalculateWeights Sampling	X	apenas as métricas	ficheiro separado	
Metrics	X		ficheiro separado	gera ficheiros separados para cada métrica
CalculateRank Sampling	X		ficheiro separado	guarda o *rank* e os valores usados para o gerar

um possuir detalhes de implementação distintos. No caso do cálculo dos subconjuntos o seu resultado não é fácil de interpretar pelo ser humano. No entanto, conseguimos ter uma ideia de como vão os conjuntos a partir dos parâmetros que são passados para gerar os mesmos. Deste modo, em vez de guardar o resultado produzido pelo algoritmo, acabamos por guardar os parâmetros do mesmo. No caso das métricas, é provável que se pretenda usar duas ou mais, tornando difícil a compreensão da informação por parte do utilizador ao ser mostrada toda junta, como tal, esta informação está dividida em vários ficheiros. O ficheiro principal, referente à execução das métricas que apresenta o tempo total de execução de métricas e quais as métricas que foram usadas. Para a informação das métricas guarda o seu tempo de execução e num ficheiro com com o prefixo "*metric*" guarda o resultado da sua execução (esta operação é efetuada dentro do método que invoca as métricas).

Para o cálculo do *rank* decidimos guardar o número total de caraterísticas do conjunto de dados e o número de caraterísticas selecionadas, juntamente com a lista de caraterísticas e o tempo de execução. Guardar o número total de caraterísticas é importante para conseguir perceber a redução que o algoritmo gerou. Apesar da lista de resultados ser a única coisa que o algoritmo retorna, esta não é informativa o suficiente para conseguir-mos perceber a evolução, porque não conseguimos perceber a distância entre as caraterísticas do *rank*. Para conseguirmos adicionar mais informação a esta secção necessitámos de modificar o método que gera o *rank* para que este formate os dados e os guarde no ficheiro "rank_sampling". Neste ficheiro são guardados os resultados do algoritmo, juntamente com os valores que o levaram a efetuar esta seleção. Deste modo conseguimos criar diagramas que demonstrem melhor a contribuição das caraterísticas.

4.2 Programa auxiliar de visualização

Este programa auxiliar tem como objetivo apresentar a informação guardada nos ficheiros de *log* de forma organizada e estruturada para que seja fácil de perceber a evolução do algoritmo. Para que a este programa seja útil para quem usa o algoritmo este deve de apresentar os dados num formato que seja fácil de interpretar pelo ser humano, para isso, este deve de apresentar uma interface gráfica fácil de entender e utilizar. Começamos por efetuar pesquisa sobre possíveis bibliotecas para criar interfaces gráficas em Python. Numa pequena pesquisa encontramos com as bibliotecas `TKinter` [22], `WxPython` [66] e `PyQt` [42], que funcionam em Windows, macOS e Linux [13].

Como ainda não tínhamos informação suficiente para decidir que biblioteca usar, já que estas 3 não apresentavam nenhuma limitação a nível de sistema operativo, decidimos procurar mais informação sobre as mesmas. Durante esta pesquisa reparamos que a interface da biblioteca `TKinter` não é muito apelativa, apresentando um aspeto antiquado, motivo pelo qual acabamos por a descartar. Acabamos por decidir usar a biblioteca `PyQt`, que é uma biblioteca que apresenta mais funcionalidades [43].

Como não conhecíamos a biblioteca necessitados de estudar o funcionamento da mesma. Começamos por criar uma aplicação simples apenas para apresentar um "*Hello World*". Durante o estudo da biblioteca acabamos por descobrir que esta apresentava parte das funcionalidades que pretendíamos. A classe *QLabel* para conseguirmos apresentar texto, várias classes para definir o *layout* da aplicação, a classe *QPushButton* que nos permite criar botões e também apresenta a classe *QTableWidget* que nos permite criar tabelas na aplicação, no entanto é necessário configurar a tabela, uma vez que esta não é compatível com as tabelas da biblioteca `Pandas`.

Com estas classes conseguimos criar uma aplicação simples, no entanto precisávamos de uma maneira para conseguir apresentar mais do que uma página na aplicação. Para isso recorremos ao uso da classe *QStackedWidget* que permite adicionar várias "páginas" à aplicação. O termo "páginas" é usado com aspas devido ao facto de estas serem implementadas como *QWidget* e querermos apresentar como semelhança às páginas web. A classe *QStackedWidget* permite que existam várias "páginas" onde só uma pode ser visto de cada vez, apresentando métodos para se poder alterar a "páginas" que é apresentada.

Após termos todo o conhecimento base necessário, começamos a criar uma hierarquia de classes para suportar o funcionamento da nossa aplicação. Consideramos que cada "página" apresenta informação de apenas uma região do código, tendo em conta o funcionamento do algoritmo. Começamos por criar a classe *BasePage* que estende de *QWidget* e é composta por uma *QLabel* para apresentar o nome da secção do código e um *QVBoxLayout* que é um *layout* vertical no qual se deve de colocar as várias componentes que pretendemos apresentar na "páginas", incluindo a própria *QLabel*. A classe *ChildPage* que estende de *BasePage* e acrescenta um *QPushButton*, que serve para navegar para para a "página pai".

Com estas duas classes geramos a hierarquia entre as páginas que pretendíamos, usando botões para navegar entre elas, no entanto, não conseguimos um *layout* que consegui-se expressar a ordem pela qual as operações ocorriam dentro do algoritmo (Figura A.5). Apesar de estarmos

a usar a *QTableWidget* para apresentar a informação, esta também nos estava a gerar código repetido, tendo em conta que obtínhamos os dados através do *DataFrame* da biblioteca `Pandas`. Procuramos uma maneira de usar o o *DataFrame* diretamente na tabela, deste modo poderíamos generalizar o uso da tabela. Durante a procura encontramos a solução [17], que é um modelo de tabela que pode ser usado como modelo da classe *QTableView*. Com esta combinação conseguimos substituir a apresentação de tabelas para uma solução mais genérica e simples.

Não estávamos satisfeitos com a apresentação que tínhamos das componentes do algoritmo, então decidimos passar a apresentar estas no formato de um diagrama de blocos, deste modo é mais fácil de perceber a ordem pela qual as operações acontecem no algoritmo. A biblioteca `PyQt` já apresenta classes que funcionam como uma folha, onde é possível adicionar várias figuras à mesma. Também apresenta classes de figuras como uma linha reta, retângulo e polígonos. Começamos por usar a classe *QGraphicsScene* para poder desenhar as figuras e a classe *QGraphicsView* que é necessária para visualizar a classe anterior, a classe *QGraphics-RectItem* para criar os blocos e a classe *QGraphicaLineItem* para interligar os blocos. Apesar de conseguirmos obter um resultado similar ao que pretendíamos, ainda faltavam detalhes que consideramos importantes, tais como o nome da operação dentro do botão e a orientação existente entre as operações.

Tendo isto em conta, decidimos criar a nossa própria implementação do diagrama de blocos, começando por criar a classe *ClassBlock*, que estende da classe *QGraphicsRectItem*, mantendo toda a parte do retângulo e adicionamos um texto (classe *QGraphicsTextItem*), para o qual calculamos a posição para este ficar centrado no retângulo. Com esta classe já conseguimos identificar a operação que o bloco representa. De seguida criámos a classe *Arrow* que estende da classe *QGraphicsLineItem*, à qual se acrescenta a ponta da seta, através da classe *QGraphics-PolygonItem*. Esta classe recebe os 2 blocos que esta ligar, sendo a ponta da seta orientada ao segundo bloco. É necessário identificar a direção e o sentido que a seta deve ter, e a partir destes calcular a ponta da seta.

Com estas duas classes já conseguimos criar diagramas de blocos, mas estes são puramente visuais. Para que se consiga ter blocos que nos redirecionem para a "página" que representa o bloco acabamos por criar a classe *ButtonClassBlock*, que estende da classe *ClassBlock*, acrescentando-lhe o comportamento do botão, ou seja, ao sobrevoar este muda de cor e executa uma determinada função (que necessita ser injetada) quando este é premido.

Como pretendíamos criar mais do que um diagrama de blocos, acabamos por criar a *Block-DiagramScene*, que estende de *QGraphicsScene* (em anexo, na Figura A.4 podemos observar o diagrama de classe resultante). Criámos esta classe para conseguir redefinir o tamanho da cena, conseguir injetar as funções que pretendíamos para os blocos e principalmente para a função de correção do tamanho da fonte, uma vez que definir o tamanho de fonte correto para os blocos independentemente não nos garante que todos fiquem com a mesma fonte, deste modo, este método verifica o tamanho máximo de posteriormente define este para todos os blocos. Também acabamos por criar a classe *BlockDiagramView*, que estende da classe *QGraphicsView*, que guarda uma instância da cena e para alterar o método de ajuste da página para este alterar a cena. Aproveitamos que criámos esta classe para permitir que se injete

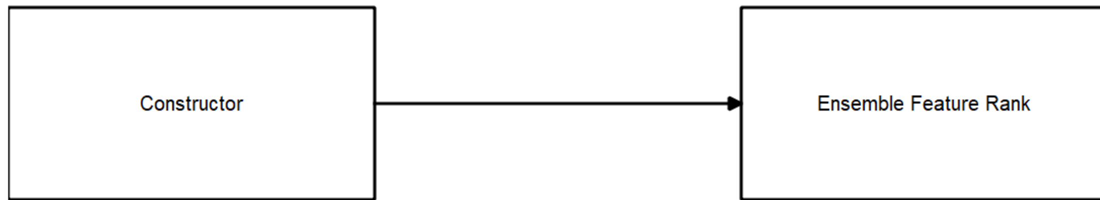


Figura 4.2: *Exemplo de diagrama de blocos usando as classes desenvolvidas para a aplicação auxiliar de visualização da evolução do algoritmo*

as funções dos botões a partir da mesma, assim não precisamos de guardar tantas variáveis para conseguir configurar corretamente os vários diagramas de blocos. Na 4.2 pode-se observar o aspeto com que ficou o diagrama de blocos gerado pelas classes desenvolvidas. Também acabamos por criar as classes *BasePageWithBlockDiagram* e *ChildPageWithBlockDiagram* que estendem das classes de hierarquia anterior, uma vez que estas apenas acrescentam a configuração do diagrama de blocos às classes da hierarquia.

De momento já temos uma boa maneira de apresentar as operações do algoritmo, através dos diagramas de blocos. Também já temos um bom formato das páginas e um bom formato para apresentar a maioria da informação das operações do algoritmo através de tabelas. Mas nesta solução é difícil de entender a informação gerada por estas operações, tal como as métricas ou o *rank* final. Para apresentarmos esta informação de uma maneira mais fácil de entender, optámos por apresentar apenas parte da mesma, através de gráficos. Decidimos usar a biblioteca `MatPlotLib` [63] para desenhar os gráficos de barras devido ao facto de já termos utilizado esta anteriormente e devido ao facto de esta integrar com a biblioteca PiQt [19].

Começamos por fazer gráficos de barras para apresentar as métricas, recorrendo ao uso da função *bar* da biblioteca `MatPlotLib`. Para gerar os gráficos começamos por carregar a tabela resultante da aplicação da métrica ao conjunto de dados, ordenar as caraterísticas por ordem decrescente da métrica e por fim selecionando as 10 primeiras. Os gráficos apresentam as métricas no eixo do x e os valores no eixo do y. Como por omissão este acaba por usar cores diferentes para as colunas acabamos por ter de configurar a cor (azul), para que este não apresente nenhum tipo de conotação ou cause algum tipo de mal intendido na interpretação.

A execução do algoritmo apresenta várias iterações no cálculo das métricas, no entanto, no gráfico apenas conseguimos mostrar a informação de uma iteração. Para resolver este problema adicionamos um título ao gráfico para o utilizador conseguir perceber em que iteração se encontra e também tivemos de adicionar 2 botões para que o utilizador consiga navegar pelas iterações (botões *next* e *previous*). Apenas com os gráficos das iterações é difícil ter uma perceção de quais as caraterísticas mais relevantes. Para isso passamos a apresentar 2 gráficos por iteração: o gráfico cumulativo até à iteração atual e o gráfico da iteração (Figura 4.3b).

Apesar de ser muito mais simples ver a informação a partir dos gráficos ainda não estávamos contentes com esta interface, uma vez que era complicado acompanhar uma dada caraterística ao longo das iterações, devido ao facto de estas oscilarem de posição e por vezes nem aparecerem nas melhores. Para resolver este problema decidimos adicionar a possibilidade

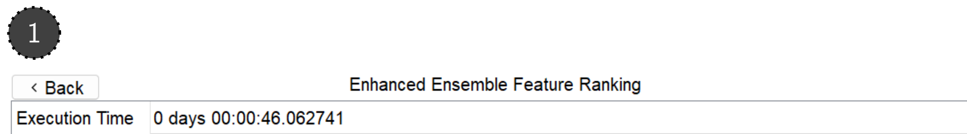
do utilizador selecionar a caraterística no gráfico para esta ser destacada, recorrendo à biblioteca `mplcursors` [37]. Esta biblioteca permite a interação com os gráficos da biblioteca `MatPlotLib`, através da criação de um cursor. Esta apresenta um cursor por omissão que apresenta a informação da barra de forma precisa, a qual consideramos relevante, e o qual deixamos ficar para nos tirar possíveis duvidas sobre os valores apresentados. Também criámos um cursor que nos permite selecionar uma caraterística e através de uma pequena alteração na geração de cores (caraterística selecionada a verde) para o gráfico nos permite mostrar esta seleção ao longo das várias iterações.

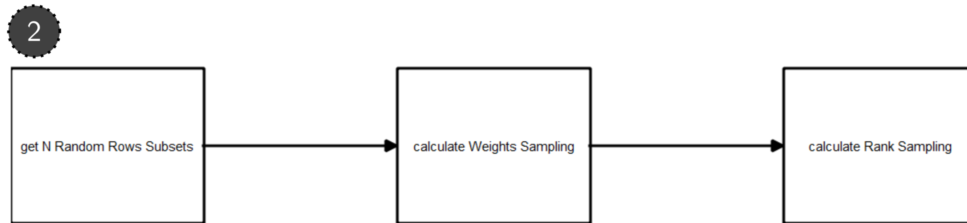
Com uma versão já refinada dos gráficos e considerando que a seleção de uma coluna visa observar a sua evolução, implementámos uma funcionalidade para animar as iterações. Esta opção permite avançar automaticamente pelas iterações até alcançar o resultado final. Para isso adicionamos 2 botões (classe *QPushButton*): botão resumir/pausar e botão de parar (remover seleção e voltar ao inicio). Para efetuar a animação recorremos ao uso da classe *QTimer*, demorando 1 segundo para avançar para o próximo gráfico. Também acabamos por adicionar a opção para selecionar o número de caraterísticas a presentar, através da classe *QSpinBox*, já que por vezes é possível que uma coluna nunca apareça nas 10 melhores das iterações, mas como é consistente acabe por ocupar uma posição no gráfico cumulativo. Esta seleção permite-nos atingir um máximo de 100 colunas, mas a partir de 50 deixa de ser possível ver os nomes das caraterísticas no eixo do y, para aumentar a legibilidade do gráfico, tendo em conta que conseguimos saber a caraterística sobrevoando a barra.

Os gráficos das métricas estão limitados entre 0 e 1 para as iterações e entre 0 e o número de iterações para os gráficos cumulativos e é possível saber em que iteração se encontra através do título dos gráficos. As métricas não guardam estado, ou seja, quando acedemos à página apresentam sempre o estado inicial. Para a página da última operação realizada no algoritmo o gráfico também apresenta a funcionalidade de seleção de colunas e de mostrar informação da coluna quando esta é sobrevoada. Para este gráficos valores estão limitados entro o valor mais alto e mais baixo das caraterísticas apresentadas. Na Figura 4.3b podemos ver os gráficos gerados, juntamente com todas as funcionalidades que desenvolvemos.

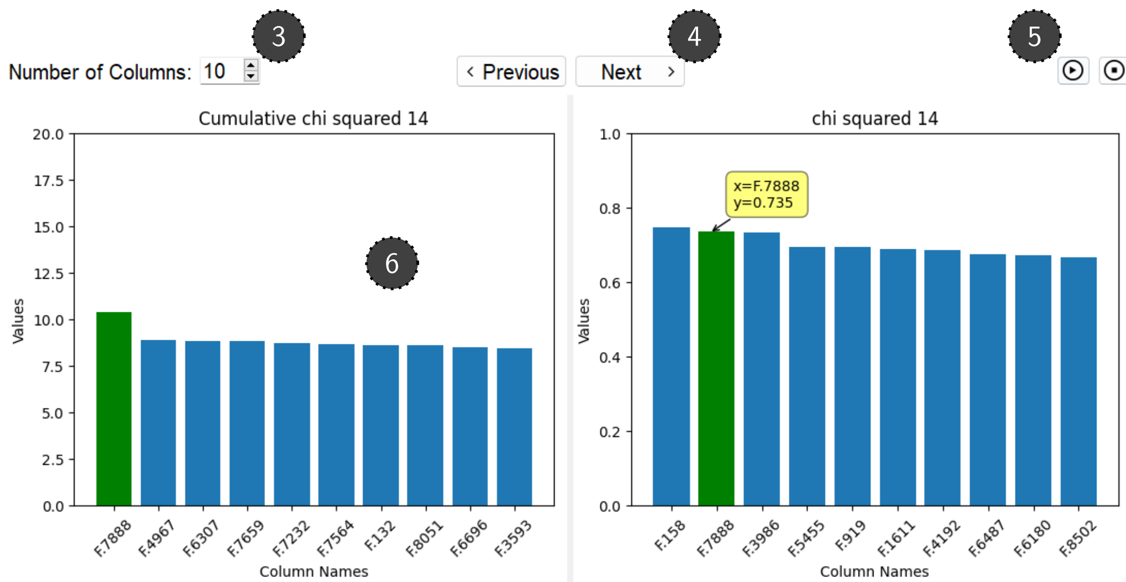
A Figura 4.3a ilustra um exemplo da interface da aplicação e dos painéis de controlo. Diagramas de blocos foram escolhidos para representar visualmente a estrutura e sequência das várias secções do algoritmo (Figura 4.3a.❷). As setas no diagrama elucidam a sequência de execução do algoritmo, enquanto os blocos funcionam como botões clicáveis que permitem aceder a informações detalhadas sobre cada secção. Nos casos em que a informação detalhada é apresentada sem uma ordem de execução específica, como acontece com as métricas, os botões sem setas indicam a ausência de interconexão entre as secções (Figura 4.3b).

A navegação na hierarquia da aplicação deve ser intuitiva e simples. Para alcançar isto, incorporámos botões de retorno para regressar ao nível anterior da hierarquia, com a navegação para a frente facilitada pelos blocos dos diagramas e botões de página (Figura 4.3a.❶). Mantendo um layout amigável, o botão de retorno está consistentemente posicionado no canto superior esquerdo, alinhado com as convenções da maioria das aplicações e navegadores. O foco principal permanece na informação apresentada.





(a) *Diagrama de blocos da aplicação*



(b) *Dashboard que mostra a relevância das caraterísticas*

Figura 4.3: *Páginas ilustrativas da aplicação. A interface mostra os blocos do diagrama do algoritmo, permitindo um raciocínio sequencial (2). Ao clicar num dos blocos, tem-se acesso a um painel com informações mais detalhadas e contextuais, neste caso, para os diagramas gerados para as métricas (6).*

Os dados densos gerados pelas secções do algoritmo e pelas métricas muitas vezes apresentam desafios de compreensão. Para resolver isto, desenvolvemos diagramas visuais que mostram os resultados de uma forma mais acessível. Para facilitar a navegação entre várias iterações, introduzimos botões interativos e ativámos a seleção de colunas (Figura 4.3b), permitindo aos utilizadores focarem-se em características específicas entre os diferentes diagramas. Adicionalmente, um diagrama comutativo foi integrado para oferecer uma visão holística da evolução das características ao longo das iterações, complementado por uma opção para modificar o número de colunas do diagrama (Figura 4.3b.❸). Os utilizadores podem navegar pelas iterações manualmente (Figura 4.3b.❹). Em cenários sem iterações, os diagramas exibem apenas o número de colunas selecionadas. A escolha da paleta de cores é crucial, uma vez que as cores podem influenciar as reações dos utilizadores. Uma animação orienta os utilizadores ao selecionar uma

Tabela 4.2: Caraterísticas das "páginas" da aplicação auxiliar para visualizar a evolução do algoritmo com descrição de caraterísticas especiais dos gráficos

Página	Diagrama de Blocos	Tabela	Gráfico	Descrição
MainPage	●			Página inicial da aplicação
ConstructorPage		●		Apresenta informação simples do construtor
DiscretizerPage		●	●	Apresenta o tempo de execução e um gráfico com a distribuição da classe
EnsembleFeature RankingPage	●	●		Apresenta informação geral do algoritmo
GetNRandomRows SubsetsPage		●		Apresenta informação sobre as partições geradas para o algoritmo
CalculateWeights SamplingPage		●		Apresenta informação geral das métricas e permite navegar para as mesmas
MetricPage		●	●	Apresenta informação sobre a métrica, o gráfico da iteração e o cumulativo com seleção de coluna e número de colunas
CalculateRank SamplingPage		●	●	Apresenta informação sobre o *rank* e o gráfico do *rank* com seleção do número de colunas

coluna, facilitando o processo de navegação (Figura 4.3b.❺). Os diagramas gerados nas iterações são o diagrama cumulativo até à enésima iteração, na esquerda, e o diagrama da enésima iteração, na direita ((Figura 4.3b.❻). A escala do diagrama cumulativo é é entre o mínimo e o máximo possível para o número total de iterações ($valor\ mínimo \times tota\ de\ iterações$ e $valor\ máximo \times total\ de\ iterações$).

Na tabela 4.2 podemos observar todas as classes que foram criadas para gerar as "páginas". De todas as classes da tabela, apenas a *MetricPage* é que gera mais do que uma página, uma vez que esta é usada para todas as métricas. Caso se tenha alguma funcionalidade específica que não dê para identificar apenas pelas demarcações na tabela, estas encontram-se descritas na descrição. No caso da classe *MetricPage*, esta recebe como um dos parâmetros o nome, que é usado para carregar o nome do ficheiro onde foi guardado o resultado na métrica e deste modo conseguimos criar dinamicamente as várias métricas.

Todas as classes que criámos recebem parâmetros para conseguir definir a fonte, a margem e, nos casos em que existe diagramas de blocos, estas também são configuráveis, para que todas as instâncias usadas para criar a aplicação apresentem o mesmo visual. Todo o código desenvolvido para a interface visual encontra-se na diretoria *KnowledgeViewer*, existindo sub diretorias para as "páginas" e para os diagramas de blocos. Dentro destas também existe a subdiretoria de

implementações. Relativamente às implementações da tabela, estas encontram-se na diretoria raiz do programa auxiliar.

5 Transformar em biblioteca

Este capítulo explica o processo de transformar o código do algoritmo e programa auxiliar de visualização em uma biblioteca Python disponível no repositório principal Python Package Index (PyPI) [9]. Durante este capítulo explicamos as decisões tomadas e as dificuldades que obtivemos ao longo do percurso. Entre estas dificuldades estão a generalização do código para não ser dependente da diretoria onde se executa, suporte para ser independente do sistema operativo, documentação e suporte dos *icons* da aplicação auxiliar de visualização.

Para transformar o algoritmo numa biblioteca, este deve de apresentar algumas caraterísticas adicionais, tais como uma diretoria para guardar os ficheiros temporários independente do sistema operativo, documentação, entre outros.

Em Python existem bibliotecas que conseguem criar a documentação a partir do código, tal como a biblioteca `Sphinx` [11], que é usada em Python, no Linux Kernel e Project Jupyter. Como esta biblioteca é usada em projetos que conhecemos bem, decidimos usá-la para criar a documentação. Para usar esta biblioteca seguimos o guia que esta disponibilizado em [24]. Após executar os passos indicados no guia [24], começamos a efetuar pequenas alterações ao ficheiro de configurações da documentação (`source/conf.py`), começando por alterar as informações do projeto (autor, versão, . . .).

Optamos por utilizar uma extensão que gera a documentação automaticamente. Para tal, acabamos por utilizar a extensão autoapi [60] que gera a documentação diretamente a partir do código fonte da biblioteca. Este apenas necessita de adicionar o código presente na Listagem 5.1. Com o uso do autoapi apenas necessitamos de a página do índice da documentação, que foi gerado com base no *template* presente na Listagem 5.2.

Os conteúdos que se encontram entre aspas são *placeholders* para os valores que se pretendem usar. Na linha 1, é o titulo da página, na linha 4 a descrição da mesma. Na linha 6 é a indicação de um bloco de código da linguagem indicada, seguido do código indentado. Por fim, o código da linha 10 a 15 é responsável por gerar os *links* do índice e de pesquisa. Como a maior parte do código (da biblioteca) já se encontrava comentado, apenas se efetuou ligeiras alterações à formatação dos comentários para que estes fossem interpretados pela biblioteca. Ao compilar os ficheiros de acordo com o guia [24], a biblioteca gera a documentação.

```
extensions = ['autoapi.extension']      # indicar o uso do autoapi

autoapi_dirs = ['../../src/xefr4py']    # indicar localização do codigo
```

Listagem 5.1: Adição ao ficheiro de configuração para usar autoapi

```
"Title"
================================

"Small description"

.. code-block:: "language"
    "code"

Indices and tables
================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Listagem 5.2: Estrutura dos ficheiro de índice da documentação

Passando agora para o detalhe do sistema de ficheiros, atualmente este está a guardar os ficheiros na diretoria onde a aplicação é executada, mas as bibliotecas costumam ter diretorias fixas para guardar os seus ficheiros temporários, independente do sistema operativo. Efetuando uma pequena pesquisa, conseguimos encontrar uma solução simples capaz de resolver este problema [1] através da verificação do sistema operativo e utilização das pastas de ficheiros de programas referentes a cada um dos sistemas. Deste modo os *logs* da biblioteca são guardados na diretoria `xefr4py_logs`, na diretoria raiz definida com base no sistema operativo.

Com este código podemos substituir o que seria o caminho relativo para a diretoria onde se quer salvar os ficheiros pela execução de uma função que seleciona o *path* de acordo com o sistema operativo que se está a usar. Esta alteração permite que a nossa biblioteca não apresente qualquer tipo de dependência do sitio onde é executada (independentemente do sistema operativo).

Conseguimos resolver o problema dos ficheiros temporários que são usados pela componente de visualização, mas ainda temos problemas para apresentar os *icons* do *dashboard*. Atualmente, estamos a usar o *path* relativo, com base na diretoria onde efetuamos os testes do mesmo, para apresentar os *icons*. Isto significa que, caso se execute o *dashboard* noutra diretoria, este não apresenta os *icons*, o mesmo se sucede se este for executado noutro dispositivo. Para resolver este problema, recorremos ao uso da biblioteca Importlib Resources [20], uma vez que, através de uma sintaxe semelhante ao *import* do Python, esta consegue obter o *path* absoluto para os *icons*, através do método `path('icon.path', 'icon-name.png')` (é necessário converter o resultado para string a través do métodos `str()` para se conseguir usar o resultado com o `PyQt`). Através disto, conseguimos obter o *path* absoluto para os *icons*, sem ter problemas com o dispositivo e diretoria onde se executa.

Por fim, após efetuarmos todas as alterações necessárias para que a biblioteca fosse independente da nossa máquina e do sistema operativo, começamos a pesquisar como é que se criam e publicam bibliotecas em Python. Precisamos de voltar a alterar a maneira como se ia buscar os *icons*, uma vez que agora estes têm de estar dentro da informação da biblioteca, passando a usar o código presente na Listagem 5.3. Este código segue a mesma lógica, mas permite a geração por partes, dando para reutilizar a raiz do *path*. Esta alteração foi necessária porque com o método `path()` não conseguíamos definir o *package* e o *path* para o *icon* ao mesmo tempo.

```
PACKAGE_NAME: str = "packagename"          # nome do package
ICONS_DIRECTORY: str = "resources/icons"    # path relativo para a diretoria dos icons do package

ICONS: Traversable = pkg_resources.files(PACKAGE_NAME).joinpath(ICONS_DIRECTORY)

BUTTON_ICON: str = str(ICONS.joinpath("icon-name.webp"))     # nome do icon que se pretende
```

Listagem 5.3: Bloco de código usado para obter o *path* para os *icons*

Como já sabíamos quando iniciamos o projeto que este devia dar origem a uma biblioteca, começamos logo a seguir algumas caraterísticas, tal como a estrutura do projeto, de acordo com o guia [48]. Acabamos a usar também na parte final para gerar os ficheiros necessários para a criação do *package* (`LICENCE`, `README.md` e `pyproject.toml`.

O ficheiro `LICENCE` é a licença para utilizar a biblioteca, a qual acabamos por usar a licença MIT, que é muito comum em projetos *open source*. O ficheiro `README.md` é a pagina inicial dos repositórios *git*. É usado como uma introdução ao código disponível, com alguns exemplos de como o usar. Tendo isso em conta, acabamos por usar uma *template* para o mesmo. Efetuarmos as alterações necessárias e adicionarmos os conteúdos que pretendíamos (incluindo exemplos de uso da biblioteca) para que este fosse bastante fácil de perceber para quem pretender usar a biblioteca.

Passando agora para o ficheiro principal, `pyproject.toml`, usamos como base a *template* fornecida no guia [48]. Alteramos os campos informativos de acordo com as informações da nossa biblioteca (*name*, *version*, *authors*, *description*, *classifiers*), definimos o readme, a licença e os links do projeto (repositório github (*Homepage*) e reportar erros (*Issues*)).

O ficheiro `LICENCE` é a licença para utilizar a biblioteca, a qual acabamos por usar a licença MIT, que é muito comum em projetos *open source*. O ficheiro `README.md` é a pagina inicial dos repositórios *git*. É usado como uma introdução ao código disponível, com alguns exemplos de como o usar. Tendo isso em conta, acabamos por usar uma *template* para o mesmo. Efetuarmos as alterações necessárias e adicionarmos os conteúdos que pretendíamos (incluindo exemplos de uso da biblioteca) para que este fosse bastante fácil de perceber para quem pretender usar a biblioteca.

Passando agora para o ficheiro principal, `pyproject.toml`, usamos como base a *template* fornecida no guia [48]. Alteramos os campos informativos de acordo com as informações da

nossa biblioteca (*name*, *version*, *authors*, *description*, *classifiers*), definimos o readme, a licença e os links do projeto (repositório github (*Homepage*) e reportar erros (*Issues*)).

Como a biblioteca apresenta algumas dependências e de modo a automatizar o processo de gestão de dependências, decidimos recorrer a uma ferramenta de automação de dependências. Após uma pequena pesquisa, deparámos-nos com um *plugin* para o hatching [38] (ferramenta de compilação do *package* que estamos a utilizar) que consegue gerar as dependências de forma dinâmica a partir do ficheiro `requirements.txt` que é facilmente gerado/atualizado pelos IDEs. Com isto diminuímos os problemas da gestão de dependências, mas para conseguir utilizar este *plugin* é necessário efetuar umas alterações ao ficheiro, como se pode ver na Listagem 5.4. Adicionando este código, as dependências são geradas automaticamente durante o *build* da biblioteca.

```
1  ...
2  [build-system]
3  requires = ["hatchling", "hatch-requirements-txt"]  # adicionar o plugin
       para gerar dependencias
4  ...
5  [project]
6  dynamic = ["dependencies"]                          # indicar que as
       dependenciasvão ser geradas dinamicamente
7  ...
8  [tool.hatch.metadata.hooks.requirements_txt]        # configurar o plugin
       com o ficheiro de dependencias
9  files = ["requirements.txt"]
10 ...
```

Listagem 5.4: Alterações a realizar ao ficheiro pyproject.toml para conseguir usar dependências geradas de forma dinâmica

Em condições normais de *build*, os recursos não são incluídos no *package*, no entanto, necessitamos de incluir os *icons* no *package* para que o *dashboard* os consiga usar. Para resolver este problema adicionamos o código presente na Listagem 5.5 ao ficheiro `pyproject.toml`, que força a inclusão e mapeamento do que pretendemos. É importante realçar que é necessário colocar o nome do nosso *package* na diretoria onde pretendemos guardar, para que este fique dentro do mesmo. Isto permite copiar pastas ou ficheiros, mas caso seja ficheiros, é necessário apresentar o seu nome completo no destino também, uma vez que esta operação copia apenas os conteúdos, deste modo também permite alterar o nome dos ficheiros para que apresentem um nome diferentes dentro do *package*.

```
1  ...
2  [tool.hatch.build.targets.wheel.force-include]      # forçar a inclusão dos
       resursos
3  "resources/icons" = "xefr4py/resources/icons"       # mapear a diretoria com
       os recursos para onde estes vão
4  ...
```

Listagem 5.5: Alterações a realizar ao ficheiro pyproject.toml para adicionar recursos ao *package*

Como já tínhamos os ficheiros todos configurados, decidimos fazer upload da biblioteca para o

repositório de teste do PyPI [1]. Após efetuarmos a compilação e upload da biblioteca, quando acedemos à sua respetiva página no site, constatamos que, apesar de termos uma pasta chamada `docs`, estes não foram incorporados na página da biblioteca. Deste modo, precisamos de arranjar maneira de guardas os `docs` e para percebermos como isto é feito, acabamos por aceder ás páginas no repositório PyPI dos *packages* `NumPy` e `Pandas`. Ao observar estes, reparamos que a documentação está alocada num site externo, nomeadamente no site do próprio *package*. Como não temos site, esta opção não é valida.

Após uma breve pesquisa, decidimos usar o site read the docs [12]. Este site suporta a documentação escrita com o sphix e além disso, ele acede diretamente ao github do projeto, compila e dá *host* à documentação que gera. O site também suporta *webhook* para compilar automaticamente a documentação quando é efetuado um *commit* no repositório, garantindo que a documentação estás sempre atualizada de acordo com o código (e respetivos comentários).

[1] o PyPI apresenta um repositório onde os utilizadores podem efetuar testes sem interferir com o repositório principal de *packotes* Python chamado test PyPI `https://test.pypi.com`

6 Avaliação

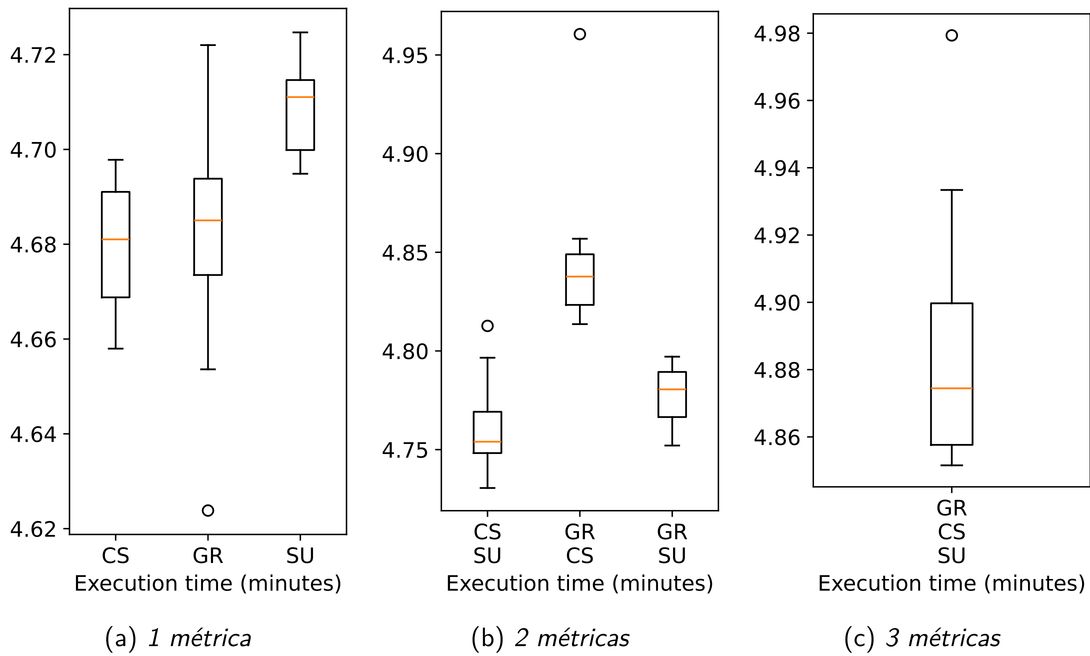
Este capítulo é focado nos resultados obtidos com a implementação em Python e a comparação destes com os resultados da implementação em R. Efetuamos testes face á semelhança dos resultados e relativamente ao tempo de execução, em função das métricas, número de caraterísticas e número de instâncias.

Para efetuar os testes recorremos ao uso de 2 *datasets* reais: o Arcene [26], mais concretamente apenas a sua componente de treino, que tem 127 instâncias e 10.000 caraterísticas e o KDD CUP 2008 [55] com 102.294 instâncias e 127 caraterísticas. Os testes com estes *datasets* foram feitos usando as combinações das métricas *gain ratio* (GR), *chi squared* (CS) e *symmetrical uncertainty* (SU). Não usamos a métrica *random forest importance* (RFI) porque esta não dava para aplicar em R aos *datasets*. Para cada combinação foram efetuadas 10 repetições (número padrão de testes realizados para diminuir o efeito de anomalias).

Como se pode ver pelas Figuras 6.1 e 6.2 (as figuras apresentam escalas diferentes), para o *dataset* KDD Cup 2008, a implementação em Python apresenta um aumento de tempo pouco significativo por métrica (menos de 5% por métrica), enquanto a implementação em R apresenta um tempo diretamente proporcional ao número de métricas. Apesar de para uma métrica o tempo de execução em R ser 2,5 vezes mais rápido do que em Python, para 3 métricas a diferença já é pouco significativa (menos de 10%). Isto demonstra que para casos de uso reais, onde o *dataset* apresenta um grande número de instâncias, o algoritmo em R é mais rápido caso tenha menos de 3 métricas, em Python seria mais rápido.

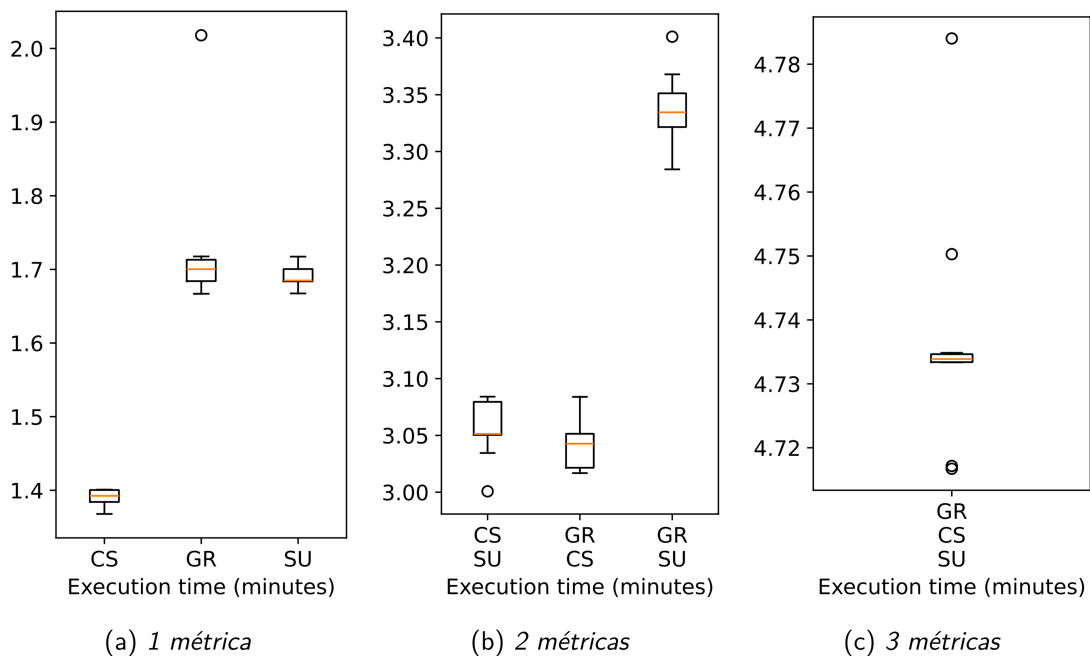
Nas Figuras 6.3 e 6.4 (as figuras apresentam escalas diferentes), para o *dataset* Arcene, na implementação em Python, o número de métricas manteve um baixo impacto no tempo de execução, tal como a implementação em R continua a ter um aumento de tempo proporcional. Como se pode observar, ao contrário do conjunto KDD Cup 2008, como este dataset apresenta uma grande dimensionalidade, a implementação em Python consegue ser sempre pelo menos 7 vezes mais rápida. Isto torna-a uma opção muito mais viável, independentemente do número de métricas.

No primeiro caso de estudo, *dataset* KDD Cup 2008, existe um grande número de caraterísticas, enquanto no segundo caso, *dataset* Arcene, existe um grande número de instâncias. Com apenas estes 2 *datasets*, apesar de bastante distintos, não são suficientes para perceber o desempenho



(a) *1 métrica* (b) *2 métricas* (c) *3 métricas*

Figura 6.1: *Tempo de execução para o dataset KDD Cup 2008 em Python*

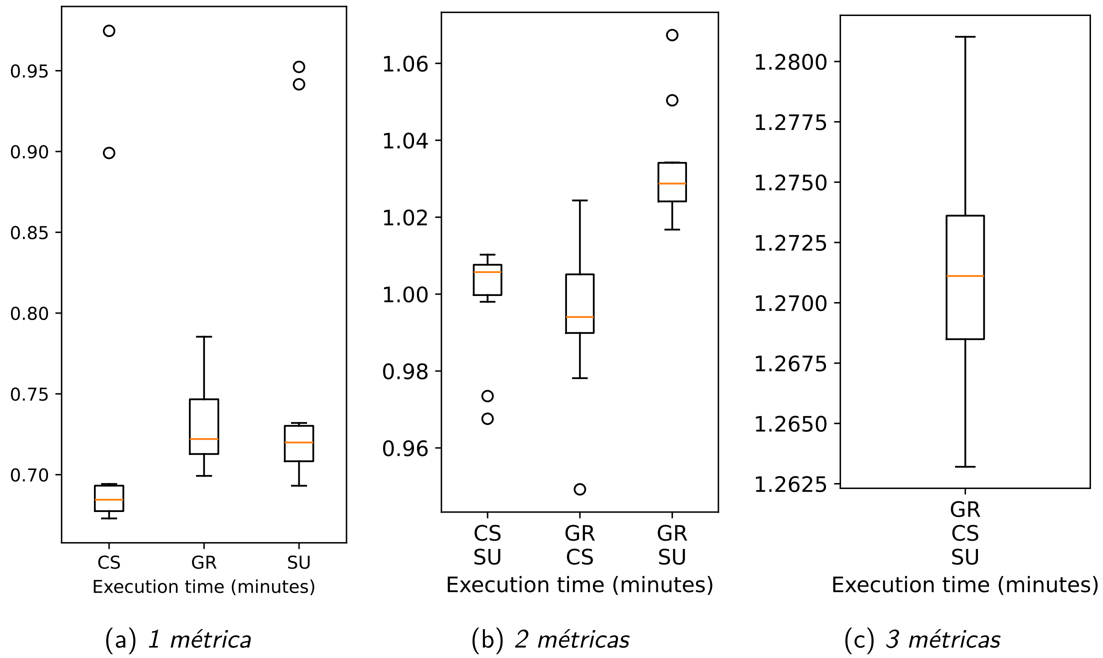


(a) *1 métrica* (b) *2 métricas* (c) *3 métricas*

Figura 6.2: *Tempo de execução para o dataset KDD Cup 2008 em R*

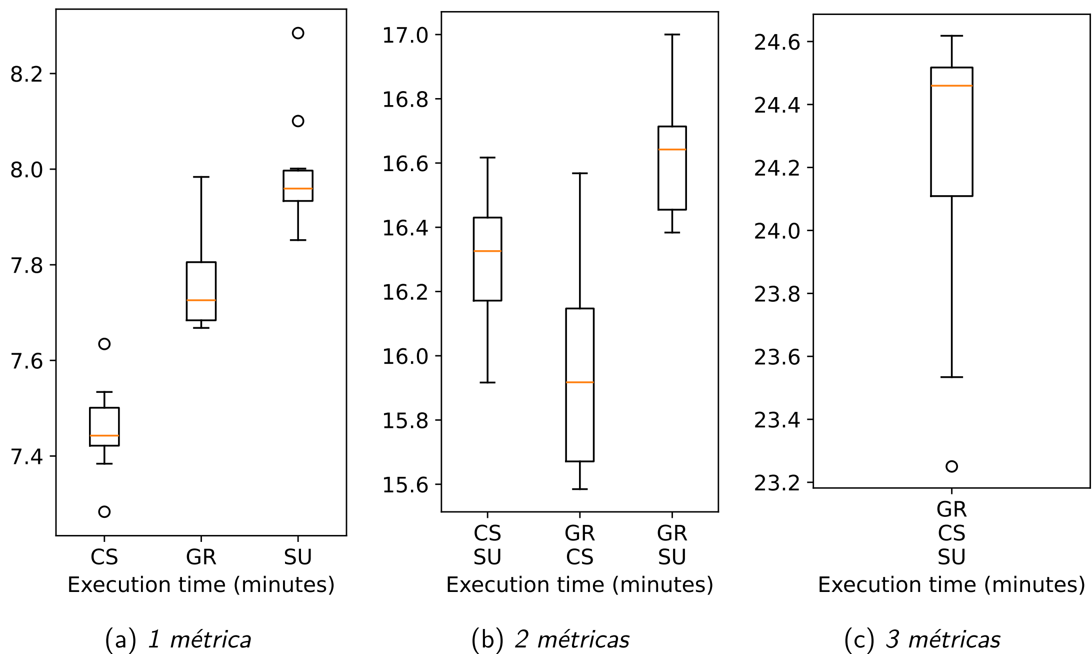
das implementações para *datasets* com mais instâncias e/ou caraterísticas. Para realizar estes testes criamos *datasets* sintéticos, de modo a estudar o que causa mais impacto do desempenho de cada implementação (instâncias vs caraterísticas). Sintetizamos estes *datasets* a partir do Arcene, tendo sintetizado os *datasets* com as dimensões presentes na Tabela 6.1.

Como já realizamos testes para perceber a diferença de desempenho causada pelo número de métricas, procedemos á realização dos restantes testes com 3 métricas, para que este resultado seja mais próximo da realidade.



(a) *1 métrica* (b) *2 métricas* (c) *3 métricas*

Figura 6.3: *Tempo de execução para o dataset Arcene em Python*



(a) *1 métrica* (b) *2 métricas* (c) *3 métricas*

Figura 6.4: *Tempo de execução para o dataset Arcene em R*

Devido a problemas na implementação em R (devido ao facto de usar a JVM e esta ter memória limitada), não é possível usar a implementação em R para o dataset 4. Por problemas de falta de memória no computador usado para fazer os testes, também não é possível usar os *datasets* 1. No entanto, todos estes datasets são compatíveis com a versão em Python, o que impossibilita a apresentação de uma comparação direta de desempenho entre as versões. Assim, podemos concluir que a implementação em Python se revela mais versátil do que a versão em R, principalmente devido a limitações de memória associadas ao uso da JVM.

O impacto das métricas no tempo de execução justifica-se com base em que parte da execução

dataset	instâncias	caraterísticas
dataset 1	100	100.000
dataset 2	100	1.000.000
dataset 3	10.000	10.000
dataset 4	1.000.000	100

Tabela 6.1: Dimensões dos *datasets* sintéticos

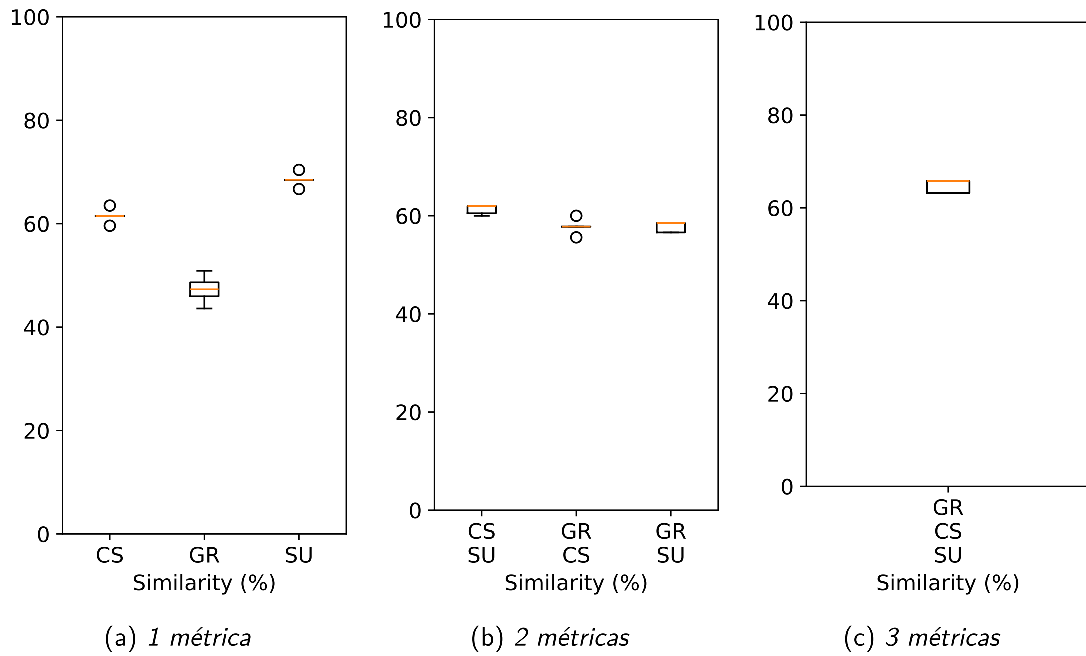
do algoritmo é que é efetuada a discretização dos dados, uma vez que em Python esta é realizada antes do algoritmo em si, enquanto no R é efetuada dentro das métricas. Como a discretização tende a ser muito mais pesada do que a aplicação das próprias métricas, o tempo de execução final do algoritmo é altamente influenciado pela discretização.

Mas o tempo de execução não é o fator importante deste trabalho. O objetivo é que a implementação em Python tenha resultados semelhantes aos resultados da implementação em R. Para percebermos se tal acontece vamos comparar os resultados das implementações em Python e R com um resultado base da execução em R. Consideramos que os resultados são semelhantes se existir uma diferença pequena (menos de 10%) entre a similaridade de R e Python face á *baseline*(|similaridade R-*baseline* - similaridade Python-*baseline*| $<$ 10%).

Nas Figuras 6.5 a 6.8 podemos observar esta comparação. As Figuras 6.5 e 6.7 são referentes ao Python e as 6.6 e 6.8 são referentes ao R. Como se pode constatar, apesar das implementações apresentarem diferenças substanciais na componente aleatória [7], conseguimos resultados por vezes com maior similaridade com o *baseline* do que os próprios resultados em R. Na Figura 6.5a consegui-mos perceber que a métrica *gain ratio* apresenta menos similaridade do que os restantes resultados em Python, mas mesmo assim continua próximo (2% de diferença do Python para o R) dos resultados em R. Para as restantes métricas e combinações conseguiu destacar-se dos resultados em R.

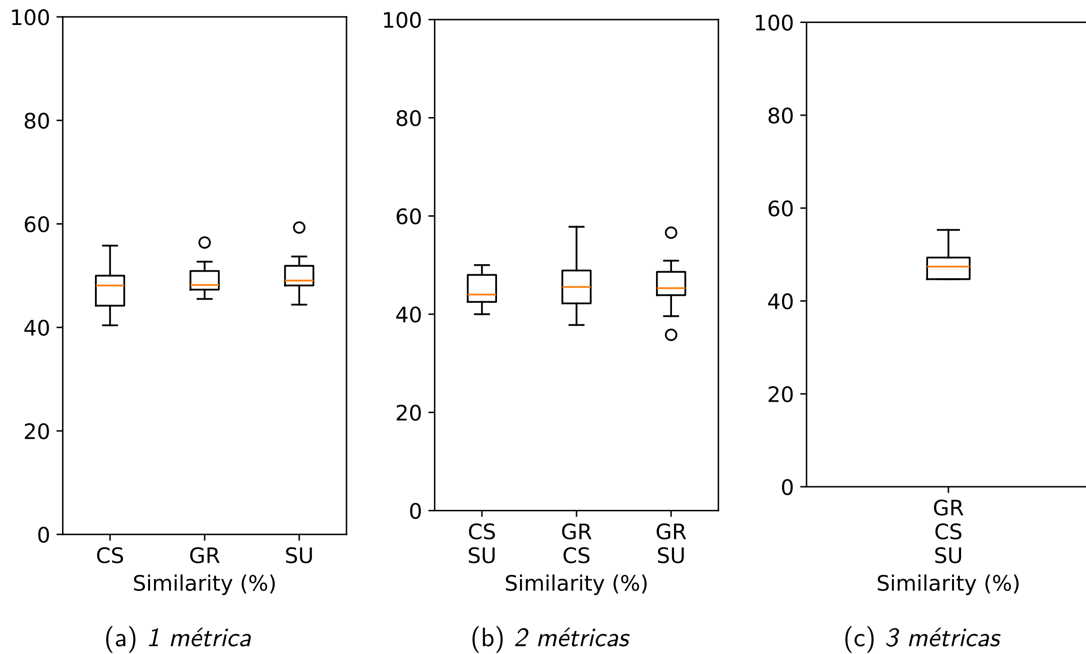
Através destas Figuras pode-se observar que os resultados em R apresentam uma dispersão muito maior que os resultados em Python, possivelmente devido á maneira com que a aleatoriedade funciona nas linguagens. Juntando esta informação com a dos tempos de execução temos motivos para acreditar que a implementação tem potencial para ser mais rápido do que em R sem apresentar nenhum tipo de compromisso na qualidade dos resultados.

Para as Figuras 6.7 e 6.8 os resultados do Python já são mais próximos dos resultados em R. No entanto ainda se consegue observar a diferença da dispersão estatística das duas linguagens.



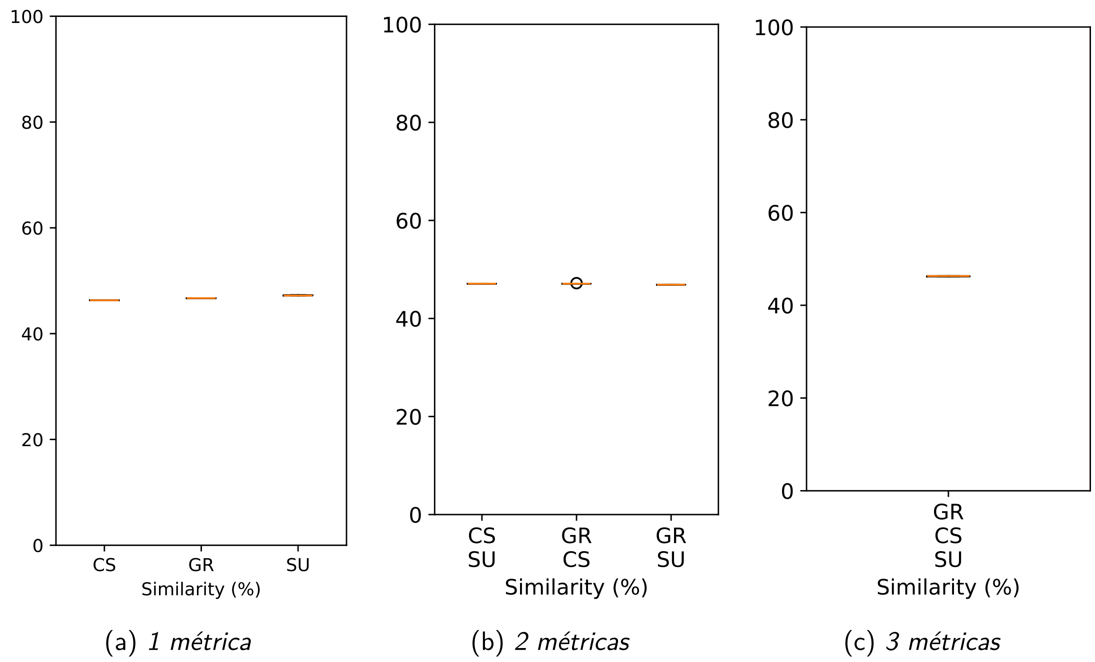
(a) *1 métrica* (b) *2 métricas* (c) *3 métricas*

Figura 6.5: *Comparação do resultado gerado em Python com a baseline em R para o dataset KDD Cup 2008*



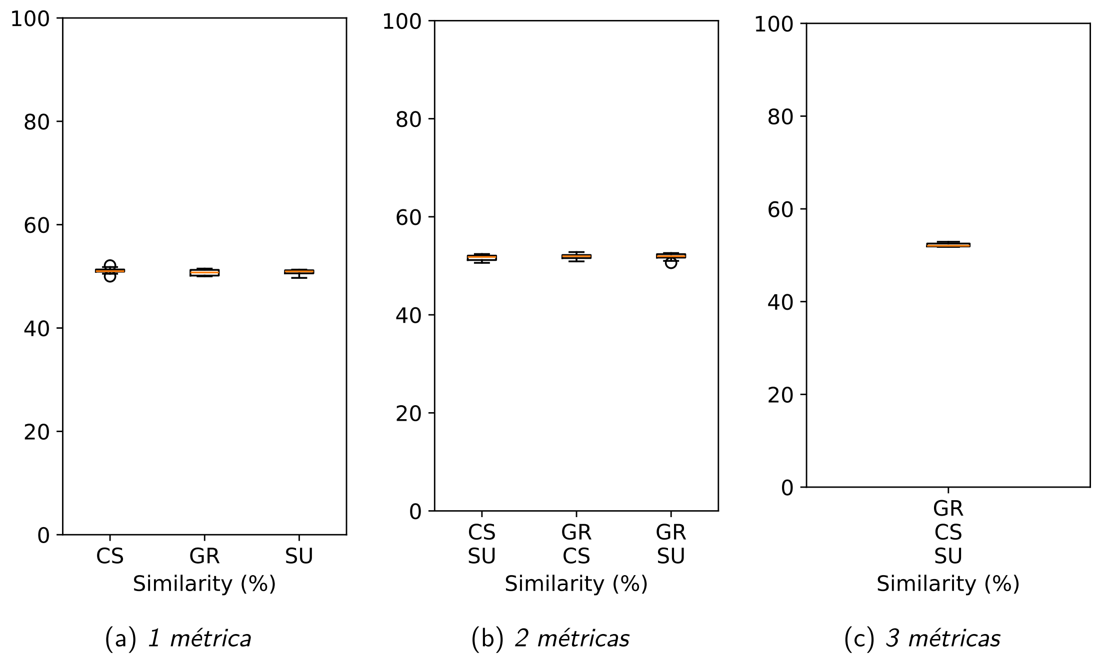
(a) *1 métrica* (b) *2 métricas* (c) *3 métricas*

Figura 6.6: *Comparação do resultado gerado em R com a baseline em R para o dataset KDD Cup 2008*



(a) *1 métrica* (b) *2 métricas* (c) *3 métricas*

Figura 6.7: *Comparação do resultado gerado em Python com a baseline em R para odataset Arcene*



(a) *1 métrica* (b) *2 métricas* (c) *3 métricas*

Figura 6.8: *Comparação do resultado gerado em R com a baseline em R para o dataset Arcene*

7 Conclusões e trabalho futuro

Com este trabalho conseguimos transportar o algoritmo EEFR de R para Python, que é uma linguagem muito mais usada, transportando este para uma comunidade mais ampla e com melhorias face á versão em R. Estas melhorias incluem seleção da classe e lista negra de caraterísticas, além de um *dashboard* para observar o desenvolvimento do resultado gerado pelo algoritmo.

Ao transportamos o algoritmo de R para Python estamos a abrir novos horizontes para a sua utilização, uma vez que a linguagem Python tem uma base de utilizadores maior que R. No entanto, mudar para Python apresenta diversos problemas, devido ás diferenças das linguagens. Ao separarmos devidamente as etapas do algoritmo também conseguimos atenuar a diferença de desempenho das linguagens.

Podemos demonstrar que as implementações apresentam resultados similares, uma vez que pelos resultados apresentados no capítulo 6, mesmo nos casos onde a implementação em R é mais rápida para apenas 1 métrica, esta diferença de desempenho é atenuada quando se usam várias métricas, enquanto existe outros casos em que a implementação em Python é sempre mais rápida.

As funções adicionadas (seleção da classe e lista negra) permitem que os utilizadores da biblioteca possam alterar facilmente a classe e remover caraterísticas que não são relevantes do conjunto de dados (tal como os IDs) sem precisar de recorrer a ferramentas para realizar este pré-processamento. Em R, não era possível executar o algoritmo sem que este tivesse uma coluna chamada "class", uma vez que isto estava *hard coded* na lógica do mesmo.

Passando agora para uma faceta completamente nova neste algoritmo, o *dashboard* tem como objetivo ajudar o utilizador a perceber como o algoritmo funciona internamente, tornando assim o algoritmo "explicavél". A "explicabilidade" é uma caraterística essencial para áreas criticas, tal como a medicina, deste modo abre-se novos horizontes para este algoritmo.

Desenvolver um *dashboard* interativo é um passo importante para melhorar a usabilidade e a explicabilidade do algoritmo. Ao visualizar as métricas internas e as ordenações do algoritmo, os utilizadores podem obter uma compreensão mais profunda da importância das características e do comportamento do algoritmo. A nossa abordagem de guardar e visualizar não só auxilia na compreensão das decisões do algoritmo, mas também garante transparência e fidelidade

na explicação do modelo. Apesar da complexidade inerente do algoritmo e dos desafios apresentados por conjuntos de dados de grandes dimensões, a nossa avaliação revela resultados promissores em termos de usabilidade e explicabilidade.

Para trabalho futuro é importante adicionar mais métricas e criar suporte de estrutura fixa para que caso as métricas disponibilizadas não cheguem ou não se adequem ao problema o utilizador possa usar métricas criadas por ele. Isto pode implicar trocar parte da lógica como se define as métricas e como estas são aplicadas para permitir uma estrutura mais flexível. Também se deve adicionar suporte para receber *arrays* do `NumPy`, que também são muito utilizados, já que o programa só recebe *Dataframes* do `Pandas`.

Pode-se melhorar as funcionalidades do *dashboard* através de testes com utilizadores, avaliando a sua usabilidade e ajustando-o conforme o *feedback* recebido. O desenvolvimento e aperfeiçoamento contínuo dos métodos explicáveis de seleção de características têm um potencial significativo para impulsionar a adoção de modelos de aprendizagem automática mais transparentes e confiáveis.

Bibliografia

[1] H. Abe. *Python: Getting AppData folder in a cross-platform way — stackoverflow.com*. Ago. de 2024. URL: https://stackoverflow.com/a/61901696 (ver p. 38).

[2] A. Adadi e M. Berrada. "Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)". Em: *IEEE Access* 6 (2018), pp. 52138–52160. DOI: 10.1109/ACCESS.2018.2870052 (ver p. 7).

[3] C. Agarwal, O. Queen, H. Lakkaraju e M. Zitnik. "Evaluating explainability for graph neural networks". Em: *Scientific Data* 10.1 (mar. de 2023), p. 144. ISSN: 2052-4463. DOI: 10.1038/s41597-023-01974-x. URL: https://doi.org/10.1038/s41597-023-01974-x (ver p. 7).

[4] S. Ali, T. Abuhmed, S. El-Sappagh, K. Muhammad, J. M. Alonso-Moral, R. Confalonieri, R. Guidotti, J. Del Ser, N. Díaz-Rodríguez e F. Herrera. "Explainable Artificial Intelligence (XAI): What we know and what is left to attain Trustworthy Artificial Intelligence". Em: *Information Fusion* 99 (2023), p. 101805. ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2023.101805. URL: https://www.sciencedirect.com/science/article/pii/S1566253523001148 (ver p. 7).

[5] D. Amorim, M. Pato e N. Datia. "Explainable Feature Ranking using Interactive Dashboards". Em: *2024 28th International Conference Information Visualisation (IV)*. IEEE. 2024. DOI: 110.1109/IV64223.2024.00050 (ver p. 3).

[6] A. Arias-Duart, F. Parés, D. Garcia-Gasulla e V. Gimenez-Abalos. "Focus! rating XAI methods and finding biases". Em: *2022 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE. 2022, pp. 1–8 (ver p. 7).

[7] N. Batra. *Creating same random number sequence in Python, NumPy and R — stackoverflow.com*. Ago. de 2024. URL: https://stackoverflow.com/questions/22213298/creating-same-random-number-sequence-in-python-numpy-and-r (ver p. 46).

[8] B. Brode. *The Problem with Big Data: It's Getting Bigger - DATAVERSITY*. Fev. de 2024. URL: https://www.dataversity.net/the-problem-with-big-data-its-getting-bigger/ (ver p. 1).

[9] P. community. *PyPI The Python Package Index — pypi.org*. Ago. de 2024. URL: https://pypi.org/ (ver p. 37).

[10] DeepAI. *Feature reduction Definition | DeepAI*. Fev. de 2024. URL: https://deepai.org/machine-learning-glossary-and-terms/feature-reduction (ver p. 5).

[11] S. developers. *Sphinx — Sphinx documentation — sphinx-doc.org*. Ago. de 2024. URL: https://www.sphinx-doc.org/en/master/index.html (ver p. 37).

[12] I. Read the Docs. *Full featured documentation deployment platform — about.readthedocs.com*. Ago. de 2024. URL: https://about.readthedocs.com/ (ver p. 41).

[13] M. Driscoll. *How to Build a Python GUI Application With wxPython – Real Python*. Mar. de 2024. URL: https://realpython.com/python-gui-with-wxpython/ (ver p. 29).

[14] R. Dwivedi, D. Dave, H. Naik, S. Singhal, R. Omer, P. Patel, B. Qian, Z. Wen, T. Shah, G. Morgan e R. Ranjan. "Explainable AI (XAI): Core Ideas, Techniques, and Solutions". Em: *ACM Comput. Surv.* 55.9 (jan. de 2023). ISSN: 0360-0300. DOI: 10.1145/3561048. URL: https://doi.org/10.1145/3561048 (ver p. 8).

[15] D. Eppstein. *Curse of dimensionality - Wikipedia*. Fev. de 2024. URL: https://en.wikipedia.org/wiki/Curse_of_dimensionality (ver p. 1).

[16] P. europeu. *EU AI Act: first regulation on artificial intelligence | Topics | European Parliament — europarl.europa.eu*. Ago. de 2024. URL: https://www.europarl.europa.eu/topics/en/article/20230601STO93804/eu-ai-act-first-regulation-on-artificial-intelligence (ver p. 8).

[17] eyllanesc. *python - How to display a Pandas data frame with PyQt5/PySide2 - Stack Overflow*. Mar. de 2024. URL: https://stackoverflow.com/a/44605011 (ver p. 30).

[18] U. M. Fayyad e K. B. Irani. "Multi-interval discretization of continuousvalued attributes for classification learning". Em: *Thirteenth International Joint Conference on Articial Intelligence*. Vol. 2. Morgan Kaufmann Publishers, 1993, pp. 1022–1027 (ver pp. 14, 17).

[19] M. Fitzpatrick. *Matplotlib plots in PyQt5, embedding charts in your GUI applications*. Mar. de 2024. URL: https://www.pythonguis.com/tutorials/plotting-matplotlib/ (ver p. 31).

[20] P. S. Foundation. *importlib.resources – Package resource reading, opening and access — docs.python.org*. Ago. de 2024. URL: https://docs.python.org/3/library/importlib.resources.html (ver p. 38).

[21] P. S. Foundation. *logging — Logging facility for Python — docs.python.org*. Ago. de 2024. URL: https://docs.python.org/3/library/logging.html (ver p. 23).

[22] P. Fundation. *tkinter — Python interface to Tcl/Tk — Python 3.12.2 documentation*. Mar. de 2024. URL: https://docs.python.org/3/library/tkinter.html (ver p. 29).

[23] A. Gandomi e M. Haider. "Beyond the hype: Big data concepts, methods, and analytics". Em: *International journal of information management* 35.2 (2015), pp. 137–144 (ver p. 1).

[24] *Getting Started — Sphinx documentation — sphinx-doc.org*. Ago. de 2024. URL: https://www.sphinx-doc.org/en/master/usage/quickstart.html (ver p. 37).

[25] H. M. C. Gomes. *Desenvolvimento de um package em R para Ensemble Feature Ranking–EFR*. 2021 (ver pp. 2, 7).

[26] I. Guyon, S. Gunn, A. Ben-Hur e G. Dror. *Arcene*. 2008. URL: https://doi.org/10.24432/C58P55 (ver p. 43).

[27] K. Hornik, C. Buchta, T. Hothorn, A. Karatzoglou, D. Meyer e A. Zeileis. *RWeka: R/Weka Interface*. Jan. de 2024. URL: https://cran.r-project.org/web/packages/RWeka/index.html (ver p. 15).

[28] J. Huang, Z. Wang, D. Li e Y. Liu. "The Analysis and Development of an XAI Process on Feature Contribution Explanation". Em: *2022 IEEE International Conference on Big Data (Big Data)*. 2022, pp. 5039–5048. DOI: 10.1109/BigData55660.2022.10020313 (ver p. 7).

[29] G. Hughes. "On the mean accuracy of statistical pattern recognizers". Em: *IEEE transactions on information theory* 14.1 (1968), pp. 55–63 (ver p. 1).

[30] J. Inlow. *discrust*. Jan. de 2024. URL: https://pypi.org/project/discrust/ (ver p. 17).

[31] K. Jabeen, M. A. Khan, M. Alhaisoni, U. Tariq, Y.-D. Zhang, A. Hamza, A. Mickus e R. Damaševičius. "Breast cancer classification from ultrasound images using probability-based optimal deep learning feature fusion". Em: *Sensors* 22.3 (2022), p. 807 (ver p. 6).

[32] Jiffyclub. *Performance of Pandas Series vs NumPy Arrays – Pen and Pants*. Fev. de 2024. URL: https://penandpants.com/2014/09/05/performance-of-pandas-series-vs-numpy-arrays/ (ver p. 20).

[33] L. Kotthoff, P. Schratz e A. Jiménez. *fselector/R at master · larskotthoff/fselector — github.com*. Jan. de 2024. URL: https://github.com/larskotthoff/fselector/tree/master/R (ver pp. 14, 15).

[34] B. Kovalerchuk, R. Andonie, N. Datia, K. Nazemi e E. Banissi. "Visual knowledge discovery with artificial intelligence: Challenges and future directions". Em: *Integrating artificial intelligence and visualization for visual knowledge discovery*. Springer, 2022, pp. 1–27 (ver p. 2).

[35] B. Kovalerchuk e E. McCoy. "Explainable Machine Learning for Categorical and Mixed Data with Lossless Visualization". Em: *Artificial Intelligence and Visualization: Advancing Visual Knowledge Discovery*. Springer, 2024, pp. 73–123 (ver p. 7).

[36] B. La Rosa, G. Blasilli, R. Bourqui, D. Auber, G. Santucci, R. Capobianco, E. Bertini, R. Giot e M. Angelini. "State of the Art of Visual Analytics for eXplainable Deep Learning". Em: *Computer Graphics Forum* 42.1 (2023), pp. 319–355. DOI: https://doi.org/10.1111/cgf.14733. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14733. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14733 (ver p. 7).

[37] A. Lee. *mplcursors ; Interactive data selection cursors for Matplotlib ; mplcursors 0.5.3 documentation — mplcursors.readthedocs.io*. Ago. de 2024. URL: https://mplcursors.readthedocs.io/en/stable (ver p. 32).

[38] O. Lev. *Hatch - Hatch — hatch.pypa.io*. Ago. de 2024. URL: https://hatch.pypa.io/latest/ (ver p. 40).

[39] A. Liaw e M. Wiener. *randomForest: Breiman and Cutlers Random Forests for Classification and Regression — cran.r-project.org*. Ago. de 2024. URL: https://cran.r-project.org/web/packages/randomForest/index.html (ver p. 17).

[40] M. P. de Mina Pato. *GitHub - matpato/XEFR4Py: Explainable Feature Ranking using Interactive Dashboards — github.com*. https://github.com/matpato/XEFR4Py. [Accessed 27-09-2024] (ver p. 3).

[41] M. P. de Mina Pato. *XEFR4Py — pypi.org*. Ago. de 2024. URL: https://pypi.org/project/XEFR4Py/ (ver p. 3).

[42] Misinahaiya. *PyQt - Python Wiki*. Mar. de 2024. URL: https://wiki.python.org/moin/PyQt (ver p. 29).

[43] Misinahaiya. *What is the differences between Tkinter, WxWidgets and PyQt, and PySide? - Stack Overflow*. Mar. de 2024. URL: https://stackoverflow.com/a/75845583 (ver p. 29).

[44] D. Moura. *R vs Python vs Julia: Efficient code | Towards Data Science*. Fev. de 2024. URL: https://towardsdatascience.com/r-vs-python-vs-julia-90456a2bcbab (ver p. 15).

[45] pandas. *pandas - Python Data Analysis Library — pandas.pydata.org*. Ago. de 2024. URL: https://pandas.pydata.org/ (ver p. 15).

[46] M. Pato. *The ISELthesis LaTeX Template's Manual*. Instituto Superior de Engenharia de Lisboa (ISEL-IPL). 2024. URL: https://github.com/matpato/iselthesis (ver p. viii).

[47] C. Perlich, P. Melville, Y. Liu, G. Świrszcz, R. Lawrence e S. Rosset. "Breast cancer identification: Kdd cup winner's report". Em: *ACM SIGKDD Explorations Newsletter* 10.2 (2008), pp. 39–42 (ver p. 6).

[48] PyPa. *Packaging Python Projects - Python Packaging User Guide — packaging.python.org*. Ago. de 2024. URL: https://packaging.python.org/en/latest/tutorials/packaging-projects (ver p. 39).

[49] P. Reutemann. *python-weka-wrapper3*. Jan. de 2024. URL: https://pypi.org/project/python-weka-wrapper3/ (ver p. 17).

[50] P. Romanski, L. Kotthoff e P. Schratz. *FSelector: Selecting Attributes*. Jan. de 2024. URL: https://cran.r-project.org/web/packages/FSelector/index.html (ver pp. 11, 13, 15).

[51] M. Sahakyan, Z. Aung e T. Rahwan. "Explainable Artificial Intelligence for Tabular Data: A Survey". Em: *IEEE Access* 9 (2021), pp. 135392–135422. DOI: 10.1109/ACCESS.2021.3116481 (ver p. 7).

[52] V. N. P. d. Santos. *Modelo de data mining para detecção de tumores em exames de rastreio*. 2013 (ver pp. 2, 7).

[53] SciPy. *SciPy - — scipy.org*. Ago. de 2024. URL: https://scipy.org/ (ver p. 16).

[54] B. Shetty. *What is Curse of Dimensionality? A Complete Guide | Built In*. Fev. de 2024. URL: https://builtin.com/data-science/curse-dimensionality (ver p. 2).

[55] SIGKDD. *SIGKDD : KDD Cup 2008 : Breast cancer — kdd.org*. Ago. de 2024. URL: http://www.kdd.org/kdd-cup/view/kdd-cup-2008/Data (ver p. 43).

[56] A. Singh. *Ensemble Learning : Ensemble Techniques*. Fev. de 2024. URL: https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-for-ensemble-models/ (ver p. 2).

[57] S. K. Singhi e H. Liu. "Feature subset selection bias for classification learning". Em: *Proceedings of the 23rd international conference on Machine learning*. 2006, pp. 849–856 (ver p. 2).

[58] N. J. Smith. *patsy - Describing statistical models in Python; patsy 0.5.1+dev documentation — patsy.readthedocs.io*. Ago. de 2024. URL: https://patsy.readthedocs.io/en/latest/ (ver p. 18).

[59] C. Spencer. *weka*. Jan. de 2024. URL: https://pypi.org/project/weka/ (ver p. 17).

[60] *Sphinx AutoAPI 3.3.0 documentation — sphinx-autoapi.readthedocs.io*. Ago. de 2024. URL: https://sphinx-autoapi.readthedocs.io/en/latest/tutorials.html (ver p. 37).

[61] M. A. Talukder, M. M. Islam, M. A. Uddin, A. Akhter, K. F. Hasan e M. A. Moni. "Machine learning-based lung and colon cancer detection using deep feature extraction and ensemble learning". Em: *Expert Systems with Applications* 205 (2022), p. 117695 (ver p. 6).

[62] benchmarksgame team. *Python 3 vs Java - Which programs are fastest? (Benchmarks Game) — benchmarksgame-team.pages.debian.net*. Ago. de 2024. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python.html (ver p. 18).

[63] M. development team. *Matplotlib; Visualization with Python — matplotlib.org*. Ago. de 2024. URL: https://matplotlib.org/ (ver p. 31).

[64] N. Team. *NumPy - — numpy.org*. Ago. de 2024. URL: https://numpy.org/ (ver p. 15).

[65] scikit-learn Team. *scikit-learn: machine learning in Python ; scikit-learn 1.5.2 documentation — scikit-learn.org*. Ago. de 2024. URL: https://scikit-learn.org/stable/ (ver p. 17).

[66] T. wxPython Team. *Welcome to wxPython! | wxPython*. Mar. de 2024. URL: https://wxpython.org/index.html (ver p. 29).

[67] E. Tjoa e C. Guan. "A Survey on Explainable Artificial Intelligence (XAI): Toward Medical XAI". Em: *IEEE Transactions on Neural Networks and Learning Systems* 32.11 (2021), pp. 4793–4813. DOI: 10.1109/TNNLS.2020.3027314 (ver p. 8).

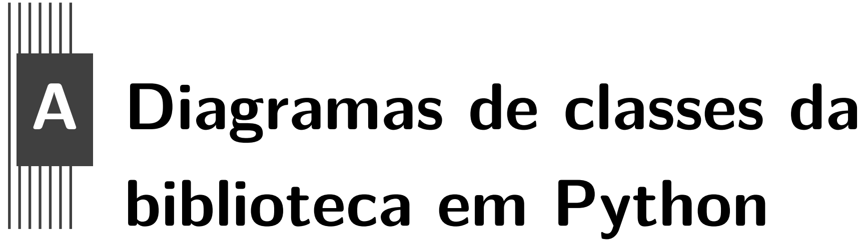
[68] A. Uberoi. *Introduction to Dimensionality Reduction - GeeksforGeeks*. Fev. de 2024. URL: https://www.geeksforgeeks.org/dimensionality-reduction/ (ver p. 1).

[69] W. University. *trunk/weka/src/main/java/weka/filters/supervised/attribute/Discre- tize.java · main · WEKA / weka · GitLab — git.cms.waikato.ac.nz*. Jan. de 2024. URL: https://git.cms.waikato.ac.nz/weka/weka/-/blob/main/trunk/weka/src/main/java/weka/filters/supervised/attribute/Discretize.java (ver p. 18).

[70] M. Wardrop. *Introduction - Formulaic — matthewwardrop.github.io*. Ago. de 2024. URL: https://matthewwardrop.github.io/formulaic/ (ver p. 18).

[71] S. Yu, M. Jin, T. Wen, L. Zhao, X. Zou, X. Liang, Y. Xie, W. Pan e C. Piao. "Accurate breast cancer diagnosis using a stable feature ranking algorithm". Em: *BMC Medical Informatics and Decision Making* 23.1 (2023), pp. 1–18 (ver p. 7).

[72] J. Zacharias, M. von Zahn, J. Chen e O. Hinz. "Designing a feature selection method based on explainable artificial intelligence". Em: *Electronic Markets* 32.4 (dez. de 2022), pp. 2159–2184. ISSN: 1422-8890. DOI: 10.1007/s12525-022-00608-1. URL: https://doi.org/10.1007/s12525-022-00608-1 (ver p. 7).



A Diagramas de classes da biblioteca em Python

Neste anexo estão presentes todos os outros diagramas de classe da biblioteca que não foram apresentados durante o documento principal, mas os quais representam todas as classes da biblioteca. Nestes diagramas pode-se observar toda a estrutura, o que pode ajudar a perceber a lógica por de trás das decisões que explicamos anteriormente.

@is@a@figure

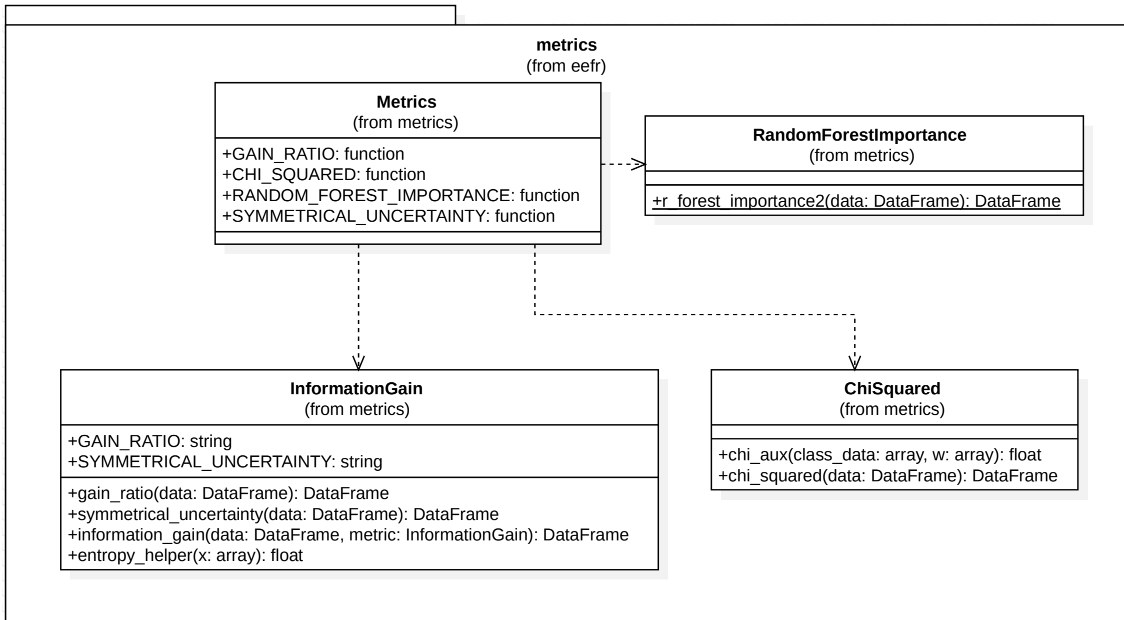


Figura A.1: *Diagrama de classe do package metrics*

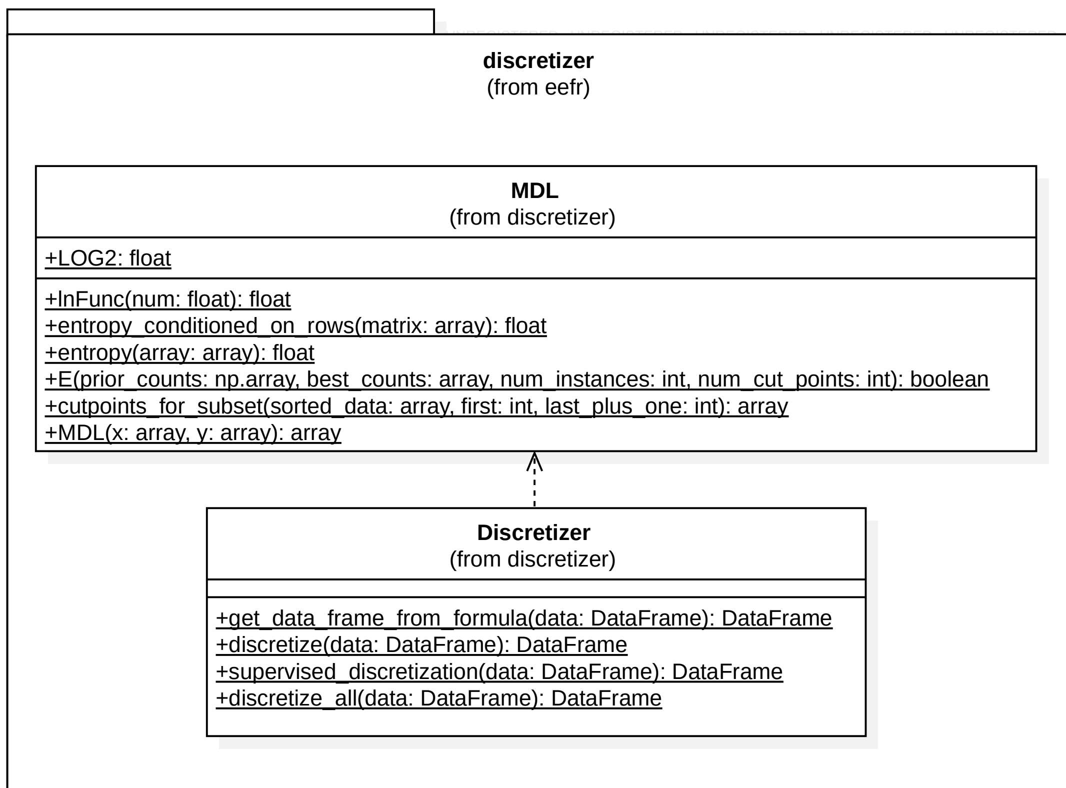


Figura A.2: *Diagrama de classe do package discretizer*

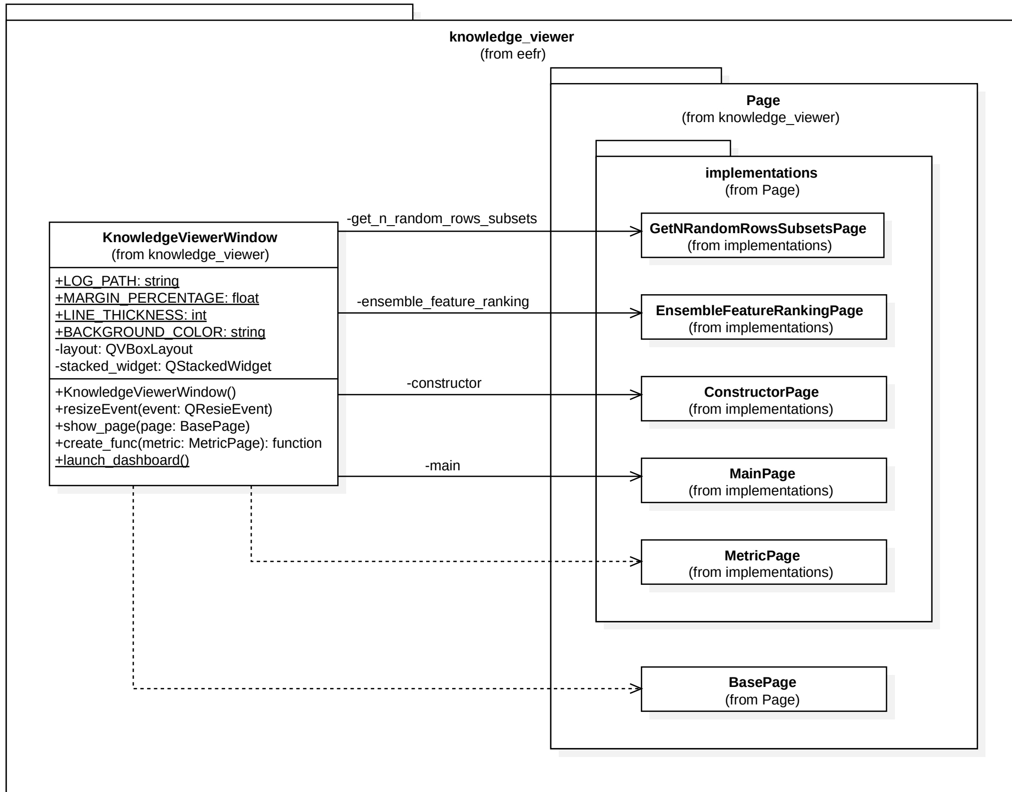


Figura A.3: *Diagrama de classe do package knowledge viewer*

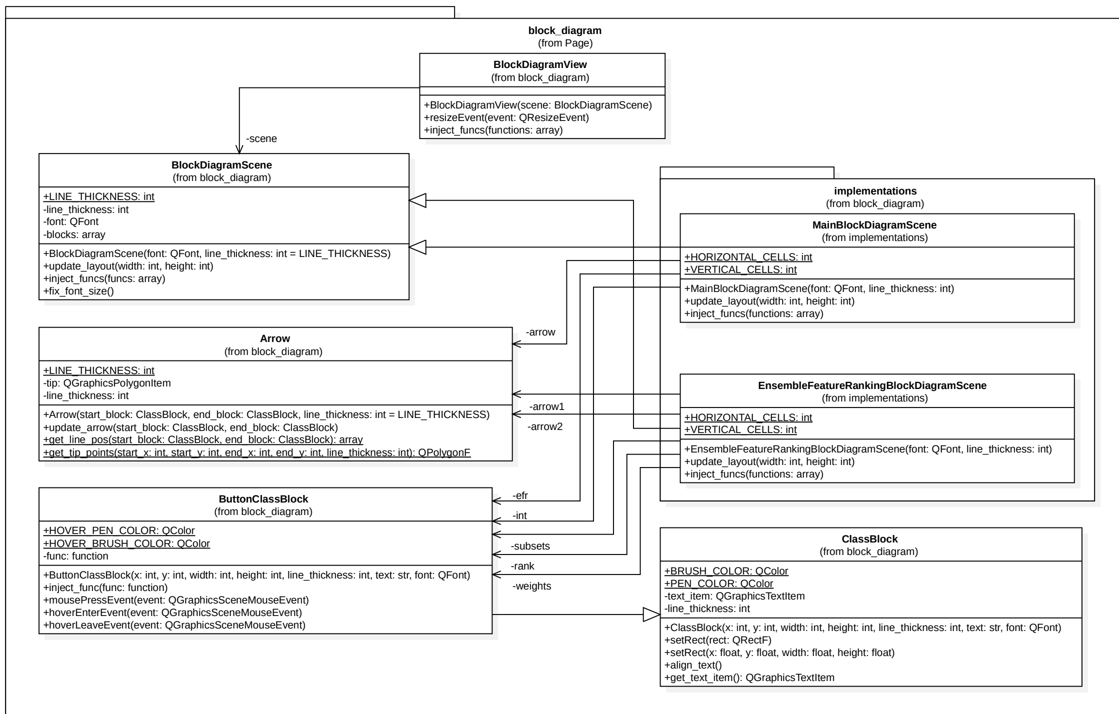


Figura A.4: *Diagrama de classe do package block diagram*

Figura A.5: *Diagrama de classe do package page*