Universidade da Beira Interior
Departamento de Informática

Automatic Keyword Extracting from Texts YAKE! Refactoring

Tiago Leandro Oliveira Valente

Licenciatura em Engenharia Informática

Orientador:

Professor Ricardo Campos, UBI

Co-orientador:

Arian Pasquali, Faktion

15 de Junho, 2025

Agradecimentos

A conclusão deste trabalho não seria possível sem todo o apoio que tive ao longo deste projeto. Quero agradecer, em especial, ao Professor Doutor Ricardo Campos, por toda a sua contínua e importantíssima paciencia, disposição e orientações, assim como também agradecer toda a ajuda e esclarecimentos dados ao longo deste projeto pelo co-orientador Arian Pasquali. Esta oportunidade de desenvolvimento deste projeto foi profundamente enriquecedora e satisfatória, proporcionando uma experiência abrangente de aplicação de novos conceitos teóricos de engenharia de *software* a um caso real com impacto direto. Gostava também de alargar os meus agradecimentos ao meu pai, mãe, irmão, família e amigos por todo o suporte que me foi dado ao longo desta etapa.

Sem todos eles, este projeto não teria sido possível.

Tiago

Resumo

Num panorama em que a manutenção de software de código *open source* é frequentemente dificultada por repositórios desorganizados, ausência de documentação e falta de processos autónomos e automáticos, a *refatorização* de código e a implementação de pipelines de Continuous Integration (CI)/Continuous Deployment (CD) assumem um papel central na melhoria da qualidade e sustentabilidade dos sistemas. O presente projeto teve como objetivo a reestruturação, através de práticas modernas de desenvolvimento, do código do Yet Another Keyword Extractor! (YAKE!) (Yet Another Keyword Extractor!), um algoritmo de extração automática de palavras-chave, amplamente utilizado na comunidade. A *refatorização* do código envolveu a adoção de boas práticas de engenharia de software, complementada pela análise estática com recurso a ferramentas como *Ruff* e *Pylint*, assegurando assim a consistência e qualidade do código. Foi também desenvolvida uma *pipeline* de CI/CD com recurso ao *GitHub Actions*, permitindo automatizar testes, processos de validação e de lançamento de novas releases do YAKE!, facilitando assim a manutenção contínua do projeto. Adicionalmente, foi criado , com recurso ao *Fumadocs* um novo *website* de documentação técnica do YAKE!, promovendo a transparência e acessibilidade do projeto junto da comunidade cientifica. O trabalho realizado permitiu consolidar uma base sólida para futuras expansões do YAKE!, tornando o sistema efetivamente mais colaborativo e preparado para evolução contínua, permitindo, no futuro o desenvolvimento de novas funcionalidades requeridas pela comunidade de utilizadores do YAKE!, nos issues do Github No futuro, será pertinente explorar a integração de Large Language Modelss (LLMs) no YAKE!, de forma a avaliar o seu impacto na extração de palavras-chave e no enriquecimento semântico dos resultados.

Palavras-Chave

Extração de palavras-chave, Engenharia de Software, Refatoração, YAKE!

Abstract

In a landscape where open source software maintenance is often hindered by disorganized repositories, lack of documentation, and absence of autonomous and automated processes, code refactoring and the implementation of CI/CD pipelines play a central role in improving the quality and sustainability of systems. This project aimed to restructure, through modern development practices, the code of YAKE! (Yet Another Keyword Extractor!), an automatic keyword extraction algorithm widely used in the community. The code refactoring involved adopting good software engineering practices, complemented by static analysis using tools such as Ruff and Pylint, thus ensuring code consistency and quality. A CI/CD pipeline was also developed using GitHub Actions, enabling the automation of testing, validation processes, and release management of new YAKE! versions, thereby facilitating continuous project maintenance. Additionally, a new technical documentation website for YAKE! was created using Fumadocs, promoting transparency and accessibility of the project within the scientific community. The work performed established a solid foundation for future YAKE! expansions, making the system effectively more collaborative and prepared for continuous evolution, enabling future development of new functionalities required by the YAKE! user community, as documented in GitHub issues. In the future, it will be pertinent to explore the integration of LLMs into YAKE! to evaluate their impact on keyword extraction and semantic enrichment of results.

Keywords

Keyword extraction, Software Engineering, Refactoring, YAKE!

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de excertos de códigos

Acronyms

API Application Programming Interface

CD Continuous Deployment

CI Continuous Integration

CI/CD Continuous Integration / Continuous Deployment

DevOps Development Operations

FAQ Frequently Asked Questions

JSON JavaScript Object Notation

KEA Automatic Keyphrase Extraction

LLM Large Language Models

MDX Mark Down X

NLP Natural Language Processing

PEP8 Python Enhancement Proposal 8

PyPi Python Package Index

RAKE Rapid Automatic Keyword Extraction

SEO Search Engine Optimization

SRP Single Repository Principle

SSG Static Site Generation

TF.IDF Term Frequency - Inverse Document Frequency

UI User Interface

URL Uniform Resource Locator

UX User Experience

WCAG Web Content Accessibility Guidelines

YAKE! Yet Another Keyword Extractor!

Declaração de uso de "Generative AI"

Durante a preparação deste trabalho, o autor utilizou Github CoPilot e Writefull, para ajuda na estruturação, reorganização de secções e ortografia. Estas ferramentas foram utilizadas para *Sentence Polishing* (tornar explicações mais claras), suporte com *writer's block*, e formatação para LaTeX. Após a utilização destas ferramentas, o autor reviu e editou cuidadosamente todo o conteúdo, assumindo total responsabilidade pelo conteúdo da publicação.

Capítulo

1

Introdução

A refatoração de sistemas existentes e a automatização dos seus processos de desenvolvimento são elementos centrais no contexto atual da engenharia de software, em particular quando se trata de ferramentas *open-source* amplamente utilizadas. O YAKE! [1] (Yet Another Keyword Extractor) é um desses casos: um algoritmo de extração automática de palavras-chave, reconhecido pela sua simplicidade e eficácia, e amplamente utilizado em tarefas de Processamento de Linguagem Natural (PLN). YAKE! já conta com mais de 7 Milhões de downloads [2], e 750 citações em seus papers.

Apesar da sua relevância, a ausência de atualizações significativas desde 2018, aliada a uma estrutura de código pouco modular, à inexistência de validações automáticas e a uma documentação técnica reduzida, dificultava a integração de novas funcionalidades e a manutenção do projeto, tornando premente a sua refatoração à luz de boas práticas de engenharia de software e da adoção de ferramentas modernas de desenvolvimento, teste e automação.

Este projeto teve, assim, como objetivo principal a modernização do YAKE!, através da refatoração do seu código-fonte, da introdução de ferramentas de análise de qualidade de código e da criação de uma pipeline de integração e entrega contínuas (CI/CD). Adicionalmente, foi desenvolvido um novo site de documentação, com recurso ao *Fumadocs*, que oferece uma apresentação clara do projeto, do seu funcionamento e da sua documentação técnica.

[1] http://yake.inesctec.pt/
[2] https://pepy.tech/projects/yake

1.1 Motivação

A evolução rápida das tecnologias e das exigências de utilização impõe uma manutenção constante dos sistemas de software. Sem práticas modernas de desenvolvimento, como pipelines de CI/CD e validação automática de código, ferramentas valiosas correm o risco de se tornarem obsoletas ou abandonadas [1]. Este risco é particularmente relevante no contexto de projetos open-source, cuja sustentabilidade depende não apenas da sua utilidade, mas também da sua capacidade de acompanhar as práticas modernas de engenharia de software. Neste contexto, projetos como o YAKE! necessitam de permanente atualização, de modo a assegurar a sua sustentabilidade e relevância no ecossistema de ferramentas de Natural Language Processing (NLP). A modernização do projeto visa ainda facilitar novas contribuições, através da melhoria da estrutura do código e da documentação, removendo barreiras que limitam o crescimento da comunidade. Paralelamente, a adoção de práticas contemporâneas de desenvolvimento contribui para a garantia da qualidade e fiabilidade do software, aspetos essenciais para ferramentas utilizadas em contextos de investigação. Finalmente, esta atualização prepara o projeto para futuras expansões e integrações, assegurando a sua adaptabilidade face à contínua evolução tecnológica [2].

1.2 Contextualização e Problema

O YAKE! (Yet Another Keyword Extractor) [3] é uma ferramenta de extração automática de palavras-chave, independente de domínio e língua, que se destaca pela sua abordagem baseada em heurísticas estatísticas que analisam características do texto para identificar termos relevantes, sem necessidade de recursos externos. Desenvolvido originalmente para suprir a necessidade de uma solução simples e eficaz para identificar termos relevantes em textos, o YAKE! rapidamente ganhou reconhecimento na comunidade de Processamento de Linguagem Natural (PLN) [3, 4, 5].

Não obstante a sua relevância, a ausência de atualizações significativas ao longo do tempo resultava em desafios técnicos que restringiam o potencial de crescimento e manutenção do projeto. A base de código original carecia de uma arquitetura modular e organizada, o que dificultava a compreensão e a manutenção por parte de novos contribuidores. Paralelamente, a inexistência de validações automáticas, nomeadamente testes automatizados e ferramentas de análise de qualidade, aumentava o risco de introdução de bugs. A documentação limitada agravava ainda mais a situação, ao não fornecer ori-

[3] `http://yake.inesctec.pt/`

entações suficientes para utilizadores e desenvolvedores, criando barreiras à adoção e à participação na comunidade. Por fim, o processo de desenvolvimento manual, sem pipelines de integração contínua e entrega contínua (CI/CD), tornava o lançamento de novas versões moroso e sujeito a erros.

1.3 Objetivos

O presente trabalho tem como objetivo principal a refatorização e modernização do projeto YAKE! através da implementação de práticas contemporâneas de engenharia de software. Baseando-se na metodologia proposta no enunciado do projeto, pretende-se transformar o YAKE! num projeto de referência em termos de boas práticas de desenvolvimento open-source.

Os objetivos específicos deste trabalho incluem:

- **Reestruturação arquitetural**: Reorganizar o código base do YAKE!, separando o núcleo do algoritmo dos componentes de demonstração, docker e API, criando repositórios dedicados para cada componente;
- **Implementação de pipelines CI/CD**: Estabelecer pipelines de integração e entrega contínua utilizando ferramentas modernas de DevOps, automatizando processos de teste, validação e distribuição do software;
- **Refatorização do código**: Reestruturar o código seguindo as melhores práticas de Python moderno, incluindo:
  - Utilização de templates cookiecutter para estruturação de projetos Python;
  - Conformidade com padrões modernos de Python (PEP 8);
  - Implementação de ferramentas de linting e formatação de código (Ruff, Pylint, Black);
  - Configuração de GitHub Actions para construção e teste automatizados;
- **Desenvolvimento de testes automatizados**: Criar e automatizar novos testes de resultados, *refatorizar* testes existentes e estabelecer datasets de referência para validação automática, garantindo que os resultados se mantêm consistentes;
- **Melhoria da documentação**: Limpar e reorganizar a documentação, focando exclusivamente no núcleo do YAKE!, revisar o README e desenvolver tutoriais em Jupyter Notebook para facilitar a adoção;

- **Garantia de qualidade**: Implementar práticas de gestão de qualidade de código, incluindo análise estática, cobertura de testes e validação automática, assegurando a fiabilidade e robustez da ferramenta.

- **Desenvolvimento de API RESTful**: Implementar uma API moderna e robusta para o algoritmo YAKE!, permitindo a integração fácil da ferramenta em sistemas externos e facilitando o acesso programático às funcionalidades de extração de palavras-chave.

Estes objetivos visam não apenas modernizar o código base do YAKE!, mas também estabelecer uma infraestrutura robusta que facilite futuras contribuições da comunidade, assegure a qualidade e fiabilidade do software, e prepare o projeto para expansões e integrações tecnológicas futuras. O trabalho pretende transformar o YAKE! num exemplo de boas práticas de desenvolvimento de projetos open-source no domínio do processamento de linguagem natural.

1.4 Contribuições

As principais contribuições deste trabalho foram estruturadas para abordar os problemas identificados. Neste contexto destacam-se os seguintes aspetos fundamentais: **Refatoração estrutural**: Reorganização completa do código-fonte do **YAKE!** (**YAKE!**) de forma modular, limpa e orientada para testes, melhorando significativamente a sua legibilidade e facilidade de manutenção. Esta reorganização da estrutura dos ficheiros e respetivo código estabelece uma base sólida para desenvolvimentos futuros e facilita a compreensão do algoritmo. **Implementação de Continuous Integration (CI)/Continuous Deployment (CD)**: Desenvolvimento de uma *pipeline* completa de integração e entrega contínuas com *GitHub Actions*, automatizando rigorosamente os processos de testes, verificação de código e lançamentos. Esta automação garante a qualidade consistente do software e reduz significativamente os erros humanos no processo de desenvolvimento. **Validação de qualidade**: Introdução de ferramentas de validação automática de qualidade de código, nomeadamente *Ruff* [4], *Black* [5] e *Pylint* [6], para garantir consistência, formatação adequada e deteção precoce de problemas potenciais. Estas ferramentas asseguram a aderência a padrões de codificação estabelecidos e facilitam a manutenção a longo prazo. **Modernização da documentação**: Desenvolvimento

[4] https://docs.astral.sh/ruff/
[5] https://pypi.org/project/black/
[6] https://pypi.org/project/pylint/

de um novo website com *Fumadocs*[7], reunindo informações claras sobre o funcionamento do **YAKE!**, explicações técnicas detalhadas e documentação atualizada sobre o projeto[8]. Esta plataforma oferece uma experiência de utilizador moderna e acessível. **Facilitação de contribuições**: Redução significativa das barreiras à entrada para novos contribuidores através de melhor estruturação, documentação clara e processos automatizados. Esta abordagem promove a colaboração e sustentabilidade do projeto a longo prazo.

1.5 Planeamento e Cronograma

O desenvolvimento do projeto foi estruturado em sete tarefas distintas (T1 a T7), com duração total de 15 semanas. O planeamento 1.1 uma progressão lógica e incremental, começando pela análise e compreensão do código atual do YAKE!, passando pela implementação de pipelines de integração contínua, refatoração do código segundo boas práticas modernas, desenvolvimento de testes automatizados, melhoria da documentação técnica, criação de um website de documentação, e finalizando com a realização do relatório final e preparação da apresentação dos resultados obtidos.

T1: Configuração e Planeamento do Projeto (2 semanas)

- Estudo aprofundado do artigo científico e análise da base de código YAKE! existente;
- Divisão da base de código YAKE!, mantendo apenas o núcleo do algoritmo no repositório principal;
- Separação dos componentes de demonstração, docker e API para repositórios dedicados;
- Configuração do ambiente de desenvolvimento e ferramentas necessárias;

T2: Implementação de Pipeline CI/CD (1 semana)

- Estudo de ferramentas e práticas recentes de DevOps;
- Avaliação e seleção das ferramentas mais adequadas para o projeto (GitHub Actions, Chef, Puppet);

[7] `https://fumadocs.dev/docs/ui`
[8] `https://liaad.github.io/yake/docs/--home`

- Design e implementação dos workflows da pipeline Continuous Integration / Continuous Deployment (CI/CD);

T3: Refatoração de Código (4 semanas)

- Refatoração do código seguindo as melhores práticas de Python moderno;
- Utilização de template cookiecutter para estruturação do projeto Python;
- Implementação de padrões modernos de Python conforme especificação PEP 8;
- Configuração de ferramentas de linting e formatação de código (Ruff);
- Configuração de GitHub Actions para construção e teste automatizados;

T4: Desenvolvimento e Automatização de Testes (3 semanas)

- Criação e automatização de novos testes de resultados;
- Refatoração dos testes existentes seguindo boas práticas;
- Implementação de suporte para datasets de referência para testes automatizados;
- Validação dos datasets de referência e verificação da consistência dos resultados;
- Avaliação da cobertura de testes e implementação de testes adicionais;

T5: Melhoria da Documentação Técnica (1 semana)

- Limpeza e reorganização da documentação existente, focando exclusivamente no núcleo do YAKE!;
- Revisão e atualização do ficheiro README do projeto;
- Criação de tutorial em Jupyter Notebook para facilitar a utilização;

T6: Desenvolvimento de API (2 semanas)

- Design e implementação de uma API RESTful para o algoritmo YAKE!;
- Desenvolvimento da documentação da API;
- Testes de integração e validação da API;

T7: Elaboração do Relatório Final e Preparação da Apresentação (2 semanas)

- Redação do relatório técnico final do projeto;
- Preparação da apresentação dos resultados obtidos;
- Revisão final da documentação e ajustes necessários;
- Validação final de todos os componentes desenvolvidos;

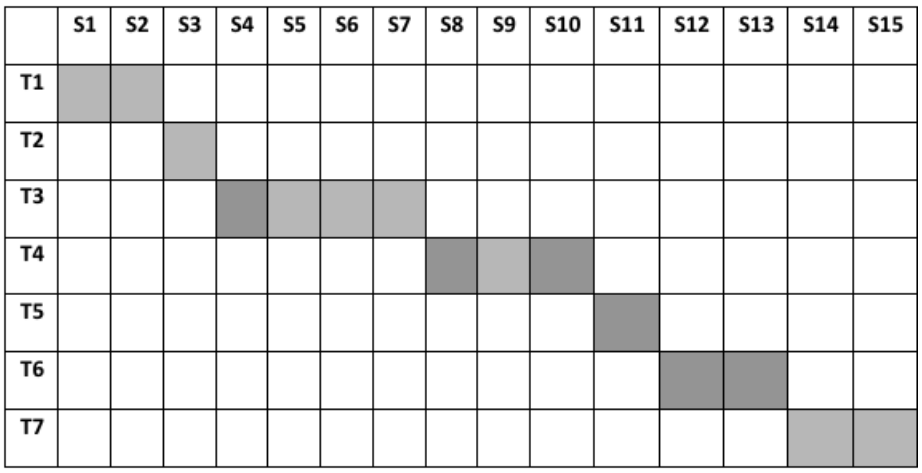

Figura 1.1: Cronograma de execução das tarefas do projeto ao longo das 15 semanas

1.6 Estrutura do Documento

Este documento foi dividido em cinco capítulos. O Capítulo 1 introduz a motivação do projeto, objetivos, *timeline* e a organização do documento. O Capítulo 2 explora o estado da arte, funcionamento do YAKE!, tecnologia e conteúdos relevantes para o projeto. O Capítulo 3 descreve a arquitetura do sistema, tecnologias e métricas de avaliação selecionadas, especificando e explicando as técnicas usadas na refatoração do algoritmo. O Capítulo 4 detalha os workflows de desenvolvimento e automação implementados, assim como as ferramentas de validação de qualidade. O Capítulo 5 apresenta o novo *website* para documentação do projeto e a estrutura da documentação. O Capítulo 6 sintetiza os resultados obtidos, avalia o cumprimento dos objetivos e identifica oportunidades para desenvolvimento futuro.

Capítulo

2

Estado-da-arte

A extração automática de palavras-chave emergiu como uma necessidade fundamental no processamento de linguagem natural face ao crescimento exponencial de conteúdo digital. Esta área de investigação visa identificar automaticamente os termos mais representativos de um documento, facilitando tarefas como categorização [6], recuperação de informação [7], resumo automático [8] e análise de conteúdo [9].

As abordagens tradicionais podem ser categorizadas em três grandes paradigmas: métodos estatísticos, métodos de aprendizagem automática supervisionada, e métodos baseados em grafos. Cada categoria apresenta vantagens e limitações específicas que moldaram a evolução desta área de investigação, culminando no desenvolvimento de métodos mais avançados como o Yet Another Keyword Extractor! (YAKE!) [10].

2.1 Métodos de Extração de Palavras-chave

2.1.1 Métodos Estatísticos

Embora inicialmente desenvolvido no contexto da *recuperação de informação*, o Term Frequency - Inverse Document Frequency (TF.IDF) tem sido amplamente utilizado na extração de palavras-chave, baseando-se na premissa de que termos frequentes num documento específico, mas raros numa coleção de documentos, tendem a ser mais *informativos* [11]. A métrica combina duas componentes fundamentais, ilustrados na Equação 2.1:

$$TF\text{-}IDF(t,d) = TF(t,d) \times IDF(t) \tag{2.1}$$

onde $TF(t,d)$ representa a frequência do termo t no documento d, e $IDF(t) = \log \frac{N}{df(t)}$ penaliza termos que aparecem em muitos documentos da coleção. Esta abordagem apresenta, no entanto, **limitações significativas**. A *dependência obrigatória* de uma coleção de documentos (*corpus*) representa um obstáculo considerável em cenários onde apenas um documento está disponível. Além disso, a *incapacidade de capturar relações semânticas* entre termos limita substancialmente a qualidade dos resultados, particularmente em documentos curtos onde o contexto é reduzido. A *sensibilidade à qualidade e representatividade* do *corpus* constitui outro desafio, uma vez que um *corpus* mal construído pode comprometer significativamente o desempenho do algoritmo.

As abordagens estatísticas para extração de palavras-chave fundamentam-se na análise de *propriedades estatísticas* dos termos no documento, utilizando medidas como *frequência*, *posição* e *co-ocorrência* para determinar a relevância dos candidatos. Entre os métodos mais reconhecidos destaca-se o **Rapid Automatic Keyword Extraction (RAKE)** (*Rapid Automatic Keyword Extraction*) [12], que identifica palavras-chave através da análise de co-ocorrência de palavras num grafo, calculando pontuações baseadas na frequência e grau de cada palavra no grafo construído, e o **YAKE!** (*Yet Another Keyword Extractor*), que será detalhadamente discutido na Secção 2.2, representando uma *evolução significativa* nas técnicas estatísticas não supervisionadas.

2.1.2 Métodos Baseados em Grafos

Inspirado no algoritmo *PageRank* [13], o *TextRank* [14] modela documentos como *grafos* onde nós representam termos e arestas indicam relações de *co-ocorrência*. O algoritmo aplica iterativamente a fórmula 2.2:

$$WS(V_i) = (1-d) + d \times \sum_{V_j \in In(V_i)} \frac{w_{ji}}{\sum_{V_k \in Out(V_j)} w_{jk}} WS(V_j) \tag{2.2}$$

onde d é o fator de amortecimento e w_{ij} representa o peso da aresta.

O TextRank oferece a vantagem de ser **completamente não supervisionado**, eliminando a necessidade de dados de treino. A sua capacidade de capturar *relações estruturais* entre termos e *adaptabilidade* a diferentes tipos de texto tornam-no versátil para diversas aplicações.

Não obstante, o método apresenta *sensibilidade ao tamanho da janela de co-ocorrência*, que pode afetar significativamente os resultados dependendo da configuração escolhida. O *desempenho variável* em documentos curtos e a *complexidade computacional* em textos longos constituem desafios adicionais que limitam a sua aplicabilidade em cenários específicos.

2.1.3 Métodos de Aprendizagem Automática Supervisionada

O Automatic Keyphrase Extraction (KEA), desenvolvido por Witten et al. [15], representa um *marco* na aplicação de *aprendizagem automática supervisionada* à extração de palavras-chave. Este algoritmo utiliza características como TF.IDF e **posição do termo** no documento para treinar classificadores que distinguem entre palavras-chave genuínas e candidatos irrelevantes.

O KEA distingue-se pela utilização de *Naive Bayes* [15] como *classificador base*, incorporando **informação posicional** que reconhece a tendência de termos importantes aparecerem em posições estratégicas do documento. A capacidade de *adaptação a domínios específicos* através de retreino representa uma vantagem significativa para aplicações especializadas.

Contudo, esta abordagem supervisionada introduz **limitações consideráveis**. A *dependência crítica* de dados de treino de qualidade constitui um grande obstáculo, especialmente considerando o *custo elevado* de anotação manual de dados de treino. A *dificuldade de generalização* para novos domínios exige frequentemente novos ciclos de treino, tornando o processo dispendioso e demorado. Estas limitações são particularmente evidentes em *línguas com poucos recursos*, onde a disponibilidade de dados anotados é escassa.

2.1.4 Métodos Baseados em Modelos de Linguagem

O surgimento de *modelos de linguagem pré-treinados* revolucionou a extração de palavras-chave, introduzindo métodos que aproveitam **representações contextuais densas**. O **KeyBERT** [16] exemplifica esta abordagem, utilizando embeddings *BERT* [17] para calcular *similaridades semânticas* entre candidatos a palavras-chave e o documento completo, selecionando termos que maximizam a representatividade semântica.

Métodos mais recentes como **KeyBARThez** [18] e **EmbedRank** [19] estendem esta filosofia, incorporando técnicas de *diversificação* para evitar redundância semântica entre palavras-chave selecionadas. Abordagens baseadas em **Sentence-BERT** [20] e **RoBERTa** [21] têm demonstrado capacidades superiores em domínios específicos, aproveitando a capacidade dos *transformers* [22] para capturar *dependências de longo alcance* e *nuances semânticas subtis*.

Estas abordagens apresentam a vantagem de *não necessitarem de dados de treino específicos* para a tarefa de extração de palavras-chave, beneficiando do **conhecimento pré-adquirido** durante o treino dos modelos base. A capacidade de capturar *similaridades semânticas complexas* permite identificar palavras-chave conceptualmente relacionadas com o conteúdo do documento, mesmo quando não partilham termos lexicais comuns.

2.2 YAKE! - *Yet Another Keyword Extractor*

O algoritmo **YAKE!** [3, 4, 5] foi desenvolvido numa altura onde as limitações dos métodos tradicionais de extração de palavras-chave se tornavam cada vez mais evidentes face ao exponencial crescimento de conteúdo digital. A necessidade de processar documentos individuais sem dependência de grandes *corpora* ou dados de treino específicos criou uma lacuna significativa no panorama de técnicas disponíveis.

Esta ferramenta introduziu uma abordagem completamente nova baseada em *heurísticas estatísticas locais* extraídas exclusivamente do documento, trabalhando assim através de uma metodologia que não necessita de supervisão ou *corpora* de textos, facilitando a sua aplicação para várias linguagens ou domínios. Para além disso, é capaz de estimar um grau de relevância para as palavras-chave extraídas através de um sistema de pontuação inversamente proporcional à importância do termo.

Este algoritmo revolucionou a área ao demonstrar que características puramente intrínsecas ao documento podem ser suficientes para identificar termos relevantes com elevada precisão. A ausência de dependências externas torna o **YAKE!** particularmente adequado para cenários onde apenas um documento está disponível ou onde a construção de um *corpus* representativo é impraticável.

2.2.1 Arquitetura e *Pipeline* do YAKE!

A arquitetura do **YAKE!** [4] é composta por cinco componentes principais: (1) **configuração e inicialização**, são definidos os parâmetros essenciais como o tamanho da janela de coocorrência (w), o número máximo de *n-gramas* (n) e o limiar de deduplicação (θ); (2) **pré-processamento**, constitui a etapa seguinte, envolvendo normalização e preparação do texto através de segmentação de frases, tokenização e identificação de *stopwords*. Esta fase é crucial para garantir a qualidade dos dados que alimentam as etapas subsequentes da *pipeline*; (3) **extração de características**, representa o núcleo algorítmico do sistema, onde são avaliadas cinco características estatísticas fundamentais que capturam diferentes aspetos da importância de cada termo. O **ranking e filtros** combinam estas características numa pontuação composta, gerando n-gramas e ordenando os candidatos por relevância; (4) **ranking e filtragem**, representa o núcleo algorítmico do sistema, onde são avaliadas cinco características estatísticas fundamentais que capturam diferentes aspetos da importância de cada termo. O **ranking e filtros** combinam estas características numa pontuação composta, gerando n-gramas e ordenando os candidatos por relevância; (5) **deduplicação**, remove similaridades através de medidas

de distância, garantindo que a lista final de palavras-chave não contém redundâncias que possam comprometer a efetividade dos resultados.

2.2.2 Pré-processamento e Identificação de Termos Candidatos

O processo inicia-se com a divisão do texto em frases utilizando o segmentador baseado em regras **segtok** [1], especificamente desenvolvido para textos indo-europeus. Esta escolha tecnológica reflete uma abordagem pragmática que reconhece as especificidades linguísticas dos textos alvo.

Cada frase é posteriormente dividida em *chunks* sempre que existe pontuação e tokenizada, permitindo não apenas a identificação de palavras mas também o isolamento de informação ruidosa como URLs, emails ou caracteres especiais. Esta abordagem granular garante que elementos não textuais não interfiram com a análise das palavras-chave.

Durante a tokenização, cada *token* é anotado com delimitadores específicos que refletem a sua natureza: `<d>` para dígitos ou números, `<u>` para conteúdo não analisável (URLs, emails, caracteres especiais), `<a>` para acrónimos (termos apenas com letras maiúsculas), `<U>` para termos que começam com maiúscula (excluindo início de frase), e `<p>` para conteúdo analisável restante.

Esta anotação é crucial para a posterior seleção de termos candidatos, sendo considerados apenas *tokens* marcados como `<a>`, `<U>` ou `<p>`. A decisão de considerar automaticamente *tokens* com menos de três caracteres como *stopwords* reflete uma heurística prática baseada na observação de que termos muito curtos raramente constituem palavras-chave significativas.

2.2.3 Extração de Características Estatísticas

O núcleo do **YAKE!** reside na computação de cinco características estatísticas que capturam diferentes aspetos da importância de cada termo 1-grama. Esta abordagem multifacetada reconhece que a relevância de um termo não pode ser adequadamente capturada por uma única métrica.

A característica T_{Case} **(Capitalização)** mede a importância baseada na capitalização do termo, assumindo que termos em maiúsculas tendem a ser mais relevantes 2.3:

$$T_{Case} = \frac{\max(TF(U(t)), TF(A(t)))}{\ln(TF(t))} \tag{2.3}$$

[1] https://pypi.org/project/segtok/

onde $TF(U(t))$ é o número de ocorrências do termo começando com maiúscula, $TF(A(t))$ é o número de vezes que aparece como acrónimo, e $TF(t)$ é a frequência total. Esta heurística reconhece convenções tipográficas comuns onde termos importantes são frequentemente capitalizados.

A $T_{Position}$ **(Posição no Documento)** baseia-se no pressuposto bem estabelecido de que termos relevantes aparecem mais frequentemente no início do documento:

$$T_{Position} = \ln(\ln(3 + \text{Median}(Sen_t))) \tag{2.4}$$

onde Sen_t é o conjunto de posições das frases onde o termo ocorre. O duplo logaritmo suaviza diferenças entre termos com grandes variações de mediana, evitando que diferenças extremas de posição dominem excessivamente a pontuação final.

A característica T_{FNorm} **(Frequência Normalizada)** normaliza a frequência do termo pela média e desvio padrão das frequências de termos não-*stopword*:

$$T_{FNorm} = \frac{TF(t)}{\text{MeanTF} + 1 \times \sigma} \tag{2.5}$$

Esta normalização evita enviesamento para documentos longos e valoriza termos cuja frequência está acima da média balanceada pelo desvio padrão, proporcionando uma medida mais equilibrada da importância relativa.

A T_{Rel} **(Relação Contextual)** mede a dispersão do termo relativamente ao seu contexto, assumindo que termos que co-ocorrem com muitos termos diferentes tendem a ser menos específicos:

$$T_{Rel} = 1 + (D_L + D_R) \times \frac{TF(t)}{\text{MaxTF}} \tag{2.6}$$

onde D_L e D_R representam a dispersão à esquerda e direita do termo numa janela de tamanho w, respetivamente. Esta métrica captura a intuição de que termos altamente conectados podem ser menos discriminativos.

Finalmente, a $T_{Sentence}$ **(Distribuição por Frases)** quantifica a distribuição do termo ao longo de diferentes frases:

$$T_{Sentence} = \frac{SF(t)}{\#\text{Sentences}} \tag{2.7}$$

onde $SF(t)$ é o número de frases distintas onde o termo aparece. Esta característica reconhece que termos que aparecem consistentemente ao longo do documento tendem a ser mais centrais ao seu conteúdo.

2.2.4 Calculo da Pontuação de Termos

As cinco características são combinadas numa pontuação única $S(t)$ que reflete a importância do termo 1-grama através da seguinte formula:

$$S(t) = \frac{T_{Rel} \times T_{Position}}{T_{Case} + \frac{T_{FNorm} + T_{Sentence}}{T_{Rel}}} \tag{2.8}$$

Esta fórmula foi concebida de forma a *valores menores indiquem maior relevância*, uma função que simplifica a interpretação dos resultados. A divisão de T_{FNorm} e $T_{Sentence}$ por T_{Rel} mitiga termos que, apesar de frequentes, são contextualmente pouco específicos, garantindo que a especificidade contextual é adequadamente valorizada.

2.2.5 Geração de N-gramas e Pontuação de Palavras-chave Candidatas

O algoritmo gera sequências contíguas de 1 a n termos, mas apenas considerando sequências que satisfazem critérios rigorosos de qualidade. As sequências devem pertencer à mesma frase e *chunk*, não podem começar nem terminar com *stopwords*, e devem ser constituídas apenas por termos classificados como <a>, <U> ou <p>.

Para cada palavra-chave candidata kw, a pontuação é calculada através de uma fórmula que combina as pontuações individuais dos termos constituintes:

$$S(kw) = \frac{\prod_{t \in kw} S(t)}{KF(kw) \times \left(1 + \sum_{t \in kw} S(t)\right)} \tag{2.9}$$

onde $KF(kw)$ é a frequência da palavra-chave candidata. Esta formulação equilibra a qualidade individual dos termos com a frequência da sequência completa. Para *stopwords* interiores (como "of"em "language of thought"), aplica-se uma abordagem baseada em probabilidades de bigramas para mitigar o impacto negativo que estes termos funcionais poderiam ter na pontuação.

2.2.6 Deduplicação e *Ranking* Final

O processo termina com uma fase de deduplicação que remove candidatos similares utilizando medidas de distância como Levenshtein [2], Jaro-Winkler

[2] https://www.geeksforgeeks.org/dsa/introduction-to-levenshtein-distance/

[3] ou *sequence matcher* [4]. Entre dois candidatos considerados similares (distância superior ao limiar θ), mantém-se aquele com menor pontuação $S(kw)$, garantindo que variações redundantes não distorçam os resultados finais.

A lista final é ordenada por pontuação ascendente, onde as palavras-chave mais relevantes ocupam as primeiras posições. Esta ordenação permite aos utilizadores selecionar facilmente o número de palavras-chave desejado para as suas aplicações específicas.

2.2.7 Avaliação

Campos et al. [4] avaliaram o desempenho do **YAKE!** em múltiplos *datasets*, demonstrando sua superioridade em relação aos métodos tradicionais como TF-IDF e KEA. Os resultados revelaram que o **YAKE!** é particularmente eficaz em documentos curtos e médios, onde métodos dependentes de *corpora* frequentemente falham devido à insuficiência de contexto estatístico.

A eficácia em textos de domínios especializados representa outra vantagem significativa, especialmente em cenários onde não há dados de treino disponíveis. A capacidade de funcionar efetivamente em múltiplas linguagens sem necessidade de adaptação demonstra a robustez da abordagem baseada em características estatísticas universais.

2.3 Engenharia de Software para Ferramentas de Processamento de Linguagem Natural

O desenvolvimento de ferramentas de processamento de linguagem natural (NLP) modernas requer a aplicação de princípios sólidos de engenharia de *software*, que transcendem a mera implementação de algoritmos e abrangem aspetos como arquitetura, qualidade, manutenibilidade e experiência do utilizador.

2.3.1 Arquiteturas e Padrões de Desenvolvimento

As arquiteturas modernas para ferramentas de NLP privilegiam a modularidade e a separação de responsabilidades. Martin Fowler [1] estabelece os fundamentos dos padrões arquiteturais que promovem a reutilização e manutenibilidade. O padrão de *pipeline* é particularmente relevante, permitindo o processamento sequencial de dados através de componentes especializados

[3] https://en.wikipedia.org/wiki/Jaro-Winkler

[4] https://www.educative.io/answers/what-is-sequencematcher-in-python

[23]. Implementações modernas como `spaCy` [24] e `NLTK` [25] exemplificam como esta abordagem facilita a composição de funcionalidades complexas através de componentes simples e testáveis.

2.3.2 Qualidade de Software e Práticas Modernas

A garantia de qualidade em *software* científico requer práticas específicas que considerem tanto a correção dos algoritmos quanto a robustez da implementação. Greg Wilson [26] identifica práticas essenciais para desenvolvimento de *software* científico, incluindo controlo de versão, testes automatizados e documentação adequada. A integração contínua (CI) e entrega contínua (CD), cujos princípios teóricos foram estabelecidos por Martin Fowler [27], constituem práticas fundamentais para garantir a qualidade e estabilidade do *software*.

O *GitHub Actions* [28] emerge como plataforma líder para implementação de *pipelines* CI/CD, oferecendo integração nativa com repositórios `Git` e suporte extensivo para linguagens de programação diversas. *Jenkins* [29] e *GitLab CI* [30] constituem alternativas consolidadas no mercado, proporcionando funcionalidades avançadas para organizações com requisitos específicos de *deployment* e segurança.

A **análise estática** de código constitui outra prática essencial para manter qualidade e consistência. Para *Python*, linguagem predominante em NLP, ferramentas como `Pylint` [31] e `Flake8` [32] oferecem análise de qualidade de código, detetando potenciais problemas e violações de convenções. `Black` [33] e `isort` [34] garantem formatação consistente e organização de importações, reduzindo discussões sobre estilo e aumentando a legibilidade do código.

Capítulo

3

Refatoração

A implementação original do algoritmo **YAKE!** apresentava características típicas de sistemas desenvolvidos com foco principal na *funcionalidade*, sem considerar profundamente aspectos que facilitam a manutenção e melhoram *extensibilidade*. Este capítulo descreve o processo sistemático de *refatoração* aplicado ao código, começando pela análise da estrutura original, identificação dos problemas específicos encontrados, e posteriormente apresentando as estratégias de *refatoração* implementadas e os resultados obtidos.

3.1 Métricas de Qualidade de Código

Para orientar o processo de *refatoração*, foi utilizada a ferramenta *Pylint* [31] como métrica principal de qualidade de código. O *Pylint* é uma ferramenta de análise estática que avalia código Python segundo múltiplos critérios, incluindo conformidade com convenções de nomenclatura (PEP 8), complexidade de métodos, qualidade da documentação e aderência a boas práticas de programação. A ferramenta atribui uma pontuação entre 0 e 10, onde valores superiores a 8 indicam código de alta qualidade adequado para ambientes de produção.

A análise inicial utilizando o *Pylint* revelou uma pontuação de apenas **3.21/10**, indicando múltiplos problemas de qualidade de código que necessitavam intervenção sistemática para alcançar um código limpo e de fácil manutenção.

3.2 Estrutura Original do Sistema YAKE!

Antes de apresentar os problemas identificados, é fundamental compreender a arquitetura original do sistema **YAKE!**. O código encontrava-se organizado numa estrutura relativamente simples, mas com uma distribuição inadequada de *responsabilidades*. No contexto da arquitetura de software, o termo "responsabilidade"refere-se ao conjunto específico de funções, propósitos ou tarefas que um código (como uma classe ou módulo) deve desempenhar. Idealmente, cada componente do sistema deve ter *responsabilidades* bem definidas e coesas, conforme destacado por Martin Fowler [1]. No entanto, o sistema original apresentava responsabilidades dispersas e definidas vagamente, o que dificulta significativamente a manutenção, compreensão e evolução do código. Esta situação problemática é ilustrada no excerto de Código 3.1.

```
yake/
          StopwordsList/
                stopwords_ar.txt
                stopwords_en.txt
                ... (outros idiomas)
          yake.py
          datarepresentation.py
          cli.py
          Levenshtein.py
          __init__.py
```

Excerto de Código 3.1: Estrutura original de ficheiros do YAKE!

O sistema original concentrava-se em dois ficheiros principais:

- **yake.py**: Continha a classe principal `KeywordExtractor` responsável pela coordenação do algoritmo, configuração de parâmetros e interface pública para extração de palavras-chave. Este ficheiro também incluía lógica de pré-processamento de texto e seleção de algoritmos de deduplicação.
- **datarepresentation.py**: Concentrava todas as estruturas de dados fundamentais do sistema, incluindo as classes `DataCore` (processamento central do documento), `SingleWord` (representação de termos individuais) e `ComposedWord` (candidatos a palavras-chave compostas). Este ficheiro totalizava mais de 800 linhas de código.

Esta organização criava um sistema onde a lógica estava dispersa entre poucos ficheiros com múltiplas *responsabilidades*, dificultando a manutenção e extensibilidade do código.

3.3 Problemas Identificados

A análise detalhada do código original revelou múltiplos problemas estruturais e de qualidade que comprometiam a manutenção do sistema.

3.3.1 Estrutura Monolítica

O principal problema identificado foi a concentração excessiva de *responsabilidade* no ficheiro `datarepresentation.py`. Este ficheiro continha três classes principais (`DataCore`, `ComposedWord` e `SingleWord`) totalizando mais de **800 linhas de código**. A classe `DataCore`, responsável pelo processamento central do documento, concentrava simultaneamente as *responsabilidade* de *tokenização* de texto, gestão de estruturas de dados, cálculo de estatísticas e construção de grafos de co-ocorrência.

Esta abordagem *monolítica* criava dificuldades significativas para navegação, compreensão e manutenção do código [1]. Como observa Fowler [1]: *"Large classes are a prime candidate for Extract Class. Often a class starts small and well designed, but grows over time"*.

Classes extensas violam o princípio Single Repository Principle (SRP) (*Single Responsibility Principle*) e dificultam a aplicação de testes unitários, uma vez que múltiplas *responsabilidade* ficam entrelaçadas numa única estrutura de dados [1].

3.3.2 Excesso de Atributos de Instância

As classes do ficheiro `datarepresentation.py` sofriam de acúmulo excessivo de atributos. A análise estática identificou o problema **R0902** (*Too many instance attributes*) em múltiplas classes. A classe `SingleWord`, por exemplo, apresentava 17 atributos diretos, excedendo significativamente o limite recomendado de 7 atributos por classe estabelecido pelo *Pylint*, como demonstrado no excerto de Código 3.2.

```
# Ficheiro: datarepresentation.py - Classe single_word com
   17+ atributos diretos
class single_word(object):
    def __init__(self, word, left_co_occur, right_co_occur,
                 freq, left_freq, right_freq):
        self.vocab = word
        self.tf = freq
        self.WFreq = freq           # R0902: Too many
           instance attributes (17/7)
        self.WCase = 1
        self.WPos = left_freq
        self.WSpread = right_freq
```

```
        self.WRel = 1
        self.PL = 0
        self.PR = 0
        self.H = 0
        self.G = None
        self.sentences = []
        self.left_co_occur = left_co_occur
        self.right_co_occur = right_co_occur
        self.co_occur = []
        self.id = 0
        self.sig = 1
```

Excerto de Código 3.2: Classe `SingleWord` do ficheiro `datarepresentation.py` com excesso de atributos

O excesso de atributos na classe `SingleWord` criava dificuldades para compreensão da textitresponsabilidade da classe e tornava complexa a inicialização e manutenção dos objetos.

3.3.3 Ausência de Documentação

Tanto o ficheiro `yake.py` quanto `datarepresentation.py` sofriam de **ausência completa de documentação**, violando múltiplas regras do *Pylint* relacionadas com a falta de docstrings em módulos (C0114), classes (C0115) e métodos (C0116), como ilustrado no excerto de Código 3.3.

```
# Ficheiro: datarepresentation.py - Sem documentacao (C0114,
    C0115, C0116)
class DataCore(object):
    def __init__(self, windowsSize=1, tagsToDiscard=set()):
        # Sem docstring da classe ou metodo
        # ...

    def build_data_graph(self, windowsSize):
        # Metodo sem documentacao sobre parametros ou
           funcionalidade
        # ...

    def get_term(self, str_word):
        # Metodo sem documentacao sobre proposito ou valor
           de retorno
        # ...
```

Excerto de Código 3.3: Código original do ficheiro `datarepresentation.py` sem documentação

A ausência de documentação no ficheiro `datarepresentation.py` tornava particularmente difícil compreender o propósito e funcionamento das

classes principais do sistema, especialmente considerando a complexidade das estruturas de dados envolvidas.

3.3.4 Configuração Dispersa

No ficheiro `yake.py`, os parâmetros de configuração eram armazenados como atributos individuais da classe `KeywordExtractor`, resultando num construtor extenso com múltiplos parâmetros explícitos, como ilustrado no excerto de Código 3.4

```
# Ficheiro: yake.py - Construtor com multiplos parametros
def __init__(self, lan="en", n=3, dedup_lim=0.9, dedup_func=
    "seqm",
                 window_size=1, top=20, features=None, stopwords
                    =None):
```

Excerto de Código 3.4: Construtor extenso no ficheiro `yake.py` com configuração dispersa

Esta abordagem no ficheiro `yake.py` dificultava a manutenção e a adição de novos parâmetros, além de criar dependências *rígidas* entre componentes, problema identificado como indicativo da necessidade de aplicação do padrão *Parameter Object* [1].

3.3.5 Lógica Condicional Complexa

O ficheiro `yake.py` apresentava estruturas condicionais *extensas* para seleção de algoritmos e métodos. A escolha da função de deduplicação, por exemplo, era realizada através de blocos `if-elif` extensos, criando código duplicado e dificultando a *extensibilidade* do sistema.

3.3.6 Violações de Convenções de Nomenclatura

Ambos os ficheiros principais violavam sistematicamente as convenções *Python* (Python Enhancement Proposal 8 (PEP8)) [35], conforme identificado pelo *Pylint* e apresentado no excerto de Código 3.5.

```
# Ficheiro: datarepresentation.py - Violacoes identificadas
    (C0103):
# - Class name "composed_word" doesn't conform to PascalCase
# - Class name "single_word" doesn't conform to PascalCase
# - Attribute name "G" doesn't conform to snake_case
# - Method name "getTerm" doesn't conform to snake_case

class composed_word(object):  # C0103: doesn't conform to
    PascalCase
```

```
    def getTerm(self, term):  # C0103: doesn't conform to
        snake_case
         return self.G[term]   # C0103: Attribute name "G"
            doesn't conform

# W0102: Dangerous default value set() as argument
def __init__(self, windowsSize=1, tagsToDiscard=set()):
    self.tagsToDiscard = tagsToDiscard
```

Excerto de Código 3.5: Violações de nomenclatura no ficheiro `datarepresentation.py`

Estas violações no ficheiro `datarepresentation.py` comprometiam a legibilidade e consistência do código, dificultando a colaboração entre desenvolvedores e a manutenção a longo prazo.

3.4 Estratégias de *refatoração* Aplicadas

Com base na análise dos problemas identificados e seguindo as práticas recomendadas [1], foram aplicadas técnicas **específicas** de *refatoração* que resultaram numa melhoria *sistemática* da qualidade do código.

3.4.1 Consolidação de Atributos através do Padrão *State Object*

Para mitigar o problema de projeto identificado pelo código **R0902** (*"Too many instance attributes"*), que indica excesso de atributos numa classe e compromete a sua legibilidade e manutenção, foi adotado o padrão *State Object*. Esta estratégia de *encapsulamento* organiza os atributos relacionados numa única estrutura hierárquica de dicionários, reunindo **configurações**, **estatísticas** e **coleções** num único atributo denominado `_state`.

Do ponto de vista de *engenharia de software*, o padrão *State Object*, descrito por Martin Fowler [1], promove **coerência**, **modularidade** e **extensibilidade**, permitindo que diferentes categorias de dados sejam agrupadas de forma lógica. Esta abordagem favorece a *separação de responsabilidade*, reduz o *aglormerado* entre componentes e facilita a inspeção ou serialização do estado interno da instância.

O Código 3.6 apresenta a implementação desta solução na nova versão da classe `DataCore`:

```
# Ficheiro: data/core.py - Consolidacao usando State Object
class DataCore:
    def __init__(self, text, stopword_set, config=None):
        # Initialize default configuration if none provided
        if config is None:
            config = {}

        # Extract configuration values with appropriate defaults
        windows_size = config.get("windows_size", 2)
        n = config.get("n", 3)
        tags_to_discard = config.get("tags_to_discard", set(["u", "
            d"]))
        exclude = config.get("exclude", set(string.punctuation))

        # Initialize the state dictionary containing all component
            data structures
        self._state = {
            # Configuration settings
            "config": {
                "exclude": exclude,  # Punctuation and other
                    characters to exclude
                "tags_to_discard": tags_to_discard,  # POS tags to
                    ignore
                "stopword_set": stopword_set,  # Set of stopwords
                    for filtering
            },
            # Text corpus statistics
            "text_stats": {
                "number_of_sentences": 0,  # Total count of
                    sentences
                "number_of_words": 0,  # Total count of processed
                    words
            },
            # Core data collections for analysis
            "collections": {
                "terms": {},  # Dictionary mapping terms to
                    SingleWord objects
                "candidates": {},  # Dictionary mapping keywords to
                     ComposedWord objects
                "sentences_obj": [],  # Nested list of processed
                    sentence objects
                "sentences_str": [],  # List of raw sentence
                    strings
                "freq_ns": {},  # Frequency distribution of n-grams
                     by length
            },
            # Graph for term co-occurrence analysis
            "g": nx.DiGraph(),  # Directed graph for co-occurrences
        }
```

Excerto de Código 3.6: Implementação do padrão State Object na classe `DataCore` refatorada

Esta reorganização reduziu significativamente o número de atributos diretos da classe, agrupando-os logicamente e facilitando a gestão do estado

interno.

3.4.2 Implementação de *Property Accessors* para Compatibilidade

Para manter compatibilidade entre código existente e as novas alterações, foram implementados *property accessors* que preservam a interface original enquanto utilizam a nova estrutura interna. Esta técnica, conhecida como *Encapsulate Field* [1], permite a *refatoração incremental* sem quebrar dependências externas, como demonstrado no excerto de Código 3.7. Os *property accessors* representam uma solução elegante para modificar a implementação interna de uma classe sem afetar o código cliente dependente. Esta abordagem garante transparência na migração, permitindo que o código continue a aceder aos atributos utilizando a mesma sintaxe, independentemente da mudança de atributos diretos para um dicionário de estado centralizado. A transparência elimina modificações extensivas no código dependente e reduz drasticamente o risco de introdução de *bugs* durante a *refatoração*. A implementação oferece um ponto de controlo centralizado onde cada propriedade pode incluir validação, *logging* ou transformações específicas, proporcionando flexibilidade arquitetural ao permitir que a estrutura interna evolua independentemente da interface externa. O Código 3.7 demonstra diferentes padrões: propriedades *read-only* como *exclude* e *terms*, propriedades *read-write* como *number-of-sentences* e acesso direto como *g* para o grafo NetworkX. Esta estratégia alinha-se com as melhores práticas de engenharia de software, nomeadamente o princípio da substituição de Liskov [36] e contratos de interface, garantindo comportamento consistente independentemente da implementação subjacente.

```
# Ficheiro: data/core.py - Property accessors para
   compatibilidade
@property
def exclude(self):
    """Get the set of characters to exclude from processing.
       """
    return self._state["config"]["exclude"]

@property
def terms(self):
    """Get the dictionary of SingleWord objects representing
        individual terms."""
    return self._state["collections"]["terms"]

@property
def g(self):
```

```
    """Get the directed graph representing term co-
       occurrences."""
    return self._state["g"]

@property
def number_of_sentences(self):
    """Get the total number of sentences in the document."""
    return self._state["text_stats"]["number_of_sentences"]

@number_of_sentences.setter
def number_of_sentences(self, value):
    """Set the total number of sentences in the document."""
    self._state["text_stats"]["number_of_sentences"] = value
```

Excerto de Código 3.7: Property accessors para compatibilidade retroativa na classe `DataCore`

3.4.3 Estratégia *Replace Conditional with Polymorphism*

Para eliminar as estruturas condicionais complexas presentes no ficheiro `yake.py`, foi aplicada a técnica *"Replace Conditional with Polymorphism"* [1] através do uso de dicionários de funções. Esta abordagem mostrou-se particularmente **eficaz** em *Python* devido à natureza de *primeira classe* das funções na linguagem, como ilustrado do excerto de Código 3.8:

```
# Ficheiro: core/yake.py - Substituicao de if-elif por
   mapeamento direto
def _get_dedup_function(self, dedup_func):
    """Get deduplication function based on specified
       algorithm."""
    dedup_functions = {
        "seqm": seqm,
        "jaro": jaro,
        "jaro_winkler": jaro_winkler
    }
    return dedup_functions.get(dedup_func, seqm)
```

Excerto de Código 3.8: Substituição de estruturas condicionais por dicionário de funções

3.5 Nova Arquitetura Modular

A *refatoração* resultou numa arquitetura **significativamente** mais modular e organizada, substituindo a estrutura original de dois ficheiros principais por uma hierarquia bem definida de módulos especializados.

3.5.1 Estrutura Hierárquica Reorganizada

A nova estrutura organizou os componentes seguindo o princípio da *responsabilidade única*, como apresentado no excerto de Código 3.9.

```
yake/
          core/                          # Nucleo algoritmico
                StopwordsList/
                highlight.py
                Levenshtein.py
                yake.py
          data/                          # Estrutura de dados
                __init__.py
                composed_word.py
                core.py
                single_word.py
                utils.py
          __init__.py
          cli.py
```

Excerto de Código 3.9: Estrutura modular final após refatoração

Esta organização separou claramente o **núcleo algorítmico** (`core/`) das **estruturas de dados** (`data/`), proporcionando maior clareza arquitetural e facilitando a manutenção modular [37].

3.5.2 Modularização por Responsabilidade

Cada módulo passou a ter uma *responsabilidade específica* e **bem definida**:

- **data/core.py**: Concentra-se no processamento central do texto e gestão de estruturas de dados principais, substituindo a parte correspondente do ficheiro original `datarepresentation.py`.
- **data/single_word.py**: Dedica-se exclusivamente à representação e análise estatística de termos individuais, extraída e refinada a partir da classe original `SingleWord`.
- **data/composed_word.py**: Responsável pela gestão de candidatos a palavras-chave compostas e suas métricas, baseada na classe original `ComposedWord` mas significativamente melhorada.
- **data/utils.py**: Organiza as funções utilitárias para processamento de texto que anteriormente estavam dispersas pelos ficheiros principais.
- **core/yake.py**: Mantém a interface principal do algoritmo mas com *responsabilidades* mais focadas na coordenação dos componentes.

Esta separação modular facilita a realização de *testes unitários independentes* para cada componente e permite a evolução de cada módulo sem afetar os demais.

3.6 Implementação de Documentação Completa

A etapa **final** e *crucial* da *refatoração* foi a adição de documentação completa seguindo padrões PYTHON estabelecidos pela *PEP 257* [38]. Esta melhoria revelou-se *fundamental* para alcançar a pontuação final de **8.45/10**. O Código 3.10 exemplifica a qualidade da documentação implementada.

```
# Ficheiro: data/core.py - Documentacao completa
class DataCore:
    """
    Core data representation for document analysis and
       keyword extraction.

    This class processes text documents to identify
       potential keywords based on
    statistical features and contextual relationships
       between terms. It maintains
    the document's structure, processes individual terms,
       and generates candidate
    keywords using graph-based co-occurrence analysis.

    The class implements the State Object pattern to manage
       complex internal state
    while providing a clean external interface through
       property accessors.

    Attributes:
        See property accessors below for available
           attributes.
    """

    def __init__(self, text, stopword_set, config=None):
        """
        Initialize the data core with text and configuration
           .

        Args:
            text (str): The input text to analyze for
               keyword extraction
            stopword_set (set): A set of stopwords to filter
                out non-content words
            config (dict, optional): Configuration options
               including:
```

```
            - windows_size (int): Size of word window
                for co-occurrence (default: 2)
            - n (int): Maximum length of keyword phrases
                (default: 3)
            - tags_to_discard (set): POS tags to ignore
                (default: {"u", "d"})
            - exclude (set): Characters to exclude (
                default: string.punctuation)

        Raises:
            TypeError: If text is not a string or
                stopword_set is not a set
            ValueError: If configuration parameters are
                outside valid ranges
        """
```

Excerto de Código 3.10: Documentação completa implementada na classe `DataCore`

3.7 Avaliação dos Resultados

O processo de *refatoração* foi monitorizado através de métricas **objetivas** do *Pylint* [31], demonstrando melhorias *consistentes* ao longo do tempo. A Tabela 3.1 apresenta a progressão na pontuação, culminando na implementação de documentação completa que elevou **significativamente** a pontuação final.

Tabela 3.1: Progressão das métricas de qualidade durante a *refatoração*

Período	**Pontuação Pylint**	**Principais Melhorias**
Inicial	3.21/10	Baseline
Fevereiro 2025	3.94/10	Tamanhos de linha e `isinstance()`
Fevereiro 2025	5.07/10	Indexação e especificação de exceções
Fevereiro 2025	6.57/10	Geradores e `enumerate()`
Março 2025	6.59/10	Redução de código duplicado
Março 2025	6.86/10	Melhoria na documentação
Março 2025	7.23/10	Separação de atributos
Junho 2025	**8.45/10**	**Documentação completa (docstrings)**

A progressão *constante* da pontuação demonstra a eficácia da abordagem *incremental* de *refatoração*. O resultado final de 8.45 pontos demonstra aderência a padrões **internacionais** de qualidade de software estabelecidos pela ISO/IEC 25010 [39].

Tabela 3.2: Comparação final das métricas de qualidade

Métrica	Antes	Depois	Melhoria
Pontuação *Pylint*	3.21/10	8.45/10	+163%
Documentação	0%	100%	+100%
Atributos máximos por classe	17+	7	-59%
Conformidade PEP8	Parcial	Completa	100%
Modularização	Monolítica	Modular	Estruturação

3.8 Validação e Testes

Para assegurar que o processo de *refatoração* não introduziu defeitos funcionais, foi desenvolvida um conjunto abrangente de testes. Os *testes funcionais* verificaram que os resultados de extração de palavras-chave permaneceram iguais aos obtidos pela versão original, enquanto os *testes de performance* confirmaram a ausência de degradação significativa no desempenho do sistema. Adicionalmente, os *testes de integração* validaram o funcionamento correto da comunicação entre os módulos *refatorados*. A análise de *cobertura de testes*, realizada através do comando `uv run pytest -vv --cov=yake tests/`, revelou uma cobertura de **74%** do código após a conclusão da *refatoração*. Esta cobertura traduz-se em **572 linhas testadas** de um total de **772** *statements*, representando uma base sólida para a validação da funcionalidade do sistema. Os módulos `yake/__init__.py` e `yake/core/Levenshtein.py` alcançaram *cobertura completa* de **100%**, enquanto os módulos críticos `yake/data/core.py` e `yake/data/utils.py` apresentaram coberturas elevadas de **92%** e **97%**, respetivamente. Em contraste, identificaram-se *oportunidades de melhoria* nos módulos `yake/data/composed_word.py` com **49%** de cobertura e `yake/cli.py` *sem cobertura de testes*.

A distribuição atual da cobertura concentra-se adequadamente nos *componentes essenciais* do algoritmo **YAKE!**, garantindo que as *funcionalidades centrais* estão devidamente validadas. A ausência de testes no módulo `yake/cli.py` é aceitável considerando que este contém principalmente código de *interface de linha de comando*, enquanto o módulo `yake/data/composed_word.py` representa uma *área prioritária* para futuras melhorias na cobertura de testes.

3.9 Conclusão

A transformação sistemática do código **YAKE!** de uma pontuação inicial de **3.21/10** para **8.45/10** no *Pylint*, representando uma melhoria de **163%**, demonstra que técnicas estruturadas de *refatoração*, quando aplicadas sistematicamente e orientadas por métricas objetivas, podem produzir melhorias substanciais na qualidade e manutenção do *software*. A *refatoração* transformou um sistema originalmente composto por dois ficheiros principais com *responsabilidades* sobrepostas numa arquitetura modular bem estruturada, onde cada componente tem *responsabilidades* claramente definidas. Esta transformação não apenas melhorou as métricas de qualidade, mas também estabeleceu uma base sólida para futuras extensões e manutenções do sistema **YAKE!**.

Capítulo

4

Implementação de CI/CD Pipeline

A implementação de uma *pipeline* de CI/CD robusto constitui um pilar fundamental para o desenvolvimento sustentável de software de qualidade. No contexto da *refatoração* do YAKE!, a criação de um sistema automatizado de testes, validação e *deploy* tornou-se essencial para garantir que as melhorias de código não introduzissem regressões funcionais e que a qualidade fosse mantida consistentemente ao longo do desenvolvimento. Este capítulo detalha a estratégia Development Operations (DevOps) implementada, fundamentada em princípios estabelecidos na literatura, descrevendo os *workflows* automatizados, os processos de validação e as ferramentas de monitorização que suportam o ciclo de vida do software YAKE!.

4.1 Fundamentação Teórica e Estratégia DevOps

4.1.1 Princípios Fundamentais

A estratégia DevOps adotada neste projeto baseia-se em princípios consolidados na literatura de engenharia de software, especificamente nas obras de Martin Fowler [27] sobre *Continuous Integration* e *Delivery*. Esta abordagem fundamenta-se numa filosofia holística que visa quebrar as barreiras tradicionais entre desenvolvimento e lançamento. O conceito de *Integração Contínua*, inicialmente proposto por Kent Beck [40] no contexto da Programação Extrema, constitui a base desta abordagem. Esta prática implementa a validação automática de cada mudança introduzida no código através de *pipelines* de integração robustas, garantindo que todas as alterações sejam sistematicamente testadas e validadas antes da sua incorporação no repositório principal. A implementação de *Qualidade como Código* (*Quality as Code*) repre-

senta uma evolução paradigmática baseada nos trabalhos de Gene Kim [41] sobre práticas DevOps. Nesta abordagem, os critérios de qualidade são definidos, versionados e executados como parte integral da *pipeline* de desenvolvimento, garantindo que os padrões de qualidade sejam consistentemente aplicados e monitorizados ao longo de todo o ciclo de vida do projeto. O mecanismo de *Feedback* Rápido é fundamentado nos princípios *Lean* aplicados ao desenvolvimento de software [42], sendo essencial para a deteção precoce de problemas através de testes automatizados e ferramentas de monitorização contínua. Este sistema permite que as equipas identifiquem e resolvam *issues* rapidamente, reduzindo significativamente o tempo entre a introdução de um problema e a sua resolução. Finalmente, a implementação de *Deploys* Automatizados baseia-se nos princípios de *Deployment Pipeline* descritos por Martin Fowler [27], eliminando a variabilidade e os erros humanos associados aos processos manuais de implementação.

4.1.2 Ferramentas e Tecnologias Selecionadas

A seleção das ferramentas foi baseada em critérios de maturidade, integração com o ecossistema Python e facilidade de manutenção. A implementação baseou-se nas seguintes tecnologias:

- **GitHub Actions**: Plataforma de orquestração dos *workflows* CI/CD, escolhida pela sua integração nativa com repositórios GitHub e capacidade de automação robusta [28]

- **uv**: Ferramenta moderna para gestão rápida de dependências e ambientes virtuais Python, selecionada pela sua performance superior comparativamente ao `pip` tradicional [1]

- **pytest**: *Framework* de testes unitários e de integração, padrão de facto na comunidade Python para testes automatizados [43]

- **ruff**: *Linter* moderno implementado em Rust, escolhido pela sua velocidade e capacidades de formatação de código Python [2]

- **black**: Formatador automático de código Python, selecionado pela sua filosofia de formatação não-configurável que elimina debates sobre estilo de código [3]

[1] `https://docs.astral.sh/uv/`
[2] `https://docs.astral.sh/ruff/`
[3] `https://pypi.org/project/black/`

- **Next.js**: *Framework* React para criação de sites de documentação estática, escolhido pela sua capacidade de geração de sites otimizados [4]

4.2 Arquitetura da Pipeline

A *pipeline* CI/CD foi estruturada seguindo o padrão de *Deployment Pipeline* proposto por Martin Fowler [27], implementando múltiplas camadas de validação sequenciais, cada uma com responsabilidades específicas. Esta estrutura em camadas permite a identificação rápida de problemas e a falha rápida (*fail-fast*), princípio fundamental para manter ciclos de *feedback* eficientes.

A arquitetura implementada segue uma sequência rigorosa onde cada etapa representa um ponto de validação que deve ser aprovado antes de prosseguir para a etapa seguinte. Esta abordagem sequencial garante que apenas código que passa por todas as validações é, eventualmente aceite.

A *pipeline* implementa múltiplas etapas distintas de validação, cada uma com objetivos específicos:

1. **Push/PR**: Ativação automática da *pipeline* mediante *push* ou criação de *pull request*

2. **Lint & Format**: Verificação de conformidade com padrões de codificação e formatação automática

3. **Install & Dependencies**: Instalação e verificação de dependências do projeto

4. **Unit Tests**: Execução de testes unitários para validação de componentes individuais

5. **Quality Gates**: Verificação de métricas de qualidade e cobertura de código

6. **Build & Package**: Construção e empacotamento da distribuição final

7. **Deploy to PyPI**: Publicação automática no repositório Python Package Index

8. **Update Documentation**: Atualização e publicação automática da documentação

[4] `https://nextjs.org/`

4.3 Implementação dos Workflows

4.3.1 Arquitetura de um workflow

Para demonstrar a aplicação prática dos princípios teóricos de CI/CD descritos anteriormente, apresenta-se o *workflow* de testes definido no ficheiro `test.yml`, que implementa uma estratégia de ativação automática baseada em eventos do repositório:

```
name: Test
on:
  push:
    branches: [ "master" ]
  pull_request:
    branches: [ "master" ]
  workflow_dispatch:
```

Excerto de Código 4.1: Configuração de ativação do workflow de testes

Esta configuração 4.1 garante que os testes são executados em três cenários distintos: (1) sempre que há um *push* para o *branch* principal, implementando o princípio de validação contínua; (2) para cada *pull request*, garantindo que alterações propostas são validadas antes da integração; e (3) através de ativação manual (`workflow_dispatch`), permitindo execução sob chamada para cenários específicos de *debugging* ou validação. O processo de execução segue uma sequência rigorosa de passos, cada um com responsabilidades específicas na cadeia de validação:

Passo 1: Preparação do Ambiente

```
- uses: actions/checkout@v4
- name: Set up Python
  uses: actions/setup-python@v4
  with:
    python-version: '3.10'
```

Excerto de Código 4.2: Preparação do ambiente de execução

Esta etapa 4.2 inicial estabelece um ambiente Python limpo e consistente, utilizando a versão 3.10 especificada. A utilização de uma versão fixa elimina variabilidade entre execuções e garante reprodutibilidade dos resultados.

Passo 2: Gestão de Dependências

```
- name: Install uv
  run: pip install uv
- name: Create virtual environment
  run: uv venv
- name: Install dependencies
  run: |
    uv pip install -e ".[dev]"
    uv pip install pytest pytest-cov
```

Excerto de Código 4.3: Instalação e gestão de dependências

A gestão de dependências 4.3 utiliza a ferramenta `uv` para otimização de *performance*, criando um ambiente virtual isolado que previne problemas de conflitos de versões. A instalação inclui tanto as dependências do projeto (`-e ".[dev]"`) quanto as ferramentas específicas de teste (`pytest` e `pytest-cov`).

Passo 3: Execução dos Testes

```
- name: test
  run: uv run pytest -vv --cov=yake tests/
```

Excerto de Código 4.4: Comando de execução dos testes

A execução dos testes 4.4implementa uma abordagem *verbose* (`-vv`) que fornece *output* detalhado para facilitar o *debugging* em caso de falhas. A medição de cobertura de código (`-cov=yake`) garante que existe uma métrica quantitativa da eficácia dos testes implementados.

4.3.2 Integração com Makefile

O *workflow* está intrinsecamente ligado ao `Makefile` do projeto, que define comandos padronizados para execução local. Esta integração garante consistência entre desenvolvimento local e validação automatizada, como ilustrado no seguinte excerto de Código 4.5:

```
test:
    uv run pytest -vv --cov=yake test_*.py
```

Excerto de Código 4.5: Comando de teste no Makefile

Esta abordagem permite que os desenvolvedores executem localmente exatamente os mesmos comandos que são executados na *pipeline* automatizada, eliminando discrepâncias entre ambientes de desenvolvimento e CI.

4.4 Sistema de Testes Automatizados

A *suite* de testes foi estruturada seguindo as melhores práticas de teste de software, implementando testes unitários e de integração que validam tanto componentes individuais quanto funcionalidades completas do sistema. Os testes cobrem diferentes configurações linguísticas suportadas pelo YAKE!, garantindo que a *refatoração* não compromete a funcionalidade em nenhum idioma. Um aspeto crítico da *refatoração* foi garantir que os resultados algorítmicos permanecessem iguais após as modificações de código. Para tal, foi implementado um sistema de validação que compara os resultados obtidos com a nova implementação contra valores de referência estabelecidos.

O *workflow* de testes fornece *feedback* imediato através de múltiplos canais:

- **Status Visual**: Integração com a interface GitHub que exibe o estado dos testes diretamente nos *pull requests*
- **Logs Detalhados**: Saída *verbose* que permite identificação rápida da causa de falhas
- **Métricas de Cobertura**: Relatórios automáticos de cobertura de código que informam sobre a abrangência dos testes
- **Integração com Quality Gates**: Falha/Aviso automático da *pipeline* caso os critérios de qualidade não sejam cumpridos

4.5 Monitorização e Gates de Qualidade

Uma *pipeline* de qualidade implementa verificações automáticas de conformidade com padrões de codificação estabelecidos. Esta componente é fundamental para manter a consistência e legibilidade do código ao longo do desenvolvimento, implementando o conceito de *Quality Gates*.

4.5.1 Gates Aplicadas

O processo executa múltiplas ferramentas de análise estática: `ruff` para *linting* moderno e correção automática de problemas simples, e análise complementar de padrões de código Python. Esta abordagem multi-ferramenta garante uma cobertura abrangente de potenciais problemas de qualidade. A formatação automática de código é implementada através da ferramenta `black`, que aplica um estilo de formatação consistente e não-configurável. Esta abordagem elimina debates sobre preferências de formatação e garante

uniformidade visual em todo o projeto. O sistema implementa *gates* de qualidade rigorosos que impedem a integração de código que não cumpra os padrões estabelecidos. Estes *gates* incluem verificações de cobertura de testes (mínimo de 70%) e conformidade com as métricas de avaliação de código do PyLint (mínimo 8/10 no sistema de pontuação). Os *gates* são configurados para avisar/falhar automaticamente quando os critérios não são cumpridos, forçando a correção de problemas antes da integração. Esta abordagem preventiva é fundamental para manter a qualidade consistente do *codebase* ao longo do tempo.

4.5.2 Gestão de Artefactos e Limpeza do Ambiente

Para complementar o ecossistema de desenvolvimento automatizado, o `Makefile` inclui funcionalidades de manutenção e limpeza do ambiente que são essenciais para a higiene do repositório e prevenção de problemas relacionados com artefactos de compilação obsoletos. O comando `make clean` implementa uma estratégia abrangente de limpeza que remove sistematicamente todos os artefactos temporários gerados durante os processos de desenvolvimento, como demonstrado no excerto de Código 4.6:

```
clean:
rm -rf build/
rm -rf dist/
rm -rf .egg-info/
find . -type d -name pycache -delete
find . -type f -name ".pyc" -delete
```

Excerto de Código 4.6: Comando de limpeza no Makefile

Esta implementação aborda múltiplas categorias de artefactos:

- **Diretórios de *Build***: Remoção de `build/` e `dist/` que contêm distribuições compiladas
- **Metadados de Pacote**: Eliminação de diretórios `*.egg-info/` gerados durante a instalação
- ***Cache* Python**: Limpeza recursiva de diretórios `pycache` e ficheiros `.pyc` compilados

A disponibilidade deste comando é particularmente importante em cenários de *debugging* da *pipeline* CI/CD, onde artefactos residuais podem interferir com a execução de testes ou *builds* subsequentes. A integração desta funcionalidade no `Makefile` garante que tanto desenvolvedores locais quanto *workflows* automatizados podem facilmente restabelecer um estado limpo do

ambiente quando necessário. Esta abordagem alinha-se com os princípios de reprodutibilidade defendidos na literatura DevOps, permitindo que qualquer estado inconsistente seja rapidamente corrigido através da remoção de artefactos potencialmente corrompidos ou obsoletos.

4.6 Deploy e Versionamento Automatizado

4.6.1 Estratégia de Versionamento Semântico

O sistema implementa versionamento semântico automatizado baseado no padrão SemVer [5], utilizando *labels* de *pull requests* para determinar o tipo de incremento de versão necessário. Esta abordagem permite controlo preciso sobre as versões publicadas, distinguindo entre correções de *bugs* (incremento *patch*), novas funcionalidades (incremento *minor*) e alterações que quebram compatibilidade (incremento *major*). O processo de versionamento é completamente automatizado, eliminando erros humanos na atribuição de números de versão e garantindo consistência com as convenções estabelecidas pela comunidade de desenvolvimento Python.

4.6.2 Deploy Automatizado para PyPI

O *deploy* para o (Python Package Index (PyPi)) foi automatizado para garantir que cada *release* oficial é publicada de forma consistente e sem intervenção manual. O processo inclui construção automática de distribuições *wheel* e *source*, validação em ambiente de teste (TestPyPI), e publicação final no repositório principal. Esta automação elimina potenciais erros no processo de publicação e garante que todas as *releases* seguem exatamente os mesmos procedimentos, mantendo a qualidade e consistência das distribuições publicadas.

4.7 Resultados e Avaliação

4.7.1 Impacto na Qualidade do Software

A implementação da *pipeline* CI/CD resultou em melhorias mensuráveis na qualidade do software. A cobertura de testes aumentou significativamente, e o número de defeitos detetados em produção reduziu drasticamente devido à deteção precoce através dos testes automatizados. A consistência do código

[5] `https://semver.org/`

melhorou substancialmente através da aplicação automática de padrões de formatação e *linting*, eliminando variações estilísticas entre diferentes contribuidores e facilitando a manutenção a longo prazo.

4.7.2 Impacto na Produtividade

A automação resultou em ganhos significativos de produtividade para a equipa de desenvolvimento. O *feedback* rápido fornecido pelos *workflows* automatizados permite correção imediata de problemas, reduzindo o tempo necessário para resolução de *issues*. A eliminação de tarefas manuais repetitivas liberou tempo para atividades de maior valor, permitindo foco em melhorias algorítmicas e desenvolvimento de novas funcionalidades.

4.8 Conclusão

A implementação de uma *pipeline* CI/CD robusto, baseado em princípios estabelecidos na literatura de engenharia de software, representou um marco fundamental na evolução do projeto YAKE!. O sistema automatizado não apenas garantiu a manutenção da qualidade durante o processo de *refatoração*, mas também estabeleceu uma base sólida para desenvolvimento futuro sustentável. Os resultados demonstram que a aplicação de práticas DevOps modernas, fundamentadas teoricamente e suportadas por ferramentas adequadas, pode transformar significativamente a eficiência e confiabilidade do desenvolvimento de software. A *pipeline* implementada serve como modelo para projetos similares, demonstrando que mesmo bibliotecas académicas podem beneficiar enormemente de práticas de engenharia de software profissionais. A automação completa do ciclo de validação, teste, *deploy* e documentação permitiu que a equipa de desenvolvimento se concentrasse nas melhorias algorítmicas e arquiteturais, confiando que a infraestrutura automatizada garantiria a consistência e qualidade do produto final.

Capítulo

5

Página de Documentação

A criação de uma página de documentação moderna e acessível representa um aspeto fundamental para a adoção e utilização eficaz de qualquer biblioteca de software. No contexto do projeto **YAKE!**, a implementação de uma plataforma de documentação interativa tornou-se essencial para suportar tanto utilizadores iniciantes quanto desenvolvedores experientes, proporcionando acesso fácil e estruturado à informação técnica, exemplos práticos e recursos de aprendizagem.

Este capítulo detalha o processo de desenvolvimento e implementação da página de documentação do **YAKE!**, construída com tecnologias modernas e integrada no *pipeline* de desenvolvimento contínuo. A solução implementada não apenas substitui documentação estática tradicional, mas oferece uma experiência abrangente que vai além da documentação técnica, funcionando como o *site* principal do projeto com secções dedicadas a projetos relacionados, contribuidores, manual de utilização e recursos da comunidade.

5.1 Requisitos e *Design* da Interface

5.1.1 Análise de Requisitos

A definição dos requisitos da plataforma baseou-se na análise das necessidades dos diferentes tipos de utilizadores da biblioteca **YAKE!** e nas melhores práticas de documentação técnica moderna, considerando que o *site* deveria servir como portal principal do projeto.

Os **requisitos funcionais** identificados englobam a implementação de um sistema de navegação hierárquico que permita acesso rápido a diferentes secções da documentação e do projeto, interface adaptável a diferentes dispositi-

vos e tamanhos de ecrã, conformidade com padrões de acessibilidade *web* do Web Content Accessibility Guidelines (WCAG), integração de *notebooks* Jupyter através de *links* para Google Colab, e flexibilidade para adicionar conteúdo não-técnico como informações sobre contribuidores e projetos relacionados.

Relativamente aos **requisitos não-funcionais**, foram estabelecidos critérios de *performance* com tempo de carregamento otimizado, otimização para motores de busca (Search Engine Optimization (SEO)), estrutura de código modular e bem documentada para garantir manutenibilidade, e arquitetura que permita fácil adição de novo conteúdo para assegurar escalabilidade.

5.1.2 Arquitetura da Informação

A estrutura da informação foi organizada seguindo princípios de arquitetura da informação centrada no utilizador, expandindo além da documentação técnica tradicional. O *design* seguiu uma abordagem que permite navegação intuitiva desde informações gerais do projeto até detalhes específicos de implementação.

A organização do conteúdo foi estruturada contemplando uma secção de *Getting Started* para introdução rápida e exemplos básicos, *API Documentation* para referência das classes e métodos através de documentação manual via Mark Down X (MDX), *Examples* contendo *notebooks* Jupyter com *links* diretos para Google Colab, *Contributing Guide* com instruções para contribuidores, *Related Projects* apresentando projetos derivados ou relacionados, e *Team* com informações sobre contribuidores e mantenedores.

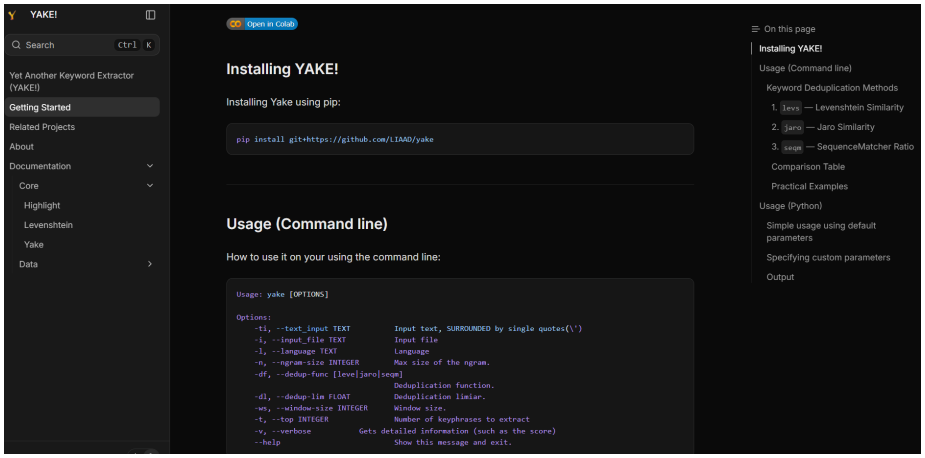

Figura 5.1: Página inicial "Getting Started"(escuro) mostrando a estrutura hierárquica de navegação e integração com Google Colab

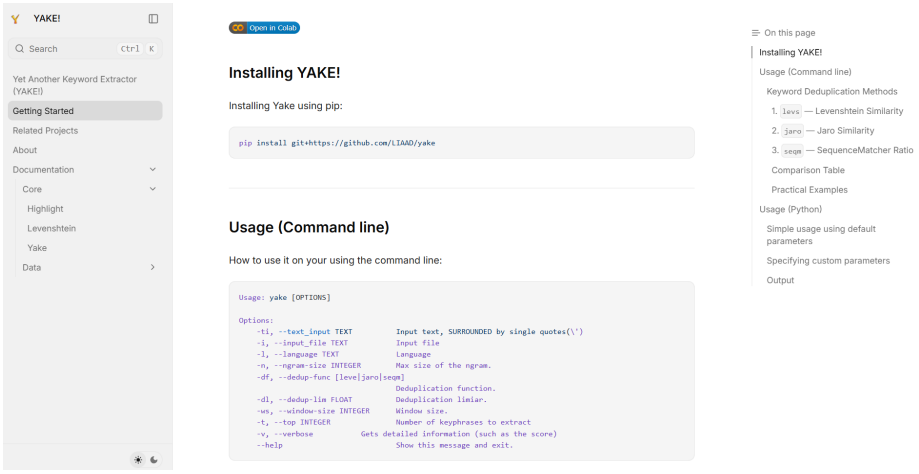

Figura 5.2: Página inicial "Getting Started"(claro) mostrando a estrutura hierárquica de navegação e integração com Google Colab

A Figura 5.2 ilustra a implementação da página inicial de introdução, onde é possível observar o sistema de navegação lateral estruturado hierarquicamente, o índice de conteúdos da página atual, e a integração *seamless* com Google Colab através do botão "Open in Colab".

5.1.3 *Design System* e Interface

O *design* da interface seguiu princípios modernos de User Experience (UX)/User Interface (UI), implementando um sistema de *design* consistente que reflete a identidade visual do projeto **YAKE!**. A **tipografia** baseou-se na fonte *Inter*, escolhida pela sua excelente legibilidade em interfaces digitais. A paleta de cores foi desenvolvida com foco na acessibilidade, garantindo *ratios* de contraste adequados e suporte para temas "claro"e "escuro". O **sistema de componentes** foi construído utilizando princípios de *design* atómico, englobando elementos base como botões, *inputs*, ícones e *badges* incluindo *badges* de qualidade do projeto como certificações e métricas, componentes compostos como *cards* de navegação e blocos de código, e secções completas como *sidebar*, *header*, *footer* e navegação principal.

5.2 *Fumadocs*

5.2.1 Contexto e Alternativas Avaliadas

A escolha da tecnologia para implementar a plataforma envolveu uma análise de várias soluções disponíveis no ecossistema de documentação técnica.

As **alternativas consideradas** incluíram *Sphinx* como o *standard* para documentação Python, *MkDocs* enquanto *framework* Python simples para documentação, *Docusaurus* como *framework* desenvolvida pelo Facebook, *GitBook* enquanto plataforma comercial, e outras soluções especializadas em documentação técnica.

5.2.2 Critérios de Avaliação

A avaliação das alternativas baseou-se em critérios específicos abrangendo ***performance*** relacionada com velocidade de carregamento e otimização, **flexibilidade** na capacidade de criar um *site* completo do projeto e não apenas documentação, ***developer experience*** considerando a facilidade de desenvolvimento e manutenção, **suporte MDX** para a capacidade de integrar conteúdo híbrido *markdown*/React, **customização** avaliando a flexibilidade para personalização visual e funcional, e **ecossistema** analisando a comunidade e ferramentas disponíveis.

5.2.3 Justificação da Escolha do *Fumadocs*

A escolha do *Fumadocs* foi fundamentada em várias vantagens técnicas e estratégicas que se alinhavam perfeitamente com os objetivos do projeto.

As **vantagens técnicas** contemplam *performance* superior baseada em Next.js com *Static Site Generation* (Static Site Generation (SSG)), proporcionando pré-renderização de todas as páginas em *build time*, *code splitting* automático, e otimização de *assets*. A utilização de tecnologias modernas abrange React 18, TypeScript nativo, Tailwind CSS para *styling*, e **suporte nativo para MDX**, permitindo integração *seamless* de conteúdo *markdown* com componentes React personalizados.

As **vantagens estratégicas** mais significativas foram a **flexibilidade para criar um *site* completo do projeto** em vez de apenas documentação técnica, permitindo secções para projetos relacionados, informações sobre a equipa, e outros conteúdos não-técnicos. O **excelente suporte MDX** torna o processo de criação e atualização de conteúdo extremamente simples, permitindo que contribuidores adicionem documentação rica sem conhecimento técnico avançado.

5.3 Funcionalidades Implementadas

5.3.1 Sistema de Navegação Hierárquico

O sistema de navegação foi implementado utilizando a estrutura de ficheiros como fonte de verdade, gerando automaticamente *sidebar navigation* com hierarquia de páginas, *breadcrumbs* baseados no *path* do ficheiro, navegação *Previous/Next* sequencial, e *table of contents* extraído dos *headings* MDX.

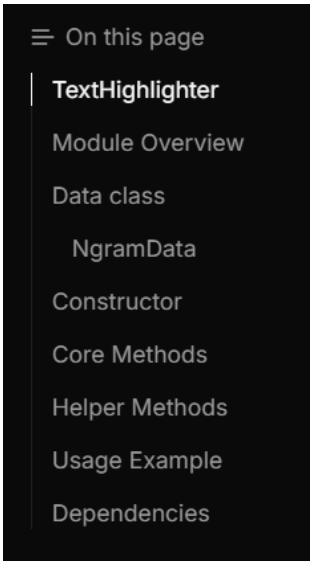

Figura 5.3: Sistema de navegação "On this page"mostrando a estrutura hierárquica dos conteúdos da página atual

A Figura 5.3 demonstra o sistema de índice automático "On this page"que extrai a estrutura hierárquica dos *headings* da página atual, proporcionando navegação rápida e contextual dentro do conteúdo.

5.3.2 Documentação Manual via MDX

Nota importante: Devido à conversão para *site* estático, funcionalidades que requerem Application Programming Interface (API)s dinâmicas como pesquisa *server-side* e extração automática de *docstrings* não são viáveis. Esta limitação foi identificada como área de melhoria futura.

A documentação das classes e métodos foi implementada através de **documentação manual utilizando MDX**, aproveitando a flexibilidade desta tecnologia para criar conteúdo rico e interativo. Embora manual, este processo é **extremamente simples graças ao suporte MDX**, proporcionando documentação detalhada que vai além das *docstrings* básicas, integração de exemplos

de código com *syntax highlighting*, componentes interativos para demonstrar funcionalidades, e informações contextuais e casos de uso avançados.

Figura 5.4: Exemplo de documentação via MDX, mostrando detalhes técnicos de implementação de uma função

A Figura 5.4 ilustra um exemplo da documentação criada através de MDX, onde é possível observar a riqueza de detalhes técnicos, incluindo parâmetros, valores de retorno, e lógica de implementação que vai muito além das *docstrings* tradicionais.

5.3.3 Exemplos e Demonstrações

O *site* inclui **exemplos detalhados das classes principais** com explicações passo-a-passo, casos de uso práticos, e informações contextuais que não estão disponíveis nas *docstrings* básicas. **Nota**: O *site* não permite execução direta de

código devido às limitações do ambiente estático, mas todos os exemplos estão disponíveis via Google Colab para execução interativa.

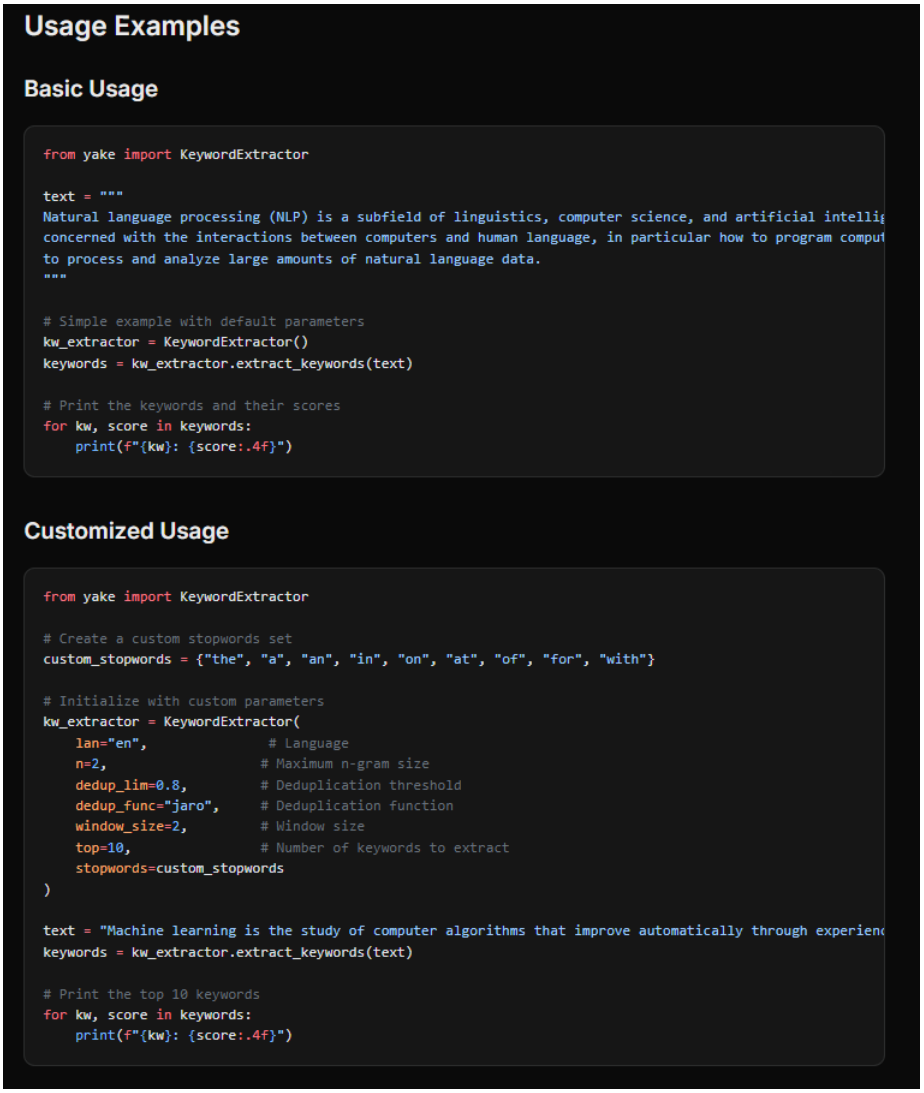

Figura 5.5: Exemplos de utilização básica e customizada da biblioteca YAKE, com código Python e explicações detalhadas

A Figura 5.5 apresenta exemplos práticos de utilização da biblioteca, demonstrando tanto a utilização básica com parâmetros *default* quanto configurações customizadas avançadas, incluindo *syntax highlighting* e comentários explicativos detalhados.

5.4 Melhorias na Apresentação do Projeto

5.4.1 README Completamente Renovado

Foi criado um **README completamente novo, muito mais limpo e *user-friendly***, contemplando introdução clara e concisa ao **YAKE!**, instalação simplificada com comandos *copy-paste*, exemplos básicos para *quick start*, *links* para documentação completa, *badges* de qualidade e certificações, secção de contribuição bem estruturada, e informações sobre licenciamento e citação académica.

A Figura 5.6 mostra a transformação significativa do README do projeto, onde é possível observar a estrutura mais limpa e profissional, incluindo *badges* de qualidade, descrição concisa das funcionalidades, e exemplos práticos de instalação e utilização básica.

5.4.2 Repositório *Demo* Separado

Foi criado um **repositório separado dedicado à *demo*** [1] **do YAKE!**, incluindo **tutorial detalhado** de como executar a *demo*, **guia de otimização** para tirar melhor proveito das funcionalidades, exemplos de casos de uso específicos, configurações recomendadas para diferentes cenários, e *troubleshooting* e Frequently Asked Questions (FAQ).

Esta separação permite manter o repositório principal focado no código da biblioteca, enquanto a *demo* tem o seu próprio espaço para evolução e experimentação.

5.5 Limitações Identificadas e Melhorias Futuras

5.5.1 Limitações Atuais

Devido à natureza estática do *site*, necessária para compatibilidade com GitHub Pages, algumas funcionalidades avançadas não estão disponíveis. A **pesquisa *server-side*** através de APIs dinâmicas não funciona em ambiente estático. A **documentação API automática** requer extração automática de *docstrings* que necessita processamento *server-side*. A **execução de código no *site*** não é possível executar código Python diretamente no *browser*, sendo a alternativa disponibilizada via Google Colab.

[1] `https://github.com/LIAAD/yake_demo`

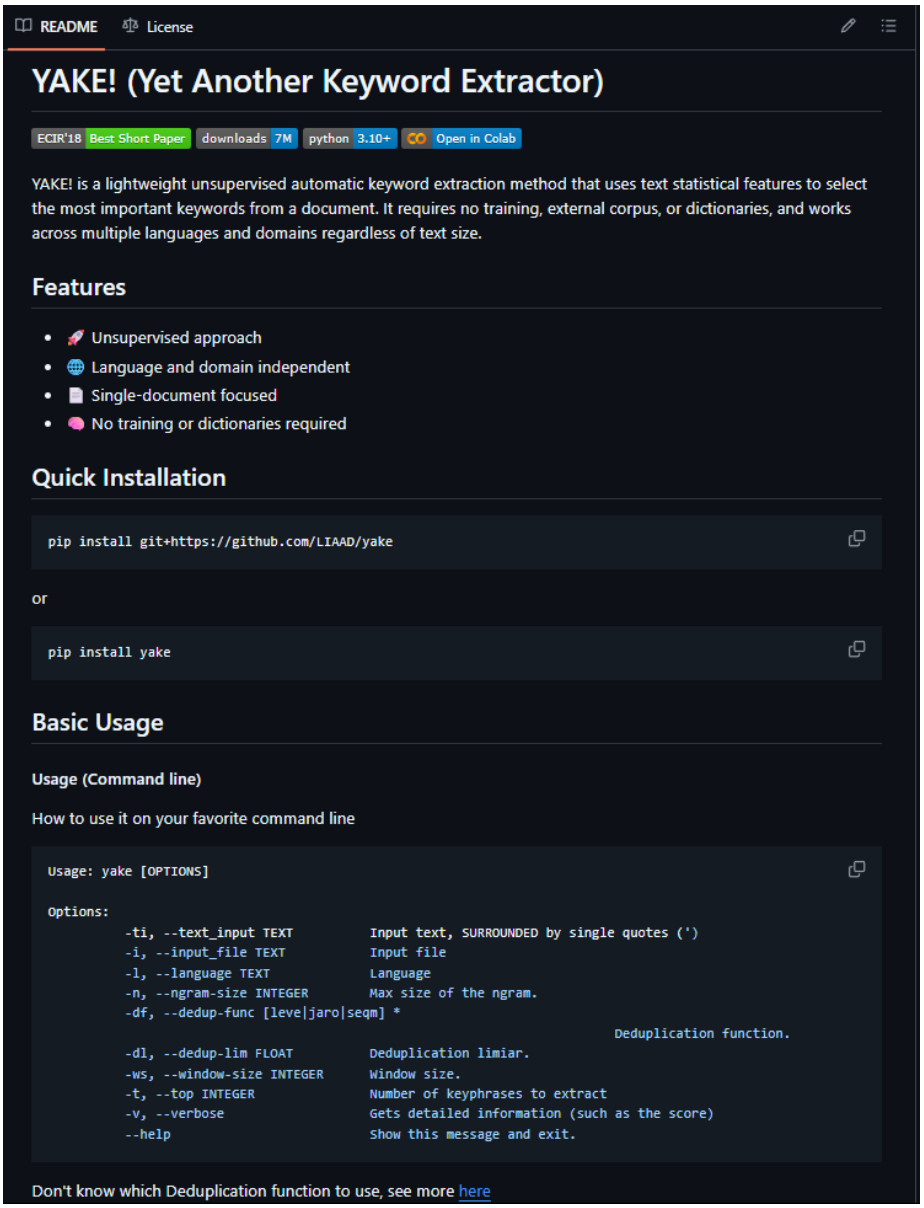

Figura 5.6: Visão geral do README renovado, mostrando estrutura limpa, badges de qualidade e exemplos de utilização

5.5.2 Melhorias Futuras Identificadas

As seguintes funcionalidades foram identificadas para implementação futura, organizadas por categoria de desenvolvimento.

As **funcionalidades técnicas avançadas** contemplam a implementação de pesquisa *client-side* usando índices pré-gerados, sistema de documentação

API automática através de *build-time processing* e editor *online* para testar **YAKE!**.

As **funcionalidades de comunidade** abrangem sistema de comentários e *feedback* integrado, tradução automática para múltiplas linguagens, documentação para múltiplas versões da biblioteca, e métricas detalhadas de utilização. .

5.6 Resultados e Impacto

A nova documentação e apresentação do projeto resultaram em melhorias mensuráveis na perceção e utilização da biblioteca. O **maior profissionalismo** transmitido pelo *site* completo do projeto aumenta significativamente a credibilidade junto da comunidade académica e de desenvolvimento. A **facilidade de contribuição** através do processo simplificado via MDX reduziu substancialmente as barreiras para novos contribuidores. O **melhor *onboarding*** proporcionado pelo README renovado e documentação estruturada facilita consideravelmente a adoção por novos utilizadores. O **reconhecimento académico** destacado através dos *badges* e certificações enfatiza a qualidade científica do trabalho desenvolvido.

5.7 Conclusão

A implementação da nova plataforma de documentação para o **YAKE!** representa uma evolução significativa na apresentação e acessibilidade do projeto. A escolha do *Fumadocs* provou ser acertada, principalmente devido ao **excelente suporte MDX** que torna a criação e manutenção de conteúdo extremamente simples, e à **flexibilidade para criar um *site* completo do projeto** que transcende a documentação técnica tradicional.

As **principais contribuições** englobam uma arquitetura moderna e escalável baseada em tecnologias *web* contemporâneas, documentação rica e acessível criada manualmente via MDX, integração *seamless* com Google Colab para exemplos interativos, README completamente renovado com foco na experiência do utilizador, repositório *demo* separado com tutoriais detalhados, e integração de *badges* e certificações que aumentam substancialmente a credibilidade do projeto.

Embora existam limitações inerentes à natureza estática da solução, que foram identificadas para melhorias futuras, a plataforma atual fornece uma base sólida para a evolução contínua da documentação e apresentação do projeto **YAKE!**. Esta implementação estabelece um novo padrão de qualidade e profissionalismo na sua apresentação à comunidade científica e de desenvolvimento, contribuindo significativamente para a sua adoção e reconhecimento no ecossistema de ferramentas de processamento de linguagem natural.

Capítulo

6

Conclusão

Este projeto reforçou a importância do estudo e aplicação sistemática de princípios de Engenharia de *Software*, para melhorar a qualidade, compreensão e acessibilidade de qualquer programa. .

O trabalho seguiu uma abordagem metodológica estruturada que integrou múltiplas dimensões da engenharia de *software*, constituindo uma mais-valia para a comunidade científica. Contribuiu para demonstrar o compromisso com o projeto através de uma melhoria fundamental na arquitetura e estrutura do sistema, facilitando simultaneamente o seu desenvolvimento e futuras contribuições.

A conclusão deste projeto de refatoração e modernização do **YAKE!** estabelece uma base sólida para desenvolvimentos futuros que podem expandir significativamente o impacto e utilidade da ferramenta. Entre as prioridades para trabalho futuro destaca-se o desenvolvimento de uma nova RestFul API, visto que foi o ponto do projeto onde acabou por não ser trabalhado. Outras prioridades incluem a exploração da escalabilidade e implementações que visem melhorias de *performance*.

Apêndice

A

Proposta de Projeto

O seguinte apêndice apresenta o documento do projeto proposto.

UNIVERSIDADE BEIRA INTERIOR

FACULDADE ENGENHARIA
Departamento de Informática

Automatic Keyword Extraction from Texts – YAKE! Refactoring

Proposta de Projeto. Lic. Eng. Informática

Orientador: Ricardo Campos (ricardo.campos@ubi.pt)

CoOrientador: Arian Pasquali (a.pasquali@faktion.com)

Objectives

Can you imagine how difficult would it be to analyze thousands of reports, tweets, without a keyword extraction system?

Keyword extraction (aka keyphrase extraction or detection [1]) is a natural language processing (NLP) technique that aims to automatically extract the most relevant words and phrases from a text (see Fig. 1), thus providing a concise summary of the text's content and of the main topics therein discussed. This is useful for several downstream tasks such as text analysis, social media monitoring, text summarization, text generation, text tagging, information retrieval, etc.

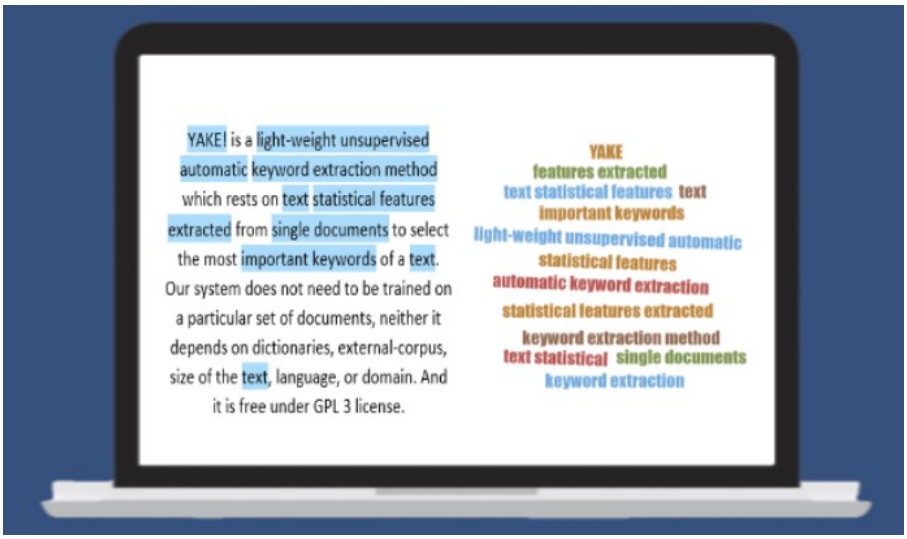

Fig. 1: Annotated text by YAKE demo: http://yake.inesctec.pt

In this work, the student is challenged to refactor the code of an already existing keyword extraction system and to implement a few novel features. In particular, the work will rely on YAKE! a popular unsupervised algorithm [2] that has been the subject of several uses by the

research community in a number of downstream tasks (e.g., text summarization, chatbots, Q&A systems, etc). A great example of its use was the creation of the general index, which used YAKE! to extract more than 19 billion keywords from over 100M documents (see INESC TEC and Nature web articles). Also John Snow Labs, through its Spark NLP package (claimed to be the most widely used NLP library in the business sector) uses YAKE! as their portfolio solution, among many other companies.

The objective of this project is to refactor the code of YAKE!. In addition to this, the student should implement and make available an API of the algorithm.

Workplan

T1: Project setup and planning (understand existing YAKE! codebase) (2 weeks)

- Study paper and codebase.
- Split Yake codebase. It should have just Yake core. Demo, docker and api should have a dedicated repo.
- Focus: core YAKE.

T2: CD/CI (Study and Design the pipeline) (1 weeks)

- Study recent DevOps tools and practices.
- Study what tool better works for the project (Git Actions, Cheff, Puppet)
- Desing the workflow of the pipeline.

T3: Code Refactoring (refactor and clean up the existing code) (4 weeks)

- Refactor code following Python best practices
    - Use cookiecutter template (GitHub - audreyfeldroy/cookiecutter-pypackage: Cookiecutter template for a Python package.)
    - Refactor code following modern Python standards.
    - Python spec https://peps.python.org/pep-0008/
    - Use lint and code formatting tool (Ruff (astral.sh))
    - Setup github actions for building and testing (Building and testing Python - GitHub Docs)

T4: Testing (3 weeks)

- Write and automate new result tests.
- Refactor existing tests.

- Support reference dataset for automated test.
- Show test reference dataset and check if scores keep the same.

T5: Documentation (1 week)

- Clean documentation. Documentation should focus on core YAKE only.
- Review README.
- Write Jupyter notebook tutorial.

T6: API development (2 weeks)

T7: Report writing and presentation (2 weeks)

Table 1 presents the distribution of tasks for each of the 15 weeks.

Table 2: Chronology of tasks (T) per week (S).

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15
T1															
T2															
T3															
T4															
T5															
T6															
T7															

Technical and Academic Prerequisites

- **Proficiency in Python**: Strong programming skills in Python, as YAKE! is primarily implemented in Python.
- **Natural Language Processing (NLP) Knowledge**: A solid understanding of NLP concepts, including text preprocessing, feature extraction, and keyword extraction algorithms.
- **GitHub and Open Source Collaboration**: Familiarity with GitHub for version control and experience with open-source collaboration.
- **APIs design**: Knowledge of RESTful API design and implementation.

Expected Results

- Code refactored following modern Python best practices.
- Automated CI/CD.
- New and automated Linting and Result tests
- New RESTful API

Bibiography

[1] Papagiannopoulou, E., and Tsoumakas, G. (2019). A Review of Keyphrase Extraction. WIREs Data Mining and Knowledge Discovery, vol 10(2).

[2] Campos, R., Mangaravite, V., Pasquali, A., Jorge, A., Nunes, C. and Jatowt, A. (2020). YAKE! Keyword Extraction from Single Documents using Multiple Local Features. In: Information Sciences Journal. Elsevier, Vol 509, pp 257-289, ISSN 0020-0255

Bibliografia

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2nd edition, 2019.

[2] Nadia Eghbal. Roads and bridges: The unseen labor behind our digital infrastructure. Technical report, Ford Foundation, 2016.

[3] Ricardo Campos, Vítor Mangaravite, Arian Pasquali, Alípio Jorge, Célia Nunes, and Adam Jatowt. YAKE! Collection-independent automatic keyword extractor. In *European Conference on Information Retrieval*, pages 806–810, Grenoble, France, 2018. Springer.

[4] Ricardo Campos, Vinícius Mangaravite, Arian Pasquali, Alípio Jorge, Célia Nunes, and Adam Jatowt. Yake! keyword extraction from single documents using multiple local features. *Information Sciences*, 509:257–289, 2020.

[5] Ricardo Campos, Vítor Mangaravite, Arian Pasquali, Alípio Jorge, Célia Nunes, and Adam Jatowt. YAKE! Keyword extraction from single documents using multiple local features. *Information Sciences*, 509:257–289, 2020.

[6] Pin Ni, Yuming Li, and Victor Chang. Research on text classification based on automatically extracted keywords. *International Journal of Enterprise Information Systems*, 16:1–16, 10 2020.

[7] Wafaa El-Kassas, Cherif Salama, Ahmed Rafea, and Hoda Mohamed. Automatic text summarization: A comprehensive survey. *Expert Systems with Applications*, 165:113679, 07 2020.

[8] Rúben Almeida, Ricardo Campos, Alípio Jorge, and Sérgio Nunes. Indexing Portuguese NLP Resources with PT-Pump-Up. In *Proceedings of the 16th International Conference on Computational Processing of Portuguese (PROPOR'24)*, Santiago de Compostela, Spain, March 2024.

[9] Swagata Duari and Vasudha Bhatnagar. sCAKE: Semantic connectivity aware keyword extraction. *arXiv preprint arXiv:1811.10831*, 2018. Also published in *Information Sciences* (477:100–117, 2019).

[10] Eleni Papagiannopoulou and Grigorios Tsoumakas. A review of keyphrase extraction. *WIREs Data Mining and Knowledge Discovery*, 10(e1339), 2020.

[11] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988.

[12] Stuart Rose, Dave Engel, Nick Cramer, and Wendy Cowley. Automatic keyword extraction from individual documents. In *Text Mining: Applications and Theory*, pages 1–20. John Wiley & Sons, 2010.

[13] Michael Kearns. "important" vertices and the pagerank algorithm. Lecture note, nets 112: Networked life, University of Pennsylvania, Department of Computer and Information Science, Philadelphia, PA, USA, 2013. Available at `https://www.cs.bilkent.edu.tr/~guvenir/courses/CS550/Workshop/Yasin_Uzun.pdf` – (corrigido o link para o arquivo PageRank da Univ. of Pennsylvania).

[14] Rada Mihalcea and Paul Tarau. TextRank: Bringing order into text. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pages 404–411, Barcelona, Spain, 2004.

[15] Ian H. Witten, Gordon W. Paynter, Eibe Frank, Carl Gutwin, and Craig G. Nevill-Manning. KEA: Practical automatic keyphrase extraction. In *Proceedings of the Fourth ACM Conference on Digital Libraries*, pages 254–255, Berkeley, CA, USA, 1999. ACM.

[16] Maarten Grootendorst. Keybert: Minimal keyword extraction with bert, 2020. GitHub repository.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 4171–4186, 2019.

[18] Florian Boudin, Ygor Gallina, and Akiko Aizawa. Keybarthez: Leveraging pre-trained language models for keyphrase generation. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 5384–5390, 2020.

[19] Kamil Bennani-Smires, Claudiu Musat, Andreea Hossmann, Michael Baeriswyl, and Martin Jaggi. Simple unsupervised keyphrase extraction

using sentence embeddings. In *Proceedings of the 22nd Conference on Computational Natural Language Learning*, pages 221–229, 2018.

[20] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pages 3982–3992, 2019.

[21] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008, 2017.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[24] Matthew Honnibal and Ines Montani. spacy 2: Natural language understanding with bloom embeddings, convolutional neural networks and incremental parsing. In *Proceedings of the 25th International Conference on Computational Linguistics: System Demonstrations*, pages 411–420, 2017.

[25] Edward Loper and Steven Bird. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics*, pages 63–70. Association for Computational Linguistics, 2002.

[26] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. Best practices for scientific computing. *PLOS Biology*, 12(1):e1001745, 2014.

[27] Martin Fowler. Continuous Integration. `https://martinfowler.com/articles/continuousIntegration.html`, 2013. Accessed: 2024-12-30.

[28] GitHub, Inc. GitHub Actions Documentation. `https://docs.github.com/en/actions`, 2019. Accessed: 2024-12-30.

[29] John Ferguson Smart. *Jenkins: The Definitive Guide*. O'Reilly Media, 2011.

[30] GitLab Inc. Gitlab ci/cd, 2016.

[31] Logilab. Pylint - Python Code Analysis Tool. Software Tool, 2003.

[32] Python Code Quality Authority. Flake8: Your tool for style guide enforcement, 2023.

[33] Python Software Foundation. Black: The uncompromising code formatter, 2023.

[34] Timothy Crosley. isort: A python utility to sort imports, 2023.

[35] Guido van Rossum, Barry Warsaw, and Nick Coghlan. PEP 8 – Style Guide for Python Code. Python Enhancement Proposal, 2001.

[36] Barbara H. Liskov. Behavioral subtyping using invariants and constraints. 1999.

[37] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[38] David Goodger and Guido van Rossum. PEP 257 – Docstring Conventions. Python Enhancement Proposal, 2001.

[39] ISO/IEC 25010:2011 - Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models, 2011.

[40] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[41] Gene Kim, Jez Humble, Patrick Debois, and John Willis. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, 2016.

[42] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional, 2003.

[43] Holger Krekel et al. pytest: helps you write better programs. `https://pytest.org`, 2004. Accessed: 2024-12-30.