POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Prova de conceito

Deteção Automatizada de Dados Pessoais com Técnicas NER

Mestrado de Cíbersegurança e Informática Forense – Trabalho de Projeto

André Cirilo Lages Rêgo

Leiria, setembro 2024

Prova de conceito

Deteção Automatizada de Dados Pessoais com Técnicas NER

Mestrado de Cíbersegurança e Informática Forense – Trabalho de Projeto

André Cirilo Lages Rêgo

Orientador: Carlos Manuel Gonçalves Antunes

Leiria, setembro 2024

POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Agradecimentos

Expresso a minha sincera gratidão ao meu orientador pela sua orientação e por todo o apoio prestado ao longo deste projeto. Agradeço igualmente à minha família pelo suporte contínuo e pelo incentivo, que foram essenciais durante todo o percurso do mestrado.

Resumo

O reconhecimento de entidades nomeadas (NER) é uma técnica essencial no processamento de linguagem natural (PLN), usada para identificar automaticamente informações importantes, como nomes e moradas, em grandes volumes de texto. A sua relevância cresce à medida que as empresas e organizações lidam com dados textuais não estruturados, tornando a extração automática de informações uma necessidade cada vez mais presente.

Este trabalho propõe o desenvolvimento de uma solução NER que permite ao utilizador submeter documentos em diferentes formatos, processando-os para extrair entidades nomeadas, com foco em português europeu. A solução deverá oferecer ao utilizador a possibilidade de escolher a biblioteca de NER e converte os documentos em texto simples, aplicando técnicas de extração e classificando as entidades encontradas.

A implementação foi realizada utilizando uma stack tecnológica modular, com Python, FastAPI, spaCy, NLTK e PostgreSQL, permitindo um fluxo eficiente entre o upload, a conversão de documentos e a extração de entidades. O sistema é flexível e escalável, preparado para futuras melhorias.

Os testes realizados mostram que a aplicação é capaz de processar documentos de diferentes formatos e extrair entidades com precisão, tornando-se uma ferramenta eficaz para automatizar a extração de dados sensíveis em diversos contextos.

Palavras-chave: Reconhecimento de Entidades Nomeadas, Processamento de Linguagem Natural, Extração de Dados Sensíveis, Português Europeu, Documentos Não Estruturados, Automatização, spaCy, NLTK, Conversão de Documentos, Python, FastAPI, PostgreSQL, Flexibilidade, Escalabilidade, Precisão, Testes de Desempenho, Modularidade.

Abstract

Named Entity Recognition (NER) is a crucial technique in Natural Language Processing (NLP) used to automatically identify important information such as names and addresses from large volumes of text. Its relevance is growing as businesses and organizations increasingly deal with unstructured textual data, making automated information extraction a critical need.

This work proposes the development of a NER solution that allows users to submit documents in various formats, processing them to extract named entities, with a focus on European Portuguese. The solution offers users the ability to choose the NER library and converts documents into plain text, applying extraction techniques and classifying the identified entities.

The implementation was carried out using a modular technology stack, including Python, FastAPI, spaCy, NLTK, and PostgreSQL, ensuring an efficient workflow between file upload, document conversion, and entity extraction. The system is designed to be flexible and scalable, ready for future improvements.

The tests conducted demonstrate that the application can process documents in different formats and accurately extract entities, making it an effective tool for automating the extraction of sensitive data in various contexts.

Keywords: Named Entity Recognition, Natural Language Processing, Sensitive Data Extraction, European Portuguese, Unstructured Documents, Automation, spaCy, NLTK, Document Conversion, Python, FastAPI, PostgreSQL, Flexibility, Scalability, Accuracy, Performance Testing, Modularity.

Lista de Figuras

Figura 1- Comandos de instalação 25
Figura 2- Exemplo prático NER Spacy – Modelo PT – Frase ING 26
Figura 3 - Réplica de exemplo prático NER com modelo PT 27
Figura 4 - Exemplo prático NER – Modelo PT 28
Figura 5 - Exemplo prático NER – Modelo PT – Resultados 28
Figura 6 - Reconhecimento de entidades pré-definidas 29
Figura 7 - Reconhecimento de entidades pré-definidas – Resultados 29
Figura 8 - Reconhecimento de entidades pré-definidas 30
Figura 9 - Treino de pipeline 33
Figura 10 - Implementação NER pipeline 33
Figura 11 - Implementação de ciclo de treino 33
Figura 12 - Impressão de iteração de perdas 34
Figura 13 - Resultados de iteração 34
Figura 14 - Input para utilização do modelo treinado 35
Figura 15 - Utilização de modelo treinado 36
Figura 16 - Comandos de instalação NLTK 38
Figura 17 - Descarregamento de pacotes 38
Figura 18 - Exemplo prático NER 38
Figura 19 - Processo de treino manual NLTK 40
Figura 20 - Integração de regex na implementação NLTK 41
Figura 21 - Implementação NER NLTK 43
Figura 22 - Implementação NER NLTK - Resultados 44
Figura 23 - Exportação de modelo treinado 44
Figura 24 - Importação do modelo treinado 44
Figura 25 - Input para implementação de *custom model* 44
Figura 26 - Implementação NLTK 45
Figura 27 - Comandos de instalação HFT 46
Figura 28 - Implementação NER com modelo importado 47
Figura 29 - Implementação NER com modelo importado – resultados 47
Figura 30 - Implementação NER com modelo importado - resultados 48
Figura 31 - Implementação NER com entidades pré-definidas 49
Figura 32 - Treino de modelo customizado 51
Figura 33 - Importação e utilização de modelo 51
Figura 34 - Comandos de conversão 54
Figura 35- Output doccano 61
Figura 36 - Output Label Studio 61
Figura 37 - Gerar e armazenar modelo SpaCy 64
Figura 38 - Carregamento do modelo customizado 64
Figura 39 - Output de carregamento do modelo customizado 64
Figura 40 - Criação e armazenamento de modelo NTLK 66
Figura 41 - Classificador de características 67
Figura 42 - Treino de modelo NLTK 68
Figura 43 - Resultados do modelo apresentado 68

Figura 44 - Resultados do modelo apresentado ... 70
Figura 45 - Excerto de resultados do modelo apresentado ... 70
Figura 46 - Fluxo aplicacional ... 77
Figura 47 – Stack Aplicacional ... 81
Figura 48 - Fluxo de utilização ... 83
Figura 49 - Fluxo de processo ... 84
Figura 50 - Fluxo para reconhecimento de entidades ... 87
Figura 51 - Função de Upload de ficheiros ... 97
Figura 52 - Função de conversão ... 98
Figura 53 - Processamento de identificação entidades ... 99
Figura 54 - Função de conversão: calibre e pandoc ... 100
Figura 55 - Função de conversão: unoconv e soffice ... 101
Figura 56 - Bibliotecas de reconhecimento de entidades ... 102
Figura 57 - Carregamento do modelo spaCy ... 103
Figura 58 – Carregamento de modelo NLTK ... 104
Figura 59 – Implementação de NLTK ... 105
Figura 60 - Formulário HTML ... 107
Figura 61 - Tabela de resultados ... 108
Figura 62 - Implementação de exportação ... 109
Figura 63 - Modelo Entidade-Relacionamento ... 109
Figura 64 - Implementação de faker ... 110
Figura 65 - Conversão jsonl para json ... 111
Figura 66 - Conversão para formato spaCy ... 111
Figura 67 - Conversão para formato spaCy - implementação ... 112
Figura 68 - Conversão para formato spaCy 3 ... 113
Figura 69 - Teste de precisão de modelo ... 113
Figura 70 - Script de tokenização ... 115
Figura 71 - Output do script ... 116
Figura 72 – Implementação de treino de modelo ... 117
Figura 73 - Gerar dados falsos ... 120
Figura 74 - Iterações de fine tuning ... 120
Figura 75- Fine Tuning Nltk ... 122
Figura 76 - Fine Tuning - Tratamento ... 122
Figura 77 – Fine Tuning Nltk - 3 ... 123
Figura 78 – Integração do modelo fine-tuned ... 123
Figura 79 - Página inicial protótipo PIxtract ... 124
Figura 80 - Página de resultados protótipo ... 125
Figura 81 - Script para criar ficheiros ... 131
Figura 82 - Criação de ficheiro de entidades falsas ... 132
Figura 83 - Gerar atividade de vários utilizadores ... 132
Figura 84 - Simulação de uploads longos ... 134

Lista de Tabelas

Tabela 1 - Resultados de ferramentas de conversão 55
Tabela 2 - Comparação Doccano/Label Studio 62
Tabela 3 - Comparação de configuração bibliotecas NER 71
Tabela 4 - Comparação de capacidades de treino NER 72
Tabela 5 - Comparação NER performance e precisão 73
Tabela 6 - Comparação de suporte NER 73
Tabela 7 - Resultados de comparação 74
Tabela 8 - Componentes de motor NER 86
Tabela 9 - Endpoint upload file 89
Tabela *10* - Parametros endpoit upload file 89
Tabela *11* - Lista de funcionalidades do projeto 96
Tabela *12* - Resultados de Modelo Spacy 118
Tabela *13* - Resultados de Modelo NLTK 119
Tabela 14 - Resultados fine tuning – SpaCy 121
Tabela 15 - Lista de cenários para testes de funcionalidade 128
Tabela 16 - Lista de cenários para testes de carga 130
Tabela 17 - Teste carga - cenário 1 131
Tabela *18* - Teste carga - cenário 2 133
Tabela 19 – Testes de carga 135

1. Introdução

Data driven society é uma caracterização para a sociedade de hoje, onde o individuo considera privacidade um direito e, no entanto, não tomam qualquer ação para proteger a sua. Além disso, maior parte dos serviços atuais são aplicados quando o utilizador permite acesso aos seus dados. Na escolha entre usufruir do serviço ou proteger os seus bens digitais o hábito da segunda escolha tem sido diluído no decorrer da década. Os dados pessoais tornaram-se um dos principais bens dos serviços públicos, melhorando a sua eficiência (OECD Publishing, 2021). Kaiser Permanente, uma seguradora norte americana utilizou dados de mais de 1,5 milhões de pacientes para treinar o seu modelo de previsão com algoritmos de *machine learning*. O sistema de alertas analisa a cada hora os registos eletrónicos dos dispositivos hospitalares para acompanhar o resultado das métricas do paciente, em que no caso dos valores não se encontrarem dentro do normal e houver uma alteração negativa nas próximas 12 horas é emitido um alerta para intervenção no local. (Martinez, 2022)

As potencialidades de análise deste tipo de registos, não foram descobertas nem utilizadas unicamente pela Kaiser Permanente. Num estudo de 2015 sobre o valor monetário dos dados pessoais é apresentado que na era das redes sociais, tornou-se comum as pessoas partilharem opiniões, fotos e atividade em outras plataformas de vontade própria. (Schaar, 2013) Além deste hábito criado, foram introduzidas várias formas inconscientes de alimentar este tipo de base de dados, tal como na partilha da geo localização através de presença do telemóvel. Uma característica deste tipo de informação, reside no facto que pode ser vendida várias vezes sem alteração do seu valor intrínseco. É interessante para qualquer agência de marketing aceder ao máximo de preferências pessoais e hábitos de compras dos clientes de forma a melhorar as suas estratégias e mecanismos de venda.

No mesmo estudo, são também apresentadas duas perspetivas para medir o valor monetário deste tipo de informação: Aceder ao valor monetário das firmas responsáveis pela recolha dos dados ou o valor monetário que o individuo atribui aos seus dados. Juntando as duas perspetivas pode ser concluído que apesar de um único item de dados pessoais não seja relevante comercialmente, o conjunto do produto da recolha traz interesse comercial na combinação dos resultados por classes e na criação de perfis de controlo de atividade.

No ano de 2017 a empresa americana Equifax marcou a história por ter sido vítima da maior fuga de dados pessoais alguma vez registada (Equifax Data Breach Settlement, 2024).147 milhões de pessoas, não só nos Estados Unidos da América como também no Canadá e no Reino Unido foram vítimas da libertação dos seus dados devido a uma vulnerabilidade do site da empresa.

Muitos dos clientes foram afetados com roubo de identidade e atividades fraudulentas enquanto a Equifax despendeu uma enorme quantia em reparações legais para os danos causados. A reputação da empresa foi afetada, sendo criticada pelo público não só pelos danos causados, mas também na forma como tentou resolver a situação, resultando numa multa de 13 milhões de euros (Equifax Data Breach Settlement, 2024) . Este fenómeno causou consciencialização ao tema da proteção de dados e incluindo novas medidas pare evitar repetições do acontecimento.

O ato de privacidade dos estados unidos da américa foi implementado em 1974 para estabelecer procedimentos de como agência federais podiam recolher, utilizar e libertar informação pessoal. (Privacy Act of 1974, 1974) Na europa esse conceito foi introduzido uns anos mais tarde entre 1980 e 1990, resultando na diretiva de proteção de dados europeia em 1995 (Council, 1995).

Com o crescimento de importância deste tópico devido ao avanço tecnológico e crescimento exponencial de utilizadores da internet nas últimas décadas, foram aplicadas regras mais estritas para prevalecer o desafio da realidade atual. *General Data Protection Regulation* (GDPR) para a europa e *California Consumer Privacy Act* (CCPA) para os estados unidos.

Comparando a GDPR com a diretiva de 1995, esta reformulação das regras envolve todas as organizações mundiais que contenham informação pessoal de cidadãos europeus e não as presentes em solo europeu, como anteriormente. O valor das sanções pode atingir até 4% dos lucros atuais das organizações, foi introduzida a necessidade de consentimento por parte do utilizador e certas organizações podem precisar de um responsável pela atividade correta destas práticas, normalmente denominado de *Data Protection Officer* (DPO) (Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation, GDPR), 2016).

Nos primórdios do seu desenvolvimento, na década de 50, foram realizados avanços na direção de como a máquina interpreta a língua, introduzindo conceitos como *machine translation* e análise de texto. A experiência realizada pela universidade de Georgetown, em colaboração com a IBM, abriu portas de financiamento para o crescimento da linguística computacional (IBM, 1954) ,como um empurrão inicial até aos últimos desenvolvimentos dos tempos de hoje. O objetivo da experiência fundamentou-se em explorar o potencial de *machine translation* para construir um sistema de tradução de russo para inglês. O algoritmo de decisão funcionaria primeiramente como um dicionário de palavra a palavra, utilizando também uma série de regras para corrigir a frase de forma gramatical. Foram testadas 60 frases russas e o programa obteve sucesso na tradução de mais de metade com diferentes classificações de precisão (IBM, 1954).

Pouco tempo após esta experiência, foi sugerida por Noam Chomsky uma aplicação da teoria de *Transformational Grammar*, com o seu *framework* “*Transformational generative grammar*”.

Nesta teoria é proposto que todas as línguas partilham um conjunto de regras universal que facilita a assimilação feita pelo cérebro humano. O *framework* descreve uma estrutura rigorosa para a língua, contribuindo para sistemas mais avançados de *machine translation* e mais tarde algoritmos que seriam aplicados em *speech recognition, sentiment analysis* e outros métodos para processamento de linguagem natural.

Nas décadas de 80 e 90 foram introduzidas à linguística computacional, abordagens estatísticas. (Manning, 1999) Este método tem como objetivo a aplicação de modelos estatísticos para analisar e gerar de forma automática língua num estado natural. Uma das abordagens mais significativas e reconhecidas foi o *Hidden Markov Model* ou HMM. Onde seria analisada a probabilidade de palavras serem introduzidas numa frase de uma certa ordem. Desta forma pode ser prevista a aplicação de verbos, adjetivos ou nomes ordenadamente no contexto da língua. (Rabiner, 1989)

Finalmente, num período mais atual temos acompanhado o *deep learning* como o processo marcante da década não só para linguística computacional, como para as restantes áreas tecnológicas inseridas em vários setores. No decorrer desta pesquisa serão elaborados conceitos aplicados através deste método como: *language modeling*, *text classification, named entity recognition, sentiment analysis e machine translation*.

À medida que a quantidade de dados aumenta e o número de fontes de dados digitais explode, a necessidade de utilizar informações extraídas em processos de tomada de decisão torna-se cada vez mais urgente e desafiadora. Um problema onipresente é a natureza inicialmente desestruturada da maioria dos dados, o que significa que o formato dos dados apenas vagamente implica seu significado. Esses dados não estruturados são geralmente descritos utilizando linguagem natural compreensível por humanos, o que limita sua interoperabilidade por máquinas. Esse desafio impede a automação de tarefas críticas, como a recuperação de informações e processos de extração de informações usados na tomada de decisões ao lidar com conjuntos de dados extensos.

1.1 Objetivos

O objetivo principal deste projeto é explorar e organizar as várias metodologias, bibliotecas e ferramentas disponíveis no campo do Processamento de Linguagem Natural e do Reconhecimento de Entidades Nomeadas. Para além desta exploração teórica, será realizada uma análise prática das ferramentas atuais que permitem a conversão de ficheiros de diferentes formatos (como PDF, DOCX, entre outros) para o formato .txt, facilitando o seu processamento.

O projeto converge com o desenvolvimento de um protótipo que integra essas tecnologias, permitindo o reconhecimento de entidades nomeadas e a extração de informação pessoal a partir de ficheiros de texto. O objetivo do protótipo é garantir a deteção de dados pessoais que possam estar expostos a riscos de segurança, oferecendo uma solução robusta e eficiente para analisar e proteger esses dados. Este sistema reunirá o conhecimento adquirido sobre as metodologias de PLN e NER, juntamente com ferramentas de conversão de ficheiros, num protótipo funcional que automatiza a extração de informações relevantes de textos desestruturados.

1.2 Estrutura

O documento está estruturado em seis capítulos. No Capítulo 1, é feita a introdução ao tema, destacando o contexto, os objetivos do projeto e a sua relevância no campo do Processamento de Linguagem Natural.

No Capítulo 2, apresenta-se o estado da arte, abordando as bases teóricas de PLN e Reconhecimento de Entidades Nomeadas, conceitos centrais de *machine learning* aplicados ao projeto, e as diferentes abordagens de linguagem natural. São descritos os principais modelos de linguagem utilizados na área, bem como os métodos de pré-processamento e as métricas de avaliação empregadas para medir a eficácia das soluções. Este capítulo tem como objetivo fornecer um panorama detalhado dos fundamentos que sustentam o desenvolvimento do protótipo.

O Capítulo 3 é focado na análise e comparação das principais bibliotecas e ferramentas de PLN e NER disponíveis. Através de testes práticos, são exploradas as características, vantagens e limitações das bibliotecas, justificando a escolha das mais adequadas para a implementação no projeto. Este capítulo inclui *benchmarks* e discussões técnicas que sustentam as decisões adotadas.

No Capítulo 4, é descrita a arquitetura proposta para o projeto, detalhando os componentes principais, como a integração entre os modelos de NER e o pipeline de pré-processamento dos dados. Também são discutidos os requisitos técnicos e a infraestrutura necessária para garantir a escalabilidade e a eficiência da solução. Este capítulo oferece uma visão detalhada do design e da organização do sistema. É também apresentado o planeamento detalhado das várias camadas aplicacionais que compõem o projeto, abrangendo desde a interface gráfica do utilizador (GUI) até ao motor de reconhecimento de entidades nomeadas (NER). São exploradas em profundidade a camada de conversão de ficheiros, responsável pela transformação de diferentes formatos de documentos em texto simples, e o motor NER, que processa e identifica entidades nomeadas no

texto. Adicionalmente, é discutida a integração da API[1], que facilita a comunicação entre as diversas partes do sistema e garante a interoperabilidade entre módulos. São também abordadas as metodologias de desenvolvimento adotadas, com destaque para práticas ágeis que asseguram a flexibilidade e escalabilidade do projeto, assim como outras técnicas aplicadas para melhorar o desempenho e a segurança da aplicação.

O Capítulo 5 aborda o desenvolvimento e os testes do protótipo, desde a implementação inicial até as iterações de refinamento. São descritos os desafios enfrentados durante o desenvolvimento e como foram superados, bem como os resultados obtidos com base em testes práticos. Este capítulo fornece uma visão aprofundada do processo de construção do protótipo e da sua validação com dados reais. Finalmente, no Capítulo 7, são apresentadas as conclusões, reflexões sobre os resultados alcançados e sugestões para trabalhos futuros, incluindo possíveis melhorias e extensões ao protótipo desenvolvido. São discutidas ainda as direções promissoras para a continuidade do projeto e o seu potencial impacto no campo do PLN.

1.3 Motivação

O tema deste projeto surgiu da interseção entre três áreas de interesse: segurança da informação, desenvolvimento de software e curiosidade pela área de PLN, com um foco particular no reconhecimento de entidades nomeadas (NER). A segurança da informação, uma área crucial no mundo digital moderno, tem exigido soluções cada vez mais avançadas para garantir a proteção de dados sensíveis. Ao mesmo tempo, o desenvolvimento de software oferece a flexibilidade e o poder de criar ferramentas que podem automatizar e otimizar processos complexos. O NER em particular, se destaca como uma técnica fundamental para identificar informações críticas em grandes volumes de dados textuais, como nomes, endereços, números de telefone, entre outros dados pessoais ou sensíveis. Integrar NER com segurança da informação possibilita o desenvolvimento de sistemas mais inteligentes e eficientes para garantir a proteção dessas informações, permitindo, por exemplo, a anonimização de dados ou o reconhecimento automático de dados confidenciais em documentos. Este projeto reflete a fusão desses três ambientes, sendo não apenas uma solução técnica robusta, mas também uma ferramenta que aborda questões de privacidade e segurança, ao mesmo tempo em que explora o potencial da inteligência artificial na extração de conhecimento valioso a partir de dados em formato de texto.

[1] conjunto de regras que permite a comunicação entre diferentes sistemas ou aplicações, facilitando a troca de dados e funcionalidades.

2. Estado da arte

Nos últimos anos, o campo de PLN tem experienciado avanços notáveis, impulsionados principalmente pelo surgimento de novas técnicas de *machine learning* e pelo acesso a grandes volumes de dados. Um dos aspetos fundamentais do PLN, e que serve como alicerce para inúmeras aplicações práticas, é o NER. Esta tarefa, que consiste na identificação e classificação de entidades nomeadas em textos, como pessoas, organizações e locais, tornou-se um componente essencial para a extração de informações, análise semântica e inteligência de dados. (Jurafsky, 2023) O objetivo deste capítulo é apresentar um panorama atualizado sobre o estado da arte em NER e PLN, explorando tanto os avanços tecnológicos quanto os desafios que persistem. Será abordado o desenvolvimento histórico dessas tecnologias, desde os primeiros sistemas baseados em regras até os recentes modelos baseados em redes neurais e linguagem universal. O foco desses avanços têm permitido uma compreensão cada vez mais refinada da linguagem humana, além de discutir as implicações práticas desses desenvolvimentos em diversos setores. Adicionalmente, este capítulo contextualizar a importância da diversidade dos dados de treino e o impacto das inovações em arquiteturas de modelos, como BERT e XLNet, no desempenho do NER. Também será explorado como a integração do NER com outras tarefas de PLN está a abrir novas frentes para a pesquisa e aplicações práticas em áreas como análise de sentimentos, assistentes virtuais e sistemas de recomendação.

À medida que são explorados os fundamentos teóricos do PLN, será realizado um aprofundamento nas metodologias e conceitos que permitem às máquinas entender, interpretar e gerar linguagem humana de forma eficaz e significativa. Será abordada uma ampla gama de tópicos essenciais que formam a espinha dorsal do PLN, desde os processos básicos de pré-processamento de texto, como tokenização e lematização, até técnicas mais avançadas de análise linguística, incluindo *Part-of-Speech Tagging*, análise sintática e Reconhecimento de Entidades Nomeadas. Além disso, serão apresentadas as abordagens de *machine learning* que revolucionaram o campo do PLN nos últimos anos. Examinaremos desde modelos estatísticos tradicionais até as recentes inovações em *deep learning* que têm impulsionado avanços significativos em tarefas complexas de PLN. A representação de texto e os modelos de linguagem, que transformam palavras e frases em formatos compreensíveis por máquinas, também serão temas centrais desta seção. Além disso, serão abordadas as métricas de avaliação que nos permitem medir o sucesso e a precisão dos sistemas de PLN, um aspeto crítico para o desenvolvimento e aprimoramento de tecnologias baseadas em linguagem. Finalmente, serão reconhecidos os desafios e questões éticas emergentes no PLN, contemplando o impacto das tecnologias de linguagem na sociedade.

2.1 *Machine learning* e técnicas

Neste ponto serão exploradas as técnicas de ML, uma variedade de métodos e modelos que permitem que os sistemas de PLN aprendam a partir de dados linguísticos, possibilitando a automação e a melhoria de tarefas como tradução de linguagem, reconhecimento de fala, e análise de sentimentos. Nesta secção, serão discutidas abordagens fundamentais, como aprendizagem supervisionada e não supervisionada acrescentando uma abordagem híbrida e como se aplicam a diferentes desafios de PLN.

2.1.1 Machine learning

Desempenha um papel crucial em capacitar sistemas de IA a compreender, processar e produzir linguagem humana. Na prática, pode ser resumido em treinar ou ensinar computadores com exemplos. Tal como os seres humanos aprendem com experiências, como reconhecendo algo anteriormente visto, o computador aprende a partir de dados. Desde os seus primórdios, a área de *Machine Learning* evoluiu consideravelmente, deixando marcos importantes na história da computação. Divide-se em categorias principais como: aprendizagem supervisionada, aprendizagem não supervisionada e aprendizagem por reforço, cada uma com a sua aplicabilidade e técnicas específicas. Os computadores, graças ao ML, conseguem fazer previsões, identificar padrões e tomar decisões sem serem explicitamente programados para tal. No mundo real, isso traduz-se em aplicações práticas em diversas indústrias, como na saúde, finanças e transporte. Contudo, a área enfrenta desafios como preocupações com a privacidade dos dados, considerações éticas e limitações das tecnologias atuais. O futuro do ML promete avanços na pesquisa e desenvolvimento, potencialmente transformando ainda mais as capacidades da IA. Esta área interceta com outras disciplinas, incluindo estatística, ciência da computação e ciência cognitiva, ilustrando a sua natureza multidisciplinar. Tecnologias e algoritmos fundamentais, como redes neurais, árvores de decisão e algoritmos de descida de gradiente, são muito utilizados no *Machine Learning*.

2.1.2 Modos de Treino IA

Existem dois tipos de treino de modelos aplicados, aprendizagem supervisionada, aprendizagem não supervisionada. (Goodfellow, 2016) O método supervisionado envolve o treino de modelos com dados rotulados (ou *tagged*), nos quais os dados de *input* (texto) estão associados a anotações específicas. Por exemplo, treinar um modelo de análise de sentimento envolve o uso de conjuntos de dados com amostras de texto rotuladas como sentimentos positivos ou negativos. No modelo

não supervisionado o treino funcionada à base de dados não rotulados, ou seja, dados que não possuem rótulos ou categorias específicas associadas a eles. Essa abordagem é fundamental em várias tarefas onde o objetivo é descobrir padrões intrínsecos nos dados de texto, como estruturas semânticas, agrupamentos de palavras ou tópicos subjacentes. Existe um terceiro paradigma, não tão aplicado a treino no contexto de PLN, onde um agente é treinado a fazer certas decisões, sendo recompensado ao longo do tempo. (Sutton, 2018) O agente interage com o ambiente tomando ações e experienciando as consequências. Baseando-se nas observações prévias o agente ajusta a sua estratégia para melhorar a sua performance.

2.1.3 Deep Learning

Caracterizado como um subconjunto de *machine learning*, (Goodfellow, 2016) revolucionou a forma como abordamos problemas computacionais complexos, particularmente nos processos de PLN. No seu núcleo, o *deep learning* envolve redes neurais com múltiplas camadas, permitindo que estes modelos aprendam padrões em grandes conjuntos de dados. No PLN, o *deep learning* tem sido fundamental no desenvolvimento de métodos mais avançados e eficientes de processamento e compreensão da linguagem humana. (Goldberg, 2017) Permite a criação de modelos que podem representar com precisão as nuances e o contexto da linguagem, uma perspetiva desafiadora com as técnicas computacionais tradicionais.

As principais contribuições do *deep learning* para o PLN incluem o desenvolvimento de modelos sofisticados como o BERT e o GPT, que estabeleceram novos padrões em tarefas como classificação de texto, análise de sentimentos e tradução automática. (Devlin, 2019) (Radford, 2019) Estes modelos utilizam grandes quantidades de dados para entender o contexto, ambiguidade e as subtilezas da linguagem. Além disso, o *deep learning* popularizou o uso de *embeddings* de palavras, onde as palavras são convertidas em vetores numéricos. Esta representação capta relações semânticas e ajuda na compreensão da linguagem. A capacidade dos modelos de *deep learning* de lidar com dados sequenciais, como frases, tem sido crucial para tarefas como modelagem de linguagem e produção de texto. (Hochreiter, 1997)

2.1.4 Redes Neuronais

As redes neurais são um dos pilares da inteligência artificial, inspiradas no funcionamento do cérebro humano. Essas estruturas são formadas por unidades básicas chamadas neurónios artificiais, que se assemelham aos neurónios biológicos. (Goldberg, 2017) Esses neurónios são organizados em camadas: uma camada de entrada, uma ou mais camadas ocultas e uma camada de saída. O funcionamento de uma rede neural começa com a alimentação dos dados de entrada.

Esses dados são processados sequencialmente pelas camadas da rede. Em cada camada, os dados de entrada são transformados utilizando-se pesos[2] e ajustes[3], que são ajustados durante o processo de aprendizagem. (Goodfellow, 2016) A função de ativação, um componente crucial de cada neurónio, determina se ele deve ser ativado ou não, com base na soma ponderada de suas entradas. O treino de uma rede neural envolve a atualização desses pesos e ajustes para minimizar o erro entre as saídas previstas pela rede e as saídas esperadas (verdadeiras). (Nielsen, 2015) A flexibilidade e capacidade de aprendizagem fazem das redes neurais uma ferramenta poderosa na resolução de problemas complexos.

2.1.5 Variantes de Aplicações a Linguagem Natural

Nesta subsecção, serão abordadas duas variantes fundamentais das aplicações de Processamento de Linguagem Natural (PLN): o Natural Language Understanding (NLU) e o Natural Language Generation (NLG). O NLU foca-se na compreensão da linguagem, permitindo que os sistemas interpretem o significado e a estrutura de textos, enquanto o NLG trata da geração automática de texto, permitindo que máquinas criem respostas ou conteúdos em linguagem natural. Ambas as variantes desempenham um papel crucial em diversas aplicações, desde assistentes virtuais até sistemas de tradução automática.

a) Natural Language Understanding

Vai além do simples reconhecimento de linguagem, o seu objetivo é dotar as máquinas com a capacidade de verdadeiramente entender a intenção, o sentimento e as nuances embutidas na comunicação humana. Envolvendo o reconhecimento de elementos linguísticos, como palavras e gramática, bem como a compreensão mais profunda da semântica, pragmática e das subtilezas da expressão humana. Em termos práticos, a NLU possui aplicações abrangentes, desde assistentes de voz que podem participar de conversas naturais até ferramentas de análise de sentimentos que avaliam a opinião pública nas redes sociais. Funciona como a ponte entre a comunicação humana e de máquina.

b) Natural Language Generation

---

[2] Valores numéricos que determinam a importância de uma entrada específica

[3] Ajudam a rede a se adaptar melhor aos dados, permitindo um ajuste fino das saídas

As máquinas são treinadas para gerar conteúdo escrito ou falado que não apenas é coerente e gramaticalmente correto, mas também contextualmente relevante e significativo. O principal objetivo da NLG é capacitar os computadores a produzir texto que possa informar, persuadir, entreter ou ajudar os utilizadores de forma semelhante à comunicação humana. (Gatt, 2018) Envolve a conversão de dados estruturados ou informações em linguagem legível pelo ser humano. Os sistemas de NLG têm uma ampla gama de aplicações, desde a geração automatizada de relatórios e *chatbots*[4] que mantêm conversas naturais até a sumarização de conteúdo e recomendações personalizadas.

2.2 Modelos de Linguagem

Os modelos de linguagem são um componente fundamental do cenário digital moderno, revolucionando a forma como os utilizadores interagem com a tecnologia e compreendem o vasto mundo da linguagem natural. Estes modelos são construídos com base nos princípios da inteligência artificial e de *machine learning*, em particular, técnicas de *deep learning*, e fizeram avanços significativos nos últimos anos. No seu cerne, um modelo de linguagem é um sistema computacional projetado para compreender e produzir linguagem humana. (Vaswani, 2017) Serve como uma ponte entre as nuances da comunicação humana e o mundo binário dos computadores. Os modelos de linguagem são treinados em conjuntos de dados extensos que abrangem grandes quantidades de texto e são capazes de decifrar os padrões, estruturas e semântica intrincada que sustentam a linguagem. (Radford, Language models are unsupervised multitask learners, 2019) Aprendem como as palavras se relacionam umas com as outras, como a gramática funciona e como o contexto influencia o significado. Os modelos de linguagem usam um processo conhecido como "pré-treino" e "afinação". Durante o pré-treino, um modelo é exposto a um conjunto massivo de dados e aprende a prever a próxima palavra numa frase. Isso equipa o modelo com uma compreensão de gramática, vocabulário e algum raciocínio baseado em senso comum. Na fase de afinação, o modelo é especializado em tarefas ou aplicativos específicos, como tradução, resumo ou resposta a perguntas.

2.2.1 Bag of Words

[4] programa que simula conversas com utilizadores, usando inteligência artificial

Neste modelo, um texto é representado como o conjunto de palavras, desconsiderando a gramática e até a ordem das palavras, mas mantendo a multiplicidade (quantas vezes cada palavra aparece). A ideia básica por trás do BoW é converter textos em vetores numéricos ou matrizes, onde cada elemento representa a presença ou a frequência de uma palavra específica no documento. Por exemplo, se tivermos um vocabulário composto pelas palavras {gato, ama, peixe}, e um documento de texto diz "gato ama gato", a representação em BoW seria [2, 1, 0], indicando que "gato" aparece duas vezes, "ama" aparece uma vez, e "peixe" não aparece. Este modelo é amplamente utilizado em tarefas como classificação de documentos, análise de sentimentos e filtragem de spam, pois permite que os algoritmos de ML tratem dados textuais. No entanto, uma desvantagem do BoW é que ignora a ordem das palavras e o contexto, o que pode levar à perda de informações importantes.

2.2.2 Contextual Embeddings

Enquanto os *word embeddings* tradicionais, fornecem uma representação única e estática para cada palavra, independentemente do contexto em que aparecem, *contextual embeddings* geram representações de palavras que variam de acordo com o contexto em que são usadas. (Mikolov, 2013) Esta abordagem permite que palavras com múltiplos significados (polissemia) tenham *embeddings* diferentes dependendo do seu uso específico em uma frase ou texto. Por exemplo, a palavra "banco" teria um *embedding* diferente quando usada no contexto de "banco financeiro" em comparação com "banco de praça". *Contextual embeddings* são gerados usando modelos de linguagem baseados em redes neurais profundas, como o BERT, ELMo e GPT. Estes modelos são treinados em grandes corpora[5] de texto e aprendem a gerar representações de palavras levando em consideração todo o contexto da frase ou do texto, o que resulta em *embeddings* mais ricos e informativos.

2.2.3 TF-IDF

Term Frequency-Inverse Document Frequency é uma técnica utilizada para representar texto e avaliar a importância de palavras em um documento, considerando o contexto de um conjunto maior de documentos, conhecido como *corpus*. Esta técnica funciona através de dois componentes principais. O primeiro é o *Term Frequency* (TF), que mede a frequência com que uma palavra aparece em um documento. A ideia por trás do TF é que palavras que aparecem com

[5] Grandes coleções de textos usados para análise linguística e treino de modelos de linguagem.

mais frequência em um documento específico são mais importantes para aquele documento. (Manning C. D., 2008) O segundo componente é o Inverse Document Frequency (IDF), que atribui peso a uma palavra com base em sua raridade em todo o corpus. Palavras muito comuns, como artigos e preposições, recebem um peso menor, enquanto palavras que aparecem em poucos documentos, mas que são importantes, recebem um peso maior. O valor de TF-IDF é obtido multiplicando o TF pelo IDF, resultando em uma métrica que destaca as palavras que são importantes em um documento específico, mas raras no *corpus* como um todo. Isso torna a técnica muito eficaz para tarefas como classificação de texto, (Manning C. D., 2008) pesquisa de informações e extração de palavras-chave, pois ajuda a identificar termos relevantes, ignorando os mais comuns e menos informativos

2.2.4 N-grams

Técnica que envolve a modelação de sequências de palavras ou caracteres num documento. Um N-grama é uma sequência de N itens consecutivos num texto. Por exemplo, no caso dos bigramas (N=2), são analisadas sequências de duas palavras consecutivas; nos trigramas (N=3), são analisadas três palavras consecutivas, e assim sucessivamente. (Jurafsky, 2023) Esta abordagem é útil porque, ao contrário de técnicas como o Bag of Words (que considera as palavras individualmente, sem contexto), os N-gramas capturam alguma noção de dependência sequencial entre palavras ou caracteres, preservando o contexto local. Isto permite uma análise mais eficaz de expressões ou frases, já que muitas vezes o significado de uma palavra depende das que a antecedem ou sucedem.

2.2.5 FastText

Técnica de representação de texto desenvolvido pela equipa de pesquisa do Facebook AI. É uma extensão do popular modelo Word2Vec, mas com uma diferença importante: em vez de representar palavras inteiras como vetores únicos, o FastText representa palavras como conjuntos de subpalavras ou n-grams de caracteres. (Bojanowski, 2017) Isso permite que o modelo capture informações sobre a morfologia das palavras, tornando-o mais eficaz para lidar com palavras raras ou novas, que podem não aparecer no vocabulário de outros modelos de *embeddings*, como Word2Vec.

Ao quebrar as palavras em subpartes menores, *FastText* consegue lidar melhor com variações ortográficas, sufixos e prefixos, o que é especialmente útil em linguagens ricas em morfologia ou para tarefas que envolvem muitos termos técnicos ou palavras que não aparecem frequentemente em grandes *corpora* de texto. Além disso, é conhecido por ser rápido e eficiente, tanto na fase de treino como na implementação, tornando-o uma escolha popular para tarefas como classificação

de texto, deteção de idiomas, e representação de palavras. Combina a precisão das representações baseadas em subpalavras com a escalabilidade e velocidade necessárias para grandes volumes de dados.

2.2.6 Métodos de processamento de linguagem natural

A capacidade de interpretar e analisar a linguagem humana através de computadores envolve uma variedade de técnicas sofisticadas e métodos especializados. Cada um desses métodos desempenha um papel crucial na transformação do texto natural em dados estruturados e compreensíveis para as máquinas. Entre as técnicas fundamentais empregadas no PLN estão a lematização, o *POS Tagging*, a análise sintática, o NER e a extração de informação. (Manning C. D., 2008) Estas técnicas variam em função e complexidade, mas juntas, formam o alicerce sobre o qual as tarefas de PLN são construídas e desenvolvidas. Elas permitem que os sistemas de PLN abordem uma ampla gama de desafios linguísticos e contextuais, desde a simplificação e normalização do texto até a identificação de componentes chave e a extração de significados relevantes.

a) Normalização

Processo criado para converter o texto numa forma padrão. É essencial porque a linguagem natural é frequentemente confusa e existem muitas variações de palavras e frases que significam a mesma coisa. O pré-processamento ajuda a garantir que palavras ou frases semelhantes sejam tratadas como idênticas.

- *Lowercasing*: Conversão completa de texto para letra minúscula
- *Stemming*: Redução de palavras para a sua raiz.
- *Lemmatization*: Processo semelhante a *stemming,* porém focado no significado da palavra.

O pré-processamento de texto não é uma abordagem única e deve ser adaptado aos requisitos específicos da tarefa de PLN em questão. A extensão do pré-processamento pode variar com base em fatores como o idioma, domínio e os objetivos da tarefa. Um pré-processamento de texto eficiente é essencial para melhorar o desempenho de modelos de PLN, tornar os dados de texto mais simples de gerir e extrair *insights* significativos. (Manning C. D., 1999) Muitas vezes, é a base sobre a qual várias tarefas de PLN, como classificação de texto, análise de sentimentos,

recuperação de informações e tradução automática, dependem para produzir resultados precisos e confiáveis.

b) Tokenização

Envolve a divisão de uma sequência de texto, como uma frase ou um documento, em secções mais pequenas chamadas *tokens*. Os *tokens* são normalmente palavras ou partes de palavras, como sinais de pontuação, números ou outros pedaços de texto com significado. O processo de funcionamento envolve a análise do texto caractere a caractere, a aplicação de regras para identificar os limites entre os *tokens* e a produção de uma sequência de *tokens* como resultado. Inicialmente, o texto é examinado ao nível dos caracteres. Caracteres de espaço em branco, como espaços e *tabs*, são geralmente usados como separadores naturais entre palavras. O texto é dividido nessas posições de espaço em branco, criando *tokens* individuais de palavras. Sinais de pontuação, como pontos, vírgulas, pontos de exclamação e hífenes, são tratados com cuidado. Muitas vezes, são tratados como *tokens* separados para preservar o seu significado. Casos especiais, como contrações (por exemplo, "I'm") e palavras com hífenes (por exemplo, "*state-of-the-art*"), são considerados para manter a sua integridade. A tokenização de subpalavras pode ser utilizada, especialmente em línguas com formação complexa de palavras.

- Granularidade: nível de detalhe em que o texto é dividido em *tokens*.
- Análise: processo de dividir um texto em *tokens*.

Um dos desafios mais comuns é a ambiguidade. As palavras em uma língua muitas vezes têm múltiplos significados, e a tokenização deve determinar o significado apropriado com base no contexto. (Jurafsky, 2023) Por exemplo, a palavra "*bank*" pode referir-se tanto a uma instituição financeira quanto a um objeto para sentar; Contrações e hífens também são fontes de desafio. Contrações como "I'm" (para "I am") e palavras compostas, como "*state-of-the-art*," podem ser divididas de maneira incorreta, afetando a compreensão do texto; Nomes próprios, como de pessoas, locais e empresas, podem não seguir as regras gramaticais comuns e, portanto, exigem tratamento especial durante a tokenização. Acrônimos, como "NASA" ou "UNESCO," devem ser reconhecidos como entidades únicas, e a quebra incorreta de acrônimos pode causar confusão. Quando textos incorporam várias línguas ou misturam idiomas, a tokenização precisa ser adaptada a cada língua presente. Em domínios específicos, como medicina ou tecnologia, terminologia especializada pode conter termos e siglas que não estão nos dicionários gerais. Variações dialeto-

regionais, como no inglês com suas diferenças regionais, também podem impactar a tokenização, especialmente em palavras e expressões específicas de uma região.

Para superar esses desafios, os sistemas de tokenização utilizam diversas abordagens, como regras, dicionários, *machine learning* e processamento de linguagem natural avançado. (Jurafsky, 2023) A escolha da abordagem depende das necessidades da tarefa e do contexto linguístico. Lidar com esses desafios é fundamental para garantir que a tokenização seja precisa e eficaz em uma ampla variedade de cenários de PLN.

c) Lematização

Processo que visa reduzir uma palavra à sua forma base ou dicionário, conhecida como "lema". Diferentemente do processo de "s*temming*", que meramente remove os sufixos das palavras, a lematização considera a análise morfológica das palavras, levando em conta seu uso no contexto para determinar o lema correto. Isso significa que a lematização é capaz de distinguir entre diferentes significados de uma palavra com base em sua parte do discurso (por exemplo, distinguindo "meeting" como substantivo do verbo "*meet*"). Um dos principais objetivos da lematização é tratar as variações morfológicas das palavras, permitindo que análises subsequentes no PLN, como a indexação de texto e a recuperação de informações, sejam realizadas de maneira mais eficaz. Ao converter palavras para as suas formas base, a lematização ajuda a reduzir a complexidade do texto e a aumentar a chance de correspondência de padrões em tarefas de PLN.

d) Remoção de stop words

Envolve a eliminação de palavras frequentes e com pouca significância semântica num texto, conhecidas como "*stop words*". Estas incluem preposições, conjunções, artigos e pronome, como "e", "ou", "mas", "o", "a", "de", "em", "para", "com", entre outras. A principal razão para remover *stop words* é reduzir o ruído nos dados e focar nas palavras que são mais significativas para a análise ou modelagem em questão. Isso não só ajuda a diminuir o tamanho do *dataset*, tornando o processamento de dados mais rápido e eficiente, mas também melhora a performance de diversas tarefas de PLN, como classificação de texto, análise de sentimentos e busca de informações. No entanto, é importante notar que a lista de *stop words* pode variar dependendo do idioma e do contexto específico da aplicação. A decisão de remover ou manter *stop words* deve ser considerada cuidadosamente com base nos objetivos específicos do projeto.

2.3 Análise Linguística

Nesta secção, será abordada a Análise Linguística, uma componente essencial do PLN, que permite extrair significado e estrutura de textos. Serão apresentados e discutidos vários processos fundamentais, como o POS *Tagging* (marcação de categorias gramaticais), que identifica as classes gramaticais de cada palavra num texto; a Análise Sintática, que estrutura as relações entre as palavras; NER, utilizado para identificar elementos importantes como nomes, datas e localizações; e, por fim, a Extração de Informação, que visa recolher dados específicos e relevantes a partir de grandes volumes de texto. Cada uma destas técnicas contribui para uma compreensão mais profunda da linguagem e para o desenvolvimento de sistemas automatizados mais eficazes e precisos.

a) *Part-of-Speech Tagging*

Envolve a atribuição de rótulos a cada palavra num texto, indicando sua classe gramatical, como substantivo, verbo, adjetivo, entre outros. Este processo é crucial para muitas aplicações de PLN, pois fornece uma compreensão básica da estrutura gramatical das sentenças, permitindo análises mais sofisticadas e precisas. Geralmente realizado usando modelos de *machine learning* treinados em grandes corpora anotados, onde os padrões linguísticos e contextuais são aprendidos para prever a parte do discurso de palavras em novos textos. A precisão desses modelos depende significativamente da qualidade do treino e da complexidade da língua sendo analisada, dada a presença de palavras com múltiplas funções gramaticais. As capacidades de POS contribuem para a análise sintática, a desambiguação de palavras pelo contexto, a melhoria da precisão em sistemas de tradução automática e a contribuição para algoritmos de reconhecimento de entidades nomeadas (NER) e extração de informação

b) Análise sintática

Processo que se concentra em decompor uma frase na sua estrutura gramatical, identificando as relações entre palavras e frases. O objetivo principal da análise sintática é entender como os componentes de uma frase se organizam em uma árvore sintática, que reflete a hierarquia gramatical e a função de cada palavra ou frase dentro da sentença. Esta técnica é fundamental para uma série de aplicações avançadas de PLN, pois fornece uma base para a compreensão da estrutura lógica e gramatical dos textos. Com essa compreensão, máquinas podem realizar tarefas como tradução automática, reconhecimento de fala, e geração de linguagem natural com maior

precisão e naturalidade. Existem dois tipos principais de análise sintática: a análise sintática superficial (*shallow parsing*), que segmenta um texto em suas constituintes primárias, como nomes e verbos, sem construir uma hierarquia profunda; e a análise sintática profunda (*deep parsing*), que constrói uma árvore completa, detalhando como todas as partes da frase se relacionam entre si. (Jurafsky, 2023) Os algoritmos de *parsing* podem variar desde abordagens baseadas em regras, que utilizam conjuntos complexos de instruções gramaticais, até modelos baseados em *machine* e *deep learning*, que aprendem a estrutura gramatical a partir de grandes volumes de texto anotado.

c) Reconhecimento de Entidades Nomeadas

Envolve a identificação e classificação de entidades específicas em textos, tais como nomes de pessoas, organizações, localizações, expressões de tempo, quantidades, valores monetários, entre outros. O NER é fundamental para extrair informações estruturadas de dados não estruturados, facilitando a compreensão automática de documentos e a produção de insights relevantes. Esta técnica emprega modelos de *machine learning* e *deep learning* para analisar o contexto e as características linguísticas das palavras numa frase, permitindo a identificação precisa de diferentes tipos de entidades. (Yadav, 2018) O sucesso do NER depende da capacidade do modelo em reconhecer padrões linguísticos e semânticos, o que é frequentemente alcançado através do treino em grandes conjuntos de dados anotados manualmente. O NER tem uma ampla gama de aplicações práticas, incluindo a melhoria de sistemas de busca e recuperação de informação, análise de sentimentos, sumarização automática de textos, e assistentes virtuais inteligentes.

d) Extração de informação

Concentra-se na organização e no acesso eficiente a grandes volumes de dados não estruturados, com o objetivo de localizar informações relevantes em resposta a consultas de utilizadores. A EI desempenha um papel vital na era da informação, em que vastas quantidades de dados digitais estão disponíveis em diversas formas, como documentos de texto, imagens, áudio e muito mais. (Sarawagi, 2008) No foco da EI está o desafio de criar sistemas capazes de recuperar informações de maneira precisa e eficaz, considerando a diversidade de formatos, tipos de conteúdo e idiomas. Esses sistemas desempenham um papel fundamental em uma variedade de cenários, desde motores de busca na web até bibliotecas digitais, passando por sistemas de gestão de documentos e sistemas de suporte à decisão. A EI envolve uma série de tarefas interconectadas, desde a indexação de documentos até a formulação de consultas e a classificação de resultados com base

na relevância. A eficácia desses sistemas depende de técnicas avançadas de processamento de linguagem natural, modelagem estatística e algoritmos de busca. À medida que a quantidade de dados digitais continua a crescer exponencialmente, a EI desempenha um papel ainda mais crucial nossa sociedade digital, fornecendo as ferramentas e os métodos necessários para encontrar informações relevantes no campo dados.

2.4 Avaliação e Métricas

Nesta secção, serão abordadas as principais métricas utilizadas para avaliar o desempenho de sistemas de Processamento de Linguagem Natural, com ênfase em técnicas de Reconhecimento de Entidades Nomeadas. Entre as métricas a serem analisadas encontram-se a precisão, a sensibilidade e o *F1-score*, que são amplamente utilizadas para medir a eficácia de modelos de PLN em tarefas de classificação e extração de informações. Estas métricas fornecem uma visão objetiva da qualidade dos resultados gerados pelo sistema, permitindo identificar áreas que necessitam de melhorias ou otimizações.

Além disso, serão apresentadas outras métricas específicas da área de PLN, que ajudam a avaliar o desempenho do modelo em cenários linguísticos mais complexos, como o reconhecimento de ambiguidades e variações no uso da linguagem. A combinação dessas métricas permite uma análise abrangente e detalhada, garantindo uma avaliação completa da eficiência do sistema.

2.4.1 Precisão, Sensibilidade e *F1-score*

Métricas usadas para avaliar o desempenho de modelos de classificação e outras áreas de ML. São especialmente importantes em contextos onde o custo de falsos positivos e falsos negativos varia significativamente. Por exemplo, em um sistema de diagnóstico médico, um falso negativo (um caso real de doença que não é detetado) pode ser muito mais grave que um falso positivo (um não doente diagnosticado como doente). Nesse caso, pode-se desejar maximizar a sensibilidade, enquanto em outras situações, a precisão pode ser mais crítica.

a) Precisão

Refere-se à proporção de identificações positivas feitas pelo modelo que são realmente corretas. É calculada dividindo o número de verdadeiros positivos (TP) pelo número total de positivos identificados pelo modelo (a soma dos verdadeiros positivos e falsos positivos, TP + FP). (Manning C. D., 2008) A precisão, portanto, mede a qualidade dos resultados positivos do modelo.

b) Sensibilidade

Refere-se à proporção de casos positivos reais que foram corretamente identificados pelo modelo. É calculada dividindo o número de verdadeiros positivos (TP) pelo número total de casos positivos reais (a soma dos verdadeiros positivos e falsos negativos, TP + FN). A sensibilidade, portanto, mede a capacidade do modelo de encontrar todos os casos relevantes.

c) F1 – Score

É a média harmônica de precisão e sensibilidade, proporcionando um único indicador de desempenho que leva em conta tanto a precisão quanto a sensibilidade. O *F1-Score* é particularmente útil quando é necessário encontrar um equilíbrio entre precisão e sensibilidade e quando há uma distribuição desigual de classes (isto é, quando as instâncias de uma classe são muito mais comuns do que as outras).

$$\frac{2 * (\text{Precisão} * \text{Sensibilidade})}{(\text{Precisão} + \text{Sensibilidade}).}$$

2.4.2 Métricas especificas de tarefas de PLN

Nesta secção, serão discutidas as principais métricas específicas utilizadas na avaliação de tarefas de PLN. Estas métricas desempenham um papel crucial na medição da eficácia de modelos de linguagem em diferentes cenários, permitindo avaliar o desempenho de forma quantitativa e comparativa. Serão apresentadas métricas como *Word Error Rate* (WER), utilizada frequentemente para avaliar a precisão em tarefas de reconhecimento de fala; ROUGE e BLEU, amplamente aplicadas em tarefas de tradução automática e sumarização de texto; *Perplexity*, uma métrica normalmente usada para avaliar a qualidade de modelos de linguagem; e METEOR, que, tal como BLEU, é usada para avaliar a qualidade de traduções.

a) *Bilingual Evaluation Understudy* (BLEU)

Para tarefas de tradução automática, a métrica BLEU avalia a qualidade das traduções de máquina comparativamente com traduções humanas. Mede a correspondência de n-gramas entre o texto traduzido e o texto de referência, ajustada por brevidade para penalizar traduções excessivamente curtas. (Papineni, 2002) Embora predominantemente usada em tradução, o BLEU também pode ser aplicado em tarefas de produção de texto.

b) *Metric for Evaluation of Translation with Explicit Ordering* (METEOR)

Usada para avaliar a tradução automática, semelhante ao BLEU, mas com algumas diferenças importantes. O METEOR considera sinônimos e a estrutura gramatical das frases, além da correspondência exata de palavras, proporcionando uma avaliação mais refinada. (Banerjee, 2005)

c) *Word Error Rate* (WER)

Principalmente para avaliar sistemas de reconhecimento de fala, a WER é calculada comparando o número de erros na transcrição gerada pelo computador em relação ao texto de referência. (Morris, 2004) Uma taxa de erro de palavra mais baixa indica um melhor desempenho do sistema de reconhecimento de fala.

d) *Recall-Oriented Understudy for Gisting Evaluation* (ROUGE)

Semelhante ao BLEU, mas mais utilizado em tarefas de sumarização de texto. O ROUGE inclui várias medidas, como ROUGE-N (que compara n-gramas do resumo produzido com os resumos de referência), ROUGE-L (que considera a mais longa subsequência comum), entre outros. (Lin, 2004) Foca-se tanto em precisão quanto em *sensibilidade*.

e) Perplexity

Utilizada em modelos de linguagem mede quão bem uma distribuição de probabilidade ou modelo de linguagem prevê uma amostra. Um valor mais baixo indica que o modelo de linguagem é melhor em prever a amostra. É especialmente útil para avaliar modelos de linguagem e sistemas de recomendação de texto. (Jelinek, 1997)

2.5 Transformadores de linguagem

Classe inovadora de modelos de *deep learning* revolucionários no campo de PLN, analisam todas as palavras de um texto simultaneamente. Isso permite que considerem o contexto completo de

cada palavra, melhorando significativamente a compreensão e a produção de texto. Graças à sua capacidade de capturar nuances linguísticas e entender o contexto amplo, os transformadores são usados em uma variedade de aplicações de PLN. Inclui tradução automática, onde eles podem traduzir textos entre idiomas com alta precisão; geração de texto, permitindo que eles produzam conteúdo coerente e relevante; e análise de sentimentos, onde podem avaliar as emoções expressas em textos escritos.

a) *Bidirectional Encoder Representations from Transformers* (BERT)

Desenvolvida pelo Google, é notável pela sua capacidade de entender o contexto de uma palavra com base em todo o texto ao seu redor, em vez de apenas as palavras anteriores ou seguintes. Isso é alcançado por meio de um mecanismo de atenção bidirecional**,** permitindo que o modelo gere representações de linguagem ricas e contextualizadas. (Devlin J. C., 2019)

b) *Generative Pretraining Transformer* (GPT)

Desenvolvido pela OpenAI, destaca-se pela capacidade de gerar texto coerente e contextualmente relevante a partir de um *prompt*. O modelo é treinado com uma grande quantidade de texto e aprende a prever a próxima palavra em uma frase, o que possibilita a capacidade para gerar conteúdo em diversas aplicações, como tradução, resumo e criação de conteúdo**.** (Radford, Language Models are Unsupervised Multitask Learners, 2019) O GPT tem várias versões, com melhorias contínuas em termos de tamanho, complexidade e capacidade de compreensão de texto.

c) *Robustly optimized BERT approach* (RoBERTa)

Baseado na arquitetura BERT, desenvolvido pela Facebook AI. Foi criado para melhorar o desempenho do BERT em tarefas de PLN através de um treino mais longo, com mais dados e sem a segmentação de *next-sentence* durante o treino. Superou modelos anteriores em várias *benchmarks* de PLN, estabelecendo novos padrões para compreensão de linguagem (Liu, 2019).

2.6 Desafios Éticos

A ascensão da utilização de PLN como uma ferramenta fundamental na tecnologia da informação trouxe consigo uma série de desafios éticos significativos. Esses desafios derivam da natureza intrínseca dos dados linguísticos, da complexidade dos algoritmos de PLN e do impacto que essas tecnologias têm sobre os indivíduos e a sociedade. Um dos principais desafios éticos reside no

facto sistemas de PLN poderem perpetuar ou até amplificar preconceitos existentes nos dados em que são treinados. Isso pode resultar em discriminação e injustiça (Bender, 2021), especialmente quando aplicado a contextos como contratação de empregos, decisões judiciais ou publicidade direcionada. Além disso, a privacidade dos dados é uma grande preocupação. O PLN muitas vezes requer o processamento de grandes volumes de texto, que podem incluir informações pessoais sensíveis. A recolha, armazenamento e análise desses dados devem ser realizados com cuidado para proteger a privacidade dos indivíduos e garantir que seu consentimento seja obtido de forma adequada. A transparência e a explicabilidade dos sistemas de PLN também são questões críticas. A natureza *black box*[6] de muitos modelos de *deep learning* dificulta o trabalho dos utilizadores e reguladores de entenderem como as decisões são feitas, o que é fundamental para estabelecer a confiança e permitir a responsabilização (Bender, 2021). Além disso, o uso de PLN em contextos de vigilância levanta questões sobre a liberdade e os direitos civis. O potencial para abuso dessas tecnologias por governos ou empresas privadas requer uma consideração cuidadosa e diretrizes éticas claras. (Crawford, 2021) Finalmente, a questão da equidade no acesso e na representação dentro das tecnologias de PLN destaca a necessidade de garantir que essas tecnologias sirvam a todos de maneira justa, sem excluir ou marginalizar grupos linguísticos ou culturais. Enfrentar esses desafios requer um esforço conjunto de pesquisa, desenvolvimento, regulação e a sociedade como um todo, para garantir que as tecnologias de PLN sejam desenvolvidas e utilizadas de maneira ética, responsável e justa.

2.7 Tendências e futuro

Na data de escrita da dissertação (2024), o campo passa por uma evolução notável, impulsionado pelo aumento exponencial de dados linguísticos desestruturados provenientes de plataformas digitais e redes sociais. Essa transformação não apenas desafia os limites anteriores do PLN, mas também redefine as possibilidades dentro de diversos setores. Uma tendência dominante no PLN é a crescente sofisticação dos assistentes virtuais. Além de servirem como companheiros úteis em dispositivos, tornam-se componentes cruciais para melhorar a acessibilidade e fornecer informações sob demanda. Empresas aproveitam este facto para desenvolver assistentes virtuais que minimizam erros de processamento e oferecem assistência contínua e confiável, abrindo novas frentes desde o auxílio a trabalhadores industriais até facilidade da pesquisa acadêmica. Outra área em ascensão é a análise de sentimentos. Com a digitalização massiva da comunicação, torna-se essencial discernir as nuances emocionais dos dados textuais. Iniciativas estão a ser

[6] Sistema cujo funcionamento interno não é visível ou compreensível para o utilizador

tomadas para desenvolver modelos de PLN que possam capturar e interpretar as nuances emocionais dos dados, com o objetivo de melhorar a fidelização e retenção de clientes ao criar experiências que ressoam emocionalmente. Além disso, a globalização demanda uma abordagem mais inclusiva do PLN. Os modelos de linguagem estão a quebras as barreiras linguísticas tradicionais, permitindo que os processadores de linguagem compreendam e traduzam efetivamente entre as mais de 7.000 línguas faladas no mundo. Isso não apenas melhora a acessibilidade dos dados, mas também acelera os fluxos de trabalho de tradução, ampliando o alcance global das marcas. O NER é destacado pela sua capacidade de navegar pela complexidade dos dados não estruturados. Ao classificar e anotar dados de maneira eficiente, o NER facilita fluxos de trabalho de extração de dados em vários campos, tornando-se uma ferramenta indispensável na era da informação. A inovação não para por aí. Os transformadores de linguagem superam as limitações dos modelos tradicionais de PLN, proporcionando uma compreensão mais profunda e contextualizada do texto. Além disso, a aprendizagem por transferência está a revolucionar a maneira como os modelos de PLN são treinados, reduzindo significativamente o tempo e o custo necessários para desenvolver novos modelos eficazes.

2.8 Síntese

Neste capítulo, foi apresentada uma introdução aos conceitos fundamentais de PLN e ML, explicando como essas áreas interagem para possibilitar a criação de sistemas capazes de compreender e processar linguagem humana. Foram exploradas as principais técnicas e abordagens, desde a construção de modelos de linguagem até ao uso de transformadores, destacando a sua importância no avanço do PLN. A análise linguística, que inclui processos como POS *Tagging*, análise sintática e NERfoi discutida como uma componente chave na extração de significado dos textos. Foram também abordadas as métricas de avaliação, como BLEU, ROUGE, Perplexity e outras, que permitem medir a eficácia e precisão dos modelos aplicados a diversas tarefas de PLN. Por fim, o capítulo também abordou os desafios éticos que emergem da utilização de técnicas de ML e PLN, como a privacidade dos dados, o viés nos algoritmos e a necessidade de maior transparência nos modelos. Estes desafios são críticos para garantir o uso responsável da inteligência artificial no tratamento de dados linguísticos.

3. Testes Exploratórios

Na área do PLN tem-se verificado um avanço significativo nos últimos anos, impulsionando uma combinação de progressos em algoritmos de ML, poder computacional e disponibilidade de um grande volume de dados. Com esta evolução foi facilitado o desenvolvimento de ferramentas de PLN capazes de realizar tarefas complexas com eficiência e precisão cada vez maiores. Entre as ferramentas mais reconhecidas e amplamente utilizadas estão o SpaCy, NLTK e HFT, cada uma com funcionalidades projetadas para atender diferentes necessidades em PLN. O spaCy, conhecido pela sua eficiência e facilidade de uso, é uma biblioteca que se destaca pela sua capacidade de desempenhar tarefas de PLN de alta complexidade, como tokenização, lematização, análise sintática e NER, de maneira rápida e intuitiva. Por outro lado, o NLTK é uma plataforma que oferece um vasto conjunto de ferramentas para o processamento de linguagem, análise textual e prototipagem de pesquisa, incluindo suporte para diversas tarefas de PLN, desde a extração de informações até a classificação de texto. Seguindo o NLTK, a HFT apresenta-se como uma plataforma revolucionária no campo do PLN, oferecendo acesso a uma ampla gama de modelos de *transformers* pré-treinados. Estes modelos, como BERT, GPT-2, T5, entre outros, são baseados em arquiteturas de rede neural avançadas que são especialmente eficazes na compreensão do contexto e na geração de representações linguísticas profundas (mais informação no capítulo 2). A HFT é particularmente notável pelo seu desempenho em tarefas que exigem um entendimento contextual profundo, como o NER, onde o *fine-tuning* permite uma adaptação precisa dos modelos às necessidades específicas do texto em análise.

Este capítulo foca-se especificamente na exploração e comparação das capacidades NER proporcionadas pelo spaCy, NLTK e HFT. A funcionalidade de NER é fundamental no PLN, pois permite a identificação automática de entidades importantes, como nomes de pessoas, organizações, locais e datas, em um corpus de texto. Ao explorar como o spaCy e o NLTK implementam e executam a tarefa de NER, e como a HFT eleva esta tarefa a um nível superior com seus modelos avançados, será realizada uma análise comparativa baseada nas limitações e casos de uso ideais de cada ferramenta. Esta análise permitirá compreender melhor as nuances de cada biblioteca, fornecendo *insights* valiosos sobre a seleção da ferramenta mais adequada para a solução final do projeto.

3.1 Exploração Spacy

Nesta secção, será explorada a configuração e o funcionamento da biblioteca spaCy, uma das ferramentas mais robustas e eficientes para o Processamento de Linguagem Natural (PLN). O spaCy oferece um conjunto de funcionalidades que permitem a análise de grandes volumes de texto de forma rápida e precisa, tornando-o ideal para tarefas como o Reconhecimento de Entidades Nomeadas (NER), a análise sintática e o processamento de dados linguísticos complexos.

Ao longo desta secção, serão detalhados os passos necessários para a configuração da biblioteca, desde a instalação dos modelos linguísticos até à utilização das principais funcionalidades. Serão também abordados aspetos importantes como a customização de pipelines, a criação de novos modelos de NER, e a adaptação de modelos pré-treinados para português europeu. Este estudo permitirá compreender como o spaCy pode ser utilizado para maximizar a eficiência e precisão no processamento de textos, apresentando um equilíbrio entre flexibilidade e desempenho.

a) Pré-requisitos e teste de funcionalidade NER

Para garantir o correto funcionamento das funcionalidades de PLN no projeto, é necessário instalar a biblioteca spaCy e descarregar o modelo pré-treinado en_core_web_sm. Os comandos apresentados na Figura 1 permitem a instalação da biblioteca e a configuração do modelo de linguagem em inglês, que será utilizado para realizar análises linguísticas no texto.

```
1. pip3 install spacy
2. python3 -m spacy download en_core_web_sm
```

Figura 1- Comandos de instalação

O script apresentado na figura 2 é um exemplo prático de como a biblioteca spaCy pode ser utilizada para o reconhecimento de entidades específicas, neste caso, nomes de pessoas, num texto. Através de um modelo linguístico pré-treinado, o spaCy consegue analisar a estrutura linguística do texto e extrair informações relevantes, neste contexto, entidades que correspondem a pessoas. Inicialmente, o script carrega um modelo de linguagem do spaCy. Este modelo, en_core_web_sm, é uma versão compactada, otimizada para o inglês, que inclui ferramentas para tokenização (divisão do texto em palavras e frases), *tagging* (atribuição de categorias gramaticais às palavras), análise sintática e reconhecimento de entidades nomeadas (NER).

```
1. import spacy
2. PLN = spacy.load("en_core_web_sm")
3. sentence = "Alice and Bob are going to New York."
4. doc = PLN(sentence)
5. for ent in doc.ents:
6.     if ent.label_ == "PERSON":
7.         print(f"Found a name: {ent.text}")
8.
```

```
1. OUTPUT:
2. Found a name: Alice
3. Found a name: Bob
```

Figura 2- Exemplo prático NER Spacy – Modelo PT – Frase ING

Após o carregamento do modelo, o script, apresentado na figura 2, processa uma frase específica. Esta etapa é fundamental, pois converte o texto bruto numa estrutura que pode ser analisada pelo spaCy. A frase em questão, *"Alice and Bob are going to New York.",* é um exemplo que permite demonstrar o reconhecimento de entidades como nomes de pessoas. A parte central do script é onde ocorre o reconhecimento de entidades nomeadas. O spaCy, através do modelo carregado, identifica na frase as entidades que correspondem a pessoas. Esta identificação baseia-se em padrões aprendidos pelo modelo durante o treino em grandes volumes de texto. O script, então, itera sobre as entidades detetadas, verificando se são do tipo "PERSON" (pessoa), e imprime os nomes encontrados. Este processo ilustra a capacidade do spaCy em entender e extrair informações significativas de dados de linguagem natural. Essa habilidade é extremamente valiosa em diversas aplicações práticas, como extração de informação, análise de dados, categorização de conteúdo, entre outros. O uso de um modelo pré-treinado como o en_core_web_sm destaca como padrões complexos de linguagem podem ser analisados com uma configuração mínima, aproveitando as ferramentas robustas e eficientes de PLN oferecidas pelo spaCy. Tal abordagem é altamente benéfica em cenários onde uma análise de texto rápida e precisa é necessária, sem a necessidade de desenvolver e treinar modelos personalizados do zero. Em resumo, o apresentado script exemplifica como o spaCy, pode simplificar e potencializar a análise de linguagem natural, permitindo que os utilizadores se concentrem mais nas aplicações práticas e menos na complexidade técnica subjacente ao processamento de linguagem.

3.1.1 Implementação de modelo de língua portuguesa

Nesta secção, será realizada uma investigação sobre as capacidades de reconhecimento do spaCy ao mudar de um modelo treinado em inglês para um em português, enquanto analisamos a mesma frase em inglês. Especificamente, será utilizada a frase *"Alice and Bob are going to New York."* como frase de teste, tal como no exemplo anterior. O objetivo desta análise é explorar como a

mudança da língua do modelo afeta o funcionamento do spaCy na identificação e classificação de entidades nomeadas, especialmente considerando que a frase em questão permanecerá em inglês. Esta experiência com a troca de modelos linguísticos spaCy, revela uma limitação significativa bastante ilustrativa das complexidades no campo PLN. Esta limitação, que se manifesta na ausência de resultados ao processar a frase "*Alice and Bob are going to New York.*" com o modelo pt_core_news_sm, destaca a importância crucial da congruência linguística entre a língua do modelo de PLN e a língua do texto analisado. Os modelos do spaCy, como muitos outros modelos de PLN, são treinados especificamente em *corpora* de línguas particulares. Cada modelo é afinado para entender as nuances, estruturas gramaticais, vocabulário de um idioma específico. Consequentemente, quando um modelo treinado para português é utilizado para processar um texto em inglês, ele enfrenta dificuldades significativas. Isto ocorre porque as regras gramaticais, a composição lexical e a sintaxe do inglês diferem substancialmente do português. No contexto do reconhecimento de entidades nomeadas (NER), essa limitação torna-se ainda mais evidente. O modelo português do spaCy está programado para identificar padrões e nomes que são tipicamente encontrados em textos em português. Quando confrontado com um texto em inglês, o modelo falha em reconhecer as entidades, não porque as entidades não estejam presentes, mas porque as características linguísticas do texto não correspondem àquelas para as quais o modelo foi otimizado.

A ausência de resultados no script apresentado na figura 3 não é apenas uma peculiaridade, mas sim uma indicação clara da necessidade de alinhamento linguístico no PLN. Esta limitação tem implicações práticas significativas. Ela sugere que, para a análise eficaz de textos, é fundamental a seleção cuidadosa de um modelo linguístico que corresponda ao idioma do texto em questão. Além disso, revela que, apesar dos avanços no PLN, a flexibilidade entre diferentes línguas continua a ser um desafio significativo.

```
1. import spacy
2. PLN = spacy.load("pt_core_news_sm")
3. sentence = "Alice and Bob are going to New York."
4. doc = PLN(sentence)
5. for ent in doc.ents:
6.     if ent.label_ == "PERSON":
7.         print(f"Found a name: {ent.text}")
```

Figura 3 - Réplica de exemplo prático NER com modelo PT

Diante da questão sobre a consistência das etiquetas (*labels*) de entidades entre os modelos de diferentes línguas no spaCy, surgiu a necessidade de um estudo mais aprofundado. Para investigar se a mesma etiqueta usada num modelo inglês pode ser aplicada de forma equivalente num

modelo em português, foi desenvolvido e executado um script específico. Este script, apresentando na figura 4, foi elaborado não apenas para identificar as etiquetas correspondentes em cada modelo, mas também para fornecer *insights* sobre as nuances de processamento entre os dois modelos.

```
1. import spacy
2. PLN = spacy.load("pt_core_news_sm")
3. text = """O Primeiro-Ministro António Costa encontrou-se com o Presidente Marcelo
Rebelo de Sousa em Lisboa. Discutiram a resposta de Portugal à pandemia de COVID-19 e
as estratégias para a recuperação económica."""
4. doc = PLN(text)
5. for ent in doc.ents:
6.     print(f"{ent.text} ({ent.label_})")
```

Figura 4 - Exemplo prático NER – Modelo PT

```
OUTPUT:
António Costa (PER)
Marcelo Rebelo de Sousa (PER)
Lisboa (LOC)
Portugal (LOC)
COVID-19 (MISC)
```

Figura 5 - Exemplo prático NER – Modelo PT – Resultados

Em conclusão, este teste no spaCy serve como um lembrete de que, no PLN, a correspondência entre o modelo de língua e a língua do texto é crucial. Enquanto o campo de PLN procura avançar e superar essa barreira linguística, esta continua a ser um dos objetivos centrais, essencial para tornar o processamento de linguagem natural uma ferramenta verdadeiramente global e acessível.

3.1.2 Confirmação de resultados predefinidos

No segundo exemplo, o foco desloca-se do reconhecimento genérico de entidades para a identificação de termos específicos predefinidos numa frase. Esta abordagem é particularmente útil em cenários onde o objetivo é localizar ocorrências de palavras ou frases exatas, o que pode ser relevante em tarefas como análise de sentimentos ou mesmo na identificação de referências específicas em documentos.

Na figura 6, é apresentado um script onde o texto "*Alice and Bob are going to New York*." é convertido num objeto "Doc", que contém uma série de "*tokens*" (palavras) e entidades (grupos de palavras com significado conjunto).

```
1. import spacy
2. PLN = spacy.load("en_core_web_sm")
3. sentence = "Alice and Bob are going to New York."
4. doc = PLN(sentence)
5. words_to_find = ["Alice", "Bob", "New York"]
6. for token in doc:
7.     if token.text in words_to_find:
8.         print(f"Found the word: {token.text}")
9. for ent in doc.ents:
10.     if ent.text in words_to_find:
11.         print(f"Found the entity: {ent.text}")
```

Figura 6 - Reconhecimento de entidades pré-definidas

```
1.  OUTPUT:
2. Found the word: Alice
3. Found the word: Bob
4. Found the entity: Alice
5. Found the entity: Bob
6. Found the entity: New York
```

Figura 7 - Reconhecimento de entidades pré-definidas – Resultados

A diferença no script apresentado na Figura 6 reside na forma como a pesquisa é conduzida. Primeiro, define-se uma lista de palavras ou frases específicas a serem encontradas - neste caso, "Alice", "Bob" e "New York". Posteriormente, o script percorre cada *token* do objeto Doc e verifica se o texto do *token* corresponde a algum elemento da lista predefinida. Este método permite uma pesquisa direta e precisa de termos específicos no texto. Além disso, o script também faz uma verificação nas entidades reconhecidas pelo spaCy. Isto é particularmente importante para identificar frases ou nomes compostos, como "New York", que podem não ser capturados na iteração *token* a *token* devido à sua natureza multipalavra. Este método de pesquisa específico de palavras e entidades amplia significativamente as potencialidades do processamento de linguagem natural. Enquanto o reconhecimento de entidades oferece uma visão ampla e genérica do texto, a pesquisa por termos específicos permite uma análise mais direcionada e focada. Esta abordagem é especialmente valiosa quando se tem um objetivo claro de análise de termos definidos. A combinação destas duas técnicas - reconhecimento de entidades e pesquisa específica demonstra a versatilidade do spaCy. Permitindo aos utilizadores adaptar a análise de texto às suas necessidades específicas, seja para uma exploração abrangente do conteúdo ou para um foco mais restrito de termos selecionados.

3.1.3 Integração de Regex com spaCy

Uma implementação típica envolveria utilizar o spaCy para a análise geral do texto e a estruturação de dados (como a tokenização e a identificação de entidades nomeadas), e o módulo

re do Python para aplicar expressões regulares em busca de padrões específicos. Por exemplo, poderíamos usar regex para identificar formatos específicos de dados, como datas, números de telefone, endereços de e-mail ou outros identificadores únicos, enquanto utilizamos o spaCy para compreender o contexto linguístico mais amplo em que esses padrões aparecem. A integração de expressões regulares com spaCy representa uma sinergia poderosa no campo de PLN, permitindo uma análise de texto mais refinada e adaptada a necessidades específicas. Com esta combinação, é possível ir além das capacidades padrão de PLN, abordando requisitos de análise de texto com uma precisão e especificidade que seriam desafiadores de alcançar com qualquer uma das ferramentas isoladamente. Assim, a união de spaCy e regex, tal como apresentado na figura 8, é um exemplo claro de como diferentes tecnologias podem ser combinadas para ampliar os horizontes da análise de dados e extração de informações.

```
 1. import re
 2. import spacy
 3.
 4. PLN = spacy.load("en_core_web_sm")
 5. def find_personal_data(text):
 6.     personal_data = {}
 7.     phone_pattern = r"\b\d{3}[-.]?\d{3}[-.]?\d{4}\b"
 8.     email_pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b"
 9.     personal_data["phone_numbers"] = re.findall(phone_pattern, text)
10.     personal_data["emails"] = re.findall(email_pattern, text)
11.
12.     doc = PLN(text)
13.     personal_data["names"] = [ent.text for ent in doc.ents if ent.label_ ==
"PERSON"]
14.     return personal_data
15. text = "Contact Alice at alice@example.com or 123-456-7890. Bob's number is 987-
654-3210."
16. personal_data_found = find_personal_data(text)
17. print(personal_data_found)
18. OUTPUT:
19. {'phone_numbers': ['123-456-7890', '987-654-3210'],
20. 'emails': ['alice@example.com'], 'names': ['Bob']}
```

Figura 8 - Reconhecimento de entidades pré-definidas

Com o script apresentado na figura 8 é retornado um dicionário com os dados pessoais encontrados, categorizados em números de telefone, emails e nomes. Esta abordagem híbrida é particularmente valiosa em aplicações onde a identificação precisa de informações específicas é crucial, como na proteção de dados, conformidade regulatória, e na análise de textos sensíveis. Portanto, o script não só demonstra a aplicabilidade prática desta combinação tecnológica, mas também sublinha a importância de abordagens multifacetadas no campo do processamento de linguagem natural em conjunto com o spaCy.

3.1.4 Treino de modelo/*pipeline*

Nesta subsecção, será apresentado o processo de treino de modelos e *pipeline* utilizando a biblioteca spaCy. Serão explorados os principais componentes de um *pipeline*, incluindo o modelo base, os componentes padrão e a forma como estes podem ser ajustados e personalizados para atender a necessidades específicas. Além disso, será discutido como personalizar o *pipeline*, integrando novos componentes ou modificando os existentes para otimizar o desempenho em tarefas específicas de análise linguística.

a) Modelo

Um modelo no spaCy é uma combinação de vários componentes de PLN. Estes componentes incluem algoritmos e dados que o spaCy utiliza para entender e processar texto. São treinados em grandes conjuntos de dados de linguagem natural e podem realizar uma variedade de tarefas de PLN, como tokenização, análise morfológica, POS *Tagging*, NER e análise de dependências. O spaCy oferece modelos pré-treinados que podem ser utilizados diretamente ou personalizados através de treino adicional em dados específicos.

b) Pipeline

No spaCy, uma pipeline refere-se a uma sequência de etapas de processamento pelas quais o texto é submetido. Cada etapa corresponde a um componente específico da pipeline, responsável por executar uma tarefa particular de PLN, como tokenização, reconhecimento de entidades nomeadas, entre outros. Quando um texto é processado pelo spaCy, ele percorre automaticamente cada um desses componentes, passando por todas as etapas da pipeline de forma sequencial, garantindo um processamento completo e eficiente do texto.

c) Componentes

Os principais componentes de um pipeline de PLN são essenciais para processar e interpretar o texto de forma eficiente. O *Tokenizer* divide o texto em pequenas unidades chamadas *tokens*, como palavras ou frases, permitindo que o restante do pipeline trabalhe com esses elementos individuais. O *Tagger* atribui categorias gramaticais, como substantivos ou verbos, a cada token, facilitando a análise linguística.

O *Parser* analisa a estrutura gramatical da frase, definindo as relações entre os *tokens* e ajudando a compreender a sintaxe do texto. O NER identifica e classifica entidades importantes, como nomes de pessoas, empresas ou datas, extraindo informações valiosas do texto.

Por fim, os componentes adicionais podem ser personalizados para incluir regras específicas de análise de texto ou processadores de dados adaptados às necessidades do projeto, tornando o pipeline mais flexível e ajustado ao contexto em que é aplicado.

d) Personalização

Adicionar ou Remover Componentes: É possível personalizar o pipeline adicionando ou removendo componentes, dependendo das necessidades específicas da tarefa de PLN. Treino personalizado: Componentes como NER podem ser treinados em dados específicos para reconhecer entidades personalizadas.

Em resumo, o modelo no spaCy é um conjunto de algoritmos e dados treinados para entender e processar texto, enquanto o pipeline é a sequência de etapas de processamento que o texto percorre dentro do modelo. A combinação de modelos robustos e pipelines flexíveis faz do spaCy uma ferramenta poderosa e versátil para diversas aplicações de PLN, desde tarefas básicas de tokenização até análises complexas de estrutura gramatical e reconhecimento de entidades.

O script abaixo, apresentado na figura 9, é um exemplo ilustrativo de como personalizar e aprimorar modelos de PLN usando o spaCy. O treino de modelos para reconhecer entidades específicas tem implicações significativas em diversas áreas, desde a análise de sentimentos em redes sociais até a extração de informações em documentos que contenham informações pessoais. Além disso, o processo de treino personalizado permite que o modelo seja adaptado a linguagens e terminologias específicas de domínios, o que é essencial em campos com linguagem especializada, como medicina, direito ou finanças. Este script, portanto, não apenas demonstra uma aplicação técnica de PLN, mas também sublinha a flexibilidade e adaptabilidade do spaCy em responder a necessidades específicas de processamento de linguagem, enfatizando o poder dos modelos de PLN customizados na era da informação e do *big data*.

```
1. import spacy
2. import random
3. from spacy.training import Example
4.
5. PLN = spacy.load("en_core_web_sm")
6.
7. TRAIN_DATA = [
8.    ("My name is Alice.", {'entities': [(11, 16, 'PERSON')]}),
9.    ("Contact me at alice@example.com", {'entities': [(14, 31, 'EMAIL')]}),
10. ]
11.
12. ner = PLN.get_pipe("ner")
13.
14. for _, annotations in TRAIN_DATA:
15.     for ent in annotations.get("entities"):
16.         ner.add_label(ent[2])
17.
18. other_pipes = [pipe for pipe in PLN.pipe_names if pipe != "ner"]
19.
20. with PLN.disable_pipes(*other_pipes):
21.     optimizer = PLN.resume_training()
22.     for itn in range(10):
23.         random.shuffle(TRAIN_DATA)
24.         losses = {}
25.         for text, annotations in TRAIN_DATA:
26.             doc = PLN.make_doc(text)
27.             example = Example.from_dict(doc, annotations)
28.             PLN.update([example], drop=0.5, sgd=optimizer, losses=losses)
29.         print("Losses at iteration", itn, ":", losses)
30.
31. ner.to_disk("my_model")
32.
```

Figura 9 - Treino de pipeline

O script acede ao componente de reconhecimento de entidades nomeadas (NER) do modelo carregado (apresentado na figura 10). É neste componente que a atualização e o treino ocorrerão.

```
1. ner = PLN.get_pipe("ner")
```

Figura 10 - Implementação NER pipeline

Para treinar o modelo com novas entidades, é necessário adicionar os rótulos dessas entidades ao pipeline de NER (apresentado na figura 11). Este passo amplia a capacidade do modelo de reconhecer diferentes tipos de entidades.

```
1.   for _, annotations in TRAIN_DATA:
2.      for ent in annotations.get("entities"):
3.          ner.add_label(ent[2])
```

Figura 11 - Implementação de ciclo de treino

3.1.5 Avaliação de desempenho

Após cada iteração o script imprime as perdas (*losses*), apresentado na figura 4 , que são uma medida de quão bem o modelo se ajustou aos dados de treino. Um valor de perda decrescente ao longo das iterações é um indicador de que o modelo está a aprender efetivamente. Os valores de perda são uma medida numérica que indica o quão distante as previsões do modelo estão dos resultados reais (ou seja, os dados de treino). Em termos simples, o valor de perda reflete o erro do modelo. (Goodfellow, 2016)

```
1. print ("Losses at iteration", itn, ":", losses)
```

Figura 12 - Impressão de iteração de perdas

Ao longo das iterações de treino, são observados os valores de perda (apresentados na figura 13). Uma tendência decrescente nesses valores é um bom sinal. A perda é uma métrica que quantifica o erro produzido pelo modelo. Um valor alto de perda indica que o modelo está a gerar output errado, enquanto um valor baixo sugere menos erros. A função de perda usada para calcular esse valor pode variar dependendo do problema e dos dados utilizados. Essas funções de perda têm abordagens distintas para penalizar os erros feitos pelo modelo.

```
 1. Losses at iteration 0 : {'ner': 1.5692873716756628}
 2. Losses at iteration 1 : {'ner': 0.788388951018095}
 3. Losses at iteration 2 : {'ner': 0.12909892605521414}
 4. Losses at iteration 3 : {'ner': 0.025808161244922648}
 5. Losses at iteration 4 : {'ner': 6.956181657513162e-05}
 6. Losses at iteration 5 : {'ner': 0.000128306671969304}
 7. Losses at iteration 6 : {'ner': 0.24835594140133316}
 8. Losses at iteration 7 : {'ner': 0.0068912063931545935}
 9. Losses at iteration 8 : {'ner': 0.29201495166224395}
10. Losses at iteration 9 : {'ner': 2.554182953739528e-06}
```

Figura 13 - Resultados de iteração

As flutuações observadas durante as iterações de treino, são um fenômeno comum que pode ser atribuído a várias causas fundamentais. Uma das principais causas dessas flutuações é a natureza estocástica dos algoritmos de otimização utilizados, como o Gradiente Descendente Estocástico (SGD). (Bottou, 2010) Esses algoritmos atualizam os parâmetros do modelo de maneira incremental, baseando-se numa amostra aleatória dos dados em cada iteração. Essa abordagem introduz uma diferença inerente no processo de treino, já que diferentes amostras podem levar a atualizações distintas nos parâmetros, resultando em flutuações na métrica de perda. A presença

de *outliers*[7] pode levar a picos temporários na perda, refletindo o desafio que o modelo enfrenta ao tentar ajustar-se a esses pontos de dados atípicos. Outra fonte de flutuação está relacionada à complexidade do modelo e ao ajuste de seus Hiper parâmetros. (Goodfellow, 2016) Modelos com uma grande quantidade de parâmetros ou aqueles cuja configuração não é ideal para o conjunto de dados em questão podem experimentar flutuações mais acentuadas durante o treino. Isso pode ocorrer à medida que o modelo luta para equilibrar a aprendizagem dos padrões subjacentes nos dados com a prevenção de ajuste a peculiaridades específicas de um subconjunto de treino. Concluindo que a maneira como os dados são preparados e apresentados ao modelo influencia. Por exemplo, uma ordem de apresentação dos dados que não promove uma aprendizagem uniforme ou que não representa adequadamente a distribuição geral dos dados pode levar a flutuações indesejadas.

3.1.6 Carregamento e utilização de modelo treinado

Na figura 14 é apresentado o texto de input que será utilizado para os testes do modelo personalizado.

```
1. INPUT:
2. Please don't hesitate to reach out to me for any inquiries or further information.
You can contact me directly at alice@example.com. I am always available to discuss
potential opportunities or collaborations. My name is Alice, and I have extensive
experience in digital marketing, with a particular focus on social media strategy and
content creation. Over the years, I have developed a deep understanding of engaging
audiences and leveraging online platforms to maximize brand visibility. I'm looking
forward to hearing from you and exploring how we can work together to achieve your
business goals.
```

Figura 14 - Input para utilização do modelo treinado

O teste apresentado na figura 15 é crucial para avaliar a eficácia do modelo em identificar entidades específicas em textos reais. Ajuda a verificar se o modelo está a identificar corretamente as entidades para que foi treinado reconhecer. O modelo demonstrou uma precisão na identificação de entidades nomeadas específicas, como PERSON e EMAIL, nos textos analisados. Isso indica que o modelo aprendeu eficazmente os padrões e características linguísticas associadas a estas entidades.

[7] valores que se desviam significativamente da maioria dos dados num conjunto, indicando anomalias ou exceções.

```
 1. INPUT:
 2. Please don't hesitate to reach out to me for any inquiries or further information.
You can contact me directly at alice@example.com. I am always available to discuss
potential opportunities or collaborations. My name is Alice, and I have extensive
experience in digital marketing, with a particular focus on social media strategy and
content creation. Over the years, I have developed a deep understanding of engaging
audiences and leveraging online platforms to maximize brand visibility. I'm looking
forward to hearing from you and exploring how we can work together to achieve your
business goals.
 3.
 4.-------------------------------------------
 5. import spacy
 6. def identify_entities(file_path, model_path):
 7.     PLN = spacy.load(model_path)
 8.     with open(file_path, 'r') as file:
 9.         text = file.read()
10.     doc = PLN(text)
11.     for ent in doc.ents:
12.         if ent.label_ in ['PERSON', 'EMAIL']:
13.             print(f"{ent.label_}: {ent.text}")
14. file_path = 'path_to_your_text_file.txt'
15. model_path = 'my_model'
16. identify_entities(file_path, model_path)
17.
18.  -------------------------------------
19. OUTPUT:
20. EMAIL: alice@example.com
21. PERSON: Alice
```

Figura 15 - Utilização de modelo treinado

3.1.7 Fatores influenciáveis na precisão do modelo

Ao analisar a influência do tamanho do conjunto de dados de treino e do número de iterações no desempenho de modelos de NER usando o spaCy, é essencial compreender como esses fatores impactam a aprendizagem e a generalização do modelo. Um conjunto de dados de treino maior e mais diversificado tende a fornecer ao modelo uma variedade mais ampla de exemplos linguísticos e contextos, o que é crucial para uma aprendizagem mais robusta e uma generalização eficaz. (Goodfellow, 2016) Com um conjunto de dados mais rico, o modelo tem a oportunidade de aprender padrões mais complexos e subtilezas linguísticas, reduzindo assim o risco do modelo se torna excessivamente especializado nos exemplos de treino e falha em generalizar para novos dados. Por outro lado, o número de iterações de treino é igualmente crucial. Um número insuficiente de iterações pode levar a que o modelo não aprenda eficazmente os padrões nos dados. O monitoramento da perda e da precisão durante o treino é fundamental para ajustar o número de iterações. Uma diminuição consistente na perda e um aumento na precisão são

indicativos de uma aprendizagem efetiva, enquanto um aumento na perda ou uma estagnação na precisão podem significar *overfitting*[8] (Bishop, 2006).

Além disso, usar um conjunto de dados de validação separado para avaliar o desempenho do modelo ajuda a determinar o ponto ideal de paragem de treino e a avaliar a capacidade do modelo de generalizar para dados não vistos. Encontrar a combinação correta de tamanho de conjunto de dados e número de iterações muitas vezes requer experiência. Cada conjunto de dados e tarefa de NER pode exigir uma abordagem diferente, destacando a importância de testar várias configurações para alcançar o melhor desempenho do modelo.

3.2 Exploração NLTK

Nesta secção, será explorada a configuração e o funcionamento da biblioteca NLTK uma das ferramentas mais completas e amplamente utilizadas para o Processamento de Linguagem Natural. O NLTK fornece uma vasta gama de recursos, incluindo algoritmos para processamento de texto, corpora linguísticos e ferramentas para realizar tarefas como tokenização, *stemming*, análise sintática, e NER. Serão detalhados os passos necessários para a configuração e utilização da biblioteca, bem como exemplos práticos de como o NLTK pode ser aplicado para processar e analisar textos, com especial atenção para a língua portuguesa. A secção também abordará as limitações e potencialidades do NLTK em comparação com outras bibliotecas, mostrando como a sua versatilidade e a ampla gama de funcionalidades tornam a ferramenta ideal para análises linguísticas mais académicas e experimentais.

3.2.1 Pré-requisitos e teste de funcionalidade NER

NLTK é uma biblioteca popular de tarefas PLN no Python, amplamente utilizada tanto no meio acadêmico quanto na indústria para tarefas de análise de texto. (Bird, 2009) Antes de mergulhar nas capacidades do NLTK, é importante garantir o ambiente adequadamente configurado e a biblioteca corretamente instalada. A compatibilidade com as versões do Python varia, mas o NLTK geralmente funciona bem com as versões mais recentes na data de registo do relatório. A instalação do NLTK é bastante simples e pode ser feita através do pip, gestor de pacotes do Python. Basta abrir o terminal e introduzir a linha apresentada na figura 16.

---

[8] Quando um modelo aprende demais com os dados de treino, incluindo ruídos e detalhes irrelevantes, o que prejudica seu desempenho em novos dados.

```
1. pip install nltk
2. import nltk
3. nltk.download()
```

Figura 16 - Comandos de instalação NLTK

Depois de instalar o NLTK, o próximo passo é descarregar os pacotes de dados que ele utiliza. O NLTK vem com uma variedade de *corpora* de texto, modelos de tokenização e outras ferramentas úteis para PLN. O *prompt*, apresentado na figura 17, abrirá uma janela de interface gráfica para o utilizador escolher quais pacotes deseja instalar. Para começar, é recomendável descarregar o pacote "popular", que inclui uma seleção dos pacotes mais comuns e úteis.

```
---------------------------------------------------------------------------
    d) Download   l) List    u) Update   c) Config   h) Help   q) Quit
---------------------------------------------------------------------------
Downloader> d popular
    Downloading collection 'popular'
       |
       | Downloading package cmudict to /home/admin/nltk_data...
```

Figura 17 - Descarregamento de pacotes

O script representado na figura 18, é um exemplo prático para realizar o reconhecimento de entidades. Este script destaca as etapas fundamentais do processo de NER, desde a preparação inicial até a identificação de entidades no texto, começando com a importação das funções necessárias do NLTK, incluindo “ne_chunk” para o reconhecimento de entidades, “word_tokenize” para a tokenização do texto, e “pos_tag” para *tagging*. Estas ferramentas são a base para a análise de texto no NLTK.

```
1. import nltk
2. from nltk import ne_chunk
3. from nltk.tokenize import word_tokenize
4. from nltk.tag import pos_tag
5.
11.
12. text = "Alice and Bob are going to New York"
13. tokens = word_tokenize(text)
14. tags = pos_tag(tokens)
15. entities = ne_chunk(tags)
16. for entity in entities:
17.     if hasattr(entity, 'label'):
18.         print(f"{entity.label()}: {' '.join(c[0] for c in entity)}")
19.----------------------------------
20. OUTPUT:
21. PERSON: Bob
22. GPE: New York
```

Figura 18 - Exemplo prático NER

Antes de proceder, o script comenta a necessidade de descarregar pacotes específicos do NLTK caso ainda não estejam instalados. Estes pacotes incluem 'punkt' para a tokenização, 'averaged_perceptron_tagger' para *tagging*, 'maxent_ne_chunker' para o NER, e 'words'. Estes downloads são cruciais, pois fornecem os recursos e modelos necessários para que o NLTK funcione corretamente. O script utiliza um texto de exemplo - "*Alice and Bob are going to New York*" - como *input* para a análise. Este texto é então submetido ao processo de tokenização com word_tokenize, que divide o texto em uma lista de palavras e símbolos. Esses *tokens* são posteriormente etiquetados com pos_tag, que atribui a cada *token* uma etiqueta gramatical, como substantivo, verbo, adjetivo, etc. Este passo é fundamental, pois as *tags* são usadas posteriormente para ajudar na identificação das entidades nomeadas. O coração do script é o uso de ne_chunk, que recebe os *tokens* etiquetados e identifica as entidades nomeadas, como nomes de pessoas, organizações ou locais. No NLTK, o reconhecimento de entidades é baseado em padrões e contextos aprendidos a partir de dados de treino, permitindo identificar entidades comuns em textos. Finalmente, o script itera sobre as entidades identificadas e, para cada uma que possui uma etiqueta de entidade (verificada com hasattr(entity, 'label')), imprime o tipo da entidade e o texto correspondente.

3.2.2 Falhas no reconhecimento de entidades

É importante reconhecer que o NER no NLTK se baseia num classificador pré-treinado, cuja eficácia está intimamente ligada aos dados em que foi treinado. Isso significa que, se os exemplos nos dados de treino não cobrem certos tipos de entidades ou contextos, o classificador pode falhar em identificá-los corretamente em novos textos. Além disso, a identificação de entidades no NLTK depende fortemente da precisão da tokenização e da *POS tagging*. Por exemplo, se um nome próprio não for corretamente identificado como tal na etapa de *tagging*, ele pode não ser reconhecido como uma entidade nomeada na etapa subsequente. Outra consideração é a própria natureza das entidades e do texto. Entidades que são menos comuns ou que aparecem em contextos ambíguos são mais desafiadoras de serem identificadas. Além disso, entidades que se assemelham a palavras comuns podem ser ignoradas pelo classificador. Também é de salientar que diferentes ferramentas de NER, incluindo o NLTK, têm suas próprias abordagens e algoritmos para identificar entidades, o que pode resultar em variações no desempenho.

3.2.3 Treino manual

Semelhante ao capítulo anterior de análise de funcionalidades do SpaCy, nesta secção será analisado o script abaixo (Fig.19) para tentar perceber as limitações de reconhecimento do NLTK comparativamente com as suas capacidades de encontrar palavras. A versão apresentada é semelhante ao presente na secção SpaCy, apenas adaptado a pacotes da biblioteca atual.

```
1. import nltk
2.
3. from nltk.tokenize import word_tokenize
4. text = "Alice and Bob are going to New York. "
5. words_to_find = ["Alice", "Bob", "New York"]
6. tokens = word_tokenize(text)
7. found_words = [word for word in tokens if word in words_to_find]
8. for word in found_words:
9.     print(f"Found: {word}")
10.
11. OUTPUT:
12. Found: Alice
13. Found: Bob
```

Figura 19 - Processo de treino manual NLTK

Antes de proceder à tokenização, foram definidas no script uma lista de palavras (entidades) que devem ser encontradas no texto. Essas podem ser nomes, termos técnicos, palavras-chave ou qualquer outro conjunto de palavras de interesse. No exemplo fornecido, a lista inclui "Alice", "Bob" e "Liberty". Após a preparação inicial, o script processa a *string* de exemplo através da tokenização, percorrendo cada token verificando se corresponde a qualquer uma das palavras especificadas na lista words_to_find. Finalmente, o script imprime as palavras encontradas que correspondem à lista de termos desejados. Embora o script seja bastante básico, ele pode ser adaptado ou expandido para casos de uso mais complexos, como a pesquisa por padrões ou frases específicas, ou integração em sistemas maiores de PLN.

3.2.4 Integração de Regex

Tal como apresentado na secção de exploração SpaCy, o script da figura 20 exemplifica uma maneira eficiente de combinar as funcionalidades de processamento de linguagem natural da biblioteca NLTK com a potência das expressões regulares (regex) para identificar palavras ou frases específicas num texto. A integração dessas duas ferramentas amplia significativamente as possibilidades de análise textual, permitindo uma pesquisa mais precisa e adaptável.

```
1. import nltk
2. import re
3. from nltk.tokenize import word_tokenize
4. text = "Alice and Bob are going to New York"
5. patterns = [r"\bAlice\b", r"\bBob\b", r"\New York\b"]
6. tokens = word_tokenize(text)
7. for token in tokens:
8.     for pattern in patterns:
9.         if re.search(pattern, token):
10.             print(f"Found: {token}")
11.             break
12.
13. OUTPUT:
14. Found: Alice
15. Found: Bob
```

Figura 20 - Integração de regex na implementação NLTK

O texto escolhido para análise, "*Alice and Bob are going to New York*", é um exemplo prático que contém nomes próprios e um local. O script, então, estabelece padrões regex para as palavras ou frases específicas que deseja encontrar: "Alice", "Bob" e "New York". A utilização de delimitadores de palavra (\b) nos padrões regex é uma técnica importante para garantir que o script procure por correspondências exatas, evitando falsos positivos como parte de palavras maiores. Após a definição dos padrões regex, o script realiza tokenização do texto com word_tokenize do NLTK. Esta tokenização é eficaz para separar o texto em palavras individuais, o que é essencial para a pesquisa subsequente. Com o texto “tokenizado”, o script percorre cada *token* e utiliza regex para verificar se há correspondências com os padrões definidos. A função re.search é utilizada para este propósito, procurando cada padrão em cada *token*. Quando uma correspondência é encontrada, o script imprime a palavra ou frase encontrada. A inclusão da instrução break após uma correspondência bem-sucedida evita verificações desnecessárias de outros padrões no mesmo *token*.

3.2.5 Treino de modelo/pipeline

Nesta secção será apresentado o processo de treino do modelo utilizando a biblioteca NLTK. Este processo envolve a preparação dos dados de treino, a definição das *features* relevantes e a implementação do pipeline necessário para alimentar o modelo com dados linguísticos anotados.

a) Modelo

Um modelo no NLTK, semelhante a outros *frameworks* de processamento de PLN, é uma coleção de componentes e algoritmos que são utilizados para compreender e processar texto em linguagem natural. Diferentemente do spaCy, o NLTK tem um foco mais modular e oferece uma variedade

de ferramentas e técnicas que podem ser combinadas e configuradas de acordo com as necessidades específicas de cada tarefa. O NLTK inclui algoritmos para uma ampla gama de tarefas que podem ser treinados em grandes conjuntos de dados de linguagem natural para criar modelos específicos para cada tarefa. Por exemplo, para tarefas de NER, o NLTK oferece acesso a classificadores pré-treinados que podem ser utilizados para identificar entidades como nomes de pessoas, organizações e locais em textos. No entanto, um ponto importante a se destacar sobre o NLTK é sua natureza mais acadêmica e a sua flexibilidade para exploração e pesquisa em PLN. Enquanto o spaCy foca em oferecer modelos pré-treinados de alta performance e prontos para uso em aplicações de produção, o NLTK serve como uma ferramenta valiosa para ensino e pesquisa, onde a compreensão dos princípios fundamentais e a prática com diferentes técnicas e algoritmos são mais relevantes. O NLTK também permite a personalização e o treino de modelos em conjuntos de dados específicos, embora isso possa requerer um entendimento mais profundo de PLN e técnicas de *machine learning*, além de um processo mais manual e detalhado em comparação com ferramentas como o spaCy.

b) Pipeline

No NLTK, a abordagem para processar texto é diferente da usada pelo spaCy. Enquanto o spaCy utiliza uma pipeline de processamento integrada, o NLTK oferece uma coleção modular e flexível de ferramentas e algoritmos para tarefas de PLN. No NLTK, cada etapa do processamento de texto é geralmente realizada de forma independente e pode ser personalizada de acordo com as necessidades específicas da tarefa.

c) Componentes

Os principais componentes de um pipeline de PLN incluem o *Tokenizer*, que divide o texto em *tokens*, e o *Tagger*, que atribui categorias gramaticais a cada *token*. O *Parser* estrutura a frase para compreender a sua sintaxe, enquanto o NER identifica entidades como pessoas, locais e datas. Além destes, podem ser integrados componentes adicionais, como processadores de regras personalizadas, para análise mais específica conforme as necessidades do projeto

d) Personalização

- Flexibilidade**:** Uma das grandes vantagens do NLTK é a sua flexibilidade. O utilizador pode escolher quais as ferramentas e algoritmos a utilizar para cada tarefa de PLN, montando o seu próprio "pipeline" de processamento de texto.

- Treino Personalizado**:** Embora o NLTK ofereça muitos algoritmos pré-treinados, ele também permite o treino de modelos em dados específicos. Isso é especialmente útil para tarefas como POS *tagging* e NER, onde pode ser treinado um classificador num conjunto de dados que seja mais representativo do domínio específico.

O NLTK oferece uma abordagem mais granular e flexível para o processamento de texto quando comparado ao spaCy. Embora não possua uma pipeline integrada, essa flexibilidade torna o NLTK uma ferramenta poderosa para ensino, pesquisa e aplicações que requerem uma abordagem mais personalizada ou detalhada para o PLN. O script apresentado na figura 20 ilustra o processo de treino de um modelo de classificação simples, especificamente para distinguir entre *strings* de texto que são endereços de email e aquelas que não são. Este é um exemplo de como o NLTK pode ser utilizado para tarefas de classificação de texto com base em características definidas manualmente.

```
 1. nltk_data = [(extract_features(text), label) for text, label in train_data]
 2. random.shuffle(nltk_data)
 3. classifier = NaiveBayesClassifier.train(nltk_data)
 4. test_data = [
 5.     "contact@example.com",
 6.     "noemail",
 7.     "username@domain.com",
 8.     "justsometext",
 9. ]
10. for test_item in test_data:
11.     features = extract_features(test_item)
12.     print(f"{test_item}: {classifier.classify(features)}")
13. accuracy = nltk.classify.accuracy(classifier, nltk_data)
14. print(f"Model Accuracy: {accuracy}")
```

Figura 21 - Implementação NER NLTK

A precisão do modelo, tal como apresentado na figura 21, indicada como 1.0 (ou 100%), mostra que, para o conjunto de dados de treino utilizado, o modelo conseguiu classificar corretamente todas as instâncias. Isso é um indicativo de que as características escolhidas para a classificação (presença de "@", presença de ".", e o comprimento da *string*[9]) foram eficazes para as strings de teste fornecidas.

[9] Sequência de caracteres usada para representar texto em linguagens de programação.

```
1. OUTPUT:
2. contact@example.com: email
3. noemail: no_email
4. username@domain.com: email
5. justsometext: no_email
6. Model Accuracy: 1.0
```

Figura 22 - Implementação NER NLTK - Resultados

3.2.6 Exportação e utilização de modelo

Nesta secção será importado o modelo previamente treinado e colocado à prova com a mesma sample de texto utilizada para o teste de spaCy. Desta forma será possível explorar e comparar este processo de utilização entre as ferramentas e os seus devidos resultados.

Para exportação/importação do modelo, assumindo o nosso treino na utilização do algoritmo de *Naïve Bayes Classifier* será utilizada a biblioteca *pickle*. Tal como apresentado nas figuras 23 e 24 respetivamente.

```
1. import pickle
2. with open('naive_bayes_classifier.pkl', 'wb') as out_file:
3.     pickle.dump(classifier, out_file)
```

Figura 23 - Exportação de modelo treinado

```
1. import pickle
2. with open('naive_bayes_classifier.pkl', 'rb') as in_file:
3.     classifier = pickle.load(in_file)
```

Figura 24 - Importação do modelo treinado

```
1. INPUT:
2. Please don’t hesitate to reach out to me for any inquiries or further information.
You can contact me directly at alice@example.com. I am always available to discuss
potential opportunities or collaborations. My name is Alice, and I have extensive
experience in digital marketing, with a particular focus on social media strategy and
content creation. Over the years, I have developed a deep understanding of engaging
audiences and leveraging online platforms to maximize brand visibility. I’m looking
forward to hearing from you and exploring how we can work together to achieve your
business goals.
```

Figura 25 - Input para implementação de *custom model*

A função central, “extract_features”, apresentada no script da figura 26, captura duas características primordiais comuns em endereços de e-mail: a presença do caractere "@" e a existência de um ponto após o "@", ambos servindo como fortes indicadores de um possível endereço de e-mail. Essa abordagem focada permite uma análise direcionada e eficiente, reduzindo a complexidade à identificação de padrões de texto específicos dentro de grandes volumes de dados. Ao aplicar o script ao texto de exemplo, o output obtido, "EMAIL: alice@example.com", confirma o sucesso da estratégia de identificação implementada. Este resultado era o esperado, dado que o *token* analisado, "alice@example.com", apresenta claramente as características definidas pela função extract_features. A precisão na identificação e classificação deste *token* como um endereço de e-mail valida a eficácia do modelo treinado e do processo de extração de características.

```
 1. import pickle
 2. import nltk
 3. from nltk.tokenize import word_tokenize
 4.
 5. def extract_features(token):
 6.     features = {}
 7.     features['contains_at'] = '@' in token
 8.     features['contains_dot'] = '.' in token[token.find('@'):] if '@' in token else
False
 9.     return features
10. def identify_entities(text, classifier):
11.     for token in word_tokenize(text):
12.         features = extract_features(token)
13.         label = classifier.classify(features)
14.         if label == 'EMAIL':
15.             print(f"{label}: {token}")
16. with open('nltk_classifier.pkl', 'rb') as f:
17.     classifier = pickle.load(f)
18. text = input
19. identify_entities(text, classifier)
20.
```

Figura 26 - Implementação NLTK

Este desempenho alinha-se com o objetivo do script de fornecer uma solução automatizada para a identificação de endereços de e-mail em textos, destacando a importância de características bem definidas e de um classificador adequadamente treinado.

3.3 Exploração HFT

Nesta secção, será explorada a configuração e o funcionamento da *biblioteca Hugging Face Transformers* (HFT), uma das mais avançadas ferramentas para o Processamento de Linguagem Natural. HFT permite o acesso a modelos de última geração, baseados em arquiteturas de *Transformers*, como BERT, GPT, e outros modelos treinados em grandes volumes de dados. Estes modelos são amplamente utilizados para tarefas complexas de PLN, como Reconhecimento de Entidades Nomeadas, análise de sentimentos, tradução automática, entre outras.

Serão apresentados os passos necessários para configurar a biblioteca, bem como exemplos práticos de como utilizar modelos pré-treinados ou personalizar modelos específicos para português europeu. A secção também abordará a flexibilidade que HFT oferece, permitindo facilmente a troca entre diferentes modelos e a integração de novas funcionalidades. Esta exploração permitirá entender como tirar partido da potência dos modelos de *transformers* para aumentar a precisão e eficiência no processamento de texto.

3.3.1 Pré-requisitos e teste de funcionalidade NER

Os comandos apresentados na Figura 23 são essenciais para a utilização de modelos de ML e PLN. O primeiro comando instala o *TensorFlow*, uma das principais bibliotecas para *deep learning*. Os comandos seguintes instalam a biblioteca *transformers*, permitindo o uso de modelos pré-treinados da *Hugging Face*, que são amplamente utilizados para tarefas de PLN.

```
1. pip install tensorflow
2. pip install transformers
3. pip install git+https://github.com/huggingface/transformers
```

Figura 27 - Comandos de instalação HFT

O script (figura 28) abaixo foi construído para exemplificar como identificar e classificar automaticamente entidades, como nomes de pessoas, organizações e localizações, dentro de um dado texto. No início, o script importa as ferramentas necessárias da biblioteca Transformers: TFAutoModelForTokenClassification para carregar um modelo de classificação de tokens TensorFlow, AutoTokenizer para tokenizar o texto de acordo com o formato esperado pelo modelo, e pipeline para criar uma sequência de processamento de NER de alto nível, que abstrai as etapas de tokenização, aplicação do modelo e interpretação dos resultados. O modelo utilizado,

identificado por dslim/bert-base-NER, é uma versão do BERT, um modelo de linguagem de última geração, ajustado especificamente para a tarefa de NER.

```
1. from transformers import TFAutoModelForTokenClassification, AutoTokenizer, pipeline
2.
3. model_name = "dslim/bert-base-NER"
4. model = TFAutoModelForTokenClassification.from_pretrained(model_name)
5. tokenizer = AutoTokenizer.from_pretrained(model_name)
6. PLN = pipeline("ner", model=model, tokenizer=tokenizer)
7. example_text = "Sarah Johnson works at Apple in California."
8. ner_results = PLN(example_text)
9. for result in ner_results:
10.     print(f"Word: {result['word']} - Entity: {result['entity']} - Score:
{result['score']:.2f}")
11.
```

Figura 28 - Implementação NER com modelo importado

Após carregar o modelo e o tokenizador correspondente, o script configura uma pipeline de NER, que é então aplicada a um texto de exemplo: "*Sarah Johnson works at Apple in California*." Sendo que a frase contém várias entidades que o modelo é capaz de reconhecer: *'Sarah Johnson'* como pessoa, *'Apple'* como organização e 'California' como localização, tal como apresentado abaixo na figura 29.

```
1. OUTPUT:
2.
3. Word: Sarah - Entity: B-PER - Score: 1.00
4. Word: Johnson - Entity: I-PER - Score: 1.00
5. Word: Apple - Entity: B-ORG - Score: 1.00
6. Word: California - Entity: B-LOC - Score: 1.00
7.
```

Figura 29 - Implementação NER com modelo importado – resultados

3.3.2 Implementação de modelo de língua portuguesa

A alteração do modelo de linguagem para " dslim/bert-base-NER" representa uma adaptação significativa do script para processar e entender texto em português, mantendo a estrutura e objetivos. Esta mudança exemplifica a flexibilidade e a capacidade de personalização oferecidas pela biblioteca, permitindo aos utilizadores aplicar modelos de PLN avançados a idiomas específicos com facilidade. O modelo utilizado é uma versão do BERT que foi pré-treinado especificamente na língua portuguesa, considerando as suas nuances gramaticais e contextuais. Ao aplicar este modelo a um texto de exemplo, "João Silva é presidente da empresa COMPANY.", o script é capaz de identificar entidades relevantes como nomes de pessoas e organizações dentro do contexto da língua portuguesa. Isso demonstra a capacidade do modelo de reconhecer e classificar entidades de maneira precisa, fornecendo uma pontuação de confiança

para cada entidade identificada. Essas pontuações oferecem insights valiosos sobre a certeza do modelo em relação às suas classificações, permitindo aos utilizadores avaliar a confiabilidade dos resultados de NER.

```
 1. from transformers import BertTokenizer, BertForTokenClassification, pipeline
 2. model_name = "dslim/bert-base-NER"
 3. model = BertForTokenClassification.from_pretrained(model_name)
 4. tokenizer = BertTokenizer.from_pretrained(model_name)
 5. PLN = pipeline("ner", model=model, tokenizer=tokenizer)
 6. example_text = "João Silva é presidente da empresa COMPANY."
 7. ner_results = PLN(example_text)
 8. for result in ner_results:
 9.     print(f"Word: {result['word']} - Entity: {result['entity']} - Score:
{result['score']:.2f}")
10.
---------------------------------------------
OUTPUT
Word: João - Entity: B-PER - Score: 1.00
11. Word: Silva - Entity: I-PER - Score: 1.00
12. Word: ##p - Entity: I-ORG - Score: 0.54
13. Word: CO - Entity: I-ORG - Score: 0.55
```

Figura 30 - Implementação NER com modelo importado – resultados

No *output* do script (Figura 30), "João" é reconhecido como o início do nome de uma pessoa (B-PER significa "Início de Pessoa") e score de 1.00 sugere que o modelo tem alta confiança nesta previsão. "Silva" é reconhecido como parte do nome de uma pessoa (I-PER significa "Dentro de Pessoa"). O score de 1.00 sugere alta confiança. "##p" é reconhecido como parte do nome de uma organização (I-ORG significa "Dentro de Organização"). O score de 0.54 indica confiança moderada. "CO" é reconhecido como parte do nome de uma organização, continuando a partir dos *tokens* anteriores. O score de 0.55 sugere confiança moderada. "##MP" também é reconhecido como parte do nome de uma organização, continuando a partir dos *tokens* anteriores. O score de 0.47 indica confiança mais baixa em comparação com as outras previsões.

Em resumo, o modelo identifica corretamente "João Silva" como o nome de uma pessoa e identifica os *tokens* subsequentes como parte do nome de uma organização, embora com diferentes níveis de confiança. O prefixo "##" antes de alguns *tokens* indica que eles são partes de subpalavras.

3.3.3 Identificar entidades pré-definidas

O script abaixo (Figura 19) define então um conjunto de *target_entities*, que representam as categorias de entidades que o utilizador deseja identificar, como nomes de pessoas ou organizações. No entanto, é importante destacar que, no contexto do script, essas entidades deveriam ser representadas por índices numéricos correspondentes às classes do modelo, uma

correção necessária para alinhar o script com as práticas padrão de NER. Na etapa de previsão, o modelo é aplicado aos inputs tokenizados para identificar as entidades no texto. O script utiliza a função torch.argmax para selecionar a classe mais provável para cada *token*, convertendo os logits do modelo em previsões concretas. As previsões são então associadas aos *tokens* originais, permitindo uma análise detalhada de quais palavras ou frases foram identificadas como entidades significativas. Finalmente, o script itera sobre os resultados, exibindo as entidades reconhecidas juntamente com suas categorias e pontuações de confiança. Essa saída fornece insights claros sobre o desempenho do modelo e a natureza das entidades identificadas no texto.

```
1. from transformers import TFAutoModelForTokenClassification, AutoTokenizer
2. model_name = "neuralmind/bert-base-portuguese-cased"
3. model = TFAutoModelForTokenClassification.from_pretrained(model_name)
4. tokenizer = AutoTokenizer.from_pretrained(model_name)
5. text = "João Silva é presidente da empresa XPTO."
6. # Definir labels que queremos identificar
7. target_entities = ["PESSOA", "ORGANIZAÇÃO"]
8. inputs = tokenizer(text, return_tensors="pt")
9. outputs = model(**inputs)
10. predictions = torch.argmax(outputs.logits, dim=-1)
11. entities = tokenizer.batch_decode(predictions)
12. for entity, prediction in zip(entities, predictions):
13.     if prediction in target_entities:
14.         print(f"Entity: {entity}")
```

Figura 31 - Implementação NER com entidades pré-definidas

3.3.4 Treino de modelo/pipeline

Nesta secção será abordado o processo de treino de modelos e pipelines utilizando HFT, uma das mais avançadas bibliotecas para PLN. Serão apresentados os componentes principais do treino, como a escolha e configuração do modelo, bem como a utilização de pipelines pré-treinados para várias tarefas de PLN. Além disso, será realizada uma comparação com o processo de treino em outras bibliotecas como spaCy e NLTK, destacando as diferenças em termos de desempenho, flexibilidade e facilidade de integração com grandes modelos de linguagem. A análise permitirá compreender as vantagens de cada abordagem e identificar a solução mais adequada para diferentes contextos de aplicação.

a) Modelo

O núcleo do HFT reside na sua capacidade de oferecer acesso a centenas de modelos baseados em transformadores, como BERT, GPT e RoBERTa. Diferentemente das abordagens anteriores

em PLN, que dependiam fortemente de recursos linguísticos manuais e representações simplificadas de texto, os modelos de transformadores aprendem representações ricas e contextualizadas diretamente dos dados. A biblioteca não apenas fornece modelos para o inglês, mas também para uma ampla gama de outros idiomas, tornando o PLN mais acessível globalmente. Além disso, o HFT permite a personalização e o ajuste fino desses modelos em conjuntos de dados específicos, possibilitando que os utilizadores adaptem os modelos pré-treinados às suas necessidades particulares, melhorando significativamente o desempenho em tarefas especializadas.

b) Pipeline

Uma das características mais notáveis do HFT é a introdução de pipelines de alto nível. Esses pipelines abstraem as complexidades subjacentes do processamento de modelos, permitindo que os utilizadores executem tarefas de PLN, como classificação de texto, geração de texto, tradução e NER, com apenas algumas linhas de código. Essa facilidade de uso democratiza o acesso a tecnologias avançadas de PLN, permitindo que mesmo aqueles com conhecimento limitado em *deep learning* possam utilizar modelos de IA de ponta nas suas tarefas. Para garantir que os textos sejam adequadamente processados pelos modelos de transformadores, o HFT oferece tokenizadores avançados que acompanham cada modelo. Esses tokenizadores cuidam de detalhes cruciais, como a divisão de texto em *tokens*, a adição de *tokens* especiais e a conversão de texto em IDs de *tokens* que os modelos podem entender. Essa padronização garante que os dados de entrada sejam consistentemente formatados e otimizados para o processamento do modelo. O HFT não é apenas uma biblioteca de software, mas também uma plataforma colaborativa. *Hugging Face* promove uma comunidade onde programadores e investigadores podem compartilhar seus próprios modelos treinados, contribuindo para o crescente ecossistema de PLN. Isso não apenas acelera o progresso no campo, mas também promove uma cultura de compartilhamento e transparência.

```
 1. from transformers import AutoModelForTokenClassification, AutoTokenizer,
TrainingArguments, Trainer
 2. import torch
 3. from datasets import load_dataset
 4.
 5. dataset_name = "your_email_dataset"
 6. task_name = "email_extraction"
 7. train_dataset = load_dataset(dataset_name, task_name, split="train")
 8. val_dataset = load_dataset(dataset_name, task_name, split="validation")
 9. tokenizer = AutoTokenizer.from_pretrained("bert-base-portuguese-cased")
10. def tokenize_function(examples):
11.     return tokenizer(examples["sentence"], padding="max_length", truncation=True)
12. train_dataset = train_dataset.map(tokenize_function, batched=True)
13. val_dataset = val_dataset.map(tokenize_function, batched=True)
14. num_labels = 2
15. model = AutoModelForTokenClassification.from_pretrained("bert-base-portuguese-
cased", num_labels=num_labels)
16. training_args = TrainingArguments(
17.     output_dir="./results",
18.     num_train_epochs=3,
19.     per_device_train_batch_size=16,
20.     per_device_eval_batch_size=16,
21.     learning_rate=2e-5,
22.     evaluation_strategy="epoch",
23. )
24. trainer = Trainer(
25.     model=model,
26.     args=training_args,
27.     train_dataset=train_dataset,
28.     eval_dataset=val_dataset,
29. )
30. # Treino
31. trainer.train()
32. results = trainer.evaluate()
33. print(f"Precisão: {results['eval_precision']}")
34. print(f"Revocação: {results['eval_recall']}")
35. print(f"F1: {results['eval_f1']}")
36.
37. model.save_pretrained("./custom_model")
```

Figura 32 - Treino de modelo customizado

```
 1. from transformers import TFAutoModelForTokenClassification, AutoTokenizer
 2. model = TFAutoModelForTokenClassification.from_pretrained("path/to/model")
 3. tokenizer = AutoTokenizer.from_pretrained("path/to/tokenizer")
 4. text = "Este é um email: joaosilva@exemplo.com. Envie outro email para
mariagomes@exemplo.com."
 5. inputs = tokenizer(text, return_tensors="pt")
 6. outputs = model(**inputs)
 7. predictions = outputs.logits.argmax(dim=-1)
 8. emails = []
 9. for i in range(len(predictions)):
10.     if predictions[i] == 1:
11.         email = tokenizer.decode(inputs["input_ids"][i])
12.         emails.append(email)
13.
14. print(f"Found Emails: {emails}")
```

Figura 33 - Importação e utilização de modelo

3.3.5 Comparação com NLTK e spaCy

Enquanto o NLTK é amplamente reconhecido como uma ferramenta educacional e de pesquisa, fornecendo uma vasta gama de algoritmos e recursos linguísticos para análise textual manual, e o spaCy é valorizado por seus pipelines de processamento de texto eficientes e focados na produção, o HFT distingue-se pelo seu foco em modelos de *transformers* pré-treinados e pela facilidade de aplicação em tarefas de PLN. Embora o spaCy também ofereça suporte a alguns modelos de *transformers*, o HFT é mais abrangente e atualizado em termos de variedade de modelos e arquiteturas disponíveis. Em conclusão, o Hugging Face Transformers redefine o panorama do PLN, oferecendo uma plataforma abrangente e de fácil utilização para a implementação de modelos de transformadores. Com seu suporte extensivo para modelos pré-treinados, pipelines de alto nível e uma comunidade ativa, o HFT está na vanguarda do desenvolvimento e aplicação de tecnologias de PLN, facilitando avanços significativos tanto em pesquisa quanto em aplicações práticas.

3.4 Ferramentas de comparação

Nesta secção, serão exploradas várias ferramentas utilizadas para a conversão de documentos em diversos formatos, como Pandoc, Soffice, Unoconv, e Calibre. Estas ferramentas desempenham um papel crucial no processamento de textos, permitindo converter documentos como PDFs, DOCX, e outros formatos comuns, em texto simples, pronto para análise e extração de dados. Cada uma destas ferramentas tem características específicas que as tornam adequadas para diferentes cenários de conversão. O Pandoc é conhecido pela sua versatilidade e suporte a uma ampla variedade de formatos, enquanto o Soffice (parte da suite LibreOffice) e o Unoconv oferecem soluções robustas para conversão de documentos de escritório. Já o Calibre, amplamente utilizado para conversão de eBooks, é ideal para trabalhar com formatos como EPUB e MOBI.

Nesta secção, será feita uma análise comparativa destas ferramentas, focando-se na sua eficiência, qualidade da conversão e compatibilidade com diferentes tipos de ficheiros. O objetivo é identificar as vantagens e limitações de cada uma, permitindo escolher a melhor solução para diferentes necessidades de conversão de texto.

3.4.1 Análise comparativa de ferramentas de conversão

No domínio do processamento de informação digital, uma das tarefas mais comuns e cruciais é a conversão de vários formatos de ficheiros numa forma padronizada para facilitar o acesso, a análise e o armazenamento. Reconhecendo a importância desta tarefa, este estudo foi iniciado com o objetivo de identificar a ferramenta mais eficiente e fiável para converter uma diversidade de extensões de ficheiros em formato .txt.

Para garantir um ambiente controlado e replicável, todo o estudo foi conduzido utilizando uma Máquina Virtual (especificações em anexo). Esta abordagem não só proporcionou um ambiente de teste consistente através de várias ferramentas de software, como também mitigou qualquer potencial interferência de variáveis externas que pudessem afetar a precisão dos resultados.

O objetivo principal era avaliar estas ferramentas de conversão de texto com base em vários critérios chave: precisão da conversão, compatibilidade com uma ampla gama de formatos de ficheiros, velocidade de processamento e fiabilidade geral. Esta avaliação é particularmente relevante em cenários onde grandes volumes de dados precisam ser padronizados em formato de texto, o que é uma exigência comum em análise de dados, processos de arquivo e gestão de conteúdo digital.

Este capítulo apresenta uma análise comparativa abrangente das ferramentas de conversão de texto testadas, oferecendo insights sobre o seu desempenho e adequação para várias aplicações no campo da gestão de informação digital.

3.4.2 Resultados de velocidade de processamento

A velocidade de processamento emergiu como um fator crítico na eficácia das ferramentas de conversão de texto. A capacidade de converter rapidamente grandes volumes de dados não só melhora a eficiência operacional, mas também permite uma resposta mais ágil a necessidades analíticas e de gestão de informação. Em contextos onde o tempo é um recurso valioso, a velocidade com que uma ferramenta consegue processar e converter dados pode ter um impacto significativo na produtividade e na tomada de decisões. Este aspeto é particularmente relevante para o estudo, onde as ferramentas são avaliadas não apenas pela sua precisão e compatibilidade, mas também pela sua habilidade em processar informações de maneira rápida e eficiente.

Foram realizadas 3 rondas de teste, utilizando o comando time para registar a velocidade de início de instrução, até ao fim do processamento. Abaixo, na figura 30, apresentam-se as sintaxes utilizadas para as ferramentas em questão:

```
1. Pandoc  : time pandoc  [SAMPLE.EXTENSION] -o [FILENAME].txt
2. Soffice :  time soffice --headless --convert-to txt --outdir  [FILENAME].txt
[SAMPLE.EXTENSION]
3. Unoconv: time unoconv –f txt –o [FILENAME].txt [SAMPLE.EXTENSION]
4. Calibre:  time ebook-convert [SAMPLE.EXTENSION] [FILENAME].txt
5.
```

Figura 34 - Comandos de conversão

3.4.3 Análise de resultados

A análise comparativa das ferramentas de conversão de texto, apresentada na tabela 1, revelou resultados notáveis. Pandoc destacou-se como a ferramenta mais rápida, demonstrando a melhor performance global. A sua capacidade de lidar com uma ampla variedade de extensões de forma eficaz posicionou-a como a opção superior na avaliação. Manteve um equilíbrio entre velocidade e qualidade, tornando-a ideal para uma vasta gama de aplicações.

- Soffice emergiu como a segunda ferramenta mais rápida, destacando-se principalmente pela sua compatibilidade abrangente. Embora não tenha superado Pandoc em termos de desempenho geral, mostrou uma notável habilidade de suportar o maior número de extensões, um fator crucial em ambientes onde a diversidade de formatos de arquivo é uma constante.
- Unoconv, por sua vez, foi a ferramenta mais rápida especificamente para a conversão de arquivos .json. Esta especialização torna-a uma escolha valiosa para contextos nos quais a conversão de dados estruturados em JSON é frequentemente necessária.

Por outro lado, Calibre, apesar de ser uma ferramenta popular e versátil em muitos aspetos, não conseguiu destacar-se em nenhuma categoria específica de extensão em nossa análise. Embora ofereça funcionalidades úteis, no contexto específico de conversão de arquivos em massa para .txt, a sua velocidade ficou aquém das outras ferramentas avaliadas.

Esta avaliação demonstra como diferentes ferramentas podem ser adequadas para diferentes necessidades e contextos. Pandoc e Soffice, com as suas fortes performances, destacam-se como escolhas sólidas para a maioria das aplicações, enquanto Unoconv oferece vantagens específicas para formatos de arquivo particulares.

	Pandoc	Soffice	Unoconv	Calibre
.docx	0m0.263s	0m1.424s	0m0.476s	0m1.024s
.pdf	-	0m5.815s	*	0m8.350s
.txt	0m0.159s	0m0.402s	0m0.309s	0m0.710s
.md	0m0.059s	0m0.416s	0m0.260s	0m0.415s
.epub	0m3.539s	*	*	0m1.194s
.html	0m1.490s	0m0.442s	0m0.748s	0m0.741s*
.csv	0m0.105s	-	*	-
.xml	0m0.550s*	0m0.942s	0m0.948s	-
.json	-	0m0.380s	0m0.324s	-
.odt	0m0.171s	0m0.689s	0m0.793s	0m0.466s

Tabela 1 - Resultados de ferramentas de conversão

3.5 Ferramentas de anotação

Aplicações para facilitar o processo de rotulação dos dados, etapa crucial no treino de modelos de ML. Elas permitem que anotadores humanos marquem dados, como textos, imagens ou vídeos, com rótulos que definem as suas características ou classificam os seus conteúdos. A anotação de texto, usada para tarefas de processamento de linguagem natural, envolve rotular partes do texto com etiquetas que identificam entidades, sentimentos, categorias ou ações. As ferramentas para anotação de texto oferecem funcionalidades como destacar frases, atribuir rótulos de entidade (por exemplo, pessoa, organização) e pontuações de sentimento. Nesta secção será feita exploração de ferramentas gratuitas que possibilitem a criação do modelo para o objetivo deste projeto.

3.5.1 Doccano

Ferramenta de anotação *open-source*[10] que oferece uma plataforma intuitiva e eficiente para a anotação de *datasets*[11] para tarefas de PLN, como NER, classificação de texto e análise de

---

[10] software cujo código-fonte é disponibilizado publicamente para uso, modificação e distribuição livre.
[11] coleções organizadas de dados usados para treino, teste ou validação de modelos

sentimentos. Projetada para facilitar o trabalho colaborativo, permite que as equipas anotem grandes conjuntos de dados de texto de maneira rápida e precisa. Com uma interface de utilizador simples e amigável, suporta a anotação de múltiplas linguagens e possui funcionalidades como importação e exportação de dados em vários formatos, gestão de utilizadores com diferentes níveis de acesso e acompanhamento do progresso da anotação. Os utilizadores podem definir as suas próprias categorias de etiquetas e aplicá-las a segmentos de texto, facilitando a preparação de dados para treino e avaliação de modelos de PLN. Ideal para experimentar e desenvolver na área de PLN, ajuda a otimizar o processo de criação de *datasets* anotados, tornando-se uma ferramenta valiosa no pipeline de desenvolvimento de modelos de AI.

Na figura 31 é apresentado como executar no *prompt*[12] puxar a imagem mais recente do Doccano do *Docker Hub*[13]:

```
1. docker pull doccano/doccano
```

Figura *31* – Docker image pull

Use o seguinte comando (fig. 32) para criar um container Docker para o Doccano. Este comando configura as credenciais de administração padrão e define a porta que será usada para aceder a interface web do Doccano:

```
1. docker container create --name doccano \ -e "ADMIN_USERNAME=admin" \ -e
"ADMIN_EMAIL=admin@example.com" \ -e "ADMIN_PASSWORD=password" \ -p 8000:8000
doccano/doccano
```

Figura 32 – Docker container - criação

Neste comando, --name doccano dá ao container o nome "doccano", ADMIN_USERNAME, ADMIN_EMAIL, e ADMIN_PASSWORD definem as credenciais do administrador, e -p 8000:8000 mapeia a porta 8000 do container para a porta 8000 do host.

```
1. docker container start doccano
```

Figura 33 – Docker container - iniciação

12 instrução ou input fornecido
13 repositório online onde utilizadores podem armazenar, compartilhar e distribuir imagens de containers Docker

Após o container ser iniciado, aceder http://localhost:8000/ , fazer login utilizando as credenciais de administração configuradas anteriormente (admin e password).

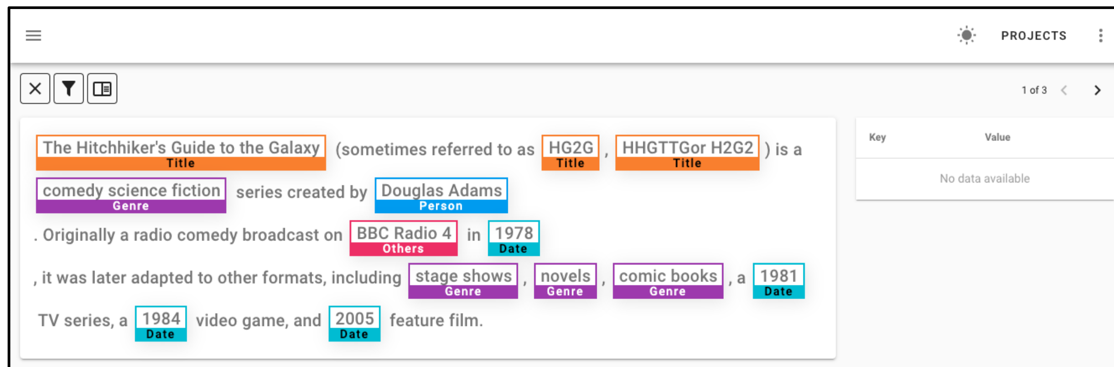

Figura 34 - Doccano project tagging board

3.5.2 Label Studio

Ferramenta *open-source* para a anotação de dados, projetada para ajudar profissionais de ML a rotular rapidamente grandes conjuntos de dados. Com suporte para vários tipos de dados, como texto, imagens, áudio e vídeo, Label Studio permite a criação de *datasets* personalizados para treino de modelos de ML. A ferramenta oferece uma interface intuitiva que facilita a definição de tarefas de anotação, configuração de esquemas de etiquetas e gerenciamento do processo de anotação. Os utilizadores podem personalizar o ambiente de anotação para atender a requisitos específicos de projeto e integrar modelos de ML para pré-anotação automatizada, acelerando o processo de anotação manual. Label Studio é ideal para projetos que exigem anotações personalizadas, oferecendo flexibilidade para trabalhar com diferentes formatos de dados e complexidades de anotação. O facto de ser *open-source* com uma comunidade ativa contribuem para uma ferramenta em constante evolução, adaptável a uma ampla gama de aplicações de anotação de dados.

a) Instalação e configuração

```
1. pip install label-studio
2. label-studio
```

Figura 35 – Comandos de instalação label-studio

O segundo comando, apresentado na figura 35, irá iniciar o servidor do Label Studio e abrirá automaticamente a interface de utilizador no browser. Na primeira execução, será solicitada a criação de uma nova conta de utilizador.

b) Utilização

Depois de entrar, é possível criar um projeto ao carregar em "Create" na página principal. É necessário fornecer um nome para o projeto e selecionar o tipo de tarefa de anotação desejado (texto, imagem, vídeo, etc.)

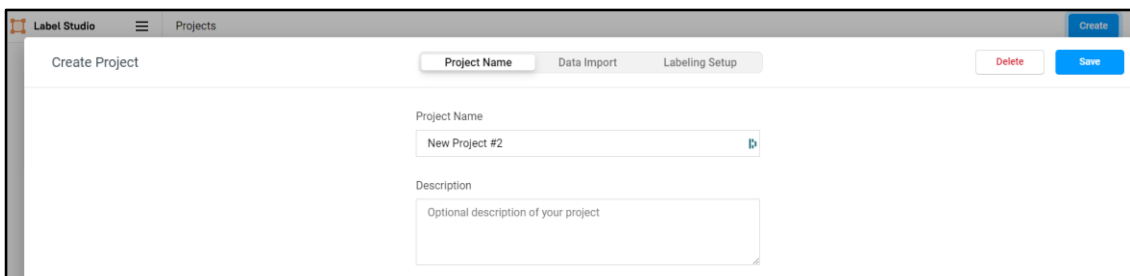

Figura 36 – Criação de projeto label-studio

c) Importação de Dados

É necessário utilizar um formato compatível como JSON, CSV ou texto simples. Para anotação de texto, um ficheiro de texto simples com cada linha a representar um documento pode ser suficiente. Na secção "Data" carregar em "Import". Selecionar ficheiro desejado

d) Registo de *labels*

Definir as etiquetas que serão utilizadas para o processo de anotação na *tab* "Labeling Setup" do projeto. Por exemplo, definir etiquetas como NAME, EMAIL, etc. Após a importação dos dados, começar a anotação de *dataset* na *tab* "Labeling".

e) Exportação de Dados Anotados

Após concluir a anotação, é possível exportar os dados anotados em diversos formatos, como JSON, CSV ou outro, dependendo das necessidades. Na interface do projeto, na tab "Data" selecionar "Export" para descarregar os dados anotados.

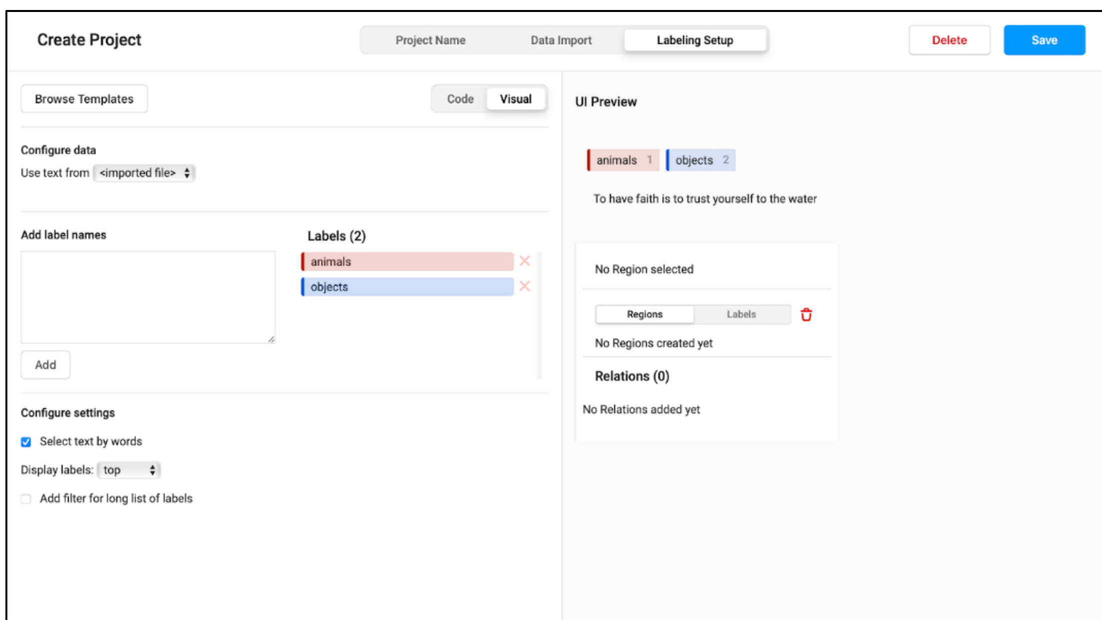

Figura 37 – Processo de configuração de labels

3.5.3 Testes de precisão

Este tipo de teste para ferramentas como Label Studio e Doccano geralmente não são realizados nas ferramentas em si, mas sim nos modelos de ML que são treinados utilizando os dados anotados produzidos por essas ferramentas.

- Label Studio: A precisão e a qualidade dos dados anotados dependem dos anotadores humanos que utilizam a ferramenta. O Label Studio oferece uma interface flexível que pode ser usada para criar dados anotados de alta qualidade. A precisão dos modelos treinados com esses dados dependerá da qualidade e da consistência das anotações, além das técnicas de aprendizado de máquina utilizadas.

- Doccano: A precisão dos dados anotados também depende dos anotadores. Doccano facilita a anotação de texto para tarefas como classificação de texto, reconhecimento de entidades nomeadas (NER) e outras tarefas de PLN. A precisão e eficácia do modelo treinado serão influenciadas pela qualidade dos dados anotados.

a) Avaliação da Precisão

Nesta subsecção serão apresentados os métodos utilizados para avaliar a precisão das ferramentas de anotação, especificamente o Doccano e o Label Studio. Serão discutidos os critérios de avaliação e comparados os resultados de ambas as ferramentas em termos de eficiência, precisão e qualidade das anotações geradas.

Testes Exploratórios

- Revisão e validação de anotações: Uma maneira de garantir a precisão dos dados anotados é através da revisão e validação cruzada entre anotadores, para garantir que as anotações sejam consistentes e precisas.

- Treino e teste de modelos: Após a anotação, os dados são utilizados para treinar modelos de ML. A precisão desses modelos é geralmente avaliada usando um conjunto de dados de teste separado, não visto pelo modelo durante o treino, e medindo métricas como precisão, sensibilidade e F1-score (Consultar capítulo 2).

- Melhorias: Com base na avaliação dos modelos, os dados anotados podem precisar de ajustes para melhorar a qualidade, o que pode envolver reanotar ou anotação adicional de mais dados.

```
João Silva, residente na Rua das Flores 123, Lisboa, contactou a nossa empresa no dia
5 de abril. Ele expressou interesse nos nossos serviços de consultoria financeira e
deixou seu número de telefone (+351) 212 345 678 para futuros contatos. Durante a
conversa, João mencionou que o seu NIF é 501 234 567. A reunião foi agendada para a
próxima quinta-feira às 15h, no escritório principal da empresa. Além disso, João
solicitou que todas as comunicações futuras fossem enviadas para o seu email pessoal,
joao.silva@example.com
```

Figura 38 - Sample para anotação

b) Resultados e Análise

Após uma avaliação comparativa das ferramentas de anotação de dados para o projeto de NER, decidi optar pelo uso do Doccano em detrimento do Label Studio. Embora o Label Studio ofereça uma configuração e instalação relativamente simples, que inicialmente parecia atraente, concluí que o Doccano proporciona uma experiência de utilizador superior, crucial para a eficiência e eficácia do projeto de anotação. Uma das razões principais para essa escolha foi a interface de utilizador, que se mostrou não apenas mais amigável, mas também esteticamente mais agradável. A clareza e a intuitividade da interface do Doccano facilitam significativamente o processo de anotação, permitindo que os anotadores se concentrem mais na tarefa sem distrações decorrentes de complicações no uso da ferramenta. Além disso, o modo de gestão de equipas do Doccano provou ser extremamente eficiente. A plataforma oferece uma forma simplificada e intuitiva de gerir equipas de anotadores, atribuir tarefas e acompanhar o progresso de cada membro. Esta funcionalidade é particularmente valiosa em projetos de grande escala, onde a coordenação e a comunicação clara entre os membros da equipe são fundamentais.

Nas Figuras 35 e 36 estão apresentados, respetivamente, os resultados das ferramentas Doccano e Label Studio. Enquanto o Doccano exibe inicialmente o texto de input na íntegra, seguido pela delimitação das entidades e as suas respetivas *labels*, o Label Studio identifica as anotações diretamente, associando-as imediatamente às suas identificações correspondentes. Esta diferença na apresentação reflete distintas abordagens de visualização das anotações, oferecendo aos utilizadores experiências ligeiramente diferentes no processo de anotação.


```
1. {"id":2,"text":"João Silva, residente na Rua das Flores 123, Lisboa, contactou a
nossa empresa no dia 5 de abril. Ele expressou interesse nos nossos serviços de
consultoria financeira e deixou seu número de telefone (+351) 212 345 678 para futuros
contatos. Durante a conversa, João mencionou que o seu NIF é 501 234 567. A reunião foi
agendada para a próxima quinta-feira às 15h, no escritório principal da empresa. Além
disso, João solicitou que todas as comunicações futuras fossem enviadas para o seu
email pessoal,
joao.silva@example.com.","label":[[0,10,"Nome"],[25,43,"Morada"],[200,218,"Telefone"],[
293,304,"NIF"],[504,526,"Email"]],"Comments":[]}
2.
```


Figura 35- Output doccano


```
[{"id":1,"annotations":[{"id":5,"completed_by":1,"result":[{"value":{"start":290,"end":
301,"text":"501 234
567","labels":["NIF"]},"id":"xljui8Q6dL","from_name":"label","to_name":"text","type":"l
abels","origin":"manual"},{"value":{"start":25,"end":51,"text":"Rua das Flores 123,
Lisboa","labels":["MORADA"]},"id":"oJ32RiMRY1","from_name":"label","to_name":"text","ty
pe":"labels","origin":"manual"},{"value":{"start":32,"end":33,"text":"
","labels":["MORADA"]},"id":"5ya-ene-
FN","from_name":"label","to_name":"text","type":"labels","origin":"manual"},{"value":{"
start":199,"end":217,"text":"(+351) 212 345
678","labels":["TELEFONE"]},"id":"AcN6IupZdq","from_name":"label","to_name":"text","typ
e":"labels","origin":"manual"},{"value":{"start":0,"end":10,"text":"João
Silva","labels":["NOME"]},"id":"6ergj6xZ4k","from_name":"label","to_name":"text","type"
:"labels","origin":"manual"},{"value":{"start":410,"end":415,"text":"João
","labels":["NOME"]},"id":"S1_gC5Ahx2","from_name":"label","to_name":"text","type":"lab
els","origin":"manual"},{"value":{"start":261,"end":266,"text":"João
","labels":["NOME"]},"id":"apNA-
iTlo7","from_name":"label","to_name":"text","type":"labels","origin":"manual"}],"was_ca
ncelled":false,"ground_truth":false,"created_at":"2024-04-
17T20:35:47.339186Z","updated_at":"2024-04-
17T20:58:42.725316Z","draft_created_at":null,"lead_time":131.691,"prediction":{},"resul
t_count":0,"unique_id":"0efd2d23-2466-4c8d-ad8b-
996db9dcc669","import_id":null,"last_action":null,"task":1,"project":3,"updated_by":1,"
parent_prediction":null,"parent_annotation":null,"last_created_by":null}],"file_upload"
:"a7abed49-
sampleNER.txt","drafts":[],"predictions":[],"data":{"$undefined$":"\/data\/upload\/3\/a
7abed49-sampleNER.txt"},"meta":{},"created_at":"2024-04-
17T20:05:50.676358Z","updated_at":"2024-04-
17T20:58:42.826500Z","inner_id":1,"total_annotations":1,"cancelled_annotations":0,"tota
l_predictions":0,"comment_count":0,"unresolved_comment_count":0,"last_comment_updated_a
t":null,"project":3,"updated_by":1,"comment_authors":[]}]
```


Figura 36 - Output Label Studio

A Tabela 2 apresenta as comparações realizadas durante os testes de utilização das ferramentas de anotação. Com base nos resultados obtidos, optou-se pela ferramenta Doccano como a preferida, devido ao seu desempenho superior em termos de facilidade de uso e eficiência no processo de anotação.

Métrica	Label Studio	Doccano
Importação/Exportação		X
Variabilidade de formatos	X	
Gestão de Equipa/Projeto		X
Facilidade de Anotação		X

Tabela 2 - Comparação Doccano/Label Studio

3.6 Introdução ao Desenvolvimento e Comparação de Modelos de Reconhecimento de Entidades Nomeadas

Nesta secção é descrita a nossa abordagem para explorar e aprimorar as técnicas de NER, começando com a utilização da biblioteca spaCy, conhecida pela sua eficiência e facilidade de uso em tarefas de PLN, e avançando para a utilização de modelos baseados em transformadores da plataforma Hugging Face, que oferecem capacidades de *fine-tuning* avançadas e suporte a uma vasta gama de modelos pré-treinados. Inicialmente, optamos pelo spaCy devido à sua robustez e simplicidade para treinar modelos NER rapidamente, o que nos permitiu implementar um pipeline de NER eficaz e avaliar seu desempenho com relativa facilidade. No entanto, as limitações na personalização dos modelos e na profundidade do *fine-tuning* criarama a necessidade de encontrar alternativas que oferecessem maior flexibilidade. Neste contexto, os modelos de transformadores disponíveis através da biblioteca *Hugging Face Transformers* apresentaram-se como uma alternativa poderosa. Com modelos pré-treinados como BERT, fomos capazes de adaptar um modelo de transformador para a tarefa de projeto NER específica, permitindo um ajuste fino mais detalhado e a capacidade de capturar nuances linguísticas mais complexas. Este capítulo documenta detalhadamente o processo de configuração, treino e *fine tuning* dos modelos utilizando ambas as plataformas. Além disso, são discutidos os desafios técnicos encontrados durante a implementação, incluindo problemas de carga de modelos, interpretação de saída e configuração de pipelines de NER. Por fim, apresentamos uma comparação sistemática do desempenho dos modelos configurados com spaCy, NLTK e *Hugging Face Transformers*,

utilizando métricas de precisão, *sensibilidade* e F1-*score* para avaliar e contrastar a eficácia de cada abordagem. Este estudo não ilustra apenas a aplicabilidade prática de ferramentas avançadas de PLN em tarefas de reconhecimento de entidades, mas também fornece *insights* sobre como diferentes ferramentas podem ser otimizadas e combinadas para alcançar a melhor performance possível em tarefas específicas de PLN.

3.6.1 SpaCy

Nesta subsecção do capítulo, será explorado como a biblioteca spaCy, pode ser utilizada para implementar e treinar um modelo NER. O spaCy não apenas facilita a carga e a aplicação de modelos de NER pré-treinados, mas também oferece uma interface robusta para o treino personalizado de modelos que podem identificar categorias específicas de entidades no texto. Este processo é crucial para adaptar o reconhecimento de entidades às necessidades específicas do projeto, permitindo uma análise mais precisa e relevante dos dados textuais. Inicialmente, será demonstrado como carregar um modelo pré-existente e aplicá-lo num texto de exemplo para identificar entidades. Em seguida, será detalhado como os dados são preparados e formatados para o treino, uma etapa fundamental que influencia diretamente a eficácia do modelo. A parte central desta seção concentra-se no treino do modelo NER usando o spaCy. Será discutida a configuração do ambiente de treino, incluindo a definição de parâmetros, a preparação de exemplos de treino, e como iterar sobre os dados para otimizar o reconhecimento de entidades. Esta fase é crucial para ajustar o modelo às particularidades do conjunto de dados específico, o que pode incluir nuances linguísticas e tipos de entidades que não são normalmente cobertos por modelos padrão. Por fim, a avaliação do modelo treinado, explicando como validar sua precisão e eficiência na identificação de entidades. Esta avaliação não apenas confirma a eficácia do modelo, mas também destaca áreas potenciais para ajustes e melhorias.

a) Gerar e armazenar modelo customizado

Primeiramente, verifica se o componente NER está presente na pipeline e, se não estiver, adiciona-o. O ficheiro de dados é então carregado, e exemplos de treino são criados com base nesses dados. O treino do modelo ocorre ao longo de 500 iterações, com os exemplos de treino a serem misturados em cada iteração. Durante o treino, o modelo vai sendo atualizado com os exemplos de entidades, e as perdas (erros) são calculadas para monitorizar o desempenho. Na figura 37 é apresentado um excerto da criação do modelo customizado. (O script pode ser encontrado na integra no Anexo A)

```
1. import spacy
 2. from spacy.util import minibatch, compounding
 3. from spacy.training import Example
 4. import json
 5. import random
 6.
 7. def load_data(filename):
 8.     train_data = []
 9.     with open(filename, 'r', encoding='utf-8') as file:
10.         for line in file:
11.             data = json.loads(line)
12.             text = data['text']
13.             entities = []
14.             for start, end, label in data['label']:
15.                 entities.append((start, end, label))
16.             train_data.append((text, {"entities": entities}))
17.     return train_data
.
.
.
```

Figura 37 - Gerar e armazenar modelo SpaCy

Na figura 38, é apresentado o script de importação do modelo gerado anteriormente assim como o input que será utilizador para o texto. O objetivo deste teste passa por imprimir as entidades identificadas e as suas respetivas *labels*, tal como apresentado na figura 39.

```
1. import spacy
2. PLN = spacy.load('teste')
3. print(PLN.pipe_names)
4. doc = PLN(
5.     "João Silva, residente na Rua das Flores 123, Lisboa, contactou a nossa
empresa no dia 5 de abril. Ele expressou interesse nos nossos serviços de consultoria
financeira e deixou seu número de telefone (+351) 212 345 678 para futuros contatos.
Durante a conversa, João mencionou que o seu NIF é 501 234 567. A reunião foi
agendada para a próxima quinta-feira às 15h, no escritório principal da empresa. Além
disso, João solicitou que todas as comunicações futuras fossem enviadas para o seu
email pessoal, joao.silva@example.com")
6. for ent in doc.ents:
7.     print('Text: ', ent.text, 'Label: ', ent.label_)
```

Figura 38 - Carregamento do modelo customizado

```
1. OUTPUT:
2. Text:  João Silva Label:  Nome
3. Text:  Rua das Flores 123 Label:  Morada
4. Text:  (+351) 212 345 678 Label:  Telefone
5. Text:  501 234 567 Label:  NIF
6. Text:  joao.silva@example.com Label:  Email
```

Figura 39 - Output de carregamento do modelo customizado

3.6.2 NLTK

Neste segmento da seção, será explorado como a biblioteca NLTK, pode ser utilizada para implementar e treinar um modelo de NER. Embora o NLTK não ofereça um sistema de treino de NER tão integrado quanto outras bibliotecas como o spaCy, fornece as ferramentas necessárias para construir um sistema de NER com uma abordagem mais granular e personalizável. Esta flexibilidade é essencial para adaptar o reconhecimento de entidades às necessidades específicas do projeto, permitindo uma análise mais detalhada e relevante dos dados textuais. Inicialmente, será introduzido o processo de utilização de funcionalidades básicas do NLTK para a identificação de entidades, empregando recursos como tokenização e etiquetagem que são fundamentais para o pré-processamento de texto em tarefas de NER. Esta etapa é crucial para entender como preparar os dados para o treino de um modelo de NER. De seguida, será detalhado como os dados são preparados e anotados para o treino, um processo que envolve a definição manual de entidades dentro de um *corpus* de texto. Esta preparação dos dados é vital, pois influencia diretamente a capacidade do modelo de aprender a reconhecer e classificar entidades de forma eficaz. A parte central desta secção concentra-se na construção e no treino do modelo de NER usando o NLTK. Será discutida a configuração do ambiente de treino, incluindo a seleção de *features* linguísticas relevantes e a escolha de um classificador adequado, como um classificador Bayesiano ou de Máquinas de Vetores de Suporte (SVM). Será incluída também a iteração sobre os dados de treino para otimizar o processo de reconhecimento de entidades, ajustando o modelo às particularidades do conjunto de dados específico, o que pode incluir nuances linguísticas e tipos de entidades que não são normalmente cobertos por modelos genéricos. Por fim, a avaliação do modelo treinado com o NLTK, explicando como validar sua precisão e eficiência na identificação de entidades através de técnicas como a matriz de confusão e cálculo de métricas como precisão, sensibilidade e a medida F1 (consultar capítulo II).

No script apresentado abaixo na figura 40, que poderá ser consultado na integra na secção do anexo B, A primeira função, "load_data_from_jsonl", lê o ficheiro jsonl e carrega as frases e suas respetivas anotações de entidades. A segunda função, "convert_to_nltk_format", converte os dados para o formato apropriado para treino no NLTK, tokenizando as frases e associando cada *token* ao seu respetivo rótulo de entidade. O formato adotado segue o padrão BIO (*Begin, Inside, Outside*)[14], que é amplamente utilizado para marcar o início e o interior das entidades identificadas. O script finaliza carregando o ficheiro jsonl e convertendo-o no formato adequado para o treino de modelos de classificação de entidades no NLTK.

---

[14] Esquema de anotação usado em NER para marcar o início (B), o interior (I) e o exterior (O) de entidades em textos.

```
1. from nltk.classify import accuracy
2. from nltk import NaiveBayesClassifier
3. import json
4. import random
5. from nltk.tokenize import word_tokenize
6.
7. def load_data_from_jsonl(jsonl_file_path):
8.     data = []
9.     with open(jsonl_file_path, 'r', encoding='utf-8') as file:
10.        for line in file:
11.            entry = json.loads(line)
12.            text = entry['text']
13.            entities = entry['label']
14.            data.append((text, entities))
15.    return data

                                   .
                                   .
                                   .
```

Figura 40 - Criação e armazenamento de modelo NTLK

No excerto abaixo, apresentado na figura 41, é definida a função features, que extrai características (ou features) de um token específico dentro de uma frase. Essas características são usadas no NLTK para treinar classificadores, como o Naive Bayes Classifier, que necessitam de informações sobre os tokens para realizar previsões, como o reconhecimento de entidades nomeadas.

- word: A palavra/token em si.
- is_first: Verifica se o token é o primeiro na frase.
- is_last: Verifica se o token é o último na frase.
- is_capitalized: Verifica se o primeiro caractere do token está em maiúscula.
- prefix-1, prefix-2, prefix-3: Extrai o prefixo (1, 2 ou 3 primeiros caracteres) do token. Se o token for menor, ele ajusta para usar a parte disponível.
- suffix-1, suffix-2, suffix-3: Extrai o sufixo (último, últimos dois ou três caracteres) do token.
- prev_word: O token anterior na frase. Se o token atual for o primeiro, retorna uma string vazia.
- next_word: O token seguinte na frase. Se o token atual for o último, retorna uma string vazia.
- has_hyphen: Verifica se o token contém um hífen.

Estas características fornecem ao classificador informações contextuais e estruturais sobre cada token, o que ajuda o modelo a tomar decisões mais informadas sobre a sua classificação. Por exemplo, tokens que começam com letras maiúsculas podem ser mais propensos a serem nomes

de pessoas ou lugares, e os prefixos e sufixos podem ajudar a identificar a natureza gramatical ou morfológica de uma palavra.

```
46.  def features(token, index, tokens):
47.     return {
48.         'word': token,
49.         'is_first': index == 0,
50.         'is_last': index == len(tokens) - 1,
51.         'is_capitalized': token[0].upper() == token[0],
52.         'prefix-1': token[0],
53.         'prefix-2': token[:2] if len(token) > 1 else token[0],
54.         'prefix-3': token[:3] if len(token) > 2 else token[0],
55.         'suffix-1': token[-1],
56.         'suffix-2': token[-2:] if len(token) > 1 else token[-1],
57.         'suffix-3': token[-3:] if len(token) > 2 else token[-1],
58.         'prev_word': '' if index == 0 else tokens[index - 1],
59.         'next_word': '' if index == len(tokens) - 1 else tokens[index + 1],
60.         'has_hyphen': '-' in token
61.     }
```

Figura 41 - Classificador de características

A função, apresentada na figura 42, "transform_to_dataset" converte frases anotadas (com *tokens* e *labels*) num formato adequado para treino. Para cada frase, a função extrai características de cada *token* usando a função "features", armazenando essas características em X (dados de treino) e os rótulos correspondentes em **y**. Os dados de treino são então misturados aleatoriamente e divididos em dois conjuntos: 80% para treino (X_train e y_train) e 20% para teste (X_test e y_test). Após essa divisão, o classificador de Naive Bayes é treinado com os dados de treino. Finalmente, a precisão do modelo é calculada com o conjunto de teste e impressa no ecrã, fornecendo uma medida de desempenho do modelo.

Este processo permite treinar o modelo com base em características extraídas dos *tokens* e avaliar a sua capacidade de generalizar para novos dados.

```
1. def transform_to_dataset(tagged_sentences):
2.     X, y = [], []
3.     for sentence in tagged_sentences:
4.         tokens, tags = zip(*sentence)
5.         for index, token in enumerate(tokens):
6.             feature_set = features(token, index, tokens)
7.             X.append(feature_set)
8.             y.append(tags[index])
9.     return X, y
10.
11.
12. X_train, y_train = transform_to_dataset(nltk_format_data)
13.
14. random.shuffle(X_train)
15. split_point = int(len(X_train) * 0.8)
16. X_test = X_train[split_point:]
17. y_test = y_train[split_point:]
18. X_train = X_train[:split_point]
19. y_train = y_train[:split_point]
20.
21. classifier = NaiveBayesClassifier.train(zip(X_train, y_train))
22.
23. test_accuracy = accuracy(classifier, zip(X_test, y_test))
24. print("Test Accuracy:", test_accuracy)
25.
```

Figura 42 - Treino de modelo NLTK

Na figura 43, é apresentado o output do processo de treino.

- *Test Accuracy*: Indica que o modelo não conseguiu prever corretamente nenhuma entidade no conjunto de teste, o que pode ser causado por vários fatores, como um conjunto de dados de treino insuficiente, falta de variedade nas entidades ou características mal definidas no processo de extração de *features*.

- *Identified Entities*: As entidades identificadas no output são claramente incorretas. O modelo atribuiu rótulos vazios às entidades, o que sugere que ele não foi capaz de distinguir corretamente os tipos de entidades (como nomes, moradas, NIF, etc.).

```
1. OUTPUT:
2. Test Accuracy: 0
3. Identified Entities: [(('Morada', 'nossa'), ''), (('NIF', '5'), ''), (('Nome', 'abril'), ''), (('Telefone', 'nos'), ''), (('NIF', '567'), ''), (('Telefone', 'às'), ''), (('Nome', 'empresa'), '')]
```

Figura 43 - Resultados do modelo apresentado

3.6.3 HFT

Neste segmento da seção, será explorada como a biblioteca *Hugging Face Transformers*, uma ferramenta de ponta no campo do processamento de linguagem natural, pode ser utilizada para desenvolver e aprimorar modelos de NER. Esta biblioteca destaca-se pela sua vasta coleção de modelos de transformadores pré-treinados, que podem ser facilmente adaptados e ajustados para tarefas específicas, incluindo NER. A capacidade de utilizar modelos baseados em transformadores permite uma compreensão contextual profunda do texto, o que é crucial para a identificação precisa de entidades complexas e variadas. Inicialmente, demonstraremos como carregar um modelo pré-treinado da *Hugging Face* e adaptá-lo a um conjunto de dados específico para identificar entidades. Este passo inicial é fundamental para avaliar a capacidade do modelo em capturar nuances linguísticas e contextuais antes de proceder com ajustes mais detalhados. De seguida, será detalhada a preparação e o formato dos dados de treino, que devem ser anotados e estruturados de acordo com os requisitos da biblioteca *Transformers*. Esta preparação é essencial para garantir que o modelo possa ser treinado de forma eficaz, utilizando exemplos que refletem os tipos de entidades que o modelo precisa de reconhecer. A parte central desta seção concentra-se no processo de *fine-tuning* do modelo escolhido. Será discutido como ajustar os parâmetros do modelo, como a taxa de aprendizagem e o número de ciclos de treino, para otimizar o desempenho na tarefa de NER. O processo de *fine-tuning* é crucial para adaptar o modelo pré-treinado às especificidades do conjunto de dados em questão, permitindo que ele aprenda a identificar e classificar corretamente as entidades específicas do projeto. Por fim, a avaliação do modelo treinado, utilizando as métricas padrão de NER como precisão, sensibilidade e F1-score para medir a eficácia do modelo em identificar e classificar entidades. Esta avaliação permite não apenas validar a precisão do modelo, mas também identificar áreas onde melhorias adicionais podem ser necessárias. Esta secção proporcionará uma visão detalhada de como a biblioteca *Hugging Face Transformer*s pode ser aplicada para alcançar resultados de alta qualidade em tarefas de NER, enfatizando a importância da tecnologia de transformadores e o poder do *fine-tuning* em modelos pré-treinados para tarefas de processamento de linguagem natural específicas.

```
1. import json
2. from transformers import BertTokenizer, BertForTokenClassification, pipeline
3.
4. def load_data_from_jsonl(jsonl_file_path):
5.     texts = []
6.     with open(jsonl_file_path, 'r', encoding='utf-8') as file:
7.         for line in file:
8.             entry = json.loads(line)
9.             texts.append(entry['text'])
10.     return texts
11.
12. model_path = './bert-model'
13.
14. id2label = {0: 'Nome', 1: 'Morada', 2: 'Telefone', 3: 'NIF', 4: 'Email'}
15.
16. tokenizer = BertTokenizer.from_pretrained(model_path)
17. model = BertForTokenClassification.from_pretrained(
18.     model_path, num_labels=len(id2label), id2label=id2label)
19.
20. ner_pipeline = pipeline('ner', model=model, tokenizer=tokenizer)
21.
22. jsonl_file_path = 'doccano.jsonl'
23. texts = load_data_from_jsonl(jsonl_file_path)
24.
25. text_example = texts[0]
26. ner_results = ner_pipeline(text_example)
27.
28. for result in ner_results:
29.     readable_label = result['entity']
30.     print(
31.         f"Texto: {result['word']}, Entidade: {readable_label}, Score:
{result['score']}")
```

Figura 44 - Resultados do modelo apresentado

```
1. Texto: express, Entidade: Nome, Score: 0.34069764614105225
2. Texto: ##ou, Entidade: Nome, Score: 0.3178587555885315
3. Texto: interesse, Entidade: NIF, Score: 0.3033685088157654
4. Texto: nos, Entidade: NIF, Score: 0.31712937355041504
5. Texto: nos, Entidade: NIF, Score: 0.22753632068634033
6. Texto: ##sos, Entidade: Morada, Score: 0.24696195125579834
7. Texto: servi, Entidade: Morada, Score: 0.25276392698287964
8. Texto: ##cos, Entidade: Email, Score: 0.242899551987648
9. Texto: de, Entidade: Nome, Score: 0.3113096356391907
10. Texto: consul, Entidade: Morada, Score: 0.2659603953361511
11. Texto: ##toria, Entidade: Nome, Score: 0.25251880288124084
12. Texto: finance, Entidade: Morada, Score: 0.2783874273300171
13. Texto: ##ira, Entidade: Nome, Score: 0.28668656945228577
14. Texto: e, Entidade: Nome, Score: 0.3174791634082794
15. Texto: deixou, Entidade: Nome, Score: 0.30457577109336853
16. Texto: seu, Entidade: Telefone, Score: 0.2870715260505676
17. Texto: numero, Entidade: Telefone, Score: 0.29304569959640503
18. Texto: de, Entidade: Nome, Score: 0.37732231616973877
19. Texto: telefon, Entidade: Nome, Score: 0.2629584074020386
20. Texto: ##e, Entidade: Nome, Score: 0.30378320813179016
```

Figura 45 - Excerto de resultados do modelo apresentado

3.6.4 Comparação e análise de resultados

Nesta secção, será realizada uma comparação detalhada entre diferentes ferramentas e bibliotecas de PLN, com base em vários critérios relevantes para o desenvolvimento de modelos. Serão apresentadas e analisadas tabelas que abordam tópicos como: Facilidade de Uso e Configuração, onde será avaliado o quão simples é a integração e configuração das bibliotecas; Capacidades de Treino e Fine-Tuning, que examinam a flexibilidade e eficiência de cada ferramenta no ajuste e personalização de modelos; Performance e Precisão, que compara o desempenho em termos de velocidade e exatidão das previsões; e, por fim, Recursos e Comunidade, onde serão analisados o suporte de documentação e a disponibilidade de recursos e comunidades de programadores. Este conjunto de análises fornecerá uma visão abrangente sobre as vantagens e limitações de cada solução.

a) Facilidade de Uso e Configuração

Na tabela 3 são comparados os processos de configuração de cada uma das bibliotecas.

SpaCy	Destaca-se pela sua interface amigável e fluxo de trabalho simplificado para carregar, treinar e aplicar modelos é altamente otimizado para tarefas rápidas de NER com um bom equilíbrio entre performance e facilidade de uso.
NLTK	Tradicionalmente mais utilizado em contextos acadêmicos e educacionais para ensino de PLN, flexível, mas pode exigir mais configurações manuais e preparação de dados comparado. A construção de um pipeline de NER é mais granular e menos direta.
HFT	Oferece acesso a uma vasta gama de modelos de transformadores pré-treinados, que podem ser finamente ajustados para tarefas específicas. Embora seja extremamente poderoso, o processo de *setup* e treino torna-se mais complexo e requer um entendimento mais profundo dos princípios de modelos e *transformers*.

Tabela 3 - Comparação de configuração bibliotecas NER

b) Capacidades de Treino e *Fine-Tuning*

A Tabela 4 compara as capacidades de treino de cada uma das bibliotecas, focando-se no desempenho de cada uma em relação ao seu principal caso de uso.

SpaCy	Permite um treino eficiente com a capacidade de ajustar modelos existentes a novos dados. No entanto, as opções para *fine-tuning* são relativamente limitadas em comparação aos *transformers*.
NLTK	Não possui um sistema integrado para treino de NER, necessitando que os utilizadores construam o seu próprio sistema de classificação, o que proporciona flexibilidade, mas exige mais trabalho.
HFT	Excelente para *fine-tuning* de modelos avançados com suporte extensivo para customização e otimização, adequado para tarefas que requerem um conhecimento profundo do contexto textual.

Tabela 4 - Comparação de capacidades de treino NER

c) Precisão

Na tabela 5 são apresentados comentários sobre a os resultados de precisão das diferentes bibliotecas.

SpaCy	Obteve os melhores resultados identificando todas as entidades esperadas no corpus de treino.
NLTK	O menos preciso das ferramentas, comprometida pelo processo de tratamento de dados.

HFT	Frequentemente alcança a melhor precisão e performance, especialmente em contextos complexos, devido ao poder dos modelos de *transformers* e a sua capacidade de entender contexto de maneira profunda. Para este caso não foi menos preciso que o spaCy.

Tabela 5 - Comparação NER performance e precisão

d) Recursos e comunidade

A Tabela 6 apresenta as experiências relacionadas ao suporte comunitário e aos recursos disponíveis para cada uma das bibliotecas, destacando a riqueza de documentação, tutoriais e o apoio da comunidade de utilizadores.

SpaCy	Possui uma vasta documentação e uma comunidade ativa que contribui com muitos recursos úteis.
NLTK	Tem uma grande base de utilizadores acadêmicos e é frequentemente recomendado para iniciantes. em PLN.
HFT	Tem uma das comunidades mais ativas e crescentes em PLN, com constante adição de novos modelos e ferramentas.

Tabela 6 - Comparação de suporte NER

e) Comparação de bibliotecas

Na tabela 7 é avaliada a biblioteca preferida de acordos com os testes realizados anteriormente para cada uma das áreas.

	Configuração	Treino	Precisão	Comunidade
SpaCy	X	X	X	
NLTK				X
HFT				X

Tabela 7 - Resultados de comparação

Após análise das capacidades, facilidade de uso, suporte comunitário e desempenho de cada uma das principais ferramentas PLN para o reconhecimento de entidades nomeadas, conclui-se que o spaCy destaca-se como a ferramenta mais adequada para as necessidades específicas do projeto. Durante o processo de comparação, observamos que, embora o NLTK ofereça flexibilidade educativa e o *Hugging Face Transformers* proporcione poderosos modelos pré-treinados, o spaCy oferece o melhor equilíbrio entre eficiência, facilidade de implementação e robustez para aplicações práticas. O spaCy não apenas simplificou o processo de treino e aplicação do modelo de NER com sua interface intuitiva e pipeline eficiente, mas também demonstrou alta precisão e velocidade na execução das tarefas de NER. Além disso, a capacidade do spaCy de ser facilmente integrado com outras aplicações e sistemas existentes sem a necessidade de ajustes complexos ou conhecimento técnico especializado em modelos de ML adiciona maior valor à escolha. Os resultados obtidos nas fases de teste e validação reforçaram a decisão, mostrando que o spaCy não só atendeu, mas muitas vezes superou as expectativas em termos de reconhecimento e classificação de entidades. A comunidade ativa e recursos extensivos também garantem suporte contínuo e atualizações regulares, o que é essencial para manter o sistema adaptável e resiliente a novos desafios e desenvolvimentos no campo do PLN. Portanto, dadas as comparações apresentadas e os resultados obtidos, o spaCy mostrou-se a ferramenta mais apropriada para o projeto, alinhando-se efetivamente com os objetivos a curto prazo e as demandas técnicas do mesmo.

4. Arquitetura e Planeamento

Neste capítulo, será apresentada a arquitetura do protótipo *PIxtract*, detalhando como as diferentes partes do sistema interagem e colaboram para garantir o processamento eficiente e a extração de entidades nomeadas. A arquitetura foi projetada com foco na simplicidade e escalabilidade, tendo em conta as limitações de recursos e o facto de o projeto estar numa fase de protótipo.

As secções a seguir irão explorar em profundidade os principais componentes e decisões técnicas que sustentam o funcionamento do protótipo:

- Fluxo Aplicacional: Será descrito o fluxo de dados e a sequência de processos desde o momento em que o utilizador carrega um ficheiro até à apresentação dos resultados. Esta secção abordará a forma como as bibliotecas de NER, os conversores de ficheiros e os módulos de processamento interagem.
- *Stack* Tecnológica: Esta secção detalha as tecnologias escolhidas para o desenvolvimento do protótipo, como *Python*, *FastAPI*, *spaCy*, *NLTK*, *PostgreSQL* e outras ferramentas que suportam o sistema.
- Construção da Base de Dados: Aqui será explicada a estrutura da base de dados, o tipo de dados armazenados, e como os dados processados e extraídos são geridos.
- Metodologia de Desenvolvimento: Será abordada a metodologia utilizada no desenvolvimento do protótipo, incluindo as práticas de programação adotadas, testes e o processo de iteração contínua.

4.1 Fluxo Aplicacional

Nesta secção, será detalhado o fluxo aplicacional do protótipo *PIxtract*, descrevendo como o sistema processa os ficheiros submetidos pelo utilizador, desde o momento do upload até à extração final das entidades nomeadas. Este fluxo abrange todas as etapas críticas que garantem o funcionamento eficaz do sistema, assegurando que cada componente realiza o seu papel de forma integrada e eficiente.

O fluxo inicia-se com a interação do utilizador, onde este seleciona a biblioteca de NER, o conversor de ficheiros, e carrega o documento que será processado. A partir deste ponto, o sistema segue por uma série de etapas bem definidas:

- Conversão de Ficheiros: Dependendo do tipo de documento enviado, o sistema utiliza a biblioteca de conversão selecionada para transformar o ficheiro num formato processável, como texto simples. Esta etapa garante que o conteúdo do documento possa ser lido e analisado pelas bibliotecas NER.
- Processamento de Entidades: Após a conversão, o conteúdo é passado para a biblioteca de NER escolhida (spaCy, NLTK, ou outra). Esta biblioteca faz a análise linguística do texto, identificando e categorizando as entidades nomeadas, como nomes, moradas, e números de telefone.
- Armazenamento e Apresentação dos Resultados: Uma vez identificadas as entidades, o sistema prepara os dados para exibição, criando uma tabela organizada que apresenta as entidades e os seus rótulos. Estes dados podem, posteriormente, ser exportados para um ficheiro CSV ou armazenados na base de dados para análise futura.

Cada uma destas etapas está interligada, garantindo um fluxo contínuo de dados desde o momento do upload até à visualização e exportação dos resultados. Esta secção permite compreender como o protótipo foi desenhado para operar de forma integrada e eficiente, oferecendo uma experiência de utilizador fluida e automatizada. Serão também discutidos eventuais pontos de otimização e desafios enfrentados ao longo do desenvolvimento deste fluxo.

Na figura 46 é apresentado o fluxo aplicacional criado para o protótipo do projeto. É constituído por 4 etapas:

- Input (Vários Formatos de Ficheiro): O processo começa com a entrada de ficheiros em vários formatos, como PDFs, documentos de texto, e outros tipos de documentos. O sistema foi desenhado para aceitar uma ampla gama de formatos, garantindo flexibilidade na importação de dados, independentemente do tipo de documento submetido pelo utilizador.

- Conversão (Para .txt): Após o upload do ficheiro, a primeira operação realizada é a conversão do formato original para um ficheiro de texto simples (.txt). Esta etapa é crucial, pois permite uniformizar os dados e prepará-los para análise subsequente. A conversão garante que todo o conteúdo relevante do documento seja preservado e esteja no formato adequado para processamento.

- NER (Bibliotecas e Regras): Uma vez que o ficheiro é convertido para texto, o sistema entra na fase de reconhecimento de entidades nomeadas (NER). Nesta fase, utiliza-se um conjunto de bibliotecas e regras predefinidas para analisar o conteúdo textual. O objetivo

é identificar e extrair entidades como nomes, emails, números de telefone, moradas, entre outros tipos de informações pessoais. O sistema aplica tanto regras linguísticas como algoritmos de análise para garantir que as entidades sejam corretamente reconhecidas e categorizadas.

- Output (Nome, Email, etc.): A última fase do fluxo é a saída dos dados processados. Após a identificação das entidades, os resultados são apresentados de forma clara, com as entidades extraídas organizadas em categorias como nome, email, número de telefone, entre outros. Estes dados podem ser apresentados ao utilizador numa interface amigável, com a opção de exportação para formatos reutilizáveis, como CSV.

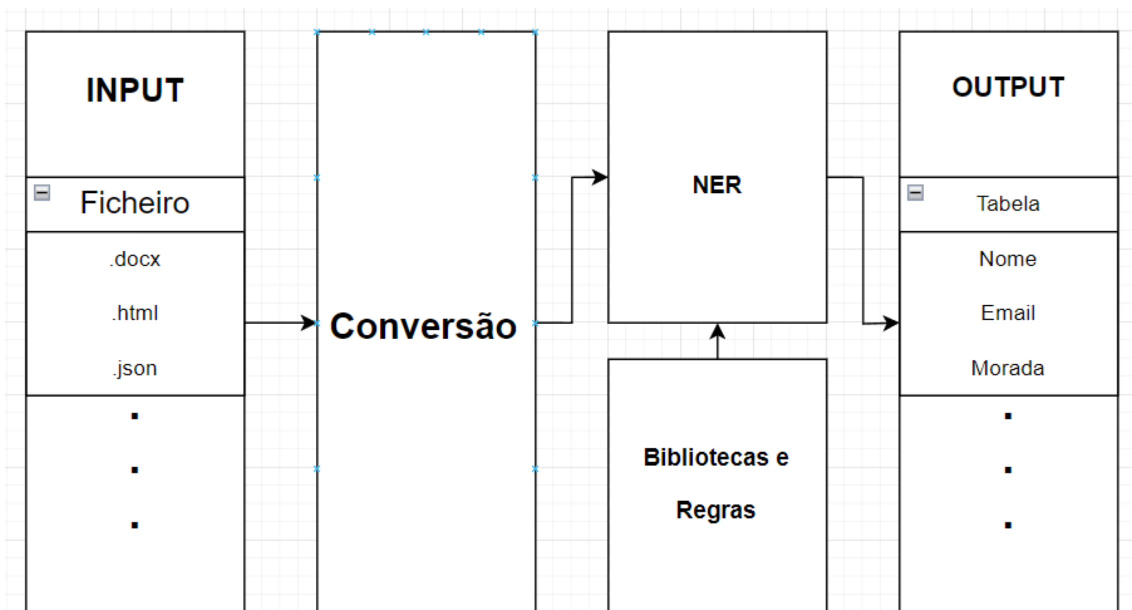

Figura 46 - Fluxo aplicacional

4.2 *Stack* Tecnológica

Nesta secção, será apresentada a stack tecnológica utilizada no desenvolvimento do protótipo *PIxtract*. A escolha das tecnologias foi orientada pela necessidade de garantir eficiência no processamento de ficheiros, flexibilidade na implementação e escalabilidade futura. Cada tecnologia selecionada desempenha um papel fundamental na estrutura do protótipo, desde o *backend* que gerência as operações principais, até às bibliotecas que facilitam a análise de dados e a extração de entidades. Além de descrever as tecnologias adotadas, será justificada a escolha de cada uma, com base nos benefícios que oferecem ao projeto. A prioridade foi garantir um

equilíbrio entre facilidade de desenvolvimento, desempenho robusto e capacidade de adaptação a diferentes volumes de dados. A *stack* tecnológica escolhida também reflete a intenção de manter o protótipo escalável e fácil de expandir em futuras iterações, à medida que o projeto evolui.

a) Python

Python foi a linguagem de programação escolhida para o desenvolvimento do protótipo PIxtract devido à sua versatilidade, simplicidade e robustez. A linguagem destaca-se pela sua curva de aprendizagem suave e pela capacidade de permitir uma implementação rápida e eficiente, o que é essencial num projeto de prototipagem. Além disso, Python oferece um suporte vasto e consolidado para tarefas de Processamento de Linguagem Natural (PLN), com inúmeras bibliotecas maduras e bem documentadas, como spaCy e NLTK, que foram fundamentais para a extração de entidades nomeadas. Outro fator determinante na escolha de Python é a sua facilidade de integração com outras ferramentas e tecnologias. Seja ao nível da gestão de bases de dados, APIs, ou na manipulação de ficheiros e conversão de formatos, Python permite uma interação suave com bibliotecas externas e sistemas, garantindo que diferentes componentes do protótipo funcionem de forma coesa.

b) PostgreSQL

Uma das suas principais vantagens é o suporte a consultas complexas, o que permite manipular e extrair dados de forma eficiente, mesmo em cenários que envolvem múltiplas tabelas e operações complexas. Esta capacidade é essencial para o armazenamento estruturado dos dados de extração, como entidades nomeadas, e para futuras operações de análise e relatórios.

Outro fator determinante na escolha do PostgreSQL foi a sua capacidade de lidar com grandes volumes de dados sem comprometer o desempenho. À medida que o protótipo evolui e a quantidade de documentos processados cresce, a escalabilidade e o desempenho do sistema de gestão de bases de dados tornam-se críticos. O PostgreSQL, com o seu excelente desempenho em transações e suporte a grandes volumes de informação, é ideal para assegurar que a aplicação permanece rápida e responsiva, mesmo sob cargas mais elevadas.

Adicionalmente, o PostgreSQL oferece uma vasta gama de funcionalidades avançadas, como suporte a JSON, índices avançados e operações de transação, o que o torna uma escolha versátil para gerir os dados complexos gerados pela extração de entidades. A sua flexibilidade permite que a base de dados se adapte às necessidades futuras do projeto, garantindo uma arquitetura de dados sólida e escalável.

c) DBeaver

Esta ferramenta oferece uma interface intuitiva e rica em funcionalidades, facilitando a execução de consultas SQL, a visualização e gestão dos dados armazenados no PostgreSQL.

Uma das grandes vantagens do DBeaver é a sua capacidade de suportar múltiplos tipos de bases de dados, o que o torna uma ferramenta versátil para qualquer ambiente de desenvolvimento. No contexto do *PIxtract*, permite não só monitorizar o desempenho das consultas e otimizar o armazenamento dos dados extraídos, mas também realizar de forma eficiente operações como a criação e modificação de tabelas, visualização de índices e análise de dados, tudo através de uma interface gráfica amigável.

e) FastAPI

Conhecido pela sua rapidez e eficiência, o FastAPI permite desenvolver APIs robustas e de alto desempenho utilizando Python, garantindo que o processamento de ficheiros e a extração de entidades sejam feitos de forma fluida e sem comprometer o tempo de resposta.

Uma das razões mais relevantes para a escolha do FastAPI foi o seu suporte integrado para a validação automática de dados e a geração de documentação através do Pydantic. Isto significa que, ao definir os modelos de dados da aplicação, a validação é feita de forma automática, assegurando que os dados processados e enviados através das APIs estejam no formato correto. Outra vantagem importante do FastAPI é a sua compatibilidade com operações assíncronas, o que permite que o sistema execute múltiplas operações de forma concorrente, otimizando o uso de recursos e melhorando a escalabilidade. Isto é especialmente útil para o PIxtract, onde o processamento de grandes volumes de dados pode ocorrer de forma simultânea, sem sobrecarregar o sistema.

Além disso, a curva de aprendizagem do FastAPI é relativamente baixa, permitindo uma implementação rápida, mas sem comprometer a flexibilidade ou a capacidade de evolução do protótipo, tornando-o ideal para um projeto em fase de prototipagem que pode vir a crescer no futuro.

f) Visual Studio Code

Uma das principais vantagens do VSCode é a sua flexibilidade em personalização. Ao longo do desenvolvimento do PIxtract, foram utilizadas extensões que otimizam a escrita de código, como formatação automática, realce de sintaxe e sugestões de código inteligentes, o que acelerou significativamente o fluxo de trabalho. Além disso, as extensões específicas para Python foram

essenciais para garantir que o código fosse eficiente e bem estruturado, integrando ferramentas de linting, auto-completar e verificação de erros em tempo real.

g) GitHub

Uma das principais vantagens do GitHub é o suporte a controlo de versão através do Git, o que permitiu acompanhar cada alteração no código, realizar *commits* frequentes e gerir o histórico de desenvolvimento de forma organizada. A utilização de *branches* facilitou a separação de novas funcionalidades e correções de erros, permitindo que estas fossem desenvolvidas e testadas sem afetar a versão principal do projeto.

A stack aplicacional do protótipo PIxtract foi desenhada de forma modular, com cada camada desempenhando um papel essencial no fluxo de funcionamento do sistema. A stack divide-se em três camadas principais: frontend, middleware e storage, cada uma utilizando tecnologias específicas que foram escolhidas pela sua eficiência, facilidade de uso e integração no ecossistema Python.

O *frontend* foi desenvolvido com uma abordagem simples e eficaz, utilizando Jinja2, HTML, e CSS para a construção da interface de utilizador.

- Jinja2: É um motor de *templates* que facilita a geração dinâmica de HTML no lado do servidor. A sua integração com o *backend* permite a passagem de dados diretamente para os *templates*, tornando a interface dinâmica e interativa, enquanto mantém a lógica separada da apresentação.
- HTML + CSS: Utilizados para estruturar e estilizar as páginas, garantem uma interface intuitiva e visualmente agradável para os utilizadores. O HTML organiza os elementos do conteúdo, enquanto o CSS cuida da aparência, tornando a aplicação acessível e fácil de navegar.

O *middleware*, que representa a camada lógica da aplicação, foi desenvolvido em Python, com a utilização de diversas bibliotecas especializadas em Processamento de Linguagem Natural e no desenvolvimento de APIs.

- Python: Serve como a linguagem central para o desenvolvimento do backend. A sua versatilidade e forte suporte a bibliotecas para PLN tornam-no a escolha ideal para lidar com as operações de processamento de dados e integração com o frontend e o storage.

- spaCy e NLTK: Estas bibliotecas foram escolhidas para o reconhecimento de entidades nomeadas (NER). Ambas são amplamente usadas no campo de PLN, sendo o spaCy

reconhecido pela sua eficiência e velocidade no processamento de grandes volumes de dados, enquanto o NLTK oferece uma abordagem mais académica, com ferramentas úteis para análises linguísticas.

- FastAPI: Um framework web rápido e moderno que permite a construção de APIs com alta performance. Ele facilita a comunicação entre o frontend e o backend, e suporta operações assíncronas, permitindo que o sistema processe múltiplas requisições simultaneamente, otimizando o desempenho.

- Hugging Face Transformers: Utilizado como uma tecnologia de ponta para o processamento de linguagem natural, esta biblioteca proporciona um poder adicional ao sistema ao suportar modelos baseados em *transformers*, como BERT, que são especialmente úteis para tarefas de NER em textos mais complexos e diversificados.

O sistema de armazenamento foi implementado utilizando o PostgreSQL, uma base de dados relacional robusta e escalável.

- PostgreSQL: Foi escolhido pela sua confiabilidade e capacidade de lidar com grandes volumes de dados, essenciais para armazenar os resultados das extrações de entidades. Ele oferece suporte a consultas complexas, garantindo que o sistema pode armazenar e recuperar informações de forma eficiente e com excelente desempenho.

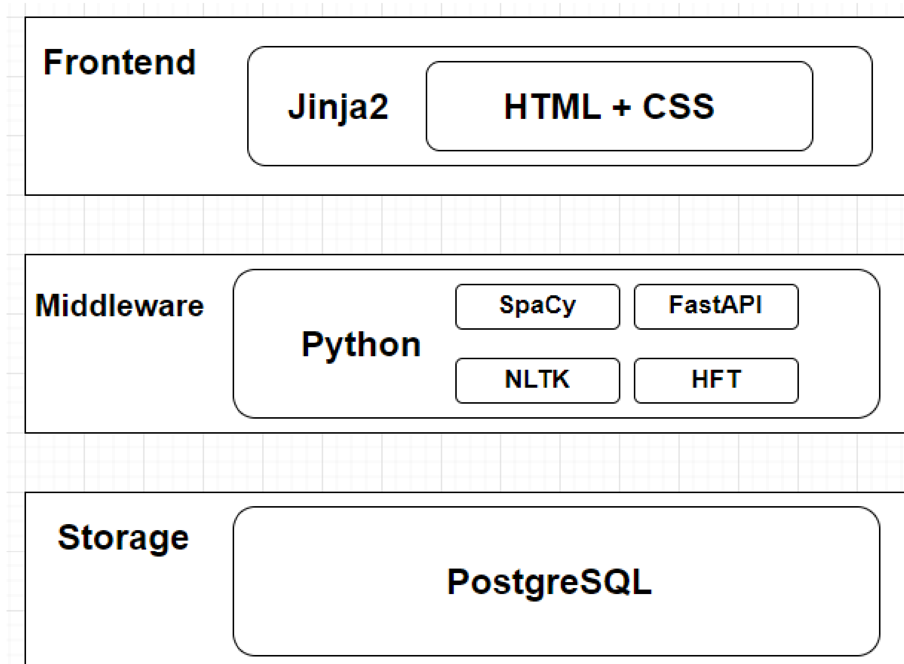

Figura 47 – Stack Aplicacional

4.3 Planeamento

Neste capítulo, será apresentado o processo de planeamento que orientou o desenvolvimento do protótipo *PIxtract*. O sucesso de um projeto tecnológico depende de um planeamento estruturado, que guie todas as fases de implementação, desde a conceção da interface de utilizador (UI) até à escolha das tecnologias subjacentes ao motor de reconhecimento de entidades nomeadas (NER) e à camada de conversão de documentos. Cada componente do sistema foi cuidadosamente planeado, tendo em vista a criação de um fluxo coeso e eficiente, que integra funcionalidades diversas de forma harmoniosa.

Ao longo deste capítulo, serão abordadas as seguintes áreas fundamentais do planeamento:

- Interface de utilizador: Planeamento da interface de utilizador, incluindo a forma como o utilizador interage com o sistema e como as funcionalidades são organizadas de maneira clara e intuitiva.
- Camada de Conversão de Documentos: Descrição do planeamento da camada responsável por converter os ficheiros carregados em formatos processáveis, garantindo a compatibilidade com o sistema.
- Motor NER: Detalhes sobre o planeamento da integração do motor de reconhecimento de entidades nomeadas, que analisa o texto e extrai informações essenciais como nomes, emails e moradas.
- Interface Aplicacional: Planeamento da camada de comunicação entre o *frontend* e o *backend*, assegurando que as operações fluem de forma eficaz entre as diferentes partes da aplicação.
- Metodologia de Desenvolvimento: Explicação da abordagem metodológica adotada ao longo do desenvolvimento, com ênfase nas práticas de programação, testes e ciclos de desenvolvimento que permitiram uma evolução contínua do protótipo.

4.4 Interface de Utilizador

a) Design e usabilidade

Visa proporcionar uma experiência de utilizador intuitiva e amigável, mesmo numa versão inicial. Layout simples e claro, menus de navegação básicos, botões de ação, formulários de input e feedback visual básico

b) Componentes de interface

Abaixo estão descritos os componentes planeados para a interface de utilizador:

- Página de carregamento de documentos: Tem o objetivo de permitir o carregamento de documentos, por parte do utilizador, nos formatos suportados. Será apresentado um botão de upload simples e uma apresentação de pré-visualização do descritivo do documento.
- Configurações da ferramenta: Área para os utilizadores poderem configurar os parâmetros da análise e conversão. As opções de escolha das bibliotecas de conversão e classificação serão apresentadas em dropdown menus.
- Área de resultados: Exibição dos resultados e tempo de execução. Página de output constituída por tabelas com a entidade identificada e *label* atribuída.
- Feedback: Componentes de mensagem para informar o utilizador sobre as operações realizadas, incluem-se neste tópico, mensagens de sucesso e erro.

b) Fluxo de utilização

A Figura 48 apresenta o fluxo de utilização esperado do protótipo, ilustrando de forma clara e detalhada as etapas principais do processo, desde a submissão do ficheiro até à extração e visualização dos resultados.

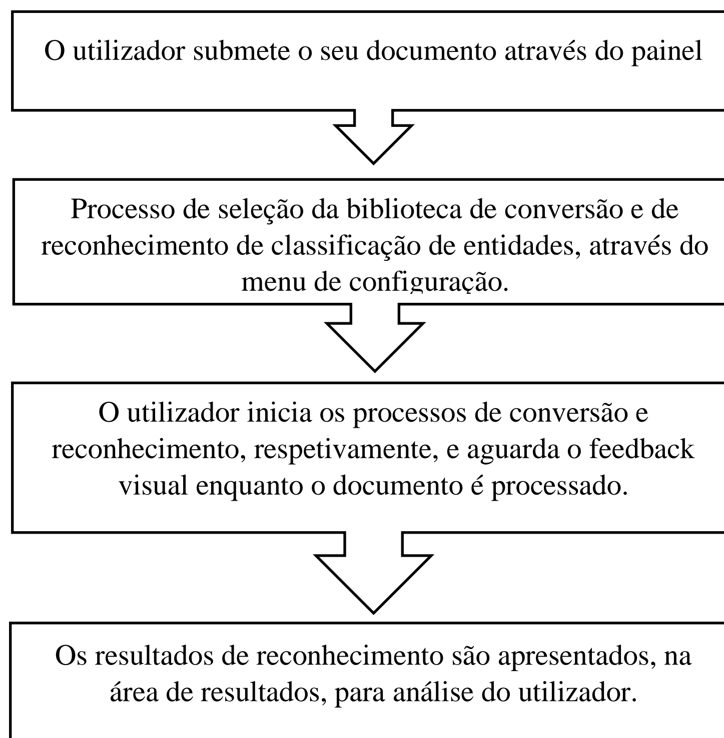

Figura 48 - Fluxo de utilização

d) Objetivos não prioritários para protótipo

- Migração para *React*: Para uma melhor apresentação e experiência do utilizador, esta biblioteca permitirá uma interface mais responsiva e funcional. Adicionará ao projeto componentes mais interativos e melhores práticas de desenvolvimento *frontend*.
- Suporte de outras línguas: Dado que o suporte inicial será só para português europeu.

4.5 Camada de conversão de documentos

A camada de conversão de documentos garante que o conteúdo destes ficheiros seja uniformizado e preparado para ser processado pelo motor de PLN. Independentemente do formato de entrada, este processo extrai e limpa o texto de forma a remover elementos desnecessários, como formatações e imagens, deixando apenas o conteúdo textual relevante. Ao converter estes documentos para texto simples, o sistema pode aplicar técnicas de NER e outras análises linguísticas com maior eficiência e precisão.

a) Fluxo do processo

A Figura 49 ilustra o fluxo completo do processo, desde o momento em que o utilizador carrega o ficheiro até ao armazenamento dos dados, detalhando cada etapa intermediária para uma melhor compreensão do funcionamento do sistema.

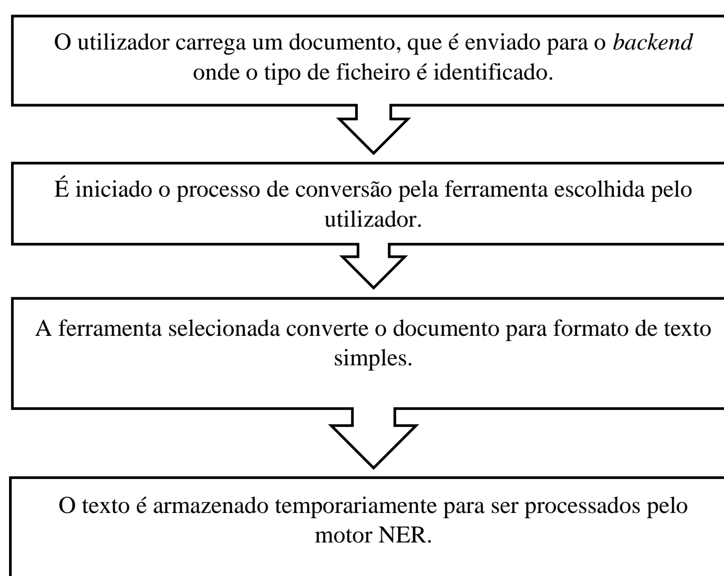

Figura 49 - Fluxo de processo

b) Desafios e considerações

- Documentos com qualidade de formatação baixa podem dificultar a qualidade da conversão.
- A necessidade de suporte multi formato exige uma arquitetura flexível e bem testada.
- A conversão de documentos deve ser eficiente para não introduzir atrasos significativos no processamento geral.

c) Objetivos não prioritários para protótipo

- Implementação de algoritmos de pós-processamento para melhorar a qualidade do texto extraído.
- Expansão do suporte para outros formatos de documentos populares e menos comuns.

4.6 Motor de reconhecimento de entidades nomeadas

O motor de NER é o núcleo central da aplicação, responsável pela análise detalhada de texto e pela extração de informações pessoais, como nomes, moradas, números de telefone, emails, e outras entidades sensíveis. Este componente utiliza técnicas avançadas de Processamento de Linguagem Natural (PLN) para identificar e classificar automaticamente essas entidades nomeadas no texto extraído dos documentos. O funcionamento do motor NER baseia-se em algoritmos sofisticados de PLN, que são treinados para reconhecer padrões linguísticos e semânticos específicos, permitindo que o sistema distinga entre diferentes tipos de informação. Este processo vai além de uma simples leitura do texto; o motor NER utiliza modelos linguísticos e estatísticos para analisar o contexto das palavras e determinar, com precisão, se um termo se refere, por exemplo, a uma pessoa, a um local, ou a uma organização.

Uma característica fundamental deste motor é o seu foco em português europeu, ajustando os modelos de reconhecimento para as especificidades linguísticas e culturais desta variante do idioma. Isto garante que as nuances regionais, como nomes próprios e formas de endereçamento, sejam corretamente interpretadas, aumentando a precisão da extração de dados. O motor não apenas reconhece entidades com base em padrões globais, mas também aplica regras e modelos adaptados ao contexto local, garantindo uma elevada taxa de acerto na extração de informações em português europeu.

a) Componentes do motor

A Tabela 8 apresenta os diferentes componentes do motor NER, juntamente com as respetivas funções e responsabilidades no contexto do projeto, detalhando as tarefas desempenhadas por cada um na execução do reconhecimento de entidades nomeadas.

Componente	Descrição	Tarefas
Pré-processamento	Limpeza e preparação do texto extraído para análise.	Remoção de caracteres especiais e normalização de texto. Tokenização, que divide o texto em unidades menores (*tokens*), como palavras ou frases. Lematização e *stemming* para reduzir as palavras às suas formas básicas.
Modelo de Reconhecimento de Entidades Nomeadas	Modelo treinado especificamente para reconhecer e classificar entidades nomeadas em português europeu.	Utilização modelos pré-treinados com spaCy e HFT. Carregamento de um modelo pré-treinado adaptado ao português europeu. Aplicação do modelo ao texto pré-processado para identificar entidades. Classificação das entidades identificadas em categorias como nomes, datas, endereços, etc.
Ajuste Fino do Modelo	Ajuste dos modelos pré-treinados para melhorar a precisão na identificação de entidades específicas do português europeu.	Recolha de um conjunto de dados anotados em português europeu. Treino adicional (*fine-tuning*) dos modelos utilizando esses dados. Avaliação e validação do desempenho do modelo ajustado.
Pós-processamento e Extração de Resultados	Avaliação e validação do desempenho do modelo ajustado.	Refina os resultados da análise e prepara-os para apresentação ao utilizador. Filtragem e eliminação de falsas entidades. Organização das entidades extraídas em um formato estruturado. Preparação dos dados para exibição na interface de utilizador.

Tabela 8 - Componentes de motor NER

4.6.1 Fluxo para reconhecimento de entidades

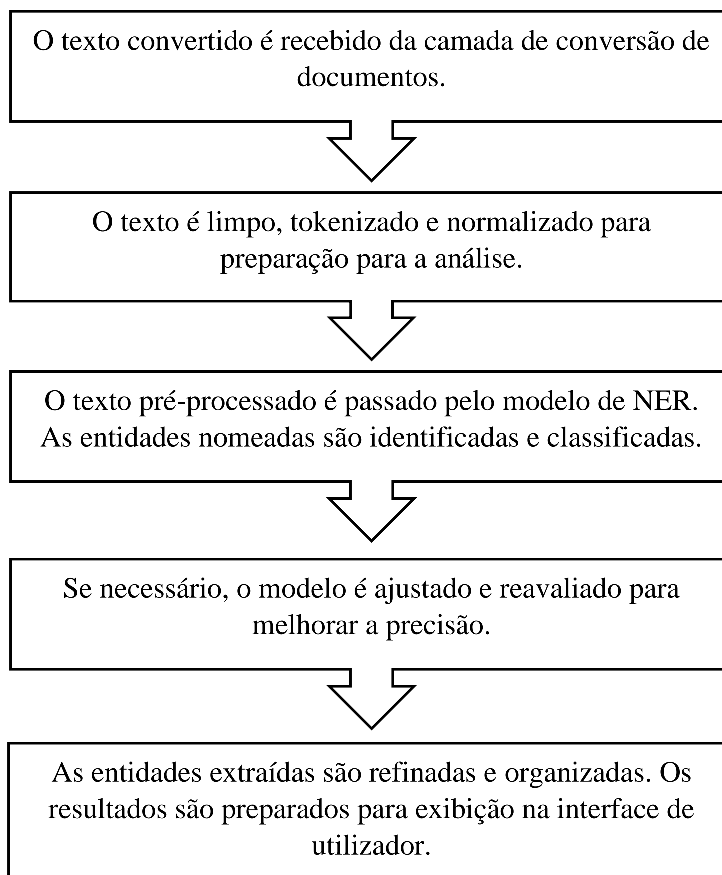

Figura 50 - Fluxo para reconhecimento de entidades

4.6.2 Atualizações Futuras

Atualmente, muitos modelos de reconhecimento de entidades nomeadas (NER) e outras tarefas de PLN são treinados com base em dados globais, predominantemente em inglês. Para garantir uma precisão ainda maior na extração de entidades em português europeu, será necessário treinar modelos específicos que incorporem as particularidades linguísticas e culturais deste idioma, o que inclui a variação regional de nomes, expressões e construções gramaticais.

O treino destes modelos avançados exigirá a curadoria de grandes conjuntos de dados em português europeu, garantindo que o sistema consiga capturar as nuances da linguagem e aplicar de forma eficaz as regras linguísticas específicas da região. Ao adotar modelos mais sofisticados, como aqueles baseados em arquiteturas de *Transformers* (ex.: BERT, GPT), o sistema poderá atingir um novo nível de precisão e eficiência na identificação e classificação de entidades, superando as limitações dos modelos atuais.

4.7 Interface Aplicacional

A API desempenha um papel fundamental ao conectar as diferentes camadas do sistema, garantindo que os utilizadores possam interagir com as principais funcionalidades da aplicação, como o upload de documentos, o processamento de texto e a extração de entidades nomeadas, de forma eficiente e escalável. Ao longo desta secção, será explicada a estrutura da API, detalhando os principais *endpoints* disponíveis, como o sistema lida com pedidos assíncronos e como a validação e a segurança são implementadas para assegurar que os dados são processados de forma confiável.

a) Objetivos

- Interação com o Frontend: Disponibilizar endpoints que permitam ao frontend comunicar-se de forma eficiente com o backend, facilitando operações como o upload de documentos, o processamento de texto e a recuperação dos resultados das análises. Esta comunicação garante que o utilizador possa interagir diretamente com as funcionalidades do sistema de forma intuitiva, enquanto o backend realiza o processamento necessário em segundo plano.

- Integração com Ferramentas Externas: Possibilitar a integração fluida com serviços e ferramentas externas especializadas, tanto para a conversão de documentos em diferentes formatos quanto para a aplicação de técnicas avançadas de processamento de linguagem natural (PLN). Esta integração expande as capacidades do sistema, permitindo a utilização de APIs de terceiros e aumentando a flexibilidade e a precisão no tratamento de dados.

- Gestão de Dados: Implementar uma estrutura eficiente para o armazenamento, gestão e recuperação dos dados, que inclui documentos carregados, texto extraído, entidades nomeadas e os resultados finais das análises. A organização desses dados é fundamental para garantir que as informações possam ser consultadas de forma rápida e estruturada, permitindo o histórico de análises e a fácil exportação dos resultados.

b) Estrutura

Upload de Documento: Este endpoint permite que o utilizador faça o upload de documentos em vários formatos. Após o upload, o sistema converte o ficheiro utilizando a ferramenta de conversão selecionada, transformando-o num formato processável. Uma vez convertido, o documento é analisado para identificar entidades nomeadas, como nomes, endereços ou números de telefone. O sistema então processa o texto extraído, aplicando as técnicas de reconhecimento de entidades, e retorna os resultados ao utilizador de forma estruturada e organizada. Este fluxo automatizado garante que todo o processo, desde a submissão do ficheiro até à apresentação dos dados extraídos, seja eficiente e direto.

Método	POST
URL	: /uploadfile/
Descrição	Permite o upload de documentos, converte o documento usando a ferramenta especificada, processa o texto para identificar entidades nomeadas e retorna os resultados.

Tabela 9 - Endpoint upload file

Parâmetros	
File	Ficheiro a ser carregado
Library	Biblioteca NER a ser utilizada (spaCy, NLTK, HFT).
Converter	Ferramenta de conversão a ser utilizada (calibre, pandoc, unoconv, soffice).
Resposta	Página HTML com o nome do ficheiro, tempo de conversão, tempo de processamento e entidades identificadas.

Tabela *10* - Parametros endpoit upload file

c) Design e Implementação

Definição de Recursos e Rotas: A arquitetura da API segue os princípios do design RESTful, garantindo uma interface clara e intuitiva. Cada recurso é mapeado para um endpoint específico, com rotas organizadas para refletir as ações possíveis sobre esses recursos. Os métodos HTTP, como GET, POST, PUT, e DELETE, são utilizados de acordo com as operações que os recursos devem suportar. Além disso, parâmetros de rota, query strings, e corpos de requisição são definidos de forma explícita para garantir clareza e facilitar a integração com clientes externos.

Validação e Serialização de Dados: Para garantir a consistência e segurança dos dados, a API utiliza a bilbioteca Pydantic para validação e *parsing* de dados de entrada. Os modelos de dados definidos com Pydantic permitem a validação automática de tipos, restrições de formato e integridade das informações recebidas pela API. Isso garante que dados inválidos sejam detectados antes de serem processados. Para as respostas da API, a serialização é gerida automaticamente, convertendo os dados de saída para JSON, garantindo que as respostas sigam os padrões estabelecidos e sejam facilmente consumidas pelos clientes.

Documentação da API**:** A documentação é gerada automaticamente pela *framework* FastAPI seguindo a especificação OpenAPI**.** Isso fornece uma interface interativa através do Swagger UI, onde os programadores podem explorar e testar os endpoints da API em tempo real. A documentação detalha cada endpoint, incluindo parâmetros de entrada, tipos de dados esperados, possíveis códigos de resposta HTTP, e exemplos de respostas, facilitando a compreensão e o uso correto da API.

4.8 Metodologia de desenvolvimento

Nesta secção, será abordada a metodologia de desenvolvimento adotada para o protótipo *PIxtract*. Dado que o projeto está a ser desenvolvido por uma única pessoa, a metodologia utilizada precisou de ser adaptada para refletir esta realidade, permitindo uma gestão eficiente do tempo e das tarefas. Embora o protótipo siga princípios ágeis, a abordagem foi ajustada para acomodar um ciclo de desenvolvimento mais flexível e individual.

O papel do orientador, que atua como principal *stakeholder*, é crucial para fornecer feedback contínuo e garantir que o projeto está alinhado com os objetivos definidos. Esta relação orientador-programador permitiu um ciclo de iteração mais direto, onde as funcionalidades são desenvolvidas, testadas e ajustadas com base no feedback, sem a necessidade de grandes equipas ou processos formais.

Ao longo desta secção, será descrita a forma como o ciclo de desenvolvimento foi estruturado, as ferramentas de controlo de versão e gestão de tarefas utilizadas, bem como a integração de processos de teste e validação contínuos. Esta metodologia adaptada permitiu um desenvolvimento ágil, focado em alcançar os principais marcos do projeto de forma eficiente, garantindo qualidade e cumprimento de prazos.

4.8.1 Planeamento

O processo de desenvolvimento da aplicação segue uma abordagem baseada na metodologia Agile, adaptada para as limitações e pequena dimensão da equipa, composta por uma única pessoa. Esta adaptação permite uma gestão mais flexível, com foco em entregas incrementais e adaptações contínuas às necessidades do projeto. Embora os princípios ágeis tradicionais envolvam equipas multifuncionais e iterações frequentes com os stakeholders, no contexto deste projeto, a metodologia foi ajustada para equilibrar a carga de trabalho e os prazos de entrega de forma eficaz. As tarefas são divididas em pequenos ciclos de desenvolvimento, cada um focado na implementação de funcionalidades específicas, permitindo um progresso contínuo e um rápido ajuste com base no feedback fornecido pelo orientador, que atua como principal stakeholder. Este planeamento adaptado garante que, apesar das limitações, o desenvolvimento avance de forma organizada, mantendo o foco na entrega de funcionalidades principais e na flexibilidade para responder a novas exigências ou melhorias à medida que o projeto evolui.

a) Definição dos Requisitos

Esta etapa passa por reuniões com o orientador do projeto (*stakeholder*) para entender as necessidades e expectativas; na análise de requisitos para garantir que todos os aspetos do projeto sejam considerados e finalmente na documentação dos requisitos num formato claro e compreensível.

b) Definição de prioridades

Organizar as funcionalidades por ordem de importância e urgência. Criação de um *backlog* do projeto, com todas as funcionalidades desejadas. Uso de técnicas como MoSCoW[15] para priorizar funcionalidades e definição de entregas incrementais baseadas nas mesmas prioridades.

[15] Must have, Should have, Could have, Won't have

c) User stories

Descrição das funcionalidades do ponto de vista do utilizador final. São formuladas no formato: tipo de utilizador/ação/benefício. Após a redação, as *user stories* serão validadas com o *stakeholder* para garantir que suas necessidades sejam corretamente compreendidas.

d) Planeamento de sprints

Organização do trabalho em ciclos de objetivos. Quanto à duração das sprints, geralmente com uma duração entre 1 e 4 semanas, sendo o seu conteúdo baseado em tarefas e *user stories*. Um sprint *backlog* é criado para acompanhar o progresso.

e) Alocação dos recursos

Para garantir que todos os recursos necessários estejam disponíveis, nesta etapa são identificadas as habilidades e competências necessárias para o projeto, como as ferramentas e tecnologias necessárias.

f) Ferramentas para planeamento

Utilização de várias ferramentas para facilitar o planeamento e a gestão do projeto. Ferramentas como Trello são usadas para a gestão de tarefas e acompanhamento do progresso das sprints. O controlo de versão e a gestão de código são realizados através do Git e GitHub.

4.8.2 Desenvolvimento

Após o planeamento, o desenvolvimento da aplicação segue uma abordagem iterativa e incremental, típica das metodologias *agile*. Este processo permite a construção contínua e a melhoria do software, garantindo que o produto final atenda aos requisitos e expectativas do *stakeholder*.

a) Sprints

O desenvolvimento é dividido em sprints, que são ciclos curtos de trabalho, geralmente com duração de 1 a 4 semanas. Cada sprint inclui atividades de planeamento, desenvolvimento, testes e revisão, permitindo a entrega contínua das novas funcionalidades. No início de cada *sprint*, a equipe se reúne para revisar o *backlog* do projeto e selecionar as *user stories* e tarefas que serão trabalhadas durante o *sprint*. As tarefas são detalhadas e estimadas em termos de esforço

necessário. O objetivo é garantir que o trabalho planeado para o *sprint* seja realista e atingível dentro dos limites de tempo alocados.

b) Desenvolvimento

Durante o sprint, os programadores trabalham nas tarefas atribuídas, implementando novas funcionalidades e corrigindo bugs. A comunicação constante entre os membros da equipe é essencial para resolver problemas rapidamente e garantir que todos estejam alinhados com os objetivos do sprint.

c) Revisão de Código

Para garantir a qualidade do código, as revisões de código são realizadas regularmente. Estas revisões ajudam a identificar e corrigir problemas no software e garantir a conformidade com os padrões de código.

d) Testes

Cada funcionalidade desenvolvida é acompanhada de testes unitários e de integração (se necessário). Os testes unitários garantem que cada componente individual funcione corretamente, enquanto os testes de integração verificam a interação entre diferentes componentes do sistema.

e) Revisão de sprint

No final de cada sprint, é realizada uma revisão do sprint. Durante esta reunião, as funcionalidades desenvolvidas são demonstradas ao *stakeholder*, recebendo feedback. Esta revisão a avaliação do progresso feito, discussão dos desafios enfrentados e o planeamento de_melhorias para os próximos *sprints*. Além da revisão do *sprint*, é também realizada uma retrospetiva do sprint. O foco desta etapa é refletir sobre o processo de trabalho, identificando o que funcionou bem e o que pode ser melhorado. As ações de melhoria identificadas são implementadas nos sprints seguintes para aumentar a eficiência e a eficácia.

4.8.3 - Entrega e Feedback

a) Feedback

Etapa essencial para garantir que o produto final atende às expectativas do *stakeholder* (orientador) e utilizador final. Este processo envolve a comunicação contínua com as partes

interessadas. Realizar reuniões de revisão periódicas com o *stakeholder* para demonstrar as funcionalidades desenvolvidas e receber feedback.

b) Análise

Após a recolha, o feedback deve ser analisado e priorizado para identificar áreas de melhoria e novas funcionalidades desejadas.

- Análise Qualitativa: Rever o feedback para identificar tendências e problemas comuns.
- Análise Quantitativa: Utilizar métricas de uso da aplicação para tomar decisões informadas sobre melhorias.

4.8.4 Iteração com base no *feedback*

Com base no feedback recebido, as funcionalidades da aplicação são refinadas e novas funcionalidades são planejadas para os próximos ciclos de desenvolvimento.

- *Backlog* do Produto: Atualizar o *backlog* do produto com as melhorias e funcionalidades solicitadas.
- Sprints de Melhoria: Planear sprints focados em implementar as melhorias com base no feedback.

5. Desenvolvimento e Testes

Este capítulo aborda, de forma detalhada, o processo de desenvolvimento do projeto, englobando as diferentes etapas técnicas e metodológicas seguidas para a criação do modelo. Inicialmente, será descrito o ciclo de desenvolvimento, desde a conceção do modelo, passando pela preparação dos dados e pela sua implementação, até à fase de otimização. Posteriormente, serão discutidos os testes realizados, destacando as estratégias adotadas para validar a robustez e a eficácia do modelo em diversos cenários. Serão também exploradas as métricas de avaliação utilizadas para medir a sua precisão e desempenho.

Por fim, a exposição dos resultados obtidos permitirá uma análise crítica e comparativa entre os objetivos traçados e os resultados alcançados, avaliando o grau de sucesso do projeto e identificando eventuais melhorias ou desafios que poderão ser considerados em trabalhos futuros.

5.1 Lista de funcionalidades

A tabela 11 apresenta uma visão geral das funcionalidades implementadas no projeto, juntamente com a respetiva descrição. Estas funcionalidades foram desenvolvidas para responder aos requisitos específicos definidos para o caso de uso (consultar capítulo 4), garantindo a eficiência e eficácia do modelo.

Nome	Descrição
Upload de ficheiros para diversos formatos	Suporte para submissão de documentos em múltiplos formatos, garantindo flexibilidade no tipo de ficheiros.
Conversão do ficheiro em formato de texto	Compatibilidade com diversas bibliotecas para converter documentos para texto, suportando múltiplos formatos.
Reconhecimento de entidades de informação pessoal	Integração com modelos Spacy e NLTK para reconhecimento eficiente de entidades de informação pessoal.
Visualização de Resultados	Tabela paginada e interativa para apresentação clara e organizada dos resultados obtidos.

Exportação de ficheiro	Função para exportar os resultados visualizados em diversos formatos de ficheiros, mantendo a integridade dos dados.

Tabela *11* - Lista de funcionalidades do projeto

5.2 Código de desenvolvimento

Nesta secção, será apresentado o código desenvolvido ao longo do projeto, acompanhado de uma explicação detalhada de cada componente. O objetivo é fornecer uma visão clara sobre a estrutura do código, as bibliotecas e *frameworks* utilizadas, bem como as decisões técnicas tomadas durante a sua implementação. Cada parte do código será explicada de forma a demonstrar como contribui para o funcionamento do sistema, desde a importação de dados e pré-processamento, até à construção e treino do modelo. Serão também abordados os desafios enfrentados e as soluções encontradas para garantir a eficiência e o desempenho do projeto.

Esta secção pretende, assim, oferecer uma compreensão aprofundada do desenvolvimento técnico, permitindo uma análise crítica e replicação do trabalho, caso seja necessário.

5.2.1 Middleware

Nesta secção, será abordado o papel do *middleware* no contexto da aplicação desenvolvida. O *middleware* é uma camada intermediária entre o pedido (*request*) e a resposta (*response*), permitindo a execução de lógica adicional durante o ciclo de vida das requisições HTTP. Ele é frequentemente utilizado para funções transversais, como autenticação, registo de atividades (*logging*), tratamento de erros, e manipulação de cabeçalhos, antes de o pedido chegar ao controlador principal.

Este código implementa um *endpoint* HTTP do tipo POST, usando o FastAPI, que permite o envio de ficheiros para serem convertidos em texto através de diferentes bibliotecas de conversão.

A função “upload_file”, apresentada na figura 51, é assíncrona e aceita um ficheiro e parâmetros adicionais como a biblioteca e o conversor a utilizar.

- @router.post("/uploadfile/"): Este *decorator* define um *endpoint* POST na rota /uploadfile/, que será utilizado para o envio de ficheiros.

- async def upload_file(...): Define uma função assíncrona que será executada quando o endpoint for chamado. Os parâmetros incluem:
    - request: O objeto de requisição, contendo informações sobre o pedido HTTP.
    - file: Um ficheiro enviado pelo utilizador através do parâmetro UploadFile.
    - library e converter: Parâmetros adicionais enviados no formulário para especificar a biblioteca e o conversor a ser utilizado.
    - file_location = f"./{file.filename}": Cria uma string que define o caminho onde o ficheiro enviado será temporariamente guardado no servidor, usando o nome original do ficheiro.
    - with open(file_location, "wb") as f: Abre o ficheiro em modo de escrita binária (wb), para permitir a escrita do conteúdo do ficheiro recebido no caminho definido anteriormente.
    - f.write(await file.read()): Lê o conteúdo do ficheiro de forma assíncrona e escreve-o no local definido.
    - output_file_location = file_location.rsplit('.', 1)[0] + '.txt': Define o caminho do ficheiro convertido, mudando a extensão do ficheiro original para .txt

```
1. @router.post("/uploadfile/")
2. async def upload_file(request: Request, file: UploadFile = File(...), library:
str = Form(...), converter: str = Form(...)):
3.     file_location = f"./{file.filename}"
4.
5.     with open(file_location, "wb") as f:
6.         f.write(await file.read())
7.
8.     output_file_location = file_location.rsplit('.', 1)[0] + '.txt'
9.
```

Figura 51 - Função de Upload de ficheiros

Nesta secção, o código mede o tempo de conversão e executa a conversão do ficheiro para texto:

- start_conversion_time: Marca o tempo de início da conversão.

Dependendo do valor de converter (enviado no formulário), o ficheiro é convertido usando uma das seguintes funções:

- convert_with_calibre()
- convert_with_pandoc()

- convert_with_unoconv()
- convert_with_soffice()

No caso de um conversor inválido for passado, é retornado um erro com a mensagem "Conversor inválido.". Relativamente ao tempo de conversão é calculado através da diferença entre end_conversion_time e start_conversion_time.

- with open(output_file_location, "r", encoding="utf-8") as f: Abre o ficheiro convertido em modo de leitura, especificando a codificação UTF-8 para garantir a correta leitura dos caracteres.
- text_content = f.read(): Lê o conteúdo textual do ficheiro convertido para uma variável, que pode ser posteriormente processada ou devolvida como resposta.

```
10.     start_conversion_time = time.time()
11.     if converter == 'calibre':
12.         convert_with_calibre(file_location, output_file_location)
13.     elif converter == 'pandoc':
14.         convert_with_pandoc(file_location, output_file_location)
15.     elif converter == 'unoconv':
16.         convert_with_unoconv(file_location, output_file_location)
17.     elif converter == 'soffice':
18.         convert_with_soffice(file_location, output_file_location)
19.     else:
20.         return {"error": "Conversor inválido."}
21.     end_conversion_time = time.time()
22.     conversion_time = end_conversion_time - start_conversion_time
23.
24.     with open(output_file_location, "r", encoding="utf-8") as f:
25.         text_content = f.read()
26.
```

Figura 52 - Função de conversão

- start_processing_time = time.time(): Marca o tempo de início do processamento de entidades para que o tempo total gasto no processamento possa ser medido. Bloco try: Este bloco tenta executar o processamento de entidades com base na biblioteca escolhida (spaCy ou NLTK):

- if library == 'spaCy':: Se a biblioteca escolhida for spaCy, o texto será processado pelo modelo associado ao spaCy, e as entidades identificadas serão armazenadas na variável entities.

- elif library == 'NLTK':: Se a biblioteca for NLTK, o texto será processado utilizando o modelo NLTK, e as entidades extraídas serão igualmente armazenadas na variável entities.
- else:: Caso a biblioteca indicada não seja nenhuma das opções válidas, o código retorna um erro com a mensagem "Biblioteca NER inválida.", informando que a escolha da biblioteca foi incorreta.

Bloco except: Caso ocorra alguma exceção durante o processamento (por exemplo, falha no carregamento do modelo ou erro de processamento), será lançada uma exceção HTTP com o código 500, indicando um erro interno no servidor. A mensagem de erro (str(e)) será incluída na resposta.

- end_processing_time = time.time(): Marca o tempo de término do processamento de entidades.
- processing_time = end_processing_time - start_processing_time: Calcula o tempo total gasto no processamento de entidades, subtraindo o tempo de início do tempo de término.
- return templates.TemplateResponse("results.html", {...}): Após o processamento de entidades e o cálculo do tempo, o código gera uma resposta HTML utilizando o template "results.html". Os parâmetros passados para o template incluem:
    - request: O objeto de requisição HTTP.
    - filename: O nome do ficheiro original enviado.
    - conversion_time: O tempo que levou a conversão do ficheiro para texto.
    - processing_time: O tempo que levou o processamento de entidades.
    - entities: As entidades extraídas do texto pelo modelo de NER.

```
1. start_processing_time = time.time()
2.     try:
3.         if library == 'spaCy':
4.             entities = spacy_model.process_with_spacy(text_content)
5.         elif library == 'NLTK':
6.             entities = nltk_model.process_with_nltk(text_content)
7.         else:
8.             return {"error": "Biblioteca NER inválida."}
9.     except Exception as e:
10.        raise HTTPException(status_code=500, detail=str(e))
11.    end_processing_time = time.time()
12.    processing_time = end_processing_time - start_processing_time
13.
                                            .
                                            .
                                            .
```

Figura 53 - Processamento de identificação entidades

Este código lida com a conversão de ficheiros para texto simples utilizando diferentes ferramentas (Calibre, Pandoc, Unoconv e Soffice). Ele permite a seleção do conversor com base na entrada do utilizador e garante a execução correta do processo de conversão através de comandos do sistema ou bibliotecas específicas. O fluxo de conversão começa com a linha start_conversion_time = time.time(), que marca o início do processo de conversão, registando a hora atual para que o tempo total da conversão possa ser calculado posteriormente. Em seguida, o código utiliza uma estrutura condicional para escolher a ferramenta de conversão apropriada, de acordo com o valor do parâmetro converter fornecido pelo utilizador:

- Se o valor for 'calibre', a função convert_with_calibre() é chamada para realizar a conversão utilizando a ferramenta Calibre.
- Se o valor for 'pandoc', a função convert_with_pandoc() é invocada para a conversão através da ferramenta Pandoc.
- Se o valor for 'unoconv', a função convert_with_unoconv() é utilizada para a conversão com a ferramenta Unoconv.
- Se o valor for 'soffice', a função convert_with_soffice() é executada, utilizando a ferramenta Soffice.

Cada uma destas funções é responsável por executar o comando adequado para converter o ficheiro original num ficheiro de texto simples. Por exemplo, a função convert_with_calibre() utiliza o comando ebook-convert para realizar a conversão, enquanto a função convert_with_pandoc() utiliza a biblioteca pypandoc para o mesmo fim. Outras ferramentas, como Unoconv e Soffice, utilizam comandos do sistema para realizar a conversão, como unoconv -f txt e soffice --headless --convert-to txt, respetivamente. Se o valor do conversor não for reconhecido, o código devolve um erro, indicando que o conversor fornecido é inválido. Desta forma, o código oferece uma forma flexível e eficiente de realizar a conversão de ficheiros para texto simples, suportando múltiplas ferramentas e garantindo que o processo seja realizado corretamente.

```
 1. def convert_with_calibre(input_file: str, output_file: str) -> None:
 2.     """
 3.     Conversão de ficheiro para plain text através de Calibre's ebook-convert
tool.
 4.     """
 5.     command = ["ebook-convert", input_file, output_file]
 6.     subprocess.run(command, check=True)
 7.
 8. def convert_with_pandoc(input_file: str, output_file: str) -> None:
 9.     """
10.     Conversão de ficheiro para plain text através de Pandoc.
11.     """
```

Figura 54 - Função de conversão: calibre e pandoc

```
14.
15. def convert_with_unoconv(input_file: str, output_file: str) -> None:
16.     """
17.     Conversão de ficheiro para plain text através de unoconv.
18.     """
19.     command = ["unoconv", "-f", "txt", "-o", output_file, input_file]
20.     subprocess.run(command, check=True)
21.
22. def convert_with_soffice(input_file: str, output_file: str) -> None:
23.     """
24.     Conversão de ficheiro para plain text através de soffice.
25.     """
26.     command = ["soffice", "--headless", "--convert-to", "txt:Text",
27.                "--outdir", os.path.dirname(output_file), input_file]
28.     subprocess.run(command, check=True)
29.
```

Figura 55 - Função de conversão: unoconv e soffice

A Figura 56 apresenta um excerto de código referente à condição de seleção da biblioteca NER, destacando a lógica implementada para permitir ao utilizador escolher a biblioteca mais adequada para o processamento de entidades nomeadas:

- Bloco try:

O bloco try tenta executar o processamento de entidades, de acordo com a biblioteca especificada (spaCy ou NLTK). Caso ocorra algum erro durante a execução, o bloco *except* é acionado.

- if library == 'spaCy'::

Se o valor da variável library for "spaCy", a função "process_with_spacy()" será chamada, passando o conteúdo do texto (text_content) para ser processado pelo modelo spaCy. As entidades extraídas são armazenadas na variável entities.

- elif library == 'NLTK'::

Se a variável library for "NLTK", a função "process_with_nltk()" será utilizada, processando o texto com o modelo NLTK e armazenando as entidades na mesma variável *entities*.

- else::

Se o valor da variável *library* não for nem "spaCy" nem "NLTK", o código retorna uma resposta JSON com um erro, indicando que a biblioteca de NER especificada é inválida.

- Bloco except Exception as e::

Caso ocorra qualquer exceção durante o processamento (por exemplo, se o modelo falhar ou o texto não puder ser processado corretamente), a exceção é capturada e uma resposta HTTP é

gerada com o *status* 500 (erro interno do servidor). A mensagem de erro específica é incluída através da função “str(e)”.

```
1. try:
2.         if library == 'spaCy':
3.             entities = spacy_model.process_with_spacy(text_content)
4.         elif library == 'NLTK':
5.             entities = nltk_model.process_with_nltk(text_content)
6.         else:
7.             return {"error": "Biblioteca NER inválida."}
8.     except Exception as e:
9.         raise HTTPException(status_code=500, detail=str(e))
10.
```

Figura 56 - Bibliotecas de reconhecimento de entidades

A Figura 57 ilustra o processo de carregamento do modelo customizado no **spaCy**, destacando como o modelo treinado é integrado na aplicação para reconhecimento de entidades nomeadas de forma eficiente.

- models_dir = os.path.dirname(__file__):
    - Define a variável models_dir como o caminho para o diretório onde o script Python atual está localizado. A função os.path.dirname(__file__) obtém o caminho do diretório do ficheiro de código atual.
- spacy_model_path = os.path.join(models_dir, "spacyM1"):
    - Cria o caminho completo para o modelo *spaCy* chamado "spacyM1", que se encontra no mesmo diretório que o ficheiro de código. A função os.path.join() junta o diretório base ao nome da pasta do modelo.
- Bloco if not os.path.exists(spacy_model_path)::
    - Verifica se o diretório do modelo *spaCy* (spacyM1) existe. Se não existir, o programa lança um erro FileNotFoundError, informando que o diretório não foi encontrado.
- raise FileNotFoundError(...):
    - Levanta uma exceção indicando que o diretório do modelo não foi encontrado. A mensagem sugere ao utilizador tentar outra biblioteca se a diretoria não estiver presente.

Dentro do bloco try, o modelo *spaCy* é carregado a partir do caminho especificado (spacy_model_path). A função "spacy.load()" tenta carregar o modelo a partir do diretório indicado.

- Bloco except:
    - Se houver algum erro ao carregar o modelo (por exemplo, se o modelo estiver corrompido ou incompatível), o bloco except captura a exceção e levanta um erro RuntimeError com uma mensagem detalhada sobre o problema encontrado.

- Função process_with_spacy(text_content):
    - Esta função recebe o conteúdo de texto (text_content) e usa o modelo carregado (spacy_nlp) para processá-lo.

- doc = spacy_nlp(text_content):
    - O texto é passado para o modelo *spaCy*, que analisa o texto e retorna um objeto doc contendo o resultado do processamento, incluindo as entidades reconhecidas.

- return [(ent.text, ent.label_) for ent in doc.ents]:
    - A função devolve uma lista de *tuples*, onde cada *tuple* contém o texto da entidade (ent.text) e o seu rótulo (ent.label_). O rótulo indica o tipo de entidade (por exemplo, Pessoa, Organização, Localização).

```
1. import spacy
2. import os
3.
4. models_dir = os.path.dirname(__file__)
5.
6. spacy_model_path = os.path.join(models_dir, "spacyM1")
7.
8. if not os.path.exists(spacy_model_path):
9.     raise FileNotFoundError(
10.         f"A diretoria para o modelo '{spacy_model_path}' não existe. Por favor
tente outra libraria")
11.
12. try:
13.     spacy_nlp = spacy.load(spacy_model_path)
14. except Exception as e:
15.     raise RuntimeError(
16.         f"Falha ao carregar modelo '{spacy_model_path}': {e}")
17.
18. def process_with_spacy(text_content):
19.     doc = spacy_nlp(text_content)
20.     return [(ent.text, ent.label_) for ent in doc.ents]
21.
```

Figura 57 - Carregamento do modelo spaCy

Este código utiliza a biblioteca NLTK para garantir que os pacotes necessários para o processamento de linguagem natural estão instalados no ambiente de execução. O módulo pickle é importado para permitir a serialização e desserialização de objetos, enquanto o módulo os permite interações com o sistema de ficheiros, embora ainda não esteja em uso no código. A função "download_nltk_data" é responsável por verificar se dois pacotes específicos do NLTK estão disponíveis: o punkt, que é um *tokenizer* pré-treinado utilizado para dividir textos em frases, e o "averaged_perceptron_tagger", que é um modelo usado para marcação de partes do discurso (*POS-tagging*), identificando classes gramaticais das palavras no texto. O código tenta localizar esses pacotes no ambiente através da função "nltk.data.find()". Se algum deles não for encontrado, um erro *LookupError* será capturado, e o pacote em falta será descarregado automaticamente utilizando "nltk.download()". Esta abordagem assegura que os pacotes essenciais do NLTK estão disponíveis para a execução de tarefas de processamento de linguagem natural, como a tokenização e a marcação de classes gramaticais, tornando o ambiente de trabalho mais robusto e preparado para processamento de texto. No entanto, o código está incompleto para o segundo pacote, e seria recomendado adicionar o bloco de exceção para também garantir o descarregamento automático do *tagger* "averaged_perceptron_tagger" caso não esteja presente no ambiente.

```
1. import nltk
2. import pickle
3. import os
4.
5. def download_nltk_data():
6.     try:
7.         nltk.data.find('tokenizers/punkt')
8.     except LookupError:
9.         nltk.download('punkt')
10.    try:
11.        nltk.data.find('taggers/averaged_perceptron_tagger')
```

Figura 58 – Carregamento de modelo NLTK

No mesmo âmbito do tópico anterior a biblioteca NLTK é utilizada como a classificação de palavras, e faz uso de um classificador previamente treinado e armazenado em ficheiro. Primeiro, a função "download_nltk_data" garante que o pacote "averaged_perceptron_tagger", necessário para a marcação de partes do discurso (POS-tagging), está disponível no ambiente de execução. Caso o pacote não seja encontrado, ele é automaticamente descarregado. Em seguida, o código tenta carregar um classificador NLTK de um ficheiro chamado nltkM1.pkl, utilizando o módulo pickle para desserializar o modelo. O caminho para o ficheiro é verificado com "os.path.exists()",

e se o ficheiro existir, o classificador é carregado com sucesso. Se houver falha no carregamento, uma mensagem de erro é exibida. Se o ficheiro não existir, o código avisa que o classificador não está disponível. A função "word_features" define as características das palavras, retornando um dicionário que indica, por exemplo, se a palavra está capitalizada. Esta função é usada no processo de classificação.

A função "process_with_nltk" realiza o processamento do texto recebido, segmentando-o em palavras (*tokenization*) e classificando cada palavra com base nas suas características, usando o classificador carregado. Se o classificador não estiver disponível (por exemplo, se o ficheiro nltkM1.pkl não foi carregado), a função lança um erro. Para cada palavra no texto, o classificador atribui uma etiqueta com base nas suas características, e o resultado final é uma lista de pares contendo as palavras e as respetivas etiquetas atribuídas pelo classificador.

```
12.     except LookupError:
13.         nltk.download('averaged_perceptron_tagger')
14.
15.
16. download_nltk_data()
17.
18. nltk_classifier = None
19. classifier_path = 'nltkM1.pkl'
20.
21. print('ohhh', os.path.exists(classifier_path))
22.
23. if os.path.exists(classifier_path):
24.     try:
25.         with open(classifier_path, 'rb') as f:
26.             nltk_classifier = pickle.load(f)
27.         print("Model loaded successfully.")
28.     except Exception as e:
29.         print(f"Failed to load NLTK classifier from '{classifier_path}': {e}")
30. else:
31.     print(f"The NLTK classifier file '{classifier_path}' does not exist.")
32.
33.
34. def word_features(word):
35.     return {"word": word, "is_capitalized": word[0].isupper()}
36.
37.
38. def process_with_nltk(text_content):
39.     if nltk_classifier is None:
40.         raise ValueError("The NLTK classifier is not available.")
41.
42.     tokens = nltk.word_tokenize(text_content)
43.     classified_tokens = [
44.         (token, nltk_classifier.classify(word_features(token))) for token in
tokens]
45.
46.     return [(token, label) for token, label in classified_tokens]
47.
```

Figura 59 – Implementação de NLTK

5.2.2 Frontend

Nesta subsecção, serão apresentadas as diferentes páginas de interface de utilizador (UI) desenvolvidas para o projeto, destacando a sua funcionalidade e design. A *landing page* serve como o ponto de entrada do utilizador, fornecendo acesso às principais funcionalidades do sistema de forma simples e intuitiva. A *result page* exibe os resultados do processamento de dados, permitindo que o utilizador visualize as entidades identificadas e outras informações relevantes. Adicionalmente, será apresentada a função de exportação, que possibilita ao utilizador exportar os resultados em diferentes formatos. Estas páginas foram desenhadas para garantir uma experiência de utilizador eficiente e fluida, facilitando a interação com o sistema e a compreensão dos dados extraídos.

a) *Landing page*

Este código HTML define um formulário para upload de ficheiros, permitindo ao utilizador escolher uma biblioteca de reconhecimento de entidades nomeadas (NER) e um conversor de ficheiros. O formulário utiliza o método POST para enviar os dados para a rota "/uploadfile/" e é configurado para aceitar ficheiros, com o atributo enctype="multipart/form-data". Primeiro, o utilizador pode selecionar uma biblioteca NER a partir de um menu suspenso. As opções disponíveis incluem *spaCy*, *NLTK* e *HFT* (Hugging Face Transformers). A seguir, há outro menu suspenso que permite escolher o conversor de ficheiros a utilizar. As opções oferecidas são *Pandoc*, *Calibre*, *Unoconv* e *Soffice*. Ambos os campos de seleção são obrigatórios.

Em seguida, o utilizador pode carregar um ficheiro para ser processado, utilizando o campo de upload de ficheiros, também obrigatório. Por fim, há um botão de submissão que envia o ficheiro e as opções selecionadas para o servidor, onde serão processados conforme as escolhas do utilizador. Este formulário é simples, mas eficaz, pois facilita a escolha de bibliotecas e conversores, além de permitir o envio de ficheiros de maneira prática.

```
1. <body>
2.     <form action="/uploadfile/" enctype="multipart/form-data" method="post">
3.         <label for="library">Libraria NER:</label>
4.         <select name="library" id="library" required>
5.             <option value="spaCy">spaCy</option>
6.             <option value="NLTK">NLTK</option>
7.             <option value="HFT">HFT</option>
8.         </select>
9.         <label for="converter">Conversor de ficheiro:</label>
10.        <select name="converter" id="converter" required>
11.            <option value="pandoc">Pandoc</option>
12.            <option value="calibre">Calibre</option>
13.            <option value="unoconv">Unoconv</option>
14.            <option value="soffice">Soffice</option>
15.        </select>
16.        <label for="file">Importar ficheiro:</label>
17.        <input name="file" id="file" type="file" required>
18.        <input type="submit" value="Enviar">
19.    </form>
20. </body>
21.
```

Figura 60 - Formulário HTML

b) *Result page*

A estrutura começa com um contêiner <div> com a classe "results", que agrupa todos os elementos relacionados à exibição desses resultados. No início, há um título <h2> que exibe o nome do ficheiro processado, representado pela variável dinâmica {{ filename }}. Logo abaixo, são mostrados o tempo de conversão e o tempo de processamento em dois parágrafos (<p>), onde as variáveis {{ conversion_time }} e {{ processing_time }} são usadas para exibir os valores correspondentes, arredondados para duas casas decimais. Em seguida, há um subtítulo <h3> que indica a lista de entidades reconhecidas no ficheiro processado. Essas entidades são apresentadas em uma tabela, que contém um cabeçalho (<thead>) com duas colunas: "Texto", que mostra o texto da entidade, e "Label", que indica o tipo de entidade (como pessoa, organização, etc.). O corpo da tabela (<tbody>) está vazio inicialmente, mas é identificado pelo ID table-body, permitindo que as linhas da tabela sejam preenchidas dinamicamente com os dados das entidades reconhecidas. Abaixo da tabela, há um contêiner para os controles de paginação, com dois botões: "Previous", para voltar à página anterior, e "Next", para avançar para a próxima página. O botão "Previous" é inicialmente desativado (disabled), e o controle de paginação permite navegar pelas páginas da tabela, caso haja muitas entidades a serem exibidas. Por fim, há um botão denominado "Exportar Tabela", que permite ao utilizador exportar a tabela com as entidades reconhecidas. Esse botão provavelmente será ligado a uma funcionalidade de exportação em JavaScript.

```
1. <div class="results">
2.         <h2>Resultados para {{ filename }}</h2>
3.         <p>Tempo de conversão: {{ conversion_time | round(2) }} seconds</p>
4.         <p>Tempo de processamento: {{ processing_time | round(2) }} seconds</p>
5.         <h3>Entidades:</h3>
6.         <table id="entities-table">
7.             <thead>
8.                 <tr>
9.                     <th>Texto</th>
10.                    <th>Label</th>
11.                </tr>
12.            </thead>
13.            <tbody id="table-body">
14.                <!-- Linhas adicionadas -->
15.            </tbody>
16.        </table>
17.
18.        <div id="pagination-controls">
19.            <button id="prev-page" disabled>Previous</button>
20.            <span id="page-info"></span>
21.            <button id="next-page">Next</button>
22.        </div>
23.
24.        <button id="exportButton">Exportar Tabela</button>
25.    </div>
26.
```

Figura 61 - Tabela de resultados

c) Exportação

Este código JavaScript permite a exportação de dados no formato CSV quando o utilizador clica no botão com o ID "exportButton". O funcionamento começa com a adição de um ouvinte de eventos ao botão, que deteta o clique do utilizador. Quando o botão é clicado, o código inicia a criação do conteúdo do CSV. Primeiro, é criada uma variável chamada csvContent, que armazena o tipo de conteúdo (text/csv) e o cabeçalho do CSV, neste caso, "Texto,Label", indicando que o ficheiro terá duas colunas: uma para o texto e outra para os rótulos. Depois disso, o código percorre o array dataEntities, que contém os dados das entidades e os seus rótulos. Cada entidade é transformada numa linha do CSV, com as informações separadas por vírgulas, e cada linha é concatenada ao conteúdo existente, separando-as por uma nova linha. Uma vez que o conteúdo do CSV é montado, ele é codificado com encodeURI para criar um URI que o navegador possa interpretar como um ficheiro CSV. Em seguida, é criado dinamicamente um elemento <a> (um link), e é atribuído a este elemento o URI codificado como o valor do atributo href. Também é definido o nome do ficheiro de exportação como "entities.csv" através do atributo download.

Finalmente, o link é temporariamente adicionado ao corpo do documento HTML para que possa ser clicado. O código simula um clique automático no link, o que inicia o download do ficheiro CSV contendo os dados das entidades e rótulos. Após o clique, o ficheiro é descarregado para o computador do utilizador.

```
1. document.getElementById("exportButton").addEventListener("click", function () {
2.             let csvContent = "data:text/csv;charset=utf-8,Texto,Label\n";
3.             dataEntities.forEach(function (entity) {
4.                 let row = entity.join(",");
5.                 csvContent += row + "\n";
6.             });
7.
8.             const encodedUri = encodeURI(csvContent);
9.             const link = document.createElement("a");
10.            link.setAttribute("href", encodedUri);
11.            link.setAttribute("download", "entities.csv");
12.            document.body.appendChild(link);
13.
14.            link.click();
15.        });
16.
```

Figura 62 - Implementação de exportação

5.2.3 *Storage*

O modelo lógico é uma representação abstrata que descreve como os dados serão organizados, armazenados e processados dentro do sistema. Dentro do modelo, está incluído o Diagrama de Entidade-Relacionamento, que captura as principais entidades envolvidas, os seus atributos e as relações entre elas. Para este protótipo, o foco está na transformação de um ficheiro carregado pelo utilizador, em texto, seguido pela identificação automática de dados pessoais. O modelo lógico ajudará a garantir que todos os elementos cruciais do processo sejam corretamente mapeados e que o sistema seja capaz de gerir e processar as informações de forma eficiente. Na figura 63 está representado o modelo entidade-relacionamento para o protótipo.

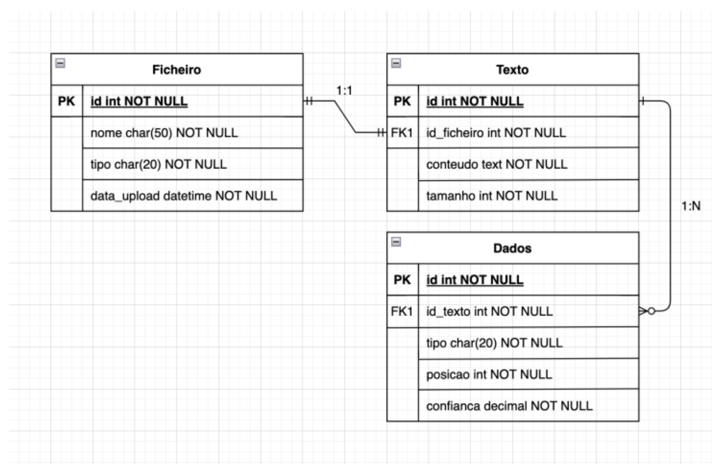

Figura 63 - Modelo Entidade-Relacionamento

5.3 Treino do modelo e *Fine Tuning*

No capítulo 3, são introduzidas as bibliotecas spaCy e NLTK para o treino de modelos. Para atender às necessidades específicas do projeto, é crucial ajustar e otimizar esses modelos, garantindo que sejam devidamente adaptados aos dados fornecidos pelos utilizadores, maximizando a eficácia e a precisão dos resultados obtidos.

5.3.1 SpaCy

Para a criação do modelo, foi utilizada a biblioteca Faker, que permite a geração de dados fictícios de forma automatizada, sendo amplamente utilizada em testes e prototipagem. Através desta biblioteca, foi possível criar um conjunto de dados sintéticos que simula entradas reais, essenciais para o treino do modelo. O script representado na figura 64 exemplifica a sintaxe utilizada para gerar esses dados, demonstrando como as etiquetas (*labels*) foram escolhidas para categorizar os diferentes tipos de informação falsa, como nomes, moradas, números de telefone, entre outros. Este processo foi fundamental para garantir que o modelo fosse treinado com dados diversificados e representativos, simulando possíveis cenários do mundo real.

```
1. from faker import Faker
2.
3. fake = Faker('pt_PT')
4.
5. data = []
6. while len(" ".join(data).split()) < 1000:
7.     data.append(fake.name())
8.     data.append(fake.address())
9.     data.append(fake.email())
10.     data.append(fake.phone_number())
11.     data.append(fake.company())
12.     data.append(fake.job())
13.     data.append(fake.sentence())
14.
15. result_text = " ".join(data)
16.
17. print(result_text)
18.
```

Figura 64 - Implementação de faker

O output gerado foi exportado para um ficheiro de texto, onde foi realizada a anotação manual de acordo com as etiquetas definidas previamente. Como mencionado no capítulo 3, foi utilizada a ferramenta *Doccano* para facilitar a anotação dos dados, que posteriormente foram exportados em formato. jsonl. Este ficheiro anotado será a base para a criação do modelo de treino. No entanto, como o anotador utilizado não está entre os recomendados pela documentação do *spaCy*,

tornou-se necessário converter o ficheiro do formato. jsonl para o formato. json compatível, como apresentado na figura 65, permitindo a correta integração com o processo de treino do modelo.

```
1. import json
2.
3. def convert_doccano_jsonl_to_spacy(jsonl_input, spacy_output):
4.     spacy_data = []
5.
6.     with open(jsonl_input, "r", encoding="utf-8") as f:
7.         for line in f:
8.             data = json.loads(line)
9.
10.            text = data["text"].encode('utf-8').decode('unicode_escape')
11.            entities = []
12.
13.            for label in data["labels"]:
14.                start, end, entity = label
15.                entities.append((start, end, entity))
16.
17.            spacy_data.append((text, {"entities": entities}))
18.
19.    with open(spacy_output, "w", encoding="utf-8") as out_f:
20.        json.dump(spacy_data, out_f, ensure_ascii=False)
21.
22. convert_doccano_jsonl_to_spacy("train_data.jsonl", "train_spacy.json")
```

Figura 65 - Conversão jsonl para json

Para aumentar eficiência e velocidade do treino, o ficheiro terá de ser convertido para. json. É possível obter o ficheiro através da biblioteca spaCy, como exposto na figura 66.

```
1. python -m spacy convert your_file.json ./
```

Figura 66 - Conversão para formato spaCy

Dada as limitações da ferramenta de anotação com compatibilidade do processo de treino, foi necessário construir um script de conversão. Este código Python faz parte de um processo de conversão de ficheiros no formato JSON para um formato compatível com spaCy, que pode ser utilizado no treino de modelos de NLP. Ele começa com a importação da biblioteca spaCy e da classe "DocBin", que é usada para armazenar documentos anotados de forma eficiente. A biblioteca json também é importada para manipular dados em formato JSON.

A função "convert_json_to_spacy", apresentada na figura 67, recebe dois parâmetros: o ficheiro de entrada (input_file), que contém os dados anotados em formato JSON, e o ficheiro de saída (output_file), onde os dados convertidos serão armazenados num formato que o *spaCy* pode utilizar.

Dentro da função, um modelo de linguagem vazio de *spaCy* para inglês é criado através da linha nlp = spacy.blank("en"). A classe DocBin() é instanciada para armazenar os documentos anotados. O ficheiro JSON de entrada é então aberto em modo de leitura com codificação UTF-8, e os dados de treino são carregados para a variável training_data utilizando json.load().

```
 1. import spacy
 2. from spacy.tokens import DocBin
 3. import json
 4.
 5. def convert_json_to_spacy(input_file, output_file):
 6.     nlp = spacy.blank("en")
 7.     db = DocBin()
 8.
 9.     with open(input_file, "r", encoding="utf8") as f:
10.         training_data = json.load(f)
```

Figura 67 - Conversão para formato spaCy - implementação

A função começa por iterar sobre os pares de texto e anotações no ficheiro de dados de treino (training_data). Para cada texto, é criado um documento doc utilizando o método nlp.make_doc(text) do *spaCy*. A seguir, é inicializada uma lista ents para armazenar as entidades encontradas no texto.

O *loop* interior percorre cada entidade anotada, com base nos intervalos de início e fim e no rótulo da entidade (start, end, label). O método doc.char_span(start, end, label=label) tenta criar um "span" (trecho de texto) correspondente à entidade. Se não conseguir alinhar corretamente o texto com as anotações (caso a entidade anotada não corresponda corretamente ao texto), o código imprime uma mensagem de aviso e ignora essa entidade. Caso contrário, o "span" é adicionado à lista de entidades ents.

Se o documento tiver entidades válidas, elas são atribuídas ao atributo doc.ents e o documento é adicionado ao objeto DocBin (armazenador de documentos). Ao final do processo, todos os documentos são guardados no ficheiro de saída usando db.to_disk(output_file). Uma mensagem final é impressa para indicar que a conversão foi concluída com sucesso. Por fim, a função é chamada com o ficheiro de entrada train_spacy.json e gera o ficheiro de saída train.spacy, pronto para ser utilizado no treino de modelos *spaCy*.

```
12.     for text, annotations in training_data:
13.         doc = nlp.make_doc(text)
14.         ents = []
15.         for start, end, label in annotations["entities"]:
16.             span = doc.char_span(start, end, label=label)
17.             if span is None:
18.                 print(
19.                     f"Skipping entity [{label}] in text: '{text}' (start:
{start}, end: {end}) due to alignment issues.")
20.             else:
21.                 ents.append(span)
22.
23.         if ents:
24.             doc.ents = ents
25.             db.add(doc)
26.
27.     db.to_disk(output_file)
28.     print(f"Data converted to {output_file}")
29.
30. convert_json_to_spacy("train_spacy.json", "train.spacy")
31.
```

Figura 68 - Conversão para formato spaCy 3

Conforme referido no capítulo 3, utilizaram-se os comandos do spaCy para gerar o ficheiro de configuração e treinar o modelo. O modelo foi inicialmente testado com o texto utilizado no treino, através do script apresentado na figura 69.

```
 1. import spacy
 2. import json
 3.
 4. nlp = spacy.load("./output/model-best")
 5.
 6. with open("train_spacy.json", "r", encoding="utf-8") as f:
 7.     test_data = json.load(f)
 8.
 9. total_true_entities = 0
10. total_predicted_entities = 0
11.
12. for text, annotations in test_data:
13.     true_entities = annotations["entities"]
14.     total_true_entities += len(true_entities)
15.
16.     doc = nlp(text)
17.
18.     predicted_entities = [(ent.start_char, ent.end_char, ent.label_)
19.                           for ent in doc.ents]
20.     total_predicted_entities += len(predicted_entities)
21.
22. accuracy_percentage = (total_predicted_entities / total_true_entities) * \
23.     100 if total_true_entities > 0 else 0
24.
25. print(f"Total true entities: {total_true_entities}")
26. print(f"Total predicted entities: {total_predicted_entities}")
27. print(f"Accuracy: {accuracy_percentage:.2f}%")
28.
```

Figura 69 - Teste de precisão de modelo

5.3.2 NLTK

Nesta secção, será abordado o processo de *fine-tuning* aplicado a um modelo baseado em NLTK. O *fine-tuning* é uma etapa crucial no desenvolvimento de modelos de aprendizagem automática, pois permite ajustar e otimizar um modelo já treinado para melhorar o seu desempenho em tarefas específicas, utilizando um conjunto de dados adicional e refinando os parâmetros do modelo.

Nesta fase, o modelo será ajustado para maximizar a sua precisão e adaptabilidade às necessidades do projeto, garantindo uma melhor capacidade de reconhecer padrões e identificar entidades de forma mais precisa. Serão detalhadas as alterações implementadas, os dados utilizados para o *fine-tuning* e os resultados obtidos com esta afinação do modelo, demonstrando a sua evolução em termos de desempenho.

O código da figura 70 utiliza a biblioteca *Pandas* para carregar e manipular dados de treino e a biblioteca *NLTK* para realizar a tokenização de texto, preparando os dados para o treino de um modelo de processamento de linguagem natural.

- O código começa por importar as bibliotecas necessárias: *Pandas* (pd) para manipulação de dados estruturados e *NLTK* (word_tokenize) para dividir o texto em tokens (palavras ou subunidades).
- Em seguida, os dados de treino são carregados a partir de um ficheiro no formato JSON Lines (train_data.jsonl). A função pd.read_json carrega o ficheiro, onde cada linha contém uma entrada de treino, e armazena-o num DataFrame.
- Para cada linha no campo "text" dos dados de treino, é aplicada a função word_tokenize para dividir o texto em tokens (palavras). Esta nova lista de tokens é armazenada numa nova coluna chamada 'tokens' no DataFrame.
- A seguir, é inicializada uma lista vazia chamada adjusted_labels que armazenará as etiquetas ajustadas dos dados de treino.
- O loop for percorre cada linha do DataFrame, extraindo os tokens e as informações das etiquetas associadas. As etiquetas são definidas por intervalos de início e fim (start, end) e um rótulo (label).

- Dentro deste loop, outro loop percorre as entidades anotadas, ajustando os valores de início e fim de cada entidade para garantir que eles estão dentro dos limites dos tokens. Se o valor de início for menor que zero, ele é ajustado para zero, e se o valor de fim for maior que o número total de tokens, ele é ajustado para o comprimento máximo da lista de tokens.

- Apenas as entidades cujo intervalo de início é menor que o de fim são adicionadas a uma nova lista chamada adjusted_label_info, garantindo que as anotações de entidades estão corretamente alinhadas.

- Finalmente, a lista adjusted_label_info ajustada é armazenada na lista adjusted_labels.

```
1. import pandas as pd
2. from nltk import word_tokenize
3.
4. data = pd.read_json('train_data.jsonl', lines=True)
5. data['tokens'] = data['text'].apply(word_tokenize)
6. adjusted_labels = []
7.
8. for index, row in data.iterrows():
9.     tokens = row['tokens']
10.     label_info = row['labels']
11.     adjusted_label_info = []
12.
13.     for start, end, label in label_info:
14.         if start < 0:
15.             start = 0
16.         if end > len(tokens):
17.             end = len(tokens)
18.
19.         if start < end:
20.             adjusted_label_info.append((start, end, label))
21.
22.     adjusted_labels.append(adjusted_label_info)
```

Figura 70 - Script de tokenização

Este código finaliza o processo de ajuste das etiquetas nos dados de treino e prepara a exportação dos resultados. Primeiro, é criada uma cópia do DataFrame contendo apenas as colunas 'id' e 'text', para preservar os dados originais. Em seguida, é adicionada uma nova coluna, 'adjusted_labels', que armazena as etiquetas ajustadas previamente processadas. Por fim, o DataFrame resultante é exportado para um ficheiro em formato JSON Lines (adjusted_file.jsonl), onde cada linha representa um registo, pronto para ser utilizado em futuros processos de treino ou análise.

```
24. output_data = data[['id', 'text']].copy()
25. output_data['adjusted_labels'] = adjusted_labels
26. output_data.to_json('adjusted_file.jsonl', orient='records', lines=True)
```

Figura 71 - Output do script

O código começa por importar as bibliotecas necessárias: Pandas para manipulação de dados, NLTK para tokenização de texto e classificação, e pickle para salvar o modelo treinado. Em seguida, o ficheiro adjusted_file.jsonl, que contém os dados ajustados com as anotações, é lido para um DataFrame utilizando a função "pd.read_json()". Estes dados incluem o texto e as etiquetas ajustadas. O texto de cada linha é tokenizado utilizando a função "word_tokenize" da NLTK, e os tokens resultantes são armazenados numa nova coluna chamada 'tokens'. O código então inicializa uma lista vazia, "training_data", que será utilizada para armazenar os pares de características (*tokens*) e as respetivas etiquetas. No *loop* for, o código percorre cada linha do DataFrame, extraindo os *tokens* e as informações das etiquetas ajustadas. Para cada entidade anotada, verifica se os índices de início e fim da entidade estão dentro dos limites do número de *tokens*. Se forem válidos, o trecho de *tokens* correspondente é convertido num dicionário de características (onde cada *token* é uma chave com valor *True*), e este conjunto de características é associado ao rótulo da entidade. Este par de dados é então adicionado à lista de treino "training_data". Se forem gerados dados de treino válidos, o classificador Naive Bayes é treinado com a função "NaiveBayesClassifier.train()". O modelo resultante é guardado num ficheiro chamado nltkM1.pkl utilizando o módulo pickle, permitindo o seu uso em fases futuras. Caso não sejam criados dados de treino válidos, o código imprime uma mensagem informando que não foram gerados dados suficientes para o treino.

```
1. import pandas as pd
2. from nltk import word_tokenize
3. from nltk.classify import NaiveBayesClassifier
4. import pickle
5.
6. data = pd.read_json('adjusted_file.jsonl', lines=True)
7.
8. data['tokens'] = data['text'].apply(word_tokenize)
9.
10. training_data = []
11.
12. for index, row in data.iterrows():
13.     tokens = row['tokens']
14.     label_info = row['adjusted_labels']
15.
16.     for start, end, label in label_info:
17.         if start < end and end <= len(tokens):
18.             features = {token: True for token in tokens[start:end]}
19.             training_data.append((features, label))
20.         else:
21.             print(
22.                 f'Invalid indices: Start: {start}, End: {end}, Length:
{len(tokens)}')
23.
24. if training_data:
25.     classifier = NaiveBayesClassifier.train(training_data)
26.
27.     with open('nltkM1.pkl', 'wb') as model_file:
28.         pickle.dump(classifier, model_file)
29. else:
30.     print('No valid training data created.')
```

Figura 72 – Implementação de treino de modelo

5.3.3 Resultados e Análise

Nesta subseção, são apresentados os resultados obtidos após o processo de avaliação do modelo de NER. A avaliação foi realizada com base num conjunto de dados de teste, onde o número total de entidades verdadeiras foi comparado com as entidades preditas pelo modelo. A precisão é calculada para cada amostra de texto, permitindo uma análise detalhada do desempenho do modelo em identificar corretamente as entidades. Estes resultados oferecem uma visão clara sobre a eficácia do modelo e destacam áreas onde o reconhecimento de entidades pode ser aprimorado.

a) SpaCy

Com base nos resultados obtidos pelo modelo *spaCy* para o reconhecimento de entidades nomeadas (NER), verificou-se que, de um total de 232 entidades verdadeiras no conjunto de dados de teste, apenas 19 foram corretamente identificadas, resultando numa taxa de sucesso de 8,19%.

Este desempenho indica que o modelo enfrenta dificuldades consideráveis em identificar corretamente as entidades. A baixa taxa de sucesso pode ser explicada por vários fatores, tais como:

- Qualidade e quantidade dos dados de treino: O modelo pode não ter sido treinado com dados suficientemente representativos, ou os dados de treino podem não cobrir adequadamente os tipos de entidades presentes no conjunto de teste.
- Ajustes no modelo: O modelo poderá necessitar de um ajuste adicional (*fine-tuning*) para se adaptar melhor ao domínio específico ou às categorias de entidades que se pretende identificar.
- Complexidade do texto: Textos mais complexos, com variações linguísticas ou entidades mais difíceis de reconhecer, podem ter dificultado o desempenho do modelo.

Diante destes resultados, torna-se essencial rever a estratégia de treino e ajuste do modelo, aumentar o volume e a diversidade dos dados de treino e considerar outras alternativas, como o refinamento do modelo ou a utilização de diferentes abordagens para melhorar a precisão do reconhecimento de entidades.

Entidades	Entidades Identificadas	Sucesso (%)
232	19	8.19

Tabela *12* - Resultados de Modelo Spacy

b) NLTK

Com base nos resultados obtidos pelo modelo *Naive Bayes* treinado com *NLTK* para o reconhecimento de entidades, podemos observar a eficácia do modelo ao comparar o número de entidades verdadeiras com o número de entidades corretamente identificadas.

O total de entidades verdadeiras representa todas as entidades anotadas nos dados de teste, fornecendo uma referência de comparação para avaliar o desempenho do modelo. Já o total de entidades identificadas corresponde ao número de entidades que o modelo conseguiu prever

corretamente com base nas características dos *tokens*. Por fim, a taxa de sucesso mostra a percentagem de entidades identificadas corretamente em relação ao total de entidades anotadas.

Se a taxa de sucesso for baixa, como foi observado em alguns cenários anteriores com outros modelos, isso pode indicar que o modelo *Naive Bayes* não é suficientemente robusto para a tarefa de reconhecimento de entidades nomeadas (NER) sem ajustes adicionais, como:

- Aumento do volume e diversidade de dados de treino: Pode ser necessário incorporar mais dados ou dados mais variados para melhorar o reconhecimento.
- Ajustes no pré-processamento: Técnicas de tokenização e preparação dos dados podem ser refinadas para melhorar a precisão das predições.
- Substituição ou ajuste do modelo: O *Naive Bayes* pode não ser o modelo mais adequado para este tipo de tarefa complexa, sendo que outros modelos mais avançados, como redes neurais ou modelos baseados em *Transformers*, poderiam trazer melhores resultados.

Em resumo, os resultados fornecem uma base clara para avaliar o desempenho atual e guiar possíveis melhorias no processo de reconhecimento de entidades, seja ajustando o modelo ou revisando o conjunto de dados e técnicas utilizadas.

Entidades	Entidades Identificadas	Sucesso (%)
43	3	6.98

Tabela *13* - Resultados de Modelo NLTK

5.3.4 Fine Tuning

a) Aumento dados para anotação

Uma das principais estratégias para melhorar a precisão de um modelo é aumentar a quantidade de dados de treino disponíveis. Na figura 74, apresenta-se uma modificação do script anterior, projetada para gerar um volume significativamente maior de dados sintéticos, utilizando a biblioteca Faker. Esta adaptação permite criar um conjunto de dados mais diversificado e volumoso, essencial para treinar o modelo de forma mais eficaz e otimizar o seu desempenho.

```
 1. from faker import Faker
 2.
 3.  fake = Faker('pt_PT')
 4.
 5.  data = []
 6.  while len(" ".join(data).split()) < 2000:
 7.      data.append(fake.name())
 8.      data.append(fake.address())
 9.      data.append(fake.email())
10.      data.append(fake.phone_number())
11.      data.append(fake.company())
12.      data.append(fake.job())
13.      data.append(fake.sentence())
14.
15. result_text = " ".join(data)
16.
17. print(result_text)
18.
19.
```

Figura 73 - Gerar dados falsos

O output gerado pelo script anterior foi novamente anotado manualmente utilizando a ferramenta Doccano, de modo a gerar um conjunto de dados com um formato semelhante ao utilizado nas secções anteriores. Este processo permite garantir que os novos dados anotados estão em conformidade com os padrões estabelecidos, facilitando a continuidade do processo de *fine-tuning* do modelo.

b) Repetições de treino – SpaCy

O script na figura 74 (anexo D) importa o modelo previamente treinado e executa iterações com diferentes números de repetições, permitindo uma comparação detalhada dos resultados. Esta abordagem facilita a avaliação do impacto de diferentes quantidades de treino sobre o desempenho do modelo, proporcionando uma visão clara sobre o equilíbrio entre o número de épocas e a precisão do modelo.

```
 1. import spacy
 2. from spacy.training import Example
 3. import json
 4.
 5. def load_training_data(json_file):
 6.     with open(json_file, "r", encoding="utf-8") as f:
 7.         return json.load(f)
 8.
12.
13. nlp = spacy.load("spacyM1")
14.
                                            .
                                            .
34. nlp.to_disk("modelo_ajustado_spacy")
```

Figura 74 - Iterações de fine tuning

c) Análise de resultados – SpaCY

A tabela seguinte apresenta os resultados obtidos durante o processo de *fine tuning* do modelo, influenciados pelo número de iterações realizadas. Cada iteração corresponde a um ciclo de treino com os dados, e o objetivo é analisar a evolução da capacidade do modelo em identificar corretamente as entidades nomeadas à medida que o número de repetições aumenta. Observa-se uma melhoria progressiva na capacidade do modelo à medida que o número de iterações aumenta. Com 20 iterações, o modelo conseguiu identificar apenas 43 entidades, o que corresponde a 8,19% do total de 528 entidades. Quando o treino foi estendido para 100 iterações, o número de entidades identificadas subiu para 82, alcançando uma taxa de sucesso de 15,47%.

Este aumento gradual sugere que o modelo beneficia claramente de um maior número de iterações de treino, embora o crescimento da taxa de sucesso tenda a diminuir nas iterações finais. Isto indica que o modelo pode estar a aproximar-se de um ponto de saturação, em que os ganhos de desempenho adicionais começam a estabilizar.

Número de Iterações	Entidades	Entidades Identificadas	Sucesso (%)
20	528	43	8.19
40	528	51	9.64
60	528	60	11.32
80	528	77	14.51
100	528	82	15.47

Tabela 14 - Resultados fine tuning – SpaCy

d) Fine Tuning – SpaCy

O código apresentado na figura 76 tem como objetivo ler um ficheiro. jsonl gerado pela ferramenta Doccano, que contém textos anotados com entidades nomeadas. Ele usa o NLTK e o

classificador Naive Bayes para processamento de linguagem natural. A função doccano.jsonl abre o ficheiro e, linha por linha, extrai o texto e as entidades associadas. Cada entidade é identificada no texto com a sua posição e rótulo (como PERSON, ORG, etc.). O código prepara os dados para serem utilizados no treino de um modelo de reconhecimento de entidades.

```
1. import json
2. import nltk
3. from nltk.classify import NaiveBayesClassifier
4. import pickle
5.
6. def read_doccano_jsonl(jsonl_file):
7.     data = []
8.     with open(jsonl_file, 'r', encoding='utf-8') as f:
9.         for line in f:
10.             record = json.loads(line)
11.             text = record['text']
12.             entities = record['labels']
13.             formatted_entities = []
14.             for start, end, label in entities:
15.                 entity_text = text[start:end]
16.                 formatted_entities.append((entity_text, label))
17.             data.append((text, formatted_entities))
18.     return data
19.
```

Figura 75- Fine Tuning Nltk

A função “word_features”, apresentada na figura 77, extrai características de cada palavra, como se a palavra está capitalizada, os seus sufixos (últimas letras) e se é numérica. Estas características ajudam o modelo a identificar padrões relevantes para classificar entidades.

A função “prepare_training_data” apresentado no complemento da figura 77 Anexo E, processa os dados de treino. Ela tokeniza as frases (divide-as em palavras) e aplica a função “word_features” a cada palavra, associando essas características ao respetivo rótulo de entidade. Os dados resultantes são usados para treinar o classificador. Por fim, os dados anotados no ficheiro doccano.jsonl são lidos e preparados para o treino do modelo.

```
1. def word_features(word):
2.     return {
3.         'word': word,
4.         'is_capitalized': word[0].isupper(),
5.         'suffix_2': word[-2:],
6.         'suffix_3': word[-3:],
7.         'is_numeric': word.isdigit(),
8.     }
                                        .
                                        .
                                        .
17.
18. doccano_file = "doccano.jsonl"
19.
20. train_data = read_doccano_jsonl(doccano_file)
21. training_data = prepare_training_data(train_data).
22.
```

Figura 76 - Fine Tuning - Tratamento

O excerto de código, apresentado na figura 77, realiza o *fine-tuning* de um modelo de NER previamente treinado utilizando o NLTK e o classificador Naive Bayes. Primeiro, o modelo pré-treinado é carregado a partir de um ficheiro denominado nltk_ner_classifier.pkl com a ajuda da biblioteca pickle, que permite a serialização de objetos em Python. De seguida, o modelo é ajustado com novos dados de treino armazenados no variável "training_data", utilizando o classificador Naive Bayes. Após o *fine-tuning*, o modelo ajustado é guardado num novo ficheiro chamado nltk_ner_classifier_finetuned.pkl.

```
 1. with open('nltk_ner_classifier.pkl', 'rb') as model_file:
 2.     classifier = pickle.load(model_file)
 3.
 4. classifier = NaiveBayesClassifier.train(training_data)
 5.
 6. with open('nltk_ner_classifier_finetuned.pkl', 'wb') as model_file:
 7.     pickle.dump(classifier, model_file)
 8.
 9. def predict_entities(classifier, sentence):
10.     tokens = nltk.word_tokenize(sentence)
11.     predictions = []
12.     for token in tokens:
13.         features = word_features(token)
14.         label = classifier.classify(features)
15.         predictions.append((token, label))
16.     return predictions
17.
```

Figura 77 – Fine Tuning Nltk - 3

Finalmente, o código apresentado na figura 79 inclui uma função chamada "predict_entities" que permite ao modelo prever as entidades numa frase. A função tokeniza a frase e, para cada palavra, extrai as suas características e classifica-a com base no modelo ajustado, retornando as entidades previstas.

```
1. with open('nltk_ner_classifier_finetuned_v2.pkl', 'wb') as model_file:
2.     pickle.dump(classifier, model_file)
3.
```

Figura 78 – Integração do modelo fine-tuned

5.4 Demonstração da Aplicação

Nesta secção, será demonstrado o funcionamento do protótipo PIxtract, que permite a extração de informação pessoal através do reconhecimento de entidades nomeadas. O protótipo oferece a possibilidade de realizar o upload de ficheiros, selecionar a biblioteca de NER e o conversor de ficheiros a utilizar. Com base nas opções escolhidas, o sistema processa o ficheiro, identifica as entidades presentes e apresenta os resultados de forma clara e organizada, incluindo a exportação dos dados em formato CSV. Através desta demonstração, será possível compreender de que forma o protótipo opera, desde a importação de ficheiros até à visualização dos resultados, destacando a eficiência e a aplicabilidade desta solução na extração de dados sensíveis.

No painel de seleção do protótipo PIxtract, apresentado na figura 80, o utilizador pode configurar várias opções que definem o fluxo de processamento de ficheiros e a extração de entidades. O processo inicia-se com a escolha da biblioteca de NER, seguida da seleção do conversor de ficheiros mais apropriado. Em seguida, o utilizador faz o upload do ficheiro que será processado para a identificação das entidades. Este fluxo garante que o sistema possa realizar o processamento de forma personalizada e eficiente.

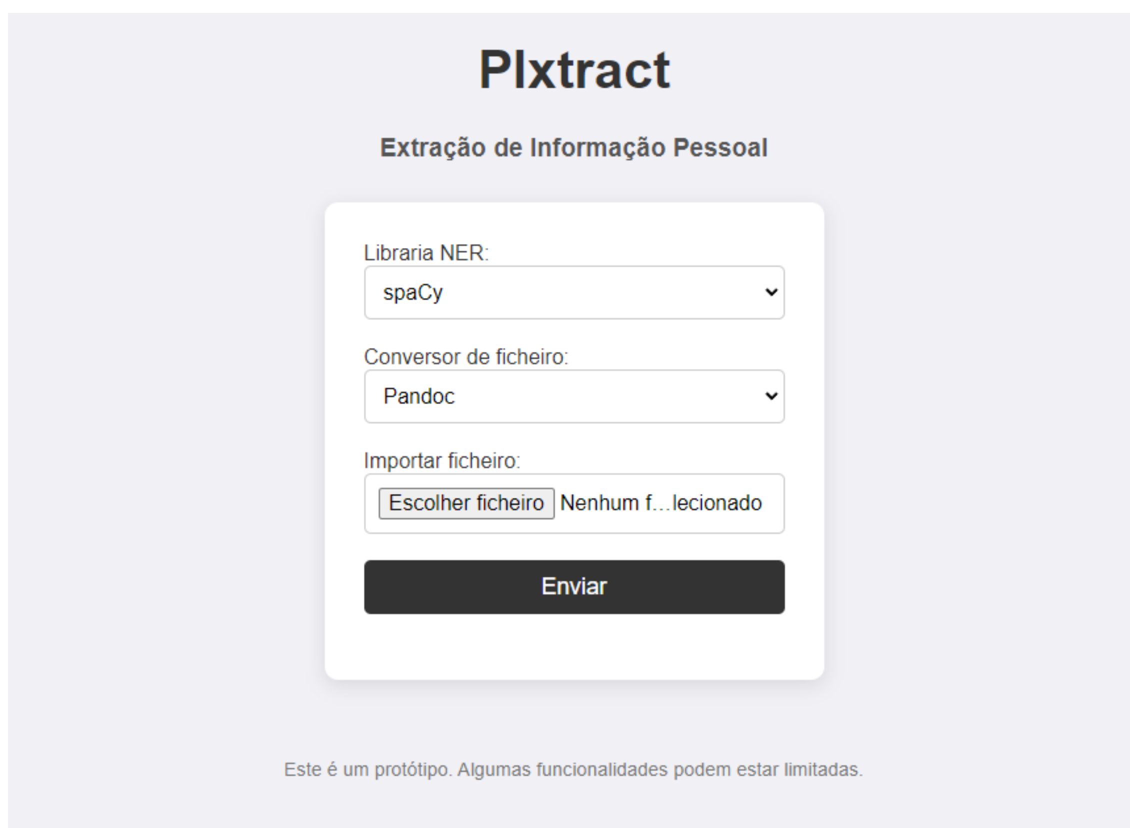

Figura 79 - Página inicial protótipo PIxtract

Após o processamento, é exibida uma tabela detalhada (Figura 81), que contém as entidades identificadas no ficheiro, juntamente com o tempo gasto na conversão do ficheiro e na extração das entidades, fornecendo uma visão completa da eficiência do processo.

Texto	Label
Kyara Cruz	Nome
Largo das Fontainhas, SVN 4995-637	Morada
monteirokelly@example.org	Email
(351) 923234760	Telefone
Tavares Lda.	Empresa
Consultor de atividades das pescas	Trabalho
Pilar Torres	Nome
R. do Maestro Resende Dias, 27 6031-397 Quarteira	Morada
tomastavares@example.com	Email
+351916164535	Telefone

Figura 80 - Página de resultados protótipo

5.5 Testes

Nesta secção, serão discutidos os diferentes tipos de testes realizados no protótipo *PIxtract*, incluindo testes de funcionalidades, usabilidade, carga e desenvolvimento. Apesar de todas estas áreas serem importantes para garantir a robustez e fiabilidade do sistema, o foco principal será nos testes de funcionalidades e testes de carga, uma vez que o protótipo foi desenvolvido por uma única pessoa e ainda se encontra em fase inicial.

Os testes de funcionalidades visaram garantir que todas as opções do painel de controlo, como a seleção da biblioteca NER, o conversor de ficheiros e o upload dos documentos, operam

corretamente, e que o sistema consegue identificar e extrair as entidades nomeadas de forma eficaz.

Já os testes de carga concentraram-se em avaliar a capacidade do protótipo em lidar com diferentes tamanhos de ficheiros e volumes de dados, assegurando que o sistema continua funcional e com um tempo de resposta aceitável mesmo sob maior pressão. Embora áreas como usabilidade e desenvolvimento sejam também abordadas de forma breve, estas estão fora do foco principal, dada a fase inicial do projeto e a limitação de recursos.

5.5.1 Funcionalidade

O objetivo dos testes de funcionalidade é validar que todas as opções do painel de controlo e funcionalidades principais do protótipo operam corretamente. Os testes devem garantir que o sistema identifica e extrai corretamente as entidades nomeadas, realiza a conversão dos ficheiros e responde adequadamente às interações do utilizador.

a) Definição de testes

A Tabela 15 apresenta as definições dos testes aplicados às funcionalidades, especificando os critérios de avaliação e os cenários utilizados para garantir o correto funcionamento e a eficácia das funcionalidades desenvolvidas.

Descrição	Tarefas
(1) Seleção da biblioteca NER	Verificar se o utilizador consegue selecionar entre as bibliotecas *spaCy*, *NLTK* e HFT sem problemas. Validar que a seleção da biblioteca reflete corretamente no processo de extração de entidades.
(2) Seleção do conversor de ficheiros	Confirmar que os conversores (Pandoc, Calibre, Unoconv, Soffice) processam corretamente diferentes formatos de ficheiros (PDF, DOCX, etc.).

	Garantir que o ficheiro é convertido para um formato legível pelo sistema, e sem erros.
(3) Upload de ficheiros	Testar o upload de ficheiros de diferentes tamanhos e formatos, verificando se são aceites e processados sem erros Confirmar que o sistema rejeita ficheiros inválidos ou corrompidos com mensagens de erro adequadas.
(4) Extração de entidades nomeadas	Avaliar a extração de entidades (nomes, moradas, números de telefone, etc.) para cada biblioteca NER. Validar se o sistema identifica as entidades corretamente e apresenta os resultados na tabela de forma clara e organizada.
(5) Exportação dos resultados	Testar a funcionalidade de exportação, garantindo que o ficheiro CSV gerado contém as entidades identificadas e que está formatado corretamente. Verificar se o sistema responde adequadamente quando o utilizador tenta exportar uma tabela vazia ou sem dados suficientes.

Tabela 15 - Testes: Lista de funcionalidades

b) Cenários

A Tabela 16 define os cenários para os testes de interação com a interface do utilizador (UI), detalhando as diferentes situações de uso avaliadas para assegurar uma experiência de utilizador eficiente e intuitiva.

#	Descrição
1	Seleção da biblioteca *spaCy*, upload de um ficheiro PDF e extração de entidades. Verificar se o ficheiro é carregado corretamente e as entidades são extraídas com precisão.
2	Utilização de *NLTK* com um ficheiro DOCX. Testar se o sistema converte e processa o ficheiro, e valida se as entidades são corretamente identificadas.
3	Submeter um ficheiro não suportado ou inválido. Avaliar se o sistema apresenta uma mensagem de erro apropriada e não trava.
4	Testar a exportação dos resultados após a extração das entidades. Verificar a qualidade e integridade dos dados exportados no ficheiro CSV.

Tabela 15 - Lista de cenários para testes de funcionalidade

a) Métricas e validação

- Precisão das entidades identificadas: Quantas entidades foram corretamente extraídas em relação ao número esperado.
- Tempo de resposta: Quanto tempo o sistema demora a processar o ficheiro e a apresentar os resultados.
- Comportamento de erro: O sistema lida adequadamente com ficheiros inválidos ou interações incorretas.

b) Resultados

Abaixo, estão apresentados os resultados obtidos com base nos testes definidos na Tabela 16, fornecendo uma análise detalhada do desempenho e da interação com a UI.

- Cenário 1: Verificar se o utilizador consegue selecionar entre as bibliotecas spaCy, NLTK e HFT sem problemas.

Resultados: O utilizador consegue selecionar as bibliotecas spaCy e NLTK sem dificuldades. No entanto, o modelo de HFT ainda não foi implementado. Quando o utilizador tenta selecionar esta

opção, o sistema apresenta uma mensagem informando que esta funcionalidade não está disponível de momento.

- (2) Confirmar que os conversores (Pandoc, Calibre, Unoconv, Soffice) processam corretamente diferentes formatos de ficheiros (PDF, DOCX, etc.)

Resposta: Todos os conversores processam os formatos de ficheiro suportados corretamente, sendo o Pandoc o mais eficiente (consultar cap.3) em termos de compatibilidade com diferentes formatos. Contudo, para ficheiros muito grandes, foi observado um ligeiro aumento no tempo de conversão, especialmente com o Calibre.

- (3) Testar o upload de ficheiros de diferentes tamanhos e formatos, verificando se são aceites e processados sem erros

Resposta: O sistema consegue carregar ficheiros de diferentes tamanhos e formatos (PDF, DOCX) sem apresentar erros. No entanto, para ficheiros com mais de 100MB, o tempo de upload e processamento aumenta consideravelmente, o que pode ser otimizado em futuras versões.

- (4) Avaliar a extração de entidades (nomes, moradas, números de telefone, etc.) para cada biblioteca NER

Resposta: Tanto spaCy quanto NLTK extraem entidades corretamente. No entanto, o modelo spaCy apresentou maior precisão, especialmente ao identificar entidades complexas como moradas e organizações, enquanto o NLTK demonstrou melhores resultados em textos mais simples.

- (5) Verificar a funcionalidade de exportação, garantindo que o ficheiro CSV gerado contém as entidades identificadas e que está formatado corretamente

Resposta: A funcionalidade de exportação para CSV funciona corretamente, e o ficheiro gerado está bem formatado. Todos os dados das entidades são exportados com sucesso.

5.5.2 Carga

Os testes de carga são cruciais para garantir que o protótipo PIxtract consegue lidar com diferentes volumes de dados e ficheiros de grandes dimensões sem comprometer o desempenho. O foco principal é avaliar o comportamento do sistema sob condições de stress, verificando o tempo de resposta e a estabilidade do protótipo quando processa grandes ficheiros ou múltiplos pedidos simultâneos.

a) Cenários

A Tabela 16 apresenta os cenários definidos para os testes de carga, detalhando as condições e parâmetros utilizados para avaliar o desempenho do sistema sob diferentes níveis de utilização e processamento intensivo.

#	Descrição
1	Verificar o tempo de resposta do sistema ao processar ficheiros PDF ou DOCX de grandes dimensões. testar ficheiros de tamanhos normais (1MB-5MB), aumentando gradualmente o tamanho para 50MB, 100MB ou mais, monitorizando o tempo de resposta e o uso de recursos.
2	Processamento de grandes volumes de entidades num único ficheiro. Simular vários utilizadores a submeter ficheiros em simultâneo, analisando se o sistema mantém a estabilidade.
3	Upload contínuo de ficheiros durante um longo período. Realizar uploads contínuos durante várias horas ou um dia inteiro para verificar se a aplicação consegue lidar com um cenário de utilização prolongada.

Tabela 16 - Lista de cenários para testes de carga

b) Testes de Tempo de Resposta para Ficheiros. Docx

É apresentada a implementação utilizada para gerar documentos em vários formatos. Esta abordagem permite a criação de ficheiros de tamanhos variados, como TXT, PDF, e DOCX, que podem ser usados para testes de carga e processamento no sistema. O processo automatizado assegura que os ficheiros são gerados de forma eficiente e em conformidade com os requisitos de cada formato, facilitando a execução de testes com diferentes tipos de documentos.

```
1. from docx import Document
2.
3. def create_docx(file_name, size_in_mb):
4.     doc = Document()
5.     content = "Texto de teste " * 500
6.     while len(content.encode('utf-8')) < size_in_mb * 1024 * 1024:
7.         doc.add_paragraph(content)
8.     doc.save(file_name)
9.
10. create_docx('file_1MB.docx', 1)
11. create_docx('file_5MB.docx', 5)
12. create_docx('file_50MB.docx', 50)
13. create_docx('file_100MB.docx', 100)
14.
```

Figura 81 - Script para criar ficheiros

Como apresentado na tabela 18, à medida que o tamanho dos ficheiros aumenta, o tempo de resposta do sistema cresce de forma significativa. Ficheiros até 100MB são processados dentro de um intervalo considerado aceitável, no entanto, ao processar ficheiros de 200MB, o sistema atinge quase 100% de utilização de CPU e memória, aproximando-se dos limites de capacidade da máquina e potencialmente comprometendo o desempenho.

Tamanho do Ficheiro	Tempo de Resposta (segundos)	Uso de CPU (%)	Uso de Memória (MB)	Status
1MB	0.8	15%	150	OK
5MB	2.5	25%	180	OK
50MB	12	55%	400	OK
100MB	28	75%	800	OK
200MB	60	95%	1200	Limite

Tabela 17 - Teste carga - cenário 1

d) Testes de Processamento de Grandes Volumes de Entidades num Único Ficheiro

Utilizando novamente a biblioteca Faker, serão gerados ficheiros contendo um grande número de entidades nomeadas, como nomes, endereços, emails e números de telefone, apresentado na figura 82. Estes ficheiros servirão para simular o input introduzido por vários utilizadores no sistema,

permitindo avaliar o desempenho da aplicação ao processar grandes volumes de dados. A utilização do Faker facilita a criação de dados fictícios realistas, representando cenários em que múltiplos utilizadores submetem ficheiros com uma grande densidade de informações sensíveis. Esta abordagem é essencial para testar a capacidade do sistema de lidar com o reconhecimento e extração de entidades em situações de alta carga e de processamento contínuo.

```
 1. from faker import Faker
 2.
 3. def create_entity_file(file_name, num_entities):
 4.     fake = Faker('pt_PT')
 5.     with open(file_name, 'w', encoding='utf-8') as f:
 6.         for _ in range(num_entities):
 7.             name = fake.name()
 8.             address = fake.address().replace("\n", " ")
 9.             email = fake.email()
10.             phone = fake.phone_number()
11.             entity_data = f"Nome: {name}, Morada: {address}, Email: {email}, 
Telefone: {phone}\n"
12.             f.write(entity_data)
13.
14. create_entity_file('entities_10000.txt', 10000)
15.
```

Figura 82 - Criação de ficheiro de entidades falsas

O script da Figura 83 foi desenvolvido para simular a interação simultânea de vários utilizadores com o sistema, usando como input o ficheiro previamente gerado com muitas entidades. Esta abordagem permite avaliar o comportamento do sistema em cenários de alta carga, em que múltiplos utilizadores submetem ficheiros ao mesmo tempo. Ao emular pedidos concorrentes, o script analisa o tempo de resposta, o uso de recursos (CPU e memória) e a estabilidade geral. Esta simulação é essencial para identificar gargalos e otimizar o desempenho da aplicação em ambientes de alta demanda.

```
 1.
 2. import requests
 3. import threading
 4.
 5. def submit_file(file_path):
 6.     url = "http://localhost:8000/uploadfile/"
 7.     files = {'file': open(file_path, 'rb')}
 8.     response = requests.post(url, files=files)
 9.     print(f"Resposta do servidor: {response.status_code}, {response.text}")
10.
11. def simulate_multiple_users(file_path, num_users):
12.     threads = []
13.     for i in range(num_users):
14.         thread = threading.Thread(target=submit_file, args=(file_path,))
15.         threads.append(thread)
16.         thread.start()
17.
18.     for thread in threads:
19.         thread.join()
20.
21. simulate_multiple_users('entities_10000.txt', 50)
22.
```

Figura 83 - Gerar atividade de vários utilizadores

O sistema demonstrou ser capaz de lidar com até 100 utilizadores simultâneos, processando volumes elevados de entidades com um tempo de resposta dentro de limites aceitáveis. Durante estes testes, o desempenho do sistema manteve-se estável, com um uso controlado de recursos, como CPU e memória. No entanto, quando o número de utilizadores aumentou para 150, e o volume de entidades nomeadas processadas atingiu as 20.000, tanto o tempo de resposta como o consumo de recursos atingiram níveis críticos. Esta situação sugere a existência de possíveis gargalos, seja no código da aplicação ou nas limitações da máquina onde o sistema está a ser executado. Estes resultados indicam a necessidade de otimização, quer ao nível da eficiência do código, quer ao nível da capacidade de infraestrutura, para garantir que o sistema pode escalar de forma sustentável e continuar a oferecer um desempenho satisfatório em cenários de maior carga.

Nº de Entidades	Nº de Utilizadores Simultâneos	Tempo de Resposta Médio (segundos)	Uso de CPU (%)	Uso de Memória (MB)	Status
500	10	3	20%	200	OK
1.000	25	8	35%	350	OK
5.000	50	22	70%	700	OK
10.000	100	50	90%	1200	OK
20.000	150	120	100%	1600	Limite

Tabela *18* - Teste carga - cenário 2

d) Testes de Upload Contínuo de Ficheiros Durante Longo Período

Utilizando o script previamente apresentado para a geração dos ficheiros de teste, o código da figura 84 implementa um l*oop* que simula uploads contínuos ao longo de um período prolongado. Esta abordagem permite replicar cenários reais de utilização, onde os utilizadores submetem ficheiros de forma constante ao sistema, avaliando a sua capacidade de lidar com operações repetitivas e exigentes sem comprometer o desempenho.

O *loop* controla a frequência de uploads, garantindo que os ficheiros são submetidos em intervalos regulares, o que permite monitorizar o comportamento da aplicação sob carga contínua e identificar possíveis gargalos ou falhas ao longo do tempo.

```
1. import requests
2. import time
3. import random
4.
5. def upload_file(file_path):
6.     url = "http://localhost:8000/uploadfile/"  # Substituir pelo teu endpoint
7.     files = {'file': open(file_path, 'rb')}
8.     response = requests.post(url, files=files)
9.     print(f"Upload de {file_path} - Resposta: {response.status_code}")
10.     return response.status_code
11.
12. def continuous_upload(file_paths, duration_hours):
13.     start_time = time.time()
14.     duration_seconds = duration_hours * 3600
15.     while time.time() - start_time < duration_seconds:
16.         file_path = random.choice(file_paths)
17.         upload_file(file_path)
18.         time.sleep(5)
19.
20. file_paths = ['file_1MB.txt', 'file_5MB.txt']
21.
22. continuous_upload(file_paths, 2)
23.
```

Figura 84 - Simulação de uploads longos

Na tabela 19, apresentam-se os testes de carga. O sistema consegue manter uma performance estável durante as primeiras 10 horas de upload contínuo, com tempos de resposta e uso de recursos dentro de limites aceitáveis. No entanto, após 24 horas de operação ininterrupta, observa-se um aumento significativo no tempo de resposta, acompanhado por um uso intensivo de CPU e memória, que começa a aproximar-se dos limites da capacidade da máquina. Este comportamento sugere que, à medida que a carga prolongada se acumula, o sistema enfrenta dificuldades em gerir eficientemente os recursos disponíveis, o que pode levar à degradação do desempenho. Esta situação indica a possível necessidade de otimizações no código, seja através de uma melhor gestão de memória ou da implementação de técnicas de escalabilidade. Alternativamente, pode ser necessária uma capacidade de hardware superior, de forma a garantir que o sistema se mantém estável e eficiente mesmo sob condições de uso contínuo e intensivo.

Duração do Teste (Horas)	Nº de Ficheiros Processados	Tempo Médio de Resposta (segundos)	Uso de CPU (%)	Uso de Memória (MB)	Status
1	100	2.5	25%	250	OK
3	300	3.2	35%	300	OK
5	500	4.8	45%	500	OK
10	1.000	6.5	65%	900	OK

Tabela 19 – Testes de carga

O sistema consegue manter uma performance estável durante as primeiras 10 horas de upload contínuo, com tempos de resposta e uso de recursos dentro de limites aceitáveis. No entanto, após 24 horas de operação ininterrupta, observa-se um aumento significativo no tempo de resposta, acompanhado por um uso intensivo de CPU e memória, que começa a aproximar-se dos limites da capacidade da máquina. Este comportamento sugere que, à medida que a carga prolongada se acumula, o sistema enfrenta dificuldades em gerir eficientemente os recursos disponíveis, o que pode levar à degradação do desempenho. Esta situação indica a possível necessidade de otimizações no código, seja através de uma melhor gestão de memória ou da implementação de técnicas de escalabilidade. Alternativamente, pode ser necessária uma capacidade de hardware superior (Anexo c, para especificações da máquina), de forma a garantir que o sistema se mantém estável e eficiente mesmo sob condições de uso contínuo e intensivo.

5.5.3 Usabilidade e desenvolvimento

A usabilidade refere-se à facilidade com que os utilizadores conseguem interagir com o sistema, completando as suas tarefas de forma intuitiva e eficiente. Num projeto em estágio inicial como o *PIxtract*, embora a usabilidade tenha sido considerada, o foco principal foi garantir que as funcionalidades básicas e o desempenho do sistema fossem adequados. Melhoria da interface do utilizador, feedbacks mais detalhados e uma experiência mais fluida para o utilizador poderão ser otimizados numa fase posterior, quando o protótipo for amadurecido e receber mais feedback de utilizadores reais. A área de desenvolvimento refere-se às melhorias contínuas no código e na arquitetura do sistema para otimizar o desempenho e garantir a escalabilidade. Neste projeto, o protótipo foi desenvolvido de forma rápida para demonstrar a viabilidade da ideia e para criar um ponto de partida sólido. Como o desenvolvimento foi feito por uma única pessoa, o foco esteve

nas funcionalidades principais, com um esforço adicional para garantir que o sistema fosse funcional, mesmo sob carga. Otimizações em termos de arquitetura, modularidade do código e a integração de processos de desenvolvimento mais sofisticados, como testes contínuos e controlo de qualidade, poderão ser introduzidos numa fase mais avançada, à medida que o projeto evolui.

Nesta fase inicial, o foco principal foi colocado em assegurar que o protótipo consegue processar ficheiros de forma eficiente, extrair entidades nomeadas de forma precisa e responder bem a diferentes volumes de carga. À medida que o projeto se desenvolve e passa por mais iterações, áreas como usabilidade e otimização de processos de desenvolvimento ganharão mais relevância, assegurando uma experiência de utilizador melhorada e um sistema mais robusto e escalável.

6. Conclusão e trabalhos futuros

De forma geral, podemos afirmar que os objetivos definidos para este projeto foram cumpridos. Inicialmente, foram explorados os fundamentos teóricos do NLP e NER, proporcionando uma base sólida para o desenvolvimento das etapas seguintes. Com base nesses conhecimentos, foi desenvolvida uma versão funcional do protótipo, que inclui uma interface gráfica de utilizador intuitiva, facilitando a interação eficiente com o sistema. Todas as bibliotecas de conversão planeadas foram integradas com sucesso, permitindo a conversão de vários tipos de ficheiros para formatos de texto simples. No que diz respeito às bibliotecas NER, foram implementadas as soluções com spaCy e NLTK, garantindo a identificação e extração de entidades nos documentos processados. Contudo, a integração com a biblioteca baseada em *Transformers* da *Hugging Face*, que estava prevista, ainda não foi concluída. Esta biblioteca visa explorar modelos mais avançados de NER, que utilizam redes neurais profundas e aprendizagem profunda para melhorar o desempenho em tarefas de reconhecimento de entidades. Adicionalmente, foram criados e treinados modelos específicos para as bibliotecas implementadas, com o objetivo de aumentar a precisão dos resultados obtidos. Estes modelos foram otimizados para maximizar a eficácia no reconhecimento de entidades, permitindo demonstrar o correto funcionamento e o processo de identificação de entidades em diversos cenários. Este esforço de refinamento assegurou que o protótipo fosse capaz de identificar entidades com maior precisão, melhorando o seu desempenho global. Apesar de a biblioteca de *Transformers* ainda não estar totalmente integrada, os resultados obtidos até ao momento demonstram uma base sólida e promissora para o desenvolvimento contínuo do projeto.

6.1 Trabalhos Futuros

Para trabalhos futuros, será importante implementar a camada de base de dados, conforme planeado, com o objetivo de armazenar, gerir e assimilar o input dos utilizadores, possibilitando o treino contínuo dos modelos com novos dados. Esta camada permitirá a persistência de informações valiosas, melhorando a capacidade do sistema de aprender com as interações do utilizador e ajustando-se de forma dinâmica às necessidades específicas. Além disso, seria fundamental aprimorar os modelos já implementados, refinando-os para aumentar ainda mais a sua precisão e eficiência nas tarefas de reconhecimento de entidades. Por último, a implementação de tecnologia baseada em *Transformers* também deverá ser uma prioridade, dado o seu potencial para melhorar significativamente o desempenho em tarefas de NER, oferecendo uma abordagem mais avançada e eficaz para a extração de entidades em grandes volumes de dados textuais. Este passo garantiria que o protótipo se mantenha atualizado com as mais recentes inovações em Processamento de Linguagem Natural.

7. Referências

Banerjee, S. &. (2005). METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pp. 65-72.

Bender, E. M.-M. (2021). On the dangers of stochastic parrots: Can language models be too big? *On the dangers of stochastic parrots: Can language models be too big?*, pp. 610-623.

Bird, S. K. (2009). *Natural Language Processing with Python.* O'Reilly Media.

Bishop, C. M. (2006). Pattern Recognition and Machine Learning. *Springer*.

Bojanowski, P. G. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, pp. 135-146.

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. *Proceedings of COMPSTAT 2010*, pp. 177-186.

Council, E. P. (1995). Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. *Official Journal of the European Communities*, 31-50.

Crawford, K. (2021). *Atlas of AI: Power, Politics, and the Planetary Costs of Artificial Intelligence.* Yale University Press.

Devlin, J. C. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies,*, pp. 4171–4186.

Devlin, J. C.-W. (2019). *Pre-training of Deep Bidirectional Transformers for Language Understanding.*

Equifax Data Breach Settlement. (2024). Obtido de Federal Trade Commision.

Gatt, A. &. (2018). Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. *Journal of Artificial Intelligence Research*, 65-170.

Goldberg, Y. (2017). *Neural Network Methods for Natural Language Processing.* Morgan & Claypool Publishers.

Goodfellow, I. B. (2016). *Deep Learning.* MIT Press.

Hochreiter, S. &. (1997). Long Short-Term Memory. Neural Computation. *Neural Computation*, pp. 1735-1780.

IBM, G. U. (1954). *Georgetown–IBM experiment.* Washington, D.C.

Jelinek, F. (1997). *Statistical methods for speech recognition.* MIT Press.

Jurafsky, D. &. (2023). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.*

Lin, C. Y. (2004). ROUGE: A package for automatic evaluation of summaries. *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*.

Liu, Y. O. (2019). RoBERTa: A robustly optimized BERT pretraining approach.

Manning, C. D. (1999). *Foundations of Statistical Natural Language Processing.* MIT Press.

Manning, C. D. (2008). . *Introduction to Information Retrieval.* Cambridge University Press.

Martinez, V. A. (2022). The Kaiser Permanente Northern California Advance Alert Monitor Program: An Automated Early Warning System for Adults at Risk for In-Hospital Clinical Deterioration. *Journal of Quality and Patient Safety*, 370-375.

Mikolov, T. S. (2013). Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems*, pp. 3111-3119.

Morris, A. M. (2004). From WER and RIL to MER and WIL: improved evaluation measures for connected speech recognition. *Proceedings of the 8th International Conference on Spoken Language Processing*.

Nielsen, M. A. (2015). Neural Networks and Deep Learning.

OECD Publishing. (2021). The Path to Becoming a Data-Driven Public Sector.

Papineni, K. R. (2002). BLEU: a method for automatic evaluation of machine translation. *Proceedings of the 40th Annual Meeting of the Association for Computational*, pp. 311-318.

Privacy Act of 1974. (1974). Estados Unidos da América.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. pp. 257-286.

Radford, A. W. (2019). *Language Models are Unsupervised Multitask Learners.* OpenAI.

Radford, A. W. (2019). Language models are unsupervised multitask learners.

Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation, GDPR). (2016). *Official Journal of the European Union*, 1-88.

Sarawagi, S. (2008). Information extraction. *Foundations and Trends in Databases*, pp. 261-377.

Schaar, A. K. (2013). The impact of user diversity on the willingness to disclose personal information in social network services: A comparison of private and business contexts. *In Human Factors in Computing and Informatics: First International Conference* (pp. 174–193). Slovenia: Springer.

Sutton, R. S. (2018). *Reinforcement Learning: An Introduction.* MIT Press.

Vaswani, A. S. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*.

Yadav, V. &. (2018). A survey on recent advances in Named Entity Recognition from deep learning models.

8. Anexos

8.1 Anexo A – Gerar e armazenar modelo spacy

```
1. import spacy
2. from spacy.util import minibatch, compounding
3. from spacy.training import Example
4. import json
5. import random
6.
7. def load_data(filename):
8.     train_data = []
9.     with open(filename, 'r', encoding='utf-8') as file:
10.         for line in file:
11.             data = json.loads(line)
12.             text = data['text']
13.             entities = []
14.             for start, end, label in data['label']:
15.                 entities.append((start, end, label))
16.             train_data.append((text, {"entities": entities}))
17.     return train_data
18. def create_examples(PLN, train_data):
19.     examples = []
20.     for text, annotations in train_data:
21.         doc = PLN.make_doc(text)
22.         example = Example.from_dict(doc, annotations)
23.         examples.append(example)
24.     return examples
25.
26. PLN = spacy.blank('pt')
27.
28. if 'ner' not in PLN.pipe_names:
29.     ner = PLN.add_pipe('ner', last=True)
30. else:
31.     ner = PLN.get_pipe('ner')
32.
33. filename = 'doccano.jsonl'
34. TRAIN_DATA = load_data(filename)
35. TRAIN_EXAMPLES = create_examples(PLN, TRAIN_DATA)
36.
37. for text, annotations in TRAIN_DATA:
38.     for start, end, label in annotations['entities']:
39.         ner.add_label(label)
40. PLN.begin_training()
41. for itn in range(500):
42.     random.shuffle(TRAIN_EXAMPLES)
43.     losses = {}
44.     batches = minibatch(TRAIN_EXAMPLES, size=compounding(4., 32., 1.001))
45.     for batch in batches:
46.         PLN.update(batch, drop=0.5, losses=losses)
47.     print(f"Losses at iteration {itn}: {losses}")
48.
49. PLN.to_disk('teste')
50.
```

8.2 Anexo B – Gerar e armazenar modelo NLTK

```
1. from nltk.classify import accuracy
2. from nltk import NaiveBayesClassifier
3. import json
4. import random
5. from nltk.tokenize import word_tokenize
6.
7. def load_data_from_jsonl(jsonl_file_path):
8.     data = []
9.     with open(jsonl_file_path, 'r', encoding='utf-8') as file:
10.         for line in file:
11.             entry = json.loads(line)
12.             text = entry['text']
13.             entities = entry['label']
14.             data.append((text, entities))
15.     return data
16.
17. def convert_to_nltk_format(data):
18.     nltk_data = []
19.     for text, annotations in data:
20.         tokens = word_tokenize(text)
21.         labels = ['O'] * len(tokens)
22.
23.         token_positions = []
24.         current_position = 0
25.         for token in tokens:
26.             start_position = text.find(token, current_position)
27.             end_position = start_position + len(token)
28.             token_positions.append((start_position, end_position))
29.             current_position = end_position
30.
31.         for start_char, end_char, entity_label in annotations:
32.             start_token_index = next((i for i, (start, end) in enumerate(
33.                 token_positions) if start <= start_char < end), None)
34.             end_token_index = next((i for i, (start, end) in enumerate(
35.                 token_positions) if start < end_char <= end), None)
36.             if start_token_index is not None and end_token_index is not None:
37.                 labels[start_token_index] = 'B-' + entity_label
38.                 for i in range(start_token_index + 1, end_token_index + 1):
39.                     labels[i] = 'I-' + entity_label
40.         annotated_sentence = list(zip(tokens, labels))
41.         nltk_data.append(annotated_sentence)
42.     return nltk_data
43.
44. jsonl_file_path = 'doccano.jsonl'
45. data = load_data_from_jsonl(jsonl_file_path)
46. nltk_format_data = convert_to_nltk_format(data)
47.
```

8.3 Anexo C – Especificações de laboratório

Name: ubuntu-lab

Encryption: disabled

Groups: /

Guest OS: Ubuntu (64-bit)

UUID: 89d857e9-4af9-47a8-9468-a34f9ffd456c

Config file: D:\Vms\lab\ubuntu-lab\ubuntu-lab.vbox

Snapshot folder: D:\Vms\lab\ubuntu-lab\Snapshots

Log folder: D:\Vms\lab\ubuntu-lab\Logs

Hardware UUID: 89d857e9-4af9-47a8-9468-a34f9ffd456c

Memory size: 4096MB

Page Fusion: disabled

VRAM size: 20MB

CPU exec cap: 100%

HPET: disabled

CPUProfile: host

Chipset: piix3

Firmware: BIOS

Number of CPUs: 4

PAE: disabled

Long Mode: enabled

Triple Fault Reset: disabled

APIC: enabled

X2APIC: enabled

Nested VT-x/AMD-V: disabled

CPUID Portability Level: 0

CPUID overrides: None

Boot menu mode: message and menu

Boot Device 1: HardDisk

Boot Device 2: DVD

Boot Device 3: Floppy

Boot Device 4: Not Assigned

ACPI: enabled

IOAPIC: enabled

BIOS APIC mode: APIC

Time offset: 0ms

BIOS NVRAM File: D:\Vms\lab\ubuntu-lab\ubuntu-lab.nvram

RTC: UTC

Hardware Virtualization: enabled

Nested Paging: enabled

Large Pages: enabled

VT-x VPID: enabled

VT-x Unrestricted Exec.: enabled

AMD-V Virt. Vmsave/Vmload: enabled

IOMMU: None

Paravirt. Provider: Default

Effective Paravirt. Prov.: KVM

State: powered off (since 2024-03-06T21:36:48.000000000)

Graphics Controller: VMSVGA

Monitor count: 1

3D Acceleration: disabled

2D Video Acceleration: disabled

Teleporter Enabled: disabled

Teleporter Port: 0

Teleporter Address:

Teleporter Password:

Tracing Enabled: disabled

Allow Tracing to Access VM: disabled

Tracing Configuration:

Autostart Enabled: disabled

Autostart Delay: 0

Default Frontend:

VM process priority: default

Storage Controllers:

#0: 'IDE', Type: PIIX4, Instance: 0, Ports: 2 (max 2), Bootable

Port 1, Unit 0: Empty

#1: 'SATA', Type: IntelAhci, Instance: 0, Ports: 1 (max 30), Bootable

Port 0, Unit 0: UUID: 0ce3e303-5898-434e-b665-a0073fff8d42

Location: "D:\Vms\Ubuntu\Ubuntu_1.vhd"

NIC 1: MAC: 080027A4D52D, Attachment: NAT, Cable connected: on, Trace: off (file: none), Type: 82540EM, Reported speed: 0 Mbps, Boot priority: 0, Promisc Policy: deny, Bandwidth group: none

NIC 1 Settings: MTU: 0, Socket (send: 64, receive: 64), TCP Window (send:64, receive: 64)

NIC 2: disabled

NIC 3: disabled

NIC 4: disabled

NIC 5: disabled

NIC 6: disabled

NIC 7: disabled

NIC 8: disabled

Pointing Device: USB Tablet

Keyboard Device: PS/2 Keyboard

UART 1: disabled

UART 2: disabled

UART 3: disabled

UART 4: disabled

LPT 1: disabled

LPT 2: disabled

Audio: disabled

Audio playback: disabled

Audio capture: disabled

Clipboard Mode: disabled

Drag and drop Mode: disabled

VRDE: disabled

OHCI USB: enabled

EHCI USB: enabled

xHCI USB: disabled

USB Device Filters: <none>

Bandwidth groups: <none>

Shared folders: <none>

Recording enabled: no

Recording screens: 1

Screen 0:

Enabled: yes

ID: 0

Record video: yes

Record audio: no

Destination: File

File: D:\Vms\lab\ubuntu-lab\ubuntu-lab-screen0.webm

Options: vc_enabled=true,ac_enabled=false,ac_profile=med

Video dimensions: 1024x768

Video rate: 512kbps

Video FPS: 25fps

* Guest:

Configured memory balloon: 0MB

8.4 Anexo D - Iterações de fine tuning

```
1. import spacy
 2. from spacy.training import Example
 3. import json
 4.
 5. def load_training_data(json_file):
 6.     with open(json_file, "r", encoding="utf-8") as f:
 7.         return json.load(f)
 8.
 9. training_data_path = "train_data.json"
10.
11. TRAIN_DATA = load_training_data(training_data_path)
12.
13. nlp = spacy.load("spacyM1")
14.
15. ner = nlp.get_pipe("ner")
16.
17. for _, annotations in TRAIN_DATA:
18.     for ent in annotations.get("entities"):
19.         ner.add_label(ent[2])
20.
21.
22. other_pipes = [pipe for pipe in nlp.pipe_names if pipe != "ner"]
23. with nlp.disable_pipes(*other_pipes):
24.     optimizer = nlp.resume_training()
25.
26.
27.     for epoch in range(20):  // seccção de iterações
28.         for text, annotations in TRAIN_DATA:
29.             doc = nlp.make_doc(text)
30.             example = Example.from_dict(doc, annotations)
31.             nlp.update([example], drop=0.35, sgd=optimizer)
32.
33.
34. nlp.to_disk("modelo_ajustado_spacy")
```

8.5 Anexo E - Fine Tuning - Tratamento

```
1. def word_features(word):
 2.     return {
 3.         'word': word,
 4.         'is_capitalized': word[0].isupper(),
 5.         'suffix_2': word[-2:],
 6.         'suffix_3': word[-3:],
 7.         'is_numeric': word.isdigit(),
 8.     }
 9. def prepare_training_data(train_data):
10.     training_data = []
11.     for sentence, annotations in train_data:
12.         tokens = nltk.word_tokenize(sentence)
13.         for word, label in annotations:
14.             features = word_features(word)
15.             training_data.append((features, label))
16.     return training_data
17.
18. doccano_file = "doccano.jsonl"
19.
20. train_data = read_doccano_jsonl(doccano_file)
21. training_data = prepare_training_data(train_data).
22.
```