Pós-Graduação em Ciência da Computação

Francisco de Assis de Souza Rodrigues

DeepNLPF: Um *Framework* para Integração de Análises Linguísticas e Anotação Semântica de Documentos Textuais

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

Francisco de Assis de Souza Rodrigues

DeepNLPF: Um *Framework* para Integração de Análises Linguísticas e Anotação Semântica de Documentos Textuais

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Orientador: Dr. Robson do Nascimento Fidalgo
Coorientador: Dr. Rinaldo José de Lima

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

R696d Rodrigues, Francisco de Assis de Souza
DeepNLPF: um *framework* para integração de análises linguísticas e anotação semântica de documentos textuais / Francisco de Assis de Souza Rodrigues. – 2019.
143 f.: il., fig., tab.

Orientador: Robson do Nascimento Fidalgo.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.
Inclui referências e apêndices.

1. Engenharia de software. 2. Processamento de linguagem natural. I. Fidalgo, Robson do Nascimento (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2020 - 60

Francisco de Assis de Souza Rodrigues

"**DeepNLPF:** Um *Framework* para Integração de Análises Linguísticas e Anotação Semântica de Documentos Textuais"

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 30 de agosto de 2019.

BANCA EXAMINADORA

Profa. Dra. Patricia Cabral de Azevedo Restelli Tedesco
Centro de Informática / UFPE

Prof. Dr. Vanilson André de Arruda Burégio
Departamento de Computação/UFRPE

Prof. Dr. Robson do Nascimento Fidalgo
Centro de Informática/UFPE **(Orientador)**

Dedico esse trabalho a Deus pela sua essência e autor de minha vida, a toda minha família que, com muito carinho e apoio, não mediram esforços para que eu chegasse até esta etapa de minha vida. Ao curso de Ciência da Computação da UFPE, e às pessoas com que convivi nesses espaços ao logo desse anos. A experiencia de uma produção compartilhada na comunhão de amigos nesses espaços foram a melhor experiencia da minha formação acadêmica.

AGRADECIMENTOS

Quando paramos os olhos diante da tela de um computador, procurando a solução de um problema persistente ou o melhor termo que transmita nossas ideias, o trabalho de pesquisa parece ser extremamente solitário. Mas não é. Trazemos nossa bagagem individual, mas ela é enriquecida pelo auxílio de milhares de seres que nos ajudaram a chegar onde chegamos. Por isso, é preciso agradecer; e eu agradeço

a Deus, Causa Primária e Inteligência Suprema;

aos meus pais Maria das Graças e José Ari (em memória), por terem me dado tudo que um filho precisa e muito mais;

à Tamires minha esposa, Ana Lis filha amada e toda minha família, por compartilhar mais uma existência, em aprendizado mútuo e mais que momentos em família;

aos Professores Robson Fidalgo e Rinaldo Lima, por ter aceitado o desafio de orientar, com paciência, segurança, transmitido sempre com tolerância, humildade e espírito científico;

à Coordenação do Programa de Pós-graduação e aos professores do curso, pelo apoio constante e a CAPES pelo apoio financeiro, sem ele isso não seria possível;

aos meus amigos e colegas de curso, Elias, Jayr, Leandro, Natália, Edson, Augusto, Flávio, Vandemberg, Osmar, Reinaldo, Diego Thurran, Alane, Julho Venâncio, Bruna, Débora, Paulo, Walter, e a todos os demais, pelo intercâmbio de ideias, certezas, dúvidas e conhecimentos;

às centenas de pesquisadores espalhados pelo mundo, cujas ideias, aplicações, artigos e livros possibilitaram que este fosse um trabalho de muitas mentes.

"Eu acredito que às vezes, são as pessoas de quem ninguém espera nada que fazem as coisas que ninguém consegue imaginar."(HODGES, 1983)

RESUMO

Atualmente as empresas vêm implementando novos modelos de negócios que dependem intensamente do Processamento de Linguagem Natural (PLN) em documentos textuais a fim de extrair informações relevantes de diversas fontes, incluindo comércio eletrônico, documentos de domínio específicos, serviços públicos, mídias sociais, etc. A implementação de um sistema de PLN requer, entre outras coisas, um considerável esforço de engenharia de software para: a criação de estruturas de dados para representação da linguagem humana; a aplicação de tais ferramentas no enriquecimento da representação textual através da análise linguística em diversos níveis (léxico, sintático e semântico); a leitura e interpretação das anotações geradas dos corpus, a criação de recursos linguísticos, entre outros. Embora existam inúmeras ferramentas de PLN amplamente utilizadas em tarefas de PLN, extração, anotação e correção linguísticas, cada uma delas fornece apenas cobertura parcial. Além disso, essas ferramentas são desenvolvidas em linguagens de programação diferentes e são disponibilizadas sem nenhuma padronização na entrada e saída de dados, o que dificulta a sua interoperabilidade devido a incompatibilidade entre as APIs, formatos de dados de saída (representação) e a tokenização básica do texto, para citar algumas. Devido a isso, selecionar as ferramentas e suas respectivas análises linguísticas de acordo com o interesse de aplicação de um usuário requer normalmente muito tempo, principalmente quando se deseja usá-las em conjunto. O objetivo deste trabalho é analisar algumas ferramentas de PLN disponíveis, propor, implementar e avaliar uma *framework* que encapsule enumeras análises linguísticas permitindo que os desenvolvedores de aplicações possam não somente executar *pipelines* de análises linguísticas mas também possam integrar ferramentas de terceiros. Além disso, visa-se fornecer uma interface gráfica (GUI) ao usuário para exploração dos recursos sem a necessidade de escrever código. A versão ora proposta do *DeepNLPF* é formada pela integração de algumas ferramentas de PLN de terceiros que foram selecionadas após uma revisão da literatura. Como contribuições deste trabalho, destacam-se: i) *wrappers* python para utilização da ferramenta de PLN CogComp NLP, SEMAFOR e SupWSD. ii) Bibliotecas Python para estatística de dados textuais, notificações, execução de scripts(Java, R, Shell Script, C/C++), arquitetura de plugins. iii) um *framework* para integração e customização de análises linguística e anotação de documentos. Finalmente, três experimentos realizados mostram que o *DeepNLPF* obteve um melhor desempenho em relação ao processamento sequencial das ferramentas de PLN testadas. Mais precisamente, cerca de 60% mais rápido em termos de tempo de processamento.

Palavras-chaves: Processamento de Linguagem Natural. *Frameworks* de PLN. Customização de *Pipeline*.

ABSTRACT

Companies today are implementing new business models that rely heavily on Natural Language Processing (PLN) in textual documents to extract relevant information from a variety of sources, including e-commerce, domain-specific documents, utilities, social media, etc. Implementing a PLN system requires, among other things, a considerable software engineering effort to: create data structures for human language representation; the application of such tools in the enrichment of textual representation through linguistic analysis at various levels (lexical, syntactic and semantic); the reading and interpretation of the corpus annotations generated, the creation of linguistic resources, among others. Although there are numerous PLN tools widely used in PLN, language extraction, annotation, and correction tasks, each provides only partial coverage. In addition, these tools are developed in different programming languages and are made available without any standardization in data input and output, which makes their interoperability difficult due to incompatibility between APIs, output data formats (representation) and basic tokenization. of the text to name a few. Because of this, selecting tools and their language analysis according to a user's application interest usually takes a lot of time, especially when you want to use them together. The objective of this paper is to analyze some available PLN tools, to propose, implement and evaluate a framework that encapsulates the language analysis allowing application developers not only to run language analysis pipelines but also to integrate third party tools. In addition, it aims to provide a graphical user interface (GUI) for resource exploration without the need to write code. The now proposed version of DeepNLPF is formed by the integration of some third party PLN tools that were selected after a literature review. Contributions of this work include: i) python wrappers for the PLN tool CogComp NLP, SEMAFOR and SupWSD. Python libraries for textual data statistics, notifications, script execution (Java, R, Shell Script, C / C ++), plugin architecture. iii) a framework for integration and customization of linguistic analysis and document annotation. Finally, three experiments show that DeepNLPF obtained better performance in relation to the sequential processing of the tested PLN tools. More precisely, about 60% faster in terms of processing time.

Keywords: Natural Language Processing. Frameworks NLP. Custom Pipeline.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AM	Aprendizagem de Máquina
API	*Interface* para Programação de Aplicações
BDNR	Banco de Dados Não Relacional
CC	Ciência da Computação
CLI	*Command-Line Interface*
CLS	*Computacion Lexical Semantics*
CPU	*Central Processing Unit*
FA	*Framework* de Aplicação
FC	*Framework* de Componentes
FCB	*Framework* Caixa Branca
FCC	*Framework* Caixa Cinza
FCP	*Framework* Caixa Preta
GPU	*Graphics Processing Unit*
GUI	*Graphical User Interface*
HPC	*High Performance Computing*
IA	Inteligência Artificial
JSON	*JavaScript Object Notation*
LC	Linguística Computacional
MR	Modelo Relacional
MT	Mineração de Texto
NER	*Named Entity Recognition*
PLN	Processamento de Linguagem Natural
POS	*Part-Of-Speech*
PP	Processamento Paralelo
SGBD	Sistema de Gerenciamento de Banco de Dados
UML	Linguagem de Modelagem Unificada
WSD	*Word Sense Disambiguation*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

A linguagem natural é uma das faculdades mais importantes do ser humano, pois ele é a única espécie conhecida que desenvolveu a linguagem falada e escrita, e é por meio dela que se comunicam, expressam suas emoções e interagem em sociedade. Existe um famoso teste denominado de "Teste *Turing*", na área de Inteligência Artificial (IA), que usa essa linguagem humana visando testar a capacidade de uma máquina exibir comportamento inteligente equivalente a um ser humano, ou indistinguível deste. Isso é possível, entre outras coisas, graças ao Processamento de Linguagem Natural (PLN), que é um campo de pesquisa em Linguística Computacional que visa automatizar a manipulação da linguagem humana seja escrita ou falada. O objetivo do PLN é portanto fornecer aos computadores a capacidade de entender e compor textos. Entender um texto significa reconhecer o contexto, fazer análise sintática, semântica, léxica e morfológica, criar resumos, extrair informação, interpretar os sentidos, analisar sentimentos e até aprender conceitos a partir de textos processados.

Mesmo com o grande avanço na área de PLN nos últimos anos, a comunicação via linguagem natural continua sendo um grande desafio, pois criar programas que sejam capazes de interpretar mensagens codificadas em linguagem natural e decifrá-las para linguagem de máquina ainda é uma questão em aberto em IA. O PLN vem sendo aplicado em muitas pesquisas e em diversos ramos destacando-se a tradução automática de texto, considerada pela maioria como o marco inicial na utilização dos computadores para o estudo das línguas naturais.

A área de PLN existe há mais de 50 anos e explora uma diversidade de análises da linguagem em vários níveis (fonológico, morfológico, sintático, semântico e pragmático) (BEYSOLOW, 2018). Além disso, o PLN é influenciado por fundamentos de várias outras disciplinas, tais como Filosofia da Linguagem, Psicologia, Lógica, Inteligencia Artificial, Matemática, Ciência da Computação, Linguística Computacional e Linguística.

PLN é um subcampo da Ciência da Computação (CC) que encontra-se em uma rápida evolução, cujas aplicações representaram uma grande parte dos avanços da IA, proporcionado às máquinas analisar e entender o sentimento em textos (PANG; LEE et al., 2008), reconhecer fala, gerar peguntas, entre outras, tudo isso, permitiu a implementação de novas aplicações inteligentes capazes de até mesmo conversar com os seres humanos, que são os casos dos [1]*chatbots* (BEYSOLOW, 2018) que lidam com solicitações de atendimento ao cliente; outras aplicações são capazes de verificar a ortografia automática em telefones celulares e assistentes IA, como Cortana e Siri, em *smartphones* (HOY, 2018).

[1] *Chatbot* é um programa de computador que tenta simular um ser humano na conversação com as pessoas. O objetivo é responder as perguntas de tal forma que as pessoas tenham a impressão de estar conversando com outra pessoa e não com um programa de computador.

Com a evolução da *Web* 2.0, as empresas vêm implementando novos modelos de negócios que dependem do PLN em dados a fim de, extrair informações relevantes de diversas fontes de dados, incluindo comércio eletrônico, documentos de domínios específicos, serviços públicos, mídias sociais, etc. Para isso, existem inúmeras ferramentas de processamento de linguagem natural amplamente utilizadas em tarefas de análise, extração, anotação e correção linguística, porem, cada uma delas fornece apenas cobertura parcial dessas inúmeras tarefas. Essas ferramentas são construídas utilizando linguagens de programação distintas e distribuídas sem padronização na entrada e saída de dados, o que dificulta a sua interoperabilidade devido a incompatibilidades e inconsistência dos dados trafegados entre as ferramentas. Um outro ponto a ser considerado é, que algumas delas são complicadas de configurar, possuem documentação muito extensa e difícil de entender ou até mesmo não possuem documentação alguma, nem todas possuem algum suporte para [2]customização de *pipeline* de análises agregando análises provenientes de outras ferramentas de PLN de terceiros, e até a necessidade da utilização e melhoria de otimizadores para diminuir do tempo de processamento de dados. Devido a esta diversidade, selecionar as melhores análises entre essas ferramentas de PLN é dispendioso, inibindo a utilização destas ferramentas por usuários em seus projetos de *software* (ZHENG et al., 2015).

Apesar do sucesso dos atuais sistemas de processamento de linguagem natural, os estudos mostram que é necessário um esforço substancial para que os usuários adotem os sistemas de PLN existentes, além disso, os usuários geralmente relatam desempenho reduzido quando um sistema existente é aplicado sem personalização além de sua finalidade original (ZHENG et al., 2015). Personalizar um sistema de processamento de linguagem natural existente requer um considerável esforço de engenharia, conhecimentos e habilidades substanciais na área de PLN tais como: criar estruturas de dados para representação de linguagem, ler anotações de corpus nesta estrutura de dados, aplicação de ferramentas prontas para aumentar a representação de texto; extração de recursos, gerar estatísticas, entre outros, o que pode ser desafiador em ambientes com conhecimentos limitados em PLN. Isso impede a adoção generalizada da tecnologia de processamento de linguagem natural por diversos domínios. Sendo assim esse trabalho parte da seguinte pergunta de pesquisa, ou seja, "Como é possível integrar diversas ferramentas de PLN em um único *framework*, de forma que ele seja extensível e ainda ser mais eficiente quanto ao tempo de processamento?". Neste contexto, possibilitar o uso de processamento de linguagem natural em aplicações simplificando o máximo possível é certamente promissor e desejável.

1.1 MOTIVAÇÃO E PROBLEMA DE PESQUISA

Algumas tarefas de mineração de texto necessitam combinar análises linguísticas para obter resultado proveitoso. Sendo assim, construir *pipelines* para análises linguística pro-

[2] Customização de *Pipeline* é a personalização, adaptação, combinação ou adequação de análises linguísticas provenientes de ferramentas diferentes para a realização de analises de texto.

venientes de ferramentas de PLN de terceiros é complexo e trabalhoso. Como também, execução de *pipeline* com muitas análises consome muito tempo e recurso computacional. Dado o exposto, este trabalho é motivado pelos seguintes fatos:

- Complexidade para integração de ferramentas de PLN de terceiros: para o usuário, construir um *pipeline* conectando ferramentas de terceiros desenvolvidas em linguagens de programação diferentes é complexo e trabalhoso devido a interoperabilidade entre as linguagens e por causa que, cada ferramenta retorna resultados em formato diferente;
- Custo computacional - para o pesquisador ou desenvolvedor de ferramentas de PLN, executar *pipeline* com muitas análises linguísticas a partir de um texto de entrada em sua aplicação consome muito tempo, otimizar o tempo de processamento é necessário;
- *Interface Gráfica* - para [3]usuários leigos, uma *interface* gráfica torna possível a utilização de PLN sem a necessidade de codificação.

O problema de pesquisa abordado nesta proposta é a integração e a customização de *pipeline* de processamento de linguagem natural para anotação de dados textuais.

1.2 OBJETIVO DA PESQUISA

O objetivo deste trabalho é [1] analisar as ferramentas de PLN disponíveis na literatura e [2] propor uma *framework* que encapsule diversas análises linguísticas provindas de ferramentas de PLN de terceiros, permitindo a execução de *pipeline* de análises linguísticas que envolvam o tratamento da língua inglesa nos níveis léxico, sintático e semântico, usando ferramentas de PLN intercaladas minimizando os problemas de integração e desempenho. Sendo assim, visa-se projetar, implementar e avaliar:

- um *framework* capaz de integrar ferramentas de PLN de terceiros e padronizar os dados gerados por essas ferramentas.
- uma estratégia para reduzir o tempo de processamento de dados por ferramentas de PLN de terceiros integradas ao *pipeline* de análise customizado.
- uma *Interface* para Programação de Aplicações (API) para permitir que os usuários possam construir aplicações utilizando todo o arsenal de ferramentas presente no *framework* com o mínimo de instalações e configurações possíveis.
- uma *Command-Line Interface* (CLI) que permita que o usuário acesse os recursos e experimenta análise das ferramentas presente no *framework* pelo terminal de comando.

[3] Usuários que possui pouca ou nenhuma experiência com PLN.

Desta forma as funcionalidades fornecidas pelo *framework* proposto, *DeepNLPF*, ajudará os usuários a realizar várias análises em dados textuais, guiando-os a escolher o mais adequado e eficiente *pipeline* possível.

1.3 CONTRIBUIÇÕES

Este trabalho provê uma forma para os usuários integrar e customizar *pipeline* de processamento de linguagem natural para anotação de dados textuais. Assim, espera-se diminuir os esforço no desenvolvimento de aplicações de PLN, bem como diminuir o tempo no processamento dos dados analisado. Este trabalho esperam-se as seguintes contribuições:

- *Wrapper* Python para utilização da ferramenta de PLN CogComp, SEMAFOR e SupWSD.
- Bibliotecas Python para estatística de dados textuais, notificações, execução de scripts (Java, R, Shell Script, C/C++), arquitetura de plugins.
- Um *Framework* para integração e customização de análises linguística e anotação de documentos.

1.4 ESTRUTURA DO DOCUMENTO

O restante deste documento é dividido em 5 capítulos. No Capítulo 2, apresenta-se os fundamentos teóricos necessários para embasar este trabalho, os quais são divididos em cinco seções principais: Processamento de Linguagem Natural, Pipeline de Processamento de Linguagem Natural, Formatos de Saída de Anotações Linguística, *Frameworks* e *Desing Patterns* e *Frameworks*, *Libraries* e *Tollkits* para PLN. No Capítulo 3, apresenta-se os trabalhos relacionados a esta proposta. No Capítulo 4, apresenta-se o *DeepNLP Framework*, com uma visão arquitetural. No Capítulo 5, apresenta-se avaliações qualitativa e quantitativa da proposta. No Capítulo 6, apresenta-se as considerações finais, as contribuições e as limitações deste trabalho, bem como direcionamentos para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo abordam-se os conceitos de Processamento de Linguagem Natural, Análises Linguísticas e Arquitetura de PLN, *Frameworks* e *Design Patterns*, Formato e Saída de Anotações Linguística. Estes conceitos formam a fundamentação teórica que servem de base para o entendimento desse trabalho.

2.1 PROCESSAMENTO DE LINGUAGEM NATURAL

Segundo (PUSTEJOVSKY; STUBBS, 2012) Processamento de Linguagem Natural (PLN) é um campo da ciência da computação e engenharia que se desenvolveu a partir do estudo da [1]Linguística Computacional (LC) e linguística no campo da Inteligência Artificial. O autor (II, 2018) define PLN como uma sub-área da Inteligência Artificial (IA), especificamente no ramo da Linguística Computacional (LC) que analisa a linguagem natural por meios automáticos, permitindo que os computadores entendam a linguagem humana.

Os objetivos do PLN vão de projetar a construir aplicativos que facilitem a interação humana com máquinas e outros dispositivos por meio do uso da linguagem natural. PLN também fornece os fundamentos teóricos e práticos para muitas aplicações que envolvem Mineração de Texto (MT). As aplicações desta tecnologia incluem tradução automática, recuperação de texto, tarefas de análise de sentimento, reconhecimento de falas, criação de resumo automático, povoamento de ontologia, extração de informações, interpretação de sentidos e até mesmo gerar respostas para perguntas.

Segundo (LIMA; FREITAS, 2014), PLN é extremamente desafiador por duas razões: (I) a linguagem natural tem muita ambiguidade e (II) as palavras podem ser combinadas em sentenças de várias maneiras, isso impossibilita o computador de entender, sendo assim, o computador fornecerá simplesmente uma lista dos possíveis cotextos e significados de uma determinada palavra ou sentença. Neste contexto, para as tarefas de PLN, os sistemas precisam ser capazes de processar e manipular a linguagem em diversos níveis da linguística conforme a Figura 1.

- **Análises Fonética e Fonologia**, é estudo dos padrões sonoros de uma linguagem particular. Aspectos do estudo incluem determinar quais fonemas são significativos e têm significado; como as sílabas são estruturadas e combinadas; e quais recursos são necessários para descrever as unidades discretas (segmentos) na linguagem e como elas são interpretadas (PUSTEJOVSKY; STUBBS, 2012).

[1] A linguística computacional é a área de conhecimento que explora as relações entre linguística e informática, tornando possível a construção de sistemas com capacidade de reconhecer e produzir informação apresentada em linguagem natural. (VIEIRA; LIMA, 2001)

Figura 1 – Níveis de Análise Linguística.

Fonte: (MCINTOSH, 2018).

- **Análise Morfológica**, é o estudo de unidades de significado em uma linguagem. Um morfema é a menor unidade de linguagem que tem significado ou função, uma definição que inclui palavras, prefixos, afixos e outras estruturas de palavras que conferem significado (PUSTEJOVSKY; STUBBS, 2012). Segundo (LIMA; FREITAS, 2014), palavras que são construídas a partir de mais de um morfema podem ser divididas em um radical e um ou mais afixos, isto é, um morfema ligado a um radical como em *"book"* + *"s"*, no qual o plural *"s"* é um morfema inflexível, pois altera a forma sem alterar sua categoria sintática.

- **Análise Lexical**, é o estudo que identifica e analisa a estrutura das palavras. O léxico de uma linguagem significa a coleção de palavras e frases em um idioma. A análise lexical divide o pedaço inteiro de texto em parágrafos, sentenças e palavras (PUSTEJOVSKY; STUBBS, 2012).

- **Análise Sintática**, é o estudo de como as palavras são combinadas para formar sentenças (PUSTEJOVSKY; STUBBS, 2012). Isso inclui examinar partes do discurso e como elas se combinam para fazer construções maiores. Segundo (LIMA; FREITAS, 2014), a sintaxe se refere ao modo como as palavras são organizadas juntas, formando

estruturas legais de uma linguagem. Em primeiro lugar, as palavras que apresentam comportamento semelhante são categorizadas em categorias sintáticas ou partes da fala (POS). Tais categorias por exemplo, incluem substantivos, verbo e adjetivo. O conhecimento sintático é expresso por regras para combinar essas categorias em frases e os papéis estruturais que essas frases podem desempenhar em uma frase. Por exemplo, a sentença (I), escrita em inglês é sintaticamente correta, enquanto a sentença (II) não é.

(I) - *Diamond is the hardest material on earth.*
(II) - *Diamond the is earth material hardest on.*

Fonte: (LIMA; FREITAS, 2014).

- **Análise Semântica**, é o estudo do significado na linguagem. A semântica examina as relações entre as palavras e o que elas estão sendo usadas para representar (PUSTEJOVSKY; STUBBS, 2012). Por exemplo, a sentença (III) é um exemplo de sentença significativa, enquanto a (IV) não é.

(III) - *I have read the book on Semantic Web.*
(IV) - *The book spoke to me about its large head.*

Fonte: (LIMA; FREITAS, 2014).

- **Análise Pragmática**, é o estudo de como o contexto do texto afeta o significado de uma expressão, e que informação é necessária para inferir um significado oculto ou pressuposto (PUSTEJOVSKY; STUBBS, 2012). O exemplo (V) mostra a ideia do estudo de como as pessoas se entendem linguisticamente.

(V)
- So, dis you?
- Hey who wouldn't?

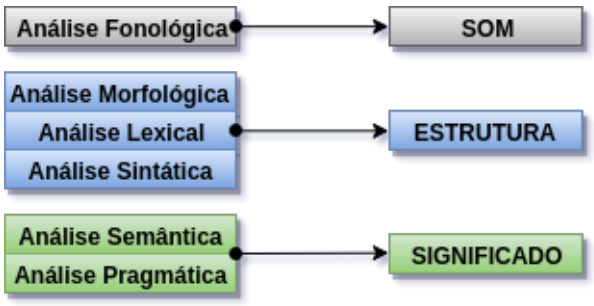
Fonte: (YULE, 2016).

Nesse trecho de fala, representa dois amigos em uma conversa, que podem implicar algumas coisas e inferir outras sem fornecer nenhuma evidência linguística clara. Portanto, a pragmática exige compreensão do que as pessoas tem em mente.

Todas as análises linguísticas citadas anteriormente podem ser agrupadas da seguinte forma:

Em PLN, essas análises linguísticas são executadas em uma arquitetura de *Pipeline* que será abordada a seguir na Seção 2.1.

Figura 2 – Grupos Níveis da Linguagem.

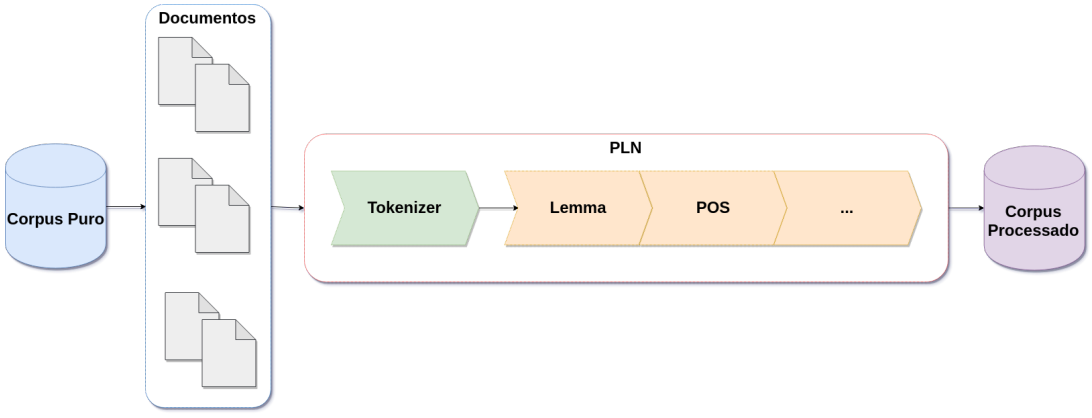


Fonte: do Autor.

2.2 PIPELINE DE PROCESSAMENTO DE LINGUAGEM NATURAL

O processamento de linguagem natural, na maioria dos aplicativos atuais de mineração de texto, consiste tipicamente na aplicação sequencial de vários componentes que executam as sub-tarefas de PNL, como ilustra a Figura 3. Normalmente, essas sub-tarefas são executadas no modo de *pipeline*, ou seja, começando com análises mais simples, como *sentence splitter* e *tokenization*. Os resultados obtidos dessas análises básicas servem como entrada para as próximas sub-tarefas complexas, como marcação POS e análise sintática (LIMA; FREITAS, 2014). A Figura 3 mostra um pipeline base de como isso ocorre.

Figura 3 – Pipeline de PLN.



Fonte: o autor

Embora essa abordagem de *pipeline* tenha obtido bastante sucesso em Processamento de Linguagem Natural (PLN), devido a sua modularidade, ela sofre vários inconvenientes do ponto de vista de utilização e desenvolvimento de aplicações, tanto para o usuário desenvolvedor e/ou pesquisador, construir um *pipeline* conectando ferramentas de PLN existentes, juntar os resultados produzidos pelas ferramentas, pode ser complexo e trabalhoso, já que muitas vezes cada sistema envia os resultados em um formato diferente

com nomenclaturas diferentes. O mesmo acontece para aplicações [2]*downstream*, a execução completa do *pipeline* a partir de um texto de entrada em sua aplicação consome muito tempo (NOJI; MIYAO, 2016). O *pipeline*, é formado por várias análises linguísticas, os sub-tópicos apresentados a seguir, objetivam fornecer uma visão geral dessas análises que serão aplicadas neste trabalho.

Sentence Splitting: consiste em determinar os limites das sentenças em um documento. Ou seja, um corpus possui vários documentos, o computador deve conseguir extrair textos desses documentos, e em seguida identificar e separar cada sentença desse texto. Veja a seguir um exemplo.

Texto de entrada para análise de *splitting*:

In the beginning God created the heavens and the earth. Now the earth was formless and empty, darkness was over the surface of the deep, and the Spirit of God was hovering over the waters. And God said, Let there be light, and there was light.

Fonte: (JAMES et al., 1969).

Saída produzida pela análise *splitting*:

["In the beginning God created the heavens and the earth.", "Now the earth was formless and empty, darkness was over the surface of the deep, and the Spirit of God was hovering over the waters.", "And God said, Let there be light, and there was light."]

Observe, que basicamente a análise *splitting* identifica cada sentença presente no texto e separa cada uma delas, formando uma lista de sentenças.

Tokenization: é o processo de dividir um fluxo de texto em palavras, símbolos ou outros elementos significativos, geralmente referidos como *tokens*. Por exemplo, os sinais de pontuação, como os períodos, podem ser problemáticos porque podem denotar o final de uma frase, o final de uma abreviação ou podem ser usados para especificar datas, números de telefone, etc. Um outro problema é sobre os espaçamentos em branco que nem sempre indicam limites de palavras, como é o caso de muitas entidades nomeadas como *"Nova York"*, que na verdade denota expressões com várias palavras formadas por mais de um *token*. Por causa disso, às vezes parece mais útil aplicar o reconhecimento de entidades nomeadas antes de realmente executar a *tokenization*.

Sentença de entrada para análise *tokenization*:

In the beginning God created the heavens and the earth.

Fonte: (JAMES et al., 1969).

[2] Softwares onde a entrada de um determinado componente depende da saída de outro componente para gerar um resultado de processamento.

Saída produzida pela análise *tokenization*:

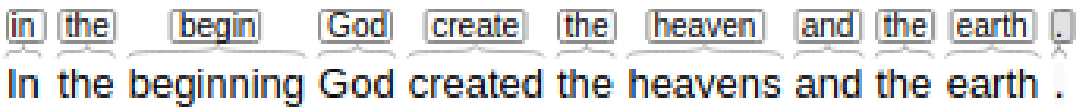
["In", "the", "beginning", "God", "created", "the", "heavens", "and", "the", "earth", "."]

Observe que o *tokenizador* analisa a sentença e identifica cada *token* armazenando-os em uma lista de *tokens*.

Stemming: é mais um processo bruto baseado em regras, pelo qual tem o objetivo de unir diferentes variações do *token*. Por exemplo: a palavra *eat* terá variações como *eating*, *eaten*, *eats* e assim por diante. Em algumas aplicações, como não faz sentido diferenciar entre *eat* e *eating*, normalmente usa-se o recurso para desviar ambas as variações gramaticais para a raiz da palavra. Embora o *stemming* seja usado na maior parte do tempo por sua simplicidade, existem casos de linguagem complexa ou de tarefas complexas de PNL onde é necessário usar a lematização (HARDENIYA et al., 2016).

Lemmatization: consiste no processo de reduzir cada palavra à sua forma básica, ou ao seu lema (HARDENIYA et al., 2016). Por exemplo, em inglês, palavras podem aparecer em muitas formas flexionadas. Considere o verbo *"to call"* que pode aparecer como *"call"*, *"called"*, *"calls"*, *"calling"* (LIMA; FREITAS, 2014). Assim, a forma básica *"call"* é o lema de suas formas flexionadas. A *lemmatization* está relacionada com o *stemming*. A diferença é que o *stemmer* opera em uma única palavra sem conhecimento do contexto e, portanto, não pode discriminar entre palavras que têm diferentes significados dependendo da parte da fala. Considerando a sentença: *"In the beginning God created the heavens and thee arth."* obtem o seguinte resultado após a *lemmatization*: *[('In', 'In'), ('the', 'the'), ('beginning', 'begin'), ('God', 'God'), ('created', 'create') ('the', 'the'), ('heavens', 'heaven'), ('and', 'and'), ('the', 'the'), ('earth', 'earth') ('.', '.')]*, ou seja, cada *token* precedido do seu *lemma*, conforme ilustra a Figura 4.

Figura 4 – Exemplo Análise *Lemmatization*.



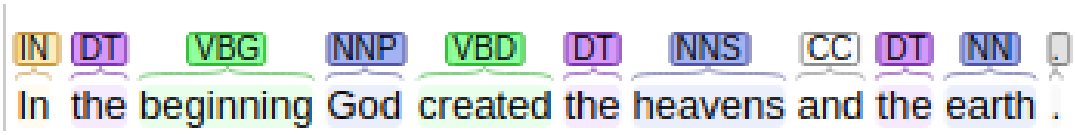
Fonte: o autor.

Part-Of-Speech Tagging: esse processo se dá após acessar os *tokens* dentro das sentenças dos parágrafos de um documento, cada *token* é marcado com seu *Part-Of-Speech* (POS) (RAO; MCMAHAN, 2019). Na lista alfabética de [3]POS Tags usadas no projeto [4]*Penn Treebank*, por exemplo: verbos, substantivos, preposições, adjetivos, indicam como uma

[3] https://www.ling.upenn.edu/courses/Fall$_2$003/$ling$001/$penn_t reebank_p os.html$
[4] https://catalog.ldc.upenn.edu/LDC99T42

palavra está funcionando dentro do contexto de uma sentença. Em inglês, como em muitos outros idiomas, uma única palavra pode funcionar de várias maneiras, (por exemplo, *building* pode ser um substantivo ou um verbo). A marcação POS implica rotular cada *token* com a *tag* apropriada, que codificará informações sobre a definição da palavra e seu uso no contexto (II, 2018). Considerando a seguinte sentença: "*In the beginning God created the heavens and the earth.*", após a análise de POS, obtém-se os tokens e seus respectivos *POS tags: [('In', 'IN'), ('the', 'DT'), ('beginning', 'VBD'), ('God', 'NNP'), ('created', 'VBD') ('the', 'DT'), ('heavens', 'NNS'), ('and', 'CC'), ('the', 'DT'), ('earth', 'NN') ('.', '.')]*. Onde, IN: *Preposition or subordinating conjunction* ; DT: *Determiner*; VBG: *Verb, gerund or present participle*; NNP: *Proper noun, singular*; VBD: *Verb, past tense*; DT: *Determiner*; NNS: *Noun, plural*; CC: *Coordinating conjunction*; NN: *Noun, singular or mass*; conforme a lista alfabética de *POS Tags* usadas no projeto *Penn Treebank* no apendice 13 desse trabalho.

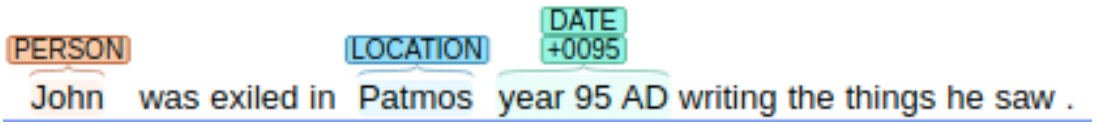
Figura 5 – Exemplo: Análise *POS Tags*.



Fonte: o autor.

Named Entity Recognition (NER): conhecida como reconhecimento de entidades nomeadas, essa análise tem por objetivo solucionar problemas de rotulagens mais comuns, tais como encontrar entidades no texto. (HARDENIYA et al., 2016). Segundo a visão do autor (LIMA; FREITAS, 2014) a análise *NER* identifica entidades nomeadas em textos e associa-lhes uma categoria semântica. Geralmente, as entidades nomeadas incluem os nomes de pessoas e organizações, expressões de datas, porcentagem, lugar, entre outras. Além disso, *NER* depende de dicionário de entidades nomeadas, muitas vezes ajustadas para um domínio especifico de interesse. A Figura 6 exemplifica a identificação de entidades nomeadas em uma dada sentença de entrada para o analisador *NER*.

Figura 6 – Exemplo Análise *NER*.



Fonte: o autor.

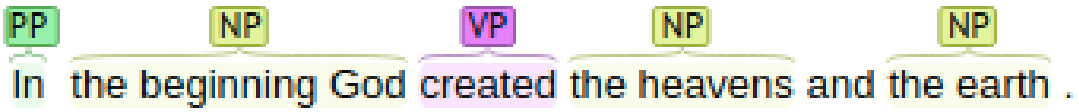
Chunking: consiste em dividir uma frase em grupos de palavras sintaticamente correlacionadas, como substantivos, verbos e frases preposicionais, não especificando nem sua estrutura interna em seu papel na sentença principalmente. Em outras palavras, são pe-

daços não agrupados de palavras sobrepostas, que formam pequenas unidades sintáticas ou frases. (RAO; MCMAHAN, 2019).

Chunking é as vezes chamado de *Chunk Parsing* ou *Shallow Parsing*. Normalmente, é necessário que os fragmentos sejam não recursivos, ou seja, nenhum outro fragmento pode ser incorporado em um fragmento. A unidade principal *(head)* em uma frase nominal em inglês é comumente o substantivo mais à direita. Por exemplo, na frase *"the exciting modern art museum"*, *"museum"* denota a unidade constituinte principal, enquanto outras palavras estão essencialmente modificando ou restringindo o significado do substantivo principal (LIMA; FREITAS, 2014).

A análise de agrupamentos geralmente adota uma abordagem de baixo para cima, ou seja, começa a detectar unidades mais simples e, em seguida, integra essas unidades em unidades mais complexas. Esse tipo de análise não descobre relações sintáticas como sujeito ou objeto. Além disso, adota uma estratégia conservadora e tende a evitar erros, uma vez que não tenta resolver ambiguidades semânticas ou sintáticas. As grandes vantagens do *chunking* são sua robustez e eficiência. Na Figura 7 confere-se o resultado da análise de *chunking* aplicada na sentença, onde NP, VP, PP significa frase nominal, frase verbal e frase preposicional, respectivamente.

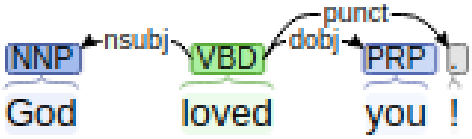
Figura 7 – Exemplo: Análise *Chunking*.



Fonte: o autor.

Dependency Parsing: os analisadores de dependência são um mecanismo para extrair a estrutura sintática de uma sentença, ligando frases com relações específicas. Eles o fazem primeiro identificando a palavra-chave de uma frase, depois estabelecem ligações entre as palavras que as modificam o *head*. O resultado é uma estrutura sobreposta de arcos que identificam sub-estruturas significativas da sentença (II, 2018). A Figura 8, ilustra um exemplo dessa análise. Onde *loved* é um VBD: *Verb, past tense* verbo passado, substantivo nominal de *God* que é um NNP: *Proper noun, singular* nome próprio singular, *loved* também é um objeto direto de *you* que é um PRP: *Personal pronoun* pronome pessoal.

Figura 8 – Exemplo: Análise *Dependency Parsing*.



Fonte: o autor.

Word Sense Disambiguation (WSD): é a tarefa de distinguir duas ou mais das mesmas grafias ou as mesmas palavras sonoras com base em seu sentido ou significado. Técnicas como [5]*Lesk*, *Similarity* e suas variações, entre outras técnicas, são utilizadas para essa tarefa (HARDENIYA et al., 2016). WSD é um modelo que enfoca o estudo do significado de palavras, um subcampo em PLN chamado de *Computacion Lexical Semantics* (CLS). Um exemplo, é a sentença *"João likes the orange color."* o desambiguador deve entender o contexto da palavra *orange*, que será desambiguada, ou seja, entender seu sentido, que nesse caso refere-se a uma cor (atributo visual). Já para a sentença *"John likes apple and orange."* o analisador deve entender que a palavra *"orange"* o seu contexto trata-se de um fruta (alimento), correspondendo a outro sentido.

Os algorítimos *WSD* fazem uso de recursos semânticos como Dicionários e/ou Tesauros e Ontologias para introduzir um modelo de análise mais rico.

- Dicionários e/ou Tesauros: é compilação completa ou parcial das unidades léxicas de uma língua (palavras, locuções, afixos etc.) ou de certas categorias específicas suas, organizadas numa ordem convencionada, ger. alfabética, e que pode fornecer, além das definições, informações sobre sinônimos, antônimos, ortografia, pronúncia, classe gramatical, etimologia etc. Ou seja, é uma linguagem documentária caracterizada pela especificidade e pela complexidade existente no relacionamento entre os termos que comunicam o conhecimento especializado.

- Ontologia: é um modelo de representação do conhecimento que, a exemplo do tesauro, é utilizada para representar e recuperar informação por meio de estruturas conceituais (no caso da ontologia o meio de ação é o informático).

Para tais recurso pode-se destacar:

- [6]*WordNet*: um grande banco de dados léxico do inglês que contem substantivos, verbos, adjetivos e advérbios, agrupados em conjuntos de sinônimos cognitivos chamados de (synsets), cada um expressando um conceito distinto (MILLER, 1995).

[5] O algoritmo de *Lesk* é um algoritmo clássico para desambiguação do senso de palavras introduzido por Michael E. *Lesk* em 1986.

[6] https://wordnet.princeton.edu/

- [7] *WordNet Domains*: é um recurso léxico criado de maneira semiautomática, aumentando o *WordNet* com rótulos de domínio. Os *Synsets* do *WordNet* foram anotados com pelo menos um rótulo de domínio semântico, selecionado de um conjunto de cerca de duzentos rótulos estruturados de acordo com a Hierarquia de Domínio do *WordNet* (BENTIVOGLI et al., 2004).

- [8] *VerbNet*: é a maior rede *on-line* de verbos em inglês que liga seus padrões sintáticos e semânticos. É um léxico de verbos de ampla cobertura hierárquica, independente do domínio, com mapeamentos para outros recursos lexicais, como *WordNet* (Miller, 1990; Fellbaum, 1998), *PropBank* (Kingsbury e Palmer, 2002) e *FrameNet* (Baker et al. , 1998) (SCHULER, 2005).

- [9] *FrameNet*: está construindo um banco de dados léxico do inglês que é legível para humanos e para máquinas, baseado na anotação de exemplos de como as palavras são usadas em textos reais. (BAKER; FILLMORE; LOWE, 1998).

- [10] *PropBank*: é um corpus anotado com proposições verbais e seus argumentos - um "banco de proposições". Embora "*PropBank*"se refira a um corpus específico produzido por Martha Palmer et al., [1] o termo *propbank* também está sendo usado como um substantivo comum que se refere a qualquer corpus que tenha sido anotado com proposições e seus argumentos. O projeto *PropBank* desempenhou um papel na pesquisa recente. No processamento de linguagem natural e tem sido usado na identificação de funções semânticas.

- [11] *Gazzetter*: é um dicionário geográfico que consiste em um conjunto de listas contendo nomes de entidades, como cidades, organizações, dias da semana, etc. Essas listas são usadas para localizar ocorrências desses nomes no texto, por exemplo. para a tarefa de reconhecimento de entidade nomeada. A palavra '*gazetteer*' é frequentemente usada de forma intercambiável para o conjunto de listas de entidades e para o recurso de processamento que faz uso dessas listas para encontrar ocorrências dos nomes no texto (BONTCHEVA et al., 2004).

A seguir na Seção 2.3 os formatos de saídas de anotações linguísticas em que podem estar sendo anotadas essas análises discutidas.

2.3 FORMATOS DE SAÍDA DE ANOTAÇÕES LINGUÍSTICA

Anotação Linguística é uma ferramenta importante para linguistas e cientistas da computação. De acordo com (IDE; ROMARY, 2004) anotação é o processo de adicionar informação

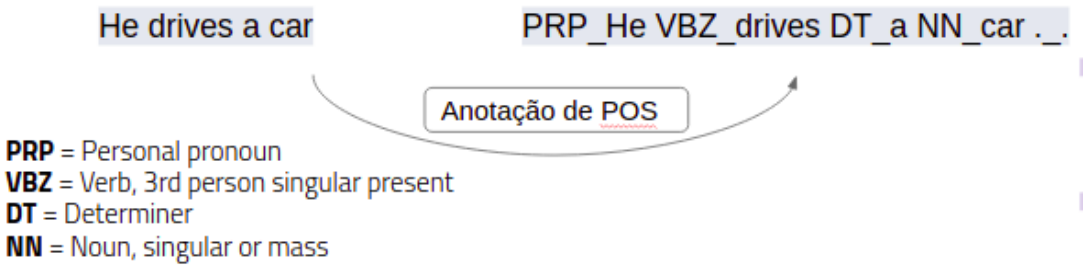
[7] http://wndomains.fbk.eu/
[8] http://verbs.colorado.edu/ mpalmer/projects/verbnet.html
[9] https://framenet.icsi.berkeley.edu/fndrupal/
[10] https://propbank.github.io/
[11] https://gate.ac.uk/sale/tao/splitch13.html

linguística a dados de linguagem (anotação de um corpus) ou a própria informação linguística (uma anotação), independente da sua representação. [12]Corpus são vários conjuntos de dados de linguagem natural. Corpus Anotado é um único conjunto de dados anotados com a mesma especificação. Os corpus anotados podem ser usados para treinar algoritmos de Aprendizagem de Máquina (AM). Segundo o autor (PUSTEJOVSKY; STUBBS, 2012) anotações linguísticas são basicamente metadados que fornecem informações adicionais sobre o texto analisado. Em aprendizagem de máquina, os metadados são aplicados como indicadores, afim de, melhorar a eficiência e ter mais relevância sobre um conjunto de dados. Antes de começar a anotar corpus, é preciso considerar os tipos de anotação de corpus existentes e quais suas peculiaridades. Nesta seção, será abordado sobre os formatos de anotações de corpus *stand-off* e *in-line*, seus recursos, vantagens, desvantagens e consequências do seu uso, como também, os tipos de arquivos *XML* e *JSON*, que possibilitam o armazenamento das anotações e oferecem suporte a consultas complexas e eficientes.

2.3.1 Anotação *In-line*

Anotações *in-line* são incluídas diretamente no dado, alterando assim os dados primários. Exemplos são anotações baseadas em *tokens* ou frases que são diretamente anexadas à *string* relacionada com separadores ou *tags* estruturais. Usa-se o termo *token* para incluir pontuação e evitar entrar na definição da palavra, pois ela pode variar em diferentes recursos. Exemplo de anotação *in-line* para a sentença *"He drives a car"*. No exemplo de anotação *Lancaster-Oslo-Bergen (LOB) style in-line*, mostra uma anotação de *POS* (ECKART, 2012).

Figura 9 – Exemplo - Anotação In-line.



Fonte: o autor.

Observe que as *tags POS* são incluídas diretamente na própria sentença, separando o *POS* do *token* pelo caractere "_", alterando diretamente o arquivo original.

[12] Um corpus é uma coleção de textos legíveis por máquina que foram produzidos em um ambiente natural de comunicação. Exemplos de corpus: *Brown Corpus*, *Child Language Data Exchange System (CHILDES)* e *Lancaster-Oslo-Bergen Corpus*

2.3.2 Anotação *Stand-off*

O formato *Stand-off* basicamente ser resume em um documento primário contendo o texto e um outro documento secundário contendo os dados anotados, contendo referencias aos dados do documento primário. Vários documentos de anotações em *stand-off* para um determinado tipo de anotação podem se referir ao mesmo documento primário (por exemplo, duas partes diferentes de anotações de fala para um determinado texto). Não há exigência de que um único documento compatível com *XML* possa ser criado pela mesclagem de documentos de anotação *stand-off* com os dados primários; ou seja, dois documentos de anotação podem especificar árvores sobre os dados primários que contêm hierarquias sobrepostas (IDE; ROMARY, 2004). A seguir, o Bloco de Código 1 exemplifica *stand-off PAULA annotation* aplicado a sentença *"He drives a car"*.

Algoritmo 1: PAULA Annotation.

```
1     <body>He drives a car.</body>
      <mark id="tok_1" xlink:href="#xpointer(string-range(//body,'',1,2))"/>
3     <mark id="tok_2" xlink:href="#xpointer(string-range(//body,'',4,5))"/>
      <!-- ... -->
5     <feat xlink:href="#tok_1" value="stts.type_pos.xml#PRP"/>
      <feat xlink:href="#tok_2" value="stts.type_pos.xml#VBZ"/>
7     <!-- ... -->
```

Observe que no exemplo mostrado, *PAULA Annotation*, mostra uma anotação de impasse fazendo uso dos *XPointers* para referências nos dados primários. Isso requer que o recurso original seja armazenado em um arquivo que pode ser referenciado pelos *XPointers*, como um arquivo *XML*. A anotação *stand-off* também é usada para anotar outros tipos de mídia além do texto, por exemplo, arquivos de áudio (ECKART, 2012).

2.3.3 Anotação *Hybrid*

Esse tipo de anotação combina as duas abordagens anteriores *stando-off* e *in-line*, como o formato *XML*, *TIGER XML*. Neste formato de anotação, o texto original é segmentado em *tokens* sem referências no recurso original. Anotação das frase sintáticas é representada como uma camada separada no topo da anotação da parte da fala. Na parte de anotação para as frases sintáticas, os *tokens* são referenciados por identificadores, e s1_1 (ECKART, 2012).

Algumas comparações podem ser feitas entre os tipos de anotações, são: existem alguns problemas com a anotação *in-line*, é que muitas vezes o recurso original não pode ser facilmente recuperado apenas removendo a anotação; outras dificuldades surgem, por exemplo, quando informações como a colocação de espaços em branco, quebras de linha ou outras informações de formatação não foram explicitadas no recurso anotado; uma outra dificuldade com formato *in-line* é a sobreposição de anotações, por exemplo, quando diferentes esquemas de anotação são aplicados aos mesmos dados, ou se informações de

Algoritmo 2: TIGER XML.

```
<s id="s1" >
    <graph root="s1_500" >
        <terminals>
            <t id="s1_1" word="He" pos="PRP" />
            <t id="s1_2" word="drives" pos="VBZ" />
            <t id="s1_3" word="a" pos="DT" />
            <t id="s1_4" word="car" pos="NN" />
            <t id="s1_5" word="." pos="." />
        </terminals>
        <nonterminals>
            <nt id="s1_502" cat="NP" >
                <edge idref="s1_1" label="--" />
            </nt>
            <nt id="s1_503" cat="NP" >
                <edge idref="s1_3" label="--" />
                <edge idref="s1_4" label="--" />
            </nt>
            <!-- ... -->
        </nonterminals>
    </graph>
</s>
```

formatação como segmentações de página se sobrepõem à anotação de unidades linguísticas, por exemplo frases. Além disso, é difícil trabalhar em paralelo com o mesmo recurso, pois as novas camadas de anotação devem ser inseridas na mesma origem ou documento (ECKART, 2012).

A anotação *stand-off* oferece uma maior flexibilidade, pois a camada de anotação é encapsulada e pode coexistir com versões alternativas ou mesmo conflitantes do mesmo tipo de anotação. Embora cada vez que os projetos prefiram o uso de anotações *stand-off* por motivos de sustentabilidade, ainda há casos em que as anotações em *in-line* são necessárias. As ferramentas de PLN precisam levar em conta mais informações (estruturais) ao trabalhar em dados de impasse que podem ter um impacto no tempo de processamento, especialmente se grandes volumes de dados tiverem que ser processados (ECKART, 2012). Esses dados podem ser armazenado em arquivos de texto como visto anteriormente, na próxima seção, será abordado sobre os formatos de arquivos de texto mais utilizados em PLN.

2.3.4 Formatos *XML* e *JSON*

Extensible Markup Language (XML) e *JavaScript Object Notation* (JSON), ambos têm padrões bem documentados na web, e são legíveis por humanos e máquinas. Nenhum deles é absolutamente superior ao outro, pois cada um é adequado para diferentes casos de uso.

- *Extensible Markup Language* (XML): é um formato de dados que foi consolidado pelo [13]W3C, sendo iniciados estudos em meados das décadas de 1990. O objetivo era criar

[13] *World Wide Web Consortium* é a principal organização de padronização da *World Wide Web - WWW*,

um tipo de formato que poderia ser lido por *software* e que tivesse flexibilidade e simplicidade, visando: a possibilidade de criação de *tags*, concentração na estrutura da informação e não na sua aparência entre outras coisas. Suas vantagens comparado ao *JSON* são: **suporte a *metadata***, uma das principais vantagens do *Extensible Markup Language* (XML) é poder inserir metadados nas *tags* na forma de atributos. *JavaScript Object Notation* (JSON) simplesmente não tem esse recurso, pois os atributos são adicionados como outros campos de membros na representação de dados; **renderização no navegador**, uma outra vantagem do *Extensible Markup Language* (XML) é que a maioria dos navegadores o processa de maneria altamente legível e organizada; **suporta a conteúdo**, o *Extensible Markup Language* (XML) tem a capacidade de comunicação dentro da mesma carga de dados, ou seja, na mesma *tag*, esse conteúdo misto é claramente diferenciado com diferentes *tags* de marcação, o que não é possível no *JavaScript Object Notation* (JSON).

- *JavaScript Object Notation* (JSON): um acrônimo para *"JavaScript Object Notation"*, é um formato de padrão aberto que utiliza texto legível para humanos, para transmitir objetos de dados. Suas vantagens comparado ao *XML* é ser menos detalhado porque o *XML* exige a abertura e o fechamento de *tags* e o *JSON* usa apenas chave/valor de forma concisa; ter menor tamanho, com a mesma quantidade de informações, o *JSON* é quase sempre significativamente menor, o que leva a uma transmissão e processamento mais rápido. Além disso, percebe-se que o *JSON* é serializado e serializado drasticamente mais rápido que o *XML*; sua compactabilidade é uma das vantagens mais significativas que o *JSON* tem sobre o *XML*, pois possui um subconjunto de *JavaScript*, portanto, o código para analisar e empacotá-lo se encaixa naturalmente com código *JavaScript*.

Cada um destes formatos de arquivos de texto, como visto, tem suas vantagens e desvantagens, cabendo portanto, ser ponderado na hora de escolher, conforme os requisitos exigidos pela aplicação a ser construída. Na próxima Seção 2.4 serão discutidos conceitos que impactam diretamente no projeto de software, são eles: *framework* e *Design Patterns*, que podem alavancar ou não o desenvolvimento de software complexos que aplicam PLN.

2.4 *FRAMEWORKS E DESIGN PATTERNS*

Nessa seção será apresentado alguns conceitos sobre *framewoks* e design *patterns* que serão aplicados na proposta desse trabalho.

com a finalidade de estabelecer padrões para a criação e a interpretação de conteúdos para a *Web*. https://www.w3.org/

2.4.1 *Frameworks*

Frameworks são e continuamente tem se tornado importantes no desenvolvimento de aplicações. Eles proporcionam diversos benefícios, entre eles, menos código para projetar e escrever; redução de custos e tempo de desenvolvimento; código mais confiável e robusto; manutenção reduzida e evolução mais simples; melhor consistência e compactabilidade entre aplicações; código de estabilização (menos erros) devido à sua utilização em várias aplicações (SOMMERVILLE, 2011). O autor (JOHNSON, 1997) define *Frameworks* como uma técnica de reutilização orientada a objeto; "Um esqueleto de um aplicativo que pode ser personalizável por um desenvolvedor"e/ou "Um *Design* reutilizável de todo ou parte de um sistema representado por um conjunto de classes abstratas e pela maneira como suas instâncias interagem".

A tecnologia ideal de reutilização fornece componentes que podem ser facilmente conectados para criar um novo sistema. O desenvolvedor não precisa saber como o componente é implementado, sendo necessário aprender como usá-lo. O sistema resultante será eficiente, com manutenabilidade e confiável. Um *Framework* fornece uma maneira padrão para os componentes manipularem erros, trocarem dados e invocarem operações uns sobre os outros. Uma segunda maneira pela qual os *Frameworks* e componentes trabalham juntos é que os *Frameworks* facilitam o desenvolvimento de novos componentes. Os aplicativos parecem infinitamente variáveis, e não importa quão boa seja uma biblioteca de componentes, ela eventualmente precisará de novos componentes (SOMMERVILLE, 2011). Os *Frameworks* permitem criar um novo componente (como uma interface de usuário) com componentes menores (como um [14] *Widget*) e também fornecem as especificações para novos componentes e um modelo para implementá-los. Os *Frameworks* podem ser classificados de diversas formas, inicialmente em dois grupos: *Framework* de Aplicação (FA) e *Framework* de Componentes (FC).

- ***Framework* de Aplicação**

  Segundo (FAYAD; SCHMIDT; JOHNSON, 1999) os *Frameworks* de Aplicação geralmente são classificados como: *Framework* Caixa Branca (FCB), *Framework* Caixa Preta (FCP) ou *Framework* Caixa Cinza (FCC).

  - ***Framework* Caixa Branca (FCB)**: *"white box"*, é mais simples projeta-lo, pois não há necessidade de prever todas as alternativas de implementação possível. O reuso é provido por herança através da implementação de *Template Methods*, ou seja, métodos abstratos, o usuário deve criar subclasses das classes abstratas contidas no *framewok* para criar aplicações especifica. O usuário deve entender detalhes de como o *framework* funciona para poder usá-lo;

[14] Um componente que pode ser utilizado em computadores, celulares, *tablets* e outros aparelhos para simplificar o acesso a um outro programa ou sistema. Eles geralmente contêm janelas, botões, ícones, menus, barras de rolagem e outras funcionalidades.

Figura 10 – Classificação de *Frameworks*.

Fonte: (SHVETS, 2019).

  - ***Framework* Caixa Preta (FCP)**: *"black box"* é mais complexo de projeta-lo por haver a necessidade de fazer previsão. O reuso é por comunicação, ou seja, é instanciado através de configurações e composições, o usuário combina diversas classes concretas existentes no *framework* para obter a aplicação desejada. Não requer compreensão de detalhes internos para produzir uma aplicação;
  - ***Framework* Caixa Cinza (FCC)**: *"gray box"* possui as características conjuntas dos *frameworks* caixa branca e preta, de forma a tentar evitar as desvantagens dos dois. Ele permite um grau de flexibilidade e extensibilidade sem expor informações internas desnecessárias.

- ***Framework* de Componentes**
  Segundo (BARRETO, 2006) um *framework* de componentes é uma entidade de software que provê suporte a componentes que seguem um determinando modelo e possibilita que instâncias destes componentes sejam plugadas no *framework* de componentes. Ele estabelece as condições necessárias para um componente ser executado e regula a interação entre as instâncias destes componentes. Um *framework* de componente pode ser único na aplicação, criando uma ilha de componentes ao seu redor, ou pode cooperar com outros componentes ou *frameworks* de componentes.

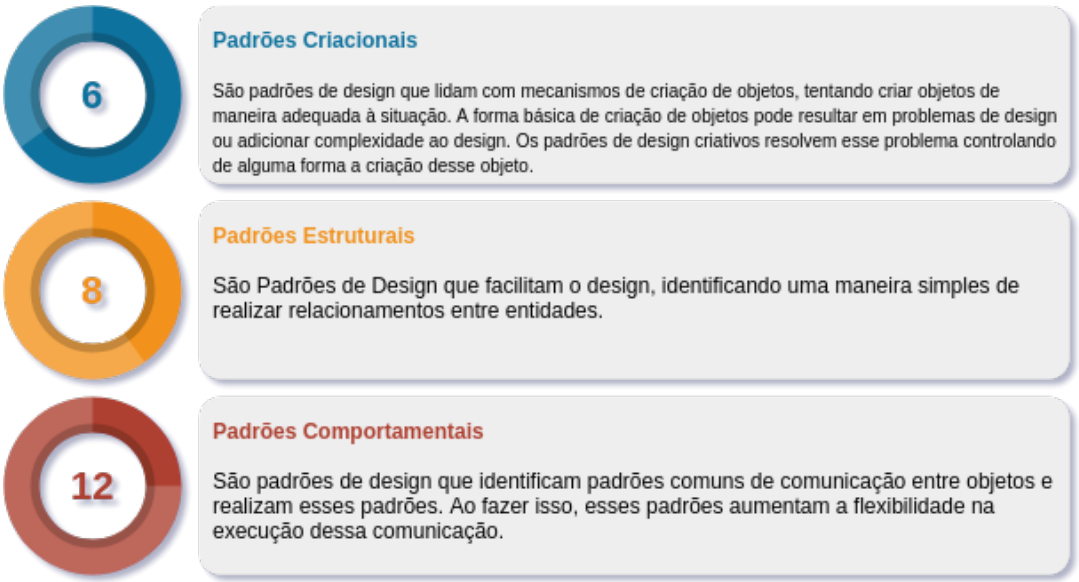
  A principal diferença entre *frameworks* de aplicação e *framework* de componentes é que, enquanto frameworks de aplicações definem uma solução inacabada que gera uma família de aplicações, um *framework* de componentes estabelece um contrato para plugar componentes.

2.4.2 *Design Patterns*

De acordo com o autor (MALDONADO et al., 2002), *Design Patterns* descrevem uma solução para um problema que ocorre com frequência durante o desenvolvimento de software. (SHVETS, 2019) define design *patterns* como uma solução geral repetível para um pro-

blema comumente ocorrido no projeto de software. Um design *patterns* não é um design acabado que pode ser transformado diretamente em código. É uma descrição ou modelo de como resolver um problema que pode ser usado em muitas situações diferentes. *Design Patterns* podem acelerar o processo de desenvolvimento, fornecendo paradigmas de desenvolvimento testados e comprovados. O *design* de *software* efetivo requer a consideração de problemas que podem não se tornar visíveis até mais tarde na implementação. A reutilização de padrões de design ajuda a evitar problemas sutis que podem causar grandes problemas e melhora a legibilidade do código para codificadores e arquitetos familiarizados com os padrões. Os padrões de projetos diferem pela sua complexidade, nível de detalhe e escala de aplicabilidade, como cita (SHVETS, 2019) em uma lista 26 padrões de projetos divididos em três grupos como ilustra Figura 11.

Figura 11 – Grupos de *Design Patterns*.



Fonte: o autor.

A Tabela 2 mostra uma breve descrição dos 26 padrões de projetos. Aqui será levado em consideração apenas os padrões de projetos (Template Method, Factory Method e Adpter) utilizados neste trabalho. Diagramas, detalhes de implementações e códigos de exemplos dos demais padrões de projetos podem ser encontrados nos sites [15]*Source Making Design Patterns* e [16]Refactoring Guro e outros.

2.4.2.1 Template Method

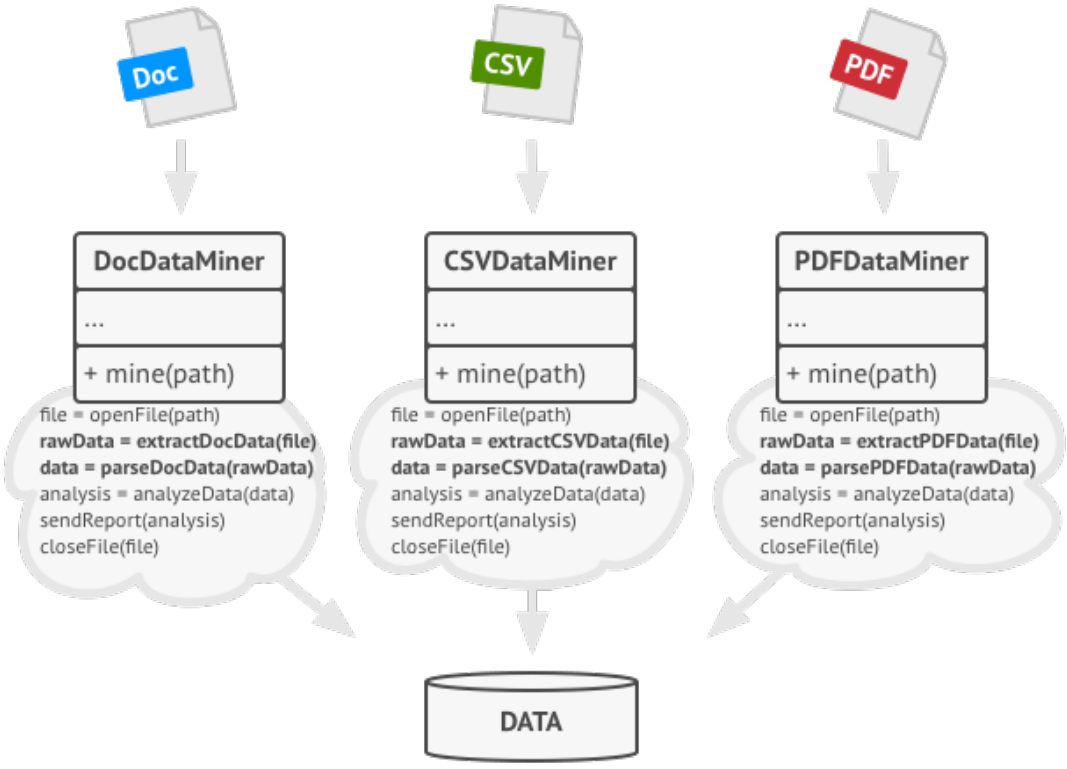
Template Method é um padrão de projeto comportamental que define o esqueleto de um algorítimo na superclasse, mas permite que as subclasses substituam etapas especificas

[15] https://sourcemaking.com/design$_p atterns$
[16] https://refactoring.guru/design-patterns

do algorítimo sem alterar a sua estrutura. Esse padrão é usado com destaque nos *frameworks*. Cada estrutura implementa as partes invariantes da arquitetura de domínio e define [17] *"placeholders"* para todas as opções necessárias ou interessantes de personalização do cliente (SHVETS, 2019).

Um exemplo de problema em que se pode estar aplicando esse padrão de projeto é: em um determinado aplicativo de mineração de dados que analisa documentos corporativos. Os usuários alimentam os documentos do aplicativo em vários formatos (PDF, DOC, CSV) e tentam extrair dados significativos desses documentos em um formato uniforme. A primeira versão do aplicativo poderia funcionar apenas com arquivos DOC. Na versão seguinte, foi capaz de suportar arquivos CSV. Um mês depois, foi adicionado a extração de dados de arquivos PDF.

Figura 12 – *Template Method* - Exemplo de Problema.



Fonte: (SHVETS, 2019).

As classes de mineração de dados continham muito código duplicado "semelhante". Embora o código para lidar com vários formatos de dados seja totalmente diferente em todas as classes, o código para processamento e análise de dados é quase idêntico. Nesse caso, seria ótimo se livrar da duplicação de código, deixando a estrutura do algoritmo intacta. Um outro problema relacionado ao código do cliente que usava essas classes. Ele tinha muitas condicionais que escolhiam um curso de ação adequado, dependendo da

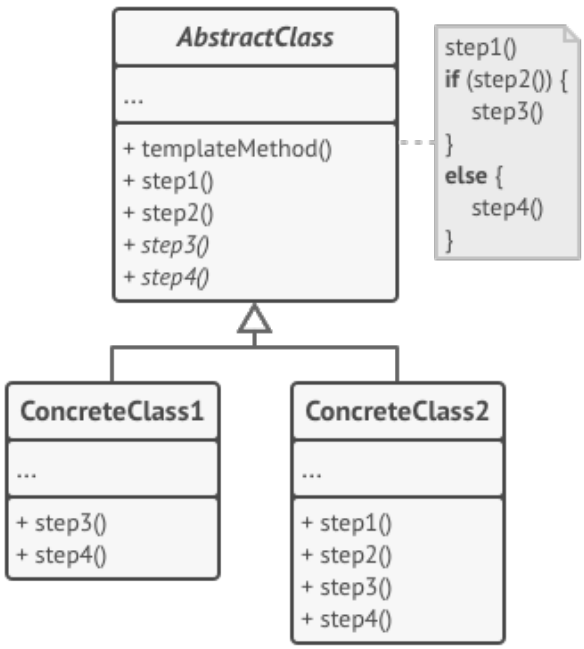
17

classe do objeto de processamento. Se todas as três classes de processamento tivessem uma interface comum ou uma classe base, você seria capaz de eliminar as condicionais no código do cliente e usar o polimorfismo ao chamar métodos em um objeto de processamento.

O padrão *Template Method* sugere que divida um algoritmo em uma série de etapas, transforme essas etapas em métodos e coloque uma série de chamadas para esses métodos dentro de um único "método de modelo". As etapas podem ser abstratas ou ter algumas implementação padrão. Para usar o algoritmo, o cliente deve fornecer sua própria subclasse, implementar todas as etapas abstratas e substituir algumas das opcionais, se necessário (mas não o próprio método de modelo).

Na Figura 13 mostra a estrutura base desse padrão de projeto, a onde a classe abstrata declara métodos que atuam como etapas de um algoritmo, bem como o método de modelo real que chama esses métodos em uma ordem específica. As etapas podem ser declaradas abstratas ou ter alguma implementação padrão. As Classes Concretas podem substituir todas as etapas, mas não o método de modelo em si.

Figura 13 – Estrutura *Template Method*.



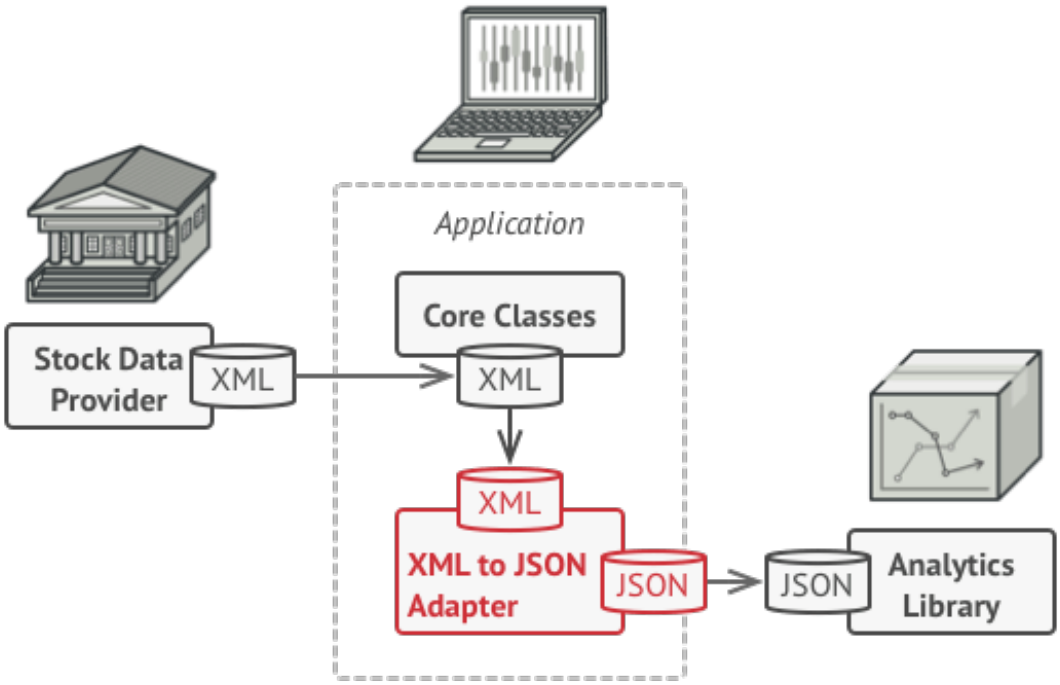
Fonte: (SHVETS, 2019).

Adapter Method é um padrão de projeto estrutural que permite que os objetos com interfaces incompatíveis colaborem (SHVETS, 2019).

Um exemplo de problema em que pode-se está empregando o padrão de projeto *Adapter* é: em uma aplicação que está sendo construída para o monitoramento do mercado de ações, o aplicativo faz o download dos dados de estoques de várias fontes no formato XML e exibe gráficos e diagramas atraentes para o usuário. Em algum momento, foi decidido melhorar o aplicativo adicionando uma analise inteligente de terceiros. O problema é, que

esse recurso consome apenas dados no formato JSON. Para contorna esse problema deve-se ser criado um adaptador para converter a interface de um objeto para outro objeto entende-lo, conforme ilustra a Figura 14.

Figura 14 – *Adapter Method* - Exemplo de Problema.



Fonte: (SHVETS, 2019).

Tabela 1 – Padrão de Projetos.

Padrões		Descrição
Criacional	Abstract Factory	Cria uma instância de várias famílias de classes.
	Builder	Separa a construção do objeto de sua representação.
	Factory Method	Cria uma instância de várias classes derivadas.
	Object Pool	Evite aquisições caras e liberação de recursos pela reciclagem de objetos que não estão mais em uso.
	Prototype	Uma instância totalmente inicializada para ser copiada ou clonada.
	Singleton	Uma classe da qual apenas uma única instância pode existir.
Estrutural	Adapter	Corresponder interfaces de diferentes classes.
	Bridge	Separa a interface de um objeto da sua implementação.
	Composite	Uma estrutura de árvore de objetos simples e compostos.
	Decorator	Adicionar responsabilidades aos objetos dinamicamente.
	Facade	Uma única classe que representa um subsistema inteiro.
	Flyweight	Uma instância refinada usada para compartilhamento eficiente.
	Private Class Data	Restringir o acesso do acessador / mutador.
	Proxy	Um objeto representando outro objeto.
Comportamental	Chain of responsibility	Uma maneira de passar um pedido entre uma cadeia de objetos.
	Command	Encapsular uma solicitação de comando como um objeto.
	Interpreter	Uma maneira de incluir elementos de linguagem em um programa.
	Iterator	Acessar sequencialmente os elementos de uma coleção.
	Mediator	Define a comunicação simplificada entre classes.
	Memento	Capturar e restaurar o estado interno de um objeto.
	Null Object	Projetado para atuar como um valor padrão de um objeto.
	Observer	Uma maneira de notificar a mudança para várias classes.
	State	Alterar o comportamento de um objeto quando seu estado muda.
	Strategy	Encapsula um algoritmo dentro de uma classe.
	Template method	Defer the exact steps of an algorithm to a subclass.
	Visitor	Define uma nova operação para uma classe sem alteração.

Fonte: do autor.

2.5 *LIBRARIES E TOOLKITS* PARA PLN

Existe uma diversidade de [18]*library*, [19]*toolkits* para PLN, que trazem inúmeros recursos para facilitar a construção de aplicações. Nesta seção objetiva-se apresenta um lista dessas ferramentas, sua categorização, análises linguísticas que realizam, idiomas suportados, linguagem de programação em que foram construídas, entre outros aspectos serão destacados. Os critérios utilizado na seleção dessas ferramentas foram: ser um *software open source*, possuir documentação, ter artigo publicado. Soluções comerciais não entraram nessa lista devido ao uso restrito que as mesmas possuí para uso demo.

- ***CogComp NLP*** é uma coleção de várias bibliotecas principais para PLN, desenvolvida pelo Cognitive Computation Group do Departamento de Computação e Ciências da Informação da Universidade da Pennsylvania. Essa ferramenta realiza diversas tarefas de PLN e outros recursos como (nlp-pipeline, core-utilities, corpusreaders, curator, edison, lemmatizer, tokenizer, transliteration, pos, ner, md, relation-extraction, quantifier, inference, depparse, verbsense, prepsrl, commasrl, similarity, temporal-normalizer, dataless-classifier, external-annotators.); o *CogComp NLP* foi construído na linguagem Java e possui API para uso de alguns recursos básicos em *Python*; sua licença de uso é *Academic*, possui uma documentação resumida e [20]ferramenta demo *online* e [21]site oficial. (KHASHABI et al., 2018)

- ***LingPipe*** é um kit de ferramentas para processamento de texto usando linguística computacional. O *LingPipe* é usado para executar tarefas como: encontrar nomes de pessoas, organizações ou locais em notícias; classificação automática de resultados da pesquisa do Twitter em categorias; sugestão de grafias corretas em consultas (BALDWIN; DAYANIDHI, 2014). O *LingPipe* disponibiliza uma API Java para utilização, [22]site oficial e ferramenta [23]demo.

- ***NLTK*** é um *ToolKit* de linguagem natural para criar programas em *Python* para trabalhar com dados de linguagem humana. Ele fornece interfaces fáceis de usar para mais de 50 recursos de corpus e léxicos, junto de bibliotecas de PLN (*tokenization*, *stemming*, *tagging*, *wrappers*, outros.) (LOPER; BIRD, 2002). Possui suporte para os

[18] *Libraries* se refere ao código que fornece funções que você pode chamar do seu próprio código para lidar com tarefas comuns. Por exemplo, uma biblioteca de matemática fornecerá funcionalidades matemáticas comuns, como funções trigonométricas ou logarítmicas. As linguagens de programação geralmente têm bibliotecas para todos os tipos de tarefas, como processamento de dados, plotagem de gráficos, análise de texto, etc. Uma vez incluídas, as bibliotecas economizam o trabalho de escrever todas essas funções.

[19] *Toolkit* é um conjunto de bibliotecas que funcionam em conjunto, mas que podem ser chamadas independentemente.

[20] http://macniece.seas.upenn.edu:4004/

[21] https://cogcomp.org/

[22] http://alias-i.com/lingpipe/index.html

[23] http://alias-i.com/lingpipe/web/demos.html

idiomas: English, French, Dutch. Documentação, Fórum, comunidade ativa através do [24]site, e também diversos livros didáticos com [25]plataforma demo *online*.

- ***SupWSD*** é um sistema para desambiguação de sentido de palavra supervisionado. A estrutura é flexível e permite que o usuário combine diferentes módulos de pré-processamento, selecione extratores de recursos e escolha qual classificador deseja usar. Além disso segundo os autores essa ferramenta é bastante leve e tem requisitos de memórias muito pequeno. Para configurar o seu sistema é necessário utilizar um arquivo XML onde é informado os parâmetros de configuração. A ferramenta está disponível no [26]*Github* sobre a licença de *software* GNU (PAPANDREA; RAGANATO; BOVI, 2017)

- ***PyWSD*** é uma *library*, implementado em *Python* para *Word Sense Disambiguation* (WSD) utilizando as tecnologias *Lesk algorithms* e *Maximizing Similarity*, para a tarefa de desambiguação de sentido no idioma English (TAN, 2014). *PyWSD* está disponível sobre a licença MIT, no [27]repositório da biblioteca você encontra também a documentação de utilização com exemplos práticos e artigo de publicação.

- ***Stanford CoreNLP*** é um conjunto de *libraries* para PLN, que realiza várias análises linguísticas *(tokenize, cleanxml, docdate, ssplit, pos, lemma, ner, entitymentions, regexner, tokensregex, parse, depparse, coref, dcoref, relation, natlog, openie, entitylink, kbp, quote, quote.attribution, sentiment, truecase, udfeats.)*, possui módulos de suporte aos idiomas (Arabic, Chinese, English, English (KBP), French, German, Spanish.) (MANNING et al., 2014). Foi inteiramente construída na linguagem Java e API's para várias outras linguagens, é distribuída sobre a licença de *software* GPL v3. É mantida pelo [28]grupo de pesquisa de PLN da Universidade de Stanford.

- ***SpaCy*** é uma biblioteca industrial para PLN com diversas análises para PLN (Tokenisation, Lemmatisation, Part-of-speech tagging, Entity recognition, Dependency parsing, Sentence recognition, Word-to-vector transformations, Many convenience methods for cleaning and normalising text.), um vasto suporte a mais de 49 idiomas (German, Greek, English, Spanish, French, Italian, Dutch, Portuguese, Afrikaans, Arabic, Bulgarian, Bengali, Catalan, Czech, Danish, Estonian, Persian, Finnish, Irish, Hebrew, Hindi, Croatian, Hungarian, Indonesian, Icelandic, Japanese, Kannada, Lithuanian, Latvian, Norwegian Bokmål, Polish, Romanian, Russian, Sinhala, Slovak, Slovenian, Albanian, Swedish, Tamil, Telugu, Thai, Tagalog, Turkish, Tatar, Ukrainian, Urdu, Vietnamese, Chinese.), construída completamente em Python

[24] https://www.nltk.org/
[25] http://text-processing.com/demo/
[26] https://github.com/SI3P/SupWSD
[27] https://github.com/alvations/pywsd
[28] https://stanfordnlp.github.io/CoreNLP/

e Cython, sendo considerada a ferramenta mais rápida em 2015, 2017 e atualmente (CHOI; TETREAULT; STENT, 2015). possui uma das mais ricas documentações, é distribuída sobre a licença MIT e está disponível no [29]site oficial.

- ***SEMAFOR*** é um *frema-semantic parser open-source*, desenvolvido pelo grupo de pesquisa de Berkeley apoiado pela [30]DARPA com o proposito de automatizar o processo de analise de sentenças em English de acordo com o [31]Berkeley *FrameNet*. Construída inteiramente em C++ e Shell Script (CHEN et al., 2010),podendo ser encontrado no [32]site oficial.

Tabela 2 – Resulmo das Ferramentas.

Nome da Ferramenta	**Categoria**
CogComp NLP	Libraries
LingPipe	Toolkits
NLTK	Toolkits
PyWSD	Libraries
SupWSD	Toolkits
Stanford CoreNLP	Libraries
SpaCy	Libraries
SEMAFOR	Libraries

Fonte: do autor.

2.6 PROCESSAMENTO PARALELO

Através do Processamento Paralelo (PP) é possível utilizar vários núcleos do computador, aumentando a quantidade de cálculos que o programa pode fazer em um determinado período de tempo sem precisar de um processador mais rápido. A ideia principal por trás dessa técnica é dividir um problema em subunidades independentes e usar múltiplos núcleos para resolver essas subunidades em paralelo (LANARO, 2019).

O processamento paralelo é necessário para resolver problemas de grande escala. As empresas por exemplo, produzem enormes quantidades de dados todos os dias, que precisam ser armazenados em vários computadores e analisados. Cientistas e engenheiros executam código paralelo em super computadores para simular sistemas massivos. O processamento paralelo permite o aproveitamento de *Central Processing Unit* (CPU) com vários núcleos, assim como *Graphics Processing Unit* (GPU) que funcionam extremamente bem com problemas altamente paralelos (LANARO, 2019).

[29] https://spacy.io/
[30] https://www.darpa.mil/
[31] http://framenet.icsi.berkeley.edu/
[32] http://www.cs.cmu.edu/ ark/SEMAFOR/

2.6.1 Multi-processos

Um processo é basicamente um programa em execução (PALACH, 2014). O Multiprocessamento é a execução de vários processos simultâneos. Processos separados não compartilham recursos. A comunicação entre processos é dispendiosa e pode prejudicar gravemente o desempenho de programas paralelos. Existem duas maneiras principais para lidar com comunicação de dados em programas paralelos (LANARO, 2019).

- Memória Compartilhada: as subunidades tem acesso ao mesmo tempo. A vantagem dessa abordagem é que você não precisa manipular explicitamente a comunicação, pois suficiente escrever ou ler a partir da memória compartilhada. No entanto, surge prolemas quando vários processos tentam acessar e alterar o mesmo local de memória ao mesmo tempo. Deve-se ter cuidado ara evitar esse conflitos usando técnicas de sincronização.

- Memória Distribuída: nesse modelo, cada processo é completamente separado dos outros e possui seu próprio espaço de memória. A sobrecarga de comunicação é tratada explicitamente entre os processo. A sobrecarga de comunicação é geralmente mais cara em comparação à memória compartilhada, já que os dados podem viajar protecionalmente através de uma interface de rede.

No Apêndice 14 encontra-se um exemplo de como implementar multiprocessamento utilizando a linguagem de programação *Python*.

2.6.2 Multi-Threads

Thread é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente, compartilhando os mesmos recursos como memória (LANARO, 2019). No campo da ciência da computação, uma *thread* é a menor unidade de comandos de programação (código) que um agendador (geralmente como parte de um sistema operacional) pode processar e gerenciar. Dependendo do sistema operacional, a implementação de *thread* varia, mas uma *thread* é normalmente um elemento de um processo. *MultiThreads* implementa mais de uma *thread* para existir e executa em um único processo, simultaneamente. (PALACH, 2014).

Aplicações *multithreads* têm várias vantagens, em comparação com aplicações sequenciais tradicionais:

- Tempo de execução mais rápido: uma das principais vantagens da simultaneidade pro meio do *multithreading* é a aceleração alcança. *Threads* separadas no mesmo programa podem se executadas concorrentemente ou em paralelo, se forme suficiente independentes uma da outra;

- Capacidade de Resposta: um programa *single-threaded* só pode processar uma tarefa de cada vez; portanto, se a *thread* de execução principal bloquear um tarefa de execução longa (ou seja, uma parte da entrada que requer computação e processamento pesados), o programa inteiro não poderá continuar com outra entra e, portanto, aparecerá ser congelado. Usando *threads* separadas para executar computação e permanecer em execução para receber entrada de usuário diferente simultaneamente, um programa *multthreaded* pode fornecer melhor capacidade de respos.

- Eficiência no consumo de recursos: Como mencionamos anteriormente, vários encadeamentos dentro do mesmo processo podem compartilhar e acessar os mesmos recursos. Consequentemente, os programas *multithread* podem servir e processar muitas solicitações de clientes para dados simultaneamente, usando significativamente menos recursos do que seria necessário ao usar programas de *single-threaded* ou *multiprocess*. Isso também leva a uma comunicação mais rápida entre os segmentos.

Dito isto, os programas *multithreaded* também têm suas desvantagens, como segue:

- *Crashes*: mesmo que um processo possa conter várias *threads*, uma única operação ilegal dentro de um *thread* pode afetar negativamente o processamento de todas as outras *threads* no processo e pode travar o programa inteiro como resultado.

- Sincronização: mesmo que o compartilhamento dos mesmos recursos possa ser uma vantagem em relação aos programas tradicionais de programação sequencial ou de multiprocessamento, uma consideração cuidadosa também é necessária para os recursos compartilhados. Normalmente, as *threads* devem ser coordenadas de maneira deliberada e sistemática, para que os dados compartilhados sejam computados e manipulados corretamente. Problemas não intuitivos que podem ser causados pela coordenação descuidada de *threads* incluem [33]*Deadlocks*, [34]*Livelocks*.

No Apêndice 15 encontra-se um exemplo de como implementar *multithreads* utilizando a linguagem de programação *Python*.

2.6.3 Engenharia de software baseada em componentes

Na criação de software existe em algum momento o reúso de software. Isso acontece muitas vezes mesmo que informal pois os desenvolvedores da aplicação percebem que podem

[33] *Deadlock* (interbloqueio, blocagem, impasse), no contexto de sistemas operacionais (SO), refere-se a uma situação em que ocorre um impasse, e dois ou mais processos ficam impedidos de continuar suas execuções - ou seja, ficam bloqueados, esperando uns pelos outros.

[34] A *livelock* é semelhante a um impasse, exceto que os estados dos processos envolvidos na *livelock* mudam constantemente em relação um ao outro, nenhum progredindo. *Livelock* é um caso especial de fome de recursos; a definição geral afirma apenas que um processo específico não está progredindo.

reutilizar códigos fazendo algumas adaptações por mais simples que seja. Abordagens orientadas a reúso dependem de uma ampla base de componentes reusáveis de software e de um *framework* de integração para composição desses componentes.

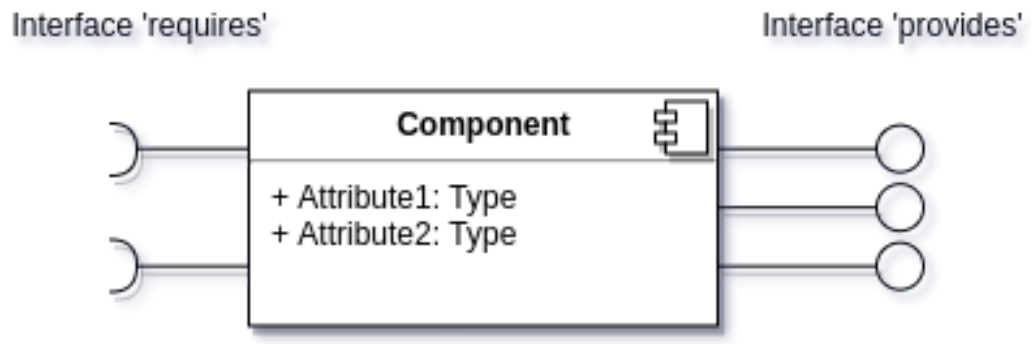
Essa abordagem segundo (SOMMERVILLE, 2011) proporciona diversas vantagens:

- **Reusabilidade**: os componentes são usualmente estruturados para serem reutilizados em diferentes cenários e aplicações variáveis. Entretanto, alguns componentes precisam ser estruturados para tarefas específicas.
- **Substituição**: os componentes podem ser substituídos por outros componentes similares.
- **Contexto não específico**: os componentes são estruturados para operar em diferentes ambientes e contextos. Informações específicas, como estado do dado, devem ser enviadas para o componente em vez de serem incluídas ou acessadas pelo componente.
- **Extensibilidade**: um componente pode ser estendido a partir de um outro componente para fornecer um novo comportamento.
- **Encapsulamento**: os componentes expõem uma *interface* para os invocadores utilizarem suas funcionalidades e não revela detalhes do seu processo interno ou alguma variável interna e estado.
- **Independência**: os componentes são estruturados para ter o mínimo de dependências com outros componentes. Por isso, componentes podem ser disponibilizados em um ambiente apropriado sem afetar outros componentes ou sistemas.
- ***Deploy***: a compatibilidade de novas versões, quando disponíveis. Você pode substituir a versão existente, sem impacto, em outros componentes do sistema como um todo.
- **Redução de custo**: o uso do componente de terceiros permite a redução do custo do desenvolvimento e manutenção.

Um componente deve ser: [1] Padronizado, significa que um componente usado precisa obedecer a um modelo de componentes padrão. Esse modelo pode definir as interfaces de componentes, metadados de componente, documentação, composição e implantação. [2] Independente, significa que um componente deve ser possível compor e implantá-lo sem precisar usar outros componentes especifico. Nessa situação, em que o componente precisa dos serviços prestados externamente, estes devem ser explicitamente definidos em uma especificação de interface '*require*'. [3] Passível de composição, para um componente ser composto, todas as interações externas devem ter lugar por meio de interfaces publicamente definidas. Além disso, ele deve proporcionar acesso externo a informações sobre

si, como seus métodos e atributos. [4] Implantável, deve ser autocontido, capaz de operar como uma entidade autônoma em uma plataforma de componentes que forneça uma implementação do modelo de componentes, o que geralmente é implantado como um serviço. [5] Documentação, deve ser completamente documentado, para que os potenciais usuários possam decidir se satisfazem a suas necessidades. A sintaxe e, idealmente, a semântica de todas as interfaces de componentes devem ser especificadas.

Figura 15 – Modelo de Interface de Componente.



Fonte: própria.

- *Interface requires*: define os serviços que são requeridos e que devem se fornecidos por outros componentes (opcional).

- *Interface provides*: define os serviços que são providos pelo componente par outros componestes (obrigatório).

2.7 CONCLUSÃO

Este capítulo apresentou vários conceitos que são extremamente necessários para a construção deste trabalho, abordando breves resumos de diversas áreas de conhecimento que constroem essa fundamentação teórica. Como referido, existem muitas ferramentas que realizam tarefas de Processamento de Linguagem Natural (PLN), sendo que cada uma com seus objetivos, umas mais específicas para tarefas de PLN e diversas ferramentas mais abrangentes, outras apenas suporte para linguagens de programação distintas. Também existem técnicas que podem ajudar a melhora o desempenho do sistema consequentemente reduzir o custo computacional. As abordagens para construção de software como *Design Pattens* e arquitetura baseada em componentes, entre outros, são extremamente necessárias para projetar sistemas estáveis, confiáveis e que supra as necessidades dos usuários. No Capítulo 3 será apresentado e discutido os principais trabalhos relacionados a proposta dessa dissertação.

3 TRABALHOS RELACIONADOS

Neste capítulo apresenta-se uma discussão dos trabalhos relacionados a esta proposta. O restante deste capítulo está estruturado da seguinte forma: na Seção 3.1 apresenta-se os principais aspectos dos trabalhos relacionados; Seção 3.2 os trabalhos: (KARIMZADEH et al., 2019), ***CLAMP – a toolkit for efficiently building customized clinical natural language processing pipelines*** (SOYSAL et al., 2017), ***xTAS: Text Analysis in a Timely Manner*** (ROOIJ et al., 2012), ***Jigg: A Framework for an Easy Natural Language Processing Pipeline*** (NOJI; MIYAO, 2016) e ***GATE: an Architecture for Development of Robust HLT Applications*** (CUNNINGHAM et al., 2002). Na Seção 3.3, realiza-se uma análise comparativa e por fim, na Seção 3.4, o posicionamento do presente trabalho quanto a sua contribuição.

3.1 PRINCIPAIS ASPECTOS DOS TRABALHOS SELECIONADOS

Os trabalhos selecionados e apresentados nesse capítulo, foram definidos com base na literatura seguindo os seguintes critérios de inclusão:

- Ser uma ferramenta para processamento de linguagem natural.
- Realizar pelo menos um tipo de análise linguística em um dos níveis da linguagem (morfológica, léxica, sintática ou semântica).
- Abordar alguma das variáveis de estudo (otimização de processamento, customização de *pipeline*, integração de recursos linguísticos).
- Está disponível na *internet*.
- Possui documentação de uso.
- Ser produto ou parte de uma publicação científica.

Muitas ferramentas contemplam esses requisitos, entretanto, não seria possível descrever todas elas nesse trabalho, sendo assim, destaca-se nesse capítulo somente aquelas que se aproximam do trabalho proposto. Esses trabalhos são discutidos e comparados de acordo com os aspectos elencados a seguir, a saber que tais aspectos foram definidos por estarem presente tanto na literatura como nos trabalhos relacionados e por jugar ser necessários para o desenvolvimento desse trabalho.

- **Domínio**: Indica se a solução construída é dependente de domínio.
- **Análises Linguísticas**: indica quais os tipos de análise de PLN que a solução possui. Isto permite aferir o quão abrangente a ferramenta é para analisar dados

textuais, e quais os níveis de análises linguísticas (morfológica, léxica, sintática, semântica, pragmática) ela consegue cobrir.

- **Ferramentas Externas**: indica se a solução construída faz uso de ferramentas de PLN de terceiros. Isto demonstra se a ferramenta incorpora algorítimos e técnicas adotadas pelos grupos e comunidades de pesquisa em PLN.

- ***Pipeline* Customizado**: indica se a solução possui alguma estrutura (*Wrapper*, *plugins*, etc.) que permita, com facilidade, o usuário adicionar novas análises provindas de outras ferramentas de PLN ao pipeline proposto. Isto demonstra o quanto extensível é a ferramenta.

- ***API***: indica se a solução possui um *Application Programming Interface*, em português, significa *Interface* de Programação de Aplicações, um conjunto de rotinas e padrões estabelecidos para a utilização das suas funcionalidades acessíveis somente por programação. Isso permite que os usuários possam construir novas aplicações, não pretendendo envolver-se em detalhes da implementação, mas apenas usar seus serviços.

- ***API RESTFul***: indica se a solução possui uma *Application Programming Interface Representational State Transfer*, em português *Inteface* de Programação de Aplicações com Transferência de Estado Representacional, uma representação padronizada, verbos e métodos usados, bem como, URLs. Isso permite que o usuário possa criar aplicações que suportam solicitações GET, PUT, POST, DELETE, comunicações frequentemente utilizadas no desenvolvimento de serviços da web para conectar e integrar serviços na *web*.

- **GUI**: indica se a solução possui uma *Graphical User Interface* no português *Interface* Gráfica para o Usuário. Isto permite uma interação do usuário com a aplicação por meio de elementos gráficos como ícones e indicadores visuais; dessa forma, usuários leigos podem experimentar a ferramenta, sem precisar escrever nenhuma linha de código, e até mesmo utiliza-lá para fins didáticos.

- **Estrategia de Processamento**: indica se a solução utiliza alguma das técnicas como (*threads*, multiprocessamento, GPU, computação distribuída, etc.) para otimizar o processamento. Tal aspecto demonstra o quanto a solução é robusta para cenários mais complexo, podendo ser usada para processar corpus extensos, compartilhando recursos de várias máquinas, reduzindo-se assim, o tempo de processamento.

- **Banco de Dados**: indica se a solução possui alguma *interface* de conexão com banco de dados, pois o uso de banco de dados otimiza o processamento de dados, em relação

a armazenamento de dados em arquivos de texto a onde existente de latência [1]I/O. O uso de banco de dados permite também, salvar o resultado da análise do *dataset* processado e disponibilizar os dados para consulta posteriormente. Em caso de erros durante o processamento dos dados, possibilita ao usuário reprocessar o *dataset* a partir de uma de um ponto específico, evitando o reprocessamento completo.

- **Estatísticas do *Dataset***: indica se a solução possui algum recurso estatístico para descrever. Isso permite que o usuário possa compreender melhor de forma geral os dados que ele está analisando.

- **Anotação do *Dataset***: indica se a solução possui algum recurso que realiza anotação do *dataset*, seja usando um formato especifico, ou linguagens de marcação como *XML* ou *JSON*.

- **Arquitetura/Padrões de Projetos**: indica quais as boas práticas da engenharia de software utilizadas. Permite visualizar as decisões tomadas para solucionar problemas correntes e recorrentes no desenvolvimento da solução, bem como a sua robustez e flexibilidade, que refletem diretamente no aumento da qualidade do código tornando-o elegante e reusável.

- **Requisitos do Sistema**: indica quais são requisitos base do sistema. Informa as dependências que o sistema tem (linguagem de programação), o que impacta diretamente na sua utilização dependendo do projeto a ser aplicado.

- **Formato de Saída**: indica quais os formatos de arquivos de saída a ferramenta disponibiliza. Isto indica quais os formato de arquivos são gerados após as análises, tais arquivos (*XML*, *JSON*, *FoLia*, etc.) podem ser consumidos depois por sistemas, aplicações de representações de conhecimento ou aprendizado de máquina.

- **Licença**: indica a natureza do sistema, se é proprietário ou *open source*. O *software*, com código fonte aberto, permite que outros pesquisadores possam estudar e melhorar as soluções já existentes, sendo assim, limita-se a discutir apenas estruturas abertas livremente disponíveis.

- **Documentação**: indica se o artefato construído possui documentação. Isto impacta na qualidade e credibilidade do que foi desenvolvido. A documentação permite ter uma comunicação clara, histórico de ações para avaliações mais minuciosas, maior controle para o desenvolvimento do projeto, alinhamento de informações e base para tomada de decisões.

[1] Na ciência da computação, I/O é um termo utilizado para designar os sistemas que fazem uso intensivo de entrada/saída (I/O).

3.2 APRESENTAÇÃO DOS TRABALHOS

Nesta seção apresenta-se os principais trabalhos relacionados a esta proposta, todos os sistemas investigados, listados na Tabela 9, foram escolhidos dentro dos aspectos citados na Seção 3.1. A saber: *GeoTxt: A scalable geoparsing system for unstructured text geolocation* (KARIMZADEH et al., 2019), *CLAMP – a toolkit for efficiently building customized clinical natural language processing pipelines* (SOYSAL et al., 2017), *xTAS: Text Analysis in a Timely Manner* (ROOIJ et al., 2012) e *Jigg: A Framework for an Easy Natural Language Processing Pipeline* (NOJI; MIYAO, 2016), *GATE: an Architecture for Development of Robust HLT Applications* (CUNNINGHAM et al., 2002) e o *FreeLing 3.0: Towards Wider Multilinguality* (PADRÓ; STANILOVSKY, 2012). Estes trabalhos foram selecionados a partir de uma busca na internet com a seguinte *string "Framework Natural Language Processing"* em bibliotecas digitais (i.e, Google Scholar, IEEExplore, ACM, , Scopus, ScienceDirect e SpringerLink), bem como na verificação dos trabalhos relacionados dos mesmo. Embora existam muitas pesquisas como (*v3NLP Framework: Tools to Build Applications for Extracting Concepts from Clinical Text* (DIVITA et al., 2016), *Zemberek, an open source NLP framework for Turkic Languages* (AKIN; AKIN, 2007), *NLP Lean Programming Framework: Developing NLP Applications More Effectively* (SCHREIBER; KRAFT; ZÜNDORF, 2018), *RetriBlog: An architecture-centered framework for developing blog crawlers* (FERREIRA et al., 2013), Um *framework* para desambiguação lexical com base no enriquecimento da Semântica de *Frames* (MATOS; SALOMÃO, 2014), *A Framework for Automated Corpus Compilationfor KeyXtract Twitter Mode* (WEERASOORIYA; PERERA; LIYANAGE, 2017), *SLING: A framework for frame semantic parsing* (RINGGAARD; GUPTA; PEREIRA, 2017), *NLP Lean Programming Framework Developing NLP Applications More Effectively* (SCHREIBER; KRAFT; ZÜNDORF, 2018), *Heracles: A framework for developing and evaluating text miningalgorithms* (SCHOUTEN et al., 2019)) que propõem *Frameworks*, *Librarys*, *ToolKits*, *Pakages*, etc, soluções tanto [2]*Open Source* como proprietárias para processamento de linguagem natural com resultados bem avançados, pouco tem sido voltado para a integração desses trabalhos, sendo essa integração o foco deste trabalho. Entretanto, citar todos eles nesse trabalho se torna praticamente inviável, sendo assim aqui consta apenas aqueles que no entender são mais próximos do que será proposto.

3.2.1 *GeoTxt: A Web API to Leverage Place References in Text* (KARIMZADEH et al., 2019).

O trabalho proposto por (KARIMZADEH et al., 2019), descreve um sistema de *geoparsing* escalável para reconhecimento e geolocalização de nomes de lugares em textos não estruturados. Segundo os autores, por estimativa da indústria, uma grande parte de todos

[2] Termo em Inglês que significa código aberto. Isso diz respeito ao código fonte de um software, que pode ser adaptado para diferentes fins.

os dados gerados hoje é desestruturados e a maioria dos dados não estruturados está na forma de documentos textuais livres, cerca de 60%. Embora a recuperação da informação tenha evoluído significativamente, ainda não é capaz de identificar informações relacionadas ao local incorporadas em texto. Sendo assim, os autores propõem o trabalho denominado de [3]GeoTxt, esse sistema fornece seis algoritmos de reconhecimento de entidade nomeadas (NER) provenientes de seis ferramentas de PLN para reconhecimento de nome de local, utilizando um mecanismo de busca corporativo para indexar, classificar e recuperar topônimos, possibilitando o *geoparsing* escalável para fluxo de texto. Isso aponta que uma fusão correta de análises, providas da integração de ferramentas de PNL, pode proporcionar melhores resultados na análise de texto.

A ferramenta *GeoTxt*, foi desenvolvida na linguagem Java utilizando o [4]Play Framework para expor as funcionalidades como [5]*endpoints* da *API* web e renderizar a interface gráfica do usuário, como ilustra a Figura 17. Os aplicativos desenvolvidos por terceiros podem consultar o serviço utilizando solicitações *HTTP GET* ou *POST* e receber a resposta do *geoparsed* como objeto [6]*GeoJSON FeatureCollection*, criado excepcionalmente para facilitar o uso com armazenamento, análise e visualização de dados.

Todas as ferramentas externas suportadas pelo *GeoTxt* foram desenvolvidas na mesma linguagem de programação (JAVA), o que permite integrar as análises de PLN das ferramentas sem problemas de interoperabilidade no pipeline de análise, problema que geralmente ocorre quando se tenta integrar análises de ferramentas de PLN desenvolvidas em linguagens de programação diferentes. O *GeoTex* foi projetado sob o modelo arquitetural Cliente/Servidor, a Figura 16, exibe um esquema explanando a comunicação entre seus principais componentes.

Essa arquitetura, ilustrada anteriormente, funciona da seguinte forma: o texto recebido é processado em um *pipeline* linear. Primeiro é realizado um pré-processamento limpando o texto para garantir a codificação e decodificação adequada das entidades *HTML* e normaliza as *hastags* de acordo com a capitalização de caracteres. Por exemplo, *#NewYork* é convertido para *New York*. Então, o *NER* é executado para extrair nomes de lugares do texto. O componente **Geocoder** usa nomes dos locais extraídos para recuperar uma lista de topônimos relevantes do índice no [7]Apache Solr. O índice do ***GeoNames gazzetter*** executa uma resolução de topônimos e atribui ao topônimo mais provável como o candidato principal a retornar na lista de classificação.

No trabalho *GeoTxt: a web API to leverage place references in text* de (KARIMZADEH

[3] http://www.geotxt.org/

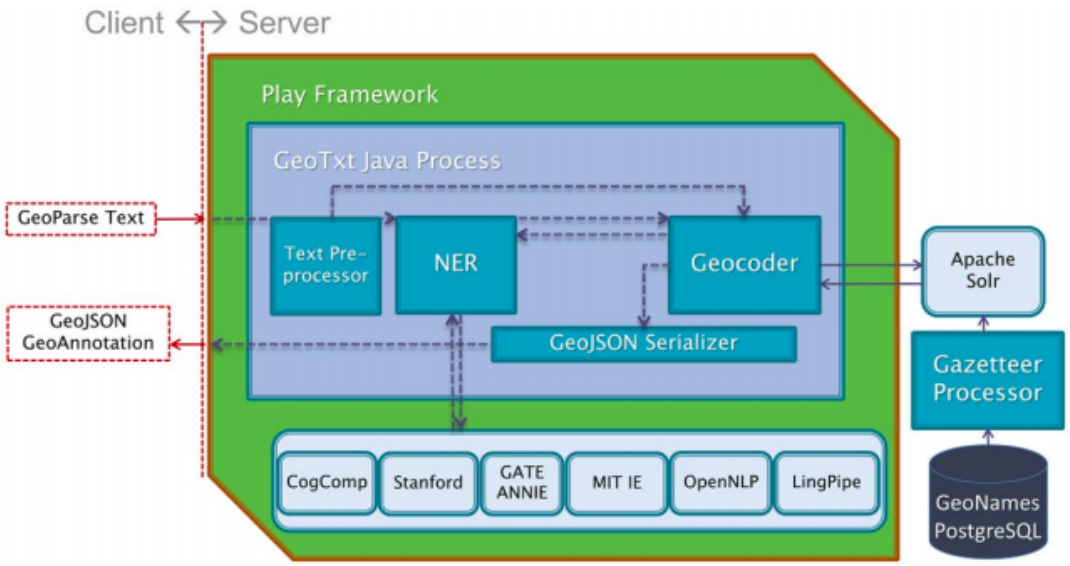
[4] O *Play Framework* é uma estrutura de aplicativo da Web de código aberto que segue o padrão arquitetural de modelo – exibição – controlador. É escrito em *Scala* e utilizável em outras linguagens de programação que são compiladas no *Bytecode* da *JVM*, por exemplo. Java. https://www.playframework.com/

[5] Uma URL onde um serviço pode ser acessado por uma aplicação cliente.

[6] http://www.geotxt.org/api/

[7] http://lucene.apache.org/solr/

Figura 16 – Arquitetura e Componentes do *GeoTxt*.



Fonte: KARIMZADEH et al.

et al., 2013), conforme a Figura 17 exibe, os autores apresentam uma [8]interface gráfica web do *GeoTxt*, onde o usuário insere um texto na caixa de texto e seleciona o tipo de analise a ser executada. Após processado, um arquivo JSON é retornado contendo as informações de geolocalizações identificada. No caso do evento, as localizações identificadas são as cidades de *San Francisco* e *Los Angeles*.

Em resumo, o trabalho apresenta um sistema para processamento de texto que, segundo os autores, é capaz de reconhecer e geo-localizar nomes de lugares em texto. O *GeoTxt* parece ser uma ferramenta bastante promissora para encontrar entidades nomeadas em texto segundo seus resultados métricos, porém, ele não abrange outros tipos de análises linguística e não possui evidencia da possibilidade de adicionar novas análises do tipo *NER* em seu *pipeline*, sendo assim, entende-se que a ferramenta não possui uma camada de personalização, assim, o usuário dependerá de outras ferramentas para integrar na sua aplicação para completar seu *pipeline*, quando necessário. Como também, segundo o que foi pontuado na Tabela 3, essa ferramenta não possui uma *API* que possibilite a sua utilização de forma *offline* e não à evidências que ela utiliza alguma estratégia para otimização (computação distribuída, multiprocessamento), nem a capacidade de analisar estatísticas em dados textuais. A Tabela 3, resume as característica do sistema, dentro dos aspectos de seleção listados na Seção 3.1.

[8] http://www.geotxt.org/

Figura 17 – *GUI* do *GeoTxt*.

Fonte: KARIMZADEH et al.

Tabela 3 – Características do Sistema *GeoTxt* de KARIMZADEH et al. .

Domínio	Geográfico (Dependente)	
Análises Linguísticas	Pré-processamento	limpeza de texto.
	Léxica	NER
	Sintática	-
	Semântica	-
Ferramentas Externas	*CogComp*, *Stanford CoreNLP*, *GATES*, *ANNIE*, *MIT IE*, *OpenNLP*, *LingPipe*.	
Pipeline Customizado	-	
API	-	
API RESTFul	+	
GUI	+	
Estrategia de Processamento	-	
Base de Dados	PostgreSQL	
Estatísticas do Corpus	-	
Arquitetura/Padrões de Projeto	Cliente/Servidor	
Requisitos do sistema	Java	
Formato de Saída	JSON	
Licença	GNU	
Documentação	+	

Fonte: do Autor.

3.2.2 *CLAMP – a toolkit for efficiently building customized clinical natural language processing pipelines* (SOYSAL et al., 2017).

O trabalho de (SOYSAL et al., 2017) apresenta o [9]*CLAMP Clinical Language Annotation, Modeling e Processing*. Segundo os autores, os sistemas existentes de PLN de natureza clínica geral, tais como [10]*MetaMap*, [11]*Clinical Text Analysis* e [12]Knowledge Extraction System, foram aplicados com sucesso na tarefa de extração de informações de textos clínicos. No entanto, os usuários finais geralmente precisam personalizar os sistemas existentes para suas tarefas individuais, o que pode exigir habilidades substanciais de PLN. Sendo assim, os autores propõem o *CLAMP*, um [13]*Toolkit* clínico para PLN que fornece não apenas componentes de PLN, mas também uma *interface* gráfica amigável que ajuda os usuários a construírem rapidamente *pipelines* de PLN personalizados para suas aplicações individuais.

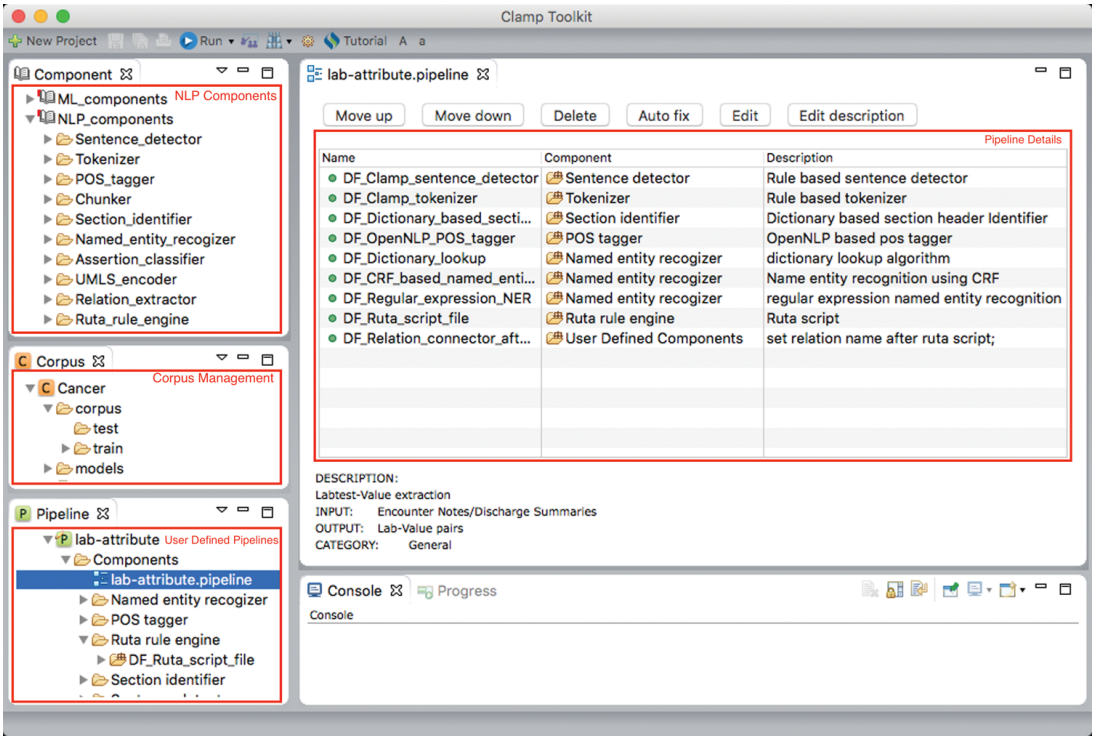
O *CLAMP* é uma ferramenta implementada na linguagem *JAVA* como um aplicativo de *desktop*. Baseia-se no [14]*Apache Unstructured Information Management Architecture (UIMA) Framework* para maximizar a sua interoperabilidade com outros sistemas baseados em [15]UIMA como [16]cTAKES. O *CLAMP* também suporta o Framework [17]*Apache UIMA Asynchronous Scaleout (AS)* para processamento assíncrono em ambiente distribuído.

A *interface* gráfica do *CLAMP* foi construída sobre o [18]*Eclipse Framework*, que fornece componentes integrados para o desenvolvimento de interfaces interativas. Na Figura 18 mostra uma captura de tela da interface principal do *CLAMP* para construir um pipeline de PLN. Os componentes de PLN integrados são listados na paleta superior esquerda e a paleta de gerenciamento de dados está posicionado na área central esquerda. Os pipelines de PLN definidos pelo usuário são exibidos na paleta esquerda inferior. Os detalhes de cada *pipeline* são exibidos na área central depois que os usuários clicam em um *pipeline*. Um *pipeline* pode ser criado visualmente ao arrastar e soltar componentes na janela do meio, seguindo ordens específicas (por exemplo, *tokenizer* dever ser antes do *NER*). Depois de selecionar os componentes de um *pipeline*, os usuários podem clicar em cada componente para personalizar suas configurações. Por exemplo, para componentes *NER*, baseados em expressões regulares ou baseados em dicionários, os usuários podem especificar suas próprias expressões regulares ou arquivos de dicionários. Para *NER* baseado em

[9] https://clamp.uth.edu/
[10] https://www.nlm.nih.gov/research/umls/implementation$_r esources/metamap.html$
[11] https://ieeexplore.ieee.org/document/7550908
[12] https://ieeexplore.ieee.org/abstract/document/8327424
[13] Conjunto de rotinas que auxiliam numa tarefa.
[14] ://uima.apache.org/
[15] Uma solução flexível para dimensionamento de pipelines de PLN mantida pela Apache Foundation
[16] Sistema de PLN para extração de informações de prontuários clínico eletrônico. https://ctakes.apache.org/
[17] https://uima.apache.org/doc-uimaas-what.html
[18] https://www.eclipse.org/modeling/emf/

aprendizado de máquina, os usuários podem trocar o modelo de aprendizado de máquina padrão com modelos treinados em dados locais.

Figura 18 – *Interface* do usuário para construir um *pipeline* no CLAMP.



Fonte: SOYSAL et al.

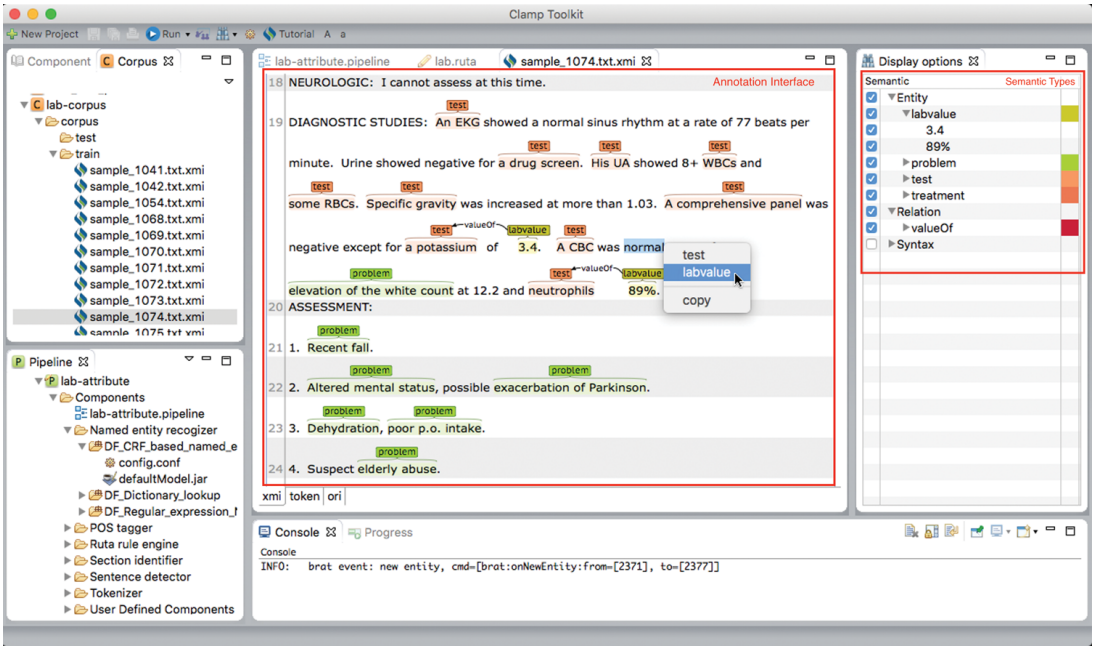
Para facilitar a construção de módulos *NER* baseados em aprendizado de máquina em dados locais, o *CLAMP* fornece uma interface para anotação de corpus e treinamento de modelo. A *GUI* mostrada na Figura 19 foi desenvolvida aproveitando a ferramenta [19]Ambiente online para anotação de texto colaborativo. *Brat Annotation* que permite ao usuário definir tipos de entidade de interesse e anotá-los seguindo as diretrizes. Depois de terminar a anotação, o usuário pode clicar no ícone de treinamento para criar modelos de [20]CRF - Conditional Random Fields usando o corpus anotado. O sistema relatará automaticamente seu desempenho com base nas configurações de avaliação especificadas pelo usuário podendo selecionar diferentes tipos de recursos para construir os modelos *NER* baseados em *CRF*.

Em resumo, o trabalho apresenta uma ferramenta que permite a customização de *pipeline* de PLN, permitindo realizar anotação de dados textuais. Porém, a ferramenta é especifica de domínio (Geográfico), sendo necessárias muitas alterações para a ferramenta ser empregada em outros domínios. Também não fica evidente como adicionar um novo

[19] https://brat.nlplab.org/

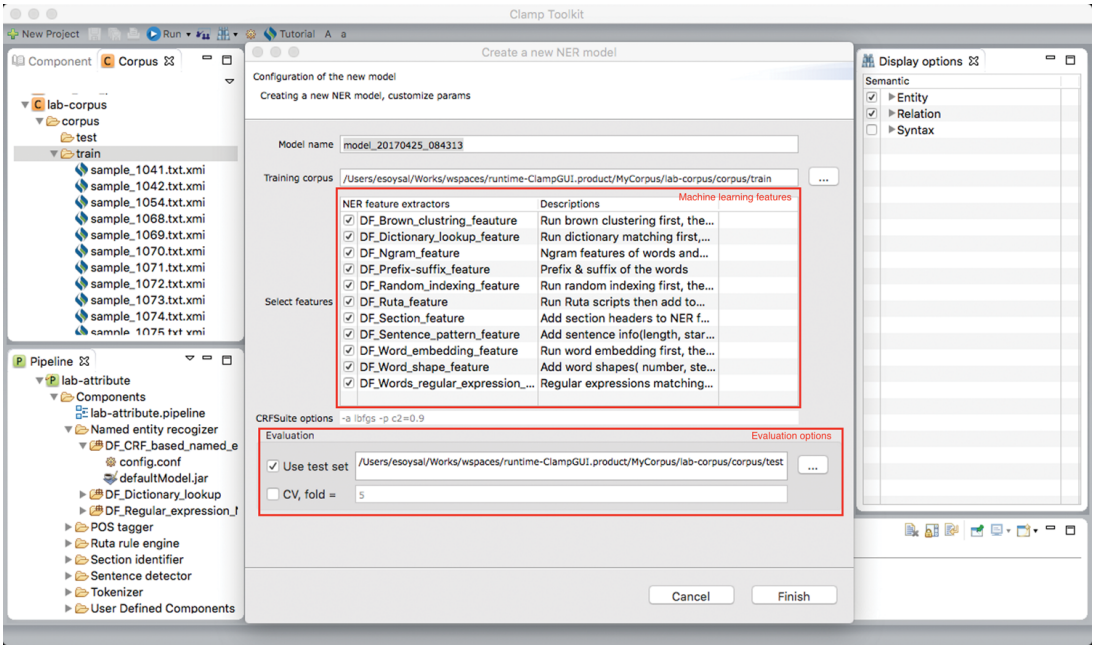
[20] Um método de aprendizado supervisionado bom para diferentes tarefas de segmentação e sequenciamento: Extração de palavras-chave, Reconhecimento de Entidade, Análise de sentimentos, Marcação de parte da fala e Reconhecimento de fala.

Figura 19 – Interface do CLAMP para anotar entidades e relações.



Fonte: SOYSAL et al.

Figura 20 – Interface para selecionar recursos e opções de avaliação para construir modelos NER baseados em aprendizado de máquina usando o CLAMP.



Fonte: SOYSAL et al.

componente de análise ao *pipeline* da ferramenta, se suporta ou não incorporar componentes externos desenvolvidos em outras linguagens de programação; não descreve a arquitetura e nem padrões de projetos utilizados no desenvolvimento; não realiza estatística dos dados textuais; não trabalha com sistema de banco de dados. A Tabela 4, resume

as características do sistema, entre outras observações dentro dos aspectos de seleção listados na Seção 3.2.2.

Tabela 4 – Características do Sistema CLAMP de SOYSAL et al. .

Domínio	Médico (Dependente)	
Análises Linguísticas	Pré-processamento	*Tokenization*
	Léxica	NER, POS
	Sintática	*Chunker*
	Semântica	-
Ferramentas Externas	OpenNLP	
Pipeline Customizado	+	
API	-	
API RESTFul	-	
GUI	+	
Estrategia de Processamento	+	
Base de Dados	-	
Estatísticas do Corpus	-	
Arquitetura/Padrões de Projeto	*Framework Eclipse*	
Requisitos do sistema	Java	
Formato de Saída	Texto	
Licença	*Academic Free*	
Documentação	+	

Fonte: do Autor.

3.2.3 *Jigg A Framework for an Easy Natural Language Processing Pipeline* (NOJI; MIYAO, 2016)

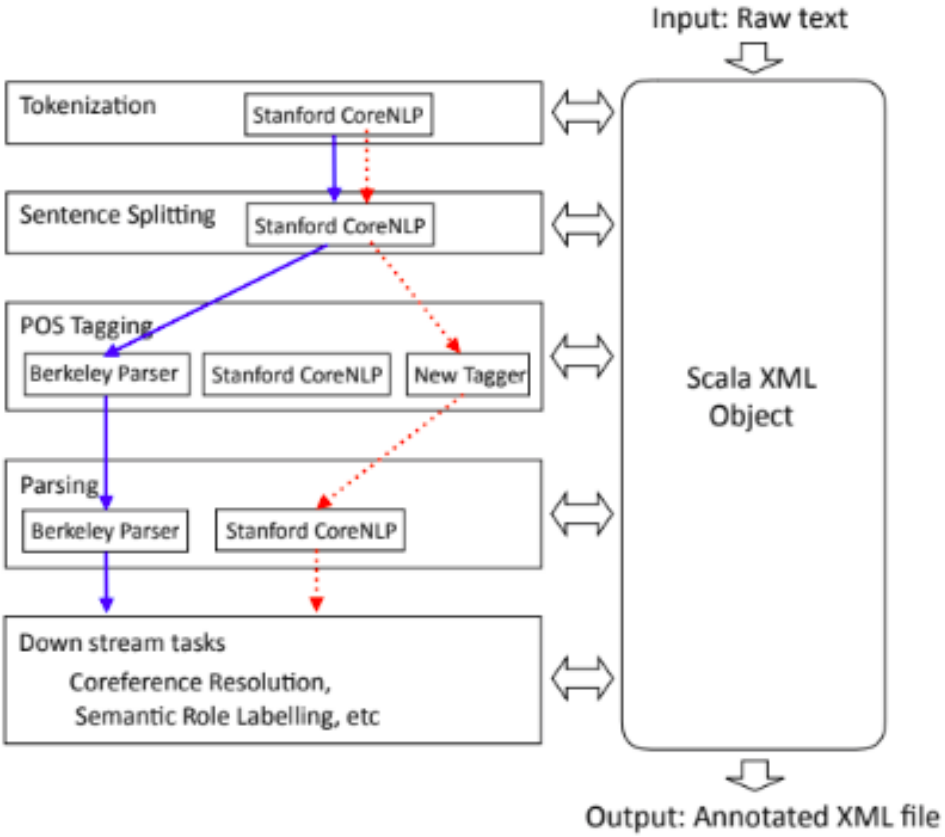
O trabalho de (NOJI; MIYAO, 2016) apresenta o *Jigg* um *Framework* que visa facilitar a adição de novas análises a um pipeline de PLN. Sua arquitetura é inspirada na ferramenta *Stanford CoreNLP*, e permite que o usuário construa um pipeline escolhendo as ferramenta em cada etapa do *pipeline* por meio de uma interface de linha de comando, ou seja, essa ferramenta é uma estrutura de integração de várias ferramentas de PLN.

Segundo os autores, um sistema comum de PLN funciona como um componente em pipeline. Por exemplo, um analisador sintático normalmente requer que uma sentença de entrada corretamente simbolizada ou atribuída a marcação de parte da fala (POS). As aŕvores sintáticas fornecidas pelo analisador podem ser necessárias em outras tarefas posteriores, como resolução de referência e rotulagem de função semântica. Embora essa abordagem baseada em pipeline tenha obtido bastante sucesso devido à sua modularidade, ela sofre vários inconvenientes do ponto de vista de uso e desenvolvimento de *software*: para o usuário, construir um *pipeline* conectando várias ferramentas existentes de forma que agregue as saídas de cada ferramenta isso se torna muito trabalhoso, pois, em geral, cada sistema envia seus resultados com formatos diferentes; no ponto de vista de pesquisadores

ou desenvolvedores de ferramentas de tarefas [21] *downstream*, aguardar o *pipeline* completo é tedioso e consome muito tempo.

Na Figura 21 a arquitetura geral do *pipeline* do *Jigg* é ilustrada. Um *pipeline* é construído escolhendo as ferramentas de anotações em cada etapa, por exemplo: as linhas em negrito ou pontilhadas na figura representa esse fluxo em cada processo do *pipeline*. As anotações são executadas em um objeto *XML Scala*. Também é possível observar uma semelhança predominante quando comparada com a arquitetura de *pipeline* do *CoreNLP*, diferenciando principalmente no agrupamento das análises. Os autores afirmam que o *CoreNLP* é basicamente um coleção de ferramentas de PLN desenvolvidas pelo grupo de PLN de *Stanford CoreNLP*, já o *Jigg* é um conjunto de componentes de PLN desenvolvidos por diversos grupos.

Figura 21 – Arquitetura de *Pipeline* do *Jigg*.



Fonte: NOJI; MIYAO

Para adicionar uma nova ferramenta ao *pipeline* do *Jigg*, o usuário deve escrever um [22] *Wrapper* seguindo a sua *API*. A grande desvantagem do *Jigg* é não fornecer suporte

[21] Atividades dependentes, que não podem ser executas, mesmo que uma tarefa de envio de dado tenha sido marcada como fechada.

[22] São Classes cuja as funcionalidades que expõem estão implementadas em outro lugar. Geralmente utilizadas para integrar funcionalidades de bibliotecas/runtimes/linguagens externas a que está sendo utilizadas pois elas fornecem uma interface nativa a linguagem para aplicações clientes.

para cenários mais complexos, quando se exige trabalhar com processamento distribuído. Ele executa apenas um documento por vez em uma única máquina, o que de fato pode torná-lo lento no processamento de corpus longos ao combinar diversos componentes de PLN.

Uma outra diferença do *Jigg* em relação ao *CoreNLP* é a nomenclatura das análises. Uma pequena melhoria no arquivo XML gerado que, segundos os autores, acreditam ser mais amigáveis para a máquina, pois utilizam nomes padronizados pela [23] *Universal Dependen-cies6*, forte candidata a se tornar padrão em PLN. Porém isso não fica claro e evidente, pois os autores não expõem nenhum exemplo comparativo e explicativo detalhando essa importância de padronização das nomenclaturas no arquivo de anotação.

Em resumo, o trabalho apresenta uma arquitetura para customização de *pipeline* em PLN baseada em *Stanford CoreNLP*. No entanto, algumas coisas ainda não foram cobertas como o caso de estrategia de processamento e outros pontos que seguem na Tabela 5, que resume as característica do sistema, dentro dos aspectos de seleção listados na Seção 3.1.

Tabela 5 – Características do Sistema *Jigg* de NOJI; MIYAO .

Domínio	Independente de Domínio	
Análises Linguísticas	Pré-processamento	*Tokenization*, *SSplit*
	Léxica	POS, NER
	Sintática	*Berkeley Parsing*
	Semântica	-
Ferramentas Externas	Stanford CoreNLP, Berkeley	
Pipeline Customizado	+	
API	+	
API RESTFul	-	
GUI	-	
Estrategia de Processamento	-	
Banco de Dados	-	
Estatísticas do Corpus	-	
Arquitetura/Padrões de Projeto	Base em *Stanford CoreNLP*	
Requisitos do sistema	Scala, Java	
Formato de Saída	XML	
Licença	Apache 2.0	
Documentação	+	

Fonte: do Autor.

3.2.4 *xTAS: Text Analysis in a Timely Manner* (ROOIJ et al., 2012).

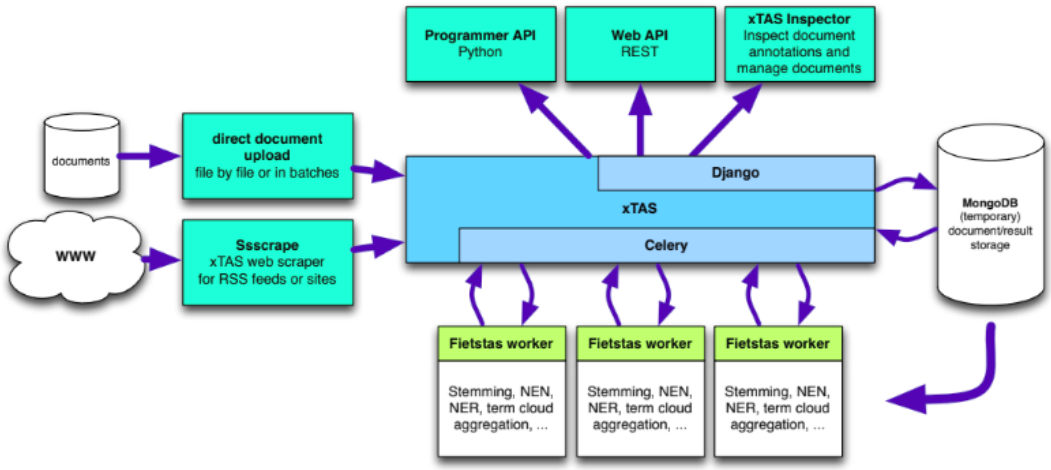
O trabalho de (ROOIJ et al., 2012) apresenta um conjunto de serviços da *web* de código aberto denominado de [24]xTAS, desenvolvido na Universidade de Amsterdã, que permite processar conteúdo textual de documentos e multilíngue.

[23] http://universaldependencies.org/
[24] http://xtas.net/

Segundo os autores, o objetivo do *xTas* é permitir que usuários executem uma variedade de processamento de texto o mais rápido possível, sem precisar se preocupar com banco de dados, armazenamento ou cache de resultados. Ele foi projetado para incorporar algorítimos de análises existentes, de código aberto e/ou proprietário, empregando uma arquitetura distribuída escalonável.

Para utilizar o *xTAS*, o usuário se comunica com a ferramenta utilizando o serviço da *web* que pode ser incluído em sua aplicação como uma biblioteca escrita na linguagem de programação *Python*. Por se tratar de um conjunto de ferramentas, ele inclui algorítimos, técnicas e recursos externos de outras ferramentas existentes, sendo extensível por meio de [25]*plung-in*. Ele utiliza o [26]*MongoDB* para armazenar documentos e resultados, e [27]*Celery* para distribuir as análises aos *node* de processamento, permitindo o processamento de documentos por demanda a fim de minimizar o tempo de espera do usuário. A Figura 22, esboça uma visão geral da arquitetura do *xTAS*.

Figura 22 – Arquitetura e Componentes do *xTAS*.

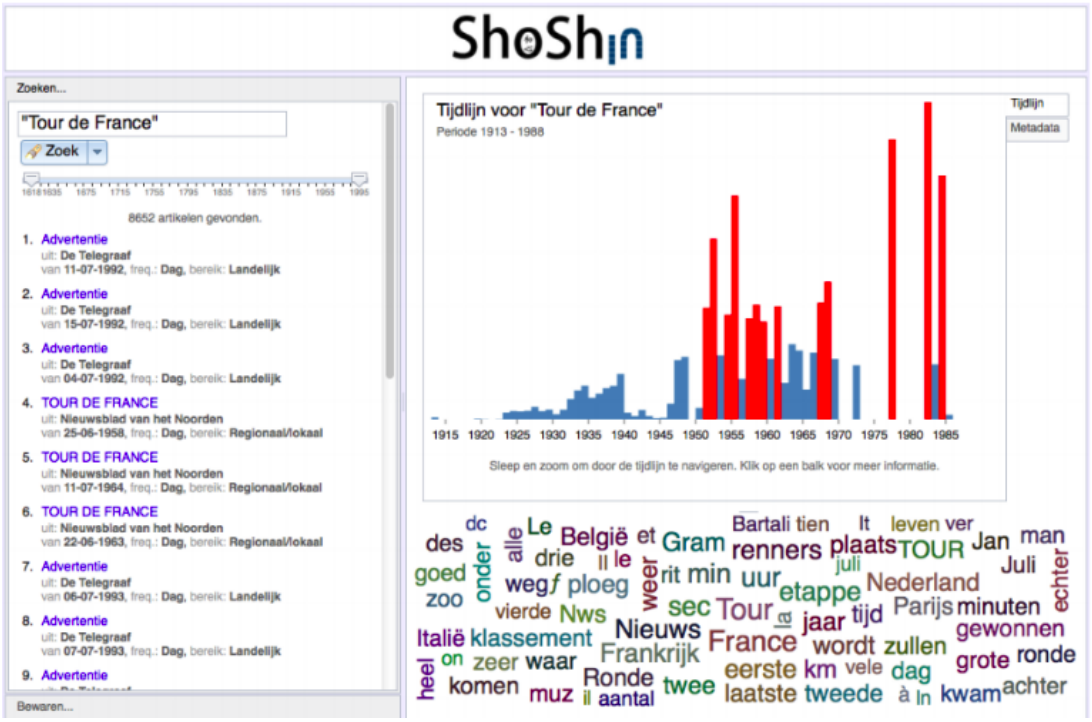


Fonte: ROOIJ et al.

Como a arquitetura ilustra, os documentos de entrada podem ser adicionados utilizando a API do programador, a *API REST* ou adicionados em lotes. O usuário então seleciona os tipos de análises e em seguida o *xTAS* distribui as atividades através do *Celery*, os resultados são retornados logo após o processamento. Apesar de o *xTAS* não possuir *GUI*, a Figura 23 exibe o *Project Infinto* utilizando o *xTAS* para analisar arquivos da biblioteca *Koninklijke*, gerando uma nuvem de palavras frequentes e histograma, com base na sentença "*Tour de France*"analisada.

[25] Em geral é um pequeno programa de computador utilizado para adicionar novas funções a outro programa maior, provendo alguma funcionalidade especial ou muito especifica.
[26] https://www.mongodb.com/
[27] http://www.celeryproject.org/

Figura 23 – *GUI Project* Infinto utilizando o *xTAS*.



Fonte: ROOIJ et al.

Em resumo, o trabalho apresenta uma ferramenta robusta para processamento de corpus, não disponibiliza uma forma de analisar estatisticamente o *corpus*, não possui uma *interface* gráfica para o usuário, não dispõem de análise linguística semântica.

Na Tabela 6, resume as características do sistema, dentro dos aspectos de seleção listados na Seção 3.1.

Tabela 6 – Características do Sistema *xTAS* de ROOIJ et al. .

Domínio	Independente de Domínio	
Análises Linguísticas	Pré-processamento	*Stemming*, *Tokenization*
	Léxica	POS, NER
	Sintática	-
	Semântica	-
Ferramentas Externas	*Stanford CoreNLP*, *SEMAFOR*, *OpenNLP*	
Pipeline Customizado	+	
API	+	
API RESTFul	+	
GUI	-	
Estrategia de Processamento	+	
Base de Dados	+	
Estatísticas do Corpus	-	
Arquitetura/Padrões de Projeto	Cliente/Servidor; *Plugin*.	
Requisitos do sistema	*Linux*, *Python* and *Java*	
Formato de Saída	JSON	
Licença	Apache v2	
Documentação	+	

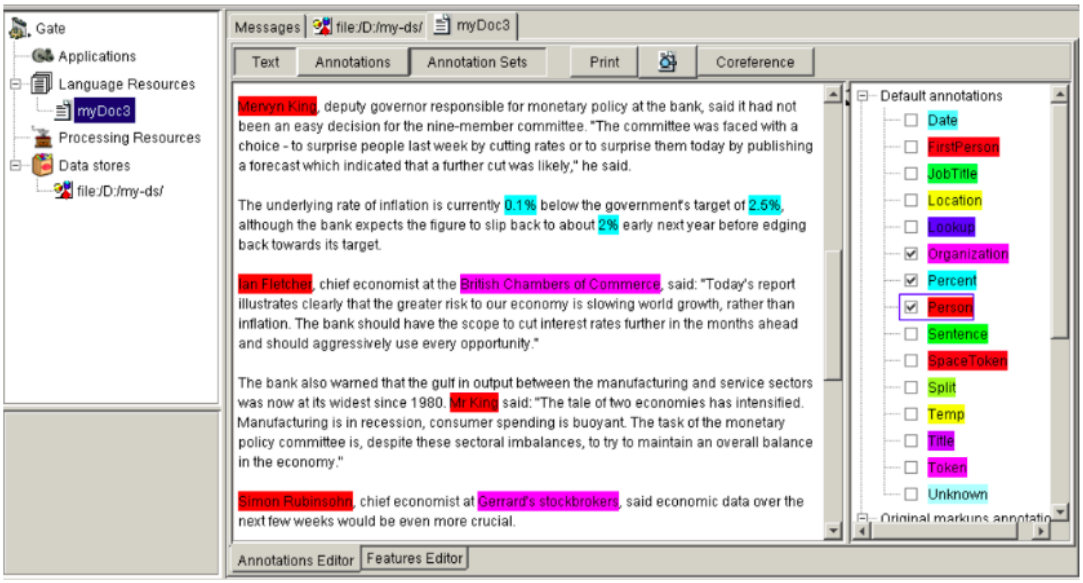
Fonte: do Autor.

3.2.5 *GATE: an Architecture for Development of Robust HLT Applications* (CUNNINGHAM et al., 2002).

O *GATE* é um ambiente de desenvolvimento gráfico e de *framework* que permite aos usuários desenvolverem e implantarem componentes e recursos de engenharia de linguagem de maneira robusta. Foi construído utilizando a linguagem de programação *JAVA*. É considerado também como um ecossistema, pois como arquitetura define a organização de um sistema de engenharia de linguagem e a atribuição de responsabilidades a diferentes componentes e garante que as integrações dos componentes satisfaçam os requisitos do sistema; como *framework* ele fornece um *design* reutilizável para um sistema de *software* e um conjunto de blocos para construção de *software* pré-fabricados que os engenheiros de PLN podem usa, estender e personalizar de acordo com as suas necessidades especificas; como ambiente de desenvolvimento ajuda os usuários a minimizarem o tempo gasto na criação de novos sistemas ou modificando os existentes, auxiliando e fornecendo um mecanismo de depuração para novos módulos. A Figura 24 mostra a *interface* gráfica do *GATE* para anotação de *corpus*, na janela esquerda encontra-se uma arvore para navegação nos diretórios e arquivos da aplicação; na janela centra, tem-se a área de edição, nela o usuário realiza a anotação dos dados textuais e por fim na janela da direita tem-se os tipos de entidades nomeadas que são identificadas após a marcação do usuário. A Figura 25 mostra todas as características da arquitetura das *APIs* do *GATE*.

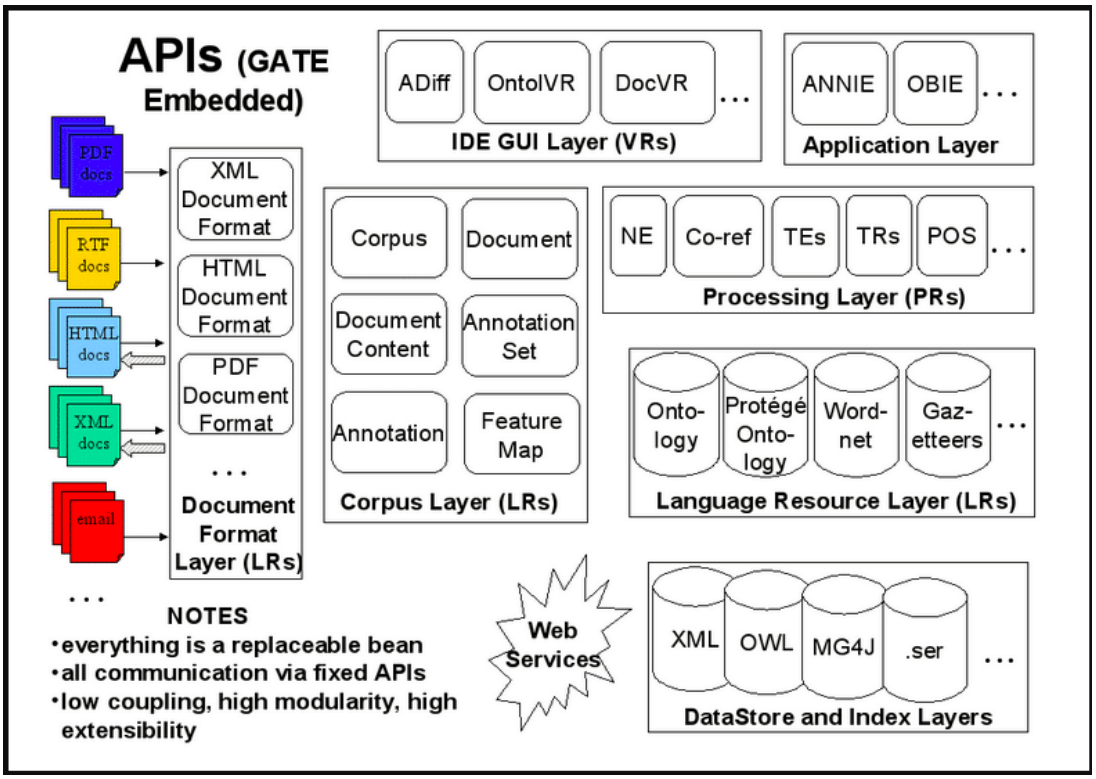
Em resumo, o trabalho apresenta uma ferramenta robusta para processamento para PLN, que realiza diversas tarefas de PLN como extração de informações, anotação de

Figura 24 – *GATE GUI* para Anotação de *Corpus*.



Fonte: CUNNINGHAM et al.

Figura 25 – *GATE APIs* Arquitetura.



Fonte: GATE

corpus, analise de sentimentos, mineração de opiniões, realizar avaliações sobre os aplicativos construídos, entre outras tarefas. O *framework* pode ser usado para desenvolver aplicativos e recursos em vários idiomas, como base no suporte *Unicode*. Entretanto, o

GATE não é para qualquer usuário, sua [28]documentação é vasta porem muito complexa para inciantes em PLN devido a sua vasta abrangência e funcionalidades.

Na Tabela 7, resume as características do sistema, dentro dos aspectos de seleção listados na Seção 3.1.

Tabela 7 – Características do Sistema GATE de CUNNINGHAM et al. .

Domínio	Independente de Domínio	
Análises Linguísticas	Pré-processamento	*Tokenizer*, *Splitter*, *Lang. Identifier*
	Léxica	*POS Tagger*, NER
	Sintática	*Chunker*, *Parser*
	Semântica	*WSD*, *Tagger*, *Gazetteer*
Ferramentas Externas	*Stanford CoreNLP*, *LingPipe*, *OpenNLP*	
***Pipeline* Customizado**	+	
API	+	
API RESTFul	+	
GUI	+	
Estrategia de Processamento	+	
Base de Dados	+	
Estatísticas do Corpus	-	
Arquitetura/Padrões de Projeto	Baseada em Componentes	
Requisitos do sistema	*Java*	
Formato de Saída	TEXT, JSON, XML	
Licença	GNU	
Documentação	+	

Fonte: do Autor.

3.2.6 *FreeLing 3.0: Towards Wider Multilinguality* (PADRÓ; STANILOVSKY, 2012).

FreeLing é uma conjunto de ferramentas de análise de linguagem de código aberto, que fornece uma ampla gama de analisadores para vários idiomas. Esse projeto foi criado e liderado por Lluís Padró através do grupo de pesquisa de processamento de linguagem natural na *Universitat Politécnica de Catalunya*. *FreeLing* foi construído na linguagem C++ e possui *API's* para *Python* e Java, para oferecer recursos de processamento de texto e anotações de idioma para desenvolvedores de aplicativos NLP. Os recurso disponíveis no *FreeLing* são mostrados na Figura 26 juntamente com sua disponibilidade para diferentes idiomas.

O *FreeLing* é personalizável, extensível e tem uma forte orientação para aplicativos do mundo real em termos de velocidade e robustez. A documentação completa do *FreeLing* pode ser visitada no [29]site oficial do projeto; acesso a fórum e ferramenta [30]demo (CARRERAS et al., 2004).

[28] https://gate.ac.uk/
[29] http://nlp.lsi.upc.edu/freeling/
[30] http://nlp.lsi.upc.edu/freeling/demo/demo.php

Figura 26 – *FreeLing*: Análises Linguísticas e Suporte a Idiomas.

	as	ca	cy	en	es	gl	it	pt	ru
Tokenization	X	X	X	X	X	X	X	X	X
Sentence splitting	X	X	X	X	X	X	X	X	X
Number detection		X		X	X	X	X	X	X
Date detection		X		X	X	X		X	X
Morphological dictionary	X	X	X	X	X	X	X	X	X
Affix rules	X	X	X	X	X	X	X	X	
Multiword detection	X	X	X	X	X	X	X	X	
Basic named entity detection	X	X	X	X	X	X	X	X	X
B-I-O named entity detection				X	X	X		X	
Named Entity Classification				X	X	X		X	
Quantity detection		X		X	X	X		X	X
PoS tagging	X	X	X	X	X	X	X	X	X
Phonetic encoding				X	X				
WN sense annotation		X		X	X				
UKB sense disambiguation		X		X	X				
Shallow parsing	X	X		X	X	X		X	
Full/dependency parsing	X	X		X	X	X			
Co-reference resolution					X				

Fonte: (PADRÓ; STANILOVSKY, 2012)

Em resumo, o trabalho apresenta uma ferramenta robusta para processamento para PLN, que contem várias análises próprias para PLN. Entretanto, ela não possui suporte a customização de *pipeline* no que se refere-se a utilização de análises linguísticas de ferramentas de PLN de terceiros junto com suas análises linguísticas próprias. Na Tabela 8, resume as características do sistema, dentro dos aspectos de seleção listados na Seção 3.1.

Tabela 8 – Características do Sistema *FreeLing* de PADRÓ; STANILOVSKY.

Domínio	Independente de Domínio	
Análises Linguísticas	Pré-processamento	*Tokenization*, *Sentence Splitting*, *Language Identifier*
	Léxica	*Number detection*, *Data detection*, *NER*, *POS tagging*
	Sintática	*BIO*, *Shallow Parsing*, *Full/dependency parsing*
	Semântica	*WN sense disambiguation*, *UKB sense disambiguation*
Ferramentas Externas	-	
***Pipeline* Customizado**	-	
API	+	
API RESTFul	-	
GUI	-	
Estrategia de Processamento	+	
Base de Dados	+	
Estatísticas do Corpus	-	
Arquitetura/Padrões de Projeto		
Requisitos do sistema	*C/C++*	
Formato de Saída	CoNLL, JSON, XML	
Licença	GNU	
Documentação	+	

Fonte: do Autor.

3.3 ANÁLISE QUALITATIVA

O objetivo desta seção é fornecer uma análise comparativa qualitativa dos trabalhos relacionados visando identificar diversos critérios que servem para tal tipo de avaliação. Dessa forma, os trabalhos foram comparados de acordo com os principais aspectos pontuados na Seção 3.1 que são detalhados em seguida.

Para a característica de **Domínio**, apenas o trabalho de (SOYSAL et al., 2017) é dependente de domínio, o que limita o poder de abrangência da ferramenta quando aplicada em outros domínios, sendo necessário a realização de inúmeras alterações para utilizá-la.

No tocante as **Análises Linguísticas** poucos trabalhos cobrem todos os níveis citados: análise léxica, sintática e semântica. Isso significa que a ferramenta não dispõem de todas as análises linguísticas, consequentemente, devem ser adicionadas pelo usuário quando necessário, se a ferramenta possibilitar essas inclusões.

Para a característica ***Pipeline* Customizado** apenas os trabalhos de (KARIMZADEH et al., 2019) e (PADRÓ; STANILOVSKY, 2012) não possui evidências de que arquitetura suporta a adição de novos componentes ao pipeline de análise, impactando na extensão das análises da ferramenta. Para a característica de ***API*** os trabalhos de (KARIMZADEH et al., 2019) e (SOYSAL et al., 2017) não possuem evidencia de tal funcionalidade.

Para ***API RESTFul*** os trabalhos de (SOYSAL et al., 2017), (NOJI; MIYAO, 2016) e (PADRÓ; STANILOVSKY, 2012) não dispõem desse recurso, o que dificulta o desenvolvimento de aplicações web clientes com tais ferramentas.

Para a característica **GUI**, os trabalhos de (NOJI; MIYAO, 2016), (ROOIJ et al., 2012) e (PADRÓ; STANILOVSKY, 2012) não possuem *interface* gráfica. Essa característica é importante para que usuários com conhecimento mínimo de PLN possam analisar corpus sem precisar programar. Uma *GUI* também pode ser útil para ensino teórico e prático para compreensão de anotação de corpus e customização de *pipeline*.

Para a característica de **Estrategias de Processamento** não fica evidente que os trabalhos de (KARIMZADEH et al., 2019) e (NOJI; MIYAO, 2016) a estratégias para computação paralela ou qualquer outro processo de otimização a fim de diminuir o tempo do processamento, que é muito importante para processamento de grandes quantidades de dados.

Para a característica de utilização de **Banco de Dados**, os trabalhos de (SOYSAL et al., 2017), (NOJI; MIYAO, 2016) e (PADRÓ; STANILOVSKY, 2012) não possuem estratégias para armazenamento secundário, a saber que, utilizar tal formato de armazenamento aumenta o desempenho e proporciona o desenvolvimento centrado na lógica da aplicação. Já para a característica de **Estatística de Corpus**, nenhum trabalho realiza essa tarefa; a se tratar de análise e anotações de *corpus* linguístico essa atividade é de razoável importância, pois permite que o usuário possa compreender melhor quais tipos de dados eles está lhe dando, bem como ser útil em um cenário de ensino de língua mais profícuo (SILVA, 2011).

Para o **Formato de Saída**, nenhuma ferramenta apresenta múltiplos formatos de

saídas de dados, fornecendo ao usuário apenas uma alternativa. Dessa forma, quando necessário outro formato, o desenvolvedor tem que criar um parser para converter no formato desejado, o que não é desejável, uma vez é interessante que a ferramenta proporcione ao usuários alternativas, já que se trata de customização. Sobre os características dos sistemas, todos os trabalhos dispõem de algum artefato factível. Na Tabela 9 resume as principais características de cada trabalho.

Tabela 9 – Principais Característica do Trabalhos.

TRABALHOS		**GeoTxt**	**CLAMP**	**Jigg**	**xTAS**	**GATE**	**FreeLing**
DOMÍNIO		INDEP.	DEP.	INDEP.	INDEP.	INDEP.	INDEP.
ANÁLISES	**Pré-proces.**	Limpeza de texto	-	*tokenization*, *SSPlit*	*tokenization*, *stemming*	*Tokenizer*, *Sentence Splitter*, *Language Identier*	*Tokenization*, *Sentence Splitter*, *Language Identier*,
	Léxica	NER	NER, POS	NER, POS	POS, NER	*POS Tagger*, *NER*	*Number detection*, *Data detection*, *NER*, *POS tagging*
	Sintática	-	*Chunker*, *Parsing*	Parsing	-	*Chunker*, *Parser*	*BIO*, *Shallow Parsing* *Full/dependency parsing*
	Semântica	-	-	-	-	*WSD*, *Tagger*, *Gazetter*	*WN WSD*, *UKB WSD*
FERRAMENTAS		*CogComp*, *CoreNLP*, *GATES*, *ANNIE*, *MIT IE*, *OpenNLP*, *LingPipe*.	*OpenNLP*	*CoreNLP*, *Berkeley*	*CoreNLP*, *SEMAFOR*, *OpenNLP*	*CoreNLP*, *LingPipe*, *OpenNLP*	-
PIPELINE CUSTOMIZADO		-	+	+	+	+	-
API		-	-	+	+	+	+
API RESTFul		+	-	-	+	+	-
GUI		+	+	-	-	+	-
ESTRATEGIA DE PROCESSAMENTO		-	+	-	+	+	+
BANCO DE DADOS		+	-	-	+	+	-
ESTATÍSTICAS		-	-	-	-	-	-
ARQUITETURA		Cliente/Servidor	*Framework* Eclipse	Base em *CoreNLP*	Cliente/Servidor, *Plugin*	Baseado em Componentes	Baseado em Cliente/Servidor
REQUISITOS		Java	Java	Java	Python, Java	Java	C/C++
FORMATO DE DADOS		JSON	Texto	XML	JSON	TEXT, JSON, XML	CoNLL, TEXT, JSON, XML
LICENÇA		GNU	Academic Free	Apache 2.0	Apache 2.0	Apache 2.0	GNU
DOCUMENTAÇÃO		+	+	+	+	+	+

Fonte: do Autor.

3.4 CONTRIBUIÇÃO DA PROPOSTA

Diante da análise apresentada, o trabalho proposto tem como objetivo atender a todas as características apresentadas na Seção 3.1, de modo a melhorar alguns aspectos específicos. Neste sentido, esta dissertação propõe um trabalho que seja:

- **(I)** independente de domínio, amplificando, que possa ser aplicado no domínio biomédico, redes sociais, jornalismo, policial, entre outros, pois os componentes utilizados podem ser aplicados em textos de diversificados;
- **(II)** que abranja as análises linguísticas nos níveis léxico, sintático e semântico, como também pré-processamento de texto, visto que nenhum dos trabalhos chega a explorar o nível semântico e não abrange uma quantidade significativa de análises. No que se refere as ferramentas externas, apenas o *GeoTxt* possui uma quantidade maior, sete no total, as demais dispõem uma ou três ferramentas externas; nesse contexto, pretende-se;
- **(III)** incluir as principais ferramentas utilizadas pelo grupo de pesquisa envolvido, ferramentas com maior numero de citações em trabalhos científicos e comunidades de mineração de texto, visando prover para o usuário um kit ferramental robusto; no que se refere a customização de *pipeline*, apenas o trabalho *GeoTxt* não permite essa tarefa, entretanto as demais, não dispõem detalhes de como as mesmas fazem para adicionar novas ferramentas de PLN ao seu *pipeline* de análise, o que é muito importante estabelecer uma arquitetura que permita que os usuários possam;
- **(IV)** integrar novas análises linguísticas sem a necessidade de fazer muitas adaptações para existir uma interoperabilidade entre ferramentas construídas em linguagens de programação diferentes e que não haja divergência no formato de saída dos dados anotados, quanto a problemas de *tokenization* com id divergentes.

Diante disso, a proposta apresentada neste trabalho é de uma arquitetura baseada em *Plugins* que proporcione ampliar com facilidade novas análises no *pipeline* customizado. Além disso, nenhum dos trabalhos relacionados apresenta uma proposta para análise estatística do *corpus*, visto que este recurso ajuda o usuário a compreender melhor a natureza dos dados. Uma outra enfase é que nenhum dos trabalhos anteriores fornece múltiplos formatos na saída dados de forma unificada em uma estrutura organizada por camada de profundidade de análise, o que é importante para que outros sistemas possam, com facilidade, consumir os resultados anotados.

No próximo capítulo apresenta-se a definição dessa proposta, bem como a sua especificação arquitetural e funcional do protótipo projetado, implementado e avaliado.

4 PROPOSTA

Neste capítulo apresenta-se a proposta desse trabalho, o *DeepNLP Framework*. Primeiramente é descrito seu propósito, bem como sua possível aplicação. Em seguida é apresentado uma visão arquitetural. Na sequência, faz-se um detalhamento dos componentes e suas *interfaces* abordando questões relacionadas à especificação e à implementação, apresentando vários diagramas de Linguagem de Modelagem Unificada (UML), visando detalhar a arquitetura do *software* construída para suportar as características levantadas no Capítulo 3. Logo depois, apresenta-se a visão funcional que descreve as funcionalidade e as atividades realizada por cada componente, e por fim, algumas considerações finais.

4.1 O *DEEPNLPF*

O *DeepNLPF*, é um *framework* para integração de análises linguísticas, customização de *pipeline*, anotação e visualização de dados textuais. Esse *framework* permite que o [1]usuário possa integrar ferramentas de PLN de terceiros por meio de *plugins*, possibilitando a seleção de diversas análises linguísticas em um *pipeline* customizado para anotação de dados textuais e, permite que os mesmos possam ser salvos em um Sistema de Gerenciamento de Banco de Dados (SGBD) para consulta posterior. Possui uma *Interface* de Programação para Aplicação *(API)* que proporciona a construção de aplicações de PLN. Igualmente, através da *API RESTFul*, seus serviços podem ser consumidos por aplicações de PLN que dispõem de serviços na nuvem. Tem suporte a *multithreads* e paralelismo, visando aproveitar ao máximo os recursos computacionais disponíveis. Uma característica extra que o *DeepNLPF* implementa é, um *DashBoard* integrado, que dispõem todos os seus recurso por meio de uma *interface* gráfica baseada em navegadores, e por fim, um sistema de notificação *desktop* e *mobile* ideal para informar ao usuário [2]*logs* durante todo o processamento.

O *DeepNLPF* pode ser empregado em diversos domínios que adotam PLN, como por exemplo extração de relações, aplicações de aprendizagem de máquina, classificação de texto, análise de sentimentos, etc. A principal motivação dessa proposta parte da complexidade e do trabalho exaustivo que se tem para integrar ferramentas de PLN de terceiros em um *pipeline* de análise. Como também, o tempo de processamento que as ferramentas levam para processar os dados, quando agrupadas diferentes análises sem empregar as devidas estratégia de desempenho.

[1] Nesse contexto, os usuários do *DeepNLPF*, são desenvolvedores de sistemas que utilizam recursos de PLN.

[2] Em computação, *log* de dados é uma expressão utilizada para descrever o processo de registro de eventos relevantes num sistema computacional. Esse registro pode ser utilizado para restabelecer o estado original de um sistema ou para que um administrador conheça o seu comportamento no passado.

O *DeepNLPF* não objetiva substituir nenhum dos trabalhos relacionados a essa proposta. Até porque os trabalhos citados funcionam muito bem para o seus propósitos. Entretanto, foram encontradas algumas lacunas no que refere-se a integração de ferramentas de PLN e customização de *pipeline*, suporte para otimização de processamento, documentação insuficiente e/ou complexa para integração de novas análises ao *pipeline*, e a ausência de suporte a estatística e visualização do dados. São justamente nesses fatores que o *DeepNLPF* visa acrescentar.

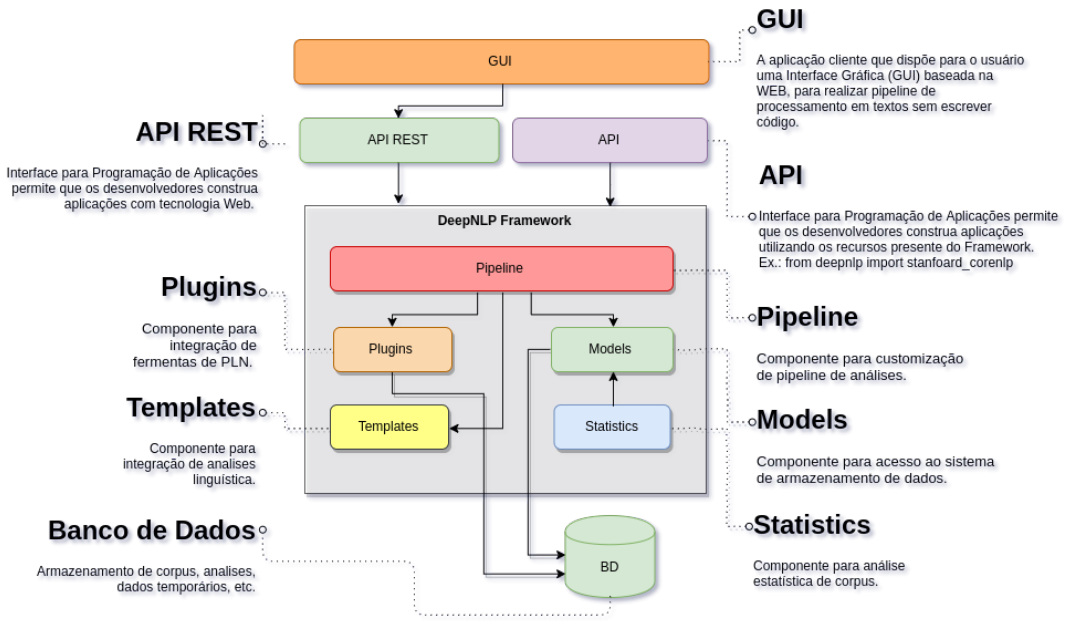
4.2 ARQUITETURA DO *DEEPNLPF*

O *DeepNLPF* é classificado como um *framework* caixa cinza, ou seja, o usuário não precisa conhecer a fundo sua implementação para utilizá-lo. A Figura 27, exibe de forma geral os principais componentes do *framework* e seus relacionamentos. O componente **GUI**, é responsável pela a aplicação cliente que dispõe para o usuário um *interface* gráfica baseada na web para acessar os recursos do *framework* sem a necessidade de escrever códigos. O componente **API REST** fornece uma *interface* para que o usuário possa construir aplicações consumindo recurso na web. O componente **API** é uma outa *interface* que permite que o usuário possa desenvolver seus programas usando os recursos do *framework*. O componente ***Pipeline*** é responsável pela [3]orquestração de todas as ações e integrações de processamento dos demais componentes; em seguida, tem-se o componente ***Statistics*** que é encarregado pela análise estatística do *corpus* e suas respectivas visualizações de dados; o componente ***Templates*** é responsável pelo gerenciamento dos [4]*Schemas* que definem os formatos dos arquivos de saída das anotações do *corpus* e integração das análises; o componente ***Plugins***, esse componente delega a responsabilidade de integrar todos os *Wrappers* de acesso às ferramentas de Processamento de Linguagem Natural (PLN) externas ao *pipeline* customizado; por fim, tem-se o componente ***Models*** que fica responsável pelo gerenciamento do sistema de conexão e requisições com o banco de dados (acesso aos *corpus*, *logs* do sistema, anotações, estatística, entre outros dados armazenados). Na próxima Seção 4.3, será explanado detalhadamente a especificação e implementação de cada componente citado.

[3] Orquestração é um termo utilizado na tecnologia da informação que significa coordenação, organização de tarefas.

[4] *Schemas* descreve formalmente os elementos em um documento, geralmente utilizado para verificar cada parte do conteúdo do item em um documento, visando identificar se ele está de acordo com as descrição do elemento que é colocado.

Figura 27 – Diagrama de Arquitetura e Componentes do *DeepNLPF*.



Fonte: do autor.

4.3 COMPONENTES DO *DEEPNLPF*

Nessa seção, será abordado cada componente detalhadamente, conforme as visões (Visão Lógica, Visão de Implementação, Visão de Processo, Visão de Implantação e Caso de Uso) do RUP (4 + 1) segundo (KRUCHTEN, 1995) que descrevem as visões arquiteturais de um sistema.

4.3.1 Componente *Models*

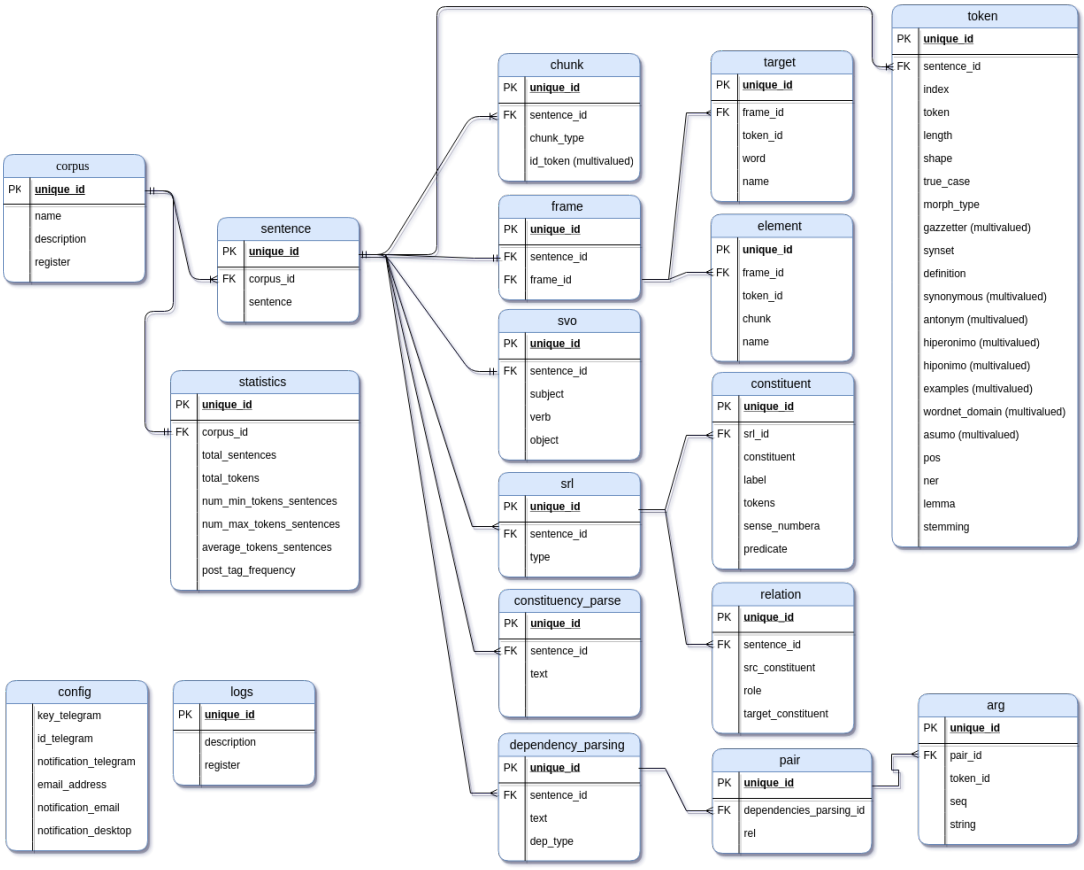
O componente *Models* tem o objetivo de permitir que o usuário possa utilizar um banco de dados para armazenamento do *corpus*, anotações linguísticas, estatísticas dos *corpus* e *logs* do sistema. A principal importância para se ter um componente como esse, é a sua capacidade de aumentar a velocidade de acesso aos dados do *Corpus* a ser processado, diferente dos arquivos de texto onde latência de entrada e saída de dados é maior, implicando no desempenho. Como também, permitir que o usuário se concentre na aplicação que está sendo construída e não perca tempo criando estratégias para o armazenamento de dados. (ROOIJ et al., 2012).

4.3.1.1 Visão Lógica

Esse componente deve atender aos critérios exibidos na Figura 28 que mostra o diagrama do Modelo Relacional (MR) da base de dados do *DeepNLPF*. Entretanto, embora o componente *Models* utilize um sistema de banco de dados não relacional por padrão, não se

limita apenas a essa abordagem, podendo ser abrangido para sistemas de banco de dados relacional como segue a figura.

Figura 28 – Modelo Relacional do Banco de Dados do *DeepNLPF*.



Fonte: do autor.

O diagrama da Figura 28, é composto por 18 tabelas. Onde tem-se as tabelas isoladas ***logs*** e ***config***, nessas tabelas são armazenadas respectivamente dados sobre os *logs* do sistema e as configurações básicas como *keys* para ativar as notificações.

Na tabela ***corpus*** são armazenadas as informações de (id, nome, descrição e data e hora) do *corpus* inserido. Essa tabela possui relacionamento com a tabela ***statistics***, que armazena informações quantitativas sobre o *corpus* (quantidade total de sentenças, quantidade total de *tokens*, quantidade mínima e máxima de *tokens* por sentenças, quantidade média de *tokens* por sentenças, frequência de *pos tags* e outras informações), já na tabela ***sentence*** são armazenadas todas as sentenças de um determinado *corpus*. Essas sentenças são formadas por N *tokens*. Sendo assim, a tabela ***token*** armazena todos os *tokens* de cada sentença do *corpus*.

A tabela ***token***, ela armazena todos os *tokens* das sentenças e outros atributos provenientes de algumas análises linguísticas, são (index do *token*, *length*, *shape*, *true_case*, *morph_type*, *gazzetter*, *synset*, *definition*, *synonymous*, *antonym*, *hiperonimo*, *hiponimo*, *examples*, *wordnet_domain, pos, lemma, stemming, ner* e *asumo*).

A tabela ***sentence*** contém relação com ***chunkings***, uma sentença pode contém N *chunk*, cada *chunk* é formado por 2 ou N *tokens* e sua dependência, no caso (*dep_type*, *id_token_1* e *id_token_2*).

A tabela ***sentence*** contém relação com a tabela ***frames***. Essa tabela *frames* armazena informações de *frames semantics parsing*. A tabela ***frames*** contém relacionamento com outras duas tabelas ***target*** e ***element*** que armazenam informações sobre esses *frames*.

A tabela ***sentence*** contém relação com a tabela ***svo***, nessa tabela contém os atributos (*subject*, *verb* e *object*) que armazenam informações da análise semântica *SVO* da sentença.

A tabela ***sentence*** contém relação com a tabela ***srl*** que armazena a análise *Semantic Role Label* que pode ser do tipo verbal, preposicional ou nominal. Nessa análise contém informações sobre as *constituent* e *relation* sobre os *tokens* analisados, essas informações são guardadas nas tabelas ***constituent*** e ***relation***.

Na tabela ***constituency_parse*** são armazenadas informações da análise *Parse Constituency*. Na tabela ***dependency_parsing*** são armazenadas as informações da análise *Dependency Parsing*. Essa tabela tem relação com outras duas tabelas *pair* e *arg* que armazenam as informações dos *tokens*.

Esse diagrama deve ser expandido, conforme o número de análises linguísticas forem incorporadas ao *framework*. Entretanto, esse diagrama discutido, visa apresentar a ideia geral de como funciona o armazenamento das análises linguísticas utilizando um modelo ER. Porém, o diagrama que representa como são armazenados os dados no banco não relacional utilizado pelo *DeepNLPF* é a Figura 29 a seguir.

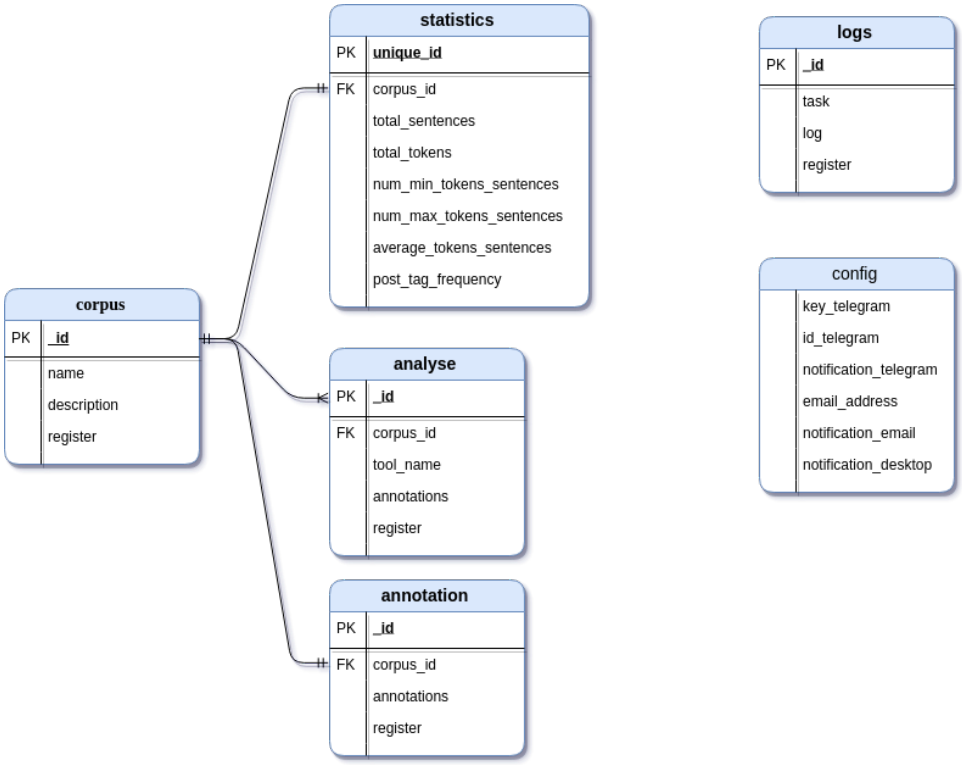
No banco de dados não relacional, as tabelas ilustradas na Figura 29 são tratadas como coleções de documentos. Nesse caso, tem-se as coleções (*corpus*, *statistics*, *analyze*, *annotation*, *logs* e *config*). Cada coleção pode conter N documentos, esses documentos são representados no formato *JSON*. Então, na coleção ***corpus*** são armazenadas as sentenças. Na coleção ***statistics*** são salvos os documentos que contém as estatísticas dos *corpus*. Na coleção ***analyze***, são armazenados os documentos contendo todas as análises processadas pelas ferramentas de PLN. Na coleção ***annotation*** são armazenados os documentos que agrupam as anotações em um só documento. Na coleção ***logs*** e ***config*** são armazenados os documentos de *logs* do sistema e suas configurações. Sendo assim, todas essas coleções funcionam da mesma forma que as tabelas relacionais discutidas anteriormente. A Figura 30 exemplifica como ficam representadas essas informações nesse tipo de armazenamento de dados.

Por padrão, o *DeepNLPF*, tem suporte ao Banco de Dados Não Relacional (BDNR) [5]*MongoDB*. Ele foi escolhido porque a ferramenta trabalha diretamente com documentos e entende-se que seria o mais apropriado para essa finalidade. Porém, isso não impede que o usuário utilize outras tecnologias de banco de dados, seja ele relacional ([6]MySQL,

[5] https://www.mongodb.com/

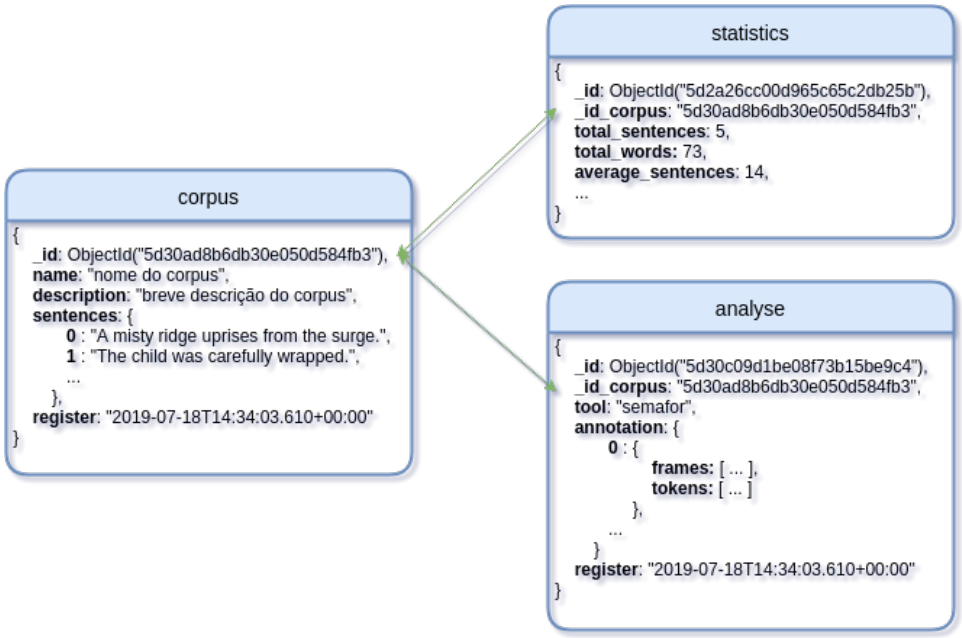
[6] O MySQL é um sistema de gerenciamento de banco de dados, que utiliza a linguagem SQL como interface. É atualmente um dos sistemas de gerenciamento de bancos de dados mais populares da

Figura 29 – Diagrama do Banco de Dados Não Relaciona do *DeepNLPF*.



Fonte: do autor.

Figura 30 – Exemplo de armazenamento de documentos em sistema de armazenamento de dados não relacional.



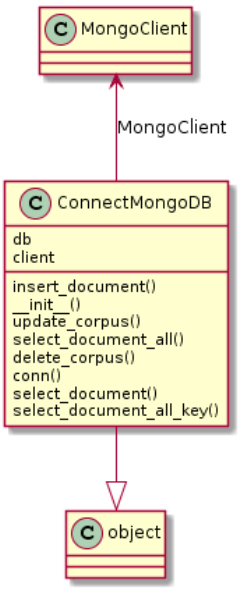
Fonte: do autor.

Oracle Corporation, com mais de 10 milhões de instalações pelo mundo.

[7]MariaDB [8]SQLite), em memória ([9]VoltDB), ou até mesmo outros tipos de armazenamento de dados, como arquivos de textos.

A Figura 31 exibe o diagrama de classe desse componente, onde a classe *ConnectMongoDB* apresenta suas funções, para isso, requer a conexão com a classe *MongoClient* para fornecer suas funcionalidades, para selecionar, adicionar, excluir e editar dados.

Figura 31 – *MongoDB* - Diagrama de Classe



Fonte: do autor.

4.3.1.2 Visão de Implementação

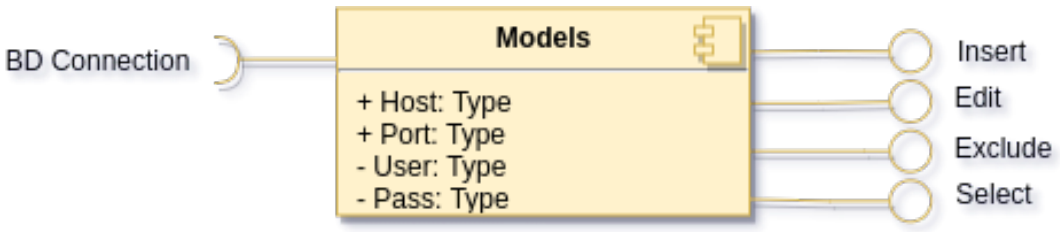
Para o usuário adicionar uma nova tecnologia para armazenamento de dados no *framework*, ele deve criar uma nova classe contendo o conector da nova base de dados, e as funções com as operações que ele desejar realizar, exemplo CRUD. Entretanto, é necessário que o usuário modele a base de dados para ser capaz de salvar os dados conforme sua necessidade, caso a que está sendo disponibilizada não supra sua necessidade. A Figura 32 ilustra a *interface* desse componente, que para funcionar requer uma conexão com um banco de dados e visa fornecer funcionalidades para inserir, editar, excluir e selecionar dados.

[7] MariaDB é um SGDB que surgiu como fork do MySQL, criado pelo próprio fundador do projeto após sua aquisição pela Oracle.

[8] SQLite é uma biblioteca em linguagem C que implementa um banco de dados SQL embutido. Programas que usam a biblioteca SQLite podem ter acesso a banco de dados SQL sem executar um processo SGBD separado.

[9] O VoltDB é um banco de dados em memória projetado por Michael Stonebraker, Sam Madden e Daniel Abadi. É um RDBMS compatível com ACID que usa uma arquitetura de nada compartilhado.

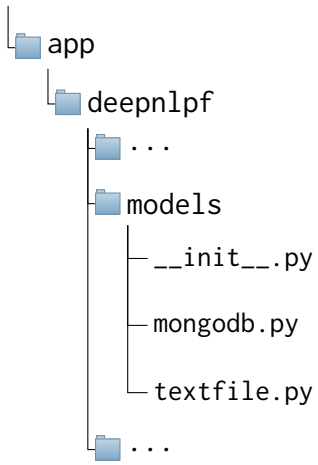
Figura 32 – *Models* - Diagrama de Componente



Fonte: do autor.

O componente discutido pode ser localizado dentro do diretório *deepnlpf* » *models*, nesse diretório contém outros arquivos como *"init.py"*, *"mongodb.py"* e *textfile.py*, conforme ilustra a estrutura em árvore ilustrada na Figura 33.

Figura 33 – Estrutura de Diretório e Arquivos do Componente *Models*.



Fonte: do autor.

O arquivo *"__init__.py"* é um arquivo padrão do *Python* para tratar um diretório como módulo, já o arquivo *mongodb.py* trata-se da classe do *DeepNLPF* que contem todas as operações de acesso ao sistema de armazenamento de dados. Essa classe contem as funções: salvar, editar, visualiza e excluir. Tais funções, são utilizadas por meio de importação, como exemplifica o Bloco de Código 3.

Na linha 1, do Bloco de Código 3, o banco de dados *MongoDB* é incluído na aplicação; na linha 2, é instanciado um objeto da classe *DataBase* que permite o acesso as funções da classe *DataBase*. É também possível implementar as próprias funções na classe. O usuário também, quando preferir, pode criar uma nova classe com suas próprias funções.

Algoritmo 3: Incluindo o banco de dados na aplicação.

```
1   from deepnlpf.models.mongodb import DataBase
    db = DataBase()
```

4.3.1.3 Salvar, visualizar e excluir documentos no banco de dados.

Para o usuário adicionar um novo documento "*corpus*"na base de dados, é preciso definir uma estrutura de documento no formato *JSON*, contendo os campos: *"name"*. Esse campo recebe o nome do corpus (campo obrigatório); o campo *"description"*, recebe uma descrição breve do corpus (campo opcional); o campo *"sentences"* recebe as sentenças do corpus e, por fim, o campo *"register"* recebe a data e hora em que esse novo documento foi salvo. Essa estrutura é ilustrada no Bloco de Código 4. Uma consideração a ser feita é que o *corpus* a ser inserido no sistema de armazenamento deve estar em um arquivo de texto, estruturado com uma sentença por linha, de preferência já pré-processado, removido caracteres especiais, possíveis ruídos do texto quando assim necessário por exemplos [10]*StopWords* e Símbolos.

Algoritmo 4: Estrutura básica do documento corpus.

```
    document = {
2       "name": "childes",
        "description": "Esse corpus contem falas de criancas.",
4       "sentences": TextFile().open_txt("/home/user/childes.txt"),
        "register": datetime.datetime.now()
6   }
```

O *DeepNLPF* inclui algumas funções que facilitam na leitura de arquivos de textos, *TXT*, *XML*, *JSON*, entre outros formatos, pois ele utiliza a biblioteca [11]Pandas que traz facilidades para trabalhar com esses tipos de armazenamento de dados. Na linha 4, do Bloco de Código 4, a chave *"sentences"* recebe um objeto da *class TextFile*, que acessa a método *open_txt()*, passando por parâmetro o caminho onde está localizado o *corpus* que será salvo no banco de dados. Na linha 5, a chave *"register"* recebe informações de data e hora através do método nativa do *Python datetime*. Sendo assim, para que o *corpus* possa ser salvo, é preciso chamar o objeto de acesso ao banco de dados (*db*), acessando o método *save_corpus()* passando por parâmetro o nome do documento a ser salvo, no caso, *document*. Se o documento for salvo com sucesso, será retornado o ID do documento. Caso contrário, será exibida uma mensagem de erro, como ilustrado no Bloco de Código 5. Em seguida o Bloco de Código 6 demonstra como selecionar e excluir um documento no sistema de armazenamento.

[10] *Stopwords* são palavras como: As, e, os, de, para, com, sem, foi... que são palavras consideradas irrelevantes para o conjunto de resultados a ser exibido nos sites de pesquisa, PLN, entre outros.

[11] Pandas é um biblioteca que fornece estruturas de dados de alto desempenho e fáceis de usar e ferramentas de análise de dados para a linguagem de programação *Python*. https://pandas.pydata.org/

Algoritmo 5: Salvando um documento no banco de dados.

```
import datetime
from deepnlpf.models.textfile import TextFile

document = {
    "name": "childes",
    "description": "Esse corpus contem falas de criancas.",
    "sentences": TextFile().open_txt("/home/user/childes.txt"),
    "register": datetime.datetime.now()
}

request = db.save_corpus(document)
print(request)

# True
# {'corpus': {'_id': ObjectId('5d0e7714476f195c7e0eab82')}}

# False
# {'corpus': {'_id': None}}
```

Algoritmo 6: Selecionando e excluindo um documento no banco de dados.

```
# seleciona um documento corpus pelo ID.
db.select_corpus("5d0e7714476f195c7e0eab82")

# exclui um documento corpus pelo ID.
db.delete_corpus("5d0e7714476f195c7e0eab82")
```

4.3.2 Componente *Plugins*

Em tarefas que necessitam de diversos níveis de análises linguísticas para se alcançar resultados proveitosos, como povoamento de ontologias (MAYNARD; LI; PETERS, 2008), extrações de ralações, análise de sentimento, entre outras aplicações, por exemplo, é preciso combinar várias análises linguísticas provenientes de ferramentas de PLN de terceiros. Essas ferramentas são construídas em linguagens de programação variadas, com configuração e uso diferentes. Isso torna o desenvolvimento de aplicações de PLN dispendioso de tempo, exigindo um esforço significativo para conseguir integrar e combinar as ferramentas ao *pipeline* de análise.

Uma estratégia utilizada para sanar esse tipo de problema de integração é a utilização de *Plugins*, que são recursos utilizados em diversas plataformas, como [12] *Wordpress*, [13] *Drupal*, [14] *Joomla*, [15] *Elgg*, entre outras. O *plugin* tem capacidade de facilitar a customização e integração de recursos extras, de forma [16] *plugin-play*, às aplicações. Essa abordagem é utilizada no *DeepNLPF* para descobrir de forma automática novas ferramentas de PLN adicionadas ao *framework*, como também, um padrão para definir como essas ferramentas

[12] https://br.wordpress.org/
[13] https://www.drupal.org/
[14] https://www.joomla.org/
[15] https://elgg.org/
[16] *Plug and play* é uma das expressões da língua inglesa muito usada na informática que significa "ligar e usar".

devem ser integradas, configuradas e utilizadas. Sendo assim, esse recurso visa facilitar e padronizar o trabalho exaustivo que existe na integração de ferramentas de PLN ao pipeline de análises.

4.3.2.1 Visão de Lógica

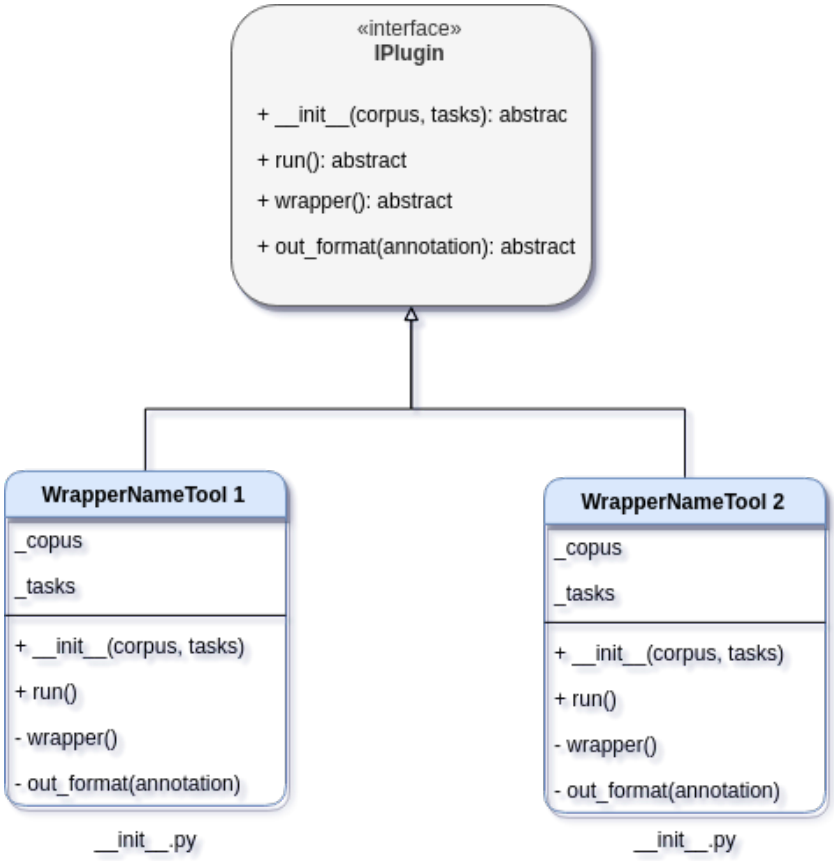
O diagrama de classe do *plugin*, foi modelado a partir do *Design Pattern Template Method*, apresentado no Capítulo 2, sendo assim, todos os *plugins* construídos pelo usuário, devem obrigatoriamente implementar todos os métodos abstratos da *interface IPlugin*. Como mostra a Figura 34, a *interface IPlugin* possui três métodos abstratos que todas as classes concretas devem implementar. Ou seja, as classes que estendem dessa classe abstrata devem incluir seus métodos: *___init___(corpus, tasks), run(), wrapper(sentence) e out_format(annotation)*. Respectivamente, o primeiro método é iniciado automaticamente quando o objeto da classe é instanciado, recebendo por parâmetro os dados a serem processados e as análises linguísticas a serem executadas. Também, é nesse método que se deve-se apontar os caminhos para diretórios da ferramenta quando necessário e inicializar qualquer outro serviços quando necessário. O segundo método, é utilizado para executar o processamento das análises. Nele é definido a estrategias de processamento utilizando *multithreads* quando necessário. No terceiro método, deve-se conter todas as funções de acesso a ferramenta de PLN externa, ou seja, às análises linguísticas que serão disponibilizadas. E por fim, o quarto método deve conter a lógica para formatar os dados de saída para o formato especificado no Bloco de Código 7. Na linha 1 um documento no formato *JSON* é criado, na linha 2, tem o id do *corpus* a ser processado; na linha 3, o nome da ferramenta utilizada, na linha 4, recebe o resultado da analise, e por fim, na linha 5, é registrada a data e o tempo em que foi processado o *corpus*. Cada método citado, deve ser implementado conforme os requisitos das ferramentas PLN de origem, fazendo todas as alterações necessárias para que o *plugin* funcione corretamente no *framework*.

Algoritmo 7: Formato de Saída da Análises.

```
1     document = {
          "_id_corpus": self._corpus['_id'],
3         "tool": "stanfordcorenlp",
          "annotation": annotation,
5         "register": datetime.datetime.now()
      }
```

O diretório *plugins* agrupará cada *plugin* adicionado ao *framework*. O *plugin* inserido deve estar dentro de um outro diretório nomeado com o nome da ferramenta de PLN a ser integrada, exemplo: *Plugin Tool 1* e *Plugin Tool 2*. Esses diretórios devem conter dois arquivos: *___init___.py e manifest.json*, onde o arquivo *___init___.py* é o *Wrapper* de acesso às funcionalidades da ferramenta de PLN externa a ser integrada ao *DeepNLPF* e o

Figura 34 – Diagrama de classe do *plugin*.

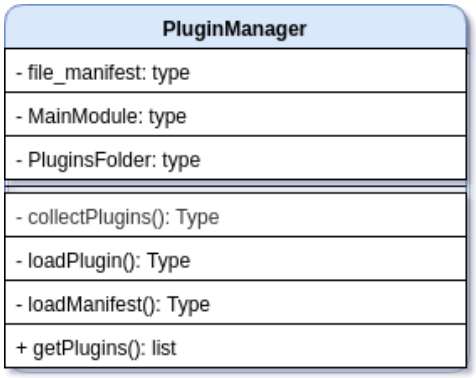


Fonte: do autor.

arquivo *manifest.json*, deve conter todas as informações necessárias para que o *framework* saiba lidar com a ferramenta incluída.

Por fim, tem-se a classe *PluginManager*, conforme ilustra o diagrama da Figura 35. Essa classe é responsável pelo carregamento de todos os *plugins* adicionados ao *DeepNLPF*. Ela funciona basicamente da seguinte forma: através do método *collectPlugins()*, são identificados os *plugins* no diretório *plugins*, verificando se as nomenclaturas dos arquivos, classes e métodos, obedecem aos padrões estabelecidos pela superclasse *IPlugin* e, em seguida, é crianda uma coleção de *plugins*; por meio das funções *loadPlugins()* e *loadManifest()*, esses dois métodos carregam os *plugins* e os *manifests* de cada uma deles em uma lista; através do método *getPlugins()*, o usuário tem acesso a lista que contém todos os *plugins* e suas respectivas atividades de análises linguísticas. É como se todos os *plugins* instalados se tornasse um único *plugin* com todas as funcionalidades herdadas. O Bloco de Código 19 apresenta como funciona o possesso de gerenciamento de *plugins* no *DeepNLPF*. No Apêndice 62 encontra-se o diagrama de fluxograma desse componente.

Figura 35 – Diagrama da classe *PluginManager*.

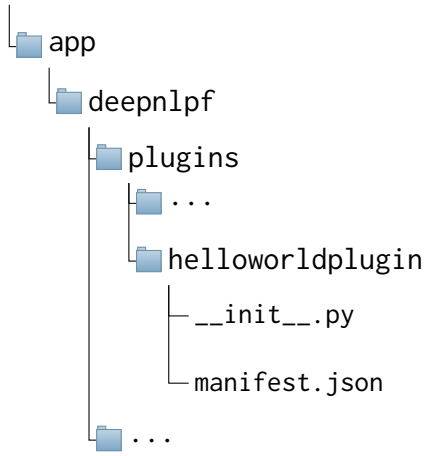


Fonte: do autor.

4.3.2.2 Visão de Implementação

Para criar um *plugin* no *DeepNLPF*, primeiramente é preciso criar um diretório com o nome da ferramenta que será integrada ao *framework*, exemplo: *helloworldplugin*, dentro dessa pasta deve conter os arquivos *___init___.py* e *manifest.json*, como segue na estrutura em árvore ilustrada pela Figura 36.

Figura 36 – Arvore Estrutural do *Plugin*.



Fonte: do autor.

O Bloco de Código 20, ilustra como arquivo *manisfest.json* deve ser implementado. Nele contem diversos campos obrigatórios e facultativos, que descrevem as características básicas do *plugin* que será adicionado ao *framework*. Os campos obrigatórios são: versão do arquivo *manifest*, nome e versão da ferramenta, categoria e por fim a opção se a ferramenta está ativada ou desativada. Já os campos opcionais são: site oficial da ferramenta, descrição básica sobre a ferramenta, nome do *author* do *plugin*, licença de uso da ferramenta, e por fim, campo obrigatório, usuário deve no mínimo adiciona uma análise que a ferramenta realiza. Dessa forma, o *DeepNLPF* terá informações que o ajuda a entender como a

nova ferramenta funciona. O Bloco de Código 8 ilustra o arquivo *___init___.py* deve ser construído.

Na linha 1 do Bloco de Código 8, é incluído a *interface IPlugin*, ela obriga que os modos abstratos sejam implementados. Na linha 3, é definido um nome para a classe que implementa a superclasse *IPlugin* (é importante que o nome da classe comece com o termo *Wrapper* para identificar que não é a ferramenta de PLN propriamente e sim um *wrapper*). A partir da linha 6, deve ser especificada como os métodos *___init___()*, *run()*, *wrapper()* e *out_format(annotation)*. Caso não sejam implementadas as funções abstratas mencionadas, uma das mensagens de erro como ilustra no Bloco de Código 9 será exibido, informando-o que as mesmas deve ser adicionadas. No Apêndice 18 encontra-se o *plugin* do *Stanford CoreNLP* como exemplo.

Algoritmo 8: Modelo do arquivo ___init___.py.

```
1  from deepnlpf.system.iplugin import IPlugin
2
3  class WrapperHelloWorld(IPlugin):
4
5      def __init__(self, corpus, tasks):
6          pass
7
8      def run(self):
9          pass
10
11     def wrapper(self, sentence):
12         pass
13
14     def out_format(self, annotation):
15         pass
```

Algoritmo 9: Mensagem de erro para métodos abstratos não implementados.

```
1  NotImplementedError subclasses must override __init__()
2
3  NotImplementedError subclasses must override run()
4
5  NotImplementedError subclasses must override wrapper()
6
7  NotImplementedError subclasses must override out_format()
```

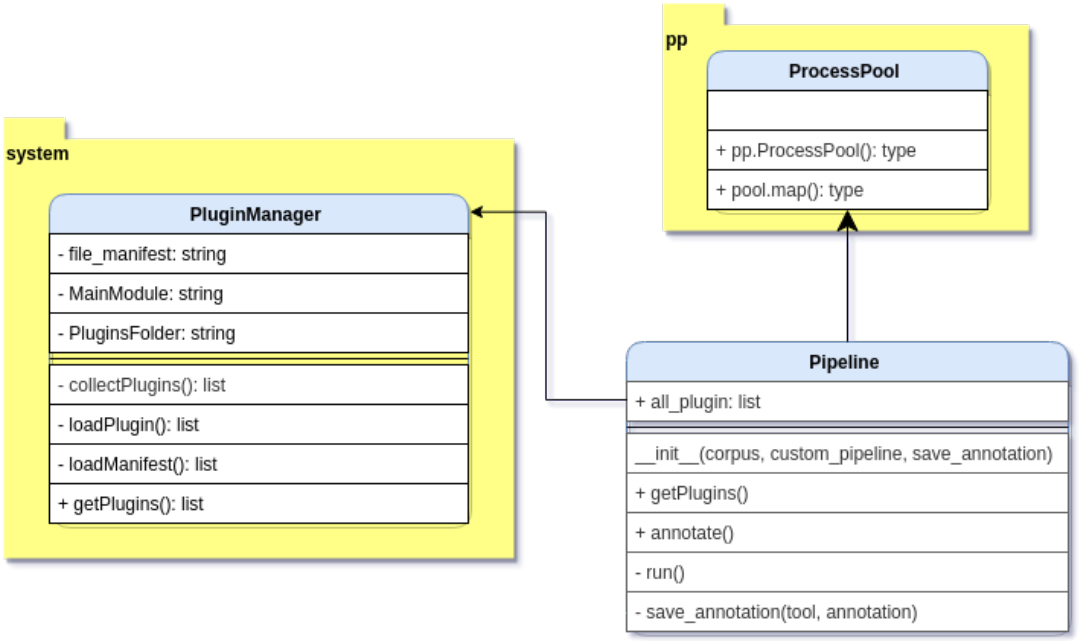
4.3.3 Componente *Pipeline*

Pipeline, pode-se dizer que é o componente mais importante do *DeepNLPF*. Esse componente é responsável pela customização e a execução das tarefas de PLN; como também, é ele que contém toda a lógica de como se dará a orquestração do *multithreads* e paralelismo entre as ferramentas de PLN e, por fim, salvar as anotações linguísticas.

A Figura 37 exibe o diagrama de classe desse componente. A classe *Pipeline* inclui a classe *PluginManager* e acessa a lista de *plugins* ativos, através do método *getPlugins()*. Na classe *Pipeline*, tem-se o construtor *___init___(corpus, custom_pipeline,*

save_annotation). Sempre que o usuário instanciar um objeto a partir dessa classe, ele deve passar para o construtor os parâmetros: *corpus* a ser processado e o *pipeline* customizado contendo as análises linguística a serem executadas, o terceiro parâmetro salvar a anotação é opcional. O método *annotate()* recebe as tarefas a serem executas e as distribui para o *Pool* de processamento, que por sua vez, orquestra a estratégia de paralelismo, chamando o método *run()* (método de execução) de cada ferramenta de PLN selecionada, passando por parâmetro para os *Wrappers* o *corpus* e o *pipeline* de análises linguística selecionadas. Cada sentença é processada por meio de *threads* (4 *threads* por padrão), esse valor pode ser alterado de acordo com a necessidade e disponibilidade de recursos computacionais. Sendo assim, os processos são executados de forma assíncrona. Cada processo é executado em um núcleo do computador individualmente e podem tem vários subprocessos *multithreads*, não necessitando um processo terminar par o outro começar, a não ser que todos os núcleos estejam ocupados, isso ocorrerá quando quantidade de ferramentas a serem utilizadas é maior que a quantidade de núcleos presentes no computador, porém ele saberá como arbitrar essa fila de processos. Quando cada processo é concluído o *Pool* recebe as anotações realizadas por cada ferramenta e envia para o método *save_annotation()* que verifica se é pra salvar a análise anotada ou apenas retorná-la para o usuário. O diagrama de atividade da Figura 38 mostra com mais detalhes os passos que ocorrem nesse componente.

Figura 37 – *Pipeline* - Diagrama da Classe.

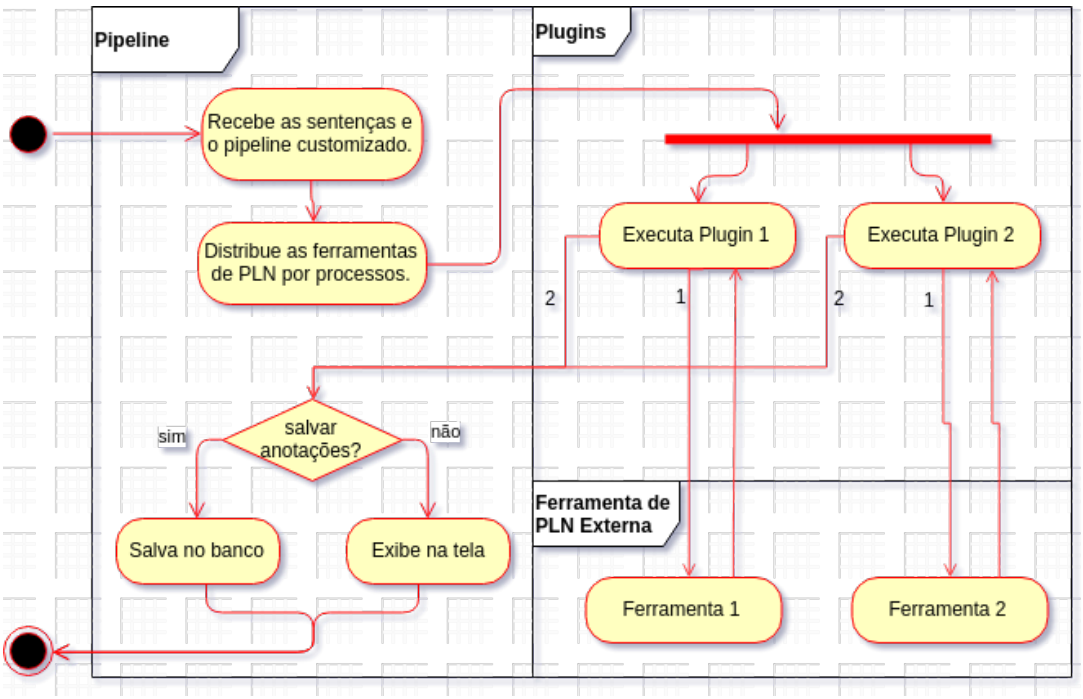


Fonte: do autor.

O Bloco de Código 21 apresenta os trechos principais de como foi implementado o componente *Pipeline* e como ocorre o processamento paralelo no *DeepNLPF*. Cada trecho, segue precedido de comentários explicando o que acontece naquele dado momento,

dispensando comentários posteriores.

Figura 38 – *Pipeline* - Diagrama de Atividade.



Fonte: do autor.

4.3.4 *Pipeline* do *DeepNLPF*

O *pipeline* do *DeepNLPF* é customizado por ferramentas de PLN de terceiros, entre elas (*Stanford CoreNLP*, *SpaCy*, *CogComp*, *SEMAFOR* e *PyWSD*). As análises linguísticas são divididas nos níveis (Léxico, Sintático e Semântico). A Figura 39 ilustra detalhes de como esse pipeline está estruturado.

Primeiramente, deve-se extrair as sentenças do *corpus* e estruturá-las em um documento de texto contendo uma sentença por linha. De preferência, as sentenças já devem estar limpas (remoção de possíveis ruídos, exemplo: *stopwords* e símbolos.) pois interferem na análise. Após isso, esse documento pode ser enviado para o *DeepNLPF* juntamente com as análises e ferramentas selecionadas para processamento.

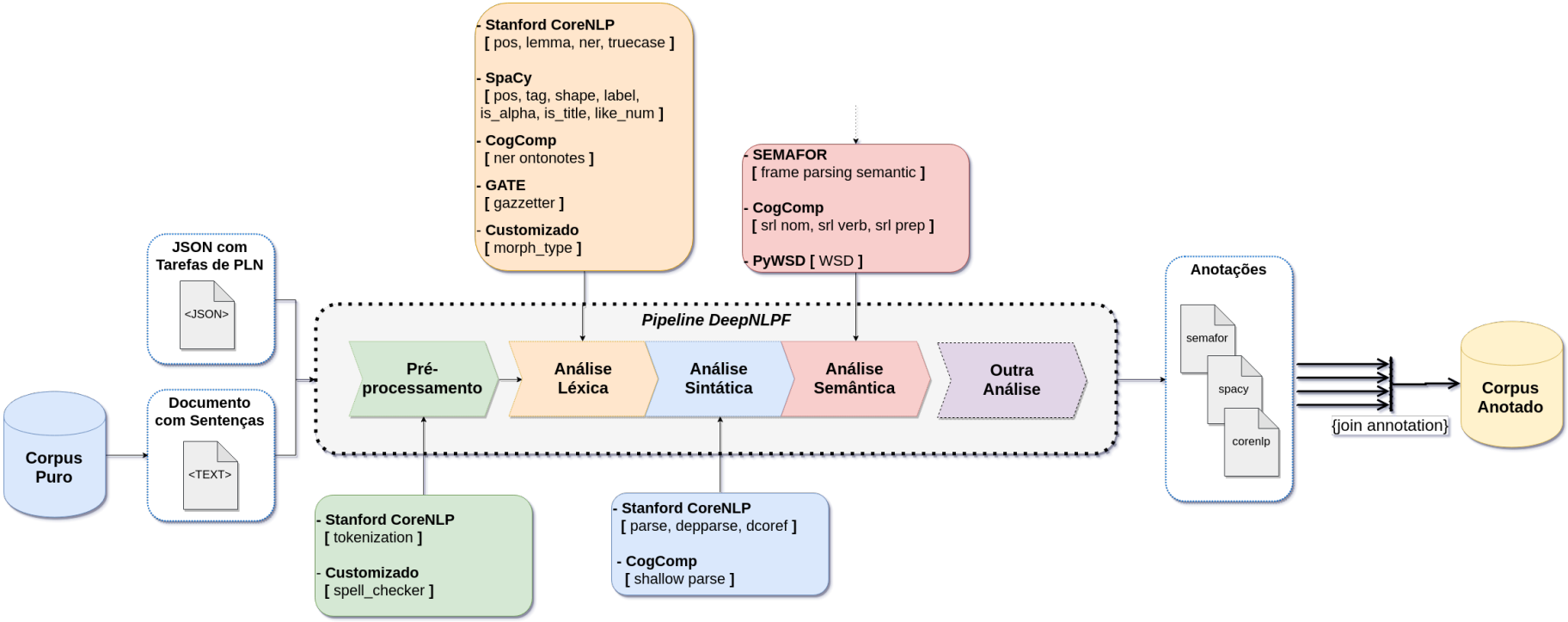
O *DeepNLPF* por padrão irá pré-processar as sentenças objetivando tokenizar todas elas com o tokenizador do *Stanford CoreNLP*, para manter uma uniformidade em números de *tokens*, e assim enviar as sentenças já tokenizadas para as demais ferramentas. Isso evita que cada ferramenta tokenise as sentenças de formas diferentes, o que pode dar conflitos na quantidade de *tokens*. Outras análises podem ser feitas nesse momento de "pré-processamento", exemplo *tokenization* e [17]*spell checker*.

[17] *Spell Checker* - verificação da ortografia antes do processamento dos dados.

No primeiro nível, análise léxica, serão executadas as análises e as ferramentas selecionadas, ou seja, com base no arquivo de entrada *JSON* tarefas de PLN, serão executadas as ferramentas e as análises correspondentes. Nesse nível o *framework* disponibiliza as seguintes análises (***Stanford CoreNLP***: *pos, lemma, ner e true case;* ***SpaCy***: *pos, tag, shape, label, is alpha, is title, e like num*; ***CogComp****: ner ontonotes*; ***GATE****: gazzetter*; **Customizado**: *morph type*). Em seguida tem-se as análises sintáticas (***Stanford CoreNLP****: parse, depparse, dcoref* e *CogComp: shallow parse*) e por fim tem-se as análises no nível semântico (***SEMAFOR****: frame parsing semantic;* ***CogComp****: srl verb, srl, prep e srl nom e* ***PyWSD****: wsd*). Entretanto para as análises *wsd* desambiguadores de sentidos das palavras, podem ser combinadas com recursos externos (dicionários linguísticos, ontologias, *tesauros*, etc.), *WordNet, WordNet Domain, Asumo* entre outros, que trazem informações detalhadas e mais precisas sobre contextos de uma da palavra. Em outras análises, outros níveis podem ser adicionados conforme a necessidade do projeto, por exemplo: análise pragmática.

Após concluir todos os passo descritos, cada ferramenta produzirá um documento contendo suas anotações individuais, contendo suas análises por sentenças. Por fim o *DeepNLPF* recupera essas anotações e compila em um único documento de anotação de forma estruturada por níveis de análises.

Figura 39 – *Pipeline* do *DeepNLPF*.



Fonte: do autor.

4.3.5 Estratégia de Processamento

Como visto no Capítulo 2, o *pipeline* de análise linguística é implementado por cada ferramenta de PLN para elas executarem as tarefas selecionadas pelo usuário. Ou seja, o usuário, utilizando uma ferramenta de PLN, escolhe os tipos de análises que serão utilizadas para processar seu *corpus*. Essas análises serão executadas em um pipeline, onde a saída de cada análise é a entrada da próxima análise e assim por diante. Porém, quando é necessário processar um *corpus* utilizando diversas análises provenientes de ferramentas de PLN diferentes, o processo pode se tornar dispendioso de tempo e, se não for utilizada uma estratégia de multiprocesso e/ou paralelismo eficiente, pode durar dias ou até mesmo semanas processando o *corpus*. Para mitigar tal problema, a Figura 40 ilustra a estratégia adotada pelo *DeepNLPF*. Quando se tem um único computador para processar toda as análises do *pipeline*, é mais vantajoso utilizar multi-processos. Para isso, o *DeepNLPF* identifica os núcleos presentes no processador e distribui cada ferramenta de PLN em um processo assíncrono, otimizando as tarefas do *pipeline*. Isso é possível devido o *DeepNLPF* fazer uso de recursos do *framework* [18]*Pathos*, um *framework* para computação heterogênea. *Pathos* fornece principalmente os mecanismos de comunicação para configurar e iniciar cálculos paralelos através de recursos heterogêneos (MCKERNS; AIVAZIS, 2010). O *DeepNLPF* também pode distribuir seu processamento para outros nós (computadores) usufruindo de processamento paralelo [19]*"Cluster"* para obter o máximo de desempenho. Isso é possível, devido ao *Pathos* fornecer *stagers* e [20]*launchers* para computação paralela e distribuída, onde cada lançador contém a lógica sintática para configurar e lançar tarefas em um ambiente de execução (MCKERNS et al., 2012).

4.3.6 *API* - Executando *Pipeline* Customizado

O *DeepNLPF* foi construído usando a linguagem de programação *Python*, assim como sua API que tem o objetivo de fornecer uma *interface* para os usuários criarem aplicativos de PLN. A linguagem *Python* foi escolhida, por possuir recursos para mineração de texto, computação científica, suporte multi-paradigma (orientado a objetos, imperativo, funcional e procedural), flexibilidade em interoperabilidade com outras linguagens, entre outras características e razões.

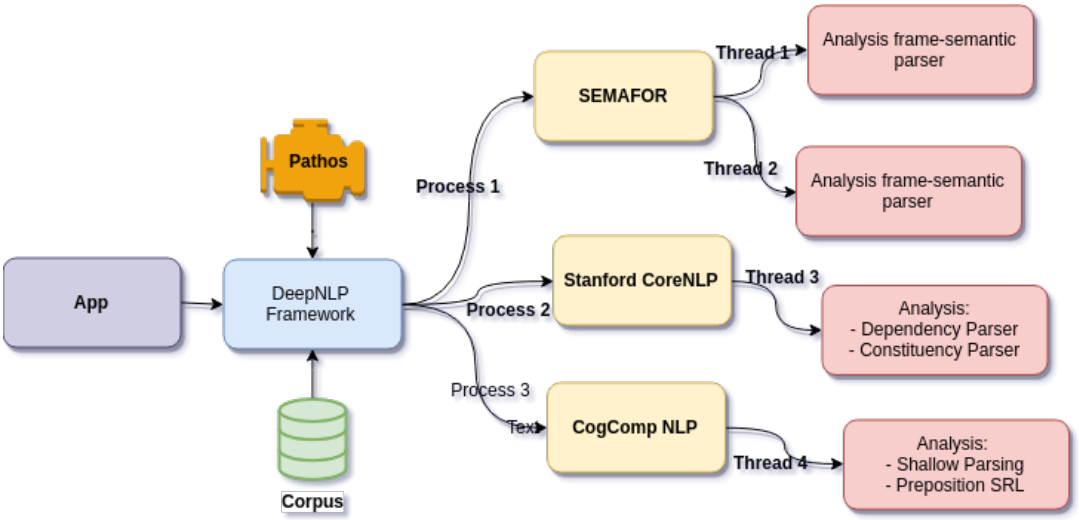
A *API* é uma das portas de entrada para começar a utilizar o *framework*. Uma característica da linguagem *Python* é permitir a criação de módulos. Esses módulos podem ser uma biblioteca, *toolkit* e até mesmo um *framework*, que são empacotados e distribuídos.

[18] https://pypi.org/project/pathos/

[19] Cluster é um termo em inglês que significa "aglomerar" ou "aglomeração" e pode ser aplicado em vários contextos. No caso da computação, o termo define uma arquitetura de sistema capaz combinar vários computadores para trabalharem em conjunto ou pode denominar o grupo em si de computadores combinados.

[20] *Launchers, são iniciadores de tarefas em computação paralela. Alguns exemplos de launchers são: launchers baseado em MPI sem fila, launchers baseado em ssh e launchers de multiprocessamento.*

Figura 40 – Estratégia de Paralelismo e *Multithreads*.



Fonte: do autor.

Sendo assim, para que o usuário comece a criar uma aplicação usando o *DeepNLPF*, deve necessário importar no projeto e instanciar um objeto da classe *Pipeline* por exemplo, criar uma anotação customizada e executar. O Bloco de Código 10 detalha como importar o componente *Annotation*, como criar um *pipeline* customizado usando JSON, e inicia um processamento de análise de um *corpus*. Na linha 1, o usuário importa o componente *Pipeline*; em seguida na linha 3, é criado um *pipeline* customizado no formato JSON, contendo as ferramentas de PLN a serem utilizadas e as suas respectivas análises a serem executadas. Na linha 23, é definido o *corpus* que será processado. Na linha 25, um objeto *annotation* da classe *Annotation* é instanciado passando por parâmetro o id do *corpus* a ser processado, e as configurações customizadas do *pipeline*. Na linha 27, é executada a anotação e impressa no terminal o resultado das análises. A linha 29 é uma extensão da linha 25, adicionando o parâmetro "True", que significa que as análises devem ser salvas no banco de dados e não visualizado no terminal comandos.

4.3.7 *DeepNLPF API RESTFul*

Uma outra forma de utilizar o *DeepNLPF* é através de sua *API RESTFul*. Ela permite que o usuário faça solicitações de serviços por meio de tecnologias *web*, *GET* e *POST*. Essa característica de serviço flexibiliza a construção de aplicações de PLN que terão serviços na *internet* (MASSE, 2011). A *API RESTFul* foi construída utilizando o *micro-framework* [21] *Flask*. Para utiliza-la, deve-se executar o serviço (*python run.py*) e posteriormente, em uma aplicação cliente, realizar as requisições via *POST* ou *GET*, passando os parâmetros

[21] http://flask.pocoo.org/

Algoritmo 10: Executando um Pipeline de Análises Customizado.

```
from deepnlpf.pipeline import Annotation

custom_pipeline = {
    'tools': [
        {
            'stanfordcorenlp': {
                'pipeline': ['pos', 'lemma', 'ner', 'parse', 'depparse']
            }
        },
        {
            'semafor': {
                'pipeline': ['parsing']
            }
        },
        {
            'cogcomp': {
                'pipeline': ['SHALLOW_PARSE', 'NER_ONTONOTES', 'SRL_NOM', '
                    SRL_VERB', 'SRL_PREP']
            }
        },
    ]
}

id_corpus = "5d0f78fedca12ea9d9363029"

annotation = Annotation(id_corpus, custom_pipeline)

print(annotation.annotate())

annotation = Annotation(id_corpus, custom_pipeline, True)
```

corretos segundo o *user guide* ou a [22]documentação no [23]*Postman*. No Bloco de Código 11 é apresentado o exemplo de como fazer uma solicitação via *GET* a *API RESTFul*.

Algoritmo 11: Salvando um documento via GET.

```
# request
http://127.0.0.1:5000/corpus_upload?path_corpus=/home/fasr/OUTPUT_sentences_2010
    .txt

# response
{
    "corpus": {
        "_id": "5d1943a1cf17107dbeda89c7"
    }
}
```

4.3.8 Componente *Templates*

No Capítulo 2 foi falado sobre formatos de saída de anotações linguísticas (*in-line, stand-off e híbrido*) e qual a sua importância. Nesse sentido, uma ferramenta de PLN deve dispor quando necessário uma saída das anotações segundo um dos padrões já definidos

[22] https://documenter.getpostman.com/view/2943437/SVSGMq2A

[23] O *Postman* é uma ferramenta que tem como objetivo testar serviços *RESTful (Web APIs)* por meio do envio de requisições HTTP e da análise do seu retorno.

Algoritmo 12: Processando um corpus via GET.

```
# request
http://127.0.0.1:5000/annotation_processing?id_corpus=5d0f7696ffee87d0b3ac1faa&
    properties={"tools":[{"stanfordcorenlp":{"pipeline":["pos","lemma","ner"
    ]}}]}

# response
[
    {
        "_id_corpus": "5d0f7696ffee87d0b3ac1faa",
        "annotation": [...],
        "register": "2019-06-30T21:16:19",
        "tool": "stanfordcorenlp"
    }
]
```

na literatura, para facilitar que outras aplicações possam consumir esses dados. Nesse sentido, o *DeepNLPF* propõem um esquema de anotação com base no modelo híbrido, capaz de encapsular e organizar as anotações por nível linguístico (sintático, léxico e semântico), no formato de arquivos *JSON* e/ou *XML*.

Todas as tarefas de análises linguísticas realizadas pelas ferramentas de PLN integradas pelo *DeepNLPF*, fornecem as anotações no formato *JSON* por padrão. Sendo assim, o componente *Template* recupera essas anotações e mapeia para o modelo estabelecido, estruturando todas as análises por camada linguística de forma que exista uma mesclagem entre essas variedades de análises, conforme ilustra o Bloco de Código 16 que está no Apêndice.

Quando o usuário necessitar dessa anotação em um outro formato de arquivo, como *XML*, por exemplo, o componente *Templates* realiza um *parse* no arquivo *JSON*, convertendo para o formato desejado. Na Figura 66, que está no Apêndice, contem a ilustração de como o arquivo de anotação *JSON* é convertido para o formato *XML*. O *parser* para conversão dos arquivos de dados, foi construído com base no padrão de projeto *Adapter*, citado no Capítulo 2, que proporciona as características necessárias para implementar uma solução que realize essa transformação e seja extensível para outros formatos de arquivos, quando necessário. No Bloco de Código 17 que está no Apêndice, exibe o resultado de saída da conversão do arquivo *JSON* para *XML*. Já no Bloco de Código 16 que está no Apêndice, é exemplificado como são anotadas as sentenças analisadas. Cada *corpus* processado gera uma arquivo *(filename)*, que contém um *(document)*, esse documento é composto por N sentenças *(sentences)*, cada sentença contém suas anotações, a começar pelo seu identificador *(s_id)*, *(text)*, números de *(tokens)* que contém a sentença *tokens*, informações extras como análise de sentimento *(sentiment)* e por fim as camadas de anotações lexical, sintática e semântica *(lexical_analisis, syntactic_analisis e semantics_analysis)*.

4.3.9 Componente *Statistcs*

O componente *Statistcs* é um componente que permite gerar estatísticas do *corpus* para o usuário compreender melhor os dados. Esse componente retorna estatísticas como: o número total de sentenças no *corpus*, o número mínimo e máximo de palavras "*tokens*"por sentença, o número médio de palavras por sentença, a frequência de palavras e a frequência de *POS tags* "classes gramaticais". Outras estatísticas personalizadas podem ser inseridas de acordo com a necessidade do usuário. Por exemplo, o usuário pode adicionar um novo método no componente *Statistics* para gerar estatística da frequência de bigramas e/ou trigramas contidos no *corpus*.

Esse componente utiliza a ferramenta de PLN *Stanford CoreNLP* e o *NLTK* por padrão para realizar algumas análises básicas como: *tokenization*, *entences ssplit*, *POS tag*, etc. O Bloco de Código 13 a seguir, exemplifica como utilizar esse componente e exibe algumas estatísticas.

Algoritmo 13: Exemplo de Utilização do Componente *Statistics*.

```
from deepnlpf.statistics import Statistics

id_corpus = "5d0f7696ffee87d0b3ac1faa"

statistics = Statistics(id_corpus)
print( statistics.full_statistics() )

{
    '_id_corpus': '5d0f78fedca12ea9d9363029',
    'total_tokens': 16768,
    'total_sentences': 1000,
    'num_min_tokens_sentences': 4,
    'num_max_tokens_sentences': 72,
    'average_tokens_sentences': 16,
    'words_frequency': [...],
    'post_tag_frequency': [...]
}
```

O componente *Statistics* também fornece ao usuário a opção de visualização dos dados de forma interativa, esse recurso permite que o usuário se concentre na análise dos dados poupando tempo escrevendo códigos para esse fim. Os recursos utilizados para gerar os gráficos e visualização são provenientes da biblioteca [24] *PlotLy*.

Visualizações de gráficos gerados automaticamente, são essenciais para análises rápidas em diversas aplicações, como por exemplo, o aplicativo *aBoard* murta2015aplicando, que trabalham diretamente com a linguagem, ajudando a pessoas com baixa ou nenhuma comunicação verbal a se comunicarem.

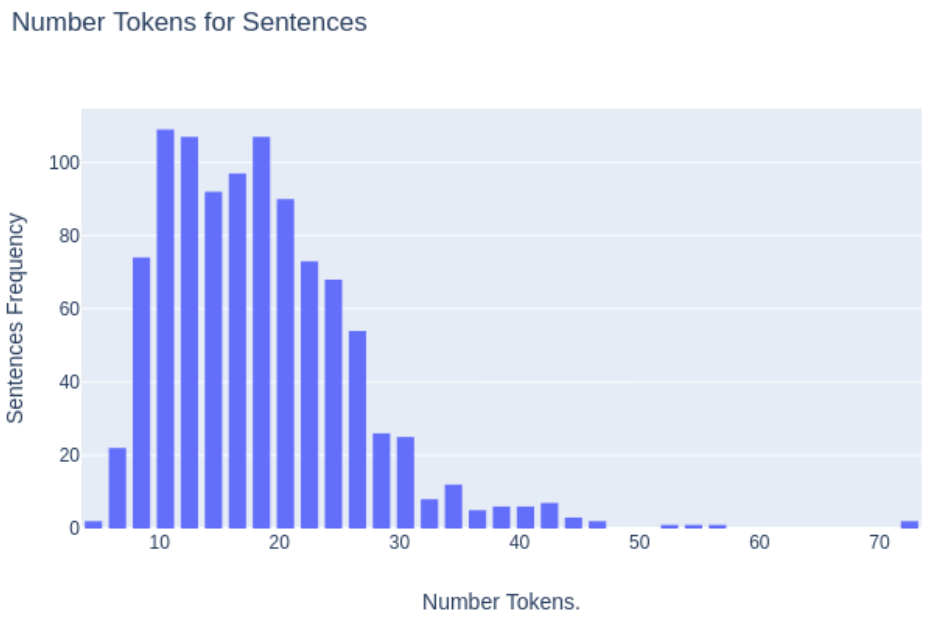
A Figura 41 mostra um histograma com a distribuição de frequência da quantidade de palavras por sentenças. Aplicando essa análise ao cenário da aplicação citada, ela pode

[24] *Plotly*, também conhecida por sua *URL*, *Plot.ly*, é uma empresa de computação técnica sediada em Montreal, *Quebec*, que desenvolve ferramentas de análise e visualização de dados on-line. https://plot.ly/

informar o tamanho das frases que uma criança já consegue formar com o aplicativo *aBoard*, demonstrando um avanço na comunicação da criança.

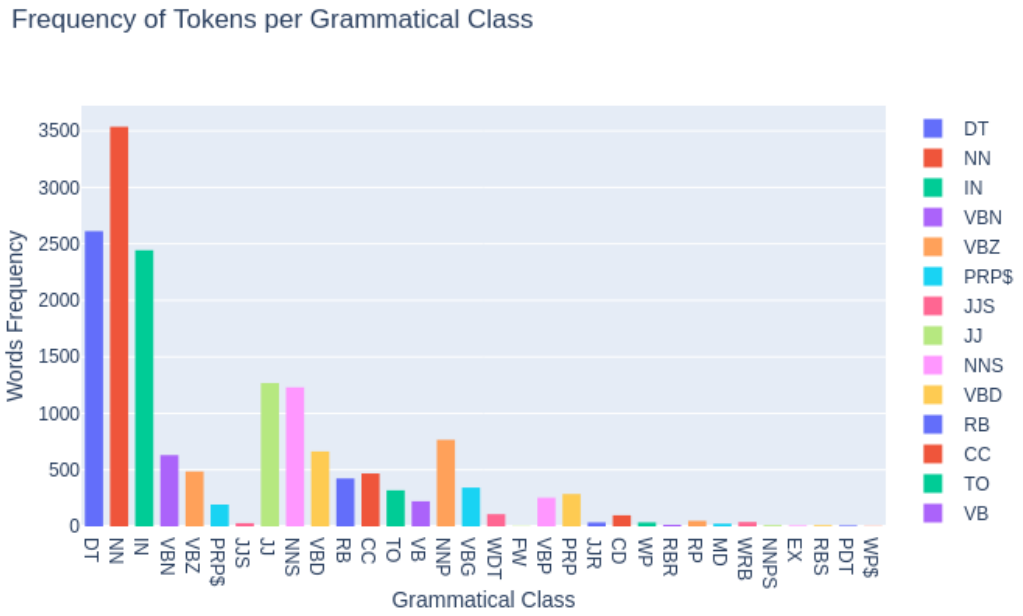
Outras análises interessantes são o histograma da frequência de *tokens* por classes gramaticais como na Figura 42 e a nuvem das palavras frequentes como mostra na Figura 43. Dentro do mesmo cenário citado anteriormente. Esses gráficos podem apontar o quão rico é o vocabulário de uma criança, analisando a repetição de palavras e a mistura de palavras na linguagem.

Figura 41 – Histograma - Frequência de *Tokens* por Sentenças.



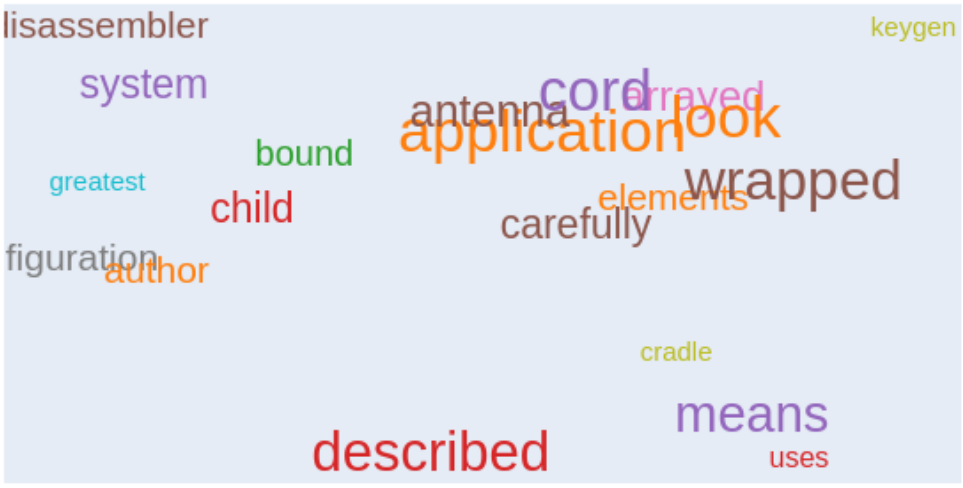
Fonte: do autor.

Figura 42 – Histograma - Frequência de *Tokens* por Classe Gramatical.



Fonte: do autor.

Figura 43 – Nuvem de Palavras Frequentes.



Fonte: do autor.

4.3.10 *DashBoard*

O *DashBoard* é uma *Graphical User Interface* (GUI) exclusiva para o usuário utilizar o *DeepNLPF* acessada através do [25]*Browser*. Esse *DashBoard* é uma aplicação cliente que possibilita o usuário experimentar os recursos da ferramenta *DeepNLPF* sem a necessidade de escrever nenhuma linha de código. O *DashBoard* foi implementado usando o

[25] Navegador de Internet.

Micro-Framework Flask. O *Template* da *interface* gráfica, é um *Web-Kit BootStrap Free* da [26]*Creative Tim*. Em resumo, o *DashBoard* é uma aplicação cliente que consome os serviços do *DeepNLPF* por meio da sua *API RESTFul*. Mais detalhes são encontrados na documentação oficial através do endereço: https://deepnlpf.github.io/site/.

4.4 CONSIDERAÇÕES FINAIS

Este capítulo apresentou o *DeepNLPF*, seu propósito e possíveis aplicações. Foram igualmente descritos em detalhes sua arquitetura e componentes principais, seu sistema de armazenamento de dados, e as estratégias de *plugins* adotadas para a futura integração de novas ferramentas, bem como a customização do *pipeline* de análises. Outros aspectos como processamento a fim de otimizar o tempo de análise, união das anotações linguísticas foram discutidos. Finalmente, uma interface gráfica chamada *DashBoard* bem como seu modo de operação pelo usuário foram apresentados. Enquanto que no Capítulo 3 foi responsável por apresentar uma discussão de uma comparação qualitativa do *DeepNLPF* quando comparado com outros trabalhos relacionados dos estado da arte, o próximo capítulo irá fazer uma análise comprativa qualitativa que por sua vez, conterá outros aspectos da contribuição do presente trabalho.

[26] https://www.creative-tim.com/product/now-ui-dashboard

5 AVALIAÇÃO

Neste capítulo, apresenta-se inicialmente uma avaliação quantitativa de performance do protótipo que implementa a proposta do *DeepNLPF* e em seguida sua análise comparativa qualitativa com outros trabalhos. Todos os experimentos descritos neste capítulo têm por objetivo responder as seguinte perguntas experimentais:

- **PE1:** Qual o custo computacional das ferramentas de PLN de terceiros executadas sequencialmente?
- **PE2:** A estrategia de paralelismo proposta do *DeepNLPF* otimiza o tempo de processamento? Qual seu desempenho para execução de *pipeline* customizado em relação as execuções das ferramentas de PLN de terceiros individualmente?
- **PE3:** Quais os critérios podem ser considerados quando avalia-se *framework* de PLN que suportem a integração de *pipelines* de PLN distintas pré-definidas ou customizadas?

Para responder as perguntas PE1 e PE2, o tempo de execução do código [1]"*turn-around time*"esse indicador tempo de execução é o mais conhecido e empregado na prática para avaliar desempenho eyerman2008system, lutov2018clubmark e representado através de uma ordem de grandeza utilizando o estudo de [2]*Profiling*. Outras análises como: uso de memória RAM e [3]*Swap* não foram exploradas nesse momento, pois os modelos carregados na memoria provenientes das ferramentas de PLN de terceiros são complexos, sendo necessário um estudo mais aprofundado individualmente para cada ferramenta.

Para responder a pergunta PE3, alguns critérios foram adotados para analisar/comparar os trabalhos relacionados. Esses critérios, objetivam verificar se os aspectos inerentes a essa proposta cobrem as características já apresentadas na Seção 3.1 deste documento. Essas características são definidas tendo em vista que as mesmas podem auxiliar no desenvolvimento, aprimoramento e uso de sistemas de PLN customizados.

[1] *Turn-around Time* ou tempo de resposta é definido como o intervalo entre o instante em que o programa é submetido ao sistema até o momento em que toda a sua saída é produzida e o programa termina.

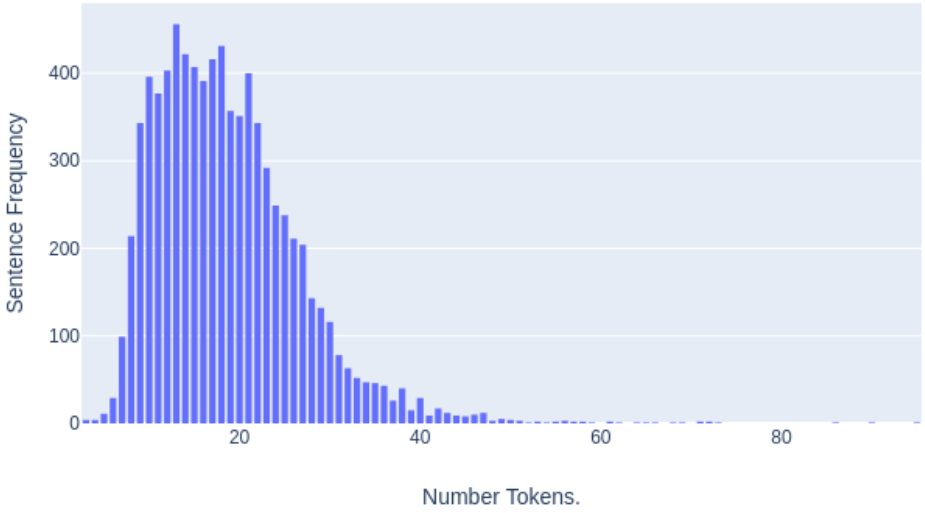
[2] *Profiling* é uma técnica que permite identificar os pontos mais intensivos dos recursos em um aplicativo. Ou seja, uma análise dinâmica que mede o tempo de execução do programa e tudo o que o compõe. Isso significa medir o tempo gasto em cada uma das suas funções. Fornecendo dados sobre onde seu programa está gastando e qual área pode valer a pena otimizar (LANARO, 2019).

[3] *Swap* é uma memória virtual, uma técnica que usa a memória secundária como uma cache para armazenamento secundário.

5.1 AVALIAÇÃO DO PROTÓTIPO - RESPONDENDO AS PES

Nesta seção, avalia-se o desempenho do *DeepNLPF*. Para isso, foi selecionado o *corpus* [4]*SemEval* 2010, que é uma competição anual que objetiva avaliar sistemas de análise semântica kim2010semeval vem diversas tarefas de Mineração de Textos. Desse *corpus*, foram extraídas 8000 sentenças e replicadas em três partes, onde supostamente foi definido com: 1ª parte, contendo 1000 sentenças (*corpus* pequeno); a 2ª parte, contendo 4000 sentenças (*corpus* médio) e a 3ª parte, com as 8000 sentenças (*corpus* grande). No histograma da Figura 44 é mostrado a distribuição de frequência da quantidade de *tokens* em relação a quantidade de sentenças, correspondente as informações presentes na Tabela 10, que descreve algumas características do *corpus*. Essas informações são úteis, pois o tamanho da sentença impacta diretamente no tempo de processamento das análises.

Figura 44 – *Corpus SemEval* 2010 - Frequência de *tokens* por sentenças.



Fonte: do autor.

Tabela 10 – Características dos dados extraídos do *corpus*.

Corpus	SemEval
Total de Sentenças Extraídas	8000
Total de *Tokens*	135.886
Tamanho Min. das Sentenças	4 *tokens*
Tamanho Max. das Sentenças	95 *tokens*
Tamanho Médio das Sentenças	17 *tokens*

Fonte: do autor.

[4] https://www.cs.york.ac.uk/semeval2010$_W$*SI*/*datasets.html*

Para realização dos experimentos de *Benchmarks*[5], foi utilizado uma computador com a seguinte configuração: 16GB de memória *RAM*, processado *Intel Core* i5-6200U CPU @ 2.30GHz x 4, sistema operacional *Linux Ubuntu 19.04 64 bits* e unidade de disco *SSD* 120GB. As execuções de todas as ferramentas foram iniciadas a partir de um *Wrapper Python* utilizando *cProfile*, módulo *Python* comumente utilizado em análise de *Profiling*, que é uma análise dinâmica que mede o tempo de execução do programa e tudo o que o compõe.

Para visualização dos resultados, foi empregada a ferramenta [6]*SnakeViz*, um visualizador gráfico para a saída do módulo *cProfile*. A Ferramenta *SnakeViz* fornece uma *interface* com gráficos interativos, permitindo navegar nos processos quando selecionados, assim, acompanhar a fundo quais os blocos de código (*loops*, funções, etc) consomem mais tempo de processamento. Também é possível visualizar uma lista completa de cada processo, funções executadas, com os seus respectivos tempos de execução entre outras informações mais abrangente. O *SnakeViz* tem dois estilos de visualização, A Figura 45 ilustra a visualização de forma geral dos resultado de uma análise *Profile* representando o gráfico do tipo *Icicle* (o padrão), e o gráfico do tipo *Sunburst* é ilustrado na Figura 46. Em ambos os gráficos, a fração de tempo consumido em uma função é representada pela extensão de um elemento de visualização, seja a largura de um retângulo ou a extensão de um arco.

Nas funções de visualizadas no modo *Icicle* são representadas por um retângulo. Uma função raiz (função inicial) é o retângulo mais alto, as funções mais abaixo são as funções chamadas posteriormente formando assim uma pilha de execuções de funções. A quantidade de tempo gasto dentro de uma função é representada pela largura do retângulo. Um retângulo que se estende ao longo da maior parte da visualização representa uma função que está ocupando a maior parte do tempo de sua função de chamada, enquanto um retângulo mais curto representa uma função que está usando quase todo o tempo.

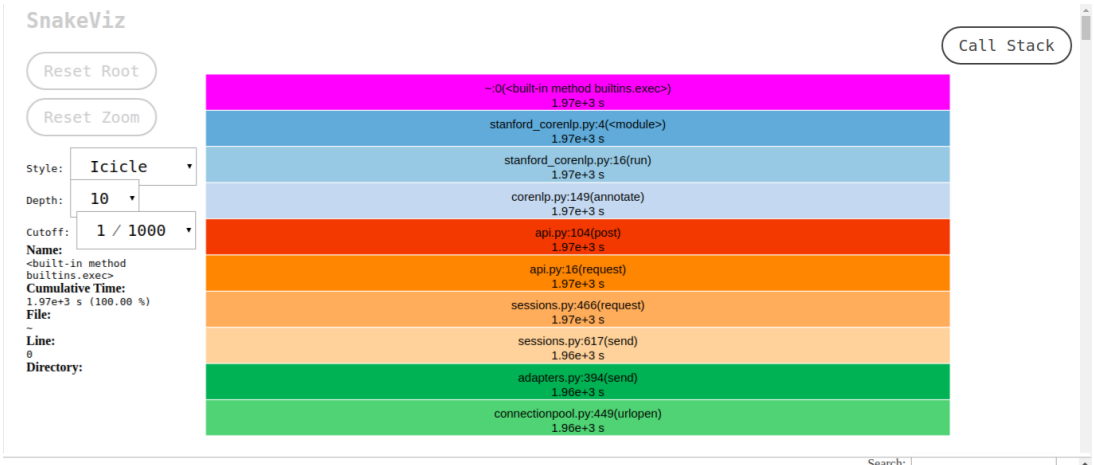
Nas funções de visualização no modo *Sunburst* são representadas como arcos. Uma função raiz é um círculo no meio, as outras funções chamadas posteriormente são envolvida nela assim por diante. A quantidade de tempo gasto dentro de uma função é representada pela extensão angular do arco (até onde vai o circulo). Um arco que envolve a maior parte do caminho ao redor do círculo representa uma função que está consumindo a maior parte do tempo de sua função de chamada, enquanto um arco menor representa uma função que consome pouco tempo. Percebe-se que tais tipos de visualização são realmente úteis pois permite rapidamente o usuário de chegar a conclusão entre diversos gráficos de desempenho em sistemas/funções distintos.

Conforme ilustra a Figura 47, as informação da função é exibida ao colocar o cursor do *mouse* sobre um retângulo ou arco, assim destacará essa função e quais outras instân-

[5] *Benchmarks* é o ato de executar a fim de avaliar o desempenho relativo de um sistema computacional através de uma série de testes padrões e ensaios nele sakis2019

[6] https://jiffyclub.github.io/snakeviz/

Figura 45 – SnakeViz *Profile* - Gráfico do Tipo *Icicle* Exemplo Visão Geral.



Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.02398	0.02398	0.02398	0.02398	~:0(<method 'writelines' of '_io._IOBase' objects>)
813	0.07099	8.732e-05	0.07099	8.732e-05	~:0(<method 'write' of '_io.TextIOWrapper' objects>)
1000	0.001402	1.402e-06	0.001402	1.402e-06	~:0(<method 'write' of '_io.StringIO' objects>)
733	0.00017	2.319e-07	0.00017	2.319e-07	~:0(<method 'values' of 'mappingproxy' objects>)
2003	0.00059	2.946e-07	0.00059	2.946e-07	~:0(<method 'values' of 'dict' objects>)
10000	0.00265	2.65e-07	0.00265	2.65e-07	~:0(<method 'values' of 'collections.OrderedDict' objects>)
6154	0.003033	4.929e-07	0.003033	4.929e-07	~:0(<method 'upper' of 'str' objects>)
3	4e-06	1.333e-06	4e-06	1.333e-06	~:0(<method 'update' of 'set' objects>)
3367	0.004339	1.289e-06	0.004386	1.303e-06	~:0(<method 'update' of 'dict' objects>)
2000	0.001572	7.86e-07	0.001572	7.86e-07	~:0(<method 'update' of 'collections.OrderedDict' objects>)
3	1.1e-05	3.667e-06	1.1e-05	3.667e-06	~:0(<method 'union' of 'set' objects>)
1	3e-06	3e-06	3e-06	3e-06	~:0(<method 'union' of 'frozenset' objects>)
2000	0.001103	5.515e-07	0.001103	5.515e-07	~:0(<method 'truncate' of '_io.StringIO' objects>)
32	2.1e-05	6.562e-07	2.1e-05	6.562e-07	~:0(<method 'translate' of 'str' objects>)
356	0.000169	4.747e-07	0.000169	4.747e-07	~:0(<method 'translate' of 'bytearray' objects>)
1	1e-06	1e-06	1e-06	1e-06	~:0(<method 'toordinal' of 'datetime.date' objects>)
4	1.2e-05	3e-06	1.2e-05	3e-06	~:0(<method 'tolist' of 'memoryview' objects>)

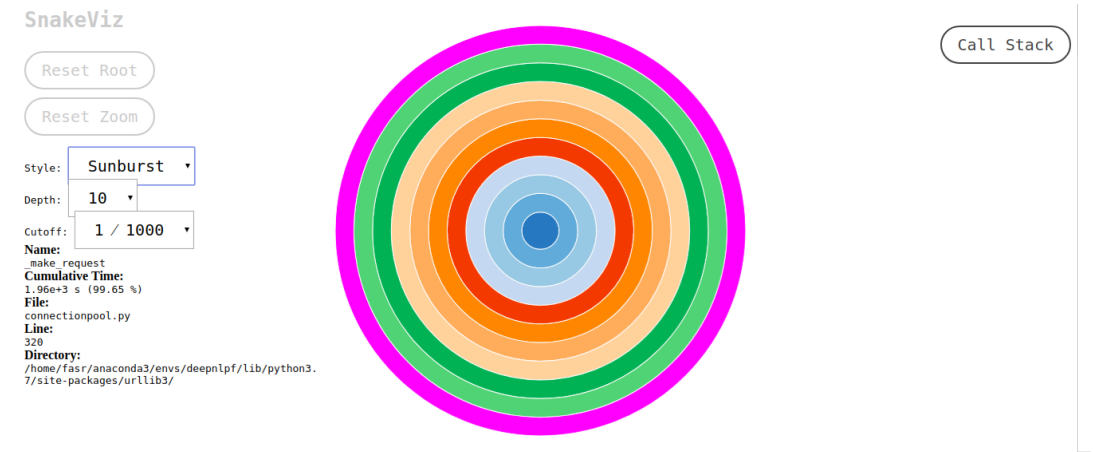
Fonte: do autor.

cias visíveis da mesma chamada de função. Também exibirá uma lista de informações à esquerda do gráfico, como:

- *Name:* nome da função;
- *Cumulative Time:* tempo acumulativo em segundos gasto na função e como uma porcentagem do tempo total de execução do programa;
- *File:* nome do arquivo no qual a função está definida;
- *Line:* número de linha na qual a função está definida;
- *Directory:* diretório do arquivo.

Nota: Para algumas funções internas, o nome do arquivo, o número da linha e o diretório serão ' ', 0 e em branco, respectivamente.

Figura 46 – SnakeViz *Profile* - Gráfico do Tipo *Sunburst* Exemplo Visão Geral.

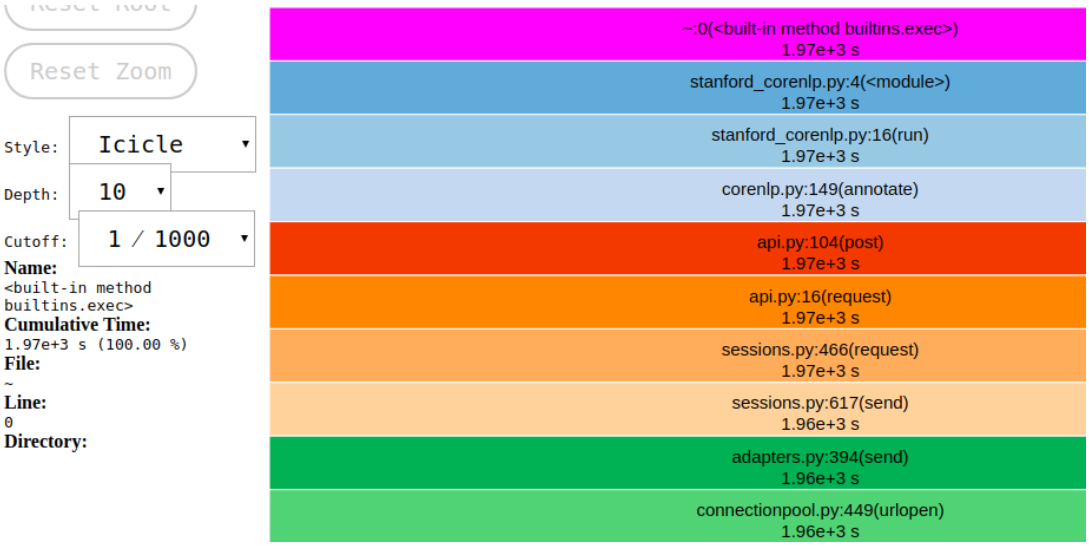


Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.02398	0.02398	0.02398	0.02398	~:0(<method 'writelines' of '_io._IOBase' objects>)
813	0.07099	8.732e-05	0.07099	8.732e-05	~:0(<method 'write' of '_io.TextIOWrapper' objects>)
1000	0.001402	1.402e-06	0.001402	1.402e-06	~:0(<method 'write' of '_io.StringIO' objects>)
733	0.00017	2.319e-07	0.00017	2.319e-07	~:0(<method 'values' of 'mappingproxy' objects>)
2003	0.00059	2.946e-07	0.00059	2.946e-07	~:0(<method 'values' of 'dict' objects>)
10000	0.00265	2.65e-07	0.00265	2.65e-07	~:0(<method 'values' of 'collections.OrderedDict' objects>)
6154	0.003033	4.929e-07	0.003033	4.929e-07	~:0(<method 'upper' of 'str' objects>)
3	4e-06	1.333e-06	4e-06	1.333e-06	~:0(<method 'update' of 'set' objects>)
3367	0.004339	1.289e-06	0.004386	1.303e-06	~:0(<method 'update' of 'dict' objects>)
2000	0.001572	7.86e-07	0.001572	7.86e-07	~:0(<method 'update' of 'collections.OrderedDict' objects>)
3	1.1e-05	3.667e-06	1.1e-05	3.667e-06	~:0(<method 'union' of 'set' objects>)
1	3e-06	3e-06	3e-06	3e-06	~:0(<method 'union' of 'frozenset' objects>)
2000	0.001103	5.515e-07	0.001103	5.515e-07	~:0(<method 'truncate' of '_io.StringIO' objects>)
32	2.1e-05	6.562e-07	2.1e-05	6.562e-07	~:0(<method 'translate' of 'str' objects>)
356	0.000169	4.747e-07	0.000169	4.747e-07	~:0(<method 'translate' of 'bytearray' objects>)
1	1e-06	1e-06	1e-06	1e-06	~:0(<method 'toordinal' of 'datetime.date' objects>)
4	1.2e-05	3e-06	1.2e-05	3e-06	~:0(<method 'tolist' of 'memoryview' objects>)

Fonte: do autor.

Figura 47 – SnakeViz *Profile* - Informação da Função.



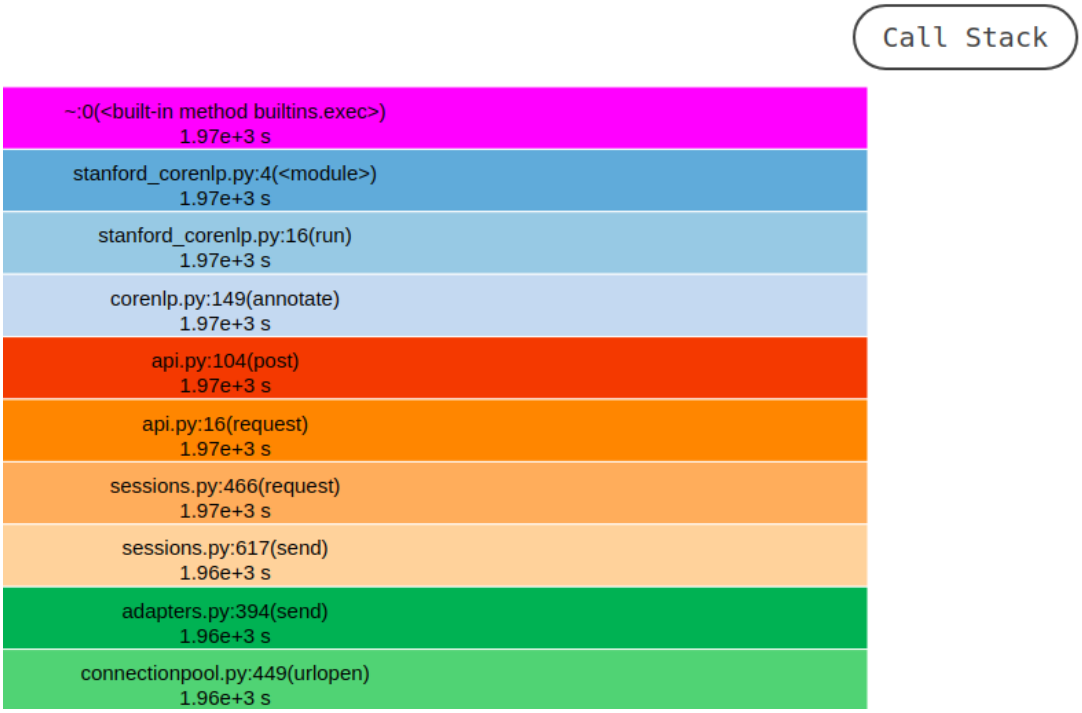
Fonte: do autor.

Ao clicar-se em uma função ampliará a visualização, fazendo com que essa função seja a nova raiz e permitindo que possa ampliar diferentes partes do *Profile*, como exibida a Figura 48.

Outra opção interessante que o *Snakeviz* fornece é a tabela de estatísticas na Figura 49 que mostra os dados de *Profiling*. A Tabela exibe uma linha exclusiva por chamada de função, onde:

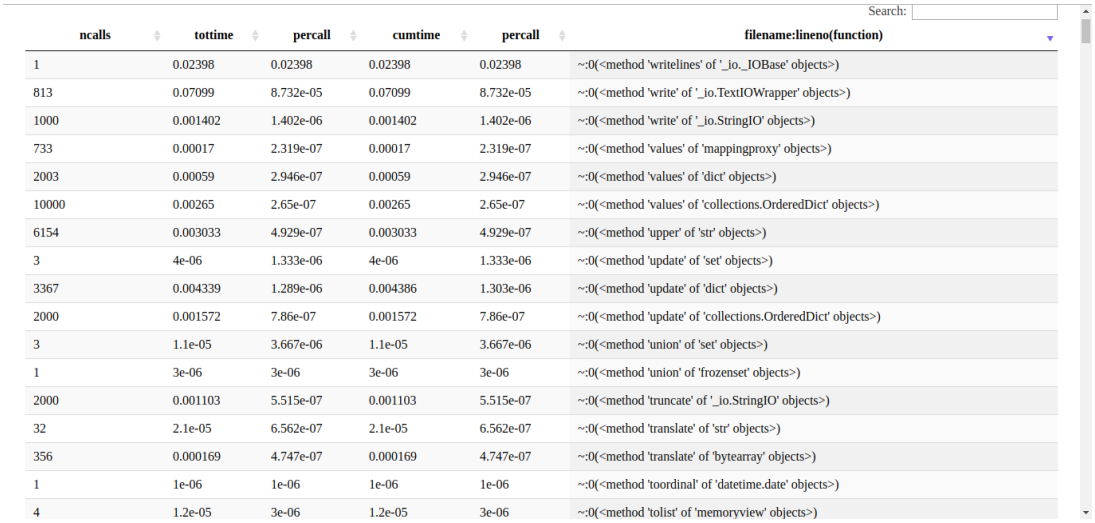
- *ncalls:* número de chamadas para a função. Se houver dois números, isso significa que a função é recursiva onde a primeira é o número total de chamadas, e a segunda é o número de chamadas primitivas (não recursivas);
- *tottime:* tempo total gasto na função, não incluindo o tempo gasto em chamadas para subfunções;
- *percall: tottime* dividido por *ncalls*;
- *cumtime:* tempo acumulativo gasto nesta função e todas as subfunções;
- *percall: cumtime* dividido por *ncalls*;
- *filename:lineno(function):* nome do arquivo e número da linha onde a função está definida e o nome da função.

Figura 48 – SnakeViz *Profile* - *Call Stack*.



Fonte: do autor.

Figura 49 – SnakeViz *Profile - Tabela de Estatísticas.*



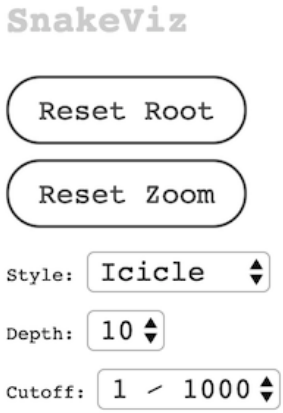
Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.02398	0.02398	0.02398	0.02398	~:0(<method 'writelines' of '_io._IOBase' objects>)
813	0.07099	8.732e-05	0.07099	8.732e-05	~:0(<method 'write' of '_io.TextIOWrapper' objects>)
1000	0.001402	1.402e-06	0.001402	1.402e-06	~:0(<method 'write' of '_io.StringIO' objects>)
733	0.00017	2.319e-07	0.00017	2.319e-07	~:0(<method 'values' of 'mappingproxy' objects>)
2003	0.00059	2.946e-07	0.00059	2.946e-07	~:0(<method 'values' of 'dict' objects>)
10000	0.00265	2.65e-07	0.00265	2.65e-07	~:0(<method 'values' of 'collections.OrderedDict' objects>)
6154	0.003033	4.929e-07	0.003033	4.929e-07	~:0(<method 'upper' of 'str' objects>)
3	4e-06	1.333e-06	4e-06	1.333e-06	~:0(<method 'update' of 'set' objects>)
3367	0.004339	1.289e-06	0.004386	1.303e-06	~:0(<method 'update' of 'dict' objects>)
2000	0.001572	7.86e-07	0.001572	7.86e-07	~:0(<method 'update' of 'collections.OrderedDict' objects>)
3	1.1e-05	3.667e-06	1.1e-05	3.667e-06	~:0(<method 'union' of 'set' objects>)
1	3e-06	3e-06	3e-06	3e-06	~:0(<method 'union' of 'frozenset' objects>)
2000	0.001103	5.515e-07	0.001103	5.515e-07	~:0(<method 'truncate' of '_io.StringIO' objects>)
32	2.1e-05	6.562e-07	2.1e-05	6.562e-07	~:0(<method 'translate' of 'str' objects>)
356	0.000169	4.747e-07	0.000169	4.747e-07	~:0(<method 'translate' of 'bytearray' objects>)
1	1e-06	1e-06	1e-06	1e-06	~:0(<method 'toordinal' of 'datetime.date' objects>)
4	1.2e-05	3e-06	1.2e-05	3e-06	~:0(<method 'tolist' of 'memoryview' objects>)

Fonte: do autor.

As colunas da tabela são classificáveis e a caixa de pesquisa pode ser usada para filtrar a tabela. Por fim tem-se os Controles onde afetam a visualização do documento, ver Figura 50.

Figura 50 – SnakeViz *Profile - Controles.*



Fonte: do autor.

- *Reset Zoom:* redefine a visualização para a função raiz selecionada.
- *Reset Root:* redefine a visualização para a função raiz principal.
- *Style:* alterna entre s estilos *icicle* e *sunburst*.
- *Depth:* controla a profundidade da pilha de chamadas, percorrendo a visualização.

- *Cutoff:* controla a exibição de funções que ocupam pouco tempo cumulativo.

Os gráficos *Icicle* e *Sunburst* apresentam, uma variedade de informações. Porem, a informação relevante para essa avaliação de desempenho é o tempo acumulativo, ou seja, o tempo total que a ferramenta levou para processar o *dataset*. Nesse exemplo mostrado pelas Figuras 48 e 49 informa o tempo (1.97e+3s equivalente à 1970s) que a ferramenta *Stanford CoreNLP* levou para executar um determinado *pipleine* de análises. Uma outra ferramenta "alternativa"que poderia ser utilizada para a visualização dos *Profiling* seria a [7]*KCachegrind*.

5.1.1 *Experimentos*

Nessa seção serão apresentados quatro experimentos que foram conduzidos com o objetivo de avaliar o tempo de desempenho das ferramentas de PLN de terceiros, consideradas aqui individualmente e, posteriormente, comparar os tempos de processamento quando essas mesmas ferramentas são processadas apartir do *DeepNLPF*. Visa-se portanto, validar experimentalmente se a estrategia de processamento paralelo empregadas pelo *DeepNLPF* realmente traz uma melhoria de desempenho significativo. Tais experimentos irão então responder as perguntas experimentais PE1 e PE2.

Experimento I - PE1

Em um primeiro momento, foi realizada uma análise utilizando as ferramentas de PLN individualmente, utilizando os *corpus* I, II e III Tabela 11, anotando-se os tempos de execução para as analises linguísticas nos níveis Léxico, Sintático e Semântico.

Os experimentos foram conduzidos considerando-se os níveis da linguagem citados, tendo em vista que algumas análises são normalmente mais rápidas do que outras com base nos estágios do *pipeline*. Em outras palavras, algumas análises mais complexas levam mais tempo para processar que outras análises mais simples, devido a ao número de análises linguísticas em questão.

O objetivo desse experimento é verificar o tempo de processamento que cada ferramenta necessita para processar os três corpus utilizando os *pipelines* de análises selecionados a seguir.

1. Análise Léxica:

    - *Stanford CoreNLP*; *Pipeline: tokenization*, *pos*, *lemma*, *ner* e *truecase*.
    - *Spacy - Pipeline: pos*, *tag*, *shape*, *label*, *is_alpha*, *is_title* e *like_num*.
    - *CogComp - Pipeline: ner ontonotes*.

[7] *KCachegrind* é uma ferramenta para criação e visualização de *Profiling*. http://kcachegrind.sourceforge.net/html/Home.html

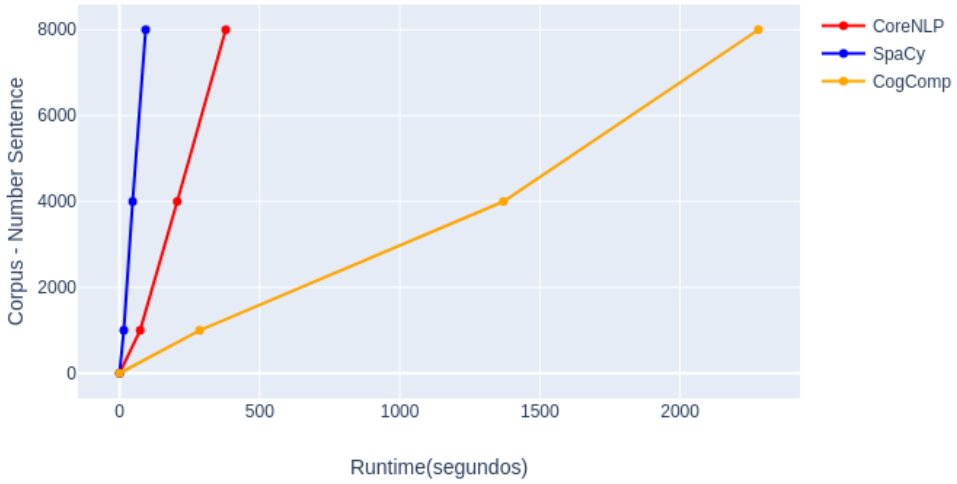
Tabela 11 – Características dos *Corpus* I, II e III.

SemEval 2010	**Sentenças**	**Descrição**
Corpus I	1000	*corpus* pequeno
Corpus II	4000	*corpus* médio
Corpus III	8000	*corpus* grande

Fonte: do autor.

A Figura 51 exibe o tempo de desempenho das ferramentas citadas. Pode-se notar que, dentre elas, a ferramenta que leva mais tempo de processamento é a *CogComp*. Entretanto, ela não foi executada localmente, e sim, *online*, requisitando os serviços diretamente ao servidor oficial da ferramenta via uma *API*. Sendo assim, espera-se que ela tenha um melhor desempenho quando executada localmente. A justificativa de não se ter executado o *CogComp* localmente é que ele exige muito de memória, devido ao tamanho dos [8]modelos carregados em memória *RAM*, o que impossibilitou a sua utilização local, devido aos recursos computacionais disponíveis serem insuficiente.

Figura 51 – Tempo de Processamento para Análise Léxica.



Fonte: do autor.

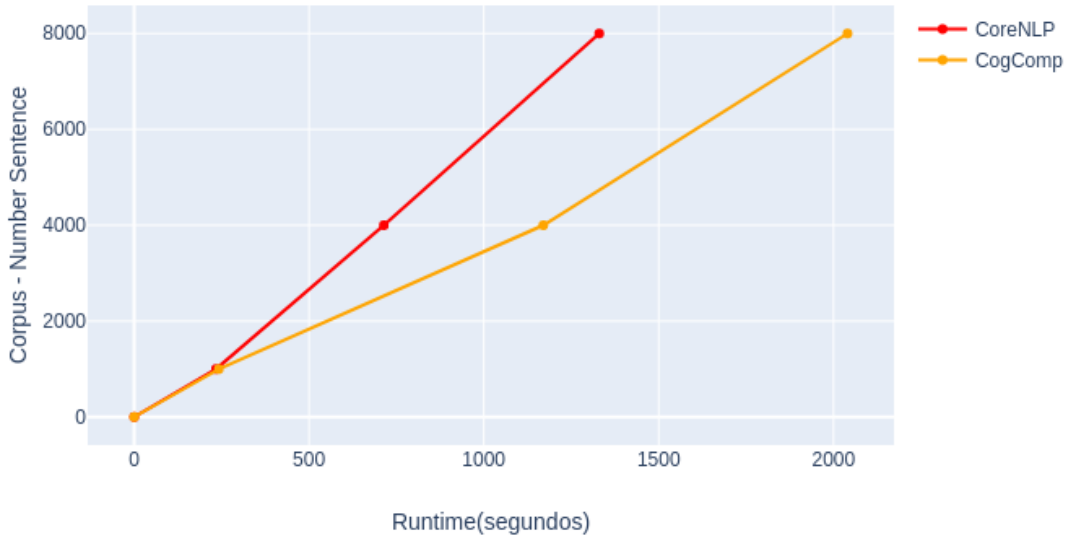
2. Análise Sintática:

   - *Stanford CoreNLP - Pipeline: parse, depparse* e *dcoref.*
   - *CogComp - Pipeline: shallow parse.*

   A Figura 52 exibe o tempo de processamento das ferramentas a ferramenta *CogComp* continua sendo a mais demorada, sendo necessários mais de 2000 segundos para analisar as 8000 sentenças.

[8] https://github.com/CogComp/cogcomp-nlp/tree/master/pipeline

Figura 52 – Tempo de Processamento para Análise Sintática.



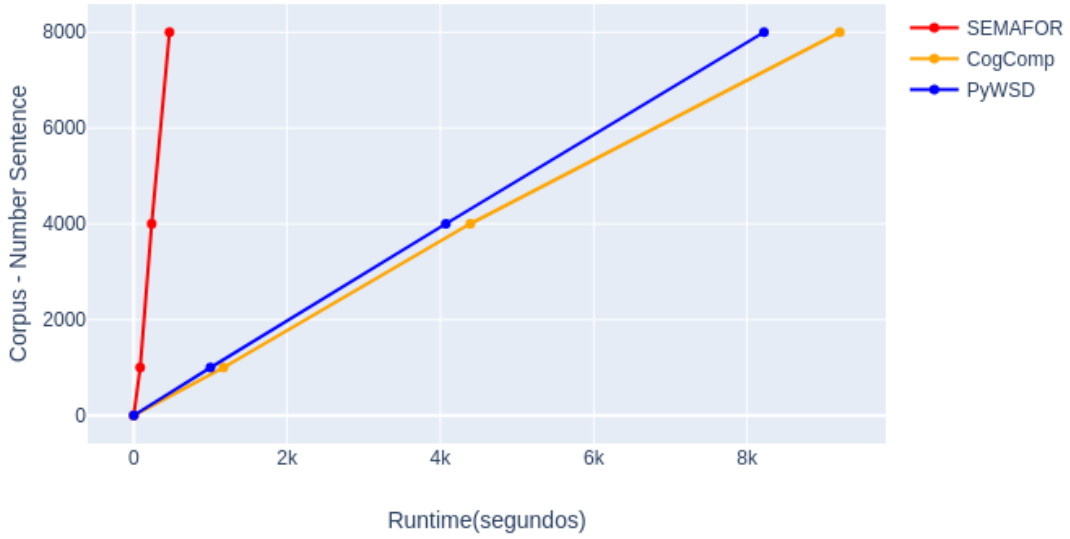
Fonte: do autor.

3. Anaĺise Semântica:

   - *SEMAFOR*: *frame parsing semantic*.
   - *CogComp*: *SRL Nom*, *SRL Verb* e *SRL Prep*.
   - *PyWSD*: *WSD*.

   A Figura 53 exibe o gráfico de linha com o desempenho para as ferramentas de PLN aplicadas. Dentre elas, as ferramentas *CogComp* e *PyWSD* são as que levam mais tempo para processar todas as sentenças. Entretanto, essa duas ferramentas foram executadas *online*, o que pode ter um melhor desempenho caso executadas localmente. Como citado anteriormente, para isso, exige-se muito recurso computacional, devido ao tamanho dos modelos carregados em memória, sendo assim, não foi possível utilizá-las localmente até o momento devido a essa limitação.

Figura 53 – Tempo de Processamento para Análise Semântica.



Fonte: do autor.

Experimento II - PE1

Em um segundo momento, cada ferramenta de PLN foi executada individualmente, uma após a outra, processando os *corpus* I, II e III. Em cada execução, foi verificado o tempo (em segundos) de processamento que levou para cada ferramenta processar todas as sentenças do *corpus*, entretanto, dessa vez foi utilizado o *pipeline* de análises completo (análise selecionadas nos níveis léxico, sintático e semântico). Com os resultados de cada execução, foram feitos os somatórios dos tempos de cada ferramenta, com o objetivo de saber quanto tempo levaria para processar cada *corpus* utilizando as ferramentas individualmente de forma sequencial (em série).

Para o *Corpus* I com 1000 sentenças, todas as ferramentas de forma sequencial gastariam em torno de 2439.6 segundos para processar todas as sentenças do *Corpus* I. Exemplo: TempoTotal = *StanfordCoreNLP* + *SpaCy* + *SEMAFOR* + *CogComp* + *SupWSD*. Para o *corpus II* com 4000 sentenças, essas ferramentas executadas de forma sequencial gastaram em torno de 9935.3 segundos para processar todas as sentenças. Por fim, para o *corpus III* com 8000 sentenças, as mesmas ferramentas tomaram cerca de 18879 segundos para processar todas as sentenças. A Tabela 12 sumariza tais resultados.

Com base na proporcionalidade entre as grandezas, pode-se avaliar um ponto de vista comparativo entre duas grandezas e extrair resultados. Sendo assim, partindo da quantidade de sentenças nos (*Corpus* I, II e III) processados em relação ao tempo de execução das ferramentas, existe uma igualdade entre as razões. Isso quer dizer, que a medida que uma grandeza aumenta, varia na mesma taxa de proporção a outra. Portanto, conforme os valores totais mostrados na Tabela 12 o tempo de processamento gasto para analisar os *corpus* I, II e II são proporcionais, onde a quantidade de sentenças presentes em cada

Tabela 12 – Tempo de Execução Individual de Cada Ferramenta de PLN.

TOOLS	PIPELINE	CORPUS I	CORPUS II	CORPUS III
Stanford CoreNLP	tokenize, ssplit, pos, lemma, ner, parse, depparse, truecase, dcoref	270	776	1630
SpaCy	pos, tag, shape, label, is_alpha, is_title, like_num	12.3	46.3	91.2
SEMAFOR	frame parsing semantic	87.3	243	448
CogComp	srl nom, srl verb, srl prep, shallow parse, ner ontonotes	1050	5110	9350
PyWSD	wsd	1020	3760	7360
RUNTIME		2439.6	9935.3 (+40,726%)	18879.2 (+77,386%)

Fonte: do autor.

um deles influencia proporcionalmente no tempo de execução.

Experimento III - PE2

Nesse experimento, foram processados os *corpus* I, II e III, utilizando as ferramentas de PLN externas a partir do *DeepNLPF*, com os mesmos *pipelines* de análises do Experimento I, a fim de se comparar os tempos de desempenho das ferramentas de PLN externas quando utilizadas normalmente (Experimento I) em relação quando utilizadas a partir do *DeepNLPF* (Experimento II).

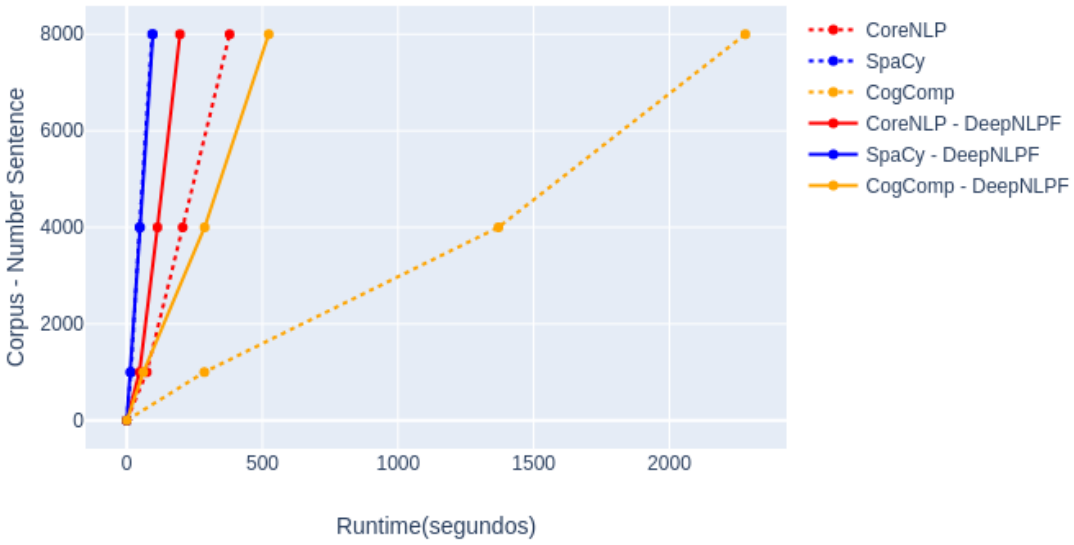
1. Análise Léxica:

   - *Stanford CoreNLP*: *tokenization*, *pos*, *lemma* e *ner*.
   - *Spacy*: *pos*, *tag*, *shape*, *label*, *is_alpha*, *is_title* e *like_num*.
   - *CogComp*: *ner ontonotes*.

   A Figura 54 mostra o gráfico de linha contendo as análises de tempo de processamento das ferramentas de PLN. As linhas pontilhadas representam as ferramentas de PLN processadas individualmente e as linhas continuas, representam as mesmas feramentas de PLN executadas individualmente, porem, a partir do *DeepNLPF* aplicando as estrategias de otimização.

No gráfico, pode ser notado que para a ferramenta *SpaCy* praticamente tem uma sobreposição das linhas, o que indica que a estrategia praticamente não afetou (melhorou) nada no desempenho dessa ferramenta. Isso significa que o *SpaCy* tem um desempenho relevante, e que mesmo utilizando estrategia de *multithreads* não é o suficiente para otimizar. Já para a ferramenta de PLN *CoreNLP* é possível observar uma melhoria significativa no desempenho quando utilizada a parti do *DeepNLPF*. Analisando os resultados da ferramenta *CogComp*, que está sendo utilizada na sua versão *online*, ouve uma melhoria muito notória, o que significa que as estrategias empregadas pelo *DeepNLPF* aumenta o desempenho da tarefa executada, tal desempenho foi tão relevante, que os valores de tempo apurados, se aproximam com os tempos das demais ferramentas executadas localmente.

Figura 54 – Comparação do Tempo de Processamento para Análise Léxica.

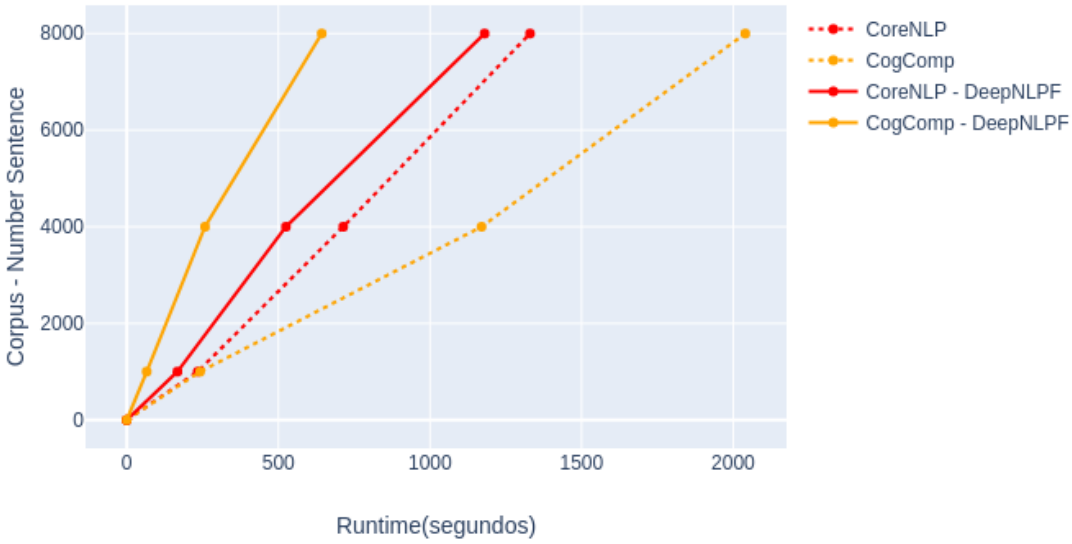


Fonte: do autor.

2. Análise Sintática:
   - *Stanford CoreNLP*: *parse*, *depparse* e *dcoref*.
   - *CogComp*: *shallow parse*.
3. Anaĺise Semântica:
   - *SEMAFOR*: *frame parsing semantic*.
   - *CogComp*: *SRL Nom*, *SRL Verb* e *SRL Prep*.
   - *PyWSD*: *WSD*.

Na Figura 56, exibe-se o gráfico de desempenho da ferramentas *SEMAFOR*, *CogComp* e *PyWSD*. Para a ferramenta *SEMAFOR*, não ouve uma diferença significativa quanto a desempenho, pois é possível visualizar que uma reta sobrepõe a outra.

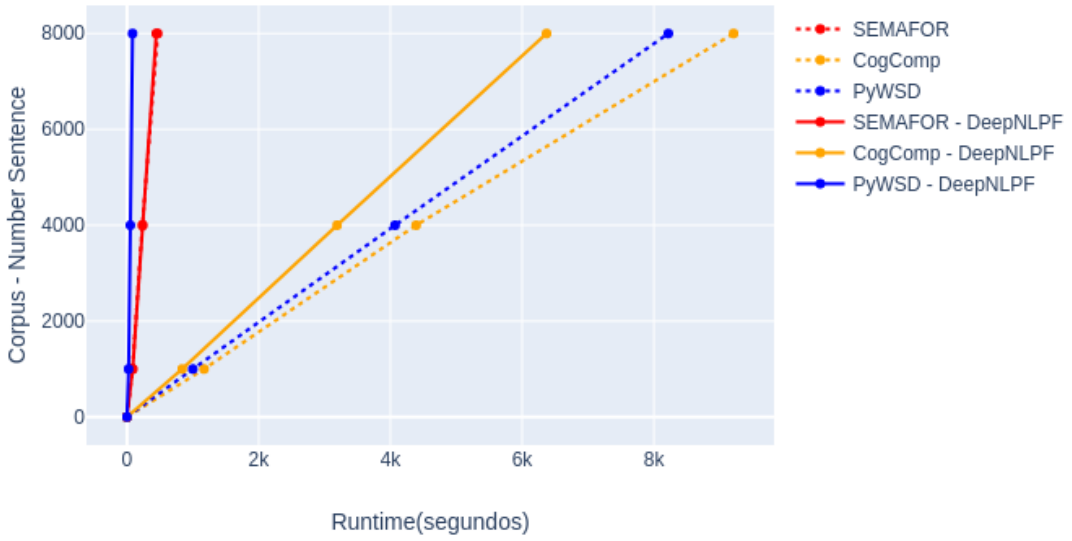
Figura 55 – Comparação do Tempo de Processamento para Análise Sintática.



Fonte: do autor.

Para a ferramenta *CogComp* ouve uma diferença de aproximadamente 2500s, o que é um resultado favorável para o *DeepNLPF*. E por fim, para a ferramenta *PyWSD*, a diferença de tempo de processamento é marcante, obtendo o melhor tempo entre todas as demais ferramentas. Esses resultados apontam que as estrategias propostas pelo *DeepNLPF* melhora consideradamente o desempenho.

Figura 56 – Comparação do Tempo de Processamento para Análise Semântica.



Fonte: do autor.

Experimento IV - PE2

Nesse quarto momento, esse experimento foi realizado seguindo os mesmos critérios anteriores, usando todas as ferramentas de PLN citadas, com os mesmo *pipelines* de análises linguísticas, porém executando-as a partir do *DeepNLPF*, integradas a ele por meio de *plugins*. Esse experimento tem o objetivo de verificar se as integrações das ferramentas externas funcionam corretamente através dos *plugins* e se as estratégias de processamento do *framework* garantem se obter um tempo de processamento reduzido. Ou seja, o objetivo é verificar o tempo de processamento do *framework* e comparar com os resultados de tempos obtidos pelas ferramentas usadas na comparação.

Após obtidos todos os tempos de processamentos dos *corpus* I, II e III, dos dois experimentos, (II) processamento sequencial das ferramentas e (III) processamento paralelo das ferramentas apartir do *DeepNLPF*, obtém-se o gráfico de linha onde é possível visualizar o tempo dos experimento com relação a quantidade de sentenças processadas Figura 57. É notório o quanto o *DeepNLPF* foi mais eficiente, praticamente reduzindo o tempo de processamento em 60%. O *DeepNLPF* processou 1000 sentenças (*corpus* I) em 824s, 4000 sentenças (*corpus* II) em 3250s. e 8000 sentenças (corpus III) em 6320s. Enquanto as ferramentas executadas sequencialmente processaram 1000 sentenças em 2439.6s, 4000 sentenças em 9935.3s e 8000s em 18879.2s .

Um fato que foi observado durante o experimento com o *DeepNLPF*, é que o tempo de processamento total do *framework* é definido pela ferramenta que leva mais tempo para processar as sentenças do *corpus*, o que era de se esperar. O que acontece é que, enquanto outros processos já tenham concluídos, apenas um único processo continua a trabalhar, e os demais núcleos do computador permanecem ociosos, o que não é desejável. Pois tais núcleos que ficaram ociosos poderiam ser utilizados para processar os restantes das análises e concluir com mais rapidez. Sendo assim, o que poderia ser ser feito para melhorar isso, é uma nova implantação de estratégias de processamento mais inteligente, que possibilite a redistribuição de processos ou até mesmo *threads* em tempos de execução.

Figura 57 – Comparação do Tempo do *DeepNLPF* com o Processamento Individual das Ferramentas.

Fonte: do autor.

5.2 ANÁLISE QUALITATIVA - PE3

Avalia-se o *DeepNLPF* em comparação com os trabalhos relacionados no que tange à presença (+) e ausência (-) das características elencadas no Capítulo 3. Na Tabela **??** apresenta-se um resumo de tal análise comparativa.

Com relação à característica **Domínio**, a presente proposta apresenta um artefato que pode ser aplicado em qualquer domínio, devido a sua abordagem generalizada, empregando recursos customizados para se adequar ou ser adequado a qualquer domínio (médico, linguístico, entre outros) que utilizam PLN.

Para atender à característica **Análises Linguísticas** nos níveis Léxico, Sintático, Semântico como também pré-processamento do texto, este trabalho engloba análises que cobrem todos os três níveis citados. Assim, o usuário pode realizar as análises linguísticas mais convenientes dentro dos níveis de linguagem de sua escolha.

Para atender à característica **Ferramentas Externas**, este trabalho dá suporte ao usuário na utilização das ferramentas mais populares no meio do PLN. Essa seleção de ferramentas foi realizada com base nos trabalhos relacionados a essa proposta, que levantou ferramentas de PLN construídas por diversas comunidades e grupos de mineração de texto.

Para a característica ***Pipeline* Customizado**, a presente proposta propõem um modelo de *Plugins* que fornece ao usuário a opção de adicionar novos recursos de análises

linguísticas e personalizar seu *pipeline*, sendo assim, o usuário pode escolher quais as análises linguísticas ele deseja utilizar e de qual ferramenta será utilizada. Alem disso, o usuário pode montar o *pipeline* mais específico para anotar o *corpus*, com as análises mais robustas possíveis, o que pode melhorar a qualidade das informações geradas sobre os dados aplicados.

No que se refere às características ***API*** e ***RESTFul***, esta proposta cobre esses quesitos dispondo uma *interface* de programação de aplicação seja ele *desktop* ou *web*, consumindo os recursos por meio de serviços, com a mesma simplicidade que a linguagem *Python* fornece.

Para atender à característica **GUI**, a presente proposta implementa um protótipo de uma aplicação *DashBoard* que permite a interação do usuário com os recursos do *DeepNLPF* por meio de elementos gráficos como, botões, ícones e indicadores visuais. Dessa forma, "usuários leigos"podem experimentar a ferramenta sem a necessidade de um conhecimento profundo a respeito de programação ou das tecnologias implementadas no *DeepNLPF*, e até mesmo utilizá-la para fins didáticos.

No que se refere à característica **Estrategia de Processamento**, esta proposta apresenta uma estratégia de divisão de atividades por processos ou nós (*cluster* - compartilhamento de atividade entre vários computadores), visando aproveitar todo o poder computacional disponível, consequentemente otimizar o tempo de espera do processamento do *corpus* analisado. Isto mostra o quanto a solução é robusta para cenários mais complexos, para processamento de *corpus* extensos por exemplo.

Para atender à característica **Banco de Dados**, este trabalho propõe uma forma de armazenamento de informações dos *corpus*, análises, anotações, estatísticas dos *corpus* e *logs* do sistema. Com isso, a implementação desta solução em um aplicativo real consome menos recursos de processamento, uma vez que diminui o tempo de acesso entrada e saída das informações, comparado ao uso de arquivos de texto.

Para a característica **Estatística de *Corpus***, esta proposta entende que é importante fornecer ao usuário uma forma de visualizar os dados no contexto geral e estatístico. Isso permite que usuário compreenda melhor os dados sob análise.

No que se refere às características de **Arquitetura/Padrões de Projetos** e **Requisitos do Sistema**, esta proposta teve o cuidado de utilizar as boas práticas e padrões já bem estabelecidos na engenharia de *software*, o que permitiu visualizar os problemas correntes e tomar as decisões escolhendo as soluções já empregadas na literatura. Isso reflete na robustez e flexibilidade da ferramenta, inferido diretamente na qualidade de código e reuso.

Para a característica **Forma de Dados**, é proposto pela solução, os formatos *XML* e *JSON*, que são os mais utilizados na literatura. Isso é importante pois possibilita que outros aplicativos possam consumir os resultados diretamente, sem a necessidade de realizar conversões de formatos de arquivos.

No que tange à característica **Licença**, a licença adotada foi (MIT), para que outros pesquisadores e desenvolvedores possam utilizar, modificar e aperfeiçoar o trabalho desenvolvido.

Por fim, não menos importante, para atender ao critério **Documentação**, essa proposta entende que esse quesito faz parte de todo *software* e que impacta na qualidade e credibilidade do que foi construído. Então, é fornecido um *userguide* para auxiliar o usuário a instalar e utilizar a ferramenta, visando estabelecer uma comunicação clara, histórico de versão [9] *(Changelogs)* e também evitar que o usuário desperdice tempo ou até mesmo desistam de utilizar.

No que se refere as perguntas PE1, PE2 e PE3 realizadas no inicio do capítulo todas elas foram respondidas positivamente. Foram obtidos os tempos de desempenhos das ferramentas de PLN externas e comparados com o tempo de desempenho do *DeepNLPF*, a qual obteve um melhor desempenho, aproximadamente 60% mais rápido. No que se refere ao *framework*, por meio das características expostas anteriormente, a proposta apresentada abrange todas elas, possibilitando a integração das ferramentas de PLN de terceiros e permitindo a customização dos *pipelines* de análises linguísticas. A presente proposta em relação aos trabalhos relacionados cobre todos os critérios citados. Entretanto, quando se considera outros critérios, tais como: suporte a multilinguagens, representação de entidades e ontologias ou base de conhecimento, o *DeepNLPF* não as satisfazem. Por outro lado, ele enfoca principalmente a integração e customização de *pipelines*, como também no tempo de processamento, sendo estas suas principais vantagens.

[9] *Changelogs*: Registro de alterações ou registo de alterações, em computação, corresponde a uma lista contendo o registro de todas alterações realizadas em um sistema, ambiente ou qualquer outro elemento.

Table 5.4 Análise Comparativa entre os trabalhos relacionados e a proposta.

TRABALHOS		**GeoTxt**	**CLAMP**	**Jigg**	**xTAS**	**GATE**	**FreeLing**	**DeepNLPF**
DOMÍNIO		INDEP.	DEP.	INDEP.	INDEP.	INDEP.	INDEP.	INDEP.
ANÁLISES	**Pré-process.**	*Stopwords*	-	*tokenization*, *SSPlit*	*tokenization*, *stemming*	*Tokenizer*, *Sent. Splitter*, *Lang. Identier*	*Tokenization*, *Sent. Splitter*, *Lang. Identier*,	*tokenize*, *ssplit*
	Léxica	NER	NER, POS	NER, POS	POS, NER	*POS Tagger*, *NER*	*Num detection*, *Data detection*, *NER*, *POS tagging*	*pos*, *lemma*, *ner*, *truecase*, *dcoref*, *is alpha*, *is iitle*, *like num*, *shape*, *ner ontonotes*,
	Sintática	-	*Chunker*, *Parsing*	Parsing	-	*Chunker*, *Parser*	*BIO*, *Shallow Parsing*, *Dep. parsing*	*parse*, *depparse*, *shallow parse*,
	Semântica	-	-	-	-	*WSD*, *Tagger*, *Gazetter*	*WN WSD*, *UKB WSD*	*Frame Passing*, *SRL Verb*, *SRL Prep*, *SRL Nom*
FERRAMENTAS EXTERNAS		*CogComp*, *CoreNLP*, *GATE*, *ANNIE*, MIT IE, *OpenNLP*, *LingPipe*.	*OpenNLP*	*CoreNLP*, *Berkeley*	*CoreNLP*, *SEMAFOR*, *OpenNLP*	*CoreNLP*, *LingPipe*, *OpenNLP*	-	*CoreNLP*, *SEMAFOR*, *PyWSD*, *SpaCy*, *CogComp*
PIPELINE CUSTOMIZADO		-	+	+	+	+	-	+
API		-	-	+	+	+	+	+
API RESTFul		+	-	-	+	+	-	+
GUI		+	+	-	-	+	-	+
ESTRAT. PROCESSAMENTO		-	+	-	+	+	+	+
BANCO DE DADOS		+	-	-	+	+	-	+
ESTATÍSTICAS		-	-	-	-	-	-	+
ARQUITETURA		Cliente/Servidor	*Eclipse*	Base em *CoreNLP*	Cliente/Servidor, *Plugin*	Baseado em Componentes	Baseado em Cliente/Servidor	Baseado em Componentes
REQUISITOS		Java	Java	Java	*Python*, *Java*	Java	C/C++	*Python*
FORMATO DE DADOS		JSON	Texto	XML	JSON	TEXT, JSON, XML	CoNLL, TEXT, JSON, XML	JSON, XML
LICENÇA		GNU	Academic Free	Apache 2.0	Apache 2.0	Apache 2.0	GNU	MIT
DOCUMENTAÇÃO		+	+	+	+	+	+	+

Fonte: do Autor.

5.3 RESUMO DO CAPÍTULO

Este capítulo apresentou a avaliação experimental e comparativa do *DeepNLPF*, que leva em consideração diversos aspectos que um sistema de PLN deve conter para integração e customização de *pipeline* de PLN. Neste sentido, a arquitetura baseada em componentes utilizada no *framework* prototipado possui evidências que suporta a integração de ferramentas de PLN de terceiros, tornando possível a customização de *pipelines* customizados com análises proveniente de ferramentas diferentes.

Para avaliar o desempenho do processamento do *pipeline* customizado composto por cinco ferramentas de PLN de terceiros, realizou-se três cenários de testes que comparavam o tempo de execução do processamento de dados nos níveis de análises da linguagem (Léxico, Sintático e Semântico). Como resultado, obteve-se um desempenho de 60% de melhoria no tempo de processamento das ferramentas integradas ao *DeepNLPF* em relação a execução dessas ferramentas de forma individual e sequencial. Esse resultado expressa a otimização obtida, possibilitando que computadores simples como *notebook* e *desktop* possam usufruir de tais ferramenta sem a necessidade de dispor de recursos computacionais mais sofisticados como [10]*High Performance Computing (HPC)*, processando corpus demasiadamente longos em pouco tempo.

Para a avaliação da ferramenta no que tange as características que uma ferramenta de PLN de integração e customização de *pipeline* deve conter em relação aos trabalhos relacionados, o presente protótipo apresenta: suporte a integração de ferramentas de PLN de terceiros permitindo a customização do *pipeline* de análises linguística. As *APIs* providas (*API RESTFul*) pela proposta para que o usuário possa construir aplicações de PLN usufruindo recurso diverso com o mínimo de configurações possíveis; uma *interface* gráfica *dashboad* que o usuário leigos possam utilizar sem a necessidade de escrever código; estrategias de otimização que visão diminuir o tempo de espera para gerar anotações de *corpus* em computadores de mesa; um sistema de gerenciamento de banco de dados que possibilita a otimização do processamento, armazenamento dos dados processados, consultas e geração de relatórios; suporte a geração de estatística de dados textuais que ajudam ao usuário interpretar os dados e obter informações relevantes sobre o *corpus* em análise; arquitetura com componentes distribuídos independente, possibilitando atualização e manutenção flexível; saída de dados no formato *JSON* e *XML* adotando anotação de hibrida por camadas linguística e por fim uma documentação para suporte ao usuário.

[10] *High Performance Computing - HPC* temo em inglês que significa: Computação de Alto Desempenho, se refere ao uso de supercomputadores ou *clusters* de vários computadores em tarefas que requerem grandes recursos de computação.

6 CONCLUSÃO

Neste capítulo apresenta-se as conclusões do trabalho desenvolvido, suas principais contribuições, limitações e sugestões de trabalhos futuros.

A motivação deste parte do entendimento que algumas tarefas de mineração de texto necessitam combinar análises linguísticas para obter resultado proveitoso. Sendo assim, construir *pipelines* para análises linguística provenientes de ferramentas de PLN de terceiros é complexo e trabalhoso. Como também, execução de *pipeline* com muitas análises consome muito tempo e recurso computacional. Por esses motivos, um *framework* para customização de *pipeline* é proposto nesse trabalho e demonstrou ser uma alternativa com potencial para resolver os problemas mencionados acima. As ferramentas aos quais foram comparados ao *DeepNLPF* são muito apropriados para o propósito pretendido, e muitos são bem suportados por uma comunidade de programação. No entanto, todos os trabalhos relacionados a essa proposta contêm limitações seja na integração, no processamento, ou na visualização de seus resultados. O *DeepNLPF* consiste em um *framaework* que suporta o gerenciamento de uma diversidade de ferramentas de PLN, o armazenamento e a unificação das saídas produzidas, bem como a visualização dos dados.

Nesse contexto, este trabalho propôs um *framework* para integração e customização de *pipelines* de PLN com diversas análises linguística em três níveis da linguagem, capaz de integrar novas ferramentas de PLN. Este desenvolvimento foi orientado por requisitos elencados para processamento de dados textuais. Diante dos requisitos, foi desenvolvido um protótipo da ferramenta. Composto por diversos componentes responsáveis por tarefas especificas para cada parte do processo, dentre eles: componente para gerenciamento de arquivos e persistência em banco de dados; componente para integração de novas ferramentas de PLN ao *pipeline*; componente de *pipeline* para processamento das análises linguísticas e gerenciamento das atividades de processamento utilizando estrategias de desempenho de paralelismo e *multithreads*; componente para integração e anotação das análises linguísticas em formatos de arquivos *JSON* e *XML*. A solução proposta, também apresentou uma *interface* gráfica para o usuário utilizar a ferramenta sem a necessidade de escrever códigos.

Além das avaliações de natureza mais qualitativas apresentadas, focamos igualmente em como verificar a integração e customização de *pipeline* com enfoque no tempo de processamento requerido para o processamento de *datasets* de documentos de vários tamanhos. Pelos resultados obtidos, pode-se constatar que utilizar o *DeepNLPF* acelera o desempenho de processamento em torno de 60% em relação a execução das ferramentas de PLN no forma padrão "sequencial".

6.1 CONTRIBUIÇÕES

Este trabalho apresenta as seguintes contribuições:

- *Wrapper* Python para utilização da ferramenta de PLN CogComp, SEMAFOR e SupWSD.
- Bibliotecas Python para estatística de dados textuais, notificações, execução de scripts (Java, R, Shell Script, C/C++), arquitetura de plugins.
- Um *Framework* para integração e customização de análises linguística e anotação de documentos.

6.2 LIMITAÇÕES

Foram identificadas algumas limitações na solução proposta, a saber:

- Não dá suporte outros idiomas.
- Erros na anotação podem ser introduzidos aos diversos *parsers* linguísticos integrados ao *framework*.
- Processamento de *datasets* extensos através do *Dashboard* pode ocasionar *timeout*, devido ao tempo de espera para processar todas as análises. Isso será considerado no futuro utilizando o [1]*Celery* para gerenciar as tarefas de segundo plano.
- Nem todas as variáveis foram avaliadas nos experimentos.

6.3 TRABALHO FUTURO

- Extração de *Features (Features Engineering)*: A partir das anotações linguísticas geradas pelo *DeepNLPF*, disponibilizar ao usuário várias formas de representação de [2]*features* ou atributos que são tipicamente usadas em diversas aplicações de Mineração de Textos, incluindo Classificação de Texto (*Text Classification*), Análise de Sentimento (*Sentiment Analysis*), Extração de Relações (*Relation Extraction*), etc.

[1] O *Celery* é uma fila de tarefas/fila de tarefas assíncrona baseada na passagem de mensagens distribuídas. Ele é focado na operação em tempo real, mas também suporta o agendamento. As unidades de execução, denominadas tarefas, são executadas simultaneamente em um ou mais servidores de trabalho usando multiprocessamento, *Eventlet* ou *gevent*. As tarefas podem ser executadas de forma assíncrona (em segundo plano) ou sincronicamente (aguarde até que estejam prontas). http://www.celeryproject.org/

[2] *Feature* é uma representação numérica de dados brutos que são formatados principalmente para serem usados por algorítimos de Aprendizagem de Máquina (AM) em IA. (ZHENG, 2018)

- Formatos de anotação linguística: Incrementar o suporte a outros formatos de anotação linguística mais expressivos, como [3]*FoLiA* combina a representação de inúmeras análises linguísticas com a declaração de tipos, conceitos e classes de entidades de forma hierárquica (GOMPEL, 2012), (GOMPEL; REYNAERT, 2013).

- Multi-idiomas: adicionar detecção de idioma automaticamente para disponibilizar somente as análises suportadas para a língua identificada.

- *Cluster*: Processamento Paralelo Distribuído: adaptar o *framework* proposto para que haja um melhor distribuição do processamento necessário entre diversos nós (*nodes*) de um *Cluster* computacional. Espera-se obter uma redução significativa no tempo de processamento, principalmente no tocante ao tratamento de *corpus* de grande tamanho.

- *Command-Line Interface* (CLI): um meio do usuário interagir com o *framework*, emitindo comandos para criar sua aplicação de PLN rapidamente utilizando o geradores de códigos, ou simplesmente executar análises passando argumentos.

- Otimização de memória *RAM*: analisar o uso dessa memória durante o processamento dos dados visando solucionar problemas como por exemplo [4]*Overflow*. Dessa forma, propor uma intervenção que gerencie esse recurso de tal maneira que não seja necessário utilizar memórias virtuais como *Swap* que possuem um baixo desempenho, o que pode impactar no desempenho do algorítimo.

- Estrutura de dados heterogenia: inserir uma camada de dados para facilitar a integração de análises linguísticas quanto as estruturas de dados usadas para representar anotações (SAMMONS et al., 2016).

[3] https://proycon.github.io/folia/

[4] *Overflow* pode ser traduzido como estouro. O significado geral é a condição que ocorre quando os dados resultantes da entrada ou do processamento exigem mais *bits* do que os que foram fornecidos no *hardware* ou no *software* para armazenar os dados. Outra definição é parte de um item de dados que não pode ser armazenada porque os dados excedem a capacidade da estrutura disponível.

REFERÊNCIAS

AKIN, A. A.; AKIN, M. D. Zemberek, an open source nlp framework for turkic languages. *Structure*, v. 10, p. 1–5, 2007.

BAKER, C. F.; FILLMORE, C. J.; LOWE, J. B. The berkeley framenet project. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS. *Proceedings of the 17th international conference on Computational linguistics-Volume 1*. [S.l.], 1998. p. 86–90.

BALDWIN, B.; DAYANIDHI, K. *Natural language processing with Java and LingPipe Cookbook*. [S.l.]: Packt Publishing Ltd, 2014.

BARRETO, C. G. Agregando frameworks de infra-estrutura em uma arquitetura baseada em componentes: Um estudo de caso no ambiente aulanet. *Rio de Janeiro*, 2006.

BENTIVOGLI, L.; FORNER, P.; MAGNINI, B.; PIANTA, E. Revising the wordnet domains hierarchy: semantics, coverage and balancing. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS. *Proceedings of the Workshop on Multilingual Linguistic Ressources*. [S.l.], 2004. p. 101–108.

BEYSOLOW, T. *Applied Natural Language Processing with Python: Implementing Machine Learning and Deep Learning Algorithms for Natural Language Processing*. [S.l.]: Apress, 2018.

BONTCHEVA, K.; TABLAN, V.; MAYNARD, D.; CUNNINGHAM, H. Evolving gate to meet new challenges in language engineering. *Natural Language Engineering*, Cambridge University Press, v. 10, n. 3-4, p. 349–373, 2004.

CARRERAS, X.; CHAO, I.; PADRÓ, L.; PADRÓ, M. Freeling: An open-source suite of language analyzers. In: *LREC*. [S.l.: s.n.], 2004. p. 239–242.

CHEN, D.; SCHNEIDER, N.; DAS, D.; SMITH, N. A. Semafor: Frame argument resolution with log-linear models. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS. *Proceedings of the 5th international workshop on semantic evaluation*. [S.l.], 2010. p. 264–267.

CHOI, J. D.; TETREAULT, J.; STENT, A. It depends: Dependency parser comparison using a web-based evaluation tool. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. [S.l.: s.n.], 2015. p. 387–396.

CUNNINGHAM, H.; MAYNARD, D.; BONTCHEVA, K.; TABLAN, V. Gate: an architecture for development of robust hlt applications. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS. *Proceedings of the 40th annual meeting on association for computational linguistics*. [S.l.], 2002. p. 168–175.

DIVITA, G.; CARTER, M. E.; TRAN, L.-T.; REDD, D.; ZENG, Q. T.; DUVALL, S.; SAMORE, M. H.; GUNDLAPALLI, A. V. v3nlp framework: tools to build applications for extracting concepts from clinical text. *eGEMs*, Ubiquity Press, v. 4, n. 3, 2016.

ECKART, K. *CLARIN-D User Guide - Resource annotations*. 2012. <https://media.dwds.de/clarin/userguide/text/annotation_aspects.xhtml>. Accessed: 2019-04-16.

FAYAD, M. E.; SCHMIDT, D. C.; JOHNSON, R. E. *Building application frameworks: object-oriented foundations of framework design*. [S.l.]: John Wiley & Sons, Inc., 1999.

FERREIRA, R.; FREITAS, F.; BRITO, P.; MELO, J.; LIMA, R.; COSTA, E. Retriblog: an architecture-centered framework for developing blog crawlers. *Expert Systems with Applications*, Elsevier, v. 40, n. 4, p. 1177–1195, 2013.

GOMPEL, M. van. *FoLiA: Format for Linguistic Annotation. Documentation*. [S.l.], 2012.

GOMPEL, M. van; REYNAERT, M. Folia: A practical xml format for linguistic annotation–a descriptive and comparative study. *Computational Linguistics in the Netherlands Journal*, v. 3, p. 63–81, 2013.

HARDENIYA, N.; PERKINS, J.; CHOPRA, D.; JOSHI, N.; MATHUR, I. *Natural Language Processing: Python and NLTK*. [S.l.]: Packt Publishing Ltd, 2016.

HODGES, A. *Alan Turing: the enigma*. New York: Simon and Schuster, 1983. ISBN 978-0-671-49207-6 978-0-671-52809-6.

HOY, M. B. Alexa, siri, cortana, and more: an introduction to voice assistants. *Medical reference services quarterly*, Taylor & Francis, v. 37, n. 1, p. 81–88, 2018.

IDE, N.; ROMARY, L. International standard for a linguistic annotation framework. *Natural language engineering*, Cambridge University Press, v. 10, n. 3-4, p. 211–225, 2004.

II, T. B. *Applied Natural Language Processing with Python Implementing Machine Learning and Deep Learning Algorithms for Natural Language Processing*. San Francisco, California, USA: Apress, 2018. 1–12 p. ISBN 978-1-4842-3733-5.

JAMES, K.; WIRE, N.; BRADFIELD, J. D.; MOORE, S. *Holy Bible*. [S.l.]: National Press, 1969.

JOHNSON, R. E. Frameworks=(components+ patterns). *Communications of the ACM*, Citeseer, v. 40, n. 10, p. 39–42, 1997.

KARIMZADEH, M.; HUANG, W.; BANERJEE, S.; WALLGRÜN, J. O.; HARDISTY, F.; PEZANOWSKI, S.; MITRA, P.; MACEACHREN, A. M. Geotxt: a web api to leverage place references in text. In: ACM. *Proceedings of the 7th workshop on geographic information retrieval*. [S.l.], 2013. p. 72–73.

KARIMZADEH, M.; PEZANOWSKI, S.; MACEACHREN, A. M.; WALLGRÜN, J. O. Geotxt: A scalable geoparsing system for unstructured text geolocation. *Transactions in GIS*, Wiley Online Library, v. 23, n. 1, p. 118–136, 2019.

KHASHABI, D.; SAMMONS, M.; ZHOU, B.; REDMAN, T.; CHRISTODOULOPOULOS, C.; SRIKUMAR, V.; RIZZOLO, N.; RATINOV, L.; LUO, G.; DO, Q. et al. Cogcompnlp: Your swiss army knife for nlp. In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC-2018)*. [S.l.: s.n.], 2018.

KRUCHTEN, P. B. The 4+ 1 view model of architecture. *IEEE software*, IEEE, v. 12, n. 6, p. 42–50, 1995.

LANARO, S. K. Q. N. D. G. *Advanced Python Programming*. [S.l.]: Packt Publishing, 2019. ISBN 9781838551216.

LIMA, R. J. d.; FREITAS, F. L. G. d. Ontoilper: an ontology-and inductive logic programming-based method to extract instances of entities and relations from texts. Universidade Federal de Pernambuco, 2014.

LOPER, E.; BIRD, S. Nltk: the natural language toolkit. *arXiv preprint cs/0205028*, 2002.

MALDONADO, J. C.; BRAGA, R. T. V.; GERMANO, F. S. R.; MASIERO, P. C. Padrões e frameworks de software. *Notas Didáticas, Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, ICMC/USP, São Paulo, SP, Brasil*, 2002.

MANNING, C.; SURDEANU, M.; BAUER, J.; FINKEL, J.; BETHARD, S.; MCCLOSKY, D. The stanford corenlp natural language processing toolkit. In: *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. [S.l.: s.n.], 2014. p. 55–60.

MASSE, M. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. [S.l.]: "O'Reilly Media, Inc.", 2011.

MATOS, E. E. da S.; SALOMÃO, M. M. M. Ludi: um framework para desambiguação lexical com base no enriquecimento da semântica de frames. *Revista Linguítica*, v. 12, n. 1, 2014.

MAYNARD, D.; LI, Y.; PETERS, W. *NLP Techniques for Term Extraction and Ontology Population.* 2008.

MCINTOSH, M. A. *Language and Its Development*. 2018. <https://brewminate.com/language-and-its-development/>. Accessed: 2019-04-16.

MCKERNS, M.; AIVAZIS, M. Pathos: a framework for heterogeneous computing. *See http://trac. mystic. cacr. caltech. edu/project/pathos*, 2010.

MCKERNS, M. M.; STRAND, L.; SULLIVAN, T.; FANG, A.; AIVAZIS, M. A. Building a framework for predictive science. *arXiv preprint arXiv:1202.1056*, 2012.

MILLER, G. A. Wordnet: a lexical database for english. *Communications of the ACM*, ACM, v. 38, n. 11, p. 39–41, 1995.

NOJI, H.; MIYAO, Y. Jigg: a framework for an easy natural language processing pipeline. *Proceedings of ACL-2016 System Demonstrations*, p. 103–108, 2016.

PADRÓ, L.; STANILOVSKY, E. Freeling 3.0: Towards wider multilinguality. In: *LREC2012*. [S.l.: s.n.], 2012.

PALACH, J. *Parallel Programming with Python*. [S.l.]: Packt Publishing Ltd, 2014.

PANG, B.; LEE, L. et al. Opinion mining and sentiment analysis. *Foundations and Trends® in Information Retrieval*, Now Publishers, Inc., v. 2, n. 1–2, p. 1–135, 2008.

PAPANDREA, S.; RAGANATO, A.; BOVI, C. D. Supwsd: A flexible toolkit for supervised word sense disambiguation. In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. [S.l.: s.n.], 2017. p. 103–108.

PUSTEJOVSKY, J.; STUBBS, A. *Natural Language Annotation for Machine Learning: A guide to corpus-building for applications*. [S.l.]: "O'Reilly Media, Inc.", 2012.

RAO, D.; MCMAHAN, B. *Natural Language Processing with PyTorch: Build Intelligent Language Applications Using Deep Learning*. [S.l.]: "O'Reilly Media, Inc.", 2019.

RINGGAARD, M.; GUPTA, R.; PEREIRA, F. C. Sling: A framework for frame semantic parsing. *arXiv preprint arXiv:1710.07032*, 2017.

ROOIJ, O. D.; VISHNEUSKI, A.; RIJKE, M. D. et al. xtas: Text analysis in a timely manner. In: CITESEER. *Dir*. [S.l.], 2012. v. 2012, p. 12th.

SAMMONS, M.; CHRISTODOULOPOULOS, C.; KORDJAMSHIDI, P.; KHASHABI, D.; SRIKUMAR, V.; ROTH, D. Edison: Feature extraction for nlp, simplified. In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*. [S.l.: s.n.], 2016. p. 4085–4092.

SCHOUTEN, K.; FRASINCAR, F.; DEKKER, R.; RIEZEBOS, M. Heracles: A framework for developing and evaluating text mining algorithms. *Expert Systems with Applications*, Elsevier, v. 127, p. 68–84, 2019.

SCHREIBER, M.; KRAFT, B.; ZÜNDORF, A. Nlp lean programming framework: Developing nlp applications more effectively. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*. [S.l.: s.n.], 2018. p. 1–5.

SCHULER, K. K. Verbnet: A broad-coverage, comprehensive verb lexicon. 2005.

SHVETS, A. *Dive Into DESIGN PATTERNS*. 2019. <https://sourcemaking.com/design-patterns-ebook>. Accessed: 2019-04-16.

SILVA, E. B. Vocabprofile: uma ferramenta linguístico-estatística para a aula de língua inglesa. *Domínios de Lingu@ gem*, v. 5, n. 1, p. 144–159, 2011.

SOMMERVILLE, I. Software engineering 9th edition. *ISBN-10137035152*, 2011.

SOYSAL, E.; WANG, J.; JIANG, M.; WU, Y.; PAKHOMOV, S.; LIU, H.; XU, H. Clamp–a toolkit for efficiently building customized clinical natural language processing pipelines. *Journal of the American Medical Informatics Association*, Oxford University Press, v. 25, n. 3, p. 331–336, 2017.

TAN, L. *Pywsd: Python Implementations of Word Sense Disambiguation (WSD) Technologies [software]*. 2014. Https://github.com/alvations/pywsd.

VIEIRA, R.; LIMA, V. L. Lingüística computacional: princípios e aplicações. In: SN. *Anais do XXI Congresso da SBC. I Jornada de Atualização em Inteligência Artificial*. [S.l.], 2001. v. 3, p. 47–86.

WEERASOORIYA, T.; PERERA, N.; LIYANAGE, S. A framework for automated corpus compilation for keyxtract: Twitter model. In: IEEE. *2017 Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer)*. [S.l.], 2017. p. 1–6.

YULE, G. *The study of language*. [S.l.]: Cambridge university press, 2016.

ZHENG, A. C. A. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. 1. ed. [S.l.]: O'Reilly Media, Inc., 2018. v. 1. ISBN 9781491953242.

ZHENG, K.; VYDISWARAN, V. V.; LIU, Y.; WANG, Y.; STUBBS, A.; UZUNER, Ö.; GURURAJ, A. E.; BAYER, S.; ABERDEEN, J.; RUMSHISKY, A. et al. Ease of adoption of clinical natural language processing software: an evaluation of five systems. *Journal of biomedical informatics*, Elsevier, v. 58, p. S189–S196, 2015.

APÊNDICE A – ALPHABETICAL LIST OF PART-OF-SPEECH TAGS USED IN THE PENN TREEBANK PROJECT.

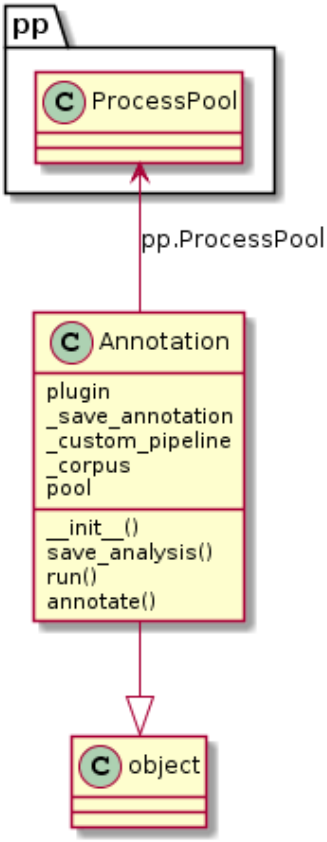
Tabela 13 – Alphabetical list of part-of-speech tags used in the Penn Treebank Project.

Tag	**Description**	**Tag**	**Description**
CC	Coordinating conjunction	PRP$	Possessive pronoun
CD	Cardinal number	RB	Adverb
DT	Determiner	RBR	Adverb, comparative
EX	Existential there	RBS	Adverb, superlative
FW	Foreign word	RP	Particle
IN	Preposition or subordinating conjunction	SYM	Symbol
JJ	Adjective	TO	to
JJR	Adjective, comparative	UH	Interjection
JJS	Adjective, superlative	VB	Verb, base form
LS	List item marker	VBD	Verb, past tense
MD	Modal	VBG	Verb, gerund or present participle
NN	Noun, singular or mass	VBN	Verb, past participle
NNS	Noun, plural	VBP	Verb, non-3rd person singular present
NNPS	Proper noun, plural	WDT	Wh-determiner
PDT	Predeterminer	WP	Wh-pronoun
POS	Possessive ending	WP$	Possessive wh-pronoun
PRP	Personal pronoun	WRB	Wh-adverb

Fonte: do Autor.

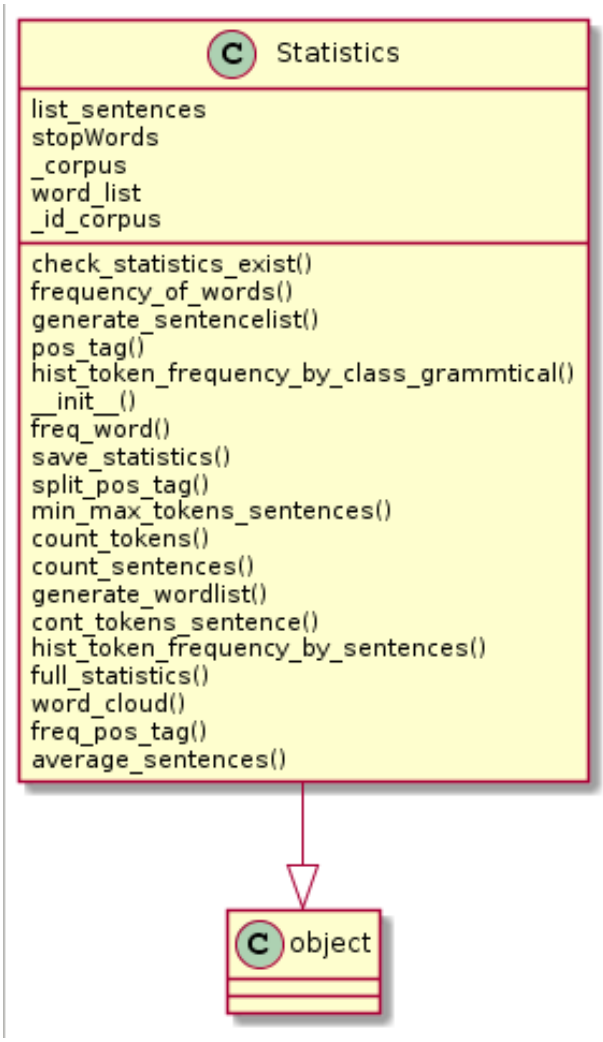
APÊNDICE B – DIAGRAMAS

Figura 58 – *Pipeline* - Diagrama de Classe



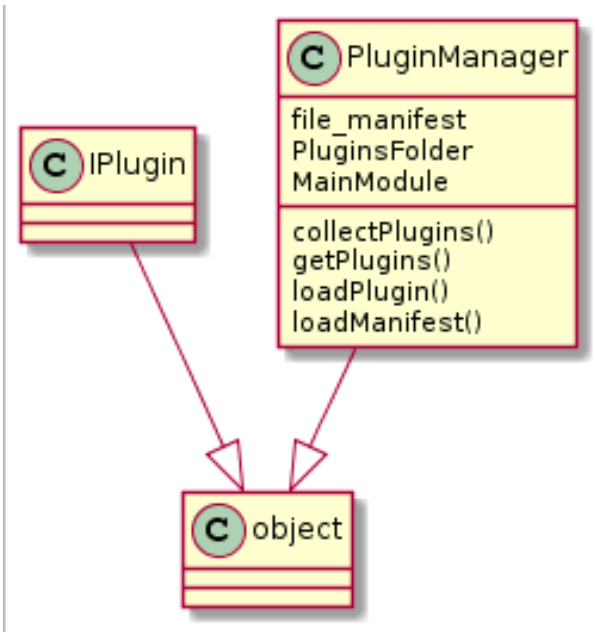
Fonte: do autor.

Figura 59 – *Statistics* - Diagrama de Classe



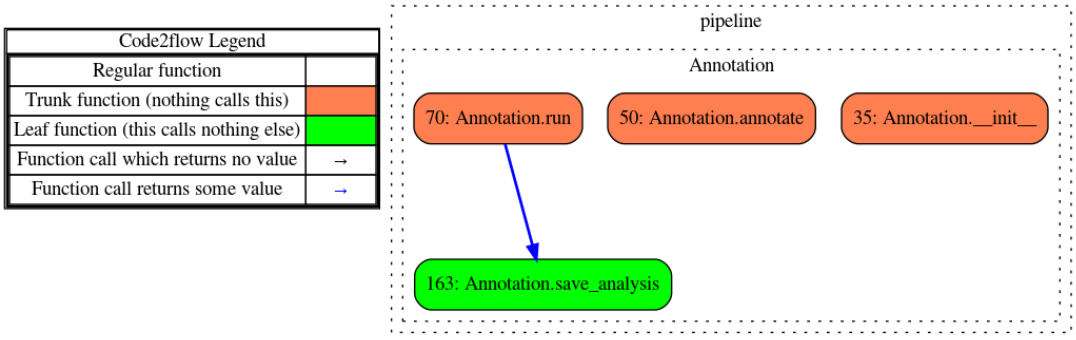
Fonte: do autor.

Figura 60 – *PluginManager* - Diagrama de Classe



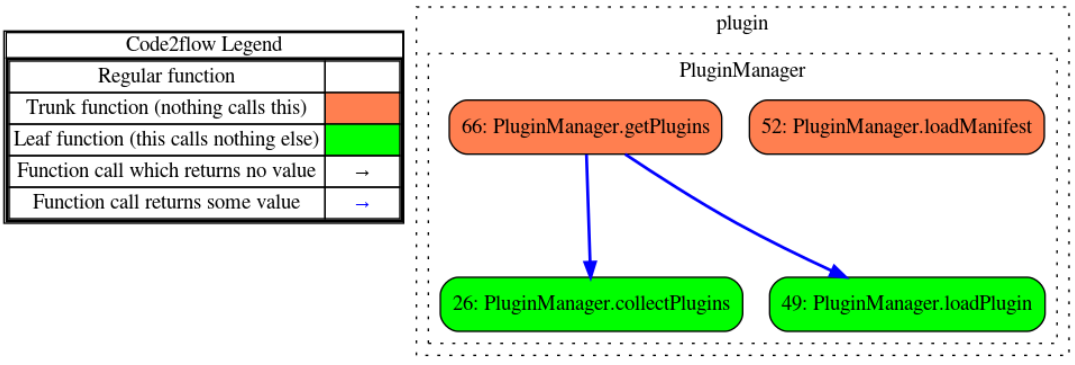
Fonte: do autor.

Figura 61 – Fluxograma do Componente *Pipeline*



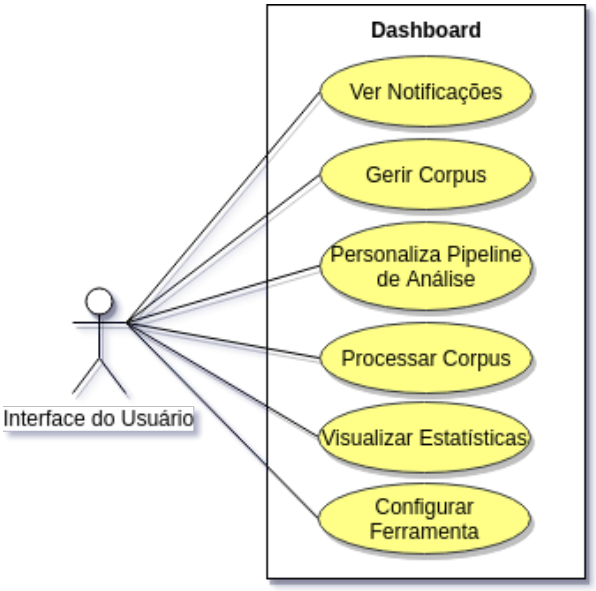
Fonte: do autor.

Figura 62 – Fluxograma do Componente *Plugin*



Fonte: do autor.

Figura 63 – *DashBoard* - Diagrama de Caso de Uso.

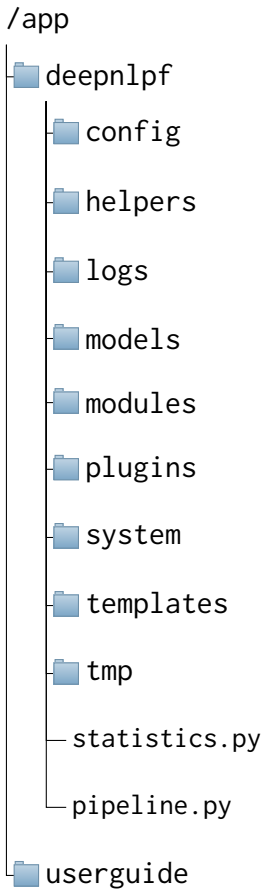


Fonte: do autor.

APÊNDICE C – ESTRUTURA DE DIRETÓRIOS E ARQUIVOS

O esquema da Figura 64, mostra a estrutura de diretórios e arquivos do *DeepNLPF*, que foi formulada com base nos sistemas de PLN citados no Capitulo 2 e 3, e *frameworks* como [1]*Flask* e [2]*Codeigniter*.

Figura 64 – Estrutura de Diretórios e Arquivos do *DeepNLPF*.



Fonte: do autor.

App é a pasta principal da aplicação, na qual está contida todo o projeto. Dentro dela estão as pastas:

- ***Config*** - Essa pasta contém vários arquivos de configuração, dentre eles, o arquivo *config.py*, para definir caminhos de ferramentas externas, entre outros recursos. No arquivo *database.py*, configuram-se informações do banco de dados (usuário, senha, etc). Por fim, no arquivos *notification.py* estão as configurações das mensagens de notificações do *DeepNLPF*.

[1] http://flask.pocoo.org/
[2] https://www.codeigniter.com/

- ***Helpers*** - Nessa pasta, o usuário pode colocar classes auxiliares para seu projeto, por exemplo, classes ([3]*DAO*, [4] CRUD, entre outras).
- ***Logs*** - Essa pasta contém arquivos relacionados aos *logs* do sistema.
- ***Models*** - Nessa pasta são colocadas todas as classes de acesso ao armazenamento de dados.
- ***Modules*** - Nessa pasta o usuário pode colocar outros módulos extras que queira utilizar em sua aplicação.
- ***Plugins*** - Essa pasta contém arquivos dos *wrappers*, desenvolvidos para a aplicação do usuário consumir ferramentas de PLN externas.
- **System** - Essa pasta contém os recurso peculiares do *DeepNLPF*, classes, *interfaces*, motores, entre outros.
- ***Templates*** - Nessa pasta ficam os templetes de modelos de arquivos XML e JSON para integração das análises linguísticas processadas. O usuário pode inserir nela outros formatos de saída que desejar utilizar em sua aplicação, por exemplo [5]CoNLL, [6]FOLIA, dentre outros.
- ***Tmp*** - Nessa pasta, ficam os arquivos temporários utilizados pelas ferramentas externas, quando necessário utilizar arquivos de texto.
- ***Statistics*** - Esse arquivo é responsável por gerar estatísticas e visualização de corpus.
- ***Pipeline*** - Esse arquivo é responsável pela customização e processamento do pipeline de análise linguística.

A pasta ***userguide*** contém instruções para auxiliar o usuário no uso da ferramenta. No Apêndice C encontra-se uma figura que ilustra um estrutura em árvore da organização dos diretórios e arquivos do *framework*.

[3] DAO é um padrão para aplicações que utilizam persistência de dados, onde tem a separação das regras de negócio das regras de acesso a banco de dados, implementada com linguagens de programação orientadas a objetos (como por exemplo Java) e arquitetura MVC, onde todas as funcionalidades de bancos de dados, tais como obter conexões, mapear objetos para tipos de dados SQL ou executar comandos SQL, devem ser feitas por classes DAO.

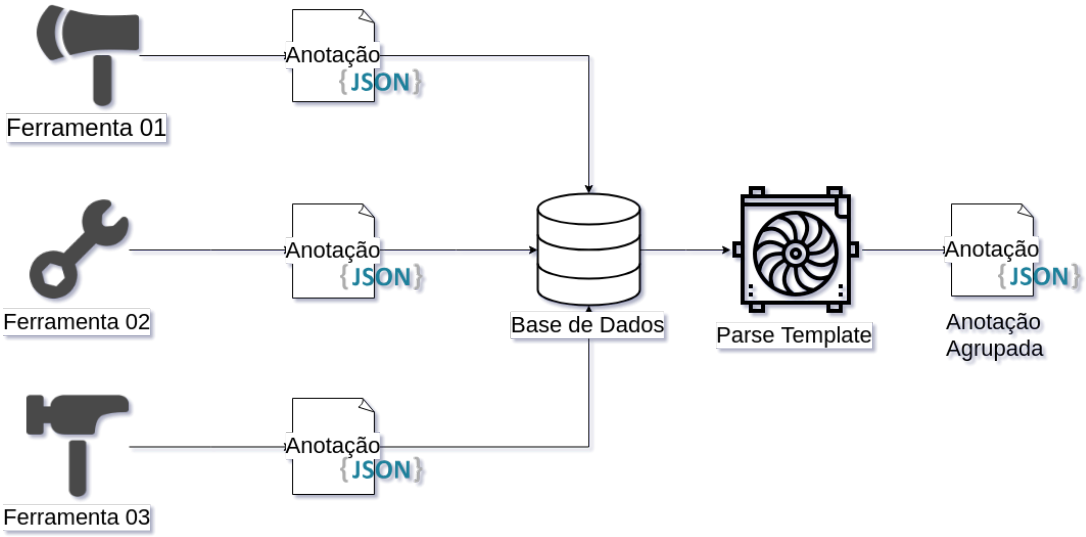
[4] *CRUD* são as quatro operações básicas (criação, consulta, atualização e destruição de dados) utilizadas em bases de dados relacionais (RDBMS) fornecidas aos utilizadores do sistema.

[5] https://universaldependencies.org/format.html

[6] https://proycon.github.io/folia/index.pt.html

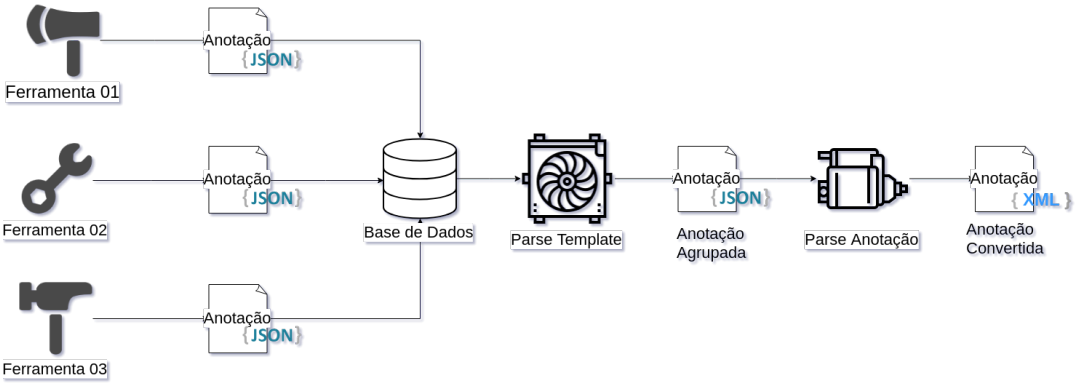
APÊNDICE D – FIGURAS

Figura 65 – Processo de Integração das Anotações Linguísticas.



Fonte: do autor.

Figura 66 – União das Anotações Linguísticas.



Fonte: do autor.

APÊNDICE E – BLOCOS DE CÓDIGOS

Algoritmo 14: Exemplo de *Multiprocess* em *Python*.

```
1
      import multiprocessing
3     import time

5     class Process(multiprocessing.Process):
          def __init__(self, id):
7             super(Process, self).__init__()
              self.id = id
9
          def run(self):
11            time.sleep(1)
              print("I'm the process with id: {}".format(self.id))
13
      if __name__ == '__main__':
15        processes = Process(1), Process(2), Process(3), Process(4)
          [p.start() for p in processes]
```

Algoritmo 15: Exemplo de *Multithreads* em *Python*.

```
# my_thread.py

import threading
import time

class MyThread(threading.Thread):
    def __init__(self, name, delay):
        threading.Thread.__init__(self)
        self.name = name
        self.delay = delay

    def run(self):
        print('Starting thread %s.' % self.name)
        thread_count_down(self.name, self.delay)
        print('Finished thread %s.' % self.name)

def thread_count_down(name, delay):
    counter = 5

    while counter:
        time.sleep(delay)
        print('Thread %s counting down: %i...' % (name, counter))
        counter -= 1

# -----------------------------------------------

# example1.py

from my_thread import MyThread

thread1 = MyThread('A', 0.5)
thread2 = MyThread('B', 0.5)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print('Finished.')
```

Algoritmo 16: Exemplo de Anotação Linguística do DeepNLPF, arquivo JSON.

```
    {
       "document": {
          "sentences": {
             "sentence": {
                "_s_id": "s_1",
                "_text": "The period of tumor shrinkage after radiation therapy is
                    often long and varied mean months .",
                "_tokens": "16",
                "_sentiment": "Negative",
                "lexical_analysis": {},
                "syntactic_analysis": {},
                "semantics_analysis": {}
             },
             ...
          },
          "_filename": "annotation.json",
          "_sentences": "980"
       }
    }
```

Algoritmo 17: Exemplo de Anotação Linguística do DeepNLPF, arquivo XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="DeepNLP-to-HTML.xsl" type="text/xsl" datetime="2018-11-16 23:36:42.555549"?>
<document filename="annotation.xml" sentences="980">
    <sentences>
        <sentence s_id="s_1" text="The period of tumor shrinkage after radiation therapy is often long and varied mean months ." tokens="16"
            sentiment="Negative">
            <lexical_analysis> ... </lexical_analysis>
            <syntactic_analysis> ... </syntactic_analysis>
            <semantics_analysis> ... </semantics_analysis>
        </sentence>
        ...
    </sentences>
</document>
```

Algoritmo 18: Exemplo de Plugin - Stanford CoreNLP.

```
from deepnlpf.system.iplugin import IPlugin

class WrapperStanfordCoreNLP(IPlugin):

    def __init__(self, corpus, tasks):
        self._corpus = corpus
        self._tasks = tasks

        from deepnlpf.config import config
        from stanfordcorenlp import StanfordCoreNLP
        self.nlp = StanfordCoreNLP(config.PATH['path_dir_stanford_corenlp'])

    def run(self):
        from deepnlpf.system.boost import Boost
        annotation = Boost().multithreading(self.wrapper, self._corpus['sentences'])
        self.nlp.close()
        return self.out_format(annotation)

    def wrapper(self, sentence):

        props = {
            'timeout': '1500000',
            'annotators': ', '.join(self._tasks),
            'pipelineLanguage': 'en',
            'outputFormat': 'json'
        }

        import json
        return json.loads(self.nlp.annotate(sentence, properties=props))

    def out_format(self, annotation):
        from deepnlpf.templates.out_format import OutFormat
        return OutFormat().doc_annotation(self._corpus['_id'], "stanfordcorenlp", annotation)
```

Algoritmo 19: Gerenciamento de *Plugins*

```
    # essa funcao cria uma colecao de plugins.
    def collectPlugins(self):
        plugins = []
        possibleplugins = os.listdir(self.PluginsFolder)

        for plugin_name in possibleplugins:
            location = os.path.join(self.PluginsFolder, plugin_name)
            if not os.path.isdir(location) or not self.MainModule + ".py" in os.
                listdir(location):
                continue
            manifest = imp.find_module(self.MainModule, [location])

            # check plugin is activated.
            location = os.path.join(self.PluginsFolder, plugin_name)
            if not os.path.isdir(location) or not self.file_manifest + ".json" in os
                .listdir(location):
                continue
            path = self.PluginsFolder + '/' + plugin_name + '/' + self.file_manifest
                 + ".json"
            info_manifest = TextFile().open_json(path)

            plugins.append({"name": plugin_name, "manifest": manifest, "is_activated
                ": info_manifest['is_activated']})
        return plugins

    # essa funcao carrega os plugins ativos.
    def loadPlugin(self, plugin):
        return imp.load_module(self.MainModule, *plugin["manifest"])

    # essa funcao carrega os arquivos manifest que contem informacoes dos plugins.
    def loadManifest(self):
        info_plugins = []
        possibleplugins = os.listdir(self.PluginsFolder)

        for plugin in possibleplugins:
            location = os.path.join(self.PluginsFolder, plugin)
            if not os.path.isdir(location) or not self.file_manifest + ".json" in os
                .listdir(location):
                continue
            path = self.PluginsFolder + '/' + plugin + '/' + self.file_manifest + ".
                json"
                info_plugins.append(TextFile().open_json(path))
        return info_plugins
```

Algoritmo 20: Modelo do arquivo *manifest.json*

```
{
    // obrigatorios.
    "manifest_version": "1",
    "name": "Hello World",
    "keyname": "helloworld",
    "version": "1.0.0",
    "category": "pln",
    "is_activated": true,
    // opcionais
    "site": "http://pluginbase.com.br",
    "description": "Base Plugins.",
    "author": "RodriguesFAS",
    "license": "MIT",
    // obrigatorio (no minimo uma analise)
    "analysis": [
        {
            "semantic": [{
                "name": "Part of Spesch (POS)",
                "keyname": "pos",
                "description": ""
                }]
        },
        {
            "syntatic": [{
                "name": "Task Two",
                "keyname": "tasktwo",
                "description": ""
                }]
        },
        {
            "semantic": [{
                "name": "Task Three",
                "keyname": "taskthree",
                "description": ""
                }]
        }
    ]
}
```

Algoritmo 21: Principais trechos de código do componente *Pipeline*.

```
    """
2       Essa funcao    inicializada altomaticamente quando o pipeline    executado,
            ela carrega todas as configuracoes necessarias.
    """
4   def __init__(self, corpus, custom_pipeline, save_analysis=False):
        ...
6       # Carrega todos os plugins ativos no DeepNLPF.
        self.plugin = PluginManager().getPlugins()
8
        # Verifica quantos n cleos (CPUs) tem dispon veis no computador.
10      self.pool = pp.ProcessPool(mp.cpu_count())
        ...
12
    # Essa funcao distribui as anotacoes por processo.
14  def annotate(self):
        ...
16      # Pega a lista de ferramentas de PLN e suas respectivas analises que o
            usuario selecionou para executar o pipeline customizado.
        list_tools = [','.join(tool.keys()) for tool in self._custom_pipeline['tools
            ']]
18
        # Executa o processamento paralelo das ferramentas com base na quantidade de
             nucleos disponiveis (CPUs), passando a lista de ferramentas de PLN
            escolhidas.
20      result = [_ for _ in tqdm(self.pool.map(self.run, new_list_tools), total=len
            (new_list_tools))]
        ...
22
    """
24      Essa funcao executa os plugins das ferramentas de PLN externas integradas ao
             DeepNLPF.
    """
26  def run(self, _tool):
        ...
28      #verifica se a ferramenta foi selecionada.
        if tool == "stanfordcorenlp":
30          # executa a ferramenta passando os dados a serem processados com suas
                respectivas analises selecionadas.
            pipeline = self._custom_pipeline['tools'][int(index)][tool]['pipeline']
32          annotation = self.plugin.WrapperStanfordCoreNLP(corpus, pipeline).run()

34          # salva a anotacao ou retorna para o usuario.
            return self.save_analysis(tool, annotation)
36      ...
```

APÊNDICE F – DASHBOARD

Para utilizar o *DeepNLPF*, é necessário iniciar a execução da *API RESTFul*. Ele será executado por padrão no endereço http://127.0.0.1:5000. Tanto no terminal quanto no *browser* será notificado que a *API RESTFul* está em execução, conforme ilustra a Figura 67. Após iniciada a execução da *API RESTFul*, deve-se executar a aplicação cliente que será aberta no *browser* no endereço http://127.0.0.1:5001, conforme apresenta a imagem 1 da Figura 68.

Figura 67 – *DeepNLPF - API RESTFul Online.*

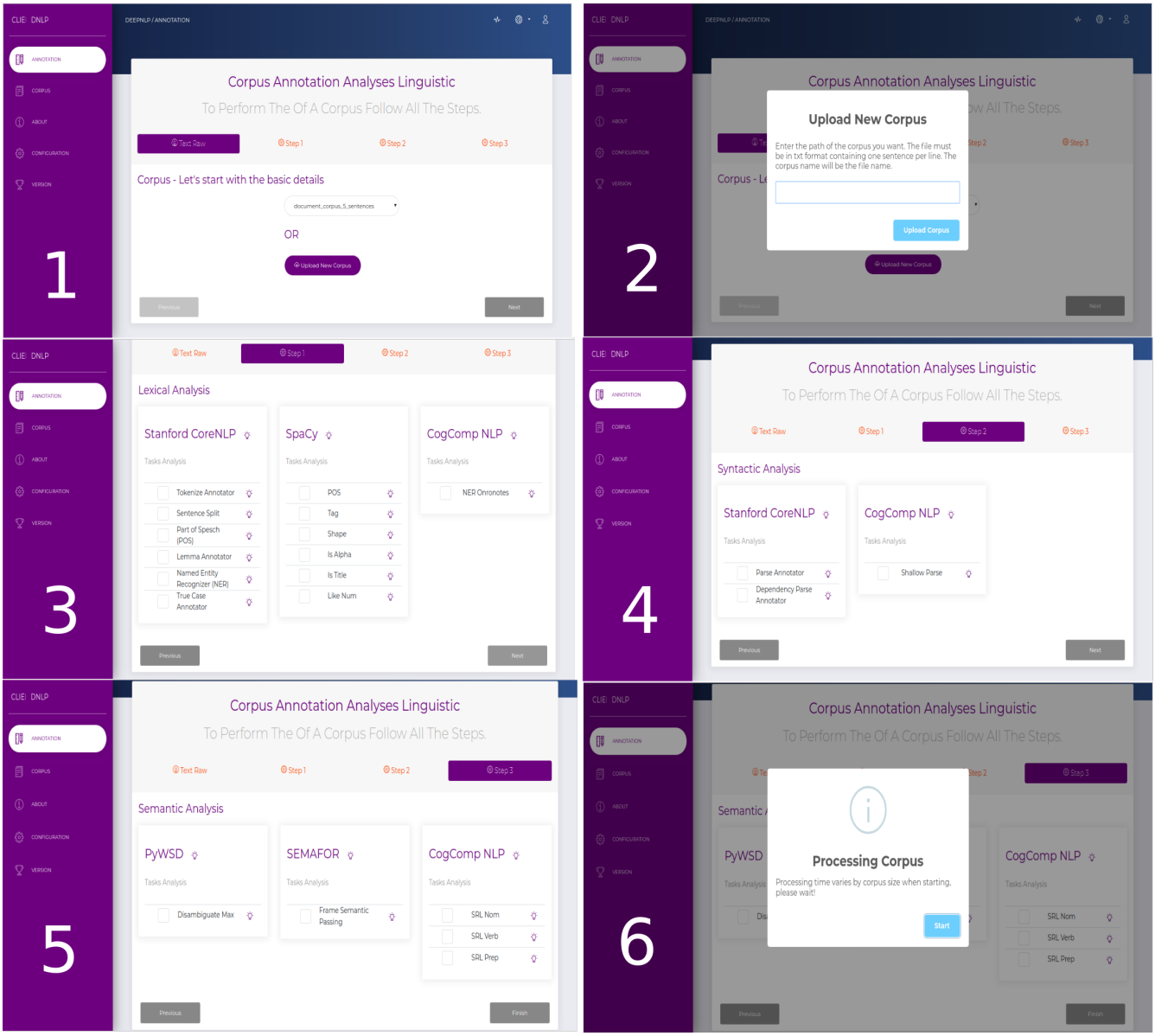
Fonte: do autor.

Ainda na imagem 1 da Figura 68, tem-se o menu lateral esquerdo com as opções (*Annotation*, *Corpus*, *Configurations* e *About*). Na imagem 2, tem-se as opções salvar *corpus* e selecionar *corpus*. Nas imagens 3, 4 e 5 respectivamente, ilustras as opções das ferramentas de PLN e suas análises agrupadas em três camadas (níveis) da linguagem (Step1 = Léxico, Step2 = Sintático e Step3 = Semântico), que podem ser selecionadas, customizando o *pipeline* de análises. Na imagem 6, ilustra a notificação que informa o estado do processamento do *corpus*, utilizando as opções (ferramentas e análises) escolhidas.

Na Figura 69 tem-se a opção do menu *Corpus*, onde a imagem 7 apresenta uma lista contendo todos os *corpus* salvos na base de dados. Cada item da lista possui um botão de ações que podem ser escolhidas pelo usuário (visualizar *corpus*, estatísticas do *corpus*, anotações do *corpus*, baixar anotações e excluir *corpus*). A imagem 8 ilustra a opção visualizar *corpus*, enquanto que as imagens 9, 10, 11 e 12, contem as estatística dos dados textuais processados, histogramas, nuvens de palavras e outras estatísticas textuais. Por fim, nas imagens 13 e 14 são exibidas as anotações do *corpus* individualmente por ferramentas de PLN.

Na Figura 70, exibe-se as telas *Configurations* e *About*, onde respectivamente encontrão-se na imagem 15 funcionalidades como notificações do estado do processamento dos dados, erros, etc, por meio da plataforma *Telegram*, *Email* ou na área de trabalho do computador do usuário. Na imagem 16, tem-se a tela *About* onde informa o que é o *DeepNLPF*, seu propósito, ajuda e muito mais. A Figura 71, exemplifica como as notificações são exibidas no *chat* do *Telegram*. Outros detalhes dessas e outras funcionalidades são encontradas no [1]manual do usuário.

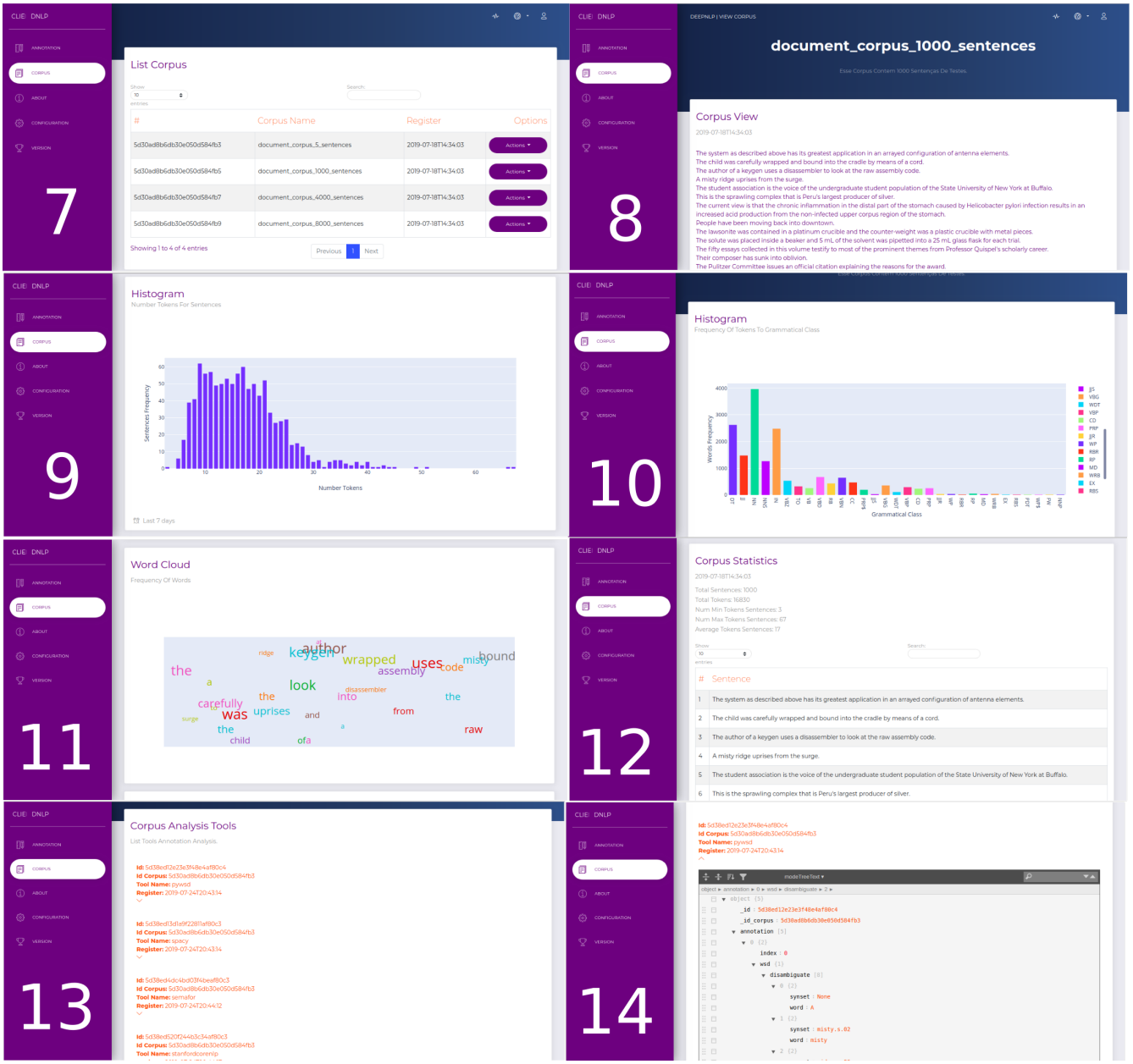
Figura 68 – *DeepNLPF DashBoard Menu Annotation* .



Fonte: do autor.

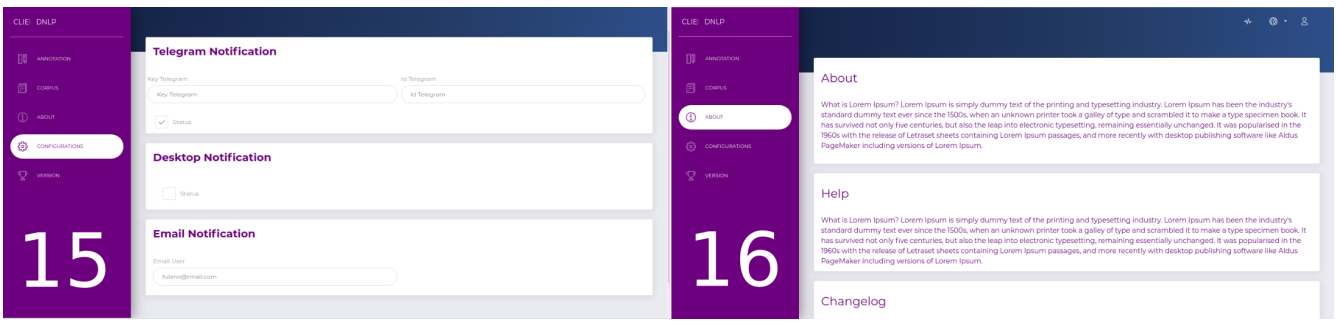
[1] https://deepnlpf.github.io/

Figura 69 – *DeepNLPF DashBoard Menu Corpus.*



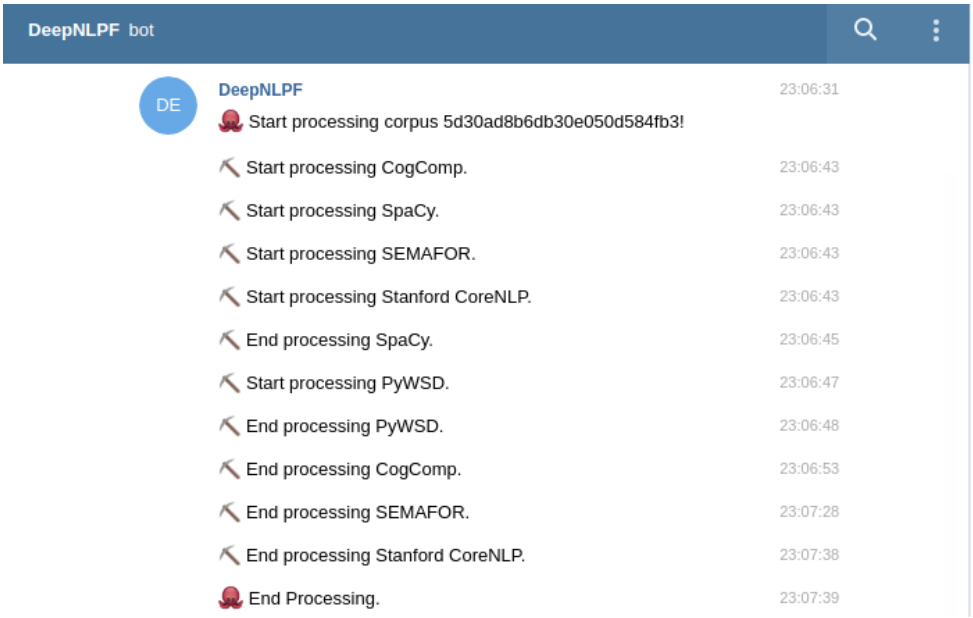
Fonte: do autor.

Figura 70 – *DeepNLPF DashBoard Menu Configuration and About.*



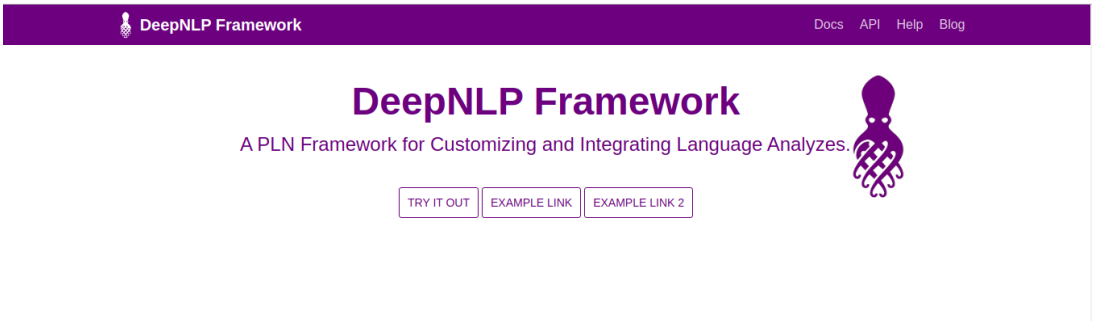
Fonte: do autor.

Figura 71 – *DeepNLPF* - Notificações no *Telegram*.



Fonte: do autor.

Figura 72 – *User Guide DeepNLPF*.



Fonte: do autor.